

Table of Contents

Joint API	4
anchors	4
connectionPoints	14
connectionStrategies	20
connectors	24
dia	33
elementTools	223
env	231
highlighters	232
layout	240
linkAnchors	251
linkTools	254
mvc	269
routers	272
shapes	279
util	292
Geometry API	315
Vectorizer API	379
JointJS+	396
JointJS+ Dijkstra	399
JointJS+ PriorityQueue	401
JointJS+ CommandManager	404
JointJS+ Validator	409
JointJS+ RecordScrollbar	413
JointJS+ SwimlaneBoundary	414
JointJS+ SwimlaneTransform	415
JointJS+ GEXF	418
JointJS+ Print	420
JointJS+ Raster	424
JointJS+ SVG	427
JointJS+ Visio	430
JointJS+ graphUtils	463
JointJS+ constructTree	464
JointJS+ shortestPath	466

JointJS+ toAdjacencyList	467
JointJS+ ForceDirected	468
JointJS+ GridLayout	470
JointJS+ StackLayout	473
JointJS+ TreeLayout	477
JointJS+ bpmn	481
JointJS+ bpmn2	489
JointJS+ chart	534
JointJS+ measurement	558
JointJS+ standard	564
JointJS+ vsm	578
JointJS+ Local	595
JointJS+ ui	597
JointJS+ Clipboard	598
JointJS+ ColorPalette	601
JointJS+ ContextToolbar	605
JointJS+ Dialog	608
JointJS+ FlashMessage	611
JointJS+ FreeTransform	614
JointJS+ Halo	617
JointJS+ Inspector	625
JointJS+ Keyboard	654
JointJS+ Lightbox	656
JointJS+ Navigator	659
JointJS+ PaperScroller	662
JointJS+ PathDrawer	675
JointJS+ PathEditor	680
JointJS+ Popup	689
JointJS+ RadioGroup	693
JointJS+ SelectBox	695
JointJS+ SelectButtonGroup	699
JointJS+ Selection	703
JointJS+ Snaplines	711
JointJS+ StackLayoutView	713
JointJS+ Stencil	719
JointJS+ TextEditor	732

JointJS+ Toolbar	746
JointJS+ Tooltip	753
JointJS+ TreeLayoutView	758

Joint API

This is the API reference to the open source JointJS core library. If you're looking for the JointJS+ diagramming toolkit documentation, you can find that [here](#).

JointJS library exports three global variables: `joint`, `V` and `g`.

The `joint` namespace contains all the objects that you will use to build your diagrams. Additionally, `joint.version` property tells you which version of JointJS you're using.

The `V` global is lightweight SVG library that we call “Vectorizer”. This tiny library makes manipulation with SVG documents much easier. JointJS uses this library internally. Normally, you don't have to get in touch with this library at all but for advanced uses, it can be handy.

The `g` global is another lightweight library used internally by JointJS that provides many useful geometry operations. Again, you might not get in touch with this library but when you do have the need to perform geometric operations in your applications, you'll certainly find it helpful.

anchors

An anchor of a link is a point in the reference element that this link wants to reach as its endpoint. (In reality, the reference element is probably in the way - then, it is the job of the [connection point function](#) to determine the actual location of the route endpoint with the obstructing reference element taken into account.) Anchors are set via an `anchor` property provided within link end definitions (i.e. the objects provided to `link.source()` and `link.target()` functions). (If the reference object is a Link, JointJS looks at `linkAnchor` [property](#) instead.)

There are many built-in anchor functions in JointJS:

- `'center'` - [default anchor at center of view bbox](#)
- `'modelCenter'` - [anchor at center of model bbox](#)
- `'perpendicular'` - [anchor that ensures an orthogonal route to the other endpoint](#)
- `'midSide'` - [anchor in the middle of the side of view bbox closest to the other endpoint](#)
- `'bottom'` - [anchor in the middle of the bottom side of view bbox](#)
- `'left'` - [anchor in the middle of the left side of view bbox](#)
- `'right'` - [anchor in the middle of the right side of view bbox](#)
- `'top'` - [anchor in the middle of the top side of view bbox](#)
- `'bottomLeft'` - [anchor at the bottom-left corner of view bbox](#)
- `'bottomRight'` - [anchor at the bottom-right corner of view bbox](#)
- `'topLeft'` - [anchor at the top-left corner of view bbox](#)

- `'topRight'` - anchor at the top-right corner of view bbox

Example:

```
link.source(model, {
  anchor: {
    name: 'midSide',
    args: {
      rotate: true,
      padding: 20
    }
  }
});
```

The default anchor function is `'center'`; this can be changed with the `defaultAnchor` paper option. Example:

```
paper.options.defaultAnchor = {
  name: 'midSide',
  args: {
    rotate: true,
    padding: 20
  }
};
```

JointJS also contains mechanisms to define one's own custom anchor functions.

anchors.bottom

The `'bottom'` anchor function places the anchor of the link in the middle of the bottom side of the view bbox. It accepts three arguments, which can be passed within the `anchor.args` property:

rotate	<i>boolean</i>	Should the anchor bbox rotate with the end view? Default is <code>false</code> , meaning that the unrotated bbox is used.
dx	<i>number</i>	Offset the anchor by <code>dx</code> . Default is <code>0</code> .
	<i>string</i>	Percentage strings (e.g. <code>'40%'</code>) are also accepted.
dy	<i>number</i>	Offset the anchor by <code>dy</code> . Default is <code>0</code> .
	<i>string</i>	Percentage strings (e.g. <code>'40%'</code>) are also accepted.

Example:

```
link.source(model, {
  anchor: {
    name: 'bottom',
    args: {
      rotate: true,
      dx: 10,
      dy: '40%'
    }
  }
});
```

anchors.bottomLeft

The `'bottomLeft'` anchor function places the anchor of the link at the bottom-left corner of the view bbox. It accepts three arguments, which can be passed within the `anchor.args` property:

rotate	<i>boolean</i>	Should the anchor bbox rotate with the end view? Default is <code>false</code> , meaning that the unrotated bbox is used.
dx	<i>number</i>	Offset the anchor by <code>dx</code> . Default is <code>0</code> .
	<i>string</i>	Percentage strings (e.g. <code>'40%'</code>) are also accepted.
dy	<i>number</i>	Offset the anchor by <code>dy</code> . Default is <code>0</code> .
	<i>string</i>	Percentage strings (e.g. <code>'40%'</code>) are also accepted.

Example:

```
link.source(model, {
  anchor: {
    name: 'bottomLeft',
    args: {
      rotate: true,
      dx: 10,
      dy: '40%'
    }
  }
});
```

anchors.bottomRight

The `'bottomRight'` anchor function places the anchor of the link at the bottom-left corner of the view bbox. It accepts three arguments, which can be passed within the `anchor.args` property:

rotate	<i>boolean</i>	Should the anchor bbox rotate with the end view? Default is <code>false</code> , meaning that the unrotated bbox is used.
dx	<i>number</i>	Offset the anchor by <code>dx</code> . Default is <code>0</code> .
	<i>string</i>	Percentage strings (e.g. <code>'40%'</code>) are also accepted.
dy	<i>number</i>	Offset the anchor by <code>dy</code> . Default is <code>0</code> .
	<i>string</i>	Percentage strings (e.g. <code>'40%'</code>) are also accepted.

Example:

```
link.source(model, {
  anchor: {
    name: 'bottomRight',
    args: {
      rotate: true,
      dx: 10,
      dy: '40%'
    }
  }
});
```

anchors.center

The `'center'` anchor function is the default anchor function. It places the anchor of the link at center of the view bbox. It accepts three arguments, which can be passed within the `anchor.args` property:

rotate	<i>boolean</i>	Should the anchor bbox rotate with the end view? Default is <code>false</code> , meaning that the unrotated bbox is used.
dx	<i>number</i>	Offset the anchor by <code>dx</code> . Default is <code>0</code> .

	<i>string</i>	Percentage strings (e.g. <code>'40%'</code>) are also accepted.
dy	<i>number</i>	Offset the anchor by <code>dy</code> . Default is <code>0</code> .
	<i>string</i>	Percentage strings (e.g. <code>'40%'</code>) are also accepted.

Example:

```
link.source(model, {
  anchor: {
    name: 'center',
    args: {
      rotate: true,
      dx: 10,
      dy: '40%'
    }
  }
});
```

anchors.custom

New anchor functions can be defined in the `joint.anchors` namespace (e.g. `joint.anchors.myAnchor`) or passed directly as a function to the `anchor` property of link source/target (or to the `defaultAnchor` [option of a paper](#)).

In either case, the anchor function must return the anchor as a `Point`. The function is expected to have the form `function(endView, endMagnet, anchorReference, args) :`

endView	<i>dia.ElementView</i>	The <code>ElementView</code> to which we are connecting. The <code>Element</code> model can be accessed as <code>endView.model</code> ; this may be useful for writing conditional logic based on element attributes.
endMagnet	<i>SVGElement</i>	The <code>SVGElement</code> in our page that contains the magnet (element/subelement/port) to which we are connecting.
anchorReference	<i>g.Point</i>	A reference to another component of the link path that may be necessary to find this anchor point. If we are calling this method for a source anchor, it is the first vertex, or if there are no vertices the target anchor. If we are calling this method for a target anchor, it is the last vertex, or if there are no vertices the source anchor...
	<i>SVGElement</i>	...if the anchor in question does not exist (yet), it is that link end's magnet. (The built-in methods usually use this element's center point as

		reference.)
args	<i>object</i>	An object with additional optional arguments passed to the anchor method by the user when it was called (the <code>args</code> property).

anchors.left

The `'left'` anchor function places the anchor of the link in the middle of the left side of the view bbox. It accepts three arguments, which can be passed within the `anchor.args` property:

rotate	<i>boolean</i>	Should the anchor bbox rotate with the end view? Default is <code>false</code> , meaning that the unrotated bbox is used.
dx	<i>number</i>	Offset the anchor by <code>dx</code> . Default is <code>0</code> .
	<i>string</i>	Percentage strings (e.g. <code>'40%'</code>) are also accepted.
dy	<i>number</i>	Offset the anchor by <code>dy</code> . Default is <code>0</code> .
	<i>string</i>	Percentage strings (e.g. <code>'40%'</code>) are also accepted.

Example:

```
link.source(model, {
  anchor: {
    name: 'left',
    args: {
      rotate: true,
      dx: 10,
      dy: '40%'
    }
  }
});
```

anchors.midSide

The `'midSide'` anchor function places the anchor of the link in the middle of the side of view bbox closest to the other endpoint. It accepts two arguments, which can be passed within the `anchor.args` property:

rotate	<i>boolean</i>	Should the anchor bbox rotate with the end view? Default is <code>false</code> , meaning that the unrotated bbox is used.
padding	<i>number</i>	Offset the anchor by <code>padding</code> away from view bbox. Default is <code>0</code> .

Example:

```
link.source(model, {
  anchor: {
    name: 'midSide',
    args: {
      rotate: true,
      padding: 20
    }
  }
});
```

anchors.modelCenter

The `'modelCenter'` anchor function places the anchor of the link at center of the model bbox.

Example:

```
link.source(model, {
  anchor: {
    name: 'modelCenter'
  }
});
```

anchors.perpendicular

The `'perpendicular'` anchor function tries to place the anchor of the link inside the view bbox so that the link is made orthogonal. The anchor is placed along two line segments inside the view bbox (between the centers of the top and bottom side and between the centers of the left and right sides). If it is not possible to place the anchor so that the link would be orthogonal, the anchor is placed at the center of the view bbox instead. The function accepts one argument, which can be passed within the `anchor.args` property:

padding	<i>number</i>	Limit the area inside the view bbox available for placing the anchor by <code>padding</code> . Default is <code>0</code> .

Example:

```
link.source(model, {
  anchor: {
    name: 'perpendicular',
    args: {
      padding: 10
    }
  }
});
```

When the link has no vertices, the other end cell's center is used as a reference point. By default, this means that a link using the `'perpendicular'` anchor “slides” alongside the source element's edge while pointing to target element's center. To invert this behavior, and have the anchor “slide” alongside the target element's edge while pointing to source element's center, pass a `priority` option to the target function:

```
link.target(model, {
  priority: true,
  anchor: {
    name: 'perpendicular',
    args: {
      padding: 10
    }
  }
});
```

anchors.right

The `'right'` anchor function places the anchor of the link in the middle of the right side of the view bbox. It accepts three arguments, which can be passed within the `anchor.args` property:

rotate	<i>boolean</i>	Should the anchor bbox rotate with the end view? Default is <code>false</code> , meaning that the unrotated bbox is used.
dx	<i>number</i>	Offset the anchor by <code>dx</code> . Default is <code>0</code> .
	<i>string</i>	Percentage strings (e.g. <code>'40%'</code>) are also accepted.
dy	<i>number</i>	Offset the anchor by <code>dy</code> . Default is <code>0</code> .
	<i>string</i>	Percentage strings (e.g. <code>'40%'</code>) are also accepted.

Example:

```
link.source(model, {
  anchor: {
    name: 'right',
    args: {
      rotate: true,
      dx: 10,
      dy: '40%'
    }
  }
});
```

anchors.top

The `'top'` anchor function places the anchor of the link in the middle of the top side of the view bbox. It accepts three arguments, which can be passed within the `anchor.args` property:

rotate	<i>boolean</i>	Should the anchor bbox rotate with the end view? Default is <code>false</code> , meaning that the unrotated bbox is used.
dx	<i>number</i>	Offset the anchor by <code>dx</code> . Default is <code>0</code> .
	<i>string</i>	Percentage strings (e.g. <code>'40%'</code>) are also accepted.
dy	<i>number</i>	Offset the anchor by <code>dy</code> . Default is <code>0</code> .
	<i>string</i>	Percentage strings (e.g. <code>'40%'</code>) are also accepted.

Example:

```
link.source(model, {
  anchor: {
    name: 'top',
    args: {
      rotate: true,
      dx: 10,
      dy: '40%'
    }
  }
});
```


anchors.topLeft

The `'topLeft'` anchor function places the anchor of the link at the top-left corner of the view bbox. It accepts three arguments, which can be passed within the `anchor.args` property:

rotate	<i>boolean</i>	Should the anchor bbox rotate with the end view? Default is <code>false</code> , meaning that the unrotated bbox is used.
dx	<i>number</i>	Offset the anchor by <code>dx</code> . Default is <code>0</code> .
	<i>string</i>	Percentage strings (e.g. <code>'40%'</code>) are also accepted.
dy	<i>number</i>	Offset the anchor by <code>dy</code> . Default is <code>0</code> .
	<i>string</i>	Percentage strings (e.g. <code>'40%'</code>) are also accepted.

Example:

```
link.source(model, {
  anchor: {
    name: 'topLeft',
    args: {
      rotate: true,
      dx: 10,
      dy: '40%'
    }
  }
});
```

anchors.topRight

The `'topRight'` anchor function places the anchor of the link at the top-right corner of the view bbox. It accepts three arguments, which can be passed within the `anchor.args` property:

rotate	<i>boolean</i>	Should the anchor bbox rotate with the end view? Default is <code>false</code> , meaning that the unrotated bbox is used.
dx	<i>number</i>	Offset the anchor by <code>dx</code> . Default is <code>0</code> .

	<i>string</i>	Percentage strings (e.g. <code>'40%'</code>) are also accepted.
dy	<i>number</i>	Offset the anchor by <code>dy</code> . Default is <code>0</code> .
	<i>string</i>	Percentage strings (e.g. <code>'40%'</code>) are also accepted.

Example:

```
link.source(model, {
  anchor: {
    name: 'topRight',
    args: {
      rotate: true,
      dx: 10,
      dy: '40%'
    }
  }
});
```

connectionPoints

A link connection point is an endpoint of the link route. This point is (usually) different from the [link anchor point](#), as it takes into account the presence of the end element. Connection points are set via an `connectionPoint` property provided within link end definitions (i.e. the objects provided to `link.source()` and `link.target()` functions).

The built-in functions work by finding an intersection between the link path (the path from the link's source anchor, through its vertices, to its target anchor). However, the functions always only have access to a single path segment; the source connectionPoint is found by investigating the first segment (i.e. source anchor - first vertex, or source anchor - target anchor if there are no vertices), while the target connectionPoint is found by investigating the last segment (i.e. last vertex - target anchor, or source anchor - target anchor). This has consequences if the investigated path segment is entirely contained within the end element.

There are four built-in connection point functions in JointJS:

- `'anchor'` - [connection point at anchor](#)
- `'bbox'` - [default connection point at bbox boundary](#)
- `'boundary'` - [connection point at actual shape boundary](#)
- `'rectangle'` - [connection point at unrotated bbox boundary](#)

Example:

```
link.source(model, {
  connectionPoint: {
    name: 'boundary',
    args: {
      sticky: true
    }
  }
});
```

The default connection point is `'bbox'` ; this can be changed with the `defaultConnectionPoint` [paper option](#).
Example:

```
paper.options.defaultConnectionPoint = {
  name: 'boundary',
  args: {
    sticky: true
  }
};
```

All four of the built-in connection point functions accept the following optional argument, in addition to their own arguments:

offset	<i>number</i>	Offset the connection point from the anchor by the specified distance along the end link path segment. Default is <code>0</code> .

Example:

```
link.source(model, {
  connectionPoint: {
    name: 'bbox',
    args: {
      offset: 10
    }
  }
});
```

JointJS also contains mechanisms to define one's own [custom connection point functions](#).

connectionPoints.anchor

The `'anchor'` places the connection point so that it coincides with the link end's anchor point (determined either by the `anchor` [function](#) or by the `defaultAnchor` [paper option](#)). The position of the connection point may be modified by several additional arguments, which may be passed within the `connectionPoint.args` property:

offset	<i>number object</i>	An object with <code>x</code> and <code>y</code> properties. The connection point will be moved: - from the anchor point by <code>x</code> pixels in the direction of the corresponding link end's path segment. - by <code>y</code> pixels in the direction of the corresponding link end's path segment, rotated by 90 degrees anti-clockwise around the anchor point. If a number is provided, it will be used as the <code>x</code> offset.
align	<i>'top' 'left' 'bottom' 'right'</i>	Offset the connection point to the point given by projecting the first vertex onto the vector which points from the anchor point in the direction specified. (If there are no vertices, use the projection of the other anchor point instead.) Notably, if the reference point is not the “direction”-most point of the two, the connection point is set to be the same as the anchor point. Let us illustrate that outcome and the other possible outcome on the <code>'top'</code> direction: - The reference point lies below the anchor point. The anchor point is the topmost point of the two, so the connection point receives both the <code>'x'</code> and <code>'y'</code> coordinates from the anchor point. The connection point is set to be coincident with the anchor point. - The reference point lies above the anchor point. In this case, the reference point is the topmost point of the two, so the connection point receives the <code>'y'</code> coordinate from the reference point. The connection point still receives the <code>'x'</code> coordinate from the anchor point. This means that the connection point is offset from the anchor point in the <code>'top'</code> direction. The three points create a right-angled triangle, with the right angle at the connection point.
alignOffset	<i>number</i>	After having determined the position of the connection point according to the <code>'align'</code> algorithm (see above), additionally offset the connection point by the specified amount in the direction specified by <code>'align'</code>.

Example:

```
link.source(model, {
  connectionPoint: {
    name: 'anchor',
    args: {
      offset: 10
    }
  }
});
```

connectionPoints.bbox

The `'bbox'` connection point function is the default connection point function. It places the connection point at the intersection between the link path end segment and the end element bbox. The position of the connection point may be modified by several additional arguments, which may be passed within the `connectionPoint.args` property:

offset	<i>number object</i>	An object with <code>x</code> and <code>y</code> properties. The connection point will be moved: • from the anchor point by <code>x</code> pixels in the direction of the corresponding link end's path segment. • by <code>y</code> pixels in the direction of the corresponding link end's path segment, rotated by 90 degrees anti-clockwise around the anchor point. If a number is provided, it will be used as the <code>x</code> offset.
stroke	<i>boolean</i>	Should the stroke width be included when calculating the connection point? Default is <code>false</code>.

Example:

```
link.source(model, {
  connectionPoint: {
    name: 'bbox',
    args: {
      offset: 10,
      stroke: true
    }
  }
});
```

connectionPoints.boundary

The `'boundary'` connection point function places the connection point at the intersection between the link path end segment and the actual shape of the end element. (If JointJS is unable to determine the actual shape - e.g. for text - the element bbox is used instead, just as in the `'bbox'` [connection point function](#).) The position of the connection point may be modified by several additional arguments, which may be passed within the `connectionPoint.args` property:

offset	<i>number object</i>	An object with <code>x</code> and <code>y</code> properties. The connection point will be moved: • from the anchor point by <code>x</code> pixels in the direction of the corresponding link end's path segment. • by <code>y</code> pixels in the direction of the corresponding link end's path segment, rotated by 90 degrees anti-clockwise around the anchor point. If a number is provided, it will be used as the <code>x</code> offset.

insideout	<i>boolean</i>	What happens if the link path never leaves the interior area of the end element (e.g. because the other end anchor lies within the first end element)? Should the path line be extended until an intersection with the boundary is found? Default is true .
extrapolate	<i>boolean</i>	What happens if the link path never enters the interior area of the end element (e.g. because the anchor lies outside the end element)? Should the path line be extended to try and find the boundary? Default is false . Note that even if this option is true , an intersection is still not guaranteed. This option takes precedence over <code>connectionPoint.args.sticky</code> .
sticky	<i>boolean</i>	What happens if the link path never enters the interior area of the end element (e.g. because the anchor lies outside the end element)? Should the closest point on the end element boundary be used instead? Default is false . Note that setting this option to true guarantees that a connection point will be found on the shape boundary.
precision	<i>number</i>	The precision of the path intersection algorithm. Uses a logarithmic scale; increasing the number by 1 reduces the maximum observed error by a factor of ten. Default is 2 , corresponding to 1% error.
selector	<i>string</i>	A selector to identify subelement/magnet of the end element at whose boundary we want the connection point to be found. Default is undefined , meaning that the first non-group descendant of the end element's node will be considered. (An example of another setting that may be useful is 'root' , which forces the usage of the root group bbox instead.). If set to false , the magnet is used as is, even if it is an SVGGroup (it's the most suitable for use in conjunction with <u>magnetSelector</u>).
stroke	<i>boolean</i>	Should the stroke width be included when calculating the connection point? Default is false .

Example:

```
link.source(model, {
  connectionPoint: {
    name: 'boundary',
    args: {
      offset: 10,
      insideout: false,
      extrapolate: true,
      sticky: true,
      precision: 3,
      stroke: true
    }
  }
});
```

connectionPoints.custom

New connection point function can be defined in the `joint.connectionPoints` namespace (e.g. `joint.connectionPoints.myConnectionPoint`) or passed directly as a function to the `connectionPoint` property of link source/target (or to the `defaultConnectionPoint` [option of a paper](#)).

In either case, the connection point function must return the connection point as a `Point`. The function is expected to have the form `function(endPathSegmentLine, endView, endMagnet, args)`:

endPathSegmentLine	<i>g.Line</i>	The link path segment at which we are finding the connection point. If we are calling this method for a source connection point, it is the first segment (source anchor - first vertex, or source anchor - target anchor). If we are calling this method for a target connection point, it is the last segment (last vertex - target anchor, or source anchor - target anchor).
endView	<i>dia.ElementView</i>	The <code>ElementView</code> to which we are connecting. The <code>Element</code> model can be accessed as <code>endView.model</code> ; this may be useful for writing conditional logic based on element attributes.
endMagnet	<i>SVGElement</i>	The <code>SVGElement</code> in our page that contains the magnet (element/subelement/port) to which we are connecting.
args	<i>object</i>	An object with additional optional arguments passed to the connection point method by the user when it was called (the <code>args</code> property).

connectionPoints.rectangle

The `'rectangle'` connection point function places the connection point at the intersection between the link path end segment and the element's unrotated bbox. The position of the connection point may be modified by several additional arguments, which may be passed within the `connectionPoint.args` property:

offset	<i>number object</i>	An object with <code>x</code> and <code>y</code> properties. The connection point will be moved: - from the anchor point by <code>x</code> pixels in the direction of the corresponding link end's path segment. - by <code>y</code> pixels in the direction of the corresponding link end's path segment, rotated by 90 degrees anti-clockwise around the anchor point. If a number is provided, it will be used as the <code>x</code> offset.

stroke	<i>boolean</i>	Should the stroke width be included when calculating the connection point? Default is false .

Example:

```
link.source(model, {
  connectionPoint: {
    name: 'rectangle',
    args: {
      offset: 10,
      stroke: true
    }
  }
});
```

connectionStrategies

Connection strategies come into play when the user modifies the position of link endpoints. There are two situations in which this is relevant:

- When the user drags a link endpoint and connects it to an element or its port. The connection strategy determines the end anchor after the user is finished dragging the link endpoint. (Note that any individual `anchor` property that might have been assigned on the dragged link endpoint will be overridden by the connection strategy. If necessary, have a look at the [custom connectionStrategy documentation](#) for information on replicating the functionality of [anchor](#) functions.)
- When a user creates a link, for example by clicking a port. The connection strategy determines the new link's source anchor.

Both the `anchor` and `connectionPoint` properties are rewritten in response to user interaction. None of the built-in connection strategies preserve the originally assigned anchor and connection point functions. To assign precisely what you need as the anchor and connection point functions, you may need to define your own [custom connection strategy](#).

Connection strategies are not assigned on a link-by-link basis. They are set with the `connectionStrategy` [option on a paper](#).

There are three built-in connection strategies in JointJS:

- `useDefaults` - [default strategy; ignore user pointer and use default anchor and connectionPoint functions](#)
- `pinAbsolute` - [use user pointer coordinates to position anchor absolutely within end element; use default connectionPoint](#)
- `pinRelative` - [use user pointer coordinates to position anchor relatively within end element; use default connectionPoint](#)

The default connection strategy is specified as `null` in [paper settings](#), which is equivalent to `joint.connectionStrategies.useDefaults`.

Built-in connection strategies are specified by reference to their name in the `joint.connectionStrategies` namespace. Example:

```
paper.options.connectionStrategy = joint.connectionStrategies.pinAbsolute;
```

connectionStrategies.custom

New connection strategies can be defined in the `joint.connectionStrategies` namespace (e.g. `joint.connectionStrategies.myConnectionStrategy`) or passed directly as a function to the `connectionStrategy` [option of a paper](#).

In either case, the connection strategy function must return an end definition (i.e. an object in the format supplied to the `link.source()` and `link.target()` functions). The function is expected to have the form `function(endDefinition, endView, endMagnet, coords):`

endDefinition	<i>object</i>	An end definition; the output of the appropriate end function (<code>link.source()</code> or <code>link.target()</code>). An object containing at least an <code>id</code> of the Element to which we are connecting. This object is expected to be changed by this function and then sent as the return value.
endView	<i>dia.ElementView</i>	The ElementView to which we are connecting. The Element model can be accessed as <code>endView.model</code> ; this may be useful for writing conditional logic based on element attributes.
endMagnet	<i>SVGElement</i>	The SVGElement in our page that contains the magnet (element/subelement/port) to which we are connecting.
coords	<i>g.Point</i>	A Point object recording the x-y coordinates of the user pointer when the connection strategy was invoked.

Custom connection strategies may be enormously useful for your users. Here we provide some examples of custom functionality.

Connecting to Ancestors

If your diagram makes heavy use of nested elements, it may be useful to always connect links to a top-level ancestor element (instead of the element on which the arrowhead was actually dropped by user interaction):

```
joint.connectionStrategies.topAncestor = function(end, endView) {  
  
    var ancestors = endView.model.getAncestors();
```

```

    var numAncestors = ancestors.length;
    var end = numAncestors ? ancestors[numAncestors - 1] : end;

    return end;
}

paper.options.connectionStrategy = joint.connectionStrategies.topAncestor;

```

Connecting to Ports

If your diagram uses ports, you usually do not want links to be able to connect anywhere else. The solution is similar to the one above:

```

joint.connectionStrategies.firstPort = function(end, endView) {

    var ports = endView.model.getPorts();
    var numPorts = ports.length;
    var end = numPorts ? { id: end.id, port: ports[0].id } : end;

    return end;
}

paper.options.connectionStrategy = joint.connectionStrategies.firstPort;

```

Replicating Built-in Anchor Functions

Furthermore, it is very easy to replicate the built-in [anchor functions](#) for connection strategy scenarios - simply apply the anchor function to the received `end` parameter:

```

joint.connectionStrategy.midSide = function(end) {

    end.anchor = {
        name: 'midSide',
        args: {
            rotate: true
        }
    };

    return end;
}

paper.options.connectionStrategy = joint.connectionStrategy.midSide;

```

Replicating Built-in Connection Point Functions

What if we needed to replicate a built-in connection point function instead? We use the same idea as in the previous example:

```
joint.connectionStrategy.boundary = function(end) {  
  
  end.connectionPoint = {  
    name: 'boundary',  
    args: {  
      offset: 5  
    }  
  };  
  
  return end;  
}  
  
paper.options.connectionStrategy = joint.connectionStrategy.boundary;
```

Of course, it is also possible to combine both of the examples and assign an `anchor` as well as `connectionPoint` to the `end` parameter of the modified link.

connectionStrategies.pinAbsolute

The `pinAbsolute` connection strategy records the coordinates of user pointer and assigns the end anchor absolutely, by reference to the top-left corner of the view `bbox` of the element above which the endpoint was dropped. Absolute positioning ensures that if the element is subsequently resized, the anchor stays at the same absolute distance from the edges (e.g. staying 10 pixels away from the left side and 20 pixels away from the top side).

The end connection point is assigned according to `defaultConnectionPoint` paper option.

Example:

```
paper.options.connectionStrategy = joint.connectionStrategies.pinAbsolute;
```

The end (source or target) that is being modified gets the `'topLeft'` anchor assigned by this connection strategy:

```
end.anchor = {  
  name: 'topLeft',  
  args: {  
    rotate: true  
    dx: (coords.x - bbox.x),  
    dy: (coords.y - bbox.y)  
  }  
};
```

connectionStrategies.pinRelative

The `pinRelative` connection strategy records the coordinates of user pointer and assigns the end anchor relatively, by reference to the top-left corner of the view bbox of the element above which the endpoint was dropped. Relative positioning ensures that if the element is subsequently resized, the anchor stays at the same relative distance from the edges (e.g. staying 25% of the way from the left side and 75% of the way from the top side).

The end connection point is assigned according to `defaultConnectionPoint` [paper option](#).

Example:

```
paper.options.connectionStrategy = joint.connectionStrategies.pinRelative;
```

The end (source or target) that is being modified gets the `'topLeft'` anchor assigned by this connection strategy:

```
end.anchor = {
  name: 'topLeft',
  args: {
    rotate: true
    dx: percentageString(coords.x - bbox.x),
    dy: percentageString(coords.y - bbox.y)
  }
};
```

connectionStrategies.useDefaults

The `useDefaults` connection strategy is the simplest connection strategy. It ignores the coordinates of user pointer and assigns the end anchor according to the `defaultAnchor` [paper option](#) and the end connection point according to the `defaultConnectionPoint` [paper option](#).

Thus, this connection strategy is equivalent to a connection strategy of `null`.

Example:

```
paper.options.connectionStrategy = joint.connectionStrategies.useDefaults;
```

connectors

Connectors take an array of link route points and generate SVG path commands so that the link can be rendered. The `connector` property of a link can be accessed with the `link.connector()` [function](#).

There are six built-in connectors in JointJS:

- `'straight'` - [connector with straight lines and different ways to handle corners \(point, rounded, bevelled, gap\)](#)
- `'jumpover'` - [connector with angled straight lines and bridges over link intersections](#)
- `'normal'` - [\(deprecated\) default connector with angled straight lines](#)
- `'rounded'` - [\(deprecated\) connector with straight lines and rounded edges](#)
- `'smooth'` - [connector interpolated as a bezier curve](#)
- `'curve'` - [connector interpolating as a curve defined by ending tangents](#)

Example:

```
link.connector('straight', {
  cornerType: 'line',
  cornerRadius: 20
});
```

The default connector is `'normal'`; this can be changed with the `defaultConnector` [paper option](#). Example:

```
paper.options.defaultConnector = {
  name: 'straight',
  args: {
    cornerType: 'line',
    cornerRadius: 20
  }
}
```

All of the built-in connectors accept the following optional argument, in addition to their own arguments:

raw	<i>boolean</i>	Should the router return the connection path as a <code>g.Path</code> rather than as a string? Default is <code>false</code> .

Example:

```
link.connector('straight', {
  raw: true
});
```

JointJS also contains mechanisms to define one's own [custom connectors](#).

Note that the modular architecture of JointJS allows mixing-and-matching connectors with [routers](#) as desired; for example, a link may be specified to use the `'jumpover'` connector on top of the `'manhattan'` router:

```

var link = new joint.shapes.standard.Link();
link.source(rect);
link.target(rect2);
link.router('manhattan');
link.connector('jumpover');

```

connectors.custom

New connectors can be defined in the `joint.connectors` namespace (e.g. `joint.connectors.myConnector`) or passed directly as a function to the `connector` property of a link (or to the `defaultConnector` [paper option](#)).

In either case, the connector function must return a `g.Path` representing the [SVG path data](#) that will be used to render the link. The function is expected to have the form `function(sourcePoint, targetPoint, routePoints, args):`

sourcePoint	<i>g.Point</i>	The source connection point.
targetPoint	<i>g.Point</i>	The target connection point.
routePoints	<i>Array<g.Point></i>	The points of the route, as returned by the router in use.
args	<i>object</i>	An object with additional optional arguments passed to the connector method by the user when it was called (the <code>args</code> property).

Example of a connector defined in the `joint.connectors` namespace:

```

joint.connectors.wobble = function(sourcePoint, targetPoint, vertices, args) {

    var SPREAD = args.spread || 20;

    var points = vertices.concat(targetPoint)
    var prev = sourcePoint;
    var path = new g.Path(g.Path.createSegment('M', prev));

    var n = points.length;
    for (var i = 0; i < n; i++) {

        var next = points[i];
        var distance = prev.distance(next);

        var d = SPREAD;
        while (d < distance) {
            var current = prev.clone().move(next, -d);
            current.offset(

```

```

        Math.floor(7 * Math.random()) - 3,
        Math.floor(7 * Math.random()) - 3
    );
    path.appendSegment(g.Path.createSegment('L', current));
    d += SPREAD;
}

path.appendSegment(g.Path.createSegment('L', next));
prev = next;
}

return path;
}

var link = new joint.shapes.standard.Link();
link.source(source);
link.target(target);

link.connector('wobble', {
    spread: 10
});

```

An example of a connector passed as a function is provided below. Notice that this approach does not enable passing custom args nor can it be serialized with the `graph.toJSON()` [function](#).

```

var link = new joint.shapes.standard.Link();
link.source(source);
link.target(target);

link.connector(function(sourcePoint, targetPoint, vertices, args) {

    var SPREAD = 20;

    var points = vertices.concat(targetPoint);
    var prev = sourcePoint;
    var path = new g.Path(g.Path.createSegment('M', prev));

    var n = points.length;
    for (var i = 0; i < n; i++) {

        var next = points[i];
        var distance = prev.distance(next);

        var d = SPREAD;
        while (d < distance) {
            var current = prev.clone().move(next, -d);
            current.offset(

```

```

        Math.floor(7 * Math.random()) - 3,
        Math.floor(7 * Math.random()) - 3
    );
    path.appendSegment(g.Path.createSegment('L', current));
    d += SPREAD;
}

path.appendSegment(g.Path.createSegment('L', next));
prev = next;
}

return path;
});

```

connectors.jumpover

The `'jumpover'` connector draws straight lines with small arcs in place of link-link intersections. (For the time being, it cannot detect intersections with `'smooth'` router links). It accepts the following additional arguments, which can be passed as the `connector.args` property:

size	<i>number</i>	The size of the jump (the diameter of the arc, or the length of the empty spot on the line). Defaults to <code>5</code> .
jump	<i>string</i>	The style of the jump. Either <code>'arc'</code> (using an <code>Arcto</code> SVG path command), <code>'cubic'</code> (using a <code>Curveto</code> path command as a normalized approximation to <code>Arcto</code>), or <code>'gap'</code> (leaving a blank space). Defaults to <code>'arc'</code> .
radius	<i>number</i>	The curve radius of the rounded corners. Default is <code>0</code> .

Example:

```

link.connector('jumpover', {
    type: 'gap'
});

```

connectors.normal

Deprecated. Use the `'straight'` connector instead.

Example:


```
link.connector('straight');
```

The `'normal'` connector is the default connector for links and it is the simplest connector. It simply connects provided route points with straight-line segments.

Example:

```
// deprecated
link.connector('normal');
```

connectors.rounded

Deprecated. Use the `'straight'` connector with `cornerType: 'cubic'` instead. (To exactly replicate the `'rounded'` connector functionality, you should also pass `precision: 0`.)

Example:

```
link.connector('straight', {
  cornerType: 'cubic',
  precision: 0,
  cornerRadius: 20
});
```

The `'rounded'` connector connects provided route points with straight lines while smoothing all corners on the route. It accepts one additional argument, which can be passed within the `connector.args` property:

radius	number	The curve radius of the rounded corners. Default is <code>10</code> .

Example:

```
// deprecated
link.connector('rounded', {
  radius: 20
});
```

connectors.smooth

The `'smooth'` connector interpolates route points using a cubic bezier curve.

Example:

```
link.connector('smooth');
```

(*Deprecated*) For the purposes of backwards compatibility, the `'smooth'` connector may also be enabled by setting the `link.smooth` property to `true`.

```
// deprecated
link.set('smooth', true)
```

connectors.straight

The `'straight'` connector connects provided route points with straight lines while handling corners on the route in different ways depending on provided arguments (point, rounded, bevelled, gap). It accepts four additional arguments, which can be passed within the `connector.args` property:

cornerType	string	(Optional) How should corners on the route be handled? The default is <code>'point'</code> . The available values are the following:	
		'point'	Route segments meet at corner point with no special behavior.
		'cubic'	A cubic curve segment is inserted to smooth the corner. The curve radius of the smoothed corner is determined by <code>cornerRadius</code> argument.
		'line'	A straight line segment is inserted at the corner to form a bevel. The distance of the two endpoints of the bevel from the corner point is determined by <code>cornerRadius</code> argument.
		'gap'	A gap is created at the corner. The distance of the two endpoints of the gap from the corner point is determined by the <code>cornerRadius</code> argument.
cornerRadius	number	(Optional) The distance of endpoints of line segments around corner points (e.g. curve radius for <code>'cubic'</code> corner type). Default is <code>10</code> . This argument is ignored for <code>cornerType: 'point'</code> .	
cornerPreserveAspectRatio	boolean	(Optional) Should distance of both endpoints be adjusted if one of them needs to be placed closer to a corner point than specified by the <code>cornerRadius</code> argument; or are the two allowed to be independent? (This situation happens when the distance between two consecutive	

		route points is lower than <code>cornerRadius * 2</code> .) Default is <code>false</code> . This argument is ignored for <code>cornerType: 'point'</code>
precision	<i>number</i>	(<i>Optional</i>) To how many decimal places should the returned point coordinates be rounded? Default is <code>1</code> . This argument is ignored for <code>cornerType: 'point'</code> .

Example of a bevelled connector:

```
link.connector('straight', {
  cornerType: 'line',
  cornerPreserveAspectRatio: true
});
```

Example of a rounded connector:

```
link.connector('straight', {
  cornerType: 'cubic',
  cornerRadius: 20
});
```

Example of a connector with gaps at corners:

```
link.connector('straight', {
  cornerType: 'gap',
  cornerRadius: 2
});
```

Example of a connector with simple angled lines connecting route points:

```
link.connector('straight');
```

connectors.curve

The `'curve'` connector interpolates route points using a catmull-rom curve converted into cubic bezier curve. Tangents at endings of a curve can be defined using several options.

Available options:

direction	<u><i>curve.Directions</i></u>	The overall direction of the curve. The default is <code>curve.Directions.AUTO</code> .

sourceDirection	<u>curve.TangentDirections</u> object number	The unit vector direction of the tangent at the source point. If the option is an object it must have x and y properties. If the option is a number, it represents an angle relative to the x-axis in the positive direction (counterclockwise).
targetDirection	<u>curve.TangentDirections</u> object number	The unit vector direction of the tangent at the target point. If the option is an object it must have x and y properties. If the option is a number, it represents an angle relative to the x-axis in the positive direction (counterclockwise).
sourceTangent	object	Vector of the tangent to the curve at the source point. Has priority over the <code>sourceDirection</code> and <code>direction</code> . Object must have x and y properties.
targetTangent	object	Vector of the tangent to the curve at the target point. Has priority over the <code>targetDirection</code> and <code>direction</code> . Object must have x and y properties.
distanceCoefficient	number	Coefficient of the tangent vector length relative to the distance between points. The default is <code>0.6</code> .
rotate	boolean	Whether the rotation of the source or target element should be taken into account. Only works for AUTO, HORIZONTAL and VERTICAL directions. The default is <code>false</code> .

The example of how to make the source point tangent always horizontal and target point tangent always go upwards:

```
link.connector('curve', {
  direction: curve.Directions.HORIZONTAL,
  targetDirection: curve.TangentDirections.UP
});
```

connectors.curve.Directions

Available values for the `direction` option.

curve.Directions.AUTO	Determines the tangent direction depending on the side of the element where the end point is located.
curve.Directions.HORIZONTAL	Limits tangents to horizontal directions (left and right) only.
curve.Directions.VERTICAL	Limits tangents to vertical directions (up and down) only.
curve.Directions.CLOSEST_POINT	Defines the direction tangent as a vector in the direction of the closest point.
curve.Directions.OUTWARDS	Defines the direction tangent as a vector in from the center of the element to the end point.

connectors.curve.TangentDirections

Available values for `sourceDirection` and `targetDirection` options.

curve.TangentDirections.AUTO	Determines the tangent direction depending on the side of the element where the end point is located.
curve.TangentDirections.UP	Sets the direction of the tangent upwards.
curve.TangentDirections.DOWN	Sets the direction of the tangent downwards.
curve.TangentDirections.LEFT	Sets the direction of the tangent to the left.
curve.TangentDirections.RIGHT	Sets the direction of the tangent to the right.
curve.TangentDirections.CLOSEST_POINT	Defines the direction tangent as a vector in the direction of the closest point.
curve.TangentDirections.OUTWARDS	Defines the direction tangent as a vector from the center of the element to the corresponding end point.

dia.attributes

The attributes in JointJS define how the graphics elements are to be rendered inside of the element and link views. All the standard SVG styling properties are available (both `kebab-case` and `camelCase` styles).

In addition JointJS modifies the behavior of existing attributes (the use of `calc()` for specifying attribute values) and defines new so-called "special" attributes and allows programmers to define their own.

`calc()`

The `calc()` function lets you perform calculations when specifying SVG attributes values.

Syntax:

The `calc()` function takes a simple expression as its parameter, with the expression's result used as the value. The expression can be any simple expression in one of the following forms:

- Variable

```
'calc(w)'
```

- Variable Addition or Subtraction

```
'calc(w + 5)'  
'calc(h - 10)'
```

- Multiplication Variable

```
'calc(2 * w)'  
'calc(0.5 * h)'
```

- Variable Division

```
'calc(w / 2)'  
'calc(h / 3)'
```

- Variable Division

```
'calc(w / 2)'  
'calc(h / 3)'
```

- Multiplication Variable Addition or Subtraction

```
'calc(2 * w + 5)'  
'calc(0.5 * h - 10)'
```

- **Variable** **Division** **Addition or Subtraction**

```
'calc(w / 2 + 5)'  
'calc(h / 3 - 10)'
```

Where:

- **Variable** is a symbol representing a value that can change, when the model attributes change (size, attrs).

variable	name	description
w	width	The current width of the model (<code>model.prop('size/width')</code>). The value can be bound to an SVGElement's size instead by using <u>ref</u> attribute.
h	height	The current height of the model (<code>model.prop('size/height')</code>). The value can be bound to an SVGElement's size instead by using <u>ref</u> attribute.
x	x	The current x coordinate of the SVGElement in the element's coordinate system. If the attribute is not bound to a specific SVGElement with <u>ref</u> attribute, the value of x is always zero.
y	y	The current y coordinate of the SVGElement in the element's coordinate system. If the attribute is not bound to a specific SVGElement with <u>ref</u> attribute, the value of y is always zero.
s	shortest	The shortest side of the rectangle. The minimum of <i>width</i> and <i>height</i> .
l	longest	The longest side of the rectangle. The maximum of <i>width</i> and <i>height</i> .
d	diagonal	The length of the diagonal of the rectangle of size <i>width</i> and <i>height</i> .

- **Multiplication** is an optional floating number factor of the variable. It's a number followed by the `*` symbol.

```
1.5 *
```

- **Division** is an optional floating number divisor of the variable. It's the `/` symbol followed by a number.

/ 2

- **Addition or Subtraction** is an optional floating number added or subtracted from the variable. It's a `+` or `-` symbol followed by a number.

+ 5

Notes:

- Expression is case-sensitive.
- The `+`, `-` and `*` operators do not require whitespace.
- No extra parentheses are allowed.
- It is permitted to nest `calc()` functions, in which case the inner ones are evaluated first. e.g.

```
'M 0 0 H calc(w - calc(h))'
```

It can be used with the following attributes:

- SVGRectElement
 - x
 - y
 - rx
 - ry
 - width
 - height
 - stroke-width
- SVGImageElement
 - x
 - y
 - width
 - height
- SVGLineElement
 - x1
 - x2
 - y1
 - y2
 - stroke-width
- SVGEllipseElement
 - cx
 - cy
 - rx
 - ry

- stroke-width
- SVGCircleElement
 - cx
 - cy
 - r
 - stroke-width
- SVGPolygonElement
 - points
 - stroke-width
- SVGPolylineElement,
 - points
 - stroke-width
- SVGPathElement
 - d
 - stroke-width
- SVGTextElement
 - x
 - y
 - dx
 - dy
 - font-size
 - stroke-width
- Any SVG element (e.g. SVGGElement)
 - transform
 - stroke-width

Examples:

```

el.resize(200, 100); // dia.Element

// <rect joint-selector="myRect" width="200" height="100" rx="20" ry="10" />
el.attr('myRect', {
  width: 'calc(w)',
  height: 'calc(h)',
  rx: 'calc(0.1*w)',
  ry: 'calc(0.1*h)'
});

// <image joint-selector="myImage" x="105" y="55" />
el.attr('myImage/x', 'calc(0.5*w+5)');
el.attr('myImage/y', 'calc(0.5*h+5)');

// <path joint-selector="myPath" d="M 10 50 190 50" />
el.attr('myPath/d', 'M 10 calc(0.5*h) calc(w-10) calc(0.5*h)')

```

```
// <polygon joint-selector="myPolygon" points="0,0 200,0 200,100 0,100" />
el.attr('myPolygon/d', '0,0 calc(w),0 calc(w),calc(h) 0,calc(h)');

// Resize the rectangle to match the text size with extra 5 pixels of padding
// <rect joint-selector="myTextBackground" />
// <text joint-selector="myText" >Some text</text>
el.attr('myTextBackground', {
  ref: 'myText',
  x: 'calc(x - 5)',
  y: 'calc(y - 5)',
  width: 'calc(w + 10)',
  height: 'calc(h + 10)',
});
```

Here is the list of all built-in attributes.

dia.attributes.atConnectionLengthIgnoreGradient

Move the subelement to the point at a given distance along the connection path but do not auto-orient it according to the path's gradient. Use a positive number to define the distance from the source of the link and a negative number from the target of the link. It is valid only within the LinkView context.

```
link.attr('rectSelector', { atConnectionLengthIgnoreGradient: 30, width: 10, height:
10, fill: 'red' });
```

dia.attributes.atConnectionLengthKeepGradient

alias: **atConnectionLength**

Move and auto-orient the subelement to the point at a given distance along the connection path. Use a positive number to define the distance from the source of the link and a negative number from the target of the link. It is valid only within the LinkView context.

```
link.attr('rectSelector', { atConnectionLength: 30, width: 10, height: 10, fill: 'red'
});
link.attr('rectSelector', { atConnectionLengthKeepGradient: 30, width: 10, height:
10, fill: 'red' });
```

dia.attributes.atConnectionRatioIgnoreGradient

Move the subelement to the point at a given ratio of the connection total length but do not auto-orient it according to the path's gradient. It accepts a number in the `[0,1]` range. It is valid only within the LinkView context.

```
link.attr('rectSelector', { atConnectionRatioIgnoreGradient: .5, width: 10, height: 10, fill: 'red' });
```

dia.attributes.atConnectionRatioKeepGradient

alias: `atConnectionRatio`

Move and auto-orient the subelement to the point at a given ratio of the connection total length. It accepts a number in the `[0,1]` range. It is valid only within the LinkView context.

```
link.attr('rectSelector', { atConnectionRatio: .5, width: 10, height: 10, fill: 'red' });
link.attr('rectSelector', { atConnectionRatioKeepGradient: .5, width: 10, height: 10, fill: 'red' });
```

dia.attributes.connection

If `true`, set the shape of the subelement to match the shape of the LinkView (set the `'d'` attribute to the result of the link connector). It's valid only for `SVGPathElement` within the LinkView context.

You can also provide an object with the following options:

Name	Type	Description
stubs	<i>Number</i>	If provided, display only the beginning and end stubs of the connection and hide the remaining central section. A positive value determines the length of each stub. If the value is negative, it determines the length of the hidden section of the connection between the two stubs.

```
link1.attr('pathSelector', { connection: true, stroke: 'red', fill: 'none' });
link2.attr('pathSelector', { connection: { stubs: -20 } });
```

dia.attributes.containerSelector

Designate another subelement of the cell as a proxy target for the cell's container, whenever this subelement (the one on which this attribute is defined) becomes the target of an embedding. (Usually, this attribute needs to be defined on the ``root`` subelement of the cell, since that is the default target for the cell's container.) Expects a selector.

A change in the cell's container affects the following characteristics of the cell view:

- embedding highlighter

```
// `standard.Rectangle` has `root`, `body` and `label` subelements
const rect = new joint.shapes.standard.Rectangle();
// we want to have the `label` on the outside and under the `body` of the rectangle:
rect.attr(['label'], { refY: '100%', textVerticalAnchor: 'top' });
// normally, when a user tries to embed a child to the rectangle, the `root`
// subelement would be highlighted (the wrapper of `body` and `label`)
// however, we want only the `body` to be highlighted, so we need to specify it as a
// proxy for `root`:
rect.attr(['root', 'containerSelector'], 'body');
```

dia.attributes.displayEmpty

Valid only for `<text>` subelements.

If set to `true`, the SVGTextElement with empty content will be rendered. It's useful when the text needs to be editable.

dia.attributes.event

The `event` attribute makes the selected node and its descendants trigger an arbitrary event when clicked (mousedown/touchstart). This event is triggered on the view itself and the paper. The paper handler is called with the signature `cellView, evt, x, y`, while the cell view handler is called only with `evt, x, y`.

```
element.attr({
  image: {
    // pointerdown on the image SVG node will trigger the `element:delete` event
    event: 'element:delete',
    xlinkHref: 'trash.png'
  },
  width: 20,
  height: 20
})
```

```

});
// Binding handler to the event
paper.on('element:delete', function(elementView, evt) {
  // Stop any further actions with the element view e.g. dragging
  evt.stopPropagation();
  if (confirm('Are you sure you want to delete this element?')) {
    elementView.model.remove();
  }
});

```

dia.attributes.fill

The `fill` attribute becomes a special attribute only in case it's defined as an object, instead of the usual SVG syntax (e.g. `"#ffaabb"`). If it's defined as an object, it is assumed to be either a gradient or pattern definition. It must have the form defined by the `defineGradient()` (resp. `definePattern()`) paper method.

Gradient

```

element.attr('rect/fill', {
  type: 'linearGradient',
  stops: [
    { offset: '0%', color: '#E67E22' },
    { offset: '20%', color: '#D35400' },
    { offset: '40%', color: '#E74C3C' },
    { offset: '60%', color: '#C0392B' },
    { offset: '80%', color: '#F39C12' }
  ]
});

```

Pattern

```

element.attr('rect/fill', {
  type: 'pattern',
  attrs: {
    width: 10,
    height: 10
  },
  markup: [{
    tagName: 'polygon',
    attributes: {
      points: '0,0 2,5 0,10 5,8 10,10 8,5 10,0 5,2'
    }
  ]
});

```

```
    }]  
  });
```

dia.attributes.filter

The `filter` attribute becomes a special attribute only in case it's defined as an object, instead of the usual SVG syntax (e.g. `"url(#myfilter)"`). If it's defined as an object, it must have the form defined by the [defineFilter\(\)](#) paper method.

```
element.attr('rect/filter', {  
  name: 'dropShadow',  
  args: {  
    dx: 2,  
    dy: 2,  
    blur: 3  
  }  
});
```

dia.attributes.highlighterSelector

Designate another subelement of the cell as a proxy target for the cell's magnet, whenever this subelement (the one on which this attribute is defined) becomes the source/target of a link. (Usually, this attribute needs to be defined on the ``root`` subelement of the cell, since that is the default target for the cell's magnet.) Expects a selector.

A change in the cell's magnet affects the following characteristics of the cell view:

- magnet highlight

It has no effect on the connection validation and link model attributes.

```
// `standard.Rectangle` has `root`, `body` and `label` subelements  
const rect = new joint.shapes.standard.Rectangle();  
// we want to have the `label` on the outside and under the `body` of the rectangle:  
rect.attr(['label'], { refY: '100%', textVerticalAnchor: 'top' });  
// normally, highlighters would appear around the `root` subelement (the wrapper of  
// `body` and `label`)  
// however, we want highlighters to highlight only to the `body` subelement, so we  
// need to specify it as a proxy for `root`:  
rect.attr(['root', 'highlighterSelector'], 'body');
```

dia.attributes.magnet

When set to `true`, the subelement can become the source/target of a link during link reconnection. Useful for so called 'ports'. When set to `magnet: 'passive'`, a new link won't be added to the graph when a user clicks on the respective magnet. This use case is common for the creation of 'input' ports.

```
const paper = new joint.dia.Paper({
  // Other Paper options
  // The default behaviour of magnet validation
  validateMagnet: function(_cellView, magnet, _evt) {
    return magnet.getAttribute('magnet') !== 'passive';
  }
});

const portIn = {
  label: {
    // Label position and markup
  },
  attrs: {
    portBody: {
      // Other portBody attributes
      magnet: 'passive' // No link added upon user click
    },
    label: { text: 'port' }
  },
  markup: [{
    tagName: 'rect',
    selector: 'portBody'
  }]
};
```

dia.attributes.magnetSelector

Designate another subelement of the cell as a proxy target for the cell's magnet, whenever this subelement (the one on which this attribute is defined) becomes the source/target of a link. (Usually, this attribute needs to be defined on the `'root'` subelement of the cell, since that is the default target for the cell's magnet.) Expects a selector.

A change in the cell's magnet affects the following characteristics of the cell view:

- link anchor
- link connection point
- connector
- router

It has no effect on the connection validation, magnet highlighter and the `source` and `target` link model attributes.

```
// `standard.Rectangle` has `root`, `body` and `label` subelements
const rect = new joint.shapes.standard.Rectangle();
// we want to have the `label` on the outside and under the `body` of the rectangle:
rect.attr(['label'], { refY: '100%', textVerticalAnchor: 'top' });
// normally, links would connect to the `root` subelement (the wrapper of `body` and `label`)
// however, we want links to connect only to the `body` subelement, so we need to
// specify it as a proxy for `root`:
rect.attr(['root', 'magnetSelector'], 'body');
```

dia.attributes.port

An object containing at least an `id` property. This property uniquely identifies the port. If a link gets connected to a magnet that has also a `port` object defined, the `id` property of the port object will be copied to the `port` property of the source/target of the link.

dia.attributes.props

Special attribute for setting the properties of the `HTMLElement`. The value is not set as a node attribute, but using the DOM property.

```
element.attr('input', {
  props: { value: 'text', readonly: false },
  type: 'text',
  placeholder: 'Enter text'
});
```

`props` is an object with one or more of the following properties: `value`, `checked`, `disabled`, `readOnly`, `contentEditable`, `multiple`, `selected`, `indeterminate`.

For example, to set the content of the `textarea` element, use the following syntax:

```
element.attr('textarea/props/value', 'textarea content');
```

When setting a value of the `select` element, pass the value of the `option` element.

```
element.attr('select/props/value', 'option1');
```

For the `select` with multiple options, pass an array of values.


```
element.attr('select/props/value', ['option1', 'option2'], { rewrite: true });
```

dia.attributes.ref

Markup or CSS selector pointing to an element that is used as a reference for relative positioning attributes.

dia.attributes.refCx

Set `cx` attribute of the subelement relatively to the width of the element referenced to by the selector in `ref` attribute. If the value is in the `[0, 1]` interval (or expressed in percentages, e.g. `'80%'`), the `cx` of the subelement will be set as a percentage of the width of the referenced element. If the value is `<0` or `>1`, the height of the subelement will be smaller/bigger than the width of the referenced element by the amount specified. Note that this makes sense only for SVG elements that support `rx` and `ry` attributes, such as `<ellipse>`.

dia.attributes.refCy

Set `cy` attribute of the subelement relatively to the height of the element referenced to by the selector in `ref` attribute. If the value is in the `[0, 1]` interval (or expressed in percentages, e.g. `'80%'`), the `cy` of the subelement will be set as a percentage of the height of the referenced element. If the value is `<0` or `>1`, the height of the subelement will be smaller/bigger than the height of the referenced element by the amount specified. Note that this makes sense only for SVG elements that support `rx` and `ry` attributes, such as `<ellipse>`.

dia.attributes.refDKeepOffset

Set `d` attribute of the `<path>` subelement relatively to the dimensions and position of the element referenced by the selector in the `ref` attribute. The `refD` path data is scaled so that the path's dimensions match the reference element's dimensions, and translated so that the path's origin matches the origin of the reference element.

The original path data offset is preserved. This means that if the top-left corner of the `refD` bounding box does not lie at `0,0`, the gap between the path and origin is preserved in the rendered shape, as well.

```
var Path = joint.dia.Element.define('examples.Path', {
  attrs: {
    path: {
      refDKeepOffset: 'M 10 10 30 10 30 30 z', // path offset is 10,10
      fill: 'red',
      stroke: 'black'
    }
  }
});
```

```

    }
  }, {
    markup: 'path'
  });

var p = (new Path()).resize(40, 40).addTo(graph);
// the rendered path's `d` attribute will be 'M 10 10 50 10 50 50 z'
// can be obtained by `p.findView(paper).vel.findOne('path').attr('d');`

```

dia.attributes.refDResetOffset

alias: refD

Set `d` attribute of the `<path>` subelement relatively to the dimensions and position of the element referenced by the selector in the `ref` attribute. The `refD` path data is scaled so that the path's dimensions match the reference element's dimensions, and translated so that the path's origin matches the origin of the reference element.

The original path data offset is not preserved. This means that if the top-left corner of the `refD` bounding box does not lie at `0,0`, the path is translated so that this gap disappears. The rendered path then fits perfectly into the reference element's bounding box.

```

var Path = joint.dia.Element.define('examples.Path', {
  attrs: {
    path: {
      refDResetOffset: 'M 10 10 30 10 30 30 z', // path offset of 10,10 will be
discarded
      fill: 'red',
      stroke: 'black'
    }
  }
}, {
  markup: 'path'
});

var p = (new Path()).resize(40, 40).addTo(graph);
// the rendered path's `d` attribute will be 'M 0 0 40 0 40 40 z'
// can be obtained by `p.findView(paper).vel.findOne('path').attr('d');`

```

dia.attributes.refDx

alias: ref-dx

Make x-coordinate of the subelement relative to the right edge of the element referenced to by the selector in `ref` attribute.

dia.attributes.refDy

Make y-coordinate of the subelement relative to the bottom edge of the element referenced to by the selector in `ref` attribute.

dia.attributes.refHeight

alias: ref-height

Set height of the subelement relatively to the height of the element referenced to by the selector in `ref` attribute. If the value is in the `[0, 1]` interval (or expressed in percentages, e.g. `'80%'`), the height of the subelement will be set as a percentage of the height of the referenced element. If the value is `<0` or `>1`, the height of the subelement will be smaller/bigger than the height of the referenced element by the amount specified. Note that this makes sense only for SVG elements that support `width` and `height` attributes, such as `<rect>`.

dia.attributes.refHeight2

Same as `refHeight`. Useful when one needs to resize an element absolutely and relatively at the same time. Note that all percentages are relative to 100% of the referenced height.

```
{ refHeight: 20, refHeight2: '-75%' } // resizes the element to 25% of the reference height (25% = 100% - 75%) plus extra 20 pixels
```

dia.attributes.refPointsKeepOffset

Set the `points` attribute of the `<polygon>` or `<polyline>` subelement relatively to the dimensions and position of the element referenced by the selector in the `ref` attribute. The `refPoints` are scaled so that the subelement's dimensions match the reference element's dimensions, and translated so that their origin matches the origin of the reference element.

The original points offset is preserved. This means that if the top-left corner of the `refPoints` bounding box does not lie at `0,0`, the gap between the points and origin is preserved in the rendered shape, as well.

```
var Polygon = joint.dia.Element.define('examples.Polygon', {  
  attrs: {
```

```

        polygon: {
            refPoints: '10,10 30,10 30,30', // points offset is 10,10
            fill: 'red',
            stroke: 'black'
        }
    }, {
        markup: 'polygon'
    });

var p = (new Polygon()).resize(40, 40).addTo(graph);
// the rendered polygon's `points` attribute will be '10,10 50,10 50,50'
// can be obtained by `p.findView(paper).vel.findOne('polygon').attr('points');`

```

dia.attributes.refPointsResetOffset

alias: refPoints

Set the `points` attribute of the `<polygon>` or `<polyline>` subelement relatively to the dimensions and position of the element referenced by the selector in the `ref` attribute. The `refPoints` are scaled so that the subelement's dimensions match the reference element's dimensions, and translated so that their origin matches the origin of the reference element.

The original points offset is not preserved. This means that if the top-left corner of the `refPoints` bounding box does not lie at `0,0`, the subelement is translated so that this gap disappears. The rendered subelement then fits perfectly into the reference element's bounding box.

```

var Polygon = joint.dia.Element.define('examples.Polygon', {
    attrs: {
        polygon: {
            refPoints: '10,10 30,10 30,30', // points offset of 10,10 will be
discarded
            fill: 'red',
            stroke: 'black'
        }
    }
}, {
    markup: 'polygon'
});

var p = (new Polygon()).resize(40, 40).addTo(graph);
// the rendered polygon's `points` attribute will be '0,0 40,0 40,40'
// can be obtained by `p.findView(paper).vel.findOne('polygon').attr('d');`

```

dia.attributes.refRCircumscribed

Set the `r` attribute of the subelement relatively to the circumscribed size (length of the diagonal of the bounding box) of the element referenced by the selector in the `ref` attribute. If the value is in the `[0, 1]` interval (or expressed in percentages, e.g. `'80%'`), the `r` of the subelement will be set as a percentage of the size of the referenced element. If the value is `<0` or `>1`, the size of the subelement will be smaller/bigger than the size of the referenced element by the amount specified. Note that this makes sense only for SVG elements that support `r` attribute, such as `<circle>`.

dia.attributes.refRInscribed

alias: refR

Set the `r` attribute of the subelement relatively to the inscribed size (width or height of the bounding box, whichever is smaller) of the element referenced by the selector in the `ref` attribute. If the value is in the `[0, 1]` interval (or expressed in percentages, e.g. `'80%'`), the `r` of the subelement will be set as a percentage of the size of the referenced element. If the value is `<0` or `>1`, the size of the subelement will be smaller/bigger than the size of the referenced element by the amount specified. Note that this makes sense only for SVG elements that support `r` attribute, such as `<circle>`.

dia.attributes.refRx

Set `rx` attribute of the subelement relatively to the width of the element referenced to by the selector in `ref` attribute. If the value is in the `[0, 1]` interval (or expressed in percentages, e.g. `'80%'`), the `rx` of the subelement will be set as a percentage of the width of the referenced element. If the value is `<0` or `>1`, the height of the subelement will be smaller/bigger than the width of the referenced element by the amount specified. Note that this makes sense only for SVG elements that support `rx` and `ry` attributes, such as `<ellipse>`.

dia.attributes.refRy

Set `ry` attribute of the subelement relatively to the height of the element referenced to by the selector in `ref` attribute. If the value is in the `[0, 1]` interval (or expressed in percentages, e.g. `'80%'`), the `ry` of the subelement will be set as a percentage of the height of the referenced element. If the value is `<0` or `>1`, the height of the subelement will be smaller/bigger than the height of the referenced element by the amount specified. Note that this makes sense only for SVG elements that support `rx` and `ry` attributes, such as `<ellipse>`.

dia.attributes.refWidth

alias: ref-width

Set width of the subelement relatively to the width of the element referenced to by the selector in `ref` attribute. If the value is in the `[0, 1]` interval (or expressed in percentages, e.g. `'80%'`), the width of the subelement will be set as a percentage of the width of the referenced element. If the value is `<0` or `>1`, the width of the subelement will be smaller/bigger than the width of the referenced element by the amount specified. For example, `'ref-width': .75` sets the width of the subelement to `75%` of the width of the referenced element. `'ref-width': 20` makes the width to be `20px` less than the width of the referenced element. Note that this makes sense only for SVG elements that support `width` and `height` attributes, such as `<rect>`.

dia.attributes.refWidth2

Same as `refWidth`. Useful when one needs to resize an element absolutely and relatively at the same time. Note that all percentages are relative to 100% of the referenced width.

```
{ refWidth: 20, refWidth2: '-75%' } // resizes the element to 25% of the reference width (25% = 100% - 75%) plus extra 20 pixels
```

dia.attributes.refX

alias: ref-x

Make x-coordinate of the subelement relative to the x-coordinate of the element referenced to by the selector in `ref` attribute. If `ref-x` is in the `(0, 1)` interval (or expressed in percentages, e.g. `'80%'`), the offset is calculated from the fraction of the bounding box of the referenced element. Otherwise, it is an absolute value in pixels.

dia.attributes.refX2

Same as `refX`. Useful when one needs to position an element absolutely and relatively at the same time.

```
{ refX: '50%', refX2: 20 } // moves the element by 50% of the referenced width plus extra 20 pixels.
```

dia.attributes.refY

alias: ref-y

Make y-coordinate of the subelement relative to the y-coordinate of the element referenced to by the selector in `ref` attribute. If `ref-y` is in the `(0,1)` interval (or expressed in percentages, e.g. `'80%'`), the offset is calculated from the fraction of the bounding box of the referenced element. Otherwise, it is an absolute value in pixels.

dia.attributes.refY2

Same as `refY`. Useful when one needs to position an element absolutely and relatively at the same time.

```
{ refY: '50%', refY2: 20 } // moves the element by 50% of the referenced height plus
extra 20 pixels.
```

dia.attributes.resetOffset

If set to `true`, the subelement offset (the distance from `x0 y0` to the most top-left point) will be reset to `x0 y0`.

```
element.attr({
  path: {
    // The path bbox for `d="M 10 10 20 20"` is x10 y10 w10 h10.
    d: 'M 10 10 20 20',
    // The path bbox will be changed to x0 y0 w10 h10.
    // This has same effect as passing path with `d="M 0 0 10 10"`
    resetOffset: true
  }
});
```

dia.attributes.sourceMarker

Valid for `<path>` subelements. It draws an SVG element at the beginning of a path (the source of the link). The element is automatically rotated based on the path direction. It must have the form defined by the `defineMarker()` paper method.

```
link.attr('line/sourceMarker', {
  type: 'circle', // SVG Circle
  fill: '#666',
  stroke: '#333',
  r: 5, // radius of the circle
  cx: 5 // move the centre of the circle 5 pixels from the end of the path
});
```

dia.attributes.stroke

The `stroke` attribute becomes a special attribute only in case it's defined as an object. This has the exact same behaviour as the `fill` attribute.

dia.attributes.style

An object containing CSS styles for a subelement.

dia.attributes.targetMarker

Valid for `<path>` subelements. The same as `sourceMarker` but draws an SVG element at the end of the path. Note the coordinate system for drawing is rotated by 180 degrees for convenience (this allows to use the same marker for source and target and have them both face the connected elements).

dia.attributes.text

Valid only for `<text>` subelements.

The `text` attribute contains the text that should be set on the subelement. If the provided string does not contain any newline characters (`'\n'`), the text is set directly as the content of the `<text>` subelement. Alternatively, if the provided string has multiple lines, each line becomes the content of a `<tspan>` child of the `<text>` subelement.

If you need the text to be automatically wrapped inside the `<text>` subelement, you should use the `textWrap` [attribute](#) instead. It can automatically add line breaks into the provided string as necessary, and mark them with newline characters.

dia.attributes.textPath

Valid only for `<text>` subelements. `textPath` can be either a string or an object.

If it is a string, it specifies a path the text will be rendered along.

If it is an object, then it can contain a `d` property that specifies the path the text will be rendered along. Alternatively use `selector` property, which is useful for targeting a specific `SVGPathElement` within the shape. The option expects a string selector (CSS or JSONMarkup selector). Use `startOffset` property to control the text position on the path (e.g. `'50%', 20`). See the `Vectorizer.text` [method](#) for more details.

dia.attributes.textVerticalAnchor

Valid only for `<text/>` subelements. The attribute is used to (top-, middle- or bottom-) align a text, relative to a point determined by reference to provided `x`, `y`, `refX`, `refY`, and similar attributes. See the `Vectorizer.text` [method](#) for more details.

dia.attributes.textWrap

Valid only for `<text>` subelements.

It enables the text wrapping for the `text` [attribute](#) (automatically breaks the lines to fit within the reference bounding box).

It expects an object with several optional properties.

`width` and `height` adjust the final size of the wrapped text.

- If not provided, the text is wrapped to fit the reference bounding box
- It can be a `calc()` expression (e.g. `'calc(w - 10)'`)
- Percentage expression (e.g. `'50%'` is half the width or height)
- A positive number is an absolute dimension (e.g. `50` fits the text within 50 pixels)
- A negative number is a relative dimension (e.g. `-10` is an equivalent to `'calc(w - 10)'` resp. `'calc(h - 10)'`)
- Use `null` to disable wrapping in a given dimension

If the text cannot fit into the bounding box as specified, the overflowing words are cut off. If the `ellipsis` option is provided, an ellipsis string is inserted before the cutoff. If no words can fit into the bounding box at all, no text is inserted.

```
element1.attr('label', {
  text: 'Text to wrap.',
  textWrap: {
    width: 'calc(w - 10)', // reference width minus 10
    height: null, // no height restriction
  }
});

element2.attr('label', {
  text: 'lorem ipsum dolor sit amet consectetur adipiscing elit',
  textWrap: {
    width: -10, // reference width minus 10
    height: '50%', // half of the reference height
    ellipsis: true // could also be a custom string, e.g. '...!?'
  }
});
```

```
}  
});
```

For more info see [util.breakText](#).

dia.attributes.title

Add the text-only description in form of a `<title>` element to the associated node. The title is not rendered as part of the graphics, but it's displayed as a tooltip.

```
element.attr('rect/title', 'Description of the rectangle');
```

dia.attributes.vertexMarker

Valid for `<path>` subelements. The same as [sourceMarker](#) but draws SVG elements at every vertex of the path.

dia.attributes.xAlignment

alias: x-alignment

If set to `'middle'`, the subelement will be centered around its new x-coordinate.

dia.attributes.yAlignment

alias: y-alignment

If set to `'middle'`, the subelement will be centered around its new y-coordinate.

dia.Cell

The basic model for diagram cells. It's a [Backbone model](#) with a few additional properties and methods. Most importantly, every cell has a [unique ID](#) that is stored in the `id` property.

dia.Cell.define

`define(type [, defaultAttributes, prototypeProperties, staticProperties])`

Helper to define a new Cell class or extend an existing one.

The `type` must be a unique identifier of the class, which determines the location of the class definition in the `joint.shapes` namespace (`type` is the path to the class definition delimited by dots: `.`). When creating an instance of the cell, attributes will be set to the value from the `defaultAttributes` , unless overridden by subclass or instance attributes.

```
// Define a new Ellipse class in `joint.shapes.examples` namespace
// Inherits from generic Element class
var Ellipse = joint.dia.Element.define('examples.Ellipse', {
  // default attributes
  markup: [{
    tagName: 'ellipse',
    selector: 'ellipse' // not necessary but faster
  }],
  attrs: {
    ellipse: {
      fill: 'white',
      stroke: 'black',
      strokeWidth: 4,
      rx: 'calc(0.5*w)',
      ry: 'calc(0.5*h)',
      cx: 'calc(0.5*w)',
      cy: 'calc(0.5*h)'
    }
  }
});

// Instantiate an element
var ellipse = (new Ellipse()).position(100, 100).size(120, 50).addTo(graph);

// Define a new ColoredEllipse class
// Inherits from Ellipse
var ColoredEllipse = Ellipse.define('examples.ColoredEllipse', {
  // overridden Ellipse default attributes
  // other Ellipse attributes preserved
  attrs: {
    ellipse: {
      fill: 'lightgray'
    }
  }
}, {
  // prototype properties
  // accessible on `this(...)` - as well as, more precisely, `this.prototype`.
  (...)`
  // useful for custom methods that need access to this instance
```

```

    // shared by all instances of the class
    randomizeStrokeColor: function() {
        var randomColor = '#' + ('000000' + Math.floor(Math.random() *
16777215).toString(16)).slice(-6);
        return this.attr('ellipse/stroke', randomColor);
    }
}, {
    // static properties
    // accessible on `this.constructor(...)`
    // useful for custom methods and constants that do not need an instance to
operate
    // however, a new set of static properties is generated every time the
constructor is called
    // (try to only use static properties if absolutely necessary)
    createRandom: function() {
        return (new ColoredEllipse()).randomizeStrokeColor();
    }
});

// Instantiate an element
var coloredEllipse = ColoredEllipse.createRandom().position(300, 100).size(120,
50).addTo(graph);

```

dia.Cell.markup

markup

Either an XML string or JSON markup specifying an array of JSON elements. Also JSON markup can be described using an [svg](#) tagged template. Used as a template to build DOM Elements on the fly when the associated cellView is rendered.

XML String

A valid XML string that contains either a single `tagName` or XML that can be parsed with [DOMParser](#) .

```

markup: 'rect'
markup: '<rect class="rectangle"/>'
markup: '<rect><g><circle/><circle/></g>'

```

Note that defining Cell markup with XML strings is slower than defining it with JSON arrays, precisely because of the need for parsing. We strongly recommend you to use the [JSON markup](#) for your Cell definitions.

JSON Markup

JSON markup is defined recursively as an array of `JSONElement` or strings representing text nodes, where `JSONElement` is a plain object with the following properties:

- `tagName` *(string) (required)* The type of element to be created.
- `selector` *(string)* A unique selector for targeting the element within the `attr` cell attribute.
- `groupSelector` *(string | string[])* A selector for targeting multiple elements within the `attr` cell attribute. The group selector name must not be the same as an existing selector name.
- `namespaceURI` *(string)* The namespace URI of the element. It defaults to SVG namespace ["http://www.w3.org/2000/svg"](http://www.w3.org/2000/svg).
- `attributes` *(object with attributes name-value pairs)* Attributes of the element.
- `style` *(object with CSS property-value pairs)* The `style` attribute of the element.
- `className` *(string)* The `class` attribute of the element.
- `children` *(JSONMarkup)* The children of the element.
- `textContent` *(string)* The text content of the element.

```
// single DOM element
markup: [{ tagName: 'rect' }]

// multiple DOM elements
markup: [{
  tagName: 'rect',
  selector: 'body'
}, {
  tagName: 'text',
  selector: 'label',
  attributes: {
    'stroke': 'none'
  }
}]

// nested DOM elements
markup: [{
  tagName: 'g',
  children: [{
    tagName: 'circle',
    selector: 'circle1',
    groupSelector: 'circles'
  },
  'text content',
  {
    tagName: 'circle',
    selector: 'circle2',
    groupSelector: 'circles'
  }
]}]
```

JSON Markup using tagged template

JSON markup also can be defined using an `svg` tagged template. This tagged template converts SVG representation of the markup into a JSON markup object. Let's write the previous example using an `svg` tagged template:

```
const svg = joint.util.svg;

// single DOM element
markup: svg`<rect\>`

// multiple DOM elements
markup: svg`
  <rect @selector="body"\>
    <text
      @selector="label"
      stroke="none"
    \>`

// nested DOM elements
markup: svg`
  <g>
    <circle
      @selector="circle1"
      @group-selector="circles"
    />
    text content
    <circle
      @selector="circle2"
      @group-selector="circles"
    />
  </g>`
```

JSON Markup attributes

Anything you define in `markup` is evaluated once at CellView creation (the DOM elements and their attributes). That means it's important to think about the runtime of your application. If you have SVG attributes that don't change throughout the runtime, you can add them to the `markup`.

As `markup` is something all instances of the element type are expected to have in common, inheriting from the subtype prototype is more efficient. Nevertheless, it is still possible to provide custom markup to individual instances of your class by providing markup later.

Anything in the `attrs` attribute is evaluated on every change of the model (e.g. a `resize` or an `attrs` change). As JointJS special attributes mostly depend on the current state of the model (`size` , `attrs` , `angle`), they should always be defined inside `attrs`.

SVG attributes that are modified at some point in the application should also be added to `attrs` (e.g. user changes the color of the element).

```

joint.dia.Element.define('standard.Rectangle', {
  attrs: {
    body: {
      width: 'calc(w)',
      height: 'calc(h)',
      stroke: 'red',
      fill: '#333333' // We wan't to modify the fill color of our instances
    },
    label: {
      // attributes
    }
  }
}, {
  markup: [{
    tagName: 'rect',
    selector: 'body',
    attributes: {
      'stroke-width': 2 // Use native SVG kebab-case for attributes in markup
    }
  }, {
    tagName: 'text',
    selector: 'label',
    attributes: {
      'stroke': 'none' // We don't need any instances to have a stroke value
    }
  }
  for text
  }
]);

```

Setting attributes with selectors

Here are simple rules how SVG attributes are set on the nodes when one uses combination of `selector` and `groupSelector`.

- `selector` is unique i.e. can target a single node only.
- `selector` takes precedence over `groupSelector`.
- `groupSelector` targeting n nodes takes precedence over `groupSelector` targeting m nodes for $n < m$.
When $n === m$ the order how the attributes are applied is unspecified.

In the example below, the circle with the selector `circle1` is filled with `"red"` color. All the other circles are filled with `"blue"` color.

```

cell.attr({
  circle1: { fill: 'red' },

```

```
    circles: { fill: 'blue', stroke: 'black' }  
  });
```

dia.Cell.prototype.getParentCell

cell.[getParentCell\(\)](#)

Return the parent cell of `cell` or `null` if there is none.

dia.Cell.prototype.isElement

cell.[isElement\(\)](#)

Always returns `false`. The reason the method is here is that both `joint.dia.Element` and `joint.dia.Link` inherit from `joint.dia.Cell`. This method is useful if you don't know what the cell is. Calling `cell.Element\(\)` is equivalent to `cell instanceof joint.dia.Element`. Example:

```
var cell = graph.getCell(myId)  
if (cell.isElement()) {  
  // Do something if the cell is an element.  
}
```

dia.Cell.prototype.isLink

element.[isLink\(\)](#)

Always returns `false`. The reason the method is here is that both `joint.dia.Element` and `joint.dia.Link` inherit from `joint.dia.Cell`. This method is useful if you don't know what the cell is. Calling `cell.isLink\(\)` is equivalent to `cell instanceof joint.dia.Link`. Example:

```
var cell = graph.getCell(myId)  
if (cell.isLink()) {  
  // Do something if the cell is a link.  
}
```

dia.Cell.prototype.parent

cell.[parent\(\)](#)

Return the `parent` property of the cell.

If the cell is a part of an embedding, the `id` of the parent cell is returned as a string.

If you need to be sure that a Cell is returned (and not its `id`), use the `cell.getParentCell` [function](#) instead.

```
cell.parent(parentId [, opt])
```

Set the `parent` property of the cell to `parentId` provided.

dia.Cell.prototype.z

```
cell.z()
```

Return `z` – the stacking order (an equivalent to HTML `z-index`).

When `z` is `undefined` or `null`, function returns `0`.

dia.CellView

The view for the `joint.dia.Cell` model. It inherits from `Backbone.View` and is responsible for:

- Rendering a cell inside of a paper
- Handling the cell's pointer events
- Provides various methods for working with the cell (visually)

To find the view associated with a specific cell (model), use the `findViewByModel` method of the paper. For example:

```
var cellView = paper.findViewByModel(cell);
```

dia.CellView.custom

When you define a custom CellView, which listens to the underlying model changes and update itself (modifying its sub-elements directly) you should follow the instructions below in order to make the view work correctly in the `async` mode.

Note, that when a view needs an update (a model attribute has changed e.g. `size` has changed), it requests the update from the paper first. The paper confirms the update immediately (`sync` mode) or in the next animation frame (`async` mode). The updates may be also held by the paper as long as the paper is `frozen` and released when the paper changes its state to unfrozen.

The update requests are sent to the paper via `flags` and later received back all at once. The paper accumulates flags received and confirms the updates when the right time has come.

Methods and terminology

`flagLabel` - is an arbitrary string or array of strings

`flags` - are encoded flag labels

`CellView.prototype.presentationAttributes` - is an object that maps model attributes to flag labels.

`CellView.prototype.confirmUpdate(flags, opt)` - This method receives all scheduled flags and based on them updates the view

`CellView.prototype.hasFlag(flags, flagLabel)` - Checks whether a `flagLabel` is present in the current update.

`CellView.addPresentationAttributes(presentationAttributes)` - Extends the `CellView` presentation attributes with another `presentationAttributes`. - The method makes sure the original `CellView` attributes are preserved. It returns a new object with all presentation attributes.

Usage

```
const FadeView = joint.dia.ElementView.extend({
  // Make sure that all super class presentation attributes are preserved
  presentationAttributes: joint.dia.ElementView.addPresentationAttributes({
    // mapping the model attributes to flag labels
    faded: 'flag:opacity'
  }),
  confirmUpdate(flags, ...args) {
    joint.dia.ElementView.prototype.confirmUpdate.call(this, flags, ...args);
    if (this.hasFlag(flags, 'flag:opacity')) this.toggleFade();
  },
  toggleFade() {
    this.el.style.opacity = this.model.get('faded') ? 0.5 : 1;
  }
});
```

`dia.CellView.prototype.findAttribute`

`cellView.findAttribute(attribute, node)`

Return the value of the specified `attribute` of `node`.

If `node` does not set a value for `attribute`, start recursing up the DOM tree from `node` to look for `attribute` at the ancestors of `node`. If the recursion reaches `CellView`'s own node and `attribute` is not found even there, return `null`.

dia.CellView.prototype.findBySelector

`cellView.findBySelector([selector, root, selectors])`

Return an array of subelements of this CellView which are identified by `selector`. If no such subelements are found, return an empty array.

This method mimics jQuery element search methods in that it always returns an array. If you are only expecting to find a single subelement (most common scenario), you can get it by accessing the first element in the returned array – e.g.

```
this.findBySelector('body')[0] .
```

If you do not specify a `selector`, the `root` of the CellView is returned as the only element of the return array. For backwards compatibility, the same applies also if you provide the special value `selector = '.'`. By default `root = this.el` (the wrapping `<g>` SVGElement of the CellView), but if you provide a different `root` (see below), then that value is returned instead.

By default, JointJS searches for matching subelements among descendants of `this.el` SVGElement. As an advanced feature, you may provide a different `root` – e.g. an `el` SVGElement of another CellView. Then JointJS searches for matching subelements among the descendants of that SVGElement.

By default, JointJS uses `this.selectors` array for reference. (The `selectors` array is a cached structure of this CellView's SVGElement descendants as defined by the markup [attribute of the Cell model](#) from which the CellView was initialized.) As an advanced feature, you may provide a different `selectors` array for the search – e.g. a `selectors` array of another CellView.

(Deprecated) For backwards compatibility, if no matching subelement is found among `selectors`, then jQuery is used to try and find an element through CSS selectors (`$(root).find(selector).toArray()`). If you are using [standard shapes](#), you may disable this functionality for increased performance by setting `useCSSSelectors = false` in the global config.

dia.CellView.prototype.findPortNode

`cellView.findPortNode(portId)`

Return the port element of this CellView that is identified by `portId` (i.e. an SVGElement within `this.el` which has `'port': portId`, an SVGElement referenced by `portRoot` selector).

If the CellView does not have a port identified by `portId`, return `null`.

`cellView.findPortNode(portId, selector)`

If `selector` is also specified, return the subelement of the port element that is identified by `selector`, according to the rules of the `cellView.findBySelector()` [function](#).

If the port element does not have a subelement that matches `selector`, return `null`.

dia.CellView.prototype.highlight

cellView.highlight([el[, options]])

Add a highlighter to the cell view.

When the method is called, the `cell:highlight` [paper event](#) is triggered.

For greater flexibility use [highlighters](#) API instead.

Arguments:

- **el** - if not provided, then the cell view's `el` will be used
- **options**:
 - **highlighter**:
 - **name** - the name of the highlighter (see [highlighters](#))
 - **options** - an options object that will be passed directly to the highlighter specified by `name`
 - **type** - the [type of highlighting](#)

dia.CellView.prototype.isDefaultInteractionPrevented

cellView.isDefaultInteractionPrevented(event)

Returns whether [preventDefaultInteraction](#) was ever called on this `event` object.

dia.CellView.prototype.preventDefaultInteraction

cellView.preventDefaultInteraction(event)

The method tells the view that its default interactivity action should not be taken as it normally would be.

For example, if the view is a [joint.dia.ElementView](#) and the user clicks on it, the view will normally enter the [interactive mode](#) and starts moving the element.

Calling `preventDefaultInteraction` will prevent the view from entering the interactive mode. The view will still trigger all events, but it will not react to them in the default way.

It is useful when you want to handle the pointer events yourself. For example, you can use it to implement your own custom interaction with the view. A selection lasso can be drawn whenever the user holds down the shift key and starts dragging from any point on the paper (regardless of whether it is an element, a link or a blank space).

```
paper.on('element:pointerdown', (elementView, evt) => {
  if (evt.shiftKey) {
    // prevent element movement
    elementView.preventDefaultInteraction(evt);
    // start drawing selection lasso
  }
});
```

The following table shows which default interaction will be stopped depending on what point the method was called.

Paper <u>events</u>	Description	Paper <u>interactive option</u>
'element:pointerdown'	Prevents element movement.	<i>elementMove</i>
'element:magnet:pointerdown'	Prevents adding link from magnet and element movement.	<i>addLinkFromMagnet</i>
'link:pointerdown'	Prevents link label and link movement.	<i>labelMove</i> and <i>linkMove</i>

dia.CellView.prototype.unhighlight

cellView.[unhighlight](#)([el[, options]])

Removes a highlighter added to the cell view.

When the method is called, the `cell:unhighlight` [paper event](#) is triggered.

It is **important** to note that if you highlighted a cell using custom options, you must provide those exact same options when using the unhighlight method.

To avoid this and for greater flexibility use [highlighters](#) API instead.

dia.Element

The basic model for diagram elements. It inherits from [joint.dia.Cell](#) with a few additional properties and methods specific to elements. For a quick introduction to elements, see our tutorial.

Elements' properties may be split into several groups according to their function:

Geometry

- `position` - coordinates of the element, provided as an object with `x` and `y` keys. The property may be accessed or set directly using regular Backbone `set('position')` / `get('position')` methods or through Element's `translate()` method.
- `angle` - angle of rotation of the element in degrees. The rotation origin is always the center of the element. The property may be accessed or set directly using regular Backbone `set('angle')` / `get('angle')` methods or through Element's `rotate()` method.
- `size` - size of the element, provided as an object with `width` and `height` keys. The property may be accessed or set directly using regular Backbone `set('size')` / `get('size')` methods or through Element's `translate()` method.

Presentation

Each `joint.dia.Element` defines its own SVG markup which is then used by `joint.dia.ElementView` to render the element to the paper. For instance, the `joint.shapes.standard.Rectangle` element (which inherits from `joint.dia.Element`) defines its markup using the JSON array notation as follows:

```
markup: [{
  tagName: 'rect',
  selector: 'body',
}, {
  tagName: 'text',
  selector: 'label'
}]
```

As we can see, the `joint.shapes.standard.Rect` shape consists of two “subelements”: one `SVGRectElement` named `'body'` and one `SVGTextElement` named `'label'`. The `attrs` object refers to the subelements' names (“selectors”) to provide SVG attributes to these constituent `SVGElements`.

Styling

The keys of the `attrs` object are selectors that match subelements defined in the element's `markup`. The values of this object are SVG attributes that will be set on the selected subelements. One can find the full list of SVG attributes and their descriptions online, e.g. on MDN.

For example, in order to set a red fill color on a subelement called `'body'`, the `attrs` object would contain:

```
body: { fill: 'red' }
```

If you simply need to change a value of an attribute, it is not recommended to modify the `attrs` object directly. Instead, use the `element.attr()` method.

We can use the `joint.shapes.standard.Rectangle` element (which inherits from `joint.dia.Element`) as an example. The `attrs` object in its definition is provided below:

```
attrs: {
  body: {
    width: 'calc(w)',
```

```

    height: 'calc(h)',
    strokeWidth: 2,
    stroke: '#000000',
    fill: '#FFFFFF'
  },
  label: {
    textVerticalAnchor: 'middle',
    textAnchor: 'middle',
    x: 'calc(0.5*w)',
    y: 'calc(0.5*h)',
    fontSize: 14,
    fill: '#333333'
  }
}

```

Notice that the object makes use of special JointJS attributes (e.g. `textVerticalAnchor`) on top of standard SVG attributes (e.g. `stroke`, `fill`). All of these “special attributes” are listed in the [attributes](#) section of this documentation, along with information on how to use `calc()` for sizing. You should also refer to our tutorial on special attributes.

Attributes

Attributes defined in `markup` are evaluated once at CellView creation, while attributes defined in `attrs` are evaluated on every model change. As JointJS special attributes usually depend on the current state of the model, we should define them in `attrs`, along with any other SVG attributes that will be modified in the runtime of your application.

Z

The `z` property specifies the stack order of the element in the SVG DOM. An element with a higher `z` value will be rendered in front of an element with a lower `z` value. ([This also applies to Links.](#))

Ports

You may define and group ports on the Element with the `ports` attribute. Each can have its own markup and attrs, and its own label. Grouped ports may share properties and behaviors. See the [element ports documentation](#) for more information.

Nesting

The last two properties of elements are `embeds` and `parent`. These two are related to elements that contain or are contained within other elements, forming a hierarchical structure.

- `embeds` - a list of `id` 's of the Cells that are embedded inside this Element.
- `parent` - the `id` of the Element that is the parent of this element. When a parent element is translated, all its children get translated too.

Events

Elements trigger several special events, detailed in the [element events documentation](#).

Custom Element

It is possible to extend the `joint.dia.Element` class to create a custom element. A custom element may override `Element` properties to assign its own defaults. These values override builtin defaults, if provided, and are applied to all instances of the new `Element` type, unless an individual instance overrides them with its own values. The following `Element` properties are applicable in this context:

- `markup` - provide default element markup for all instances of this `Element` type, specified with a [JSON array](#).
- `attrs` - provide default element styling for all instances of this `Element` type. These allow you to change the style and size of SVG elements, identified by their selectors.

Creating custom elements is explained in more detail in our tutorial.

dia.Element.events

The following list contains events that you can react on:

- `change` - generic event triggered for any change on the element - `fn(element, opt)`
- `change:position` - triggered when the element changes its position - `fn(element, newPosition, opt)`
- `change:angle` - triggered when the element gets rotated - `fn(element, newAngle, opt)`
- `change:size` - triggered when the element gets resized - `fn(element, newSize, opt)`
- `change:attrs` - triggered when the element changes its attributes - `fn(element, newAttrs, opt)`
- `change:embeds` - triggered when other cells were embedded into the element - `fn(element, newEmbeds, opt)`
- `change:parent` - triggered when the element got embedded into another element - `fn(element, newParent, opt)`
- `change:z` - triggered when the element is moved in the z-level ([toFront](#) and [toBack](#)) - `fn(element, newZ, opt)`
- `transition:start` - triggered when a transition starts. - `fn(element, pathToAttribute)`
- `transition:end` - triggered when a transition ends. - `fn(element, pathToAttribute)`

```
// Listening for changes of the position to a single element
element1.on('change:position', function(element1, position) {
  alert('element1 moved to ' + position.x + ',' + position.y);
});
// All elements events are also propagated to the graph.
graph.on('change:position', function(element, position) {
  console.log('Element ' + element.id + 'moved to ' + position.x + ',' + position.y);
});
// Using the option parameter and a custom attribute
graph.on('change:custom', function(element, custom, opt) {
  if (opt.consoleOutput) {
    console.log('Custom attribute value changed to "' + custom + '"');
  }
}
```



```
});  
element2.prop('custom', 'myValue', { consoleOutput: true });
```

dia.Element.ports

Ports

Many diagramming applications deal with elements with ports. Ports are usually displayed as circles inside diagram elements and are used not only as "sticky" points for connected links, but they also further structure the linking information. It is common that certain elements have lists of input and output ports. A link might not then point to the element as a whole, but to a certain port instead.

It's easy to add ports to arbitrary shapes in JointJS. This can be done either by passing a ports definition as an `option` in the constructor, or using the ports API to get/add/remove single or multiple ports. For more information on how to define ports please see the [Port configuration](#) section.

Port API on `joint.dia.Element`

- [`hasPort`](#) / [`hasPorts`](#)
- [`addPort`](#) / [`addPorts`](#)
- [`removePort`](#) / [`removePorts`](#)
- [`getPort`](#) / [`getPorts`](#)
- [`portProp`](#)
- [`getPortsPositions`](#)

Port configuration

```
// Single port definition  
const port = {  
  // id: 'abc', // Generated if `id` value is not present  
  group: 'a',  
  args: {}, // Extra arguments for the port layout function, see `layout.Port`  
  section  
  label: {  
    position: {  
      name: 'left',  
      args: { y: 6 } // Extra arguments for the label layout function, see  
        `layout.PortLabel` section  
    },  
    markup: [{  
      tagName: 'text',  
      selector: 'label'  
    }]  
  }  
}
```

```

    },
    attrs: {
        body: { magnet: true, width: 16, height: 16, x: -8, y: -4, stroke: 'red',
fill: 'gray'},
        label: { text: 'port', fill: 'blue' }
    },
    markup: [{
        tagName: 'rect',
        selector: 'body'
    }]
};

// a.) Add ports in constructor.
const rect = new joint.shapes.standard.Rectangle({
    position: { x: 50, y: 50 },
    size: { width: 90, height: 90 },
    ports: {
        groups: {
            'a': {}
        },
        items: [
            { group: 'a' },
            port
        ]
    }
});

// b.) Or add a single port using API
rect.addPort(port);

rect.getGroupPorts('a');
/*
[
    { * Default port settings * },
    { * Follows port definition * },
    { * Follows port definition * }
]
*/

```

id	<i>string</i>	It is automatically generated if no <code>id</code> is provided. IDs must be unique in the context of a single shape - two ports with the same port id are therefore not allowed (<code>Element: found id duplicities in ports.</code> error is thrown).
group	<i>string</i>	Group name, more info in the groups section.

args	<i>object</i>	Arguments for the port layout function. Available properties depend on the type of layout. More information can be found in layout.Port .
attrs	<i>object</i>	JointJS style attribute definition. The same notation as the <code>attrs</code> property on Element .
markup	<i>MarkupJSON string</i>	A custom port markup. The default port markup is <code><circle class="joint-port-body" r="10" fill="#FFFFFF" stroke="#000000"/></code>. The root of the port markup is referenced by the <code>portRoot</code> selector. If the markup contains more than one node, an extra group is created to wrap the nodes. This group becomes the new <code>portRoot</code>. <pre>// An example of port markup markup: [{ tagName: 'rect', selector: 'bodyInner', className: 'outer', attributes: { 'width': 15, 'height': 15, 'fill': 'red' } }, { tagName: 'rect', selector: 'bodyOuter', className: 'inner', attributes: { 'width': 15, 'height': 15, 'fill': 'blue', 'x': 10 } }]</pre>
label	<i>object</i>	Port label layout configuration. E.g. label position, label markup. More information about port label layouts can be found in layout.PortLabel .

• label.position	<i>string object</i>	Port label position configuration. It can be a <code>string</code> for setting the port layout type directly with default settings, or an <code>object</code> where it's possible to set the layout type and options. <pre> { position: 'left'} // or ... { position: { name: 'left', args: { dx: 10 } } } </pre>
◦ label.position.name	<i>string</i>	It states the layout type. It matches the layout method name defined in the <code>joint.layout.PortLabel</code> namespace: <code>name: 'left'</code> is implemented as <code>joint.layout.PortLabel.left</code>.
◦ label.position.args	<i>object</i>	Additional arguments for the layout method. Available properties depend on the layout type. More information can be found in the layout.PortLabel section.
• label.markup	<i>MarkupJSON string</i>	A custom port label markup. The default port label markup is <code><text class="joint-port-label" fill="#000000"/></code>. The root of the label markup is referenced by the <code>labelRoot</code> selector. If the markup contains more than one node, an extra group is created to wrap the nodes. This group becomes the new <code>labelRoot</code>. All <code><text/></code> nodes of the port are referenced by the <code>labelText</code> selector, unless the markup contains <code>labelText</code> explicitly. Use an empty array <code>[]</code> to prevent the label from being rendered.
z	<i>number string</i>	An alternative to HTML <code>z-index</code>. <code>z</code> sets the position of a port in the list of DOM elements within an <code>ElementView</code>.

Shapes most likely consist of 1 or more DOM elements, `<rect/>`, `<rect/><text/><circle/>` etc. Ports are placed into the main group element `elementView.el`, so it will act as the port container. Ports with `z: 'auto'` are located right after the last element in the main group. Ports with `z` defined as a number are placed before a DOM element at the position (index within the children of the container, where only the original markup elements, and ports with `z: 'auto'` are taken into account) equal to `z`.

For instance, the first shape from the demo above with the following markup...

```
markup: [{
  tagName: 'rect',
  selector: 'bodyMain',
  className: 'bodyMain'
}, {
  tagName: 'rect',
  selector: 'bodyInner',
  className: 'bodyInner'
}, {
  tagName: 'text',
  selector: 'label',
  className: 'label'
}]
```

...will be rendered like this:

```
<g model-id="...">
  <g class="joint-port"></g>      <!--
z: 0 -->
  <rect class="bodyMain"></rect>
  <g class="joint-port"></g>      <!--
z: 1 -->
  <rect class="bodyInner"></rect>
  <text class="label"></text>
  <g class="joint-port"></g>      <!--
z: 3 -->
  <g class="joint-port"></g>      <!--
z: auto -->
</g>
```

Ports will be placed in the `rotatable` group if it's defined in the shape's markup. Ports with `z: 'auto'` are located right after the last element in the `rotatable` group. In the demo

above, the second shape is defined with a `rotatable` group and the following markup:

```
markup: [{
  tagName: 'g',
  selector: 'rotatable',
  className: 'rotatable',
  children: [{
    tagName: 'g',
    selector: 'scalable',
    className: 'scalable',
    children: [{
      tagName: 'rect',
      selector: 'bodyMain',
      className: 'bodyMain'
    }]
  }, {
    tagName: 'rect',
    selector: 'bodyInner',
    className: 'bodyInner'
  }, {
    tagName: 'text',
    selector: 'label',
    className: 'label'
  }]
}]
```

It will be rendered like this:

```
<g model-id="...">
  <g class="rotatable">
    <g class="joint-port"></g>
<!-- z: 0 -->
    <g class="scalable"><rect
class="bodyMain"></rect></g>
    <g class="joint-port"></g>
<!-- z: 1 -->
    <rect class="bodyInner"></rect>
    <text class="label"></text>
    <g class="joint-port"></g>
<!-- z: 3 -->
    <g class="joint-port"></g>
<!-- z: auto -->
  </g>
</g>
```

All properties described above are optional, and everything has its own default. E.g. `element.addPorts([{}], {}]` will add 2 ports with default settings.

Port groups configuration

While single port definitions are useful, what if we want more control over our ports? This is where a port group can come into play. A group allows us to define multiple ports with similar properties, and influence the default port alignment. Any individual port can override a property in a port group definition except the type of layout (E.g. `position: 'left'`). The group definition defines the layout, and the individual port `args` are the only way a port can affect it.

```
// Port definition for input ports group
const portsIn = {
  position: {
    name: 'left', // Layout name
    args: {}, // Arguments for port layout function, properties depend on type of
layout
  },
  label: {
    position: {
      name: 'left',
      args: { y: 6 }
    },
    markup: [{
      tagName: 'text',
      selector: 'label',
    }]
  },
  attrs: {
    body: { magnet: 'passive', width: 15, height: 15, stroke: 'red', x: -8, y: -8
},
    label: { text: 'in1', fill: 'black' }
  },
  markup: [{
    tagName: 'rect',
    selector: 'body'
  }]
};

// Define port groups in element constructor
const rect = new joint.shapes.basic.Rect({
  // ...
  ports: {
    groups: {
      'group1': portsIn,
      // 'group2': ...,
      // 'group3': ...,
    },
  },
});
```

```

    items: [
      // Initialize 'rect' with port in group 'group1'
      {
        group: 'group1',
        args: { y: 40 } // Overrides `args` from the group level definition
      }
    ]
  }
});

// Add another port using Port API
rect.addPort(
  { group: 'group1', attrs: { label: { text: 'in2' }}}
);

```

position	<i>string object</i>	Port position configuration. It can be a <code>string</code> to set the port layout type directly with default settings, or an <code>object</code> where it's possible to set the layout type and options.
• position.name	<i>string</i>	It states the layout type. Match the layout method name defined in the <code>joint.layout.Port</code> namespace: name: <code>'left'</code> is implemented as <code>joint.layout.Port.left</code> .
• position.args	<i>object</i>	Arguments for the port layout function. Available properties depend on the type of layout. More information can be found in Layout.Port .
attrs	<i>object</i>	JointJS style attribute definition. The same notation as the <code>attrs</code> property on Element .
markup	<i>MarkupJSON string</i>	Custom port markup. Multiple roots are not allowed. Valid notation would be: <pre> markup: [{ tagName: 'g', children: [{ tagName: 'rect', selector: 'bodyInner', className: 'outer', attributes: { 'width': 15, 'height': 15, 'fill': 'red' } }] }] </pre>

		<pre> }, { tagName: 'rect', selector: 'bodyOuter', className: 'inner', attributes: { 'width': 15, 'height': 15, 'fill': 'blue', 'x': 10 } }] }] </pre> The default port looks like the following: <code><circle class="joint-port-body" r="10" fill="#FFFFFF" stroke="#000000"/></code>.
label	<i>object</i>	Port label layout configuration. E.g. label position, label markup. More information about port label layouts can be found in the layout.PortLabel section.
• label.position	<i>string object</i>	Port label position configuration. It can be a <code>string</code> for setting the port layout type directly with default settings, or an <code>object</code> where it's possible to set the layout type and options. <pre> { position: 'left' } // or ... { position: { name: 'left', args: { dx: 10 } } } </pre>
◦ label.position.name	<i>string</i>	It states the layout type. It matches the layout method name defined in the <code>joint.layout.PortLabel</code> namespace: <code>name: 'left'</code> is implemented as <code>joint.layout.PortLabel.left</code>.

◦ <code>label.position.args</code>	<i>object</i>	Additional arguments for the layout method. Available properties depend on the layout type. More information can be found in the layout.PortLabel section.
• <code>label.markup</code>	<i>MarkupJSON string</i>	Custom port label markup. Multiple roots are not allowed. The default port label looks like the following: <code><text class="joint-port-label" fill="#000000"/></code> .

Custom markup

Both port and port label can have custom markup...

```
var rect = new joint.shapes.basic.Rect({
  // ...
});

rect.addPort({
  markup: [{
    tagName: 'rect', selector: 'body', attributes: { 'width': 20, 'height': 20,
    'fill': 'blue' }
  }]
});

rect.addPort({
  markup: [{
    tagName: 'rect', selector: 'body', attributes: { 'width': 16, 'height': 16,
    'fill': 'red' }
  }],
  label: {
    markup: [{ tagName: 'text', selector: 'label', attributes: { 'fill': '#000000'
    }}}
  },
  attrs: { label: { text: 'port' }}
});
```

...or, it can be set as an default port markup/port label markup on an element model:

```
var rect = new joint.shapes.basic.Rect({
  portMarkup: [{
    tagName: 'rect',
    selector: 'body',
    attributes: {
      'width': 20,
      'height': 20,
      'fill': 'blue',
```

```

        'stroke': 'black',
        'stroke-width': 5 // Use native SVG kebab-case for attributes in markup
    }
  ]],
  portLabelMarkup: [{
    tagName: 'text',
    selector: 'label',
    attributes: {
      'fill': 'yellow'
    }
  }]
  // ...
});

```

dia.Element.prototype.addPort

element.[addPort](#)(port, [opt])

Add a single port, where `port` could be defined as described in section [Port interface](#)

dia.Element.prototype.addPorts

element.[addPorts](#)(ports, [opt])

Add array of `ports`. Ports are validated for `id` duplicities, if there is a id collision, no new ports are added, where `port` could be defined as describe in section [Port interface](#)

dia.Element.prototype.addTo

element.[addTo](#)(graph)

Add the element to the `graph` (an instance of `joint.dia.Graph`). This is equivalent to calling `graph.addCell(element)`.

dia.Element.prototype.angle

element.[angle](#)()

Return the rotation of the element in degrees (0° - 360°).

dia.Element.prototype.attr

`element.attr(attrs [, opt])`

Set presentation attributes ([SVG](#) and [JointJS](#) attributes) on view subelements. `attrs` can either be an object or string representing a path to a nested attribute. If it is an object, the keys of the `attrs` object are selectors ([JSON Markup Selector](#) or CSS Selector) matching the subelements. The values are objects containing SVG attributes and their values. `attrs` object will be mixed with `attrs` property of the `element` model. This is a convenient way of rewriting only some of the attributes of the subelements. For overwriting all attributes of all subelements, use `element.set('attrs', attrs)`.

```
element.attr({
  // selectors as defined in the JSON markup
  body: { width: 'calc(w)', height: 'calc(h)' },
  label: { text: 'My Label' },
  // using CSS selectors is significantly slower
  rect: { fill: 'blue' },
  text: { fill: 'white', fontSize: 15 },
  '.myrect2': { fill: 'red' }
});
```

The method is not suitable for storing custom data on the model. For that, use [prop\(\)](#) or [set\(\)](#) methods.

```
element.set('confirmed', true);
element.prop('data/count', 10);
```

`element.attr(path, value [, opt])`

An alternative call using a string/array path and a value:

```
element.attr('body/fill', 'red');
element.attr(['label', 'fontSize'], 12);
// Note: an equivalent expression is also
element.prop('attrs/label/fontSize', 12);
```

`element.attr([path])`

Get attribute value defined by a path. If no path provided the whole `attrs` object is returned.

```
element.attr('body/fill') === 'red';
```

```
element.attr(['label', 'fontSize']) === 12;
```

dia.Element.prototype.clone

element.clone(options)

Returns a new instance of the element with identical attributes. If `options.deep === true`, then all the embedded cells (elements, links) of the element are cloned as well. In this case, the return value is an array of instances rather than a single instance.

dia.Element.prototype.embed

element.embed(cell, [opt])

element.embed(cells, [opt])

Embed a cell (element or link), or an array of cells into the element. The element then becomes a parent of the embedded cell. When a parent is moved (translated), all cells embedded into that parent will move as well. If links are embedded, their vertices move with the parent. This way both options are available: if a link is not embedded but its source/target elements are and their parent moves, the embedded elements move with the parent but the link vertices stay at the same position. If the link is embedded with its source/target elements, its vertices move as the parent moves.

Any additional option or custom property provided in the options object will be accessible in the callback function of the Element `'change:embeds'` [event](#).

```
const rect1 = new joint.shapes.standard.Rectangle({
  position: { x: 100, y: 100 },
  size: { width: 90, height: 30 },
  attrs: { label: { text: 'Rect' } }
});

const rect2 = rect1.clone();
rect2.translate(100, 0);

graph.addCell([rect1, rect2]);

rect1.on('change:embeds', function(element, newEmbeds, opt) {
  console.log(opt); // {testOption: true}
});

// Add custom 'testOption' property
rect1.embed(rect2, { testOption: true });
```

dia.Element.prototype.findView

`element.findView(paper)`

Find view (`joint.dia.ElementView`) for the element model in the `paper` . This is a shortcut to the equivalent call `paper.findViewByModel(element)`

dia.Element.prototype.fitEmbeds

`element.fitEmbeds([opt])`

An alias for the `element.fitToChildren` [function](#).

dia.Element.prototype.fitParent

`element.fitParent([opt])`

Resize and reposition this element's embedding parent element such that all of its children (including this element) end up within the parent's new bounding box.

Starting from a given element, this function proceeds “upwards” to that element's parent (and further ancestors, if `opt.deep` is used). In that sense, this function is the opposite of the `element.fitToChildren` [function](#).

Available options:

padding	<i>number</i>	Inflate the embedding parent element's calculated bounding box by this much additional padding.
expandOnly	<i>boolean</i>	If true , the algorithm is only ever allowed to expand the bounding box of the embedding parent element, never to shrink it. You can visualize this setting as the algorithm dragging the top-left and bottom-right corners of the parent's bounding box outward until its children (including this element) are within the bounding box.
shrinkOnly	<i>boolean</i>	If true , the algorithm is only ever allowed to shrink the bounding box of the embedding parent element, never to expand it. You can visualize this setting as the algorithm dragging the top-left and bottom-right corners of the parent's bounding box inward until only its children (including this element) are within the bounding box. If only a portion of this element (or this element's sibling element) initially overlaps the embedding parent element, the parent's calculated bounding box will only include that portion. If there is no overlap between this element and its parent (i.e. this element is

		“outside” the parent), then this function does nothing (since satisfying the shrink-only restriction is impossible).
deep	<i>boolean</i>	If true, this algorithm is applied recursively on the embedding parent element of this element's embedding parent element, and so on. The bounding box algorithm is evaluated in reverse-depth order - starting from this element, then going up (i.e. to the topmost embedding ancestor element - or <code>opt.terminator</code>), to make sure that any paddings applied on lower-level elements are taken into account by higher-level elements. Note that if this option is used in conjunction with <code>opt.shrinkOnly</code>, the algorithm is still applied strictly recursively (i.e. one level at a time). Therefore, even if this element lies completely within the current bounding box of its grandfather element, the grandfather element's calculated bounding box will shrink only based on this element's parent - which may mean that this element will end up outside of the calculated bounding box of the grandfather element.
terminator	<i>Cell Cell.ID</i>	If <code>opt.deep</code> is true and a Cell reference or a Cell ID is provided as a value of this option, then the specified element is the last one for which the bounding box algorithm is applied during recursion - it is the last element whose bounding box is resized and repositioned based on the bounding boxes of its children. Handling of edge cases: • If the provided value refers to this element, then nothing happens. • If the provided value refers to this element's embedding parent element, then the result is the same as calling the function without <code>opt.deep</code> and <code>opt.terminator</code>. • If the provided value does not refer to an ancestor of this element, then the result is the same as calling the function without <code>opt.terminator</code>. • If the provided value does not refer to a Cell of the Element type, then the result is the same as calling the function without <code>opt.terminator</code>.

dia.Element.prototype.fitToChildren

`element.fitToChildren([opt])`

Resize and reposition this element such that all of its embedded child elements end up within this element's new bounding box.

Starting from a given element, this function proceeds “downwards” through that element's children (and further descendants, if `opt.deep` is used). In that sense, this function is the opposite of the `element.fitParent` [function](#).

Available options:

padding	<i>number</i>	Inflate this element's calculated bounding box by this much additional padding.
expandOnly	<i>boolean</i>	If true , the algorithm is only ever allowed to expand the bounding box of this element, never to shrink it. You can visualize this setting as the algorithm dragging the top-left and bottom-right corners of this element's bounding box outward until all its embedded child elements are within the bounding box.
shrinkOnly	<i>boolean</i>	If true , the algorithm is only ever allowed to shrink the bounding box of this element, never to expand it. You can visualize this setting as the algorithm dragging the top-left and bottom-right corners of this element's bounding box inward until only its embedded child elements are within the bounding box. If only a portion of an embedded child element initially overlaps this element, the calculated bounding box will only include that portion. If there is no overlap between this element and its children (i.e. all children are currently placed “outside” of this element), then this function does nothing (since satisfying the shrink-only restriction is impossible).
deep	<i>boolean</i>	If true , this algorithm is applied recursively on all embedded children of this element which have embedded children of their own, and so on. The bounding box algorithm is evaluated in reverse-depth order - starting from the deepest descendant, then going up (i.e. to this element). This ensures that any paddings applied on lower-level elements are taken into account by higher-level elements. Note that if this option is used in conjunction with <code>opt.shrinkOnly</code> , the algorithm is still applied strictly recursively - one level at a time. Therefore, even if this element's current bounding box completely overlaps one of its grandchild elements, this element's calculated bounding box will shrink only based on the grandchild's parent (this element's child), which may mean that the grandchild element will end up outside of the calculated bounding box of this element.

dia.Element.prototype.getAbsolutePointFromRelative

`element.getAbsolutePointFromRelative(x, y)`

`element.getAbsolutePointFromRelative(relativePoint)`

Accept a point in the coordinate system of the element and return the point in the graph coordinate system, represented as a `g.Point` .

The element coordinate system has its origin `[0,0]` at the `element.position()` rotated by the `element.angle()` . Each axis `x` and `y` are rotated by the same angle.

dia.Element.prototype.getAncestors

`element.getAncestors()`

Return an array of all the ancestors of this element starting from the immediate parent all the way up to the most distant ancestor.

dia.Element.prototype.getBBox

`element.getBBox([opt])`

Return the bounding box of the element model, represented as a `g.Rect`.

Keep in mind that this function reports the dimensions of the element's model, not the dimensions of the element's view. (For example, the model bounding box does not include any associated ports). See the documentation of the `elementView.getBBox` [function](#) for details about the differences.

```
if (element1.getBBox().intersect(element2.getBBox())) {  
    // intersection of the two elements  
}
```

opt	description
deep	Return the union of the element's bounding box and all the elements embedded into it.
rotate	Return the bounding box after the element's rotation.

dia.Element.prototype.getEmbeddedCells

`element.getEmbeddedCells([opt])`

Return an array of all the embedded cells of an element. If all you need is `id`'s of all the embedded cells, use `element.get('embeds')`. If `opt.deep` is `true`, all the deeply embedded cells will be returned. The order in which the cells are returned depends on the search algorithm used. By default, [Depth-first search \(DFS\)](#) algorithm is used. If `opt.breadthFirst` is `true`, the [Breadth-first search](#) algorithm will be used instead.

dia.Element.prototype.getGroupPorts

`element.getGroupPorts(groupName)`

Returns an array of all ports on the element with the given `groupName` . If there is no such a port an empty array is returned.

dia.Element.prototype.getPort

`element.getPort(id)`

Returns the port specified by an `id` . If such a port does not exist the method returns `undefined` .

dia.Element.prototype.getPortIndex

`element.getPortIndex(portId)`

Returns the port index in the array of port items.

dia.Element.prototype.getPorts

`element.getPorts()`

Returns an array of all ports on the element. If there is no port an empty array is returned.

dia.Element.prototype.getPortsPositions

`element.getPortsPositions(groupName)`

Returns the positions and the angle of all ports in the group, relatively to the element position.

dia.Element.prototype.getRelativePointFromAbsolute

`element.getRelativePointFromAbsolute(x, y)`

`element.getRelativePointFromAbsolute(absolutePoint)`

Accept a point in the graph coordinate system and return the point in the coordinate system of the element, represented as a `g.Point` .

The element coordinate system has its origin `[0,0]` at the `element.position()` rotated by the `element.angle()`. Each axis `x` and `y` are rotated by the same angle.

dia.Element.prototype.getTransitions

`element.getTransitions()`

Return an array of all active transitions (their paths).

dia.Element.prototype.hasPort

`element.hasPort(id)`

Check if an element contains a port with `id`. Returns `boolean`.

dia.Element.prototype.hasPorts

`element.hasPorts()`

Check if an element has any ports defined. Returns `boolean`.

dia.Element.prototype.insertPort

`element.insertPort(before, port, [opt])`

Insert a new port before another port, where `port` could be defined as described in section [Port interface](#) and `before` is either an index, port id or port itself.

dia.Element.prototype.isElement

`element.isElement()`

Always returns `true`. The reason the method is here is that both `joint.dia.Element` and `joint.dia.Link` inherit from `joint.dia.Cell`. This method is useful if you don't know what the cell is. Calling `cell.isElement()` is equivalent to `cell instanceof joint.dia.Element`. Example:

```
var cell = graph.getCell(myId)
if (cell.isElement()) {
```

```

    // Do something if the cell is an element.
}

```

dia.Element.prototype.isEmbeddedIn

`element.isEmbeddedIn(element [, opt])`

Return `true` if the element is embedded in another element `element`. If `opt.deep` is `false`, only direct parentage will be checked. `opt.deep` is `true` by default.

dia.Element.prototype.isLink

`element.isLink()`

Always returns `false`. The reason the method is here is that both `joint.dia.Element` and `joint.dia.Link` inherit from `joint.dia.Cell`. This method is useful if you don't know what the cell is. Calling `cell.isLink()` is equivalent to `cell instanceof joint.dia.Link`. Example:

```

var cell = graph.getCell(myId)
if (cell.isLink()) {
    // Do something if the cell is a link.
}

```

dia.Element.prototype.portProp

`element.portProp(portId, path, [value])`

Set properties, possibly nested, on the element port. This is an equivalent of the `attr()` method but this time for custom data properties.

```

element.portProp('port-id', 'attrs/circle/fill', 'red');
element.portProp('port-id', 'attrs/circle/fill'); // 'red'

element.portProp('port-id', 'attrs/circle', { r: 10, stroke: 'green' });
element.portProp('port-id', 'attrs/circle'); // { r: 10, stroke: 'green', fill: 'red' }
}

```

dia.Element.prototype.position

`element.position([opt])`

If `position()` is called without arguments, it returns the current position.

Option	Default	Description
parentRelative	false	The method returns the current position of the element relative to its parent.

`element.position(x, y, [opt])`

Set the element position to `x` and `y` coordinates. This is almost equivalent to `element.set('position', { x, y }, opt)`. However, this method provides some additional functionality.

Option	Default	Description
parentRelative	false	If set to <code>true</code> the <code>x</code> and <code>y</code> coordinates will be treated relatively to the parent element of this element. If the element has no parent or the parent is a link, the option is ignored.
deep	false	If set to <code>true</code> it will position the element and its descendants while keeping the original distances of the descendants to/from the element origin. The relative distances of the descendants are maintained.
restrictedArea	null	If set to a rectangle, the translation of the element will be restricted to that rectangle only. The <code>restrictedArea</code> is an object of the form <code>{ x: Number, y: Number, width: Number, height: Number }</code>. This is useful, e.g. if you want to restrict the translation of an embedded element within its parent. The only thing you have to do in this case is to pass the bounding box of the parent element to the <code>restrictedArea</code> option: <pre>myElement.position(x, y, { restrictedArea: myElement.getParentCell().getBBBox() })</pre> The code above makes sure that the element <code>myElement</code> never crosses the bounding box of its parent element. Note that this also works if the element <code>myElement</code> has other embedded elements. In other words, the bounding box of the <code>myElement</code> that is used to calculate the restriction is the total bounding box, including all its children (in case they are “sticking out”).

```

el1.position(100, 100);
el1.embed(el2);
el2.position(10, 10, { parentRelative: true });
el2.position() // --> 110@110
el1.position(200,200, { deep: true });
el2.position() // --> 210@210

```

```

const restrictedArea = paper.getArea();
el.position(x, y, { restrictedArea }); // --> makes sure x and y is within the area
provided

```

dia.Element.prototype.prop

`element.prop(properties)`

Set properties, possibly nested, on the element model. This is an equivalent of the `attr()` method but this time for custom data properties.

```

element.prop('name/first', 'John')
element.prop('name/first') // 'John'
element.prop({ name: { first: 'John' } })
// Nested arrays are supported too:
element.prop('mylist/0/data/0/value', 50)
element.prop({ mylist: [ { data: [ { value: 50 } ] } ] })

```

To overwrite attributes, enable *rewrite* mode by adding `{ rewrite: true }` as the 3rd argument. This differs from the default behaviour which is to merge our properties.

```

element.prop('custom/state/isCollapsed', true);
element.prop('custom/state', { isActive: false }, { rewrite: true });

// Output from element.toJSON();
// We can see our attributes have been overwritten
{
  "type": "standard.Rectangle",
  "position": { "x": 0, "y": 0 },
  "size": { "width": 1, "height": 1 },
  "angle": 0,
  "id": "b1c02090-e46a-4d90-a5dc-5096f1559b9f",
  "custom": {
    "state": {
      "isActive": false
    }
  }
}

```

```

    },
    "attrs": {}
  }
}

```

When changing model attributes via `prop()` or `attr()`, some useful information is passed along with the change event in JointJS. `propertyPath`, `propertyValue`, and `propertyPathArray` are all values which can be accessed when updating the model. This can prove useful if for some reason you need to listen to a specific attribute change.

```

graph.on('change', (cell, opt) => {
  console.log(opt);
  // --> {propertyPath: 'attrs/body/fill', propertyValue: 'cornflowerblue',
  propertyPathArray: Array(3)}

  if ('attrs' in cell.changed) {
    console.log(opt.propertyPathArray, 'was changed');
    // --> ['attrs', 'body', 'fill'] 'was changed'
  }
});

element.prop('attrs/body/fill', 'cornflowerblue');

```

Advanced: Pass `{ isolate: true }` if the property change does not affect the connected links. Typically, changing the element fill color has zero effect on attached links. By default, the element and all connected links are updated.

```

element.prop(['attrs', 'body', 'fill'], 'red', { isolate: true });

```

dia.Element.prototype.remove

`element.remove(options)`

Remove the element from the graph. All its embedded elements will get removed too and the element gets unembedded from its parent element. By default, all the associated links are removed too. To suppress this behaviour, set `options.disconnectLinks === true`. In this case, all the associated links get disconnected from this element rather than removed completely from the graph.

dia.Element.prototype.removeAttr

`element.removeAttr(path, [options])`

Remove a previously set attribute from the element. `path` can either be a string that specifies the path to the, possibly nested, attribute to be removed or an array of more paths. The associated element view makes sure the element gets re-

rendered properly. If `options` is passed, it can contain data that is passed over to the event listeners for the `change:attrs` event triggered on the element itself and also on the graph the element is in.

dia.Element.prototype.removePort

```
element.removePort(port, [opt])
```

Remove a port from an element, where `port` is a port object, or `portId`.

dia.Element.prototype.removePorts

```
element.removePorts(ports [, opt])
```

Remove an array of `ports` from the element.

The ports can be specified as Port interfaces or `portIds`. The function skips over any ports in the array that do not exist on the element.

```
element.removePorts([opt])
```

If no array is provided, the function removes all ports from the element.

dia.Element.prototype.resize

```
element.resize(width, height [, opt])
```

Resize an element in place so that the "scalable" group has width `width` and height `height`. In place in this case means that the top-left corner of the element stays at the same position after resizing. In other words, the element is stretched to the bottom/right (by default). To change the direction of resizing, set `opt.direction` to one of `'left'`, `'right'`, `'top'`, `'bottom'`, `'top-right'`, `'top-left'`, `'bottom-left'` or `'bottom-right'` (the default).

There is a difference between a classical scale and resize operations. `resize` doesn't actually scale the whole SVG `<g>` element grouping all its subelements. It only scales subelements of the `<g class="scalable">` group. This is very useful and brings a lot of flexibility in defining which subelements should be scaled and which not. Imagine a simple rectangle element with text inside. Usually, when we resize the whole element, we expect the rectangle to get scaled while the text should stay the same size, only its position should be adjusted so that the text stays in the center of the rectangle. This can be easily achieved by adding the `<rect/>` element to the `<g class="scalable">` group in the markup and positioning the text subelement relatively to the `<rect />` element: `<text ref-x=".5" ref-y=".5" ref="rect" />`. Note that neither of `ref-x`, `ref-y` and `ref` attributes is an SVG standard attribute. These are special attributes introduced by JointJS. More on these in the section on Special attributes.

dia.Element.prototype.rotate

`element.rotate(deg, [absolute, origin, opt])`

Rotate an element by `deg` degrees around its center. If the optional `absolute` parameter is `true`, the `deg` will be considered an absolute angle, not an addition to the previous angle. If `origin` is passed in the form of an object with `x` and `y` properties, then this point will be used as the origin for the rotation transformation.

dia.Element.prototype.scale

`element.scale(sx, sy, origin[, opt])`

Scales the element's position and size relative to the given origin.

dia.Element.prototype.stopTransitions

`element.stopTransitions([path])`

Stops all running transitions. If parameter `path` is provided, it will stop only transitions specified by this path.

dia.Element.prototype.toBack

`element.toBack([opt])`

Move the element so it is behind all other cells (elements/links). If `opt.deep` is `true`, all the embedded cells of this element will be updated in a Breadth-first search (BFS) fashion.

If `opt.foregroundEmbeds` is `true` (as by default), all the embedded cells will get a higher `z` index than that of this element. This is especially useful in hierarchical diagrams where if you want to send an element to the back, you don't want its children (embedded cells) to be hidden behind that element. If `opt.foregroundEmbeds` is `false`, the original order within the group is preserved, allowing children to remain behind their parents.

Set `opt.breadthFirst` to `false` to index the elements using Depth-first search (DFS).

dia.Element.prototype.toFront

`element.toFront([opt])`

Move the element so it is on top of all other cells (element/links). If `opt.deep` is `true`, all the embedded cells of this element will be updated in a Breadth-first search (BFS) fashion.

If `opt.foregroundEmbeds` is `true` (as by default), all the embedded cells will get a higher `z` index than that of this element. This is especially useful in hierarchical diagrams where if you want to send an element to the front, you don't want its children (embedded cells) to be hidden behind that element. If `opt.foregroundEmbeds` is `false`, the original order within the group is preserved, allowing children to remain behind their parents.

Set `opt.breadthFirst` to `false` to index the elements using Depth-first search (DFS).

All elements have a `z` property defining their z-level in the graph. This `z` property can even be set directly by `element.set('z', 123)`. This change will be automatically handled by the `joint.dia.Paper` object associated with the `joint.dia.Graph` object this element is part of and all the SVG elements will get resorted so that their position in the DOM reflects the `z` level.

```
container.embed(el1);
container.embed(el2);
container.toFront({ deep: true });
```

dia.Element.prototype.toJSON

`element.toJSON()`

Return a copy of the element's attributes for JSON serialization. This can be used for persistence or serialization. Note that this method doesn't return a JSON string but rather an object that can be then serialized to JSON with `JSON.stringify()`.

dia.Element.prototype.transition

`element.transition(path, value [, opt])`

Allows to change the element's property gradually over a period of time. This method lets you specify what property to change (`path`), when the transition will start (`options.delay`), how long the transition will last (`options.duration`), how the transition will run (`options.timingFunction`), and how to interpolate the property value (`options.valueFunction`).

```
element.transition('position/x', 250, {
  delay: 100,
  duration: 500,
  timingFunction: function(t) { return t*t; },
  valueFunction: function(a, b) { return function(t) { return a + (b - a) * t; } }
});
// will start changing the element's x-coordinate in 100ms, for period of 500ms.
```

JointJS comes pre-built with some common timing and interpolating functions. The timing functions are defined in the `joint.util.timing` namespace and the interpolating functions in the `joint.util.interpolate` namespace. The predefined timing functions are:

- `linear`
- `quad`
- `cubic`
- `inout`
- `exponential`
- `bounce`

and the predefined interpolating functions are:

- `number`
- `object`
- `hexColor`
- `unit`

```
element.transition('attrs/text/font-size', '1em', {
  valueFunction: joint.util.interpolate.unit,
  timingFunction: joint.util.timing.bounce
});
// will start changing the current font size value to 1em in the bounce fashion.
```

dia.Element.prototype.translate

`element.translate(tx, [ty], [opt])`

Translate an element by `tx` pixels in x axis and `ty` pixels in y axis. `ty` is optional in which case the translation in y axis will be considered zero. The optional options object `opt` can be used to pass additional parameters to the event handlers listening on `'change:position'` events. `opt.transition` can be used to initiate an animated transition rather than a sudden move of the element. See [joint.dia.Element:transition](#) for more info on transitions. If `opt.restrictedArea` is set, the translation of the element will be restricted to that area only. The `restrictedArea` is an object of the form `{ x: Number, y: Number, width: Number, height: Number }`. This is useful, e.g. if you want to restrict the translation of an embedded element within its parent. The only thing you have to do in this case is to pass the bounding box of the parent element to the `restrictedArea` option:

```
myElement.translate(50, 50, { restrictedArea:
graph.getCell(myElement.get('parent')).getBBBox() })
```

The code above makes sure that the element `myElement` never crosses the bounding box of its parent element. note that this also works if the element `myElement` has other embedded elements. In other words, the bounding box of the `myElement` that is used to calculate the restriction is the total bounding box, including all its children (in case they are “sticking out”).

dia.Element.prototype.unembed

element.[unembed](#)(cell, [opt])

element.[unembed](#)(cells, [opt])

Free up an embedded cell or an array of cells from its parent element.

dia.ElementView

The view for the [joint.dia.Element](#) model. It inherits from [joint.dia.CellView](#) and is responsible for:

- Rendering an element inside of a paper
- Handling the element's pointer events
- Provides various methods for working with the element (visually)

To find the view associated with a specific element (model), use the [findViewByModel](#) method of the paper.

```
var elementView = paper.findViewByModel(element);
```

dia.ElementView.prototype.addTools

elementView.[addTools](#)(toolsView)

Add the provided `toolsView` (of the `joint.dia.ToolsView` [type](#)) to the element view.

Adding a tools view to an element view is the last (third) step in the process of setting up element tools on an element view:

```
// 1) creating element tools
var boundaryTool = new joint.elementTools.Boundary();
var removeButton = new joint.elementTools.Remove();

// 2) creating a tools view
var toolsView = new joint.dia.ToolsView({
  name: 'basic-tools',
  tools: [boundaryTool, removeButton]
});

// 3) attaching to an element view
```

```
var elementView = element.findView(paper);
elementView.addTools(toolsView);
```

Every element view we want to attach to requires its own tools view object (`ToolsView` objects are automatically reassigned to the last element view they are added to). Similarly, every tools view we create requires its own set of tools (`ToolView` objects are automatically reassigned to the last `toolsView.tools` array they were made part of).

The element tools are added in the visible state. Use the `elementView.hideTools` [function](#) if this behavior is not desirable (e.g. if you want the element tools to appear in response to user interaction). Example:

```
elementView.addTools(toolsView);
elementView.hideTools();

paper.on('element:mouseenter', function(elementView) {
    elementView.showTools();
});

paper.on('element:mouseleave', function(elementView) {
    elementView.hideTools();
});
```

dia.ElementView.prototype.getBBox

`elementView.getBBox([opt])`

Return a bounding box of the element view.

If `opt.useModelGeometry` option is set to `true`, the resulting bounding box is calculated based on the dimensions of the element model. (This means that SVG sub elements “sticking out” of the element are excluded.) This behavior is similar to the `element.getBBox` [function](#) – but the `elementView` function transforms the bounding box to match `joint.dia.Paper` translation and scaling.

dia.ElementView.prototype.getNodeBBox

`elementView.getNodeBBox(magnet)`

Return the bounding box of the `SVGElement` provided as `magnet` (element/subelement/port of this element view).

Use the `paper.localToPaperRect` [function](#) to transform the returned bounding box to match the paper's translation and scaling.

dia.ElementView.prototype.getNodeUnrotatedBBox

`elementView.getNodeUnrotatedBBox(magnet)`

Return the unrotated bounding box of the SVGElement provided as `magnet` (element/subelement/port of this element view).

Use the `paper.localToPaperRect` [function](#) to transform the returned bounding box to match the paper's translation and scaling.

dia.ElementView.prototype.hasTools

`elementView.hasTools()`

Return `true` if this element view has a tools view attached.

`elementView.hasTools(name)`

Return `true` if this element view has a tools view of the provided `name` attached.

dia.ElementView.prototype.hideTools

`elementView.hideTools()`

Hide all tools attached to this element view.

dia.ElementView.prototype.removeTools

`elementView.removeTools()`

Remove the tools view attached to this element view.

dia.ElementView.prototype.showTools

`elementView.showTools()`

Show all tools attached to this element view.

dia.Graph.constructor

`joint.dia.Graph` is the model holding all cells (elements and links) of the diagram. It's a [Backbone model](#). The collection of all the cells is stored in the property `cells`.

The graph is a powerful data model behind all JointJS diagrams. It not only provides efficient storage for directed graphs, but also offers useful algorithms for traversing the graphs.

In order for JointJS to find the correct constructor for your cell, the `graph` option `cellNamespace` must be provided in its constructor function when a graph is instantiated. Built-in shapes are usually located in the `joint.shapes` namespace, so this is a common namespace to use. It's possible to add custom shapes to this namespace, or alternatively, you may like to use a different namespace completely.

For example, if `joint.shapes` is provided as the value of `cellNamespace`, and a cell is of type `'custom.Element'`, then the graph looks up the `joint.shapes.custom.Element` model when deserializing a graph from JSON format. If the graph is instantiated as e.g. `const graph = new joint.dia.Graph({}, { cellNamespace: myCustomNamespace })`, then the graph will read the model definition from the `myCustomNamespace.custom.Element` object instead. This option is often used in combination with the `cellViewNamespace` option on the [joint.dia.Paper](#) object.

dia.Graph.events

The following list contains events that you can react on:

- `change` - generic event triggered for any change in the graph
- `add` - triggered when a new cell is added to the graph
- `remove` - triggered when a cell is removed from the graph
- Moreover, all the events that are triggered on elements or links are propagated to the graph as well.

```
graph.on('add', function(cell) {  
    alert('New cell with id ' + cell.id + ' added to the graph.')  
})
```

dia.Graph.JSON

The JointJS graph JSON representation has the following format:

```
{  
  cells: // Array of cells (ie. links and elements).  
  {
```

```

        id: '3d90f661-fe5f-45dc-a938-bca137691eeb',// Some randomly generated
        UUID.
        type: 'basic.Rect',
        attrs: {
            'stroke': '#000'
        },
        position: {
            x: 0,
            y: 50
        },
        angle: 90,
        size: {
            width: 100,
            height: 50
        },
        z: 2,
        embeds: [
            '0c6bf4f1-d5db-4058-9e85-f2d6c74a7a30',
            'cdbfe073-b160-4e8f-a9a0-22853f29cc06'
        ],
        parent: '31f348fe-f5c6-4438-964e-9fc9273c02cb'
        // ... and some other, maybe custom, data properties
    }
]
}

```

dia.Graph.prototype.addCell

graph.addCell(cell[, opt])

Add a new cell to the graph. If `cell` is an array, all the cells in the array will be added to the graph. Any additional option or custom property provided in the options object will be accessible in the callback function of the graph add event.

If `opt.dry` is set to `true`, the graph reference is not stored on the `cell` after it's added.

If `opt.async` is set to `false`, this ensures the `cell` is rendered synchronously.

If `opt.sort` is set to `false`, the `cell` will be added at the end of the collection.

```

const rect = new joint.shapes.standard.Rectangle({
  position: { x: 100, y: 100 },
  size: { width: 90, height: 30 },
  attrs: { label: { text: 'my rectangle' } }
});
const rect2 = rect.clone();
const link = new joint.dia.Link({ source: { id: rect.id }, target: { id: rect2.id }

```



```
});
const graph = new joint.dia.Graph({}, { cellNamespace: joint.shapes });
graph.addCell(rect).addCell(rect2).addCell(link);
```

dia.Graph.prototype.addCells

```
graph.addCell(cells[, opt])
```

```
graph.addCell(cell, cell, ..[, opt])
```

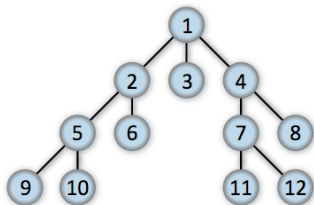
Add new cells to the graph. This is just a convenience method that wraps the [addCell](#) method.

dia.Graph.prototype.bfs

```
graph.bfs(element, iteratee [, opt])
```

Traverse the graph using the [Breadth-first search algorithm](#) starting at `element` (note the element itself will be visited too). `iteratee` is a function of the form `function(element, distance) {}` that will be called with the currently visited element and distance of that element from the root element of the search (the `element` passed to `bfs()`). If `iteratee` explicitly returns `false`, the search stops.

The following image shows the order in which elements are traversed in the graph:



Note that the `bfs()` algorithm is not only capable of traversing tree graphs but it can traverse any directed graph too.

It is smart enough not to traverse an element that was already visited.

If `opt.inbound` is `true`, reverse the search direction (it's like reversing all the link directions, i.e. swapping their source and target).

If `opt.outbound` is `true`, search follows the link directions. Calling `bfs()` with `opt.outbound` set to `true` is the most common case (graph is traversed following the direction of links).

If none of `opt.inbound` and `opt.outbound` are used or both options are set to `true`, the graph is traversed in both directions (very rare use case).

If `opt.deep` is `true`, the traversal takes into account embedded elements too. This option has the usual meaning as in other methods where `deep` option is used. For example, in a hierarchy A (top level element), A1 (embedded in A), B (top

level element), where A is not directly connected to B but its embedded element is (there is a link from A1 ----> B), `bfs(A)` would not visit B but `bfs(A, function() {}, { deep: true })` would.

dia.Graph.prototype.clear

`graph.clear([options])`

Remove all the cells from the graph. `options` object can optionally contain additional data that is passed over to the event listeners of the graph cells remove event.

dia.Graph.prototype.cloneCells

`graph.cloneCells(cells)`

Clone all the cells (elements and/or links) from the `cells` array and return an object that maps the original cell ID to the clone (i.e. an object of the form `{ [original cell ID]: [clone] }`). The reason why this object is returned instead of an array of clones is that it is very useful to know which object the clone was created for.

The number of clones returned equals `cells.length`. This function does not simply clone all the cells but it also reconstructs all the source/target and parent/embed references within `cells`. This is very useful. For example, for a graph `A --- L ----> B`, `cloneCells([A, L, B])` returns `{ A.id: A2, L.id: L2, B.id: B2 }` resulting in a graph `A2 --- L2 ----> B2`, i.e. the source and target of the link `L2` is changed to point to `A2` and `B2` (in contrast to just looping over `cells` and calling `cell.clone()` on each item).

dia.Graph.prototype.cloneSubgraph

`graph.cloneSubgraph(cells [, opt])`

Clone the whole subgraph, including all the connected links whose source/target is in the subgraph. This is equivalent to calling `graph.cloneCells(graph.getSubgraph(cells))`.

If `opt.deep` is `true`, take into account embedded cells of the subgraph cells.

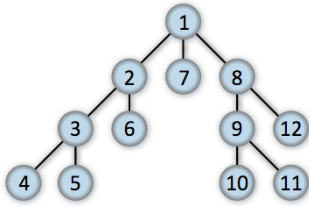
Return an object of the form `{ [original cell ID]: [clone] }`.

dia.Graph.prototype.dfs

`graph.dfs(element, iteratee [, opt])`

Traverse the graph using the Depth-first search algorithm starting at `element` (note the element itself will be visited too). `iteratee` is a function of the form `function(element, distance) {}` that will be called with the currently visited element and distance of that element from the root element of the search (the `element` passed to `dfs()`). If `iteratee` explicitly returns `false`, the search stops.

The following image shows the order in which elements are traversed in the graph:



Note that the `dfs()` algorithm is not only capable of traversing tree graphs but it can traverse any directed graph too. It is smart enough not to traverse an element that was already visited.

If `opt.inbound` is `true`, reverse the search direction (it's like reversing all the link directions, i.e. swapping their source and target).

If `opt.outbound` is `true`, search follows the link directions. Calling `dfs()` with `opt.outbound` set to `true` is the most common case (graph is traversed following the direction of links).

If none of `opt.inbound` and `opt.outbound` are used or both options are set to `true`, the graph is traversed in both directions (very rare use case).

If `opt.deep` is `true`, the traversal takes into account embedded elements too. This option has the usual meaning as in other methods where `deep` option is used. For example, in a hierarchy A (top level element), A1 (embedded in A), B (top level element), where A is not directly connected to B but its embedded element is (there is a link from A1 ----> B), `dfs(A)` would not visit B but `dfs(A, function() {}, { deep: true })` would.

dia.Graph.prototype.disconnectLinks

`graph.disconnectLinks(element)`

Disconnect all the associated links with the `element`.

dia.Graph.prototype.findModelsFromPoint

`graph.findModelsFromPoint(point)`

Find elements (instance of `joint.dia.Element`) under a certain point in the graph. `point` is an object with `x` and `y` properties. Returns an array of elements whose bounding box contains `point`. Note that there can be more than one element as elements might overlap.

dia.Graph.prototype.findModelsInArea

`graph.findModelsInArea(rect)`

Find elements (instance of `joint.dia.Element`) in a certain area in the graph. `rect` is an object with `x`, `y`, `width` and `height` properties. Returns an array of elements whose bounding box top/left coordinate falls into the `rect` rectangle.

`dia.Graph.prototype.findModelsUnderElement`

`graph.findModelsUnderElement(element [, opt])`

Find all the elements (instances of `joint.dia.Element`) that are located below `element`. `opt.searchBy` parameter optionally determines what it means for an element to be below another element. Possible values are `'bbox'` (default), `'center'`, `'origin'`, `'corner'`, `'topRight'`, and `'bottomLeft'`.

`dia.Graph.prototype.fromJSON`

`graph.fromJSON(jsonObject, [options])`

Load a graph from a JSON object (not string). Used in conjunction with the `graph.toJSON()` function.

The `options` object may contain additional data that is passed over to graph change event listeners.

Note that this method does not expect a JSON string but rather an object in the JSON format. Use `JSON.parse(jsonObject)` if you need to convert a JSON string into the object form:

```
graph.fromJSON(JSON.parse(jsonObject));
```

Example of storing stringified JSON objects:

```
var jsonString = JSON.stringify(graph.toJSON());  
// ... send jsonString to the server  
// store jsonString to localStorage or do whatever you want  
// later on ...  
graph.fromJSON(JSON.parse(jsonString));
```

Example of storing JSON objects directly:

```
var jsonObject = graph.toJSON();  
// ... send jsonObject to the server  
// store jsonObject (e.g. in a non-relational database)  
// later on ...  
graph.fromJSON(jsonObject)
```

dia.Graph.prototype.getBBox

graph.getBBox()

Returns the bounding box ([g.Rect](#)) that surrounds all cells in the graph. It returns `null` if the graph is empty.

```
var bbox = graph.getBBox().inflate(10);
```

dia.Graph.prototype.getCell

graph.getCell(id)

Get a cell from the graph by its `id`.

dia.Graph.prototype.getCells

graph.getCells()

Return an array of all elements and links in the graph. The cells are sorted by their `z` index (the smallest `z` being first).

dia.Graph.prototype.getCellsBBox

graph.getCellsBBox(cells[, opt])

Returns the bounding box ([g.Rect](#)) that surrounds all the given cells.

```
// Get the bounding box of all `el1` successors and their embeds
var bbox = graph.getCellsBBox(graph.getSuccessors(el1), { deep: true });
```

dia.Graph.prototype.getCommonAncestor

graph.getCommonAncestor(...cells)

Return the common ancestor of all the cells passed as arguments. For example, if an element `B` is embedded in an element `A` and an element `C` is also embedded in the element `A`, `graph.getCommonAncestor(B, C)` returns the element `A`. This also works on an arbitrary deep hierarchy.

dia.Graph.prototype.getConnections

`graph.getConnections(element [, opt])`

Get all links connected with `element`.

If `opt.inbound === true`, return only inbound connected links. Conversely, if `opt.outbound === true`, return only outbound connected links. If both of these options are left undefined, or if both of them are set to `true`, return both inbound and outbound links.

By default, this function returns only immediate (“shallow”) inbound and outbound links - no recursion. (Note that connections from `element` to embedded child elements, and connections to `element` from embedding parent elements count as “shallow”, too - they too are returned.)

If `opt.deep === true`, return all outside links that connect with `element` or any of its descendants (“descendants” meaning elements that are embedded or deeply embedded within `element`). The `inbound` and `outbound` options can still be applied on top of this option.

Note that the specification of `opt.deep` excludes links that connect two descendants of `element` (“enclosed” links). If you do need to find all links connected with and/or enclosed within `element`, you should use `opt.deep === true` alongside an additional option: `opt.includeEnclosed === true`.

If `opt.indirect === true`, also return links that can only be considered connected to `element` if we go “against the flow” of directed links at link-link connections.

Example use:

```
var links = graph.getConnections(element); // inbound and outbound
var links = graph.getConnections(element, { outbound: true });
var links = graph.getConnections(element, { deep: true }); // inbound and outbound
var links = graph.getConnections(element, { inbound: true, deep: true });
var links = graph.getConnections(element, { outbound: true, deep: true,
includeEnclosed: true });
var links = graph.getConnections(element, { indirect: true });
```

dia.Graph.prototype.getElements

`graph.getElements()`

Return an array of all elements in the graph. The elements are sorted by their `z` index (the smallest `z` being first).

dia.Graph.prototype.getFirstCell

graph.[getFirstCell\(\)](#)

Get the first cell (element or link) in the graph. The first cell is defined as the cell with the lowest `z` property (the cell most in the back, see the Presentation section of [joint.dia.Element](#)).

dia.Graph.prototype.getLastCell

graph.[getLastCell\(\)](#)

Get the last cell (element or link) in the graph. The last cell is defined as the cell with the highest `z` property (the cell most in the front, see the Presentation section of [joint.dia.Element](#)).

dia.Graph.prototype.getLinks

graph.[getLinks\(\)](#)

Return an array of all links in the graph. The links are sorted by their `z` index (the smallest `z` being first).

dia.Graph.prototype.getNeighbors

graph.[getNeighbors](#)(element [, opt])

Get all the neighbors of `element` in the graph. Neighbors are all the elements connected to `element` via either an inbound or an outbound link.

Accepts several options, which may be provided inside an `opt` object:

- `deep` - also return all the neighbors of all the elements embedded inside `element`.
- `inbound` - return only inbound neighbors (neighbors connected with a link whose `target` is the `element`).
- `outbound` - return only outbound neighbors (neighbors connected with a link whose `source` is the `element`).
- `indirect` - in addition to standard rules (including `deep` / `outbound` / `inbound` modifications), also return the elements that can only be considered neighbors of `element` if we go “against the flow” of directed links at link-link connections.

dia.Graph.prototype.getPredecessors

`graph.getPredecessors(element [, opt])`

Return an array of all the predecessors of `element` . By default, [Depth-first search algorithm](#) is used (important for the order of returned elements).

If `opt.breadthFirst` is set to `true` , use [Breadth-first search algorithm](#) instead.

If `opt.deep` is set to `true` , take into account embedded elements too (see [dfs\(\)](#) for details).

dia.Graph.prototype.getSinks

`graph.getSinks()`

Return an array of all the leafs of the graph. Time complexity: $O(|V|)$.

dia.Graph.prototype.getSources

`graph.getSources()`

Return an array of all the roots of the graph. Time complexity: $O(|V|)$.

dia.Graph.prototype.getSubgraph

`graph.getSubgraph(cells [, opt])`

Return an array of cells that result from finding elements/links that are connected to any of the cells in the `cells` array. This function loops over `cells` and if the current cell is a link, it collects its source/target elements; if it is an element, it collects its incoming and outgoing links if both the link ends (source/target) are in the `cells` array. For example, for a single element, the result is that very same element. For two elements connected with a link: `A --- L -> B`, the result of `getSubgraph([A, B])` is `[A, L, B]` and the result of `getSubgraph([L])` is also `[A, L, B]` .

If `opt.deep` is `true` take into account all the embedded cells too when finding neighboring links/elements.

dia.Graph.prototype.getSuccessors

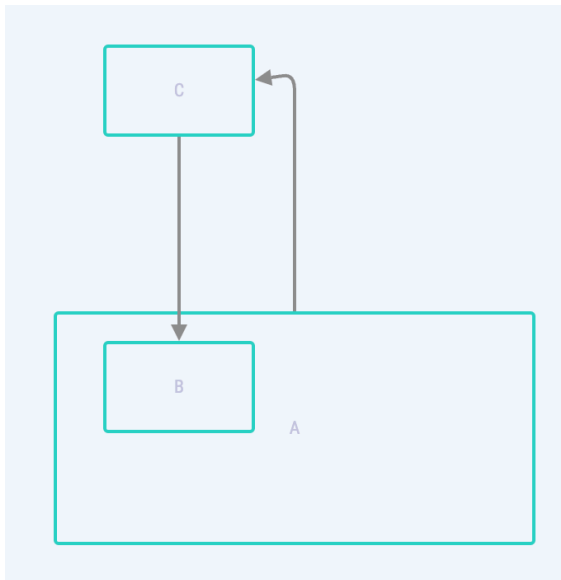
`graph.getSuccessors(element [, opt])`

Return an array of all the successors of `element` . By default, a Depth-first search algorithm is used (important for the order of returned elements).

If `opt.breadthFirst` is set to `true` , use a Breadth-first search algorithm instead.

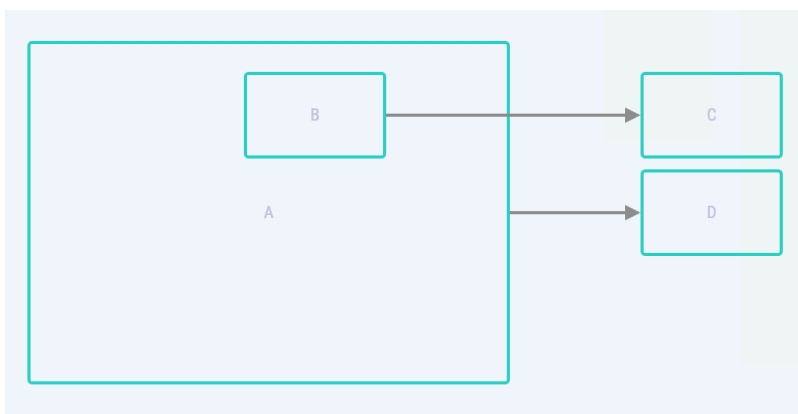
Generally, `getSuccessors` cares about the direction of the links. It follows links from their source to target only. The resulting array contains the elements that you visit if you follow the directed links. The links are simply navigated, and embedding is not considered.

In the following image, the successors of `A` are `C` and `B` . Embedding is not taken into account.



If `opt.deep` is set to `true` , embedded elements are taken into account too (see `dfs()` for details). That means elements connected to any of the descendants are also successors.

In the following image, if `{ deep: false }` , the only successor of `A` is `D` . Embedding is not taken into account. If `{ deep: true }` , and `B` is embedded in `A` , that means the successors of `A` are `C` and `D` .



`dia.Graph.prototype.isNeighbor`

`graph.isNeighbor(elementA, elementB [, opt])`

Return `true` if `elementB` is a neighbor of `elementA`. A neighbor of an element is another element connected to it via an inbound and/or outbound link.

Accepts several options, which may be provided inside an `opt` object:

- `deep` - return `true` also if `elementB` is a neighbor of an element embedded in `elementA`.
- `outbound` - return `true` only if `elementB` is a succeeding neighbor of `elementA`. For example, if `elementB` is connected to a directed link “behind” the connection of `elementA`.
- `inbound` - return `true` only if `elementB` is a preceding neighbor `elementA`. For example, if `elementB` is connected to a directed link “ahead of” the connection of `elementA`.
- `indirect` - in addition to standard rules (including `deep` / `outbound` / `inbound` modifications), also return `true` if `elementB` can only be considered a neighbor of `elementA` if we go “against the flow” of directed links at link-link connections.

dia.Graph.prototype.isPredecessor

`graph.isPredecessor(elementA, elementB)`

Return `true` if `elementB` is a predecessor of `elementA`.

dia.Graph.prototype.isSink

`graph.isSink(element)`

Return `true` if `element` is a leaf, i.e. there is no link coming out of the element. Time complexity: $O(1)$.

dia.Graph.prototype.isSource

`graph.isSource(element)`

Return `true` if `element` is a root, i.e. there is no link that targets the element. Time complexity: $O(1)$.

dia.Graph.prototype.isSuccessor

`graph.isSuccessor(elementA, elementB)`

Return `true` if `elementB` is a successor of `elementA`.

dia.Graph.prototype.maxZIndex

```
graph.maxZIndex()
```

Get the highest Z value in the graph (the value of the cell on front).

dia.Graph.prototype.minZIndex

```
graph.minZIndex()
```

Get the lowest Z value in the graph (the value of the cell on the back).

dia.Graph.prototype.removeCells

```
graph.removeCells(cells[, opt])
```

```
graph.removeCells(cell, cell, ..[, opt])
```

Removes the given cells from the graph.

dia.Graph.prototype.removeLinks

```
graph.removeLinks(element)
```

Remove all the associated links with the `element`.

dia.Graph.prototype.resetCells

```
graph.resetCells(cells[, opt])
```

```
graph.resetCells(cell, cell, ..[, opt])
```

Reset cells in the graph. Update all the cells in the graph in one bulk. This is a more efficient method of adding cells to the graph if you want to replace all the cells in one go. The `options` object can optionally contain additional data that is passed over to the event listeners of the graph reset event.

dia.Graph.prototype.search

graph.[search](#)(element, iteratee [, opt])

Traverse the graph starting at `element` following links. This function is a wrapper around [dfs\(\)](#) or [bfs\(\)](#). By default, it uses [dfs\(\)](#). If `opt.breadthFirst` is set to `true`, [bfs\(\)](#) will be used instead.

dia.Graph.prototype.toJSON

graph.[toJSON](#)()

Return an object representation of the graph, which can be used for persistence or serialization. Use the `graph.fromJSON()` [function](#) to load a previously converted graph.

Note that this method does not return a JSON string but rather an object that can then be serialized to JSON with `JSON.stringify(jsonObject)`:

```
var jsonString = JSON.stringify(graph.toJSON());
```

dia.Graph.prototype.translate

graph.[translate](#)(tx, ty [, opt])

Translate all cells in the graph by `tx` and `ty` pixels. It uses the [dia.Element.translate\(\)](#) and [dia.Link.translate\(\)](#) methods internally.

dia.HighlighterView

The base class for [highlighters](#). The *HighlighterView* class does not implement any particular visual emphasis. It takes care of the management of the instances and introduce mechanisms for displaying, updating and cleaning arbitrary markings added to the *SVGElements* of *cellViews*.

The class is not meant to be initialized with the constructor.

dia.HighlighterView.add

HighlighterView.[add](#)(cellView, selector, id, options)

Create an instance of the highlighter and add it to the *cellView*.

It returns an instance of the *HighlighterView*.

If a highlighter with the same *id* exists on the *CellView*, it is removed and a new highlighter is added and returned. You can check if a highlighter already exists by calling `HighlighterView.get()`.

- **cellView** is an instance of `dia.CellView`.
- **selector** points to an *SVGElement* of the *cellView*. It must be one of the following types:

Type	Description	Example
String	A selector from the cell's <u>markup</u> .	<pre>'root' 'body'</pre>
Object	An object with property <i>selector</i> , which is a selector from the (cell/label/port) <u>markup</u> . There is additional property <i>port</i> (id of a port) available for <code>dia.ElementView</code> . And property <i>label</i> (index of a label) is available for <code>dia.LinkView</code> . If no <i>selector</i> property is provided, the <i>root</i> is used.	<pre>{ selector: 'body' } // cells { port: 'port1', selector: 'portBody' } // elements { label: 1, selector: 'labelBody' } // links</pre>
SVGElement	An <i>SVGElement</i> , the descendant of <i>cellView.el</i> or the root itself.	<pre>cellView.el // the root cellView.el.querySelector ('rect') // an SVGRectangle</pre>

- **id** is a unique identifier of the highlighter. Only single highlighter with given *id* can exist on the *CellView* at one time.
- **options** passed to the constructor of the highlighter view.

dia.HighlighterView.get

`HighlighterView.get(cellView)`

Return an array of highlighters on the *cellView*. Only highlighters, which are instances of this class are returned.

```
// Extend the Base Highlighter Class
const ChildHighlighterView = joint.dia.Highlighter.extend({});
```

```
const h1 = joint.dia.HighlighterView.add(cellView, 'root', 'id1');
const c1 = ChildHighlighterView.add(cellView, 'root', 'id2');

const parentHighlighters = joint.dia.Highlighters.get(cellView);
assert.ok(parentHighlighters.includes(h1));
assert.ok(parentHighlighters.includes(c1));
const childHighlighters = ChildHighlighterView.get(cellView);
assert.notOk(parentHighlighters.includes(h1));
assert.ok(parentHighlighters.includes(c1));
```

HighlighterView.get(cellView, id)

Return the highlighter with given *id* on the *cellView*. The highlighter returned must be an instance of this class. If such a highlighter does not exist, **null** is returned.

```
// Extend the Base Highlighter Class
const ChildHighlighterView = joint.dia.Highlighter.extend({});

const h1 = joint.dia.HighlighterView.add(cellView, 'root', 'h1');
const c1 = ChildHighlighterView.add(cellView, 'root', 'c1');

assert.equal(h1, joint.dia.Highlighters.get(cellView, 'h1'));
assert.equal(c1, joint.dia.Highlighters.get(cellView, 'c1'));
// There is no instance of ChildHighlighterView on the CellView
assert.equal(null, ChildHighlighterView.get(cellView, 'h1'));
assert.equal(c1, ChildHighlighterView.get(cellView, 'c1'));
```

dia.HighlighterView.remove

HighlighterView.remove(cellView)

Remove all the highlighters from the *cellView*. Only highlighters, which are instances of this class are removed.

HighlighterView.remove(cellView, id)

Remove the highlighter with given *id* from the *cellView*. The highlighter which is not an instance of this class is not removed.

dia.HighlighterView.removeAll

HighlighterView.removeAll(paper)

Remove all the highlighters on *paper*. Only highlighters, which are instances of this class are removed.

HighlighterView.`removeAll`(paper, id)

Remove all the highlighters with given *id* on the *paper*. Only highlighters, which are instances of this class are removed.

dia.HighlighterView.prototype.highlight

highlighter.`highlight`(cellView, node)

Mark/Emphasize the *node* (*SVGElement*) of the *cellView*. e.g. render a rectangle above the node, add a class to the *node*.

dia.HighlighterView.prototype.MOUNTABLE

HighlighterView.prototype.MOUNTABLE

If the property is *true*, the highlighter view is attached to the DOM.

If the property is *false*, it is not attached upon highlight.

The default is *true*.

```
const ChildView = joint.dia.HighlighterView.extend({  
  
  // HighlighterView `el` is not meant to be appended  
  MOUNTABLE: false,  
  
  highlight(cellView, node) {  
    node.setAttribute('my-attribute',  
String(cellView.model.get('highlightAttribute')));  
  },  
  
  unhighlight(cellView, node) {  
    node.removeAttribute('my-attribute');  
  }  
  
});
```

dia.HighlighterView.prototype.unhighlight

highlighter.`unhighlight`(cellView, node)

Remove marks made by `highlight()` from the *node* (SVGElement) of the *cellView*.

dia.HighlighterView.prototype.UPDATABLE

HighlighterView.prototype.UPDATABLE

If the property is *true*, the highlighter is updated (*highlight()* function is called) every time when the related *cellView* requires an update.

If the property is *false*, the *highlight()* method is called only once when the highlighter is added.

The default is *true*.

```
const ChildView = joint.dia.HighlighterView.extend({

  MOUNTABLE: false,
  UPDATABLE: false,

  highlight(_cellView, node) {
    node.classList.add('my-class');
  },

  unhighlight(_cellView, node) {
    node.classList.remove('my-class');
  }

});
```

dia.HighlighterView.prototype.UPDATE_ATTRIBUTES

HighlighterView.prototype.UPDATE_ATTRIBUTES

The highlighter is updated (*highlight()* function is called) every time any of the attributes from the list change.

The value could be an array of strings or a function returning an array of strings.

The default is `[]`.

```
const ColorHighlighter = joint.dia.HighlighterView.extend({

  UPDATE_ATTRIBUTES: ['color'],

  highlight(cellView, node) {
    node.style.fill = cellView.model.get('color');
  },

});
```



```

    unhighlight(_cellView, node) {
        node.style.fill = '';
    }

});

ColorHighlighter.add(rectangle.findView(paper), 'body', 'color-highlighter');
rectangle.set('color', 'gray'); // will call highlight()

```

dia.HighlighterView.prototype.options.layer

layer - the stacking context of the highlighter. Applicable for mountable highlighters only.

<code>null</code>	Render the highlighter above the cell. It can be hidden by a cell with a higher z index.
<code>"back"</code>	Render the highlighter behind all the cells.
<code>"front"</code>	Render the highlighter in front of all the cells.

dia.HighlighterView.prototype.options.z

z - the stacking order (z-index) of the highlighter in the given stacking context. Applicable for mountable highlighters only.

dia.Link

The basic model for diagram links. It inherits from joint.dia.Cell with a few additional properties and methods specific to links. For a quick introduction to elements, see our tutorial.

Links' properties may be split into several groups according to their function:

Geometry

Links have two crucial properties: `source` and `target`. They define the starting point and the end point of the link. They can be defined with a Cell id (optionally, with additional subelement/magnet/port reference) or with a Point:

```

var link1 = new joint.dia.Link({
    source: { id: sourceId },

```

```

    target: { id: targetId, port: portId }
  });

  var link2 = new joint.dia.Link({
    source: { id: sourceId },
    target: { x: 100, y: 100 }
  });

```

The `source` and `target` properties accept additional modifier properties that modify the actual position of the link end: `anchor` / `linkAnchor`, and `connectionPoint`.

Additionally, the path of the link is determined by its `vertices`, and the applied `router` and `connector`. All these properties are described in more detail in [link geometry documentation](#).

Presentation

Each `joint.dia.Link` defines its own SVG markup which is then used by `joint.dia.LinkView` to render the link to the `paper`. For instance, the `joint.shapes.standard.Link` (which inherits from `joint.dia.Link`) defines its markup using the [JSON array notation](#) as follows:

```

markup: [{
  tagName: 'path',
  selector: 'wrapper',
  attributes: {
    'fill': 'none',
    'cursor': 'pointer',
    'stroke': 'transparent',
    'stroke-linecap': 'round'
  }
}, {
  tagName: 'path',
  selector: 'line',
  attributes: {
    'fill': 'none',
    'pointer-events': 'none'
  }
}]

```

As we can see, the `joint.shapes.standard.Link` shape consists of two “subelements”: one `SVGPathElement` named `'wrapper'` and one `SVGPathElement` named `'line'`. The `attrs` object refers to the subelements' names (“selectors”) to provide SVG attributes to these constituent `SVGElements`.

Direct use of `joint.dia.Link` is deprecated. Use links from the `standard` namespace.

The `joint.dia.Link` markup looks like the following:

```

<path class="connection"/>
<path class="marker-source"/>
<path class="marker-target"/>
<path class="connection-wrap"/>
<g class="labels" />
<g class="marker-vertices"/>
<g class="marker-arrowheads"/>
<g class="link-tools" />

```

As you can see, the link consists of a couple of SVG path elements and a couple of SVG group elements:

- `.connection` is the actual line of the link.
- `.connection-wrap` is an SVG path element that covers the `.connection` element and is usually thicker so that the link is able to handle pointer events (mousedown, mousemove, mouseup) that didn't target the thin `.connection` path exactly. This makes it easy to “grab” the link even though the mouse cursor didn't point exactly at the (usually thin) `.connection` path element.
- `.marker-source` and `.marker-target` are the arrowheads of the link.

The subelements of the link can be optionally styled through link attributes. For example:

```

link.attr({
  '.connection': { stroke: 'blue' },
  '.marker-source': { fill: 'red', d: 'M 10 0 L 0 5 L 10 10 z' },
  '.marker-target': { fill: 'yellow', d: 'M 10 0 L 0 5 L 10 10 z' }
});

```

The `markup` attribute is not the only presentation attribute that may be specified for a `joint.dia.Link`. However, the following four Link properties are deprecated and included only for backwards compatibility. Use [link tools](#) instead.

- `vertexMarkup` - provide default vertex markup for all vertices created on an instance of this Link type (on hover).
- `toolMarkup` - provide custom tool markup for all instances of this Link type (on hover).
- `doubleToolMarkup` - provide custom markup for second set of tools for all instances of this Link type (on hover, if `linkView.doubleLinkTools` is `true`).
- `arrowheadMarkup` - provide custom arrowhead markup for all instances of this Link type (on hover).

Styling

The keys of the `attrs` object are “selectors” that match subelements defined in the link's [markup](#) (see above). The values of this object are [special JointJS attributes](#) or native SVG attributes that should be set on the selected subelements. (A list of native SVG attributes and their descriptions can be found online, e.g. on [MDN](#).)

For example, in order to set a red stroke color on a subelement called `'line'`, the `attrs` object would contain:

```

line: { stroke: 'red' }

```

If you simply need to change a value of an attribute, it is not recommended to modify the `attrs` object of the link directly. You should use the `link.attr()` [method](#) instead. For example, to set the attributes according to the above example, you would write:

```
link.attr({
  line: { stroke: 'red' }
});
```

We can use the `joint.shapes.standard.Link` type (which inherits from `joint.dia.Link`) as an example. The `attrs` object in its definition is provided below:

```
attrs: {
  line: {
    connection: true,
    stroke: '#333333',
    strokeWidth: 2,
    strokeLinejoin: 'round',
    targetMarker: {
      'type': 'path',
      'd': 'M 10 -5 0 0 10 5 z'
    }
  },
  wrapper: {
    connection: true,
    strokeWidth: 10,
    strokeLinejoin: 'round'
  }
}
```

Notice that the object makes use of special JointJS attributes (e.g. `connection`, `targetMarker`) on top of native SVG attributes (e.g. `stroke`, `strokeWidth`). All of these “special attributes” are listed in the [attributes](#) section of this documentation. You should also refer to our tutorial on special attributes.

In the context of links, the most important special attribute is `connection`. It specifies that the `SVGPathElement(s)` in question should follow the path of the Link's model, as provided by the interplay of [link geometry methods](#).

Markup Attributes

Attributes defined directly inside link [markup](#) are evaluated only once at CellView creation, while attributes defined in the `attrs` object are evaluated on every model change. JointJS special attributes usually depend on the current state of the model, which means that they should be defined in the `attrs` object (alongside any SVG attributes that you expect to be modified during the runtime of your application).

Z

The `z` property specifies the stack order of the element in the SVG DOM. An element with a higher `z` value will be rendered in front of an element with a lower `z` value. ([This also applies to Elements.](#))

Labels

You may provide an array of labels to the link through the `labels` property. Each label can have its own `markup`, `size`, `attrs`, and `position` objects specified. The values in those objects take precedence over any defaults which may apply on the label.

To avoid excessive repetition, you can provide a `defaultLabel` property to the link, to set the `markup`, `size`, `attrs`, and `position` objects which should be applied to all labels on the link. The properties from `defaultLabel` act as a template, which is overwritten / extended by individually-specified label properties as appropriate.

An example of creating a new link instance with both the `defaultLabel` and `labels` properties specified can be seen below. For more details, see the [link labels documentation](#). You should also refer to our tutorial on link labels.

```
const link = new joint.shapes.standard.Link({
  source: { x: 50, y: 400 },
  target: { x: 500, y: 400 },
  defaultLabel: {
    // applied to all labels on this link:
    markup: [
      {
        tagName: 'rect',
        selector: 'body'
      }, {
        tagName: 'text',
        selector: 'label'
      }
    ],
    size: {
      // used by `calc()` expressions in `attrs`
      width: 150,
      height: 30
    },
    attrs: {
      body: {
        width: 'calc(w)',
        height: 'calc(h)',
        // center around label position:
        x: 'calc(w/-2)',
        y: 'calc(h/-2)',
        stroke: 'black',
        fill: 'white'
      },
      label: {
        textWrap: {
          width: 'calc(w-5)',
          height: 'calc(h-5)'
        },
        // center text around label position:

```

```

        // (no `x` and `y` provided = no offset)
        textAnchor: 'middle',
        textVerticalAnchor: 'middle',
        fontSize: 16,
        fontFamily: 'sans-serif'
    }
},
labels: [{
    // specification of an individual label:
    size: { width: 200 }, // partially overwrites `defaultLabel.size`
    attrs: {
        label: {
            text: 'Hello World'
        }
    },
    position: { distance: 0.25 } // overwrites built-in default
}]
});

```

Custom Properties

It is also possible to pass custom properties to the link. These may be useful to identify an individual link model for the purposes of linkView interaction (see [LinkView documentation](#) for more information).

For example, if a custom contextmenu interaction should only be enabled for `link1` but not `link2`, we could add a custom property `customLinkInteractions` to `link1`:

```

var CustomLinkView = joint.dia.LinkView.extend({
    contextmenu: function(evt, x, y) {
        if (this.model.get('customLinkInteractions')) {
            // only links with `customLinkInteractions: true`
            this.addLabel(x, y);
        }
    }
});

var paper = new joint.dia.Paper({
    //...
    linkView: CustomLinkView,
    interactive: function(cellView) {
        if (cellView.model.get('customLinkInteractions')) {
            // only links with `customLinkInteractions: true`
            return { vertexAdd: false };
        }
        return true; // otherwise
    }
});

```

```
});

var link1 = new joint.dia.Link({
  //...
  customLinkInteractions: true // right-click adds a label
});

var link2 = new joint.dia.Link({
  //...
  customLinkInteractions: false // or omit completely
});
```

Events

Links trigger several special events, detailed in the [link events documentation](#).

Custom Link

It is possible to extend the `joint.dia.Link` class to create a custom link. A custom link may override default Link properties to assign its own defaults. These values override built-in defaults, if necessary, and are applied to all instances of the new Link type, unless an individual instance overrides them with its own values. The following Link properties are applicable in this context:

- `markup` - provide default link markup for all instances of this Link type, specified with a [JSON array](#).
- `attrs` - provide default link attributes for all instances of this Link type. These allow you to change the style and size of SVG elements, identified by their selectors.
- `defaultLabel` - provide default properties (markup, size, attrs, position) for all labels created on an instance of this Link type.

The values of these defaults may be important; the `linkView.addLabel()` [shortcut function](#) is only capable of adding default labels to the link.

Creating custom links is explained in more detail in our tutorial.

Example:

```
var CustomLink = joint.dia.Link.define('examples.CustomLink', {
  defaultLabel: {
    markup: [
      {
        tagName: 'circle',
        selector: 'body'
      }, {
        tagName: 'text',
        selector: 'label'
      }
    ],
    size: {
```

```

        // used by `calc()` expressions in `attrs`
        width: 20,
        height: 20
    },
    attrs: {
        label: {
            text: '%', // default text for all labels
            fill: '#ff0000', // default text color for all labels
            fontSize: 14,
            textAnchor: 'middle',
            textVerticalAnchor: 'middle',
            pointerEvents: 'none'
        },
        body: {
            // currently, calc() is responsive to `size` property
            // uncomment to make calc() responsive to size of 'label':
            //ref: 'label', // subelement identified by 'label' selector
            fill: '#ffffff',
            stroke: '#000000',
            strokeWidth: 1,
            r: 'calc(d/2 + 2)'
        }
    },
    position: {
        // keep built-in default `distance` for all labels (0.5)
        offset: {
            y: -20 // offset by 20px upwards for all labels
        },
        args: {
            absoluteOffset: true // absolute offset for all labels
        }
    }
}
});

var link = new CustomLink({
    //...
});

```

Built-in Default Properties

To ensure backwards compatibility, the `joint.dia.Link` class comes with a private built-in `defaultLabel` property. It is reproduced here for reference:

```

defaultLabel: {
    // built-in default markup:
    // applied only if neither one of the following is provided:

```



```

// - individual label `markup` property
// - `defaultLabel.markup` property
markup: [
  {
    tagName: 'rect',
    selector: 'rect'
  }, {
    tagName: 'text',
    selector: 'text'
  }
],
// built-in default attributes:
// applied only if built-in default markup is used
attrs: {
  text: {
    fill: '#000000',
    fontSize: 14,
    textAnchor: 'middle',
    yAlignment: 'middle',
    pointerEvents: 'none'
  },
  rect: {
    ref: 'text',
    fill: '#ffffff',
    rx: 3,
    ry: 3,
    refWidth: 1,
    refHeight: 1,
    refX: 0,
    refY: 0
  }
},
// built-in default position:
// merged with `defaultLabel.position` and individual label `position`
position: {
  distance: 0.5
}
}

```

If custom `markup` object is not provided (i.e. there is no class-specific `defaultLabel.markup` object, nor any instance-specific `defaultLabel.markup`, nor an individual label-specific `markup` property), then built-in default label `markup` is applied (as reproduced above). Alongside, the built-in default label `attrs` object is applied. Note that the built-in default `attrs` object is applied as a template in this context, which means that you may enhance it with a custom `attrs` object (class-specific / instance-specific / individual label-specific). However, in the interest of keeping your code maintainable and easy to understand, it is very highly recommended that you provide both your own `markup` object and your own `attrs` object, unless you want to use the built-in default precisely as-is.

The built-in default `position` object behaves slightly differently. Regardless of `markup`, it is always merged with custom `position` objects (class-specific / instance-specific / individual label-specific) - but it has the lowest priority of the four. That is, if at least one of the custom `position` objects provides a `distance` value, that value will have precedence over the built-in default `position.distance`. If no custom `position.distance` is provided, then the built-in default is applied (placing labels at midpoints of links).

dia.Link.events

- `change` - generic event triggered for any change on the link
- `change:source` - triggered when the link changes its source
- `change:target` - triggered when the link changes its target
- `change:attrs` - triggered when the link changes its attributes
- `change:smooth` - (deprecated) triggered when the link toggled interpolation
- `change:manhattan` - (deprecated) triggered when the link toggled orthogonal routing
- `change:vertices` - triggered when the link changes its vertices array
- `change:z` - triggered when the link is moved in the z-level ([toFront](#) and [toBack](#))
- `transition:start` - triggered when a transition starts.
- `transition:end` - triggered when a transition ends.

```
link.on('change:source', function() { alert('source of the link changed') })
```

dia.Link.geometry

The shape of a link is determined by five properties - `source`, `target`, `vertices`, `router` and `connector`.

Source and Target

The `source` and `target` properties have to be provided when creating a Link. Either a `Point` can be provided (with `x` and `y` properties) or a `Cell` (Element or Link, either directly or via an `id` property).

```
link.source(new g.Point(100, 100));
link.source({ x: 100, y: 100 });

link.source(rect);
link.source({ id: rect.id });
link.source(link2);
link.source({ id: link2.id });
```

If the end is specified as a `Cell`, a specific subelement on that cell may be identified for use as the link end. The `selector` property allows specifying the subelement with a selector string, while the `magnet` property uses magnet

id, and the `port` property (on Elements only) uses the port id.

```
link.source(rect, {
  selector: 'connectorSquare'
});
link.source(link2, {
  selector: 'midPointCircle'
});
```

In any case, we need to obtain a single point from the object provided to the `source/target` property. That point is found according to additional properties provided to the function. The accepted modifier properties depend on the class of the provided object:

- `Point` or an object with `x` and `y` properties - coordinates of the point are used as the coordinates of the link's anchor directly. No modification of any kind. (Ignores `anchor/linkAnchor/connectionPoint` properties, if provided.)
- `Element` - the precise position of the link's end anchor depends on used `anchor` function, with additional optical modifications applied by `connectionPoint` function. (Ignores `linkAnchor` property, if provided.)
- `Link` - the precise position of the link's end anchor depends on provided `linkAnchor` function. (Ignores `anchor/connectionPoint` properties, if provided.)
- `Link` with specified `subelement / magnet` property - same as `Element` (see above).

The `connectionStrategy` `paper option` is also relevant to mention in this context. It determines what happens to the link end when it is modified due to specific kinds of user interaction.

Anchor

If the link end (`source/target`) is an `Element` (or a `subelement/magnet` of a `Link` – but not a `Link` itself), the precise position of the link end's anchor may be specified by the `anchor` property. Every link has two anchors; one at the source end and one at the target end.

A link end anchor is a point inside a given (sub)element that the link path wants to connect to at that source/target end - if it were not obstructed by the body of the element itself (it is the role of `connectionPoints` to then take the obstructing end element itself into account). Alongside `link vertices`, source and target anchors determine the basic link path. Then, it is the job of `connectionPoints`, `routers`, and `connectors` to modify that path to make it look good.

The anchor functions reference the end element to find the anchor point - e.g. the center of the end element, or its top-right corner. Notably, anchor functions also allow you to offset found anchor points by a custom distance in both dimensions from the standard position (e.g. 10 pixels to the right and 20 pixels below the center of an end element). Several pre-made `anchors` are provided inside the JointJS library in the `joint.anchors` namespace.

```
link.source(rect, {
  anchor: {
    name: 'bottomLeft',
    args: {
      dx: 20,
      dy: -10
    }
  }
});
```

```
    }
  });
```

If an anchor function is not provided, the `defaultAnchor` [paper option](#) is used instead. The `joint.anchors.center` [function](#) is used by default.

Link Anchor

If the link end ([source/target](#)) is a Link (not an Element or a Link subelement/magnet), the precise position of the link end's anchor may be specified by the `linkAnchor` property. If a link anchor method is used, no `connectionPoints` are applied.

A link end link anchor is a point on another link that this link's path wants to connect to at that source/target end. Alongside [link vertices](#), source and target anchors determine the basic link path.

The link anchor functions reference the end link to find the anchor point - e.g. a point at a given ratio from the start, or the closest point. Several pre-made [linkAnchors](#) are provided inside the JointJS library in the `joint.linkAnchors` namespace.

```
link.source(link2, {
  linkAnchor: {
    name: 'connectionRatio',
    args: {
      ratio: 0.25
    }
  }
});
```

If a link anchor function is not provided, the `defaultLinkAnchor` [paper option](#) is used instead. The `joint.linkAnchors.connectionRatio` [function](#) with a value of `0.5` is used by default (i.e. the anchor is placed at the midpoint of a Link by default).

Connection Point

The link end's connection point may be specified by the `connectionPoint` property. Every link has two connection points; one at the source end and one at the target end.

A link connection point is the point at which the link path actually ends at, taking the end element into account. This point will always lie on the link path (the line connecting link [anchors](#) and [vertices](#) together, in order).

The `connectionPoints` are found by considering intersections between the link path and a desired feature of the end element (e.g. bounding box, shape boundary, anchor). Although `connectionPoints` are not capable of being offset off the link path ([anchors](#) should be used to modify the link path if this is required), they can be offset *along* the path - e.g. to form a gap between the element and the actual link. Several pre-made [connectionPoints](#) are provided in the JointJS library in the `joint.connectionPoints` namespace.

```
link.source(rect, {
  connectionPoint: {
    name: 'boundary',
```

```

        args: {
            offset: 5
        }
    }
});

```

If a connection point function is not provided, the `defaultConnectionPoint` [paper option](#) is used instead. The `joint.connectionPoints.bbox` [function](#) is used by default.

Connection Strategy

Related to [anchors](#) and [connectionPoints](#) is the `connectionStrategy` [paper option](#). It allows you to specify which `anchor` and `connectionPoint` properties should be assigned to end elements in response to user interaction (e.g. in response to dragging a link arrowhead onto a new element). This setting is necessary because assigned `anchor` and `connectionPoint` are not preserved when the end element is reassigned by the user - and the paper's `defaultAnchor` and `defaultConnectionPoint` are used instead.

Several pre-made [connectionStrategies](#) are provided inside the JointJS library in the `joint.connectionStrategies` namespace, but you may find it necessary to create your own [custom connection strategies](#). This is a [paper-level setting](#); `connectionStrategies` not assigned on a link-by-link basis.

```

paper.options.connectionStrategy = joint.connectionStrategies.pinAbsolute;

```

If a connection strategy is not provided, the `joint.connectionStrategies.useDefaults` [function](#) is used by default.

Vertices

The `vertices` array is an array of user-defined points for the link to pass through. Alongside the source and target [anchors](#), the link vertices determine the basic link path. This skeleton path is then used for determining the [link route](#). The vertices can be accessed with the `link.vertices()` [function](#) and related functions.

```

link.vertices();
link.vertices([ { x: 100, y: 120 }, { x: 150, y: 60 } ]);

```

Router

Routers take an array of [link vertices](#) and transform them into an array of route points that the link should go through. This route is then used for generating the [connection SVG path commands](#). The `router` property of a link can be accessed with the `link.router()` [function](#).

A collection of pre-made [routers](#) is provided inside the JointJS library in the `joint.routers` namespace. This includes “smart routers” that are able to automatically avoid obstacles (elements) in their way.

```

link.router('manhattan', {
    excludeEnds: ['source'],
    excludeTypes: ['myNamespace.MyCommentElement'],

```

```

    startDirections: ['top'],
    endDirections: ['bottom']
  });

```

If a router is not provided, the `defaultRouter` [paper option](#) is used instead. The `joint.routers.normal` [function](#) is used by default.

Connector

Connectors take an array of [link route points](#) and generate SVG path commands so that the link can be rendered. The `connector` property of a link can be accessed with the `link.connector()` [function](#).

A collection of pre-made [connectors](#) is provided inside the JointJS library in the `joint.connectors` namespace.

```

link.connector('rounded', {
  raw: true,
  radius: 20
});

```

If a connector is not provided, the `defaultConnector` [paper option](#) is used instead. The `joint.connectors.normal` [function](#) is used by default.

Note that the modular architecture of JointJS allows mixing-and-matching connectors with routers as desired; for example, a link may be specified to use the `jumpover` connector on top of the `manhattan` router:

```

var link = new joint.shapes.standard.Link();
link.source(rect);
link.target(rect2);
link.router('manhattan');
link.connector('jumpover');

```

dia.Link.labels

JointJS supports adding labels on links. One link can have multiple labels, and each label can have different properties. For a quick introduction to link labels, see our [tutorial](#)

Properties recognized by JointJS are summarized in the following TypeScript-like schema:

```

{
  markup?: string | Array<{
    tagName: SVGElement,
    selector?: string
  }>,
  size?: {
    width?: number,

```

```

    height?: number
  },
  attrs?: {
    [key: selector]: {
      [key: SVG attribute | JointJS attribute]: any
    } | null
  },
  position?: number | {
    distance: number,
    offset?: number | { x: number, y: number },
    angle?: number,
    args?: {
      absoluteDistance?: boolean,
      reverseDistance?: boolean,
      absoluteOffset?: boolean,
      keepGradient?: boolean,
      ensureLegibility?: boolean
    }
  }
}
}

```

Default Label

The `markup`, `size`, `attrs` and `position` objects defined on individual labels may be supplemented by properties from one of a hierarchy of `defaultLabel` objects, each of them following the same schema as above.

- The highest precedence is held by the `defaultLabel` object provided when [creating a new link instance](#).
- The second-highest precedence is held by the `defaultLabel` object provided when [defining a custom Link type](#).
- Lower tiers of precedence are followed through the ancestors of the Link type (due to the way type extensions work in JointJS). Effectively, this identifies the closest ancestor's `defaultLabel` object, if any.

If a `defaultLabel` object is found in one of the above sources, its `markup`, `size`, `attrs` and `position` properties act as a template into which the individually-specified label properties are deep-merged as appropriate. In addition, the properties of the highest-precedence `defaultLabel` object may prove to be important on their own; for example, the `linkView.addLabel()` [shortcut function](#) is only capable of adding a default label to links.

Markup

The `markup` property specifies the markup of the label. It can be provided in three ways:

- As a parsable SVG (e.g. `'<rect /><text />'`).
- As a JSON array (e.g. `[{ tagName: 'rect', selector: 'body' }, { tagName: 'text', selector: 'label' }]`).
- Or as a `joint.util.svg` [ES6 tag template](#):

```

joint.util.svg`
  <rect @selector="body"/>

```

```
<text @selector="label"/>
```

Note that the latter two options allow the user to specify custom selectors for the individual SVGElements; these can then be used for targeting elements within the `attrs` [property](#).

If no `markup` property is provided on the individual label, but a `defaultLabel` is found on the link according to the `defaultLabel` [hierarchy](#), then the label markup is taken from the `defaultLabel.markup` property.

However, if no `markup` property is provided on the individual label, and no `defaultLabel` is found on the link according to the `defaultLabel` [hierarchy](#) (as may be the case when extending from out-of-the-box Link types like `joint.dia.Link` or `joint.shapes.standard.Link` without providing one's own `defaultLabel` object), then [built-in default Link markup](#) is used to maintain backwards compatibility. That object defines `markup` as a JSON array with a `<rect>` SVGElement (`'rect'` selector) under a `<text>` SVGElement (`'text'` selector).

Size

The `size` property specifies the dimensions of the label. It may be defined as an object with a `width` and/or `height` properties. These values are then used as the reference size for all `calc()` [attribute expressions](#) within the link's `attrs` object (this may be overwritten for an individual subelement with the `ref` property to choose a different subelement as a reference).

If a `defaultLabel` object was found on the link (according to the `defaultLabel` [hierarchy](#)), then the label's own `size` properties are deep-merged into it as appropriate.

There is no built-in `defaultLabel.size` object; if no `defaultLabel` is found, and no `size` is provided on the individual label, then no `size` is assigned to that label. (That causes `calc()` [attribute expressions](#) to assign `0` as the value of all their variables - which may be unexpected - unless the expression is used alongside a `ref` [property](#).)

<i>object</i>	width	Optional. The width of the label.
	height	Optional. The height of the label.

Styling

The `attrs` property is an object where the keys are CSS selectors (referring to custom selectors or SVGElements specified in `markup` - e.g. the `body` selector in the [above markup example](#)). They are expected to contain objects that specify native SVG attributes and/or JointJS [special attributes](#) (e.g. `fill`), alongside the value to be assigned (e.g. `'white'`).

If a `defaultLabel` object was found on the link (according to the `defaultLabel` [hierarchy](#)), then the label's own `attrs` properties are deep-merged into it as appropriate. For example, this may prove useful when you just need to change a handful of attributes for one specific label (e.g. `attrs: { body: { stroke: 'black' } }`). (Note that selectors defined in `defaultLabel.markup` can also be referenced from within a specific label's `attrs` .)

If the built-in default markup is [applied on an individual label](#) (i.e. no `markup` property was provided, and no `defaultLabel.markup` was found), then [several additional built-in default attributes](#) need to be automatically

applied for reasons of backwards compatibility. The individual label's `attrs` (if any) are deep-merged with `defaultLabel.attrs` (if found) as appropriate, and the resulting object is then deep-merged with the built-in default `attrs` as appropriate. (Note that this is the only case in which the absence of one property - `markup` / `defaultLabel.markup` - influences another property - `attrs` / `defaultLabel.attrs`.)

Position

Finally, the `position` property specifies the position of the label relative to the SVG path of the link. It may be defined as a number or as an object with `distance` and optionally `offset` and `args` properties.

The built-in default position (`{ distance: 0.5 }`) is used to maintain backwards compatibility. The individual label's `position` (if any) is deep-merged with `defaultLabel.position` (if found in the `defaultLabel.hierarchy`) as appropriate, and the resulting object is then deep-merged with the built-in default `position` as appropriate. (Note that the built-in default object is always merged in with the other objects, which is a difference from how the built-in defaults of `markup` and `attrs` properties are treated.)

number	If the distance is in the <code>[0, 1]</code> range (inclusive), then the position of the label is defined as a percentage of the total length of the <code>.connection</code> path (the normalized length). For example, passing the number <code>0.5</code> positions the label to the middle of the <code>.connection</code> path.			
	If the distance is larger than <code>1</code> (exclusive), the label will be positioned <code>distance</code> pixels away from the beginning of the path along the <code>.connection</code> path.			
	If the distance is a negative number, the label will be positioned <code>distance</code> pixels away from the end of the path along the <code>.connection</code> path.			
object	If <code>position</code> is specified as an object, it may have three properties:			
	distance	A number specifying the distance of the label along the <code>.connection</code> path. Same as above.		
	offset	Optional. Either a number or an object with <code>x</code> and <code>y</code> properties. <table><tr><td rowspan="2">number</td><td>If the offset is a positive number, displace the label perpendicularly to the right (in the direction of the <code>.connection</code> path) in the paper local coordinate system. (An offset of <code>0</code> is the default; it means no offset in either direction.)</td></tr><tr><td>If the offset is a negative number, displace the label perpendicularly to the left (in the direction of the <code>.connection</code> path) in the paper local coordinate system.</td></tr></table>	number	If the offset is a positive number, displace the label perpendicularly to the right (in the direction of the <code>.connection</code> path) in the paper local coordinate system. (An offset of <code>0</code> is the default; it means no offset in either direction.)
number	If the offset is a positive number, displace the label perpendicularly to the right (in the direction of the <code>.connection</code> path) in the paper local coordinate system. (An offset of <code>0</code> is the default; it means no offset in either direction.)			
	If the offset is a negative number, displace the label perpendicularly to the left (in the direction of the <code>.connection</code> path) in the paper local coordinate system.			

	<table><tr><td rowspan="2"><i>object</i></td><td>x</td><td>Offset the label by <code>x</code> in the paper local coordinate system.</td></tr><tr><td>y</td><td>Offset the label by <code>y</code> in the paper local coordinate system.</td></tr></table>	<i>object</i>	x	Offset the label by <code>x</code> in the paper local coordinate system.	y	Offset the label by <code>y</code> in the paper local coordinate system.					
<i>object</i>	x		Offset the label by <code>x</code> in the paper local coordinate system.								
	y	Offset the label by <code>y</code> in the paper local coordinate system.									
angle	Optional. A number specifying the rotation of the label along the <code>.connection</code> path, in degrees. Angles 0-180 cause the label to rotate clockwise, angles 360-180 cause the label to rotate couterclockwise. This rotation is added on top of any rotation due to the <code>args.keepGradient</code> option. (Note that the <code>args.ensureLegibility</code> option is evaluated only after all these calculations are finished.)										
args	Optional. An object with boolean options for the <code>linkView.getLabelPosition()</code> <u>function</u>. <table><tr><td>absoluteDistance</td><td>If user moves the label, record <code>distance</code> as an absolute number. (Relative distances are used by default.)</td></tr><tr><td>reverseDistance</td><td>If <code>absoluteDistance</code> is set and user moves the label, record <code>distance</code> as a negative absolute number. (Positive distances are used by default.)</td></tr><tr><td>absoluteOffset</td><td>If user moves the label, record <code>offset</code> as an object with absolute <code>x</code> and <code>y</code> coordinates. (Relative offsets are used by default.)</td></tr><tr><td>keepGradient</td><td>Adjust the base rotation angle of the label (i.e. the “zero angle” before rotation due to the <code>angle</code> setting is applied) to match the angle of incline of the path at given <code>distance</code>.</td></tr><tr><td>ensureLegibility</td><td>If <code>keepGradient</code> is set and the label ends up rotated so that its text is upside-down, rotate the label by additional 180 degrees so it becomes legible for the users.</td></tr></table>	absoluteDistance	If user moves the label, record <code>distance</code> as an absolute number. (Relative distances are used by default.)	reverseDistance	If <code>absoluteDistance</code> is set and user moves the label, record <code>distance</code> as a negative absolute number. (Positive distances are used by default.)	absoluteOffset	If user moves the label, record <code>offset</code> as an object with absolute <code>x</code> and <code>y</code> coordinates. (Relative offsets are used by default.)	keepGradient	Adjust the base rotation angle of the label (i.e. the “zero angle” before rotation due to the <code>angle</code> setting is applied) to match the angle of incline of the path at given <code>distance</code> .	ensureLegibility	If <code>keepGradient</code> is set and the label ends up rotated so that its text is upside-down, rotate the label by additional 180 degrees so it becomes legible for the users.
absoluteDistance	If user moves the label, record <code>distance</code> as an absolute number. (Relative distances are used by default.)										
reverseDistance	If <code>absoluteDistance</code> is set and user moves the label, record <code>distance</code> as a negative absolute number. (Positive distances are used by default.)										
absoluteOffset	If user moves the label, record <code>offset</code> as an object with absolute <code>x</code> and <code>y</code> coordinates. (Relative offsets are used by default.)										
keepGradient	Adjust the base rotation angle of the label (i.e. the “zero angle” before rotation due to the <code>angle</code> setting is applied) to match the angle of incline of the path at given <code>distance</code> .										
ensureLegibility	If <code>keepGradient</code> is set and the label ends up rotated so that its text is upside-down, rotate the label by additional 180 degrees so it becomes legible for the users.										

dia.Link.prototype.addTo

`link.addTo(graph)`

Add the link to the `graph` (an instance of `joint.dia.Graph`). This is equivalent to calling `graph.addCell(link)`.

dia.Link.prototype.appendLabel

`link.appendLabel(label [, opt])`

Add a new `label` at the end of the `labels` array.

dia.Link.prototype.attr

`link.attr(attrs, [, opt])`

Set presentation attributes ([SVG](#) and [JointJS](#) attributes) on subelements. This is a method analogous to `attr` method of `joint.dia.Element`. The keys of the `attrs` object are selectors ([JSON Markup Selector](#) or CSS Selector) matching the SVG element the link consists of. The values are objects containing SVG attributes and their values. `attrs` object will be mixed with `attrs` property of the `link` model. This is a convenient way of rewriting only some of the attributes of the SVG elements. For overwriting all attributes of all SVG elements, use `link.set('attrs', attrs)`.

```
link.attr({
  // selector as defined in JSON markup
  line: {
    stroke: 'red',
    targetMarker: {
      type: 'circle',
      r: 5
    }
  }
});
```

`element.attr(path, value [, opt])`

An alternative call using a string path and a value:

```
link.attr('line/stroke', 'red');
link.attr(['line', 'targetMarker', 'type'], 'path');
```

`element.attr([path])`

Get attribute value defined by a path. If no path provided the whole `attrs` object is returned.

```
var color = link.attr('line/stroke');
var targetMarkerType = link.attr(['line', 'targetMarker', 'type']);
```

dia.Link.prototype.clone

link.clone()

Returns a new instance of the link with identical attributes.

dia.Link.prototype.connector

link.connector()

Return a shallow copy of the `connector` property of the link.

For backwards compatibility, if there is no connector, the function also checks whether the legacy `smooth` property is set on the link and returns `{ name: smooth }` if it is.

link.connector(connector [, opt])

Set the `connector` of the link.

If the `connector` argument is an object, it is expected to have the form `{ name: connectorName, args?: connectorArgs }`. Here `connectorName` is expected to match either the name of a built-in connector or the name of a custom connector.

If the `connector` argument is a function, it is expected to define a custom connector with the signature `function(sourcePoint, targetPoint, vertices, connectorArgs, linkView)` that returns a string representing the SVG path data that will be used to render the link.

link.connector(connectorName [, connectorArgs, opt])

Set the `connector` of the link to have the value `{ name: connectorName, args: connectorArgs }`.

The `connectorName` string is expected to match either the name of a built-in connector or the name of a custom connector. The `connectorArgs` parameter is optional.

dia.Link.prototype.disconnect

link.disconnect()

Disconnect the link from its `source` and `target` elements. The `source` and `target` then become a point at `[0,0]` .

dia.Link.prototype.findView

```
link.findView(paper)
```

Find view (`joint.dia.LinkView`) for the link model in the `paper` . This is a shortcut to the equivalent call `paper.findViewByModel(link)` .

dia.Link.prototype.getAncestors

```
link.getAncestors()
```

Return an array of all the ancestors of this link starting from the immediate parent all the way up to the most distant ancestor.

dia.Link.prototype.getBBox

```
link.getBBox()
```

Return the bounding box of the link model, represented as a `g.Rect` .

This method creates the bounding box from the link polyline (see the `link.getPolyline` [function](#)). This simplified heuristic may cause portions of curved links to protrude out of the reported bounding box. See the documentation of the `linkView.getBBox` [function](#) for details about the differences.

dia.Link.prototype.getPolyline

```
link.getPolyline()
```

Return a polyline that approximates the shape of the link. The polyline is created from the two link endpoints, with link vertices in between.

dia.Link.prototype.getSourceCell

```
link.getSourceCell()
```

Return the source cell (element or link) of the link, or `null` if the element's source is a point.

dia.Link.prototype.getSourceElement

`link.getSourceElement()`

Return the source element of the link or `null` if there is none.

dia.Link.prototype.getSourcePoint

`link.getSourcePoint()`

Return the source point of the link.

- If the link endpoint is pinned to a specific point on the paper, return that point.
- If the link is connected to another link, return the midpoint of that link's polyline.
- If the link is connected to a port on an element, return the center of that port.
- If the link is connected to an element, return the center of that element.

dia.Link.prototype.getTargetCell

`link.getTargetCell()`

Return the target cell (element or link) of the link, or `null` if the element's target is a point.

dia.Link.prototype.getTargetElement

`link.getTargetElement()`

Return the target element of the link or `null` if there is none.

dia.Link.prototype.getTargetPoint

`link.getTargetPoint()`

Return the target point of the link.

- If the link endpoint is pinned to a specific point on the paper, return that point.
- If the link is connected to another link, return the midpoint of that link's polyline.
- If the link is connected to a port on an element, return the center of that port.

- If the link is connected to an element, return the center of that element.

dia.Link.prototype.getTransitions

`link.getTransitions()`

Return an array of all active transitions (their paths).

dia.Link.prototype.hasLabels

`link.hasLabels()`

Return `true` if the link has at least one label. Return `false` otherwise.

dia.Link.prototype.hasLoop

`link.hasLoop([opt])`

Return `true` if this link is a “loop link”. In a “loop link” `source` and `target` are equal.

If `opt.deep` is `true`, the notion of a “loop link” is extended to a deep hierarchy. For example, if the link connects a parent element with one of its embedded elements, the link is considered a “loop link”, as well.

dia.Link.prototype.insertLabel

`link.insertLabel(index, label [, opt])`

Add a new `label` at `index`. Pass index `-1` to add the label at the end of the labels array.

dia.Link.prototype.insertVertex

`link.insertVertex(index, vertex [, opt])`

Add a new `vertex` at `index`. Pass index `-1` to add the vertex at the end of the vertices array.

dia.Link.prototype.isElement

link.isElement()

Always returns `false`. The reason the method is here is that both `joint.dia.Element` and `joint.dia.Link` inherit from `joint.dia.Cell`. This method is useful if you don't know what the cell is. Calling `cell.isElement()` is equivalent to `cell instanceof joint.dia.Element`. Example:

```
var cell = graph.getCell(myId)
if (cell.isElement()) {
  // Do something if the cell is an element.
}
```

dia.Link.prototype.isEmbeddedIn

link.isEmbeddedIn(element [, opt])

Return `true` if the link is embedded in an element `element`.

If `opt.deep` is `false`, only direct parentage will be checked. `opt.deep` is `true` by default.

dia.Link.prototype.isLink

link.isLink()

Always returns `true`. The reason the method is here is that both `joint.dia.Element` and `joint.dia.Link` inherit from `joint.dia.Cell`. This method is useful if you don't know what the cell is. Calling `cell.isLink()` is equivalent to `cell instanceof joint.dia.Link`. Example:

```
var cell = graph.getCell(myId)
if (cell.isLink()) {
  // Do something if the cell is a link.
}
```

dia.Link.prototype.label

link.label(index)

Return the label at `index`.

link.label(index, properties [, opt])

Update properties of the label at `index`. By default, the new properties are merged into the old ones; pass the `{ rewrite: true }` option along to disregard old properties.

Example usage:

```
link.label(0, {
  markup: [
    {
      tagName: 'rect',
      selector: 'body'
    }, {
      tagName: 'text',
      selector: 'label'
    }
  ],
  attrs: {
    body: {
      fill: 'white' // white background
    },
    label: {
      text: 'my label', // text to show
      fill: 'blue' // blue text
    }
  },
  position: {
    distance: 0.5, // midway on the connection path
    offset: {
      x: 10, // 10 local x units to the right
      y: -5 // 5 local y units above
    },
    angle: 45, // rotate by 45 degrees clockwise
    args: {
      keepGradient: true, // auto-rotate by path slope at distance
      ensureLegibility: true // auto-rotate label if upside-down
    }
  }
});
```

Note that all labels are stored in an array on the link model under the attribute `labels`. Use the `link.labels` [function](#) to access the array.

dia.Link.prototype.labels

`link.labels()`

Return a shallow copy of labels array.

```
link.labels(labelsArray [, opt])
```

Set array of labels on the link.

dia.Link.prototype.prop

```
link.prop(properties)
```

Set properties, possibly nested, on the element model. This is equivalent to the [attr\(\)](#) method but this time for custom data properties.

```
link.prop('name/first', 'John')
link.prop('name/first') // 'John'
link.prop({ name: { first: 'John' } })
// Nested arrays are supported too:
link.prop('mylist/0/data/0/value', 50)
link.prop({ mylist: [ { data: [ { value: 50 } ] } ] })
```

As you can see, this is the exact same method as the [joint.dia.Element.prop\(\)](#) method.

To overwrite attributes, enable rewrite mode by adding `{ rewrite: true }` as the 3rd argument. This differs from the default behaviour which is to merge our properties.

```
link.prop('custom/state/isVisible', true);
link.prop('custom/state', { isActive: false }, { rewrite: true });

// Output from link.toJSON();
// We can see our attributes have been overwritten
{
  "type": "link",
  "source": { "x": 10, "y": 10 },
  "target": { "x": 100, "y": 100 },
  "id": "4ddf9c16-649b-40b3-96b3-7150711da955",
  "custom": {
    "state": {
      "isActive": false
    }
  },
  "attrs": {}
}
```

When changing model attributes via [prop\(\)](#) or [attr\(\)](#), some useful information is passed along with the change event in JointJS. `propertyPath`, `propertyValue`, and `propertyPathArray` are all values which can be accessed when updating the model. This can prove useful if for some reason you need to listen to a specific attribute change.

```
graph.on('change', (cell, opt) => {
  console.log(opt);
  // --> {propertyPath: 'attrs/line/targetMarker/fill', propertyValue:
  'cornflowerblue', propertyPathArray: Array(4)}

  if ('attrs' in cell.changed) {
    console.log(opt.propertyPathArray, 'was changed');
    // --> ['attrs', 'line', 'targetMarker', 'fill'] 'was changed'
  }
});

link.prop('attrs/line/targetMarker/fill', 'cornflowerblue');
```

Advanced: Pass `{ isolate: true }` if the property change does not affect the connected links. Typically, changing the link color has zero effect on attached links. By default, the link and all connected links are updated.

```
link.attr(['line', 'stroke'], 'red', { isolate: true });
```

dia.Link.prototype.remove

```
link.remove()
```

Remove the link from the graph.

dia.Link.prototype.removeAttr

```
link.removeAttr(path [, opt])
```

Remove a previously set attribute from the link. `path` can either be a string that specifies the path to the (possibly nested) attribute to be removed, or an array of more paths. The associated link view makes sure the link gets re-rendered properly.

If `opt` is passed, it can contain data that is passed over to the event listeners for the `change:attrs` event triggered on the link itself and also on the graph the link is in.

dia.Link.prototype.removeLabel

```
link.removeLabel(index [, opt])
```

Remove `label` at `index`. Pass index `-1` to remove the last label of the labels array.

dia.Link.prototype.removeVertex

`link.removeVertex(index [, opt])`

Remove vertex at index. Pass index `-1` to remove the last vertex of the vertices array.

dia.Link.prototype.reparent

`link.reparent()`

Automatically find and set the best parent element for the link so that when the parent element is moved, the link and all its vertices are moved too. The best parent is determined as the common ancestor of the source and target elements of the link. Useful for hierarchical diagrams. See the DEVS demo on how this can be used.

dia.Link.prototype.router

`link.router()`

Return a shallow copy of the `router` property of the link.

For backwards compatibility, if there is no router, the function also checks whether the legacy `manhattan` property is set on the link and returns `{ name: orthogonal }` if it is.

`link.router(router [, opt])`

Set the `router` of the link.

If the `router` argument is an object, it is expected to have the form `{ name: routerName, args?: routerArgs }`. Here `routerName` is expected to match either the name of a built-in router or the name of a custom router.

If the `router` argument is a function, it is expected to define a custom router with the signature `function(vertices, routerArgs, linkView)` that returns an array of route points.

`link.router(routerName [, routerArgs, opt])`

Set the `router` of the link to have the value `{ name: routerName, args: routerArgs }`.

The `routerName` string is expected to match either the name of a built-in router or the name of a custom router. The `routerArgs` parameter is optional.

dia.Link.prototype.scale

link.[scale](#)(sx, sy, origin [, opt])

Scales the link's points (vertices) relative to the given origin.

dia.Link.prototype.source

link.[source](#)()

Return a shallow copy of the `source` property of the link.

If the beginning of the link is connected to an element, an object in the format `{ id: elementID }` is returned. If the beginning of the link is specified as a point instead (the link is “pinned” to the paper at that point), an object in the format `{ x: sourceX, y: sourceY }` is returned. Furthermore, any additional arguments of the source are returned alongside those properties.

If you need to be sure that a Cell is returned (and not a Point), use the `link.getSourceCell` [function](#) instead. If you need an Element, use the `link.getSourceElement` [function](#).

link.[source](#)(source [, opt])

Set the `source` of the link.

If the link is to be connected to an element, send an object in the format `{ id: element.id }` (or a `joint.dia.Element` object). If the link is to be pinned to the paper, send an object in the format `{ x: sourceX, y: sourceY }` (or a `g.Point` object).

```
link.source(rect);
link.source({ id: rect.id });

link.source(new g.Point(100, 100));
link.source({ x: 100, y: 100 });
```

Additional options may be provided to further specify the behavior of link at the source:

```
link.source(rect, {
  selector: 'body',
  anchor: {
    name: 'bottomLeft',
    args: {
      dx: 20,
      dy: -10
    }
  }
});
```

```

    }
  }
});

```

More information can be found in [link source documentation](#).

dia.Link.prototype.stopTransitions

```
link.stopTransitions([path])
```

Stops all running transitions. If parameter `path` is provided, it will stop only transitions specified by this path.

dia.Link.prototype.target

```
link.target()
```

Return a shallow copy of the `target` property of the link.

If the ending of the link is connected to an element, an object in the format `{ id: elementID }` is returned. If the ending of the link is specified as a point instead (the link is “pinned” to the paper at that point), an object in the format `{ x: targetX, y: targetY }` is returned. Furthermore, any additional arguments of the target are returned alongside those properties.

If you need to be sure that a `Cell` is returned (and not a `Point`), use the `link.getTargetCell` [function](#) instead. If you need an `Element`, use the `link.getTargetElement` [function](#).

```
link.target(target [, opt])
```

Set the `target` of the link.

If the link is to be connected to an element, send an object in the format `{ id: element.id }` (or a `joint.dia.Element` object). If the link is to be pinned to the paper, send an object in the format `{ x: targetX, y: targetY }` (or a `g.Point` object).

```

link.target(rect);
link.target({ id: rect.id });

link.target(new g.Point(100, 100));
link.target({ x: 100, y: 100 });

```

Additional options may be provided to further specify the behavior of link at the target:

```

link.target(rect, {
  selector: 'body',

```

```
    anchor: {
      name: 'bottomLeft',
      args: {
        dx: 20,
        dy: -10
      }
    }
  });
```

More information can be found in [link target documentation](#).

dia.Link.prototype.toBack

link.[toBack\(\)](#)

Move the link so it is behind all other cells (elements/links). This is a method analogous to [toBack](#) method of `Joint.dia.Element`.

dia.Link.prototype.toFront

link.[toFront\(\)](#)

Move the element so it is on top of all other cells (elements/links). This is a method analogous to [toFront](#) method of `Joint.dia.Element`.

dia.Link.prototype.toJSON

link.[toJSON\(\)](#)

Return a copy of the link's attributes for JSON serialization. This is a method analogous to [toJSON](#) method of `Joint.dia.Element`.

dia.Link.prototype.transition

link.[transition](#)(path, value [, opt])

Allows changing of the link properties gradually over a period of time. This is method is analogous to the [transition](#) method of `joint.dia.Element`.

```
link.transition('target', { x: 250, y: 250 }, {
  delay: 100,
  duration: 500,
  timingFunction: joint.util.timing.bounce,
  valueFunction: joint.util.interpolate.object
});
// will start changing the link target coordinates in 100ms, for period of 500ms and
performing a bounce effect.
```

dia.Link.prototype.translate

`link.translate(tx, ty [, opt])`

Translate the link vertices (and source and target if they are points) by `tx` pixels in the x-axis and `ty` pixels in the y-axis.

If `opt` object is passed, it can contain data that is passed over the the event listeners for the change event on the link or graph.

dia.Link.prototype.vertex

`link.vertex(index)`

Return the vertex at `index`.

`link.vertex(index, vertex [, opt])`

Update the vertex at `index` with value of `vertex`. By default the new value is merged into the old value. Pass the `{ rewrite: true }` option along to disregard the old value.

A vertex has only two properties, `x` and `y`. It may be defined as a `g.Point` object.

Example vertex:

```
link.vertex(0, {
  x: 100,
  y: 200
});
```

Note that all vertices are stored in an array on the link model under the attribute `vertices`. Use the `link.vertices` [function](#) to access the array.

dia.Link.prototype.vertices

link.vertices()

Return a shallow copy of vertices array.

link.vertices(verticesArray [, opt])

Set array of vertices on the link.

dia.LinkView

The view for the `joint.dia.Link` model. It inherits from `joint.dia.CellView` and is responsible for:

- Rendering a link inside a paper
- Handling the link's pointer events
- Providing various methods for working with the link (visually)

To find the view associated with a specific link model, use the `paper.findViewByModel()` [method](#):

```
var linkView = paper.findViewByModel(link);
```

Custom LinkView

It is possible to use a custom default link view for all your links in a paper. This can be set up via the `linkView` option on the [paper object](#). Several options in `joint.dia.LinkView` may be overridden in a custom LinkView:

- **shortLinkLength** - if link is shorter than the length specified, the size of link tools is scaled down by half. Defaults to `105`.
- **doubleLinkTools** - render link tools on both ends of the link (only if the link is longer than `longLinkLength`). Defaults to `false`.
- **longLinkLength** - if length is longer than the length specified and `doubleLinkTools: true`, render link tools on both ends of the link. Defaults to `155`.
- **linkToolsOffset** - the offset of link tools from the beginning of the link. Defaults to `40`.
- **doubleLinkToolsOffset** - the offset of duplicate link tools from the end of the link, if length is longer than `longLinkLength` and `doubleLinkTools: true`. Defaults to `65`.

A custom LinkView type may also override default LinkView event handlers, or provide new ones. It may be necessary to modify the [interactive paper option](#) to prevent interference from builtin event handlers (most notably `vertexAdd` which listens for pointerdown events).

Example:

```

var CustomLinkView = joint.dia.LinkView.extend({
  // custom interactions:
  pointerdblclick: function(evt, x, y) {
    this.addVertex(x, y);
  },
  contextmenu: function(evt, x, y) {
    this.addLabel(x, y);
  },

  // custom options:
  options: joint.util.defaults({
    doubleLinkTools: true,
  }, joint.dia.LinkView.prototype.options)
});

var paper = new joint.dia.Paper({
  //...
  linkView: CustomLinkView,
  interactive: { vertexAdd: false } // disable default vertexAdd interaction
});

```

dia.LinkView.prototype.addLabel

linkView.[addLabel](#)(x, y [, angle, opt])

Add a new [default label](#) to the link at the (x,y) coordinates provided. See also the link.[appendLabel\(\)](#) [function](#).

linkView.[addLabel](#)(point [, angle, opt])

Add a new [default label](#) to the link at the coordinates specified by point. See also the link.[appendLabel\(\)](#) [function](#).

In either case, this method uses the linkView.[getLabelPosition\(\)](#) [function](#) to determine the new label's position. By default, position.distance is recorded relative to connection length (as a number in the [0,1] range), and position.offset is set relative to the connection (as a number). This behavior may be changed by providing an opt object with some of the accepted boolean flags:

- absoluteDistance: true records distance absolutely (as distance from beginning of link)
- reverseDistance: true switches distance to be calculated from end of link, if absoluteDistance
- absoluteOffset: true records offset absolutely (as x and y from connection)

The angle parameter, if provided, is saved as position.angle attribute inside the returned object. Two additional flags, which may be passed in the opt object, provide more control over label rotation:

- keepGradient: true - adjust the rotation of the label to match the angle of incline of the path at position.distance

- `ensureLegible: true` - if the label text ends up being upside-down, rotate the label by additional 180 degrees to ensure that the text stays legible, if `keepGradient`

The `opt` object passed to the label is recorded as `label.position.args`. The label uses these options during subsequent `labelMove` interactions.

This function is useful within custom `linkView` event listener definitions:

```
var CustomLinkView = joint.dia.LinkView.extend({
  contextmenu: function(evt, x, y) {
    this.addLabel(x, y, 45, {
      absoluteDistance: true,
      reverseDistance: true, // applied only when absoluteDistance is set
      absoluteOffset: true,
      keepGradient: true,
      ensureLegibility: true // applied only when keepGradient is set
    });
  }
});

var paper = new joint.dia.Paper({
  // ...
  linkView: CustomLinkView,
  interactive: { vertexAdd: false } // disable default vertexAdd interaction
});
```

dia.LinkView.prototype.addTools

`linkView.addTools(toolsView)`

Add the provided `toolsView` (of the `joint.dia.ToolsView` type) to the link view.

Adding a tools view to a link view is the last (third) step in the process of setting up link tools on a link view:

```
// 1) creating link tools
var verticesTool = new joint.linkTools.Vertices();
var segmentsTool = new joint.linkTools.Segments();
var boundaryTool = new joint.linkTools.Boundary();

// 2) creating a tools view
var toolsView = new joint.dia.ToolsView({
  name: 'basic-tools',
  tools: [verticesTool, segmentsTool, boundaryTool]
});

// 3) attaching to a link view
```

```
var linkView = link.findView(paper);
linkView.addTools(toolsView);
```

Every link view we want to attach to requires its own tools view object (`ToolsView` objects are automatically reassigned to the last link view they are added to). Similarly, every tools view we create requires its own set of tools (`ToolView` objects are automatically reassigned to the last `toolsView.tools` array they were made part of).

The link tools are added in the visible state. Use the `linkView.hideTools` [function](#) if this behavior is not desirable (e.g. if you want the link tools to appear in response to user interaction). Example:

```
linkView.addTools(toolsView);
linkView.hideTools();

paper.on('link:mouseenter', function(linkView) {
    linkView.showTools();
});

paper.on('link:mouseleave', function(linkView) {
    linkView.hideTools();
});
```

Note that the above example may not work as expected if `toolsView` includes the `SourceArrowhead` [tool](#) and/or the `TargetArrowhead` [tool](#) - the link tools view might not be hidden when the link is reconnected to a topic. See our [link tools tutorial](#) for more information.

dia.LinkView.prototype.addVertex

`linkView.addVertex(x, y)`

Add a new [default vertex](#) to the link at the coordinates provided, and let the linkView automatically determine the index of the new vertex in the vertices array. If you need to add the vertex at a custom index, use the `Link.addVertex()` [function](#) instead.

This method uses the `linkView.getVertexIndex()` [function](#) to determine the index of the new vertex in the `vertices` array. The linkView checks the distance of vertices in the `link.vertices` array from the beginning of path and compares it to the distance of the added vertex. The new vertex is inserted before the first farther vertex in the `vertices` array.

This function is useful within custom `linkView` event listener definitions. The following example adds a new vertex on a double click event, instead of a pointerdown event (which is the default behavior):

```
var CustomLinkView = joint.dia.LinkView.extend({
    pointerdblclick: function(evt, x, y) {
        this.addVertex(x, y);
    }
});
```

```
var paper = new joint.dia.Paper({
  // ...
  linkView: CustomLinkView,
  interactive: { vertexAdd: false } // disable default vertexAdd interaction
});
```

dia.LinkView.prototype.findLabelNode

linkView.[findLabelNode](#)(index)

Return the root *SVGElement* of the label at given **index**. If label at **index** doesn't exist, **null** is returned.

linkView.[findLabelNode](#)(index, selector)

Return an *SVGElement* referenced by **selector** of the label at given **index**. If there is no label at **index** or no *SVGElement* was found by the **selector**, **null** is returned.

dia.LinkView.prototype.getBBox

linkView.[getBBox](#)([opt])

Return a bounding box of the link view.

If `opt.useModelGeometry` option is set to **true**, the resulting bounding box is calculated based on the dimensions of the link model. (This means that a simplified polyline is used and portions of curved links may be “sticking out” of the reported bounding box.) This behavior is similar to the `link.getBBox` [function](#) – but the `linkView` function transforms the bounding box to match `joint.dia.Paper` translation and scaling.

dia.LinkView.prototype.getClosestPoint

linkView.[getClosestPoint](#)(point)

Return the point on the connection that lies closest to `point`.

dia.LinkView.prototype.getClosestPointLength

linkView.[getClosestPointLength](#)(point)

Return the length of the connection up to the point that lies closest to `point`.

dia.LinkView.prototype.getClosestPointRatio

`linkView.getClosestPointRatio(point)`

Return the ratio (normalized length) of the connection up to the point that lies closest to `point`. The returned value lies in the interval `[0,1]` (inclusive).

dia.LinkView.prototype.getConnection

`linkView.getConnection()`

Return a geometric representation of the connection (instance of `g.Path`).

dia.LinkView.prototype.getConnectionLength

`linkView.getConnectionLength()`

Return a cached total length of the connection in pixels.

dia.LinkView.prototype.getConnectionSubdivisions

`linkView.getConnectionSubdivisions()`

Return a cached array of segment subdivision arrays of the connection. (See `g.Path.prototype.getSegmentSubdivisions()` [documentation](#) for more information.)

dia.LinkView.prototype.getLabelCoordinates

`linkView.getLabelCoordinates(labelPosition)`

Return the `x` and `y` coordinates based on the provided `labelPosition` object.

See `link.label()` [documentation](#) for more information about the `position` object.

An object in the format `{ x: number, y: number }` is returned.

dia.LinkView.prototype.getLabelPosition

`linkView.getLabelPosition(x, y [, angle, opt])`

Return a label position object based on the `x` and `y` coordinates provided.

The function translates the provided coordinates and `angle` into an object with three fields:

- `distance` - the distance (following the line) of the point on the line that is closest to point `x,y` .
- `offset` - the distance between the closest point and the point `x,y` .
- `angle` - the angle of the label relative to the connection line, as determined by the `angle` parameter, or `0` if `angle` was not specified.

By default, `position.distance` is calculated as relative to connection length (as a number in the `[0,1]` range that records the length ratio), and `position.offset` is calculated as relative to the connection (as a number recording the perpendicular distance). The user may change this behavior by providing an `opt` object with some of the following accepted boolean flags:

- `absoluteDistance: true` - record distance absolutely (as absolute distance from beginning of link, a positive number)
- `reverseDistance: true` - if `absoluteDistance: true`, record distance absolutely from end of link (as a negative number)
- `absoluteOffset: true` - record offset absolutely (as `x` and `y` distance from closest point)

Please note that if the `absoluteOffset` flag is not set, label can only be placed/moved in the area that is reachable by lines perpendicular to the link (that is, the label can never be moved beyond link endpoints).

Two additional flags, which may be passed in the `opt` object, provide control over label rotation:

- `keepGradient: true` - adjust the rotation of the label to match the angle of incline of the path at `position.distance`
- `ensureLegible: true` - if the label text ends up being upside-down, rotate the label by additional 180 degrees to ensure that the text stays legible, if `keepGradient`

The `opt` object passed to the label is recorded as `label.position.args` . The label uses these options during subsequent `labelMove` interactions.

An object in the following format is returned:

```
{
  distance: number,
  offset: number | { x: number, y: number },
  angle: number,
  args?: {
    absoluteDistance?: boolean,
    reverseDistance?: boolean, // applied only when absoluteDistance is set
    absoluteOffset?: boolean,
```

```

        keepGradient?: boolean,
        ensureLegible?: boolean // applied only when keepGradient is set
    }
}

```

See `link.label()` [documentation](#) for more information about the `position` object.

This function can be used to add a custom label to the `link.labels` [array](#), in situations when the `linkView.addLabel()` [function](#) is not sufficient. For example:

```

var CustomLinkView = joint.dia.LinkView.extend({
  contextmenu: function(evt, x, y) {
    var idx = -1; // add at end of `labels`
    var label = {
      markup: '<g class="label"><circle /><path /></g>',
      attrs: {
        circle: {
          r: 15,
          fill: 'lightgray',
          stroke: 'black',
          strokeWidth: 2
        },
        path: {
          d: 'M 0 -15 0 -35 20 -35',
          stroke: 'black',
          strokeWidth: 2,
          fill: 'none'
        }
      },
      position: this.getLabelPosition(x, y, 45, {
        absoluteOffset: true,
        keepGradient: true,
        ensureLegible: true
      })
    }
    this.model.addLabel(idx, label);
  }
});

var paper = new joint.dia.Paper({
  // ...
  linkView: CustomLinkView,
  interactive: { vertexAdd: false } // disable default vertexAdd interaction
});

```


dia.LinkView.prototype.getNodeBBBox

`elementView.getNodeBBBox(magnet)`

Return the bounding box of the SVGElement provided as `magnet` (model of this link view).

Use the `paper.localToPaperRect` [function](#) to transform the returned bounding box to match the paper's translation and scaling.

dia.LinkView.prototype.getNodeUnrotatedBBBox

`elementView.getNodeUnrotatedBBBox(magnet)`

Return the unrotated bounding box of the SVGElement provided as `magnet` (model of this link view).

Use the `paper.localToPaperRect` [function](#) to transform the returned bounding box to match the paper's translation and scaling.

dia.LinkView.prototype.getPointAtLength

`linkView.getPointAtLength(length)`

Return the point on the path that lies `length` away from the beginning of the connection.

dia.LinkView.prototype.getPointAtRatio

`linkView.getPointAtRatio(ratio)`

Return the point on the path that lies `ratio` (normalized length) away from the beginning of the connection.

dia.LinkView.prototype.getSerializedConnection

`linkView.getSerializedConnection()`

Return a cached path data of the connection (the value of the `'d'` SVGPathElement attribute).

dia.LinkView.prototype.getTangentAtLength

`linkView.getTangentAtLength(length)`

Return a line tangent to the path at point that lies `length` away from the beginning of the connection.

dia.LinkView.prototype.getTangentAtRatio

`linkView.getTangentAtRatio(ratio)`

Return a line tangent to the path at point that lies `ratio` (normalized length) away from the beginning of the connection.

dia.LinkView.prototype.getVertexIndex

`linkView.getVertexIndex(x, y)`

Return the `vertices` array index at which to insert a new vertex with the provided `x` and `y` coordinates.

The `linkView` finds the point on the connection that lies closest to the point `x, y`. Then, the `linkView` iterates over the vertices in the `link.vertices` array until one is found that lies farther than the identified closest point. The index of the farther point is returned.

The returned index can be used as the first argument of the `link.addVertex` function.

dia.LinkView.prototype.hasTools

`linkView.hasTools()`

Return `true` if this link view has a tools view attached.

`linkView.hasTools(name)`

Return `true` if this link view has a tools view of the provided `name` attached.

dia.LinkView.prototype.hideTools

`linkView.hideTools()`

Hide all tools attached to this link view.

dia.LinkView.prototype.removeRedundantLinearVertices

`linkView.removeRedundantLinearVertices([opt])`

Remove any redundant vertices from the model's `vertices` array. Return the number of removed vertices.

A vertex is considered “redundant” if it lies on the straight line formed by the vertex that immediately precedes it and the vertex that immediately follows it. (Intuitively speaking, a vertex is considered “redundant” if its removal from the link does not change the shape of the link.)

dia.LinkView.prototype.removeTools

`linkView.removeTools()`

Remove the tools view attached to this link view.

dia.LinkView.prototype.requestConnectionUpdate

`linkView.requestConnectionUpdate()`

Schedule an update of the connection (anchors, connection points, route and connector) of this link view in the next animation frame (for paper `async` mode) or run immediately (for paper `sync` mode). Useful for links with an automatic router (such as `manhattan`) if the route has to be recalculated due to another element in the graph changing its position.

dia.LinkView.prototype.sendToken

`linkView.sendToken(token [, opt, callback])`

Send a token along the link. `token` is an SVG element (or Vectorizer element) that will be animated along the link path for `opt.duration` milliseconds (default is 1000ms). The `callback` function will be called once the token reaches the end of the link path.

`opt.direction` specifies whether the animation should be played forwards (`'normal'` - from the link source to target, the default) or backwards (`'reverse'`).

Use `opt.connection` to specify the `SVGPathElement` for the token to follow. It expects a string selector, e.g. `'connection'`.

```
// Send an SVG circle token along the link.
var vCircle = V('circle', { r: 7, fill: 'green' });
link.findView(paper).sendToken(vCircle, { duration: 500, direction: 'reverse' },
function() {
  console.log('animation end');
});
```

Note that in the code above, we use the [Vectorizer](#) mini-library to create the SVG `circle` element.

See the [Petri Net simulator demo](#) for a full working example.

dia.LinkView.prototype.showTools

```
linkView.showTools()
```

Show all tools attached to this link view.

dia.LinkView.prototype.sourceAnchor

```
linkView.sourceAnchor
```

A prototype variable. Contains a `g.Point` that records the position of the source anchor of the link, as determined by the [anchor function](#) specified on the link's [source](#).

dia.LinkView.prototype.sourceBBox

```
linkView.sourceBBox
```

A prototype variable. Contains a `g.Rect` that records the position and dimensions of the bounding box of the source magnet (element/subelement/port), as determined by the link's [source definition](#).

If the source is defined as a point, a `g.Rect` is returned that records the position of the point and has the dimensions (1,1).

dia.LinkView.prototype.sourcePoint

```
linkView.sourcePoint
```

A prototype variable. Contains a `g.Point` that records the position of the source connection point of the link, as determined by the [connection point function](#) specified on the link's [source](#).

dia.LinkView.prototype.targetAnchor

linkView.targetAnchor

A prototype variable. Contains a g.Point that records the position of the target anchor of the link, as determined by the anchor function specified on the link's target.

dia.LinkView.prototype.targetBBox

linkView.targetBBox

A prototype variable. Contains a g.Rect that records the position and dimensions of the bounding box of the target magnet (element/subelement/port), as determined by the link's target definition.

If the target is defined as a point, a g.Rect is returned that records the position of the point and has the dimensions (1,1).

dia.LinkView.prototype.targetPoint

linkView.targetPoint

A prototype variable. Contains a g.Point that records the position of the target connection point of the link, as determined by the connection point function specified on the link's target.

dia.Paper.constructor

`joint.dia.Paper` is the view for the `joint.dia.Graph` model. It inherits from Backbone View. Accepts an options object in its constructor with numerous settings.

When a paper is associated with a graph, the paper makes sure that all the cells added to the graph are automatically rendered.

```
var graph = new joint.dia.Graph
var paper = new joint.dia.Paper({
  el: $('#paper'),
  width: 600,
  height: 400,
  gridSize: 10,
  model: graph
});
```

```
var rect = new joint.shapes.basic.Rect({
  position: { x: 50, y: 70 },
  size: { width: 100, height: 40 }
});

graph.addCell(rect);
```

Paper automatically handles this change and renders the rectangle to the SVG document that it internally holds.

dia.Paper.events

The following list contains events that you can react on in a paper:

pointerdblclick	Triggered when pointer is double-clicked on a target (a dblclick event is detected). The callback function is passed cellView , evt , x and y as arguments.	
	cell:pointerdblclick	Triggered when pointer is double-clicked on a cell.
	link:pointerdblclick	Triggered when pointer is double-clicked on a link.
	element:pointerdblclick	Triggered when pointer is double-clicked on an element.
	blank:pointerdblclick	Triggered when pointer is double-clicked on the paper outside any cell. The callback function is passed evt , x and y as arguments.
pointerclick	Triggered when pointer is left-clicked on a target without pointer movement (a click or touchend event is detected). Occurs alongside pointerdown and pointerup events. The callback function is passed cellView , evt , x and y as arguments.	
	cell:pointerclick	Triggered when pointer is clicked on a cell.
	link:pointerclick	Triggered when pointer is clicked on a link.
	element:pointerclick	Triggered when pointer is clicked on an element.
	blank:pointerclick	Triggered when pointer is clicked on the paper outside any cell.

		The callback function is passed <code>evt</code> , <code>x</code> and <code>y</code> as arguments.							
contextmenu	Triggered when pointer is right-clicked on a target. This trigger is fired on a <code>mousedown</code> event and it does not depend on different implementations of the <code>contextmenu</code> event. The callback function is passed <code>cellView</code> , <code>evt</code> , <code>x</code> and <code>y</code> as arguments. <table><tr><td>cell:contextmenu</td><td>Triggered when pointer is right-clicked on a cell.</td></tr><tr><td>link:contextmenu</td><td>Triggered when pointer is right-clicked on a link.</td></tr><tr><td>element:contextmenu</td><td>Triggered when pointer is right-clicked on an element.</td></tr><tr><td>blank:contextmenu</td><td> Triggered when pointer is right-clicked on the paper outside any cell. The callback function is passed <code>evt</code> , <code>x</code> and <code>y</code> as arguments. </td></tr></table>	cell:contextmenu	Triggered when pointer is right-clicked on a cell.	link:contextmenu	Triggered when pointer is right-clicked on a link.	element:contextmenu	Triggered when pointer is right-clicked on an element.	blank:contextmenu	Triggered when pointer is right-clicked on the paper outside any cell. The callback function is passed <code>evt</code> , <code>x</code> and <code>y</code> as arguments.
cell:contextmenu	Triggered when pointer is right-clicked on a cell.								
link:contextmenu	Triggered when pointer is right-clicked on a link.								
element:contextmenu	Triggered when pointer is right-clicked on an element.								
blank:contextmenu	Triggered when pointer is right-clicked on the paper outside any cell. The callback function is passed <code>evt</code> , <code>x</code> and <code>y</code> as arguments.								
pointerdown	Triggered when pointer is pressed down on a target (a <code>mousedown</code> or <code>touchstart</code> event is detected). The paper also starts delegating respective <code>pointermove</code> and <code>pointerup</code> events (including their touch counterparts; see below). The callback function is passed <code>cellView</code> , <code>evt</code> , <code>x</code> and <code>y</code> as arguments. <table><tr><td>cell:pointerdown</td><td>Triggered when pointer is pressed down on a cell.</td></tr><tr><td>link:pointerdown</td><td>Triggered when pointer is pressed down on a link.</td></tr><tr><td>element:pointerdown</td><td>Triggered when pointer is pressed down on an element.</td></tr><tr><td>blank:pointerdown</td><td> Triggered when pointer is pressed down on the paper outside any cell. The callback function is passed <code>evt</code> , <code>x</code> and <code>y</code> as arguments. </td></tr></table>	cell:pointerdown	Triggered when pointer is pressed down on a cell.	link:pointerdown	Triggered when pointer is pressed down on a link.	element:pointerdown	Triggered when pointer is pressed down on an element.	blank:pointerdown	Triggered when pointer is pressed down on the paper outside any cell. The callback function is passed <code>evt</code> , <code>x</code> and <code>y</code> as arguments.
cell:pointerdown	Triggered when pointer is pressed down on a cell.								
link:pointerdown	Triggered when pointer is pressed down on a link.								
element:pointerdown	Triggered when pointer is pressed down on an element.								
blank:pointerdown	Triggered when pointer is pressed down on the paper outside any cell. The callback function is passed <code>evt</code> , <code>x</code> and <code>y</code> as arguments.								
pointermove	Triggered when pointer is moved over a target while pressed down (a <code>mousemove</code> or <code>touchmove</code> event is detected). The callback function is passed <code>cellView</code> , <code>evt</code> , <code>x</code> and <code>y</code> as arguments. <table><tr><td>cell:pointermove</td><td>Triggered when pointer is moved over a cell.</td></tr></table>	cell:pointermove	Triggered when pointer is moved over a cell.						
cell:pointermove	Triggered when pointer is moved over a cell.								

	<table><tr><td>link:pointermove</td><td>Triggered when pointer is moved over a link.</td></tr><tr><td>element:pointermove</td><td>Triggered when pointer is moved over an element.</td></tr><tr><td>blank:pointermove</td><td>Triggered when pointer is moved over the paper outside any cell. The callback function is passed <code>evt</code> , <code>x</code> and <code>y</code> as arguments.</td></tr></table>	link:pointermove	Triggered when pointer is moved over a link.	element:pointermove	Triggered when pointer is moved over an element.	blank:pointermove	Triggered when pointer is moved over the paper outside any cell. The callback function is passed <code>evt</code> , <code>x</code> and <code>y</code> as arguments.		
link:pointermove	Triggered when pointer is moved over a link.								
element:pointermove	Triggered when pointer is moved over an element.								
blank:pointermove	Triggered when pointer is moved over the paper outside any cell. The callback function is passed <code>evt</code> , <code>x</code> and <code>y</code> as arguments.								
pointerup	Triggered when pointer is released on a target after being pressed down (a <code>mouseup</code> or <code>touchend</code> event is detected). The callback function is passed <code>cellView</code> , <code>evt</code> , <code>x</code> and <code>y</code> as arguments. <table><tr><td>cell:pointerup</td><td>Triggered when pointer is released on a cell.</td></tr><tr><td>link:pointerup</td><td>Triggered when pointer is relased on a link.</td></tr><tr><td>element:pointerup</td><td>Triggered when pointer is released on an element.</td></tr><tr><td>blank:pointerup</td><td>Triggered when pointer is released on the paper outside any cell. The callback function is passed <code>evt</code> , <code>x</code> and <code>y</code> as arguments.</td></tr></table> Calling <code>evt.stopPropagation()</code> prevents triggering a subsequent <code>pointerclick</code> event.	cell:pointerup	Triggered when pointer is released on a cell.	link:pointerup	Triggered when pointer is relased on a link.	element:pointerup	Triggered when pointer is released on an element.	blank:pointerup	Triggered when pointer is released on the paper outside any cell. The callback function is passed <code>evt</code> , <code>x</code> and <code>y</code> as arguments.
cell:pointerup	Triggered when pointer is released on a cell.								
link:pointerup	Triggered when pointer is relased on a link.								
element:pointerup	Triggered when pointer is released on an element.								
blank:pointerup	Triggered when pointer is released on the paper outside any cell. The callback function is passed <code>evt</code> , <code>x</code> and <code>y</code> as arguments.								
mouseover	Triggered when pointer begins to hover directly over a target. The callback function is passed <code>cellView</code> and <code>evt</code> as arguments. <table><tr><td>cell:mouseover</td><td>Triggered when pointer begins to hover directly over a cell.</td></tr><tr><td>link:mouseover</td><td>Triggered when pointer begins to hover directly over a link.</td></tr><tr><td>element:mouseover</td><td>Triggered when pointer begins to hover directly over an element.</td></tr></table>	cell:mouseover	Triggered when pointer begins to hover directly over a cell.	link:mouseover	Triggered when pointer begins to hover directly over a link.	element:mouseover	Triggered when pointer begins to hover directly over an element.		
cell:mouseover	Triggered when pointer begins to hover directly over a cell.								
link:mouseover	Triggered when pointer begins to hover directly over a link.								
element:mouseover	Triggered when pointer begins to hover directly over an element.								

	<table> <tr> <td>blank:mouseover</td><td> Triggered when pointer begins to hover over the <code>svg</code> element of the paper outside any cell. The callback function is passed <code>evt</code> as argument. </td></tr> </table>	blank:mouseover	Triggered when pointer begins to hover over the <code>svg</code> element of the paper outside any cell. The callback function is passed <code>evt</code> as argument.						
blank:mouseover	Triggered when pointer begins to hover over the <code>svg</code> element of the paper outside any cell. The callback function is passed <code>evt</code> as argument.								
mouseout	Triggered when pointer ceases to hover directly over a target. The callback function is passed <code>cellView</code> and <code>evt</code> as arguments. <table> <tr> <td>cell:mouseout</td><td>Triggered when pointer ceases to hover directly over a cell.</td></tr> <tr> <td>link:mouseout</td><td>Triggered when pointer ceases to hover directly over a link.</td></tr> <tr> <td>element:mouseout</td><td>Triggered when pointer ceases to hover directly over an element.</td></tr> <tr> <td>blank:mouseout</td><td> Triggered when pointer ceases to hover over the <code>svg</code> element of the paper outside any cell. The callback function is passed <code>evt</code> as argument. </td></tr> </table>	cell:mouseout	Triggered when pointer ceases to hover directly over a cell.	link:mouseout	Triggered when pointer ceases to hover directly over a link.	element:mouseout	Triggered when pointer ceases to hover directly over an element.	blank:mouseout	Triggered when pointer ceases to hover over the <code>svg</code> element of the paper outside any cell. The callback function is passed <code>evt</code> as argument.
cell:mouseout	Triggered when pointer ceases to hover directly over a cell.								
link:mouseout	Triggered when pointer ceases to hover directly over a link.								
element:mouseout	Triggered when pointer ceases to hover directly over an element.								
blank:mouseout	Triggered when pointer ceases to hover over the <code>svg</code> element of the paper outside any cell. The callback function is passed <code>evt</code> as argument.								
mouseenter	Triggered when pointer enters the area above a target. The callback function is passed <code>cellView</code> and <code>evt</code> as arguments. <table> <tr> <td>cell:mouseenter</td><td>Triggered when pointer enters the area above a cell.</td></tr> <tr> <td>link:mouseenter</td><td>Triggered when pointer enters the area above a link.</td></tr> <tr> <td>element:mouseenter</td><td>Triggered when pointer enters the area above an element.</td></tr> <tr> <td>paper:mouseenter</td><td> Triggered when pointer enters the area of the paper (including paper border, if present). The callback function is passed <code>evt</code> as argument. </td></tr> </table>	cell:mouseenter	Triggered when pointer enters the area above a cell.	link:mouseenter	Triggered when pointer enters the area above a link.	element:mouseenter	Triggered when pointer enters the area above an element.	paper:mouseenter	Triggered when pointer enters the area of the paper (including paper border, if present). The callback function is passed <code>evt</code> as argument.
cell:mouseenter	Triggered when pointer enters the area above a cell.								
link:mouseenter	Triggered when pointer enters the area above a link.								
element:mouseenter	Triggered when pointer enters the area above an element.								
paper:mouseenter	Triggered when pointer enters the area of the paper (including paper border, if present). The callback function is passed <code>evt</code> as argument.								
mouseleave	Triggered when pointer leaves the area above a target. The callback function is passed <code>cellView</code> and <code>evt</code> as arguments. <table> <tr> <td>cell:mouseleave</td><td>Triggered when pointer leaves the area above a cell.</td></tr> <tr> <td>link:mouseleave</td><td>Triggered when pointer leaves the area above a link.</td></tr> </table>	cell:mouseleave	Triggered when pointer leaves the area above a cell.	link:mouseleave	Triggered when pointer leaves the area above a link.				
cell:mouseleave	Triggered when pointer leaves the area above a cell.								
link:mouseleave	Triggered when pointer leaves the area above a link.								

	<table> <tr> <td>element:mouseleave</td><td>Triggered when pointer leaves the area above an element.</td></tr> <tr> <td>paper:mouseleave</td><td> Triggered when pointer leaves the area of the paper (including paper border, if present). The callback function is passed <code>evt</code> as argument. </td></tr> </table>	element:mouseleave	Triggered when pointer leaves the area above an element.	paper:mouseleave	Triggered when pointer leaves the area of the paper (including paper border, if present). The callback function is passed <code>evt</code> as argument.				
element:mouseleave	Triggered when pointer leaves the area above an element.								
paper:mouseleave	Triggered when pointer leaves the area of the paper (including paper border, if present). The callback function is passed <code>evt</code> as argument.								
mousewheel	Triggered when mouse wheel is rotated while pointer is on a target. The callback function is passed <code>cellView</code>, <code>evt</code>, <code>x</code>, <code>y</code> and <code>delta</code> as arguments. <table> <tr> <td>cell:mousewheel</td><td>Triggered when mouse wheel is rotated while the pointer is on a cell.</td></tr> <tr> <td>link:mousewheel</td><td>Triggered when mouse wheel is rotated while the pointer is on a link.</td></tr> <tr> <td>element:mousewheel</td><td>Triggered when mouse wheel is rotated while the pointer is on an element.</td></tr> <tr> <td>blank:mousewheel</td><td> Triggered when mouse wheel is rotated while the pointer is on the paper outside any cell. The callback function is passed <code>evt</code>, <code>x</code>, <code>y</code> and <code>delta</code> as arguments. </td></tr> </table>	cell:mousewheel	Triggered when mouse wheel is rotated while the pointer is on a cell.	link:mousewheel	Triggered when mouse wheel is rotated while the pointer is on a link.	element:mousewheel	Triggered when mouse wheel is rotated while the pointer is on an element.	blank:mousewheel	Triggered when mouse wheel is rotated while the pointer is on the paper outside any cell. The callback function is passed <code>evt</code>, <code>x</code>, <code>y</code> and <code>delta</code> as arguments.
cell:mousewheel	Triggered when mouse wheel is rotated while the pointer is on a cell.								
link:mousewheel	Triggered when mouse wheel is rotated while the pointer is on a link.								
element:mousewheel	Triggered when mouse wheel is rotated while the pointer is on an element.								
blank:mousewheel	Triggered when mouse wheel is rotated while the pointer is on the paper outside any cell. The callback function is passed <code>evt</code>, <code>x</code>, <code>y</code> and <code>delta</code> as arguments.								
pan	Triggered when a pan event is emitted while the pointer is on top of the target. Pan events are similar to mousewheel events, but while mousewheel events cover only the Y axis, pan events cover both X and Y axis. This is usually possible with touchpads / trackpad devices. The callback function is passed <code>evt</code>, <code>deltaX</code> and <code>deltaY</code> as arguments. <table> <tr> <td>paper:pan</td><td> Triggered when a pan event is emitted while the pointer is on the paper or any cell. The callback function is passed <code>evt</code>, <code>deltaX</code> and <code>deltaY</code> as arguments. </td></tr> </table>	paper:pan	Triggered when a pan event is emitted while the pointer is on the paper or any cell. The callback function is passed <code>evt</code>, <code>deltaX</code> and <code>deltaY</code> as arguments.						
paper:pan	Triggered when a pan event is emitted while the pointer is on the paper or any cell. The callback function is passed <code>evt</code>, <code>deltaX</code> and <code>deltaY</code> as arguments.								
pinch	Triggered when a pinch event is emitted while the pointer is on top of the target. This is usually possible with touchpads / trackpad devices. The callback function is passed <code>evt</code>, <code>x</code>, <code>y</code> and <code>scale</code> as arguments. <table> <tr> <td>paper:pinch</td><td>Triggered when a pinch event is emitted while the pointer is on the paper or any cell.</td></tr> </table>	paper:pinch	Triggered when a pinch event is emitted while the pointer is on the paper or any cell.						
paper:pinch	Triggered when a pinch event is emitted while the pointer is on the paper or any cell.								

		The callback function is passed <code>evt</code> , <code>x</code> <code>y</code> and <code>scale</code> as arguments.
magnet	Triggered when interacting with a magnet target.	
	The callback function is passed <code>elementView</code> , <code>evt</code> , <code>magnetSVGElement</code> , <code>x</code> , <code>y</code> as arguments.	
	<code>element:magnet:pointerdown</code>	Triggered when pointer is pressed down on a magnet element.
	<code>element:magnet:pointermove</code>	Triggered when pointer is moved after <code>pointerdown</code> on the magnet element.
	<code>element:magnet:pointerup</code>	Triggered when pointer is released after <code>pointerdown</code> on the magnet element.
	<code>element:magnet:pointerclick</code>	Triggered when pointer is clicked on an element magnet and no link was created yet from this magnet (controlled by <code>magnetThreshold</code> option). Calling <code>evt.stopPropagation()</code> prevents triggering <code>cell:pointerclick</code> and <code>element:pointerclick</code> events.
	<code>element:magnet:pointerdblclick</code>	Triggered when pointer is double-clicked on an element magnet. Calling <code>evt.stopPropagation()</code> prevents triggering <code>cell:pointerdblclick</code> and <code>element:pointerdblclick</code> events.
	<code>element:magnet:contextmenu</code>	Triggered when pointer is right-clicked on an element magnet. Calling <code>evt.stopPropagation()</code> prevents triggering <code>cell:contextmenu</code> and <code>element:contextmenu</code> events.
cell:highlight	Triggered when the <code>cellView.highlight</code> <u>method</u> is called on a link or element view.	
	The callback function is invoked with the following arguments:	
	• cellView - highlighted CellView • node - highlighted node (SVGElement) of the CellView • options - options used when <code>cellView.highlight(node, options)</code> was called. The <code>type</code> property is added to the option object and specifies what type of	

highlighting interactions caused the highlight.

```
// Disable built-in highlighting
paper.options.highlighting = false;
// Handle each type of the highlight manually
paper.on('cell:highlight', (cellView, node, options) => {
  switch (options.type) {
    case: joint.dia.CellView.Highlighting.CONNECTING: {
      joint.highlighters.stroke.add(cellView, node, 'id1', {
        attrs: { 'stroke': 'blue' } });
      break;
    }
    case: joint.dia.CellView.Highlighting.EMBEDDING: {
      joint.highlighters.stroke.add(cellView, node, 'id2', {
        attrs: { 'stroke': 'red' } });
      break;
    }
    default: {
      joint.highlighters.stroke.add(cellView, node, 'id3');
      break;
    }
  }
});
```

cell:unhighlight

Triggered when the `cellView.unhighlight` method is called on a link or element view.

The callback function is invoked with the following arguments:

- **cellView** - highlighted CellView
- **node** - highlighted node (SVGElement) of the CellView
- **options** - options used when `cellView.unhighlight(node, options)` was called. The `type` property is added to the option object and specifies what type of highlighting interactions caused the unhighlight.

```
// Disable built-in highlighting
paper.options.highlighting = false;
// Handle each type of the highlight manually
paper.on('cell:unhighlight', (cellView, _node, options) => {
  switch (options.type) {
    case: joint.dia.CellView.Highlighting.CONNECTING: {
      joint.dia.Highlighter.remove(cellView, 'id1');
      break;
    }
    case: joint.dia.CellView.Highlighting.EMBEDDING: {
      joint.dia.Highlighter.remove(cellView, 'id2');
      break;
    }
  }
});
```

	<pre> } default: { joint.dia.Highlighter.remove(cellView, 'id3'); break; } } }); </pre>
cell:highlight:invalid	Triggered when a highlighter makes an attempt to highlight a node, that doesn't exist on the CellView (e.g. a port was highlighted and now it is removed). The callback function is passed <code>cellView</code>, <code>highlighterId</code> and <code>highlighter</code> as arguments. <pre> paper.on('cell:highlight:invalid', (cellView, id) => { // Remove the invalid highlighter joint.dia.HighlighterView.remove(cellView, id); }); element1.addPort({ id: 'port1' }); const elementView1 = element1.findView(paper); const highlighter1 = joint.highlighters.mask.add(elementView1, { port: 'port1' }, 'highlighter-id')); // The following line will trigger the event with (elementView1, 'highlighter-id', highlighter1) arguments. element1.removePorts(); </pre>
link:connect	Triggered when a link is connected to a cell. The event is triggered after the user reconnects a link. The callback function is passed <code>linkView</code>, <code>evt</code>, <code>elementViewConnected</code>, <code>magnet</code> and <code>arrowhead</code> as arguments.
link:disconnect	Triggered when a link is disconnected from a cell. The event is triggered after the user reconnects a link. The callback function is passed <code>linkView</code>, <code>evt</code>, <code>elementViewDisconnected</code>, <code>magnet</code> and <code>arrowhead</code> as arguments.
link:snap:connect	Triggered when a link is connected to a cell. The event (or multiple events) can be triggered while the user is reconnecting a link and <u>snapLinks</u> option is enabled. The callback function is passed <code>linkView</code>, <code>evt</code>, <code>elementViewConnected</code>, <code>magnet</code> and <code>arrowhead</code> as arguments.

link:snap:disconnect	Triggered when a link is disconnected from a cell. The event (or multiple events) can be triggered while the user is reconnecting a link and snapLinks option is enabled. The callback function is passed <code>linkView</code>, <code>evt</code>, <code>elementViewDisconnected</code>, <code>magnet</code> and <code>arrowhead</code> as arguments.
render:done	Triggered when all scheduled updates are done (i.e. all scheduled batches have finished).
[custom]	Custom cell event can be triggered on pointerdown with Event attribute. Calling <code>evt.stopPropagation()</code> prevents triggering all subsequent events.

An example of a simple `blank:pointerdown` event listener:

```
paper.on('blank:pointerdown', function(evt, x, y) {
  alert('pointerdown on a blank area in the paper.')
})
```

Consecutive `pointerdown`, `pointermove` and `pointerup` events can share information through the `evt.data` object:

```
// Create a new link by dragging
paper.on({
  'blank:pointerdown': function(evt, x, y) {
    var link = new joint.dia.Link();
    link.set('source', { x: x, y: y });
    link.set('target', { x: x, y: y });
    link.addTo(this.model);
    evt.data = { link: link, x: x, y: y };
  },
  'blank:pointermove': function(evt, x, y) {
    evt.data.link.set('target', { x: x, y: y });
  },
  'blank:pointerup': function(evt) {
    var target = evt.data.link.get('target');
    if (evt.data.x === target.x && evt.data.y === target.y) {
      // remove zero-length links
      evt.data.link.remove();
    }
  }
});
```

[dia.Paper.prototype.checkViewport](#)

`paper.checkViewport()`

For every view in the paper, determine if it fits into the viewport specified by `paper.options.viewport` function. Views that fit (views for which that function returns `true`) are attached to the DOM; views that do not are detached.

While `async` papers do this automatically, synchronous papers require an explicit call to this method for this functionality to be applied. To show all views again, use `paper.dumpViews()` .

`paper.checkViewport(opt)`

If `opt.viewport` is provided, it is used as the callback function instead of `paper.options.viewport` . The format of the callback method is described in the [option's documentation](#), as are examples of usage.

In `async` papers, providing `opt.viewport` causes viewport to be temporarily recalculated according to the provided callback function. This may be useful in situations where you need to show different views depending on context (e.g. viewing a section of the diagram vs. printing all of it).

dia.Paper.prototype.clearGrid

`paper.clearGrid()`

Hide the current grid.

dia.Paper.prototype.clientOffset

`paper.clientOffset()`

Returns coordinates of the paper viewport, relative to the application's client area.

dia.Paper.prototype.clientToLocalPoint

`paper.clientToLocalPoint(p)`

Transform client coordinates represented by point `p` to the paper local coordinates. This is especially useful when you have a mouse event object and want coordinates inside the paper that correspond to event `clientX` and `clientY` point.

```
var localPoint1 = paper.clientToLocalPoint({ x: evt.clientX, y: evt.clientY });  
// alternative method signature  
var localPoint2 = paper.clientToLocalPoint(evt.clientX, evt.clientY);
```

dia.Paper.prototype.clientToLocalRect

`paper.clientToLocalRect(rect)`

Transform the rectangle `rect` defined in the client coordinate system to the local coordinate system.

```
var bcr = paper.svg.getBoundingClientRect();
var localRect1 = paper.clientToLocalRect({ x: bcr.left, y: bcr.top, width: bcr.width,
height: bcr.height });
// alternative method signature
var localRect2 = paper.clientToLocalRect(bcr.left, bcr.top, bcr.width, bcr.height);
// Move the element to the center of the paper viewport.
var localCenter = localRect1.center();
var elSize = element.size();
element.position(localCenter.x - elSize.width, localCenter.y - elSize.height);
```

dia.Paper.prototype.defineFilter

`paper.defineFilter(filterDefinition)`

Define an SVG filter for later reuse within the paper. The method returns the filter id and the `filterDefinition` must have the following form:

```
{
  name: <name of the filter>,
  args: <filter arguments>
}
```

Where `name` is the name of the filter. See below for the list of built-in filters. `args` is an object containing filter parameters. These parameters are dependent on the filter used and are described in the list below as well. Example usage:

```
var filterId = paper.defineFilter({
  name: 'dropShadow',
  args: {
    dx: 2,
    dy: 2,
    blur: 3
  }
});
```



```
svgNode.setAttribute('filter', 'url(#' + filterId + ');
```

The following is the list of built-in filters. All these filters are defined in the `joint.util.filter` namespace. This namespace can be extended simply by adding a new method to it with one argument, an object with filter parameters, returning a string representing the SVG filter definition.

- `blur`
 - `x` - horizontal blur
 - `y` - vertical blur [optional, if not defined `y` is the same as `x`]
- `dropShadow`
 - `dx` - horizontal shift
 - `dy` - vertical shift
 - `blur` - blur
 - `color` - color
 - `opacity` - opacity
- `grayscale`
 - `amount` - the proportion of the conversion. `1` is completely grayscale. `0` leaves the element unchanged.
- `sepia`
 - `amount` - the proportion of the conversion. `1` is completely sepia. `0` leaves the element unchanged.
- `saturate`
 - `amount` - the proportion of the conversion. `0` is completely un-saturated. `1` leaves the element unchanged.
- `hueRotate`
 - `angle` - the number of degrees around the color circle the input samples will be adjusted
- `invert`
 - `amount` - the proportion of the conversion. `1` is completely inverted. `0` leaves the element unchanged.
- `brightness`
 - `amount` - the proportion of the conversion. `0` makes the element completely black. `1` leaves the element unchanged.
- `contrast`
 - `amount` - the proportion of the conversion. `0` makes the element completely black. `1` leaves the element unchanged.

dia.Paper.prototype.defineGradient

`paper.defineGradient(gradientDefinition)`

Define an SVG gradient for later reuse within the paper. The method returns the gradient id and the `gradientDefinition` must be an object with the following properties:

Name	Type	Description

type	'linearGradient' 'radialGradient'	The linearGradient or radialGradient type of the gradient.
id	string	(optional) - A unique identifier of the gradient. The <i>id</i> is generated if none provided.
attrs	object	(optional) - Additional SVG attributes for the <i>SVGGradientElement</i> .
stops	array	An array of stops .

Where a `stop` is an object with the following properties:

Name	Type	Description
color	string	Indicates what color to use at that gradient stop.
offset	number string	The offset of the gradient stop.
opacity	number	(optional) - A number in the <i>[0..1]</i> range representing the transparency of the stop color.

```
const gradientId = paper.defineGradient({
  type: 'linearGradient',
  stops: [
    { offset: '0%', color: '#E67E22' },
    { offset: '20%', color: '#D35400' },
    { offset: '40%', color: '#E74C3C' },
    { offset: '60%', color: '#C0392B' },
    { offset: '80%', color: '#F39C12' }
  ]
});

svgNode.setAttribute('fill', `url(#${gradientId})`);
```

For an introduction to gradients, please refer to the tutorial on [Filters and Gradients](#).

dia.Paper.prototype.defineMarker

`paper.defineMarker(markerDefinition)`

Define an SVG marker for later reuse within the paper. The method returns the marker id. The `markerDefinition` parameter can be an object with the following properties:

Name	Type	Description
id	string	<i>(optional)</i> - A unique identifier of the marker. The <i>id</i> is generated if none provided.
attrs	object	<i>(optional)</i> - Additional SVG attributes for the <i>SVGMarkerElement</i> .
markup	string array	JSON or string <u>markup</u> of the marker content.

The coordinate system for defining path data within `marker` is as follows:

Point (x , y)	Relative position
0,0	At <code>marker-start</code> , <code>marker-end</code> or <code>marker-mid</code> point, as appropriate.
n,0	If <code>n > 0</code> , position along path direction (i.e. start -> end).
	If <code>n < 0</code> , position opposite to path direction (i.e. start <- end).
0,m	If <code>m > 0</code> , position to the right of path direction.
	If <code>m < 0</code> , position to the left of path direction.

An example of JSON markup with `id` specified:

```
paper.defineMarker({
  id: 'my-marker',
  markup: [{
    tagName: 'circle',
    attributes: {
      'cx': '6',
      'cy': '0',
      'r': '12',
      'fill': '#7b5cce'
    }
  }, {
    tagName: 'polygon',
    attributes: {
      'points': '0,0 6,6 12,0 6,-6',
      'fill': '#d63865',
      'stroke': '#fff'
    }
  }
]}
```

```

    }]
  });

  // Draw the shape at the start of a path
  svgPathNode.setAttribute('marker-start', `url(#my-marker)`);

```

The same example in string markup:

```

paper.defineMarker({
  id: 'my-marker',
  markup: `
    <circle cx="6" cy="0" r="12" fill="#7b5cce" />
    <polygon points="0,0 6,6 12,0 6,-6" fill="#d63865" stroke="#fff" />
  `,
});

// Draw the shape at the end of a path
svgPathNode.setAttribute('marker-start', `url(#my-marker)`);

```

Alternatively, defining a marker is possible via a flat object with the following properties:

Name	Type	Description
id	string	<i>(optional)</i> - A unique identifier of the marker. The <i>id</i> is generated if none provided.
type	string	<i>(optional)</i> - The type of the SVGElement to generate to represent the marker (<code>'path'</code> , <code>'circle'</code> , <code>'ellipse'</code> , <code>'rect'</code> , <code>'polyline'</code> and <code>'polygon'</code>). Default is <code>'path'</code> .
[SVGAttribute]	any	<i>(optional)</i> - A presentation SVG attribute (e.g <code>fill: 'red'</code>) to be applied on the generated SVGElement (see <code>type</code>).

The coordinate system for defining path data is the same as above.

An example of specifying marker content via a flat object and referring to it in code via its automatically-generated `id` :

```

const markerId = paper.defineMarker({
  type: 'path', // SVG Path
  fill: '#666',
  stroke: '#333',
  d: 'M 10 -10 0 0 10 10 Z'
});

// Draw an arrow at the start and the end of a path

```

```
svgPathNode.setAttribute('marker-start', `url(#${markerId})`);
svgPathNode.setAttribute('marker-end', `url(#${markerId})`);
```

dia.Paper.prototype.definePattern

paper.[definePattern](#)(patternDefinition)

Define an SVG pattern for later reuse within the paper. The method returns the pattern id and the `patternDefinition` must be an object with the following properties:

Name	Type	Description
id	string	<i>(optional)</i> - A unique identifier of the pattern. The <i>id</i> is generated if none provided.
attrs	object	<i>(optional)</i> - Additional SVG attributes for the <i>SVGGradientElement</i> .
markup	string array	String or JSON <u>Markup</u> of the pattern content.

```
const patternId = paper.definePattern({
  attrs: {
    width: 10,
    height: 10
  },
  markup: [{
    tagName: 'polygon',
    attributes: {
      points: '0,0 2,5 0,10 5,8 10,10 8,5 10,0 5,2'
    }
  }]
});

svgNode.setAttribute('fill', `url(#${patternId})`);
```

dia.Paper.prototype.drawBackground

paper.[drawBackground](#)(opt);

Sets the paper background defined by the `opt` object. Please see the [background](#) paper option for available configuration.

dia.Paper.prototype.drawGrid

`paper.drawGrid([opt])`

Draw visual grid lines on the paper. Possible options:

- **color** - the color of the grid line (hex, RGB, etc).
- **thickness** - the thickness of the grid line (pixels).

dia.Paper.prototype.dumpViews

`paper.dumpViews([opt])`

Add all paper views into the DOM and update them to make sure that the views reflect the cells' models.

Several extra arguments may be provided in the `opt` object:

batchSize	<i>number</i>	For async papers, how many views should there be per one asynchronous process? Default is 1000 .		
progress	<i>function</i>	Callback function that is called whenever a batch is finished processing. The callback function is provided with three arguments:		
		done	<i>boolean</i>	Are we done? Was this the last batch?
		processed	<i>number</i>	How far along are we? How many elements have been processed as part of a batch so far?
		total	<i>number</i>	How many views are there? How many elements in will have been processed as part of a batch when we are done?
viewport	<i>function</i>	Callback function to determine whether a given view should be added to the DOM. By default, <code>paper.options.viewport</code> is used, but you may provide a different callback function. The format of the callback method is described in the option's documentation , as are examples of usage.		

dia.Paper.prototype.findView

`paper.findView(element)`

Find a view (instance of `joint.dia.ElementView` or `joint.dia.LinkView`) associated with a DOM element in the paper. `element` can either be a DOM element, jQuery object or a CSS selector. Sometimes, it is useful to find a view object for an element in the DOM. This method finds the closest view for any subelement of a view element.

dia.Paper.prototype.findViewByModel

`paper.findViewByModel(model)`

Find a view (instance of `joint.dia.ElementView` or `joint.dia.LinkView`) associated with a model. `model` can either be an instance of `joint.dia.Element` or `joint.dia.Link`.

dia.Paper.prototype.findViewsFromPoint

`paper.findViewsFromPoint(point)`

Find views (instance of `joint.dia.ElementView`) under a certain point in the paper. `point` is an object with `x` and `y` properties. Returns an array of views whose bounding box contains `point`. Note that there can be more than one views as views might overlap. Note there is a difference between this method and the `joint.dia.Graph.findModelsFromPoint`. A bounding box of a view can be different than the area computed by an element model position and size. For example, if a `<text>` SVG element in the shape is positioned relatively and shifted down below the normal shape area (e.g. using the [JointJS special attributes](#)), the bounding box of the view will be bigger than that of the model.

dia.Paper.prototype.findViewsInArea

`paper.findViewsInArea(rect [, opt])`

Find views (instance of `joint.dia.ElementView`) in a certain area in the paper. `rect` is an object with `x`, `y`, `width` and `height` properties. Return an array of views whose bounding box intersects the `rect` rectangle. If `opt.strict` is `true`, return an array of views whose bounding box is contained within the `rect` rectangle (i.e. not only intersects it).

dia.Paper.prototype.fitToContent

`paper.fitToContent([opt])`

Expand or shrink the paper to fit the content inside it. The method returns the area (`g.Rect`) of the paper after fitting in local coordinates.

The list of all available options can be found in the documentation of the `paper.getFitToContentArea` [function](#). You can try many of these options interactively in the [paper demo](#).

This method might internally trigger the `"resize"` and `"translate"` events. These can be handled by listening on the paper object (`paper.on('resize', myHandler)`).

```
paper.fitToContent([gridWidth, gridHeight, padding, opt])
```

Deprecated usage. All parameters are expected to be passed inside the `opt` object.

dia.Paper.prototype.freeze

```
paper.freeze()
```

Freeze an `async` paper. In this state, the paper does not automatically re-render upon changes in the graph. This is useful when adding large numbers of cells.

Use `paper.unfreeze()` to take the paper back out of the frozen state and to reflect the changes made in the graph in the meantime (if any).

dia.Paper.prototype.getArea

```
paper.getArea()
```

Return a rectangle representing the area of the paper, in local units (without transformations).

Creates a rectangle object with `{ x: 0, y: 0 }` and with `width` and `height` values according to the `paper.getComputedSize()` function. Then uses the `paper.paperToLocalRect()` function to transform that rectangle into local units.

dia.Paper.prototype.getComputedSize

```
paper.getComputedSize()
```

Return an object containing `width` and `height` properties. It resolves any computation these property values may contain (e.g resolves `"100%"` to the actual client width).

dia.Paper.prototype.getContentArea

```
paper.getContentArea()
```

Return a rectangle representing the area occupied by paper content, in local units (without transformations).

If `opt.useModelGeometry` is set (default is `false`), occupied area is calculated from the dimensions of component cells' models (not views). This option is useful whenever one or more of the paper cells are not in the DOM (e.g. during asynchronous calls). However, the method uses a simplified heuristic that is different from the one used by the browser; the mapping between dimensions of cell views and dimensions of cells models is not necessarily one-to-one. (Most notably, ports are not included in model-reported occupied areas. In addition, the area occupied by links is constructed from link vertices, which may exclude portions of protruding Curveto and Arcto segments.)

dia.Paper.prototype.getContentBBox

`paper.getContentBBox(opt)`

Return the bounding box of the content inside the paper, in client units (as it appears on the screen).

If `opt.useModelGeometry` is set (default is `false`), the bounding box is calculated from the dimensions of component cells' models (not views). This option is useful whenever one or more of the paper cells are not in the DOM (e.g. during asynchronous calls). However, the method uses a simplified heuristic that is different from the one used by the browser; the mapping between dimensions of cell views and dimensions of cells models is not necessarily one-to-one. (Most notably, ports are not included in model-reported bounding boxes. In addition, the bounding boxes of links are constructed from link vertices, which may exclude portions of protruding Curveto and Arcto segments.)

dia.Paper.prototype.getFitToContentArea

`paper.getFitToContentArea([opt])`

Return the bounding box in the local coordinate system based on the position and size of the current content and options provided.

The function accepts an object with the following options:

- `opt.gridWidth` and `opt.gridHeight` – snap the resulting width and height of the paper to a grid defined by these dimensions.
- `opt.padding` – additional padding around the resulting paper. It may be specified as a number, in which case it represents the padding width on all sides of the paper. It may be an object of the form `{ top?: [number], right?: [number], bottom?: [number], left?: [number], vertical?: [number], horizontal?: [number] }`.
- `opt.allowNewOrigin` – should the origin of the resulting paper be adjusted to match the origin of the paper content? In addition to no value being set, three values are recognized by this option:
 - `'positive'`, `'negative'`, `'any'`.
 - By default, the method only recognizes the content at positive coordinates and puts the origin of resulting paper at the original point (0,0).
 - `'positive'` – the method only recognizes the content at positive coordinates and puts the origin of resulting paper at the origin of the content. (Still, if the content starts at negative coordinates in an axis, the resulting paper's origin will be assigned `0` in that axis.)
 - `'negative'` – the method only recognizes the content at negative coordinates and puts the origin of resulting paper at the origin of the content. (However, if the content starts at positive coordinates in an axis,

the resulting paper's origin will be assigned `0` in that axis.)

- `'any'` – the method recognizes all of the paper content. The origin of the resulting paper will be at the origin of the content.
- `opt.allowNegativeBottomRight` – can the bottom-right corner of the resulting paper have negative coordinates? By default, the paper area always contains point (0,0). i.e. `false`.

```
// Resize the paper to match the tight bounding box of the content regardless of its position.  
paper.fitToContent({  
  allowNewOrigin: 'any',  
  allowNegativeBottomRight: true  
});
```

- `opt.minWidth` and `opt.minHeight` – define the minimum width and height of the resulting paper after fitting it to content.
- `opt.maxWidth` and `opt.maxHeight` – define the maximum width and height of the resulting paper after fitting it to content.
- `opt.useModelGeometry` – should paper content area be calculated from cell models instead of views? The default is `false`. See the documentation of the `paper.getContentArea` [function](#) for more details.
- `opt.contentArea` – an object of the form `{ x: [number], y: [number], width: [number], height: [number] }` is the area representing the content bounding box in the local coordinate system that paper should be fitted to. By default `opt.contentArea` is `paper.getContentArea(opt)`.

dia.Paper.prototype.hasScheduledUpdates

`paper.hasScheduledUpdates()`

Return `true` if any changes were made to the graph which are not yet reflected in the paper. (Call the `updateViews` [function](#) to update these views.) Return `false` otherwise.

dia.Paper.prototype.hideTools

`paper.hideTools()`

Hide all tools attached to all element and link views on this paper.

dia.Paper.prototype.isDefined

`paper.isDefined(graphicalObjectId)`

Return **true** if there is a graphical object (gradient, filter, marker) with `graphicalObjectId` already defined within the paper. Return **false** otherwise.

dia.Paper.prototype.isFrozen

`paper.isFrozen()`

Return **true** if the paper is currently **frozen**. Return **false** otherwise.

dia.Paper.prototype.localToClientPoint

`paper.localToClientPoint(p)`

Transform the point `p` defined in the local coordinate system to the client coordinate system.

```
var rightMidPoint = element.getBBox().rightMiddle();
var clientPoint1 = paper.localToClientPoint(rightMidPoint);
// alternative method signature
var clientPoint2 = paper.localToClientPoint(rightMidPoint.x, rightMidPoint.y);
// Draw an HTML rectangle next to the element.
var div = document.createElement('div');
div.style.position = 'fixed';
div.style.background = 'red';
div.style.left = clientPoint1.x + 'px';
div.style.top = clientPoint1.y + 'px';
div.style.width = '40px';
div.style.height = '40px';
div.style.marginLeft = '10px';
div.style.marginTop = '-20px';
document.body.appendChild(div);
```

dia.Paper.prototype.localToClientRect

`paper.localToClientRect(rect)`

Transform the rectangle `rect` defined in local coordinate system to the client coordinate system.

```
var bbox = element.getBBox();
var clientRect1 = paper.localToClientRect(bbox);
// alternative method signature
var clientRect2 = paper.localToClientRect(bbox.x, bbox.y, bbox.width, bbox.height);
```

```

// Draw an HTML rectangle above the element.
var div = document.createElement('div');
div.style.position = 'fixed';
div.style.background = 'red';
div.style.left = clientRect1.x + 'px';
div.style.top = clientRect1.y + 'px';
div.style.width = clientRect1.width + 'px';
div.style.height = clientRect1.height + 'px';
paper.el.appendChild(div);

```

dia.Paper.prototype.localToPagePoint

`paper.localToPagePoint(p)`

Transform the point `p` defined in the local coordinate system to the page coordinate system.

```

var rightMidPoint = element.getBBox().rightMiddle();
var pagePoint1 = paper.localToPagePoint(rightMidPoint);
// alternative method signature
var pagePoint2 = paper.localToPagePoint(rightMidPoint.x, rightMidPoint.y);
// Draw an HTML rectangle next to the element.
var div = document.createElement('div');
div.style.position = 'absolute';
div.style.background = 'red';
div.style.left = pagePoint1.x + 'px';
div.style.top = pagePoint1.y + 'px';
div.style.width = '40px';
div.style.height = '40px';
div.style.marginLeft = '10px';
div.style.marginTop = '-20px';
document.body.appendChild(div);

```

dia.Paper.prototype.localToPageRect

`paper.localToPageRect(rect)`

Transform the rectangle `rect` defined in local coordinate system to the page coordinate system.

```

var bbox = element.getBBox();
var pageRect1 = paper.localToPageRect(bbox);
// alternative method signature
var pageRect2 = paper.localToPageRect(bbox.x, bbox.y, bbox.width, bbox.height);

```

```
// Draw an HTML rectangle above the element.
var div = document.createElement('div');
div.style.position = 'absolute';
div.style.background = 'red';
div.style.left = pageRect1.x + 'px';
div.style.top = pageRect1.y + 'px';
div.style.width = pageRect1.width + 'px';
div.style.height = pageRect1.height + 'px';
document.body.appendChild(div);
```

dia.Paper.prototype.localToPaperPoint

`paper.localToPaperPoint(p)`

Transform the point `p` defined in the local coordinate system to the paper coordinate system.

```
var rightMidPoint = element.getBBox().rightMiddle();
var paperPoint1 = paper.localToPaperPoint(rightMidPoint);
// alternative method signature
var paperPoint2 = paper.localToPaperPoint(rightMidPoint.x, rightMidPoint.y);
// Draw an HTML rectangle next to the element.
var div = document.createElement('div');
div.style.position = 'absolute';
div.style.background = 'red';
div.style.left = paperPoint1.x + 'px';
div.style.top = paperPoint1.y + 'px';
div.style.width = '40px';
div.style.height = '40px';
div.style.marginLeft = '10px';
div.style.marginTop = '-20px';
paper.el.appendChild(div);
```

dia.Paper.prototype.localToPaperRect

`paper.localToPaperRect(rect)`

Transform the rectangle `rect` defined in the local coordinate system to the paper coordinate system.

```
var bbox = element.getBBox();
var paperRect1 = paper.localToPaperRect(bbox);
// alternative method signature
var paperRect2 = paper.localToPaperRect(bbox.x, bbox.y, bbox.width, bbox.height);
```

```
// Draw an HTML rectangle above the element.
var div = document.createElement('div');
div.style.position = 'absolute';
div.style.background = 'red';
div.style.left = paperRect1.x + 'px';
div.style.top = paperRect1.y + 'px';
div.style.width = paperRect1.width + 'px';
div.style.height = paperRect1.height + 'px';
paper.el.appendChild(div);
```

dia.Paper.prototype.matrix

`paper.matrix([SVGMatrix])`

When called with no parameter, the method returns the current transformation matrix (instance of [SVGMatrix](#)) of the paper. It sets the new viewport transformation based on the `SVGMatrix` otherwise.

```
paper.matrix({ a: 2, b: 0, c: 0, d: 2, e: 0, f: 0 }); // scale the paper twice
```

dia.Paper.prototype.pageOffset

`paper.pageOffset()`

Returns coordinates of the paper viewport, relative to the document.

dia.Paper.prototype.pageToLocalPoint

`paper.pageToLocalPoint(p)`

Transform the point `p` defined in the page coordinate system to the local coordinate system.

```
paper.on('blank:pointerup', function(evt) {
  var pagePoint = g.Point(evt.pageX, evt.pageY);
  var localPoint1 = this.pageToLocalPoint(pagePoint);
  // alternative method signature
  var localPoint2 = this.pageToLocalPoint(evt.pageX, evt.pageY);
  // Move the element to the point, where the user just clicks.
  element.position(localPoint1.x, localPoint1.y);
});
```

dia.Paper.prototype.pageToLocalRect

`paper.pageToLocalRect(rect)`

Transform the rectangle `rect` defined in the page coordinate system to the local coordinate system.

```
var x, y;
paper.on('blank:pointerdown', function(evt) {
  x = evt.pageX;
  y = evt.pageY;
});
paper.on('blank:pointerup', function(evt) {
  var pageRect = g.Rect(x, y, evt.pageX - x, evt.pageY - y).normalize();
  var localRect1 = this.pageToLocalRect(pageRect);
  // alternative method signature
  var localRect2 = this.pageToLocalRect(x, y, evt.pageX - x, evt.pageY -
y).normalize();
  // Move and resize the element to cover the area, that the user just selected.
  element.position(localRect1.x, localRect1.y);
  element.resize(localRect1.width, localRect1.height);
});
```

dia.Paper.prototype.paperToLocalPoint

`paper.paperToLocalPoint(p)`

Transform the point `p` defined in the paper coordinate system to the local coordinate system.

```
var paperCenter = g.Point(paper.options.width / 2, paper.options.height / 2);
var localPoint1 = paper.paperToLocalPoint(paperCenter);
// alternative method signature
var localPoint2 = paper.paperToLocalPoint(paper.options.width / 2,
paper.options.height / 2);
// Move the element to the center of the paper viewport.
var elSize = element.size();
element.position(localPoint1.x - elSize.width, localPoint1.y - elSize.height);
```

dia.Paper.prototype.paperToLocalRect

`paper.paperToLocalRect(rect)`

Transform the rectangle `rect` defined in the paper coordinate system to the local coordinate system.

```
var paperRect = g.Rect(0, 0, paper.options.width, paper.options.height);
var localRect1 = paper.paperToLocalRect(paperRect);
// alternative method signature
var localRect2 = paper.paperToLocalRect(0, 0, paper.options.width,
paper.options.height);
// Move and resize the element to cover the entire paper viewport.
element.position(localRect1.x, localRect1.y);
element.size(localRect1.width, localRect1.height);
```

dia.Paper.prototype.properties

The following list contains properties exposed by the paper object:

- `svg` - a reference to the SVG document object the paper uses to render all the graphics.
- `layers` - a reference to the SVG group `<g>` element that contains all the layers of the paper. Paper transformations such as scale is performed on this element and it affects all the layers inside.
- `cells` (cells layer) - a reference to the SVG group `<g>` element the paper wraps all the rendered elements and links into.
- `tools` (tools layer) - a reference to the SVG group `<g>` element the paper wraps all the rendered tools into.
- `defs` - a reference to the SVG `<defs>` element used to define SVG elements for later reuse. This is also a good place to put SVG masks, patterns, filters and gradients.

Normally, you do not need to access these properties directly but in some (advanced) situations, it is handy to have access to them.

dia.Paper.prototype.removeTools

`paper.removeTools()`

Remove all tools view objects attached to all element and link views on this paper.

dia.Paper.prototype.requireView

`paper.requireView(cell[, opt])`

Ensure that the view associated with the `cell` model is attached to the DOM and that it is updated.

This function finds the view by the cell model. If the view is not part of the paper's DOM (e.g. because it was outside the paper viewport), this function attaches it. Additionally, the view is updated to reflect any changes that may have been done on the cell model in the meantime.

Certain CellView methods require the view to be updated and present in the DOM to function properly (e.g. elementView.getBBox/linkView.getBBox). In async papers, you should precede calls to these methods with this function to guard against inconsistencies.

If `opt.silent` is set to `true`, the paper beforeRender and afterRender callbacks are not fired (and `'render:done'` event is not triggered).

dia.Paper.prototype.scale

`paper.scale([sx, sy, ox, oy])`

Scale a paper by `sx` factor in x axis and `sy` factor in y axis. `ox` and `oy` are optional and determine the origin of the scale transformation. This method effectively implements a paper zoom in/out. If the method is called a `"scale"` event is triggered on the paper. When the method is called with no parameter the current paper `scale` transformation is returned.

```
paper.scale(2) // scale 2x (uniformly)
paper.scale(2,3) // scale 2x in `x` axis 3x in `y` axis (non-uniformly)
paper.scale(2,2,100,100) // scale with the origin of the transformation at point
`x=100` and `y=100`
paper.scale() // returns e.g. { sx: 2, sy: 2 }
```

dia.Paper.prototype.scaleContentToFit

`paper.scaleContentToFit([opt])`

Scale the paper content so that it fits the paper dimensions.

Deprecated alias of the transformToFitContent.

dia.Paper.prototype.setDimensions

`paper.setDimensions(width, height)`

Change dimensions of a paper. Dimensions should always be passed to the options object of the `joint.dia.Paper` constructor. Use `setDimensions()` to change dimensions of the paper later on if needed. If the method is called a `"resize"` event is triggered on the paper.

Type	Description
Number	Dimension in pixels (e.g <code>800</code>).
String	Dimension as CSS property value (e.g <code>"100%"</code>). <i>Make sure the paper container element has size defined.</i> <pre>// Responsive Paper var paper = new joint.dia.Paper({ width: '100%', height: '100%' }); window.on('resize', joint.util.debounce(function() { paper.scaleContentToFit({ padding: 10 }); }), false);</pre>
null	No dimension set. Useful when size of the paper controlled in CSS.

dia.Paper.prototype.setGrid

`paper.setGrid(gridOption)`

Set the type of visual grid on the paper. Note, you still have to call `paper.drawGrid()` afterwards to draw the grid.

```
paper.setGrid(); // default pattern (dot) with default settings
paper.setGrid(true); // default pattern (dot) with default settings

paper.setGrid('mesh'); // pre-defined pattern with default settings

paper.setGrid({ name: 'dot', args: { color: 'hsla(212, 7%, 75%, 0.5)' }});
```

dia.Paper.prototype.setGridSize

`paper.setGridSize(gridSize)`

Set the grid size of the paper.

dia.Paper.prototype.setInteractivity

`paper.setInteractivity(interactive)`

Set the interactivity of the paper. For example, to disable interactivity:

```
paper.setInteractivity(false);
```

dia.Paper.prototype.setOrigin

```
paper.setOrigin(x, y)
```

Deprecated in favor of `translate()`. Lets you modify the origin (zero coordinates) of a paper. An origin can also be passed to the options object of the `joint.dia.Paper` constructor. If the method is called a `"translate"` event is triggered on the paper.

dia.Paper.prototype.showTools

```
paper.showTools()
```

Show all tools attached to all element and link views on this paper.

dia.Paper.prototype.transformToFitContent

```
paper.transformToFitContent([opt])
```

Transforms the paper so that it fits the content.

The function accepts an object with additional settings (all optional):

- `opt.padding` – a number or an object of the form `{ top?: [number], right?: [number], bottom?: [number], left?: [number], vertical?: [number], horizontal?: [number] }` that specifies the width of additional padding that should be added around the resulting (scaled) paper content. Default is `0`.
- `opt.preserveAspectRatio` – should the original aspect ratio of the paper content be preserved in the scaled version? Default is `true`.
- `opt.minScaleX`, `opt.minScaleY`, `opt.maxScaleX`, `opt.maxScaleY` – the minimum and maximum allowed scale factors for both axes.
- `opt.scaleGrid` – a number to be used as a rounding factor for the resulting scale. For example, if you pass `0.2` to this option and the scale factor is calculated as `1.15`, then the resulting scale factor will be rounded to `1.2`.
- `opt.useModelGeometry` – should paper content bounding box be calculated from cell models instead of views? Default is `false`. See the documentation of the `paper.getContentBBox` [function](#) for more details.
- `opt.fittingBBox` – an object of the form `{ x: [number], y: [number], width: [number], height: [number] }` is the area of the paper that the content should be scaled to. By default

`opt.fittingBBox` is `{ x: 0, y: 0, width: paper.options.width, height: paper.options.height }`, i.e. the bounding box of the paper in the paper coordinate system.

- `opt.contentArea` – an object of the form `{ x: [number], y: [number], width: [number], height: [number] }` is the area representing the content in local coordinate system that should be scaled to fit the `opt.fittingBBox`. By default `opt.contentArea` is `paper.getContentArea(opt)`.
- `opt.verticalAlign` – a string which specifies how the content should be vertically aligned in the transformed paper. The options are `'top'`, `'middle'` and `'bottom'`. By default the `opt.verticalAlign` is `'top'`.
- `opt.horizontalAlign` – a string which specifies how the content should be horizontally aligned in the transformed paper. The options are `'left'`, `'middle'` and `'right'`. By default the `opt.horizontalAlign` is `'left'`.

The function is illustrated in our [paper demo](#).

dia.Paper.prototype.translate

`paper.translate(x, y)`

Lets you modify the origin (zero coordinates) of a paper. An origin can also be passed to the options object of the `joint.dia.Paper` constructor. If the method is called a `"translate"` event is triggered on the paper. If the method is called with no parameter the current paper `translate` transformation is returned.

```
paper.translate(100, 200) // set origin to `x=100` and `y=200`  
paper.translate(100) // same as calling `translate(100,0)`  
paper.translate() // returns e.g. { tx: 100, ty: 100 }
```

dia.Paper.prototype.unfreeze

`paper.unfreeze([opt])`

Update the views of an [async](#) paper to make sure that the views reflect the cells' models; unfreeze the paper afterward.

Several extra arguments may be provided in the `opt` object:

batchSize	number	For async papers, how many views should there be per one asynchronous process? Default is <code>1000</code> .			
progress	function	Callback function that is called whenever a batch is finished processing. The callback function is provided with three arguments: <table><tr><td>done</td><td>boolean</td><td>Are we done? Was this the last batch?</td></tr></table>	done	boolean	Are we done? Was this the last batch?
done	boolean	Are we done? Was this the last batch?			

		<table> <tr> <td>processed</td><td><i>number</i></td><td>How far along are we? How many elements have been processed as part of a batch so far?</td></tr> <tr> <td>total</td><td><i>number</i></td><td>How many views are there? How many elements in will have been processed as part of a batch when we are done?</td></tr> </table>	processed	<i>number</i>	How far along are we? How many elements have been processed as part of a batch so far?	total	<i>number</i>	How many views are there? How many elements in will have been processed as part of a batch when we are done?
processed	<i>number</i>	How far along are we? How many elements have been processed as part of a batch so far?						
total	<i>number</i>	How many views are there? How many elements in will have been processed as part of a batch when we are done?						
viewport	<i>function</i>	Callback function to determine whether a given view should be added to the DOM. By default, <code>paper.options.viewport</code> is used, but you may provide a different callback function. The format of the callback method is described in the option's documentation , as are examples of usage.						
beforeRender	<i>function</i>	Callback function executed before processing scheduled updates. By default, <code>paper.options.beforeRender</code> is used, but you may provide a different callback function. The format of the callback method is described in the option's documentation .						
afterRender	<i>function</i>	Callback function executed after all scheduled updates had been processed. By default, <code>paper.options.afterRender</code> is used, but you may provide a different callback function. The format of the callback method is described in the option's documentation .						

Use `paper.freeze()` to freeze the paper again.

dia.Paper.prototype.updateViews

`paper.updateViews([opt])`

Update views in a [frozen](#) [async](#) paper to make sure that the views reflect the cells' models; keep the paper frozen afterwards.

Several extra arguments may be provided in the `opt` object:

batchSize	<i>number</i>	For async papers, how many views should there be per one asynchronous process? Default is 1000 .						
progress	<i>function</i>	Callback function that is called whenever a batch is finished processing. The callback function is provided with three arguments: <table> <tr> <td>done</td><td><i>boolean</i></td><td>Are we done? Was this the last batch?</td></tr> <tr> <td>processed</td><td><i>number</i></td><td>How far along are we? How many elements have been processed as part of a batch so far?</td></tr> </table>	done	<i>boolean</i>	Are we done? Was this the last batch?	processed	<i>number</i>	How far along are we? How many elements have been processed as part of a batch so far?
done	<i>boolean</i>	Are we done? Was this the last batch?						
processed	<i>number</i>	How far along are we? How many elements have been processed as part of a batch so far?						

		<table> <tr> <td>total</td><td>number</td><td>How many views are there? How many elements in will have been processed as part of a batch when we are done?</td></tr> </table>	total	number	How many views are there? How many elements in will have been processed as part of a batch when we are done?
total	number	How many views are there? How many elements in will have been processed as part of a batch when we are done?			
viewport	function	Callback function to determine whether a given view should be added to the DOM. By default, <code>paper.options.viewport</code> is used, but you may provide a different callback function. The format of the callback method is described in the option's documentation , as are examples of usage.			

dia.Paper.prototype.options.afterRender

`afterRender` - Callback function executed after all scheduled updates had been processed (both asynchronous and synchronous rendering). The callback function is provided with the following arguments:

stats	object	Statistics about the current update.
options	object	Parameters the current updates were scheduled with.
paper	<i>joint.dia.Paper</i>	This paper instance

dia.Paper.prototype.options.allowLink

`allowLink` - expects a function returning a boolean. The function is run when the user stops interacting with a `linkView`'s arrowhead (source or target). If the function returns `false`, the link is either removed (for links which are created during the interaction) or reverted to the state before the interaction.

```
// Return `false` if a graph cycle is detected (`graphlib` refers to a dependency of
the DirectedGraph plugin).
paper.options.allowLink = function(linkView, paper) {
  var graph = paper.model;
  return graphlib.alg.findCycles(graph.toGraphLib()).length === 0;
}
```

dia.Paper.prototype.options.anchorNamespace

`anchorNamespace` - built-in [anchors](#) are defined by default on the `joint.anchors` namespace. It is also possible to define custom anchors on this namespace. If you would like JointJS to look for anchor definitions under a different namespace, you can set the `anchorNamespace` option to your desired namespace.

dia.Paper.prototype.options.async

async - when **true**, the paper uses asynchronous rendering to display graph cells (i.e. cells added either with [graph.resetCells\(\)](#) or [graph.addCells\(\)](#) methods).

This is very useful for adding a large number of cells into the graph. The rendering performance boost is significant and it doesn't block the UI. However, asynchronous rendering brings about some caveats – at the time when you call another function...

- ...the views of freshly added cell models may not have yet been added to this paper's DOM.
- ...some views may have been removed from the DOM by the [paper.options.viewport](#) function.
- ...views already present in the DOM may not have been updated to reflect changes made in this paper's graph since the last render.
- ...re-rendering may have been manually disabled with [paper.options.frozen](#) or [paper.freeze\(\)](#).

This is an issue because certain CellView/Paper methods require the view to be updated and present in the DOM to function properly (e.g. [paper.findViewByModel\(\)](#) or [element.findView\(\)](#) / [link.findView\(\)](#), as well as [paper.getContentBBox\(\)](#) and [elementView.getBBox\(\)](#) / [linkView.getBBox\(\)](#)).

The problem may be circumvented in several Paper methods via the `useModelGeometry` option to force using model calculations instead of view measurements (e.g. [paper.getContentBBox\(\)](#), [paper.getContentArea\(\)](#), [paper.scaleContentToFit\(\)](#), [paper.fitToContent\(\)](#)). In this case, the methods refer to the (always up-to-date) model data.

For the methods that truly need a to refer to a CellView, one way to prevent inconsistencies is to rely on the **'render:done'** [paper event](#). This event signals that all scheduled updates are done and that the state of cell views is consistent with the state of the cell models.

Alternatively, you may trigger a manual update immediately before a sensitive function call. JointJS offers several suitable methods:

- [paper.requireView\(\)](#) - bring a single view into the DOM and update it.
- [paper.dumpViews\(\)](#) - bring all views into the DOM and update them.
- [paper.updateViews\(\)](#) - update all views.

dia.Paper.prototype.options.autoFreeze

autoFreeze - By default, the paper periodically checks the viewport calling [viewport](#) callback for every view. The options `autoFreeze` can be used to suppress this behavior. When the option is enabled, it freezes the paper as soon as

there are no more updates and unfreezes it when there is an update scheduled. (paper has no updates to perform when `hasScheduledUpdates()` returns `false`). In that case, the user needs to `checkViewport` manually when he wants to reinvoke `viewport` when external changes occurred, e. g. during scrolling and zooming. The freeze state will be set on the next frame cycle after processing all updates. On this cycle paper will call `viewport` callback and freeze itself.

dia.Paper.prototype.options.background

An object defining the paper background color and image. It defaults to `false` meaning there is no background set. The configuration object can have the following properties.

- **color** property sets the paper background color. It accepts all the values accepted by the CSS `background-color` property. e.g. `'red'`, `'rgba(255, 255, 0, 0.5)'`, `'radial-gradient(ellipse at center, red, green);'`
- **image** property defines the path to the background image file. e.g. `'/my-background.png'`.
- **position** is an object `{ x: Number, y: Number }` defining the background image position. It also allows to use all the CSS `background-position` property values. In that case all the paper transformations have no impact on the background image position. It defaults to `center`.
- **size** is an object `{ width: Number, height: Number }` defining the background image size. It also allows to use all the CSS `background-size` property values. In that case all the paper transformations have no impact on the background size. It defaults to `auto auto`.
- **repeat** property defines how the background image is repeated. The value could be any value accepted by the CSS `background-repeat` property and few more defined by JointJS. Those are `flip-x`, `flip-y`, `flip-xy` and `watermark`. It defaults to `no-repeat`.
- **quality** is a coefficient specifying the quality of the image (e.g. `0.5` uses only 50% the image size for creating a pattern). Applicable only for the JointJS `repeat` option values. It defaults to `1`.
- **opacity** is a number in the range `[0,1]` specifying the transparency of the background image (`0` is fully transparent and `1` is fully opaque). It defaults to `1`.
- **watermarkAngle** is an angle in degrees specifying the slope of the watermark image. Applicable only for the `'watermark'` repeat option. It defaults to `20` deg.

```
background: {  
  color: '#6764A7',  
  image: 'jointjs-logo.png',  
  repeat: 'watermark',  
  opacity: 0.3  
}
```

dia.Paper.prototype.options.beforeRender

`beforeRender` - Callback function executed before processing scheduled updates (both asynchronous and synchronous rendering). The callback function is provided with the following arguments:

<code>options</code>	<i>object</i>	Parameters the current updates were scheduled with.
<code>paper</code>	<i>joint.dia.Paper</i>	This paper instance.

dia.Paper.prototype.options.cellViewNamespace

`cellViewNamespace` - In order for the JointJS `paper` to read cell view definitions from the correct location, the `paper` option `cellViewNamespace` must be provided. Built-in cell view definitions are usually located in the `joint.shapes` namespace, so this is a common namespace to use. It's possible to add a custom cell view to this namespace, or alternatively, you may like to use a different namespace completely.

For example, if `joint.shapes` is provided as the value of `cellViewNamespace`, and a cell is of type `'custom.Element'`, then the `paper` looks up the `joint.shapes.custom.ElementView` object type when rendering the cell onto the screen. If `cellViewNamespace` is set with a value of `myCustomNamespace`, the `paper` will read view definitions from the `myCustomNamespace.custom.ElementView` object instead. This option is often used in combination with the `cellNamespace` option on the `joint.dia.Graph` object.

dia.Paper.prototype.options.clickThreshold

`clickThreshold` - When number of mousemove events exceeds the `clickThreshold` there is no `pointerclick` event triggered after mouseup. It defaults to `0`.

dia.Paper.prototype.options.connectionPointNamespace

`connectionPointNamespace` - built-in `connectionPoints` are defined by default on the `joint.connectionPoints` namespace. It is also possible to define custom `connectionPoints` on this namespace. If you would like JointJS to look for `connectionPoint` definitions under a different namespace, you can set the `connectionPointNamespace` option to your desired namespace.

dia.Paper.prototype.options.connectionStrategy

`connectionStrategy` - the connection strategy to use on this paper. It can either be a function or `null`. JointJS provides a collection of `built-in connection strategies` in the `joint.connectionStrategies` namespace;

alternatively, a [custom function](#) may be provided. See the [link geometry connectionStrategy documentation](#) for more information.

dia.Paper.prototype.options.connectorNamespace

`connectorNamespace` - built-in [connectors](#) are defined by default on the `joint.connectors` namespace. It is also possible to define custom connectors on this namespace. If you would like JointJS to look for connector definitions under a different namespace, you can set the `connectorNamespace` option to your desired namespace.

dia.Paper.prototype.options.defaultAnchor

`defaultAnchor` - an anchor function that is used by links if no `anchor` property is defined for a link end. It can either be an object (with a `name` and optional `args`) or a function. JointJS provides a collection of [built-in anchor functions](#) in the `joint.anchors` namespace; alternatively, a [custom function](#) may be provided. See the [link geometry anchor documentation](#) for more information.

dia.Paper.prototype.options.defaultConnectionPoint

`defaultConnectionPoint` - a connection point function that is used by links if no `connectionPoint` property is defined for a link end. It can either be an object or a function. JointJS provides a collection of [built-in connection point functions](#) in the `joint.connectionPoints` namespace; alternatively, a [custom function](#) may be provided. See the [link geometry connectionPoint documentation](#) for more information.

dia.Paper.prototype.options.defaultConnector

`defaultConnector` - a connector that is used by links if no `connector` property is defined on them. It can either be an object or a function. JointJS provides a collection of [built-in connectors](#) in the `joint.connectors` namespace; alternatively, a [custom function](#) may be provided. See the [link geometry connector documentation](#) for more information.

dia.Paper.prototype.options.defaultLink

`defaultLink` - link that should be created when the user clicks and drags an active magnet (when creating a link from a port via the UI). Defaults to `new joint.dia.Link`. It can also be a function with signature `function(cellView, magnet) {}` that must return an object of type `joint.dia.Link`.

dia.Paper.prototype.options.defaultLinkAnchor

`defaultLinkAnchor` - a link anchor function that is used by links if no `linkAnchor` property is defined for a link end. It can either be an object (with a `name` and optional `args`) or a function. JointJS provides a collection of [built-in link anchor functions](#) in the `joint.linkAnchors` namespace; alternatively, a [custom function](#) may be provided. See the [link geometry linkAnchor documentation](#) for more information.

dia.Paper.prototype.options.defaultRouter

`defaultRouter` - a router that is used by links if no `router` property is defined on them. It can either be an object or a function. JointJS provides a collection of [built-in routers](#) in the `joint.routers` namespace; alternatively, a [custom function](#) may be provided. See the [link geometry router documentation](#) for more information.

dia.Paper.prototype.options.drawGrid

`drawGrid` - option whether the paper grid should be drawn or not. It could be a `boolean` or an `object`. It defaults to `false`.

Define `color` and `thickness` to adjust the default grid pattern:

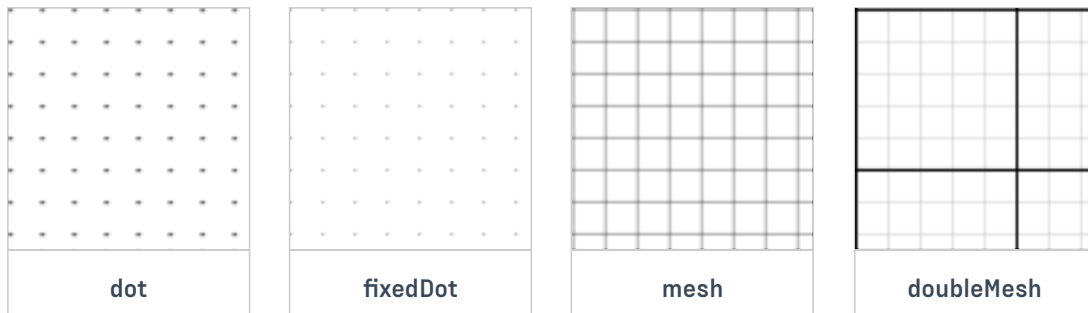
color	<i>string</i>	color of the default grid pattern.
thickness	<i>number</i>	thickness of the default grid pattern.

There are also some pre-defined grid patterns: `dot`, `fixedDot`, `mesh`, `doubleMesh`. If you'd like to use these patterns, define `drawGrid` options as follows:

name	<i>string</i>	name of the pre-defined pattern. Can be either <code>dot</code> , <code>fixedDot</code> , <code>mesh</code> , <code>doubleMesh</code>
args	<i>object array</i>	an extra argument for the pre-defined grid patterns. It can be either an <code>object</code> (e.g. <code>dot</code> , <code>mesh</code>) or an <code>array</code> (e.g. <code>doubleMesh</code>)

Pre-defined grids with default settings:

The paper from the images below has been scaled 2 times and has `gridSize` set to 10.



```
drawGrid: true // default pattern (dot) with default settings

drawGrid: 'mesh' // pre-defined pattern with default settings
drawGrid: { name: 'mesh', args: { color: 'black' } }

drawGrid: {
  name: 'doubleMesh',
  args: [
    { color: 'red', thickness: 1 }, // settings for the primary mesh
    { color: 'green', scaleFactor: 5, thickness: 5 } // settings for the secondary
    mesh
  ]
}
```

dia.Paper.prototype.options.drawGridSize

`drawGridSize` - the size of the visual grid drawn using the `drawGrid` option. If this option is set to `null` (default), the `gridSize` option is used instead. If the option is changed after the paper has been initialized, call `drawGrid` to update the grid.

dia.Paper.prototype.options.el

`el` - CSS selector, jQuery object or a DOM element holding the container for the paper

dia.Paper.prototype.options.elementView

`elementView` - object that is responsible for rendering an element model into the paper. Defaults to `joint.dia.ElementView`. It can also be a function of the form `function(element)` that takes an element model and should return an object responsible for rendering that model onto the screen (in most cases a subtype of `joint.dia.ElementView`).

dia.Paper.prototype.options.embeddingMode

`embeddingMode` - when set to `true`, the paper is set to embedding mode. In this mode, when you drag an element and drop into another element, the element below becomes a parent of the dropped element (the dropped element gets automatically embedded into the parent element). Similarly, when you drag an element out of its parent, the element gets automatically unembedded from its parent. The embedding mode also makes sure all the connected links and child elements have proper `z` index set so that they stay visible. To control what elements can be embedded into what other elements, use the `validateEmbedding()` function on the paper (see below). This is useful for hierarchical diagrams. See the [DEVS demo](#) on how this can be used.

dia.Paper.prototype.options.findParentBy

`findParentBy` - determines the way how a cell finds a suitable parent when it's dragged over the paper. The cell with the highest z-index (visually on the top) will be chosen. `findParentBy` option comes to play when the paper is in embedding mode. All possible values are

Value	Description
<code>'bbox'</code> (default)	Choose the parent from the elements under the element rectangle.
<code>'pointer'</code>	Choose the parent from the elements under the cursor (pointer).
<code>'center'</code> <code>'origin'</code> <code>'corner'</code> <code>'topRight'</code> <code>'bottomLeft'</code>	Choose the parent from the elements under a specific point of the element rectangle.
<code>function</code>	The function accepts an <code>elementView.evt</code>, <code>x</code>, <code>y</code> and it returns array of possible candidates. <pre>findParentBy: function(elementView, evt, x, y) { const graph = elementView.paper.model; return graph.findModelsFromPoint({ x, y }); }</pre>

dia.Paper.prototype.options.frontParentOnly

frontParentOnly - when set to the default `true`, only the element at the very front is taken into account for embedding. If disabled, the elements under the dragged view are tested one by one (from front to back) until a valid parent is found.

```
// If a child element is embedded and covers the area of 'app.Container', a third  
element is unable to be embedded  
validateEmbedding: function(childView, parentView) {  
    const model = parentView.model;  
    if (model.get('type') === 'app.Container') return true;  
    return false;  
},  
frontParentOnly: true,
```

dia.Paper.prototype.options.frozen

frozen - when `true` on an `async` paper, the paper is “frozen”. This means that changes made in this paper's graph are not reflected by the paper until this state is reversed. Use `paper.unfreeze()` to unfreeze the paper, and `paper.freeze()` to freeze it again. (You may also use `paper.updateViews()` to manually refresh the paper and leave it frozen.)

Example:

```
var graph = new joint.dia.Graph(); graph.resetCells([cell1, cell2]); var paper = new  
joint.dia.Paper({ el: document.getElementById('paper-holder'), model: graph, frozen:  
true }); paper.unfreeze();
```

dia.Paper.prototype.options.gridSize

gridSize - size of the grid in pixels

dia.Paper.prototype.options.guard

`guard` - guard paper from handling a browser UI event. This function is of the form `function(evt, view)` and should return `true` if you want to prevent the paper from handling the event `evt`, `false` otherwise. This is an advanced option that can be useful if you have your own logic for handling events.

dia.Paper.prototype.options.height

`height` - height of the paper (see `setDimensions()`).

dia.Paper.prototype.options.highlighterNamespace

`highlighterNamespace` - built-in `highlighters` are defined by default on the `joint.highlighters` namespace. Existing and custom highlighters are defined by extending the `base class`. If you would like JointJS to look for highlighter definitions under a different namespace, you can set the `highlighterNamespace` option to your desired namespace.

dia.Paper.prototype.options.highlighting

`highlighting` - Configure which highlighter to use (if any) for each type of interaction.

List of highlighting interactions

Name	Enum	Description	Default
default	<i>joint.dia.CellView.Highlighting.DEFAULT</i>	the default highlighter to use (and options) when none is specified	<pre>{ name: 'stroke', options: { padding: 3 } }</pre>

connecting	<i>joint.dia.CellView.Highlighting.CONNECTING</i>	when a valid link connection can be made to an element	Not defined. The default is used.
embedding	<i>joint.dia.CellView.Highlighting.EMBEDDING</i>	when a cell is dragged over another cell in embedding mode	Not defined. The default is used.
magnetAvailability	<i>joint.dia.CellView.Highlighting.MAGNET_AVAILABILITY</i>	when showing all magnets to which a valid connection can be made	<pre> { name: 'addClass', options: { className: 'available-magnet' } } </pre>
elementAvailability	<i>joint.dia.CellView.Highlighting.ELEMENT_AVAILABILITY</i>	when showing all elements to which a valid connection can be made	<pre> { name: 'addClass', options: { className: 'available' </pre>


```
table
-
cell'
}
```

If a type is set to `false`, no highlighter is used.

If a type is set to `null` | `undefined`, the default highlighter is used.

Example usages:

```
new joint.dia.Paper({
  highlighting: {
    'default': {
      name: 'stroke', // `joint.highlighters.stroke`
      options: {
        padding: 2
      }
    },
    'connecting': {
      name: 'addClass', // `joint.highlighters.addClass`
      options: {
        className: 'highlight-connecting'
      }
    },
    // Disable highlighter for embedding
    'embedding': false
  }
});
```

```
new joint.dia.Paper({
  // Disable all highlighters
  highlighting: false
});
```

List of available [highlighters](#) and their options:

- [joint.highlighters.mask](#)
- [joint.highlighters.stroke](#)
- [joint.highlighters.addClass](#)
- [joint.highlighters.opacity](#)

If you need to highlight an element or a link differently based on the cell type or one of its attribute, you can disable the paper highlighting and add a highlighter manually.

```

// Disable Embedding
paper.options.highlighting.embedding = false;

const MyHighlighters = {
  EMBEDDING: 'embedding-highlighter'
};

paper.on('cell:highlight', (cellView, node, { type }) => {
  if (type === joint.dia.CellView.Highlighting.EMBEDDING) {
    const isLink = cellView.model.isLink();
    joint.highlighters.mask.add(cellView, node, MyHighlighters.EMBEDDING, {
      attrs: {
        'stroke': isLink ? 'red' : 'blue'
      }
    });
  }
});

paper.on('cell:unhighlight', (cellView, node, { type }) => {
  if (type === joint.dia.CellView.Highlighting.EMBEDDING) {
    joint.highlighters.mask.remove(cellView, MyHighlighters.EMBEDDING);
  }
});

```

By default, when a user connects a link, the target node for the highlighter is either the root of the cell or a magnet. To change this use [highlighterSelector](#) attribute.

```

const element = new joint.shapes.standard.Rectangle({
  attrs: {
    root: {
      highlighterSelector: 'body'
    }
  }
});

// When a user tries to connect a link to the element.
paper.on('cell:highlight', (elementView, node) => {
  assert.notOk(node === elementView.el);
  assert.ok(node === elementView.el.querySelector('[joint-selector="body"]'));
});

```

dia.Paper.prototype.options.interactive

`interactive` - Configure which of the default interactions with elements and links should be enabled.

The property value defaults to `{ labelMove: false }`. This can be overwritten in three ways: with a boolean value, with an object specifying interaction keys, or with a function.

If set to `false`, all interactions with elements and links are disabled. If set to `true`, all interactions are enabled.

```
// disable all interaction
var paper = new joint.dia.Paper({
  // ...
  interactive: false,
});

// enable all interaction (including labelMove)
var paper = new joint.dia.Paper({
  // ...
  interactive: true,
});
```

Using an object, specific interactions may be disabled by assigning `false` to their corresponding property name. It is not necessary to pass `true` values; all omitted properties are assigned `true` by default. (Note that the passed object is not merged with the default; unless `labelMove` is explicitly excluded, it becomes enabled.) A full list of recognized interaction keys is provided below.

```
// disable arrowheadMove
var paper = new joint.dia.Paper({
  // ...
  interactive: { arrowheadMove: false }
});

// disable all element interactions
var paper = new joint.dia.Paper({
  // ...
  interactive: { elementMove: false, addLinkFromMagnet: false }
});
```

If defined as a function, the function is passed `cellView` (the `elementView/linkView` the user is about to interact with) as the first parameter, followed by the name of the event (`'pointerdown'`, `'pointermove'`, ...) that triggered the interaction. The return value of the function is then interpreted in the way specified above (`false` causes all interaction to be disabled, an object disables specific interactions, etc.).

```
// disable link interactions for cellViews when a custom property is set
var paper = new joint.dia.Paper({
  // ...
  interactive: function(cellView) {
    if (cellView.model.get('disableLinkInteractions')) {
      return {
```

```

        linkMove: false,
        labelMove: false,
        arrowheadMove: false,
        vertexMove: false,
        vertexAdd: false,
        vertexRemove: false,
        useLinkTools: false,
    };
}

// otherwise
return true;
}
});

```

The example below has all interactions on the link and on the elements enabled. This is the default behavior:

The following tables present a list of all recognized interaction keys, followed by an example of a paper with only the related interactive property set to `true` (and all other properties set to `false`).

Links:

linkMove	Is the user allowed to move the link?
labelMove	Is the user allowed to move the link label? Use the <code>paper.options.snapLabels</code> paper option to only allow the label to be dragged along the length of the link.
arrowheadMove	Deprecated. Use a link tool instead. (Is the user allowed to move the arrowheads?)
vertexMove	Deprecated. Use a link tool instead. (Is the user allowed to move the vertices?)
vertexAdd	Deprecated. Use a link tool instead. (Is the user allowed to add vertices by clicking along the link path?)
vertexRemove	Deprecated. Use a link tool instead. (Is the user allowed to remove vertices?)

useLinkTools	Deprecated. (Is the user allowed to use the default link buttons?)

Elements:

elementMove	Is the user allowed to move the element?
addLinkFromMagnet	Is the user allowed to add connections from magnets/ports?

The `stopDelegation` option is special. If it is `true` (default), the element's `elementMove` option determines whether the element responds to user drag.

However, if `stopDelegation` is `false` and the element is embedded within a parent, the user's dragging is delegated to the embedding parent. The parent's `elementMove` option then determines whether both elements respond to user drag. The behavior is recursive. If the embedding parent has `stopDelegation: false`, it delegates to its own embedding parent's `elementMove` option and so on. If all children within an embedding have `stopDelegation` set to `false`, then no matter which element is dragged by the user, the whole embedding is dragged.

If the element is not embedded within an element, the `stopDelegation` option is ignored (treated as `true`). There is no other element to delegate to. Then `elementMove` determines whether the element responds to user drag.

In the following example, both embedded elements have `stopDelegation: false`. Thus, when the embedded element is dragged by the user, the parent ancestor ("Movable") is dragged instead. When the parent ancestor has `elementMove` disabled ("Not movable"), nothing happens.

stopDelegation	

dia.Paper.prototype.options.labelsLayer

`labelsLayer` - controls the stacking context of the link labels.

Value	Description
false (default)	The labels are rendered as part of the links. Overlapping links with a larger <code>z</code> can cover link labels with a smaller one.
true	The labels are rendered into its own layer on top of other elements and links. A link's connection can cover no label.

dia.Paper.prototype.options.linkAnchorNamespace

`linkAnchorNamespace` - built-in `linkAnchors` are defined by default on the `joint.linkAnchors` namespace. It is also possible to define custom linkAnchors on this namespace. If you would like JointJS to look for linkAnchor definitions under a different namespace, you can set the `linkAnchorNamespace` option to your desired namespace.

dia.Paper.prototype.options.linkConnectionPoint

`linkConnectionPoint`(linkView, view, magnet, reference) - this function allows you to customize what are the sticky points of links. The function must return a point (with `x` and `y` properties) where the link sticks to the element. The function takes the link view, element view, the magnet (SVG element) the link should stick to and a reference point (either the closest vertex or the sticky point on the other side of the link). Note that there is a utility function `shapePerimeterConnectionPoint` that can be directly passed to the `linkConnectionPoint` parameter. This function tries to find the best possible connection point right on the perimeter of the connected shapes. See the function documentation for details.

dia.Paper.prototype.options.linkPinning

`linkPinning` - when set to `true` (the default), links can be “pinned” to the paper meaning a source or target of a link can be a point. If you do not want to let the user drag a link and drop it somewhere in a blank paper area, set this to `false`. The effect is that the link will be returned to its original position whenever the user drops it somewhere in a blank paper area.

dia.Paper.prototype.options.linkView

`linkView` - object that is responsible for rendering a link model into the paper. Defaults to `joint.dia.LinkView`. It can also be a function of the form `function(Link)` that takes a link model and should return an object responsible for rendering that model onto the screen (in most cases a subtype of `joint.dia.LinkView`).

dia.Paper.prototype.options.magnetThreshold

`magnetThreshold` - When defined as a number, it denotes the required mousemove events before a new link is created from a magnet. When defined as keyword `'onleave'`, the link is created when the pointer leaves the magnet DOM element. It defaults to `0`.

dia.Paper.prototype.options.markAvailable

`markAvailable` - marks all the available magnets or elements when a link is dragged (being reconnected). Default is `false`. This gives a hint to the user to what other elements/ports this link can be connected. What magnets/cells are available is determined by the `validateConnection` function. Internally, available magnets (SVG elements) are given the `'available-magnet'` class name and all the available cells the `'available-cell'` class name. This allows you to change the styling of the highlight effect.

dia.Paper.prototype.options.model

`model` - `joint.dia.Graph` object

dia.Paper.prototype.options.moveThreshold

`moveThreshold` - Number of required mousemove events before the first pointermove event is triggered. It defaults to `0`.

dia.Paper.prototype.options.multiLinks

`multiLinks` - when set to `false`, an element may not have more than one link with the same source and target element.

dia.Paper.prototype.options.origin

`origin` - position of zero coordinates of the paper in pixels (default is `{ x: 0, y: 0 }` i.e. top-left corner)

dia.Paper.prototype.options.overflow

`overflow` - if set to `true`, the content of the paper is not clipped and may be rendered outside the paper area. If set to `false`, the content is clipped, if necessary, to fit the paper area.

dia.Paper.prototype.options.perpendicularLinks

`perpendicularLinks` - if `true`, links will tend to get perpendicular to their associated objects. Default is `false`

dia.Paper.prototype.options.preventContextMenu

`preventContextMenu` - Prevents default context menu action (when right button of the mouse is clicked). It defaults to `true`.

dia.Paper.prototype.options.preventDefaultBlankAction

`preventDefaultBlankAction` - Prevents default action when an empty paper area is clicked. Setting the option to `false` will make the paper pannable inside a container on touch devices. It defaults to `true`.

dia.Paper.prototype.options.preventDefaultViewAction

`preventDefaultViewAction` - Prevents default action when a cell view is clicked. For example, setting the option to `false` will cause clicking on the view to blur the document active element. It defaults to `true`.

dia.Paper.prototype.options.restrictTranslate

`restrictTranslate` - restrict the translation (movement) of elements by a given bounding box. If set to `true`, the user will not be able to move elements outside the boundary of the paper area. It's set to `false` by default. This option also accepts a function with signature `function(elementView, x0, y0)` in which case it must return one of the following:

- **rectangle** - (an object of the form `{ x: Number, y: Number, width: Number, height: Number }`) that defines the area in which the element represented by `elementView` can be moved.
- **null** - no restriction for element translation
- **(x, y) => Position** - For a given element position `x, y` return a new position `{ x: Number, y: Number }`. The function is invoked every time the user moves the pointer.

For example, to restrict translation of embedded elements by the bounding box defined by their parent element, you can do:


```

restrictTranslate: function(elementView) {
  const parent = elementView.model.getParentCell();
  return parent
    ? parent.getBBox() // children movement is constrained by the parent area
    : null; // parent movement is not restricted
}

```

Or restrict an element movement along a path:

```

restrictTranslate: function(elementView, x0, y0) {
  // x0 and y0 are pointer coordinates at the start of the translation
  const path = new g.Path('M 20 20 L 200 200 L 380 20');
  const { x: x1, y: y1, width, height } = elementView.model.getBBox();
  // The initial difference between the pointer coordinates and the element
  position
  const dx = x1 - x0;
  const dy = y1 - y0;
  return function(x, y) {
    // Find the point on the path.
    const point = path.closestPoint({ x: x - dx, y: y - dy });
    // Put the center of the element on this point
    point.offset(-width / 2, -height / 2);
    return point;
  };
}

```

dia.Paper.prototype.options.routerNamespace

`routerNamespace` - built-in routers are defined by default on the `joint.routers` namespace. It is also possible to define custom routers on this namespace. If you would like JointJS to look for router definitions under a different namespace, you can set the `routerNamespace` option to your desired namespace.

dia.Paper.prototype.options.snapLabels

`snapLabels` - when enabled, force a dragged label to snap to its link. The default is `false`, which means that the label may also be dragged around the link.

dia.Paper.prototype.options.snapLinks

`snapLinks` - when enabled, force a dragged link to snap to the closest element/port in the given radius. The option accepts `true` in which case default values are used or an object of the form `{ radius: <value> }` where you can specify the radius (default is 50).

dia.Paper.prototype.options.snapLinksSelf

`snapLinksSelf` - when enabled, forces a dragged link end to snap to the `x` or `y` coordinate of the closest point on the current link (one of the ends and vertices) if the distance to the point on one axis is lesser than the specified radius. The option accepts `true` in which case default values are used or an object of the form `{ radius: <value> }` where you can specify the radius (the default is 20).

dia.Paper.prototype.options.sorting

`sorting` - the sorting type to use when rendering the views in this paper (i.e. "In what order should the views be rendered?").

The SVG 1.1 format does not have built-in z-index functionality, so JointJS implements it programmatically. You can set the z-index directly using using regular Backbone `set('z')` / `get('z')` methods or through `element.toFront()` / `element.toBack()` / `link.toFront()` / `link.toBack()` methods. See the [element Z documentation](#) for more information.

The Paper object exposes a `sorting` object with three values that may be used as values of this option:

- `joint.dia.Paper.sorting.EXACT` - (default) render views in exactly the same order as reported by `graph.getCells()` (views with different z-values are rendered in order, and views with the same z-value are rendered in the order in which they were added). This is by far the slowest option, present mainly for backwards compatibility.
- `joint.dia.Paper.sorting.APPROX` - render views according to their z-values. Views with different z-value are rendered in order, but the ordering of views with the same z-value is indeterminate. Similar in functionality to the `EXACT` option, but much faster.
- `joint.dia.Paper.sorting.NONE` - render views in an indeterminate order. (Note that this setting disables all `toFront` / `toBack` functions mentioned above.)

dia.Paper.prototype.options.validateConnection

`validateConnection(cellViewS, magnetS, cellViewT, magnetT, end, linkView)` - a function that allows you control which link connections can be made between the source `cellView/magnet ( cellViewS/magnetS )`, and the target `cellView/magnet ( cellViewT/magnetT )`. `end` is either the `"source"` or the `"target"`, and indicates which end of the link is being dragged. This is useful for defining whether a link starting in port `POut` of element A can lead to port `PIn` of element B. By default, all connections are allowed.

```

// The default allows all element connections
validateConnection: function(cellViewS, _magnetS, cellViewT, _magnetT, end,
_linkView) {
    return (end === 'target' ? cellViewT : cellViewS) instanceof ElementView;
}

// Other examples
validateConnection: function(cellViewS, magnetS, cellViewT, magnetT, end, linkView) {

    // Prevent Link to Link connections
    if (cellViewT.model.isLink() || cellViewS.model.isLink()) return false;

    // Prevent loop linking - This means the source and target cannot connect to the
    same magnet (e.g. port)
    if (magnetS === magnetT) return false;

    // Only allow target end to be reconnected
    if (end === 'source') return false;

    // Don't allow connection to cellView with 'isInactive' attribute of value true
    if (cellViewS.model.get('isInactive') || cellViewT.model.get('isInactive'))
return false;

    // Prevent linking from input ports (assumes you have port groups setup)
    if (magnetS && magnetS.getAttribute('port-group') === 'in') return false;

    // Allow all other connections
    return true;
}

```

dia.Paper.prototype.options.validateEmbedding

validateEmbedding - a function that allows you to control what elements can be embedded to what other elements when the paper is put into the embeddingMode (see above). The function signature is: `function(childView, parentView)` and should return `true` if the `childView` element can be embedded into the `parentView` element. By default, all elements can be embedded into all other elements (the function returns `true` no matter what).

dia.Paper.prototype.options.validateMagnet

`validateMagnet`(cellView, magnet, evt) - decide whether to create a link if the user clicks a magnet.
magnet is the DOM element representing the magnet. By default, this function returns `true` for magnets that are not explicitly set to "passive" (which is usually the case of input ports).

dia.Paper.prototype.options.validateUnembedding

`validateUnembedding` - a function that allows you to control which elements can be unembedded when the paper is put into the `embeddingMode` (see above).
The function signature is `function(childView)` and should return `true` if the `childView` element can be unembedded from the parent and become a new root.
By default, all elements can become roots (the function returns `true` no matter what).
The callback is called at the end of drag & drop action and if `false` is returned, the position of the element, as well as the parent reference is reverted to the values before dragging.
The callback is called only on elements which were previously embedded in another element.

dia.Paper.prototype.options.viewport

`viewport` - a callback function that is used to determine whether a given view should be shown in an `async` paper. If the function returns `true`, the view is attached to the DOM; if it returns `false`, the view is detached/unmounted from the DOM. The callback function is provided with three arguments:

view	<i>mvc.View</i>	The view in question
isMounted	<i>boolean</i>	Is the view currently visible in the paper?
paper	<i>dia.Paper</i>	This paper (for context)

This function is meant to support hiding of views when they are outside the viewport. In most situations, it is unnecessary to render DOM elements that are not visible by the user; removing those elements from the DOM dramatically improves rendering times of huge graphs (thousands of views) and improves smoothness of user interaction. (If you do need to show a view that falls outside of this viewport, you can manually force the view to be shown using `paper.requireView()`. If you need to show views according to a different viewport function, use `paper.checkViewport()`. If you need to force all views to be shown, use `paper.dumpViews()`.)

Example usage:

```
var viewportRect = new g.Rect(0, 0, 600, 400);
var graph = new joint.dia.Graph();
var paper = new joint.dia.Paper({
  model: graph,
  async: true,
  viewport: function(view) {
```

```

    var model = view.model;
    var bbox = model.getBBox();
    if (model.isLink()) {
        // vertical/horizontal links have zero width/height
        // we need to manually inflate the bounding box
        bbox.inflate(1);
    }
    // Return true if there is an intersection
    // Return true if bbox is within viewportRect
    return viewportRect.intersect(bbox);
}
});

```

The `viewport` function can also be used to support collapsing/uncollapsing behavior:

```

var graph = new joint.dia.Graph();
var paper = new joint.dia.Paper({
  model: graph,
  async: true,
  viewport: function(view) {
    // Return true if model is not hidden
    return !model.get('hidden');
  }
});

paper.on('element:pointerclick', function(view, evt) {
  evt.stopPropagation();
  toggleBranch(view.model);
});

function toggleBranch(root) {
  var shouldHide = !root.get('collapsed');
  root.set('collapsed', shouldHide);
  graph.getSuccessors(root).forEach(function(successor) {
    successor.set('hidden', shouldHide);
    successor.set('collapsed', false);
  });
}

```

dia.Paper.prototype.options.width

width - width of the paper (see [setDimensions\(\)](#)).

dia.ToolsView

The `joint.dia.ToolsView` class works as a container for one set of link tools (an array of `joint.dia.ToolView` objects). It is responsible for rendering the tools as a group when the tools view is attached to a `joint.dia.LinkView`.

To create a new tools view object, we call its constructor. Two optional arguments are accepted:

name	<i>string</i>	The name of this tools view. Default is <code>null</code> (no name).
tools	<i>Array<dia.ToolView></i>	An array of <u>tools</u> that should be a part of the tools view. Default is <code>[]</code> (no tools).

Creating a tools view is the second step in the process of setting up link tools on a link view:

```
// 1) creating link tools
var verticesTool = new joint.linkTools.Vertices();
var segmentsTool = new joint.linkTools.Segments();
var boundaryTool = new joint.linkTools.Boundary();

// 2) creating a tools view
var toolsView = new joint.dia.ToolsView({
  name: 'basic-tools',
  tools: [verticesTool, segmentsTool, boundaryTool]
});

// 3) attaching to a link view
var linkView = link.findView(paper);
linkView.addTools(toolsView);
```

Every link view we want to attach to requires its own tools view object (`ToolsView` objects are automatically reassigned to the last link view they are added to). Similarly, every tools view we create requires its own set of tools (`ToolView` objects are automatically reassigned to the last `toolsView.tools` array they were made part of).

dia.ToolsView.prototype.blurTool

`toolsView.blurTool(tool)`

Stop focusing the specified `tool` (of the `joint.dia.ToolView` type) within this tools view.

This function reverses the effect of the `toolsView.focusTool` function.

dia.ToolsView.prototype.blurTools

`toolsView.blurTools()`

Stop focusing any tools within this tools view.

This function reverses the effect of the `toolsView.focusTool` [function](#) applied to any tool in this tools view.

dia.ToolsView.prototype.focusTool

`toolsView.focusTool(tool)`

Focus the specified `tool` (of the `joint.dia.ToolView` [type](#)) within this tools view.

When a tool is “focused”, all other tools within this tools view are hidden and the tool's opacity is changed according to the tool's `focusOpacity` option. Only one tool can be focused at a time; the focus passes to the last tool within this tools view with which the `focusTool` function was called.

The effect of this function can be reversed for a single `tool` by the `toolsView.blurTool` [function](#). All tools can be unfocused by the `toolsView.blurTools` [function](#).

dia.ToolsView.prototype.getName

`toolsView.getName()`

Return the name of this tools view. If no name was set during the construction of the tools view, `null` is returned.

dia.ToolsView.prototype.hide

`toolsView.hide()`

Hide all tools within this tools view.

dia.ToolsView.prototype.show

`toolsView.show()`

Show all tools within this tools view.

dia.ToolsView.prototype.options.layer

layer - the stacking context of the tools view.

<code>"tools"</code> <i>(default)</i>	Render the tools in front of all the cells.
<code>null</code>	Render the tools above the cell. It can be hidden by a cell with a higher z index.

dia.ToolsView.prototype.options.z

z - the stacking order (z-index) of the tools view in the given stacking context.

dia.ToolView

The `joint.dia.ToolView` class is the parent class of all link tool views included in the `joint.linkTools` namespace. It is responsible for rendering the tool within a `joint.dia.ToolsView`, when the tools view is attached to a `joint.dia.LinkView`.

Creating link tools objects is the first step in the process of setting up link tools on a link view:

```
// 1) creating link tools
var verticesTool = new joint.linkTools.Vertices();
var segmentsTool = new joint.linkTools.Segments();
var boundaryTool = new joint.linkTools.Boundary();

// 2) creating a tools view
var toolsView = new joint.dia.ToolsView({
  name: 'basic-tools',
  tools: [verticesTool, segmentsTool, boundaryTool]
});

// 3) attaching to a link view
var linkView = link.findView(paper);
linkView.addTools(toolsView);
```

Every link view we want to attach to requires its own tools view object (`ToolsView` objects are automatically reassigned to the last link view they are added to). Similarly, every tools view we create requires its own set of tools

(`ToolView` objects are automatically reassigned to the last `toolsView.tools` array they were made part of).

dia.ToolView.prototype.blur

`toolView.blur()`

Stop focusing the tool within its containing `toolsView`.

This function reverses the effect of the `toolView.focus` [function](#).

dia.ToolView.prototype.focus

`toolView.focus()`

Focus the tool within its containing `toolsView`.

When a tool is “focused”, all other tools within the `toolsView` are hidden and the tool's opacity is changed according to the tool's `focusOpacity` option. Only one tool can be focused at a time; the focus passes to the last tool within a tools view on which the `focus` function was called.

The effect of this function can be reversed by the `toolView.blur` [function](#).

dia.ToolView.prototype.getName

`toolView.getName()`

Return the name of the tool. The names of built-in link tools are kebab-case versions of their class names (e.g. the name of `joint.linkTools.SourceAnchor` is `'source-anchor'`).

When creating a custom tool (e.g. by extending the `joint.linkTools.Button` [class](#)), it is recommended that you provide a custom `name` prototype property, as well:

```
joint.linkTools.InfoButton = joint.linkTools.Button.extend({
  name: 'info-button',
  options: {
    markup: [{
      tagName: 'circle',
      selector: 'button',
      attributes: {
        'r': 7,
        'fill': '#001DFF',
        'cursor': 'pointer'
      }
    ]
  }
});
```

```

    }, {
      tagName: 'path',
      selector: 'icon',
      attributes: {
        'd': 'M -2 4 2 4 M 0 3 0 0 M -2 -1 1 -1 M -1 -4 1 -4',
        'fill': 'none',
        'stroke': '#FFFFFF',
        'stroke-width': 2,
        'pointer-events': 'none'
      }
    }
  ]],
  distance: 60,
  offset: 0,
  action: function(evt) {
    alert('View id: ' + this.id + '\n' + 'Model id: ' + this.model.id);
  }
}
});

```

dia.ToolView.prototype.hide

`toolView.hide()`

Hide the tool.

The effect of this function can be reversed by the `toolView.show` [function](#).

dia.ToolView.prototype.isVisible

`toolView.isVisible()`

Return `true` if the tool is currently visible.

A tool is not visible if it has been hidden (e.g. with the `toolView.hide` [function](#)) or if another tool within the containing `toolsView` has been focused (e.g. with the `toolView.focus` [function](#)).

dia.ToolView.prototype.show

`toolView.show()`

Show the tool.

The effect of this function can be reversed by the `toolView.hide` [function](#).

elementTools

An element tool is a view that renders a certain type of control elements on top of the `ElementView` it is attached to; for example the `Remove tool` creates an interactive button above the element (this button then allow the user to remove the element). Element tools all inherit from the `joint.dia.ToolView` [class](#). A collection of tools is added to a `ToolsView`; a tools view is then added to the elementView with the `linkView.addTools()` [function](#).

The JointJS library comes with a collection of pre-made element tool definitions in the `joint.elementTools` namespace:

- Boundary - shows element bbox
- Remove - adds an interactive remove button

To create a new element tool, we call its constructor. Example:

```
var removeTool = new joint.elementTools.Remove({
    rotate: true
});
```

In addition, the `joint.elementTools` namespace contains a customizable button class:

- Button - adds a customizable button

Example:

```
var infoTool = new joint.elementTools.Button({
  markup: [{
    tagName: 'circle',
    selector: 'button',
    attributes: {
      'r': 7,
      'fill': '#001DFF',
      'cursor': 'pointer'
    }
  }, {
    tagName: 'path',
    selector: 'icon',
    attributes: {
      'd': 'M -2 4 2 4 M 0 3 0 0 M -2 -1 1 -1 M -1 -4 1 -4',
      'fill': 'none',
      'stroke': '#FFFFFF',
      'stroke-width': 2,
      'pointer-events': 'none'
    }
  }],

```

```

    x: '100%',
    action: function(evt) {
        alert('View id: ' + this.id + '\n' + 'Model id: ' + this.model.id);
    }
});

```

All of the built-in element tools accept the following optional argument, in addition to their own arguments:

focusOpacity	<i>number</i>	What should be the opacity of the tool when it is focused (e.g. with the <code>toolView.focus</code> function)? Default is undefined , meaning that the tool's opacity is kept unchanged.

Example:

```

var boundaryTool = new joint.elementTools.Boundary({
    focusOpacity: 0.5
});

```

elementTools.Boundary

The `Boundary` element tool renders a rectangular border to show the bounding box of the element. It accepts a few additional arguments, which can be passed as an object to the element tool constructor:

padding	<i>number object</i>	This option determines whether the boundary area should be visually inflated and if so, by how much. Default is 10 ({ <code>left: 10</code> , <code>top: 10</code> , <code>right: 10</code> , <code>bottom: 10</code> }).
rotate	<i>boolean</i>	Should the boundary rotate according to the element angle? Default is false .
useModelGeometry	<i>boolean</i>	If this option is set to true , the position of the boundary is calculated based on the dimensions of the element model.

Example:

```

var boundaryTool = new joint.elementTools.Boundary({
    focusOpacity: 0.5,
    padding: 20,
    useModelGeometry: true
});

```

elementTools.Button

The `Button` element tool allows you to have a custom button rendered at a given position above the element. It accepts a few additional arguments, which can be passed as an object to the element tool constructor:

x	<i>string number</i>	Use percentage strings (e.g. <code>'40%'</code>) or <code>calc()</code> expression (e.g. <code>'calc(0.4 * w)'</code>) to position the button relatively to the element width/height. A number means distance from the top-left corner of the element. Default is <code>0</code> .
y		
rotate	<i>boolean</i>	Should the button rotate according to the element angle around the element center? Default is <code>false</code> .
offset	<i>object</i>	Additional offset of the button from the position defined by <code>x</code> and <code>y</code> . Default is <code>{ x: 0, y: 0 }</code> .
action	<i>function</i>	What should happen when the user clicks the button? Default is <code>undefined</code> (no interaction). The callback function is expected to have the signature <code>function(evt, elementView, buttonView)</code> where <code>evt</code> is a <code>jQuery.Event</code> <code>object</code> . The element view is available inside the function as <code>this</code> ; the element model is available as <code>this.model</code> .
markup	<i><u>JSONMarkup</u></i>	The markup of the button, provided in the <u>JointJS JSON format</u> . Default is <code>undefined</code> (no content).
scale	<i>number</i>	Scale the button up or down on a 2D plane. The default is <code>1</code> .

Example of a useful custom info button:

```
var infoButton = new joint.elementTools.Button({
  focusOpacity: 0.5,
  // top-right corner
  x: '100%',
  y: '0%',
  offset: { x: -5, y: -5 },
  action: function(evt) {
    alert('View id: ' + this.id + '\n' + 'Model id: ' + this.model.id);
  },
  markup: [{
    tagName: 'circle',
    selector: 'button',
```

```

        attributes: {
            'r': 7,
            'fill': '#001DFF',
            'cursor': 'pointer'
        }
    }, {
        tagName: 'path',
        selector: 'icon',
        attributes: {
            'd': 'M -2 4 2 4 M 0 3 0 0 M -2 -1 1 -1 M -1 -4 1 -4',
            'fill': 'none',
            'stroke': '#FFFFFF',
            'stroke-width': 2,
            'pointer-events': 'none'
        }
    }
}
});

```

The `elementTools.Button` class can also be extended, to create a reusable custom button type. Then, a new instance of the custom button type can be obtained by calling its constructor:

```

var InfoButton = joint.elementTools.Button.extend({
    name: 'info-button',
    options: {
        focusOpacity: 0.5,
        distance: 60,
        action: function(evt) {
            alert('View id: ' + this.id + '\n' + 'Model id: ' + this.model.id);
        },
        markup: [{
            tagName: 'circle',
            selector: 'button',
            attributes: {
                'r': 7,
                'fill': '#001DFF',
                'cursor': 'pointer'
            }
        }, {
            tagName: 'path',
            selector: 'icon',
            attributes: {
                'd': 'M -2 4 2 4 M 0 3 0 0 M -2 -1 1 -1 M -1 -4 1 -4',
                'fill': 'none',
                'stroke': '#FFFFFF',
                'stroke-width': 2,
                'pointer-events': 'none'
            }
        }
    ]
});

```

```

    }
  }
});

var infoButton = new InfoButton();

```

elementTools.Connect

The `Connect` tool allows the user to create links in a drag & drop fashion. The tool extends the [Button](#) tool and accepts additional arguments, which can be passed as an object to the connect tool constructor:

	magnet <code>string</code> <code>SVGElement</code> <code>(view: dia.LinkView) => SVGElement</code>	Choose the source magnet of the link view which the new link should be connected to. <pre> 'body' // as a string Selector linkView.el // as an SVGElement (linkView) => linkView.el // as a function </pre> The callback function is expected to have the signature <code>function(linkView, toolView)</code> where <code>linkView</code> is the link view with the tool attached. The connect button view is available inside the function as <code>this</code> and as the last parameter.

Example:

```

const connectButton = new joint.elementTools.Connect({
  x: '100%',
  y: '0%',
  offset: { x: -5, y: -5 },
  magnet: 'body'
});

```

elementTools.Control

The `Control` element tool is an abstract class which allows you to build tools to control the look or shape of an element by simply dragging a UI handle. It accepts a few additional arguments, which can be passed as an object to the element

tool constructor:

selector	<i>string</i> <i>null</i>	The element selector pointing to an element subnode, which the tool draws a frame around during dragging. If null provided, no frame will be shown.
padding	<i>number</i>	The padding between the area and the bounding box of the node designated by selector .
handleAttributes	<i>function</i>	An object with SVG attributes to be applied to the tool's handle.
scale	<i>number</i>	Scale the button up or down on a 2D plane. The default is 1 .

```
namespace Control {
  interface Options extends dia.ToolView.Options {
    selector?: string;
    padding?: number;
    handleAttributes?: Partial<attributes.NativeSVGAttributes>
  }
}

abstract class Control extends dia.ToolView {
  protected getPosition(view: dia.ElementView): dia.Point;
  protected setPosition(view: dia.ElementView, coordinates: g.Point): void;
  protected resetPosition(view: dia.ElementView): void;
}
```

<code>getPosition(view: dia.ElementView): dia.Point;</code>	The method should return the position of the handle based on a model value. The position is defined in the element model coordinate system (point <code>[0,0]</code> is the top-left corner; point <code>[width, height]</code> is the bottom-right corner - the element's rotation does not affect this).
<code>setPosition(view: dia.ElementView, coordinates: g.Point): void;</code>	The method is executed each time the handle is moved. It's supposed to set the new model value derived from the current pointer <code>coordinates</code> (defined in the element model coordinate system)
<code>resetPosition(view: dia.ElementView): void;</code>	The method is executed when the handle is double-clicked. It's supposed to reset the model value back to a default.

Here's an ES5 example of a tool, which provides the user a way to modify the border radius of an rectangle.

```
var RadiusTool = elementTools.Control.extend({

  getPosition: function(view) {
    var model = view.model;
```



```

        var size = model.size();
        var tilt = model.topRy();
        return { x: size.width / 2, y: 2 * tilt };
    },

    setPosition: function(view, coordinates) {
        var model = view.model;
        var size = model.size();
        var tilt = Math.min(Math.max(coordinates.y, 0), size.height / 2);
        model.topRy(tilt, { ui: true, tool: this.cid });
    },

    resetPosition: function(view) {
        var radius = this.options.defaultRadius || 0;
        model.attr(['body'], { rx: radius, ry: radius });
    }
});

```

TypeScript example of the same control:

```

interface RadiusControlOptions extends elementTools.Control.Options {
    defaultRadius?: number;
}

class RadiusControl extends elementTools.Control {

    protected getPosition(view: dia.ElementView): dia.Point {
        const { model } = view;
        const radius = model.attr(['body', 'ry']) || 0;
        return { x: 0, y: radius };
    }

    protected setPosition(view: dia.ElementView, coordinates: dia.Point): void {
        const { model } = view;
        const { height } = model.size();
        const radius = Math.min(Math.max(coordinates.y, 0), height / 2);
        model.attr(['body'], { rx: radius, ry: radius });
    }

    protected resetPosition(view): void {
        const { model } = view;
        const { defaultRadius = 0 } = this.options;
        model.attr(['body'], { rx: defaultRadius, ry: defaultRadius });
    }
}

```

Add the tool to a rectangle element to allow the user adjust its "border radius".

```
rectangle.findView(paper).addTools(new dia.ToolsView({
  tools: [
    new RadiusTool({ handleAttributes: { fill: 'orange' }})
  ]
}));
```

elementTools.HoverConnect

The `HoverConnect` tool allows the user to create links from elements in a drag & drop fashion. The tool extends the `Connect` tool. The difference is that the button appears along the invisible track path (in the shape defined by `trackPath`) at the point where the user moves the mouse over the track. It accepts additional arguments, which can be passed as an object to the hover connect tool constructor:

trackWidth	<i>number</i>	The thickness of the track path. The default is <code>15</code> .
trackPath	<i>string</i> (view: dia.ElementView) => string	The SVG list of <u><code>Path Commands</code></u> . It may contain <u><code>calc expressions</code></u> which are evaluated in the context of the related element view (or model when <code>useModelGeometry</code> is set to <code>true</code>). The default is <code>'M 0 0 H calc(w) V calc(h) H 0 Z'</code> (a rectangle in size of the element).

Example:

```
const hoverButton = new joint.elementTools.HoverConnect({
  useModelGeometry: true,
  trackPath: (view) => view.model.attr(['body', 'd'])
});
```

elementTools.Remove

The `Remove` element tool renders a remove button at a given position above the element. It accepts a few additional arguments, which can be passed as an object to the element tool constructor:

x	<i>string number</i>	Use percentage strings (e.g. <code>'40%'</code>) to position the button relatively to the element width/height. A number means distance from the top-left corner of the element. Default is <code>0</code> .
y		
rotate	<i>boolean</i>	Should the button rotate according to the element angle around the element center? Default is <code>false</code> .

offset	<i>object</i>	Additional offset of the button from the position defined by <code>x</code> and <code>y</code> . Default is <code>{ x: 0, y: 0 }</code> .
action	<i>function</i>	What should happen when the user clicks the remove button? Default: <pre>function(evt, elementView, toolView) { elementView.model.remove({ ui: true, tool: toolView.cid }); }</pre>
markup	<i><u>JSONMarkup</u></i>	The markup of the button, provided in the JointJS JSON format . Default is <code>undefined</code> (no content).
scale	<i>number</i>	Scale the button up or down on a 2D plane. The default is <code>1</code> .

Example:

```
var removeButton = new joint.elementTools.Remove({
  focusOpacity: 0.5,
  rotate: true,
  // top-mid
  x: '50%',
  y: '0%',
  offset: { x: 10, y: 10 }
});
```

env.addTest

`env.addTest(name, fn)`

Add a custom feature-detection test where `name` is a string which uniquely identifies your feature test and `fn` is a function that returns `true` if the browser supports the feature, and `false` if it does not.

```
joint.env.addTest('customTest', function() {
  // Just as an example, we will always return true here.
  return true;
});

if (joint.env.test('customTest')) {
```

```
// Feature is supported.  
}
```

env.test

env.test(name)

Tests whether the browsers supports the given feature or not. Returns **true** if the feature is supported, otherwise it returns **false**.

```
if (joint.env.test('someFeature')) {  
  // Feature is supported.  
}
```

JointJS ships with the following tests:

- **svgforeignobject** - Tests whether the browser supports foreignObject.

highlighters

Highlighters can be used to provide visual emphasis to an element; during user interactions for example.

Highlighters inherit from dia.HighlighterView.

In the above demo, we listen to the paper for the `element:pointerclick` event and highlight the entire element we clicked on.

```
paper.on('element:pointerclick', (elementView) => {  
  joint.highlighters.mask.add(elementView, { selector: 'root' }, 'my-element-  
highlight', {  
    deep: true,  
    attrs: {  
      'stroke': '#FF4365',  
      'stroke-width': 3  
    }  
  });  
});
```

We also listen to the paper for the `element:magnet:pointerclick` event and highlight the port we clicked on.

```
paper.on('element:magnet:pointerclick', (elementView, magnet, evt) => {  
  // Prevent highlighting the body of the element
```

```

    evt.stopPropagation();
    // Find the port ID of the selected magnet
    const port = cellView.findAttribute('port', magnet);
    joint.highlighters.mask.add(elementView, { port }, 'my-port-highlight', {
      attrs: {
        'stroke': '#FF4365',
        'stroke-width': 3
      }
    });
  });
});

```

Then we listen for the `link:pointerclick` event and highlight the line node of the link we clicked on.

```

paper.on('link:pointerclick', (linkView) => {
  joint.highlighters.mask.add(linkView, { selector: 'line' }, 'my-link-highlight',
  {
    // Draw the highlighter under the LinkView
    layer: 'back',
    attrs: {
      'stroke': '#FF4365',
      'stroke-width': 3,
      'stroke-linecap': 'square'
    }
  });
});

```

Next we set up a listener for a custom event `link:label:pointerdown` and highlight the label we clicked on.

A custom paper event can be defined with [event attribute](#).

```

paper.on('link:label:pointerdown', (linkView, evt) => {
  // Prevent highlighting the line of the link
  evt.stopPropagation();
  // Find the index of the selected label
  const label = cellView.findAttribute('label-idx', evt.target);
  joint.highlighters.mask.add(linkView, { label }, 'my-label-highlight', {
    // Decrease the gap between the label and highlighter
    padding: 1,
    attrs: {
      'stroke': '#FF4365',
      'stroke-width': 3
    }
  });
});

```

Generally, it's possible to highlight a cell (Element or Link) or a part of the cell calling the `add()` method of a highlighter.

```

// Add Mask Highlighter with ID `my-mask-highlighter` to the CellView.
// Note: `root` is a shortcut for `{ selector: 'root' }`
joint.highlighters.mask.add(cellView, 'root', 'my-mask-highlighter', {
  deep: true
});

// Add class name `my-highlight` to the CellView's body node.
joint.highlighters.addClass.add(cellView, 'body', 'my-class-highlighter', {
  className: 'my-highlight'
});

// Add Stroke Highlighter to a specific port node (using default options).
joint.highlighters.stroke.add(cellView, { port: 'port-id-1', selector: 'portBody' },
'my-port-highlighter');

```

To unhighlight a cell, call the `remove()` method.

```

// Remove all highlighters from the CellView
joint.dia.HighlighterView.remove(cellView);

// Remove the highlighter with ID `my-highlighter` from the CellView
joint.dia.HighlighterView.remove(cellView, 'my-highlighter');

// Remove all Mask highlighters from the cellView
joint.highlighters.mask.remove(cellView);

// Remove Stroke Highlighter with ID `my-highlighter` from the cellView.
joint.highlighters.stroke.remove(cellView, 'my-highlighter');

// If you have a reference to a highlighter, calling its prototype `remove()` method
is also valid.
const highlighter = joint.dia.HighlighterView.get(cellView, 'my-highlighter');
highlighter.remove();

```

To see if a cell has a highlighter, call the `get()` method.

```

// Get all the highlighters (an array) from the CellView
joint.dia.HighlighterView.get(cellView);

// Get the highlighter with ID `my-highlighter` from the CellView
joint.dia.HighlighterView.get(cellView, 'my-highlighter');

// Get all Mask highlighters from the cellView
joint.highlighters.mask.get(cellView);

// Get Stroke Highlighter with ID `my-highlighter` from the cellView.

```

```
// If there is no such highlighter (ID or Type does not match, `null` is returned).  
joint.highlighters.stroke.get(cellView, 'my-highlighter');
```

When adding a highlighter, prefer using a node selector over a reference to an actual node.

```
// Prefer this:  
joint.highlighters.mask.add(elementView, { port: 'port1' }, 'port-highlight');  
// Over this:  
joint.highlighters.mask.add(elementView, elementView.findPortNode('port1'), 'port-highlight');
```

- Using the node selector supports asynchronous rendering. No need to wait for an element to be rendered, when adding the highlighter.
- Using the node reference highlights the specific node only. If the view is redrawn, the node could have been replaced with another node which wouldn't be highlighted.

If a node (determined by the node selector) stops to exist (e.g. a port is removed) the `cell:highlight:invalid` event is triggered on the paper.

```
// If we don't want the highlighter wait for the port to reappear, we can remove it when the event occurs.  
paper.on('cell:highlight:invalid', (cellView, id) => {  
  joint.dia.HighlighterView.remove(cellView, id);  
})
```

highlighters.addClass

Toggles a class name on an arbitrary cell view's SVG node.

Available options:

- **className** - the class name to toggle on the cell's node

Example usage:

```
joint.highlighters.addClass.add(cellView, 'root', 'my-highlighter-id', {  
  className: 'some-custom-class'  
});
```

highlighters.custom

New highlighter can be defined by extending the `joint.dia.HighlighterView` base class.

```
// A simple highlighter, which draws a rectangle over the entire CellView
const MyHighlighter = joint.dia.HighlighterView.extend({

  tagName: 'rect',

  attributes: {
    'stroke': 'blue',
    'fill': 'blue',
    'fill-opacity': 0.1,
    'pointer-events': 'none'
  },

  options: {
    padding: 5
  },

  // Method called to highlight a CellView
  highlight(cellView, _node) {
    const { padding } = this.options;
    const bbox = cellView.model.getBBox();
    // Highlighter is always rendered relatively to the CellView origin
    bbox.x = bbox.y = 0;
    // Increase the size of the highlighter
    bbox.inflate(padding);
    this.vel.attr(bbox.toJSON());
  },

  // Method called to unhighlight a CellView
  unhighlight(_cellView, _node) {
    // Cleaning required when the highlighter adds
    // attributes/nodes to the CellView or Paper.
    // This highlighter only renders a rectangle.
  }

});

MyHighlighter.add(el1.findView(paper), 'root', 'my-custom-highlighter', {
  layer: 'front', // "layer" is an option inherited from the base class
  padding: 10
});
```

highlighters.list

Adds a list of arbitrary SVGElements to the cell view.

Available options:

- **layer** - the stacking order of the highlighter. See [dia.Highlighter](#) for supported values.
- **attribute** - the name of the model attribute to read the list items from (and automatically update the list when the attribute changes). The option is mandatory and it's an array of items (each item is then passed as is to the `createListItem` method).
- **gap** - the space between items. Expects a `number`.
- **size** - the size of each item. It's either a `number` or an object with `width` and `height` properties.
- **direction** - specifies how list items are placed in the list container defining the main axis. It's either `row` (for growth on `x` coordinate) or `column` (for growth on `y` coordinate).
- **position** - specifies the position and the alignment of the list container. The value is a position name. Using one of the position values sets the list anchor and the position within the item view to that value. For instance the `bottom-right` value positions the list by its bottom-right corner to the bottom-right corner of the element.
- **margin** - the space around the list container. It's a `number` or an object with `left`, `top`, `right`, `bottom` properties.

Methods:

highlighters.list.createListItem

`protected createListItem(item, size, currentListItem)`

It accepts the `item` (a value at a position from the model attributes's array), the normalized `size` object and `currentListItem` SVGElement if such a node was created in a previous call. It returns an SVGElement.

It's an abstract method meant to be overridden.

The list optimizes the updates i.e. the method is not executed if the particular item does not change.

Example usage:

```
class StatusList extends highlighters.list {  
  
    // list of colored ellipses  
    createListItem(color, { width, height }) {  
        const { node } = V('ellipse', {  
            'rx': width / 2,  
            'ry': height / 2,  
            'cx': width / 2,  
            'cy': height / 2,  
            'fill': color,  
            'stroke': '#333',  
            'stroke-width': 2,  
        });  
        return node;  
    }  
}
```

```

}

StatusList.add(element.findView(paper), 'root', 'status-list', {
  attribute: 'status',
  size: 10,
  gap: 2
});

element.set('status', ['green', 'green', 'red'])

```

highlighters.mask

Adds a stroke around an arbitrary cell view's SVG node.

It's an alternative to the stroke highlighter.

Pros:

- **more precise** - equidistant padding along the node
- **more powerful** - ability to draw a stroke around the descendant nodes or draw a stroke around a path with no fill

Cons:

- **lesser performance** - uses SVG masking to find the outline
- **no fill** - the color inside the mask must stay transparent.
- **no dasharray** - the stroke patterns of dashes and gaps are not applicable.

Available options:

- **layer** - the stacking order of the highlighter. See dia.Highlighter for supported values.
- **padding** - the space between the stroke and the node
- **deep** - if the node is a container element (e.g. *SVGGroupElement*), draw the mask around its descending nodes. Otherwise a rectangular mask is drawn.
- **attrs** - an object with SVG attributes to be set on the mask element
- **maskClip** - it determines the area which is affected by a mask. The painted content of an element must be restricted to this area. It defaults to 20 (20 pixels around the node's bounding box).

Example usage:

```

joint.highlighters.mask.add(cellView, 'body', 'my-highlighter-id', {
  padding: 5,
  attrs: {
    'stroke-width': 3,
    'stroke': '#FF0000',
    // round the mask at the corners of paths

```

```

        'stroke-linejoin': 'round',
        // round the mask at the end of open subpaths
        'stroke-linecap': 'round'
        // to change the opacity use `rgba` color format
        // 'stroke': 'rgba(255, 0, 0, 0.5)',
        // this would affect the opacity of the stroke and the padding
        // 'stroke-opacity': 0.5
    }
});

```

highlighters.opacity

Changes the opacity of an arbitrary cell view's SVG node.

When a cell is highlighted with the opacity highlighter, the node determined by the selector is given the `.joint-highlight-opacity` class. To customize the look of this highlighted state, you can add custom CSS rules that affect this class name.

This highlighter does not currently have any options.

Example usage:

```

joint.highlighters.opacity.add(cellView, 'body', 'my-highlighter-id');

```

highlighters.stroke

Adds a stroke around an arbitrary cell view's SVG node.

Available options:

- **layer** - the stacking order of the highlighter. See [dia.Highlighter](#) for supported values.
- **padding** - the space between the stroke and the element
- **rx** - the stroke's border radius on the x-axis
- **ry** - the stroke's border radius on the y-axis
- **attrs** - an object with SVG attributes to be set on the highlighter element
- **useFirstSubpath** - draw the stroke by using the first subpath of the target `el` compound path.

Example usage:

```

joint.highlighters.stroke.add(cellView, 'body', 'my-highlighter-id', {
    padding: 10,
    rx: 5,

```

```

    ry: 5,
    useFirstSubpath: true,
    attrs: {
      'stroke-width': 3,
      'stroke': '#FF0000'
    }
  });

```

layout.DirectedGraph

Automatic layout of directed graphs. This plugin uses the open-source (MIT license) [Dagre](#) library internally. It provides a wrapper so that you can call the layout directly on JointJS graphs.

Note that you must include both the [Dagre](#) and [Graphlib](#) libraries as dependencies if you wish to use the `layout.DirectedGraph` plugin.

Usage

`joint.layout.DirectedGraph` plugin exposes the `joint.layout.DirectedGraph.layout(graphOrElements, opt)` function. The first parameter `graphOrElements` is the `joint.dia.Graph` or an array of `joint.dia.Elements` we want to layout. The second parameter `options` is an object that contains various options for configuring the layout.

```

var graphBBBox = joint.layout.DirectedGraph.layout(graph, {
  nodeSep: 50,
  edgeSep: 80,
  rankDir: "TB"
});
console.log('x:', graphBBBox.x, 'y:', graphBBBox.y)
console.log('width:', graphBBBox.width, 'height:', graphBBBox.height);

```

A full blog post explaining this example in more detail can be found [here](#).

Example with clusters

Configuration

The following table lists options that you can pass to the `joint.layout.DirectedGraph.layout(graph, opt)` function:

nodeSep	a number of pixels representing the separation between adjacent nodes in the same rank

edgeSep	a number of pixels representing the separation between adjacent edges in the same rank
rankSep	a number of pixels representing the separation between ranks
rankDir	direction of the layout (one of <code>"TB"</code> (top-to-bottom) / <code>"BT"</code> (bottom-to-top) / <code>"LR"</code> (left-to-right) / <code>"RL"</code> (right-to-left))
marginX	number of pixels to use as a margin around the left and right of the graph.
marginY	number of pixels to use as a margin around the top and bottom of the graph.
ranker	Type of algorithm to assign a rank to each node in the input graph. Possible values: <code>'network-simplex'</code> (default), <code>'tight-tree'</code> or <code>'longest-path'</code> .
resizeClusters	set to <code>false</code> if you don't want parent elements to stretch in order to fit all their embedded children. Default is <code>true</code> .
clusterPadding	A gap between the parent element and the boundary of its embedded children. It could be a number or an object e.g. <code>{ left: 10, right: 10, top: 30, bottom: 10 }</code> . It defaults to <code>10</code> .
setPosition(element, position)	a function that will be used to set the position of elements at the end of the layout. This is useful if you don't want to use the default <code>element.set('position', position)</code> but want to set the position in an animated fashion via <u>transitions</u> .
setVertices(link, vertices)	If set to <code>true</code> the layout will adjust the links by setting their vertices. It defaults to <code>false</code> . If the option is defined as a function it will be used to set the vertices of links at the end of the layout. This is useful if you don't want to use the default <code>link.set('vertices', vertices)</code> but want to set the vertices in an animated fashion via <u>transitions</u> .
setLabels(link, labelPosition, points)	If set to <code>true</code> the layout will adjust the labels by setting their position. It defaults to <code>false</code> . If the option is defined as a function it will be used to set the labels of links at the end of the layout. Note: Only the first label (<code>Link.label(0);</code>) is positioned by the layout.
dagre	Optional. The DirectedGraph plugin has the dependency to the <u>dagre</u> lay-outing tool. The DirectedGraph is expecting to have the <code>dagre</code> variable in the global namespace, you can, however, pass your own instance of the <code>dagre</code> using this property.
graphlib	Optional. The DirectedGraph plugin has the dependency to the <u>graphlib</u> - a directed multi-graph library. The DirectedGraph is expecting to have the <code>graphlib</code> variable

	in the global namespace, you can, however, pass your own instance of the <code>graphlib</code> using this property.
exportElement(element)	Convert element attributes into dagre <u>node</u> attributes. By default, it returns the <u>element attributes</u> below.
exportLink(link)	Convert link attributes into dagre <u>edge</u> attributes. By default, it returns the <u>link attributes</u> below.

Additionally, the layout engine takes into account some properties on elements/links to fine tune the layout further. These are:

size	element	An object with <code>`width`</code> and <code>`height`</code> properties representing the size of the element.
minLen	link	The number of ranks to keep between the source and target of the link.
weight	link	The weight to assign edges. Higher weight edges are generally made shorter and straighter than lower weight edges.
labelPosition	link	Where to place the label relative to the edge. <code>'l'</code> = left, <code>'c'</code> = center (default), <code>'r'</code> = right.
labelOffset	link	How many pixels to move the label away from the edge. Applies only when <code>labelPosition</code> is left or right.
labelSize	link	The width and height of the edge label in pixels. e.g. <code>{ width: 100, height: 50 }</code>

The `layout()` function returns a bounding box (`g.Rect`) of the resulting graph.

API

The `layout.DirectedGraph` plugins extends the `joint.dia.Graph` type with functions that make it easy to convert graphs to and from the Graphlib graph format. This allows you to easily use many of the graph algorithms provided by this library.

toGraphLib()	Convert the JointJS <code>joint.dia.Graph</code> object to the Graphlib graph object. <pre> var graph = new joint.dia.Graph; // ... populated the graph with elements connected with links graphlib.alg.isAcyclic(graph.toGraphLib()) // true if the graph is acyclic </pre>

	Please note, the <code>toGraphLib</code> has the dependency to the graphlib - a directed multi-graph library. You can either have the <code>graphlib</code> in the global namespace or you can pass the <code>graphlib</code> instance as an option property.
<code>fromGraphLib(glGraph, opt)</code>	Convert the Graphlib graph representation to JointJS <code>joint.dia.Graph</code> object. <code>opt.importNode</code> and <code>opt.importEdge</code> are functions that accept a Graphlib node and edge objects and should return JointJS element/link. <pre> var g = new graphlib.Graph(); g.setNode(1); g.setNode(2); g.setNode(3); g.setEdge(1, 2); g.setEdge(2, 3); var graph = new joint.dia.Graph; graph.fromGraphLib(g, { importNode: function(node) { return new joint.shapes.basic.Rect({ position: { x: node.x, y: node.y }, size: { width: node.width, height: node.height } }); }, importEdge: function(edge) { return new joint.dia.Link({ source: { id: edge.v }, target: { id: edge.w } }); } }); </pre>

layout.Port

Port layouts are functions that accept an array of port's `args` and return an array of port positions. Positions are relative to the element model bounding box. For example if we have an element at position `{ x:10, y:20 }` with a relative port position `{ x:1, y:2 }`, the absolute port position will be `{ x:11, y:22 }`.

Port layout can be defined only at the `group` level. Optionally you can pass some additional arguments into the layout function via `args`. The `args` is the only way to adjust port layout from the port definition perspective.

```

const rect = new joint.shapes.standard.Rectangle({
  // ...
  ports: {

```

```

    groups: {
      'a': {
        position: {
          name: 'layoutType',
          args: {},
        }
      }
    },
    items: [
      // initialize 'rect' with port in group 'a'
      {
        group: 'a',
        args: {} // overrides `args` from the group level definition.
      },
      // ... other ports
    ]
  }
});

// ....
// add another port to group 'a'.
rect.addPort({ group: 'a' });

```

Pre-defined layouts:

On sides

A simple layout suitable for rectangular shapes. It evenly spreads ports along a single side.

name	string	Can be either 'left', 'right', 'top', 'bottom' .		
args	object	x	number	Overrides the x / y value calculated by the layout function. It can be defined as a percentage string e.g. '50%', <u>calc()</u> expression or a number
		y	number	
		dx	number	Added to the x value calculated by the layout function
		dy	number	Added to the y value calculated by the layout function
		angle	number	The port rotation angle.

```

{
  name: 'left',

```



```

    args: {
      x: 10,
      y: 10,
      angle: 30,
      dx: 1,
      dy: 1
    }
  }
}

```

Line

A layout which evenly spreads ports along a line defined by a `start` and `end` point.

name	string	'line'			
args	object	start	{ x: number string, y: number string }	The line starting point.	x and y can be a percentage string e.g. '50%', <code>calc()</code> expression, or a number
		end		The line end point	

```

{
  name: 'line',
  args: {
    start: { x: 10, y: 10 },
    end: { x: 'calc(w)', y: '50%' }
  }
}

```

Absolute

It lays a port out at the given position (defined as `x`, `y` coordinates or percentage of the element dimensions).

name	string	'absolute'			
args	object	x	number string	Sets the port's <code>x</code> coordinate. It can be defined as a percentage string e.g. '50%', <code>calc()</code> expression or a number	
		y			
		angle	number	The port rotation angle.	

```

{
  name: 'absolute',
  args: {
    x: '10%',
    y: 10,
    angle: 45
  }
}

```

Radial

Suitable for circular shapes. The `ellipseSpreads` evenly spreads ports along an ellipse. The `ellipse` spreads ports from the point at `startAngle` leaving gaps between ports equal to `step`.

name	string	Can be either <code>'ellipse'</code> or <code>'ellipseSpread'</code> .		
args	object	x	number	Overrides the <code>x / y</code> value calculated by the layout function. It can be defined as a percentage string e.g. <code>'50%'</code> , <code>calc()</code> expression or a number
		y	number	
		dx	number	Added to the <code>x</code> value calculated by the layout function
		dy	number	Added to the <code>y</code> value calculated by the layout function
		dr	number	Added to the port delta rotation
		startAngle	number	Default value is <code>0</code> .
		step	number	Default <code>360 / portsCount</code> for the <code>ellipseSpread</code> , <code>20</code> for the <code>ellipse</code>
		compensateRotation	boolean	set <code>compensateRotation: true</code> when you need to have ports in the same angle as an ellipse tangent at the port position.

```

{
  name: 'ellipseSpread',
  args: {
    dx: 1,
    dy: 1,

```

```

        dr: 1,
        startAngle: 10,
        step: 10,
        compensateRotation: false
    }
}

```

Custom layout

An alternative for built-in layouts is providing a function directly, where the function returns an array of port transformations (position and angle).

```

/**
 * @param {Array<object>} portsArgs
 * @param {g.Rect} elBBox shape's bounding box
 * @param {object} opt Group options
 * @returns {Array<joint.layout.Port.Transformation>}
 */
function(portsArgs, elBBox, opt) {
    // Distribute ports along the sinusoid
    return portsArgs.map((portArgs, index) => {
        const step = -Math.PI / 8;
        const y = Math.sin(index * step) * 50;
        return {
            x: index * 12,
            y: y + elBBox.height,
            angle: 0
        };
    });
}

```

Port layouts demo

layout.PortLabel

Port label layout functions calculate port label positions relatively to port positions.

Pre-defined port label layouts

On Sides

Simple label layout suitable for rectangular shapes. It places the label on arbitrary side of a port. The `args` object is optional.

name	<i>string</i>	Can be either <code>'left'</code> , <code>'right'</code> , <code>'top'</code> , or <code>'bottom'</code> .		
args	<i>object</i>	x	number	Overrides the <code>x</code> value calculated by the layout function.
		y	number	Overrides the <code>y</code> value calculated by the layout function.
		angle	number	The port label rotation angle (in degrees in clockwise direction).
		attrs	object	JointJS-style attributes applied on label's DOM element and its children. The same notation as the <code>attrs</code> property on Element . Overridden by port group <code>attrs</code> , if provided.

```
label: {
  position: {
    name: 'right',
    args: {
      x: 0,
      y: 0,
      angle: 0,
      attrs: {}
    }
  }
}
```

Inside/Outside

Places the label inside or outside of a rectangular shape. Where 'oriented' versions rotate the text towards the element center. The `args` object is optional.

name	<i>string</i>	Can be either <code>'inside'</code> , <code>'outside'</code> , <code>'insideOriented'</code> , or <code>'outsideOriented'</code> .		
args	<i>object</i>	offset	number	Offset in direction from the shape's center. Overridden by <code>x</code> and <code>y</code> values, if provided.
		x	number	Overrides the <code>x</code> value calculated by the layout function.
		y	number	Overrides the <code>y</code> value calculated by the layout function.

		angle	number	The port label rotation angle (in degrees in clockwise direction). Overrides the value calculated by the <code>'insideOriented'</code> and <code>'outsideOriented'</code> layout functions.
		attrs	object	JointJS-style attributes applied on label's DOM element and its children. The same notation as the <code>attrs</code> property on Element . Overridden by port group <code>attrs</code> , if provided.

```

label: {
  position: {
    name : 'outsideOriented',
    args: {
      offset: 10,
      attrs: {}
    }
  }
}

```

Radial

Places the label outside of a circular shape. Where the 'oriented' version rotates the text towards the element center. The `args` object is optional.

name	<i>string</i>	Can be either <code>'radial'</code> , or <code>'radialOriented'</code> .		
args	<i>object</i>	offset	number	Offset in direction from the shape's center. Overridden by <code>x</code> and <code>y</code> values, if provided.
		x	number	Overrides the <code>x</code> value calculated by the layout function.
		y	number	Overrides the <code>y</code> value calculated by the layout function.
		angle	number	The port label rotation angle (in degrees in clockwise direction). Overrides the value calculated by the <code>'radialOriented'</code> layout function.
		attrs	number	JointJS-style attributes applied on label's DOM element and its children. The same notation as the <code>attrs</code> property on Element . Overridden by port group <code>attrs</code> , if provided.

```

label: {
  position: {
    name: 'radialOriented',
    args: {
      offset: 0,
      attrs: {}
    }
  }
}

```

Manual placement

It allows setting label position directly.

name	string	'manual'		
args	object	x	number	Sets the label's x coordinate.
		y	number	Sets the label's y coordinate.
		angle	number	The port label rotation angle (in degrees in clockwise direction).
		attrs	object	JointJS-style attributes applied on label's DOM element and its children. The same notation as the <code>attrs</code> property on Element . Overridden by port group <code>attrs</code> , if provided.

```

label: {
  position: {
    name: 'manual',
    args: {
      x: 10,
      y: 20,
      angle: 45,
      attrs: {}
    }
  }
}

```

Port label layout demo

linkAnchors

A link anchor of a link is a point on the reference link that this link wants to reach as its endpoint. Link anchors are set via a `linkAnchor` property provided within link end definitions (i.e. the objects provided to `link.source()` and `link.target()` functions). (If the reference object is an Element, JointJS looks at `anchor` property instead.)

There are several built-in linkAnchor functions in JointJS:

- `'connectionRatio'` - default link anchor at specified ratio at reference link
- `'connectionLength'` - link anchor at specified length along reference link
- `'connectionPerpendicular'` - link anchor that ensures an orthogonal route to the reference link
- `'connectionClosest'` - link anchor at the point on reference link that is closest to the last vertex on link path

Example:

```
link.source(link2, {
  linkAnchor: {
    name: 'connectionRatio',
    args: {
      ratio: 0.25
    }
  }
});
```

The default link anchor function is `'connectionRatio'`; this can be changed with the `defaultLinkAnchor` paper option. Example:

```
paper.options.defaultLinkAnchor = {
  name: 'connectionLength',
  args: {
    length: 20
  }
};
```

JointJS also contains mechanisms to define one's own custom link anchor functions.

linkAnchors.connectionClosest

The `'connectionClosest'` link anchor function places the anchor of the link at the point along the reference link that is the closest to the appropriate reference point of this link. (If we are calling this method for a source anchor, the

reference point is the first vertex; if there are no vertices, it is the target anchor. If we are calling this method for a target anchor, the reference point is the last vertex; if there are no vertices, it is the source anchor.)

Example:

```
link.source(link2, {
  linkAnchor: {
    name: 'connectionClosest'
  }
});
```

linkAnchors.connectionLength

The `'connectionLength'` link anchor function places the anchor of the link at a point a specified length along reference link (counting from reference link's source). It accepts one argument, which can be passed within the `linkAnchor.args` property:

length	<i>number</i>	The length at which to place the target anchor. Default is <code>20</code> , meaning a point 20 pixels from the source of the reference link.

Example:

```
link.source(link2, {
  linkAnchor: {
    name: 'connectionLength',
    args: {
      ratio: 50
    }
  }
});
```

linkAnchors.connectionPerpendicular

The `'connectionPerpendicular'` link anchor function tries to place the anchor of the link inside the view bbox so that the link is made orthogonal. The anchor is placed along two line segments inside the view bbox (between the centers of the top and bottom side and between the centers of the left and right sides). If it is not possible to place the link anchor so that the resulting link would be orthogonal, the anchor is placed at the point on the reference link that is the closest to the appropriate reference point on this link. (If we are calling this method for a source anchor, the reference point is the first vertex; if there are no vertices, it is the target anchor. If we are calling this method for a target anchor, the reference point is the last vertex; if there are no vertices, it is the source anchor.)

Example:

```
link.source(link2, {
  anchor: {
    name: 'connectionPerpendicular'
  }
});
```

When the link has no vertices, the other end cell's center is used as a reference point. By default, this means that a link using the `'connectionPerpendicular'` anchor “slides” alongside the source reference link while pointing to target's center. To invert this behavior, and have the link “slide” alongside the target reference link while pointing to source's center, pass a `priority` option to the target function:

```
link.target(link2, {
  priority: true,
  anchor: {
    name: 'connectionPerpendicular',
  }
});
```

linkAnchors.connectionRatio

The `'connectionRatio'` link anchor function places the anchor of the link at a point at a specified ratio of the reference link's length (from reference link's source). It accepts one argument, which can be passed within the `linkAnchor.args` property:

ratio	<i>number</i>	The length ratio of the target anchor. Default is <code>0.5</code> , meaning the midpoint of the reference link.

Example:

```
link.source(link2, {
  linkAnchor: {
    name: 'connectionRatio',
    args: {
      ratio: 0.25
    }
  }
});
```

linkAnchors.custom

New link anchor functions can be defined in the `joint.linkAnchors` namespace (e.g. `joint.linkAnchors.myLinkAnchor`) or passed directly as a function to the `linkAnchor` property of link source/target (or to the `defaultLinkAnchor` [option of a paper](#)).

In either case, the link anchor function must return the link anchor as a `Point`. The function is expected to have the form `function(endView, endMagnet, anchorReference, args)`:

endView	<i>dia.LinkView</i>	The LinkView to which we are connecting. The Link model can be accessed as <code>endView.model</code> ; this may be useful for writing conditional logic based on link attributes.
endMagnet	<i>null</i>	(Not used) This argument is only present to ensure that the signature of custom <code>linkAnchor</code> methods is the same as the signature of custom anchor methods .
anchorReference	<i>g.Point</i>	A reference to another component of the link path that may be necessary to find this anchor point. If we are calling this method for a source anchor, it is the first vertex; if there are no vertices, it is the target anchor. If we are calling this method for a target anchor, it is the last vertex; if there are no vertices, it is the source anchor...
	<i>SVGElement</i>	...if the anchor in question does not exist (yet), it is that link end's magnet. (The built-in methods usually use this element's center point as reference.)
args	<i>object</i>	An object with additional optional arguments passed to the link anchor method by the user when it was called (the <code>args</code> property).

linkTools

A link tool is a view that renders a certain type of control elements on top of the [LinkView](#) it is attached to; for example the [Vertices tool](#) creates an interactive handle above every vertex (these handles then allow the user to move and/or delete each vertex). Link tools all inherit from the `joint.dia.ToolView` [class](#). A collection of tools is added to a [ToolsView](#); a tools view is then added to the linkView with the `linkView.addTools()` [function](#).

The JointJS library comes with a collection of pre-made link tool definitions in the `joint.linkTools` namespace:

- `Vertices` - [adds handles above link vertices](#)
- `Segments` - [adds handles above link segments](#)

- SourceArrowhead - adds a handle above link source
- TargetArrowhead - adds a handle above link target
- SourceAnchor - adds a handle above link source anchor
- TargetAnchor - adds a handle above link target anchor
- Boundary - shows link bbox
- Remove - adds an interactive remove button

To create a new link tool, we call its constructor. Example:

```
var verticesTool = new joint.linkTools.Vertices({
  snapRadius: 10
});
```

In addition, the `joint.linkTools` namespace contains a customizable button class:

- Button - adds a customizable button

Example:

```
var infoTool = new joint.linkTools.Button({
  markup: [{
    tagName: 'circle',
    selector: 'button',
    attributes: {
      'r': 7,
      'fill': '#001DFF',
      'cursor': 'pointer'
    }
  }, {
    tagName: 'path',
    selector: 'icon',
    attributes: {
      'd': 'M -2 4 2 4 M 0 3 0 0 M -2 -1 1 -1 M -1 -4 1 -4',
      'fill': 'none',
      'stroke': '#FFFFFF',
      'stroke-width': 2,
      'pointer-events': 'none'
    }
  }],
  distance: 60,
  offset: 0,
  action: function(evt) {
    alert('View id: ' + this.id + '\n' + 'Model id: ' + this.model.id);
  }
});
```

All of the built-in link tools accept the following optional argument, in addition to their own arguments:

focusOpacity	<i>number</i>	What should be the opacity of the tool when it is focused (e.g. with the <code>toolView.focus</code> function)? Default is undefined , meaning that the tool's opacity is kept unchanged.

Example:

```
var verticesTool = new joint.linkTools.Vertices({
  focusOpacity: 0.5
});
```

linkTools.Boundary

The `Boundary` link tool renders a rectangular border to show the bounding box of the link. It accepts a few additional arguments, which can be passed as an object to the link tool constructor:

padding	<i>number object</i>	This option determines whether the boundary area should be visually inflated and if so, by how much. Default is 10 ({ <code>left: 10</code> , <code>top: 10</code> , <code>right: 10</code> , <code>bottom: 10</code> }).
useModelGeometry	<i>boolean</i>	If this option is set to true , the position of the boundary is calculated based on the dimensions of the link model.

Example:

```
var boundaryTool = new joint.linkTools.Boundary({
  focusOpacity: 0.5,
  padding: 20,
  useModelGeometry: true
});
```

linkTools.Button

The `Button` link tool allows you to have a custom button rendered at a given position along the link. It accepts five additional arguments, which can be passed as an object to the link tool constructor:

distance	<i>number</i>	Distance at which the button should be placed. Negative numbers are accepted; then the distance is counted from the end of the link. Default is 0 .

	<i>string</i>	Percentage strings (e.g. <code>'40%'</code>) are also accepted.
rotate	<i>boolean</i>	Should the button rotate according to the slope of the link at the position specified by distance ? Default is <code>false</code> .
offset	<i>number</i>	Relative offset of the button from the link. Positive numbers mean that the button should be offset to the right of the link (relative to the direction from source to target); negative numbers mean that the button should be offset to the left of the link (relative to the direction from source to target). Default is <code>0</code> .
action	<i>function</i>	What should happen when the user clicks the button? Default is <code>undefined</code> (no interaction). The callback function is expected to have the signature <code>function(evt, linkView, buttonView)</code> where <code>evt</code> is a <code>jQuery.Event</code> <u>object</u> . The related link view is available inside the function as <code>this</code> . The link model is available as <code>this.model</code> .
markup	<i><u>JSONMarkup</u></i>	The markup of the button, provided in the <u>JointJS JSON format</u> . Default is <code>undefined</code> (no content).
scale	<i>number</i>	Scale the button up or down on a 2D plane. The default is <code>1</code> .

Example of a useful custom info button:

```
var infoButton = new joint.linkTools.Button({
  focusOpacity: 0.5,
  distance: 60,
  action: function(evt) {
    alert('View id: ' + this.id + '\n' + 'Model id: ' + this.model.id);
  },
  markup: [{
    tagName: 'circle',
    selector: 'button',
    attributes: {
      'r': 7,
      'fill': '#001DFF',
      'cursor': 'pointer'
    }
  }, {
    tagName: 'path',
    selector: 'icon',
    attributes: {
      'd': 'M -2 4 2 4 M 0 3 0 0 M -2 -1 1 -1 M -1 -4 1 -4',
      'fill': 'none',
```

```

        'stroke': '#FFFFFF',
        'stroke-width': 2,
        'pointer-events': 'none'
    }
  ]
});

```

The `linkTools.Button` class can also be extended, to create a reusable custom button type. Then, a new instance of the custom button type can be obtained by calling its constructor:

```

var InfoButton = joint.linkTools.Button.extend({
  name: 'info-button',
  options: {
    focusOpacity: 0.5,
    distance: 60,
    action: function(evt) {
      alert('View id: ' + this.id + '\n' + 'Model id: ' + this.model.id);
    },
    markup: [{
      tagName: 'circle',
      selector: 'button',
      attributes: {
        'r': 7,
        'fill': '#001DFF',
        'cursor': 'pointer'
      }
    }, {
      tagName: 'path',
      selector: 'icon',
      attributes: {
        'd': 'M -2 4 2 4 M 0 3 0 0 M -2 -1 1 -1 M -1 -4 1 -4',
        'fill': 'none',
        'stroke': '#FFFFFF',
        'stroke-width': 2,
        'pointer-events': 'none'
      }
    }
  ]
});

var infoButton = new InfoButton();

```

linkTools.Connect

The `Connect` tool allows the user to create links in a drag & drop fashion. The tool extends the [Button](#) tool and accepts additional arguments, which can be passed as an object to the connect tool constructor:

magnet	<pre>string SVGElement (view: dia.ElementView) => SVGElement</pre>	Choose the source magnet of the element view which the new link should be connected to. <pre>'body' // as a string Selector</pre> <pre>linkView.el // as an SVGElement</pre> <pre>(linkView) => linkView.el // as a function</pre> The callback function is expected to have the signature <code>function(elementView, toolView)</code> where <code>elementView</code> is the element view with the tool attached. The connect button view is available inside the function as <code>this</code> and as the last parameter.

Example:

```
const connectButton = new joint.linkTools.Connect({
  rotate: true,
  distance: -20,
  offset: 20,
  magnet: 'body'
});
```

linkTools.HoverConnect

The `HoverConnect` tool allows the user to create links from other links in a drag & drop fashion. The tool extends the [Connect](#) tool. The difference is that the button appears along the invisible track path (in the shape of a `linkView`) at the point where the user moves the mouse over the track. It accepts additional arguments, which can be passed as an object to the hover connect tool constructor:

trackWidth	<pre>number</pre>	The thickness of the track path. The default is <code>15</code> .

Example:

```
const hoverButton = new joint.linkTools.HoverConnect({
  magnet: 'body',
```

```

        trackWidth: 10
    });

```

linkTools.Remove

The `Remove` link tool renders a remove button at a given position along the link. It accepts five additional arguments, which can be passed as an object to the link tool constructor:

distance	<i>number</i>	Distance at which the button should be placed. Negative numbers are accepted; then the distance is counted from the end of the link. Default is <code>60</code> .
	<i>string</i>	Percentage strings (e.g. <code>'40%'</code>) are also accepted.
rotate	<i>boolean</i>	Should the button rotate according to the slope of the link at the position specified by <code>distance</code> ? Default is <code>false</code> .
offset	<i>number</i>	Relative offset of the button from the link. Positive numbers mean that the button should be offset to the right of the link (relative to the direction from source to target); negative numbers mean that the button should be offset to the left of the link (relative to the direction from source to target). Default is <code>0</code> .
action	<i>function</i>	What should happen when the user clicks the remove button? Default: <pre> function(evt, linkView, toolView) { linkView.model.remove({ ui: true, tool: toolView.cid }); } </pre> The callback function is expected to have the signature <code>function(evt)</code> where <code>evt</code> is a <code>jQuery.Event</code> <u>object</u>. The button view is available inside the function as <code>this</code>; the button model is available as <code>this.model</code>.
markup	<u><i>JSONMarkup</i></u>	The markup of the button, provided in the <u>JointJS JSON format</u>. Default: <pre> [{ tagName: 'circle', selector: 'button', attributes: { 'r': 7, 'fill': '#FF1D00', 'cursor': 'pointer' } } </pre>

		<pre> }, { tagName: 'path', selector: 'icon', attributes: { 'd': 'M -3 -3 3 3 M -3 3 3 -3', 'fill': 'none', 'stroke': '#FFFFFF', 'stroke-width': 2, 'pointer-events': 'none' } }] } }</pre>
scale	<i>number</i>	Scale the button up or down on a 2D plane. The default is 1 .

Example:

```

var removeButton = new joint.linkTools.Remove({
  focusOpacity: 0.5,
  rotate: true,
  distance: -20,
  offset: 20
});
```

linkTools.Segments

The `Segments` link tool renders handles above all segments of the link (as determined by the [link connector](#)). It accepts four additional arguments, which can be passed as an object to the link tool constructor:

redundancyRemoval	<i>boolean</i>	If the user arranges two (or more) segments so that they lie on a single line, should the middle one(s) be considered redundant and removed? Default is true . Note that this setting is not applied until the user actually moves one of the segments in question; this means that segments can still be arranged in this “redundant” fashion, using the <code>link.vertices</code> function , for example.
segmentLengthThreshold	<i>number</i>	The minimum segment length for which to display a segment handle (to prevent the handle from overflowing its segment). Default is 40 .
snapRadius	<i>number</i>	While the user is moving the segment, from how far away should the segment snap in order to arrange itself in line with another segment? Default is 10 .

snapHandle	<i>number</i>	If the <code>snapRadius</code> option is set to <code>true</code> and the segment is snapped in place while the user moves the segment handle, should the handle follow the user pointer or should the handle stay snapped with the segment until it un-snaps? Default is <code>true</code> , meaning that the handle snaps with the segment.
stopPropagation	<i>boolean</i>	Should be events stopped from propagating to the paper? Default is <code>true</code> .
handleClass	<i>mvc.View</i>	The view for the segment handle. By default it uses <code>linkTools.Segments.SegmentHandle</code> class.
scale	<i>number</i>	Scale the segment handles up or down on a 2D plane. The default is <code>1</code> .

Example:

```
var segmentsTool = new joint.linkTools.Segments({
  focusOpacity: 0.5,
  redundancyRemoval: false,
  segmentLengthThreshold: 50,
  snapHandle: false,
  snapRadius: 10
});
```

linkTools.SourceAnchor

The `SourceAnchor` link tool renders a handle above the source anchor of the link (as determined by the [anchor function](#) applied on [link source](#)). It accepts several additional arguments, which can be passed as an object to the link tool constructor:

redundancyRemoval	<i>boolean</i>	If the user moves the anchor so that it lies on a single line with two (or more) vertices, should the middle one(s) be considered redundant and removed? Default is <code>true</code> . Note that this setting is not applied until the user actually moves the anchor; this means that the anchor and vertices can still be arranged in this “redundant” fashion, using the <code>link.vertices</code> function , for example.
restrictArea	<i>boolean</i>	Should the user only be allowed to move the anchor in a restricted area (the area of the bounding box of the source magnet)? Default is <code>true</code> .
areaPadding	<i>number</i>	If the <code>restrictArea</code> option is set to <code>true</code> , the user can only move the anchor in a restricted area (the area of the bounding box of the source magnet). JointJS shows this restriction by drawing a boundary around that area. This

		option determines whether this boundary area should be visually inflated and if so, by how much. Default is <code>10</code> . Note that this is a purely cosmetic setting; regardless of the provided value, the movement stays restricted to the original uninflated bounding box.												
snapRadius	<i>number</i>	While the user is moving the anchor, from how far away should the segment snap in order to arrange itself in line with the anchor reference? Default is <code>10</code> . (For link source, the anchor reference is the first vertex. If there are no vertices, it is the target anchor.)												
resetAnchor	<i>boolean object</i>	When the user double clicks the anchor tool, the following action should be performed: • <code>true</code> - remove the current source anchor from the link. It is a default. • <code>false</code> - do nothing • <code>{ name: [AnchorName], args: [AnchorArguments] }</code> - replace the current source anchor with the option value.												
snap	<i>function</i>	What snap function should be applied when the user moves the anchor? Default is a simple function that emulates the snapping behavior of <u>Vertices</u> and <u>Segments</u> link tools: If the value of one of the user pointer coordinates is within <code>snapRadius</code> of the value of a coordinate of the anchor reference, snap to the reference value. (For link source, the anchor reference is the first vertex. If there are no vertices, it is the target anchor.) The callback function must return the anchor as a <code>g.Point</code> and have the signature <code>function(coords, endView, endMagnet, endType, linkView, toolView):</code> <table> <tr> <td>coords</td><td><i>g.Point</i></td><td>A Point object recording the x-y coordinates of the user pointer when the snap function was invoked.</td></tr> <tr> <td>endView</td><td><i>dia.ElementView</i></td><td>The ElementView which contains the restricted area. The Element model can be accessed as <code>endView.model</code>; this may be useful for writing conditional logic based on element attributes.</td></tr> <tr> <td>endMagnet</td><td><i>SVGElement</i></td><td>The SVGElement in our page that contains the magnet (element/subelement/port) which serves as the restricted area.</td></tr> <tr> <td>endType</td><td><i>string</i></td><td>Is the end view the source (<code>'source'</code>) or target (<code>'target'</code>)</td></tr> </table>	coords	<i>g.Point</i>	A Point object recording the x-y coordinates of the user pointer when the snap function was invoked.	endView	<i>dia.ElementView</i>	The ElementView which contains the restricted area. The Element model can be accessed as <code>endView.model</code> ; this may be useful for writing conditional logic based on element attributes.	endMagnet	<i>SVGElement</i>	The SVGElement in our page that contains the magnet (element/subelement/port) which serves as the restricted area.	endType	<i>string</i>	Is the end view the source (<code>'source'</code>) or target (<code>'target'</code>)
coords	<i>g.Point</i>	A Point object recording the x-y coordinates of the user pointer when the snap function was invoked.												
endView	<i>dia.ElementView</i>	The ElementView which contains the restricted area. The Element model can be accessed as <code>endView.model</code> ; this may be useful for writing conditional logic based on element attributes.												
endMagnet	<i>SVGElement</i>	The SVGElement in our page that contains the magnet (element/subelement/port) which serves as the restricted area.												
endType	<i>string</i>	Is the end view the source (<code>'source'</code>) or target (<code>'target'</code>)												

				of the link?
		linkView	<i>dia.LinkView</i>	The LinkView to which the tool is attached. The Link model can be accessed as <code>linkView.model</code> ; this may be useful for writing conditional logic based on link attributes.
		toolView	<i>dia.ToolView</i>	This tool. May be useful for writing conditional logic based on tool options (e.g. the default algorithm refers to <code>toolView.options.snapRadius</code>).
scale	<i>number</i>	Scale the anchor element up or down on a 2D plane. The default is <code>1</code> .		

Example:

```
var sourceAnchorTool = new joint.linkTools.SourceAnchor({
  focusOpacity: 0.5,
  redundancyRemoval: false,
  restrictArea: false,
  snapRadius: 20
});
```

An example of a useful custom `snap` function is provided below. It snaps the anchor to the center of the closest side of the restricted area.

```
var snapAnchor = function(coords, endView, endMagnet) {

  // remove rotation of the restricted area
  var bbox = endView.getNodeUnrotatedBBox(endMagnet);
  var angle = endView.model.angle();
  var origin = endView.model.getBBox().center();
  coords.rotate(origin, angle);

  // identify the side nearest to pointer coords
  var anchor;
  var side = bbox.sideNearestToPoint(coords);
  switch (side) {
    case 'left': anchor = bbox.leftMiddle(); break;
    case 'right': anchor = bbox.rightMiddle(); break;
    case 'top': anchor = bbox.topMiddle(); break;
    case 'bottom': anchor = bbox.bottomMiddle(); break;
  }
}
```

```

    // rotate the anchor according to original rotation of restricted area
    return anchor.rotate(origin, -angle);
};

var sourceAnchorTool = new joint.linkTools.SourceAnchor({
  snap: snapAnchor;
});

```

If the user moves the anchor away from its original position, the anchor position may be reset by double-clicking the anchor handle.

linkTools.SourceArrowhead

The `SourceArrowhead` link tool renders an arrow-like handle above the source connection point of the link (as determined by the [connectionPoint function](#) applied on [link source](#)). It accepts a few additional arguments, which can be passed as an object to the link tool constructor:

scale	<i>number</i>	Scale the arrowhead element up or down on a 2D plane. The default is <code>1</code> .

Example:

```

var sourceArrowheadTool = new joint.linkTools.SourceArrowhead({
  focusOpacity: 0.5
});

```

linkTools.TargetAnchor

The `TargetAnchor` link tool renders a handle above the target anchor of the link (as determined by the [anchor function](#) applied on [link target](#)). It accepts several additional arguments, which can be passed as an object to the link tool constructor:

redundancyRemoval	<i>boolean</i>	If the user moves the anchor so that it lies on a single line with two (or more) vertices, should the middle one(s) be considered redundant and removed? Default is <code>true</code> . Note that this setting is not applied until the user actually moves the anchor; this means that the anchor and vertices can still be arranged in this “redundant” fashion, using the <code>link.vertices</code> function , for example.
restrictArea	<i>boolean</i>	Should the user only be allowed to move the anchor in a restricted area (the area of the bounding box of the target magnet)? Default is <code>true</code> .

areaPadding	<i>number</i>	If the <code>restrictArea</code> option is set to <code>true</code>, the user can only move the anchor in a restricted area (the area of the bounding box of the target magnet). JointJS shows this restriction by drawing a boundary around that area. This option determines whether this boundary area should be visually inflated and if so, by how much. Default is <code>10</code>. Note that this is a purely cosmetic setting; regardless of the provided value, the movement stays restricted to the original uninflated bounding box.									
snapRadius	<i>number</i>	While the user is moving the anchor, from how far away should the segment snap in order to arrange itself in line with the anchor reference? Default is <code>10</code>. (For link target, the anchor reference is the last vertex. If there are no vertices, it is the source anchor.)									
resetAnchor	<i>boolean</i> <i> object</i>	When the user double clicks the anchor tool, the following action should be performed: <code>true</code> - remove the current target anchor from the link. It is a default. <code>false</code> - do nothing <code>{ name: [AnchorName], args: [AnchorArguments] }</code> - replace the current target anchor with the option value.									
snap	<i>function</i>	What snap function should be applied when the user moves the anchor? Default is a simple function that emulates the snapping behavior of <u>Vertices</u> and <u>Segments</u> link tools: If the value of one of the user pointer coordinates is within <code>snapRadius</code> of the value of a coordinate of the anchor reference, snap to the reference value. (For link target, the anchor reference is the last vertex. If there are no vertices, it is the source anchor.) The callback function must return the anchor as a <code>g.Point</code> and have the signature <code>function(coords, endView, endMagnet)</code>: <table> <tr> <td>coords</td><td><i>g.Point</i></td><td>A Point object recording the x-y coordinates of the user pointer when the snap function was invoked.</td></tr> <tr> <td>endView</td><td><i>dia.ElementView</i></td><td>The ElementView which contains the restricted area. The Element model can be accessed as <code>endView.model</code>; this may be useful for writing conditional logic based on element attributes.</td></tr> <tr> <td>endMagnet</td><td><i>SVGElement</i></td><td>The SVGElement in our page that contains the magnet (element/subelement/port) which serves as the restricted area.</td></tr> </table>	coords	<i>g.Point</i>	A Point object recording the x-y coordinates of the user pointer when the snap function was invoked.	endView	<i>dia.ElementView</i>	The ElementView which contains the restricted area. The Element model can be accessed as <code>endView.model</code> ; this may be useful for writing conditional logic based on element attributes.	endMagnet	<i>SVGElement</i>	The SVGElement in our page that contains the magnet (element/subelement/port) which serves as the restricted area.
coords	<i>g.Point</i>	A Point object recording the x-y coordinates of the user pointer when the snap function was invoked.									
endView	<i>dia.ElementView</i>	The ElementView which contains the restricted area. The Element model can be accessed as <code>endView.model</code> ; this may be useful for writing conditional logic based on element attributes.									
endMagnet	<i>SVGElement</i>	The SVGElement in our page that contains the magnet (element/subelement/port) which serves as the restricted area.									

		<table> <tr> <td>endType</td><td><i>string</i></td><td>Is the end view the source (<i>'source'</i>) or target (<i>'target'</i>) of the link?</td></tr> <tr> <td>linkView</td><td><i>dia.LinkView</i></td><td>The LinkView to which the tool is attached. The Link model can be accessed as <code>linkView.model</code>; this may be useful for writing conditional logic based on link attributes.</td></tr> <tr> <td>toolView</td><td><i>dia.ToolView</i></td><td>This tool. May be useful for writing conditional logic based on tool options (e.g. the default algorithm refers to <code>toolView.options.snapRadius</code>).</td></tr> </table>	endType	<i>string</i>	Is the end view the source (<i>'source'</i>) or target (<i>'target'</i>) of the link?	linkView	<i>dia.LinkView</i>	The LinkView to which the tool is attached. The Link model can be accessed as <code>linkView.model</code> ; this may be useful for writing conditional logic based on link attributes.	toolView	<i>dia.ToolView</i>	This tool. May be useful for writing conditional logic based on tool options (e.g. the default algorithm refers to <code>toolView.options.snapRadius</code>).
endType	<i>string</i>	Is the end view the source (<i>'source'</i>) or target (<i>'target'</i>) of the link?									
linkView	<i>dia.LinkView</i>	The LinkView to which the tool is attached. The Link model can be accessed as <code>linkView.model</code> ; this may be useful for writing conditional logic based on link attributes.									
toolView	<i>dia.ToolView</i>	This tool. May be useful for writing conditional logic based on tool options (e.g. the default algorithm refers to <code>toolView.options.snapRadius</code>).									
scale	<i>number</i>	Scale the anchor element up or down on a 2D plane. The default is <i>1</i> .									

Example:

```
var targetAnchorTool = new joint.linkTools.TargetAnchor({
  focusOpacity: 0.5,
  redundancyRemoval: false,
  restrictArea: false,
  snapRadius: 20
});
```

An example of a useful custom `snap` function is provided below. It snaps the anchor to the center of the closest side of the restricted area.

```
var snapAnchor = function(coords, endView, endMagnet) {

  // remove rotation of the restricted area
  var bbox = endView.getNodeUnrotatedBBox(endMagnet);
  var angle = endView.model.angle();
  var origin = endView.model.getBBox().center();
  coords.rotate(origin, angle);

  // identify the side nearest to pointer coords
  var anchor;
  var side = bbox.sideNearestToPoint(coords);
  switch (side) {
    case 'left': anchor = bbox.leftMiddle(); break;
    case 'right': anchor = bbox.rightMiddle(); break;
    case 'top': anchor = bbox.topMiddle(); break;
    case 'bottom': anchor = bbox.bottomMiddle(); break;
  }
}
```

```

    }

    // rotate the anchor according to original rotation of restricted area
    return anchor.rotate(origin, -angle);
};

var targetAnchorTool = new joint.linkTools.TargetAnchor({
  snap: snapAnchor;
});

```

If the user moves the anchor away from its original position, the anchor position may be reset by double-clicking the anchor handle.

linkTools.TargetArrowhead

<>The `TargetArrowhead` link tool renders an arrow-like handle above the target connection point of the link (as determined by the [connectionPoint function](#) applied on [link target](#)). It accepts a few additional arguments, which can be passed as an object to the link tool constructor:

scale	<i>number</i>	Scale the arrowhead element up or down on a 2D plane. The default is <code>1</code> .

Example:

```

var targetArrowheadTool = new joint.linkTools.TargetArrowhead({
  focusOpacity: 0.5
});

```

linkTools.Vertices

The `Vertices` link tool renders handles above all [vertices of the link](#). It accepts three additional arguments, which can be passed as an object to the link tool constructor:

redundancyRemoval	<i>boolean</i>	If the user arranges three (or more) vertices so that they lie on a single line, should the middle one(s) be considered redundant and removed? Default is <code>true</code> . Note that this setting is not applied until the user actually moves one of the vertices in question; this means that vertices can still be arranged in this “redundant” fashion, using of the <code>link.vertices</code> function , for example.
snapRadius	<i>number</i>	While the user is moving the vertex, from how far away should the vertex snap in order to arrange itself perpendicularly to another vertex? Default is

		20 .
vertexAdding	<i>boolean</i>	Can the user add new vertices (by clicking a segment of the link)? Default is true .
stopPropagation	<i>boolean</i>	Should be events stopped from propagating to the paper? Default is true .
handleClass	<i>mvc.View</i>	The view for the vertex handle. By default it uses <code>linkTools.Vertices.VertexHandle</code> class.
scale	<i>number</i>	Scale the vertices handles up or down on a 2D plane. The default is 1 .

Example:

```
var verticesTool = new joint.linkTools.Vertices({
  focusOpacity: 0.5,
  redundancyRemoval: false,
  snapRadius: 10,
  vertexAdding: false,
});
```

mvc.Listener

A listener is an object that allows you to listen for events triggered by other objects and, if needed, remove them all at once.

```
// Example setup for switching between display mode and edit mode

class ViewController extends mvc.Listener {

  startListening() {
    const [{ paper }] = this.callbackArguments;
    this.listenTo(paper, 'element:mouseenter', (_appContext, elementView) => {
      joint.highlighters.mask.add(elementView, 'body', 'highlighted-element');
    });
    this.listenTo(paper, 'element:mouseleave', (_appContext, elementView) => {
      joint.highlighters.mask.remove(elementView, 'highlighted-element');
    });
  }
}

class EditController extends mvc.Listener {
```

```

startListening() {
    const [{ paper }] = this.callbackArguments;
    this.listenTo(paper, 'element:pointerdblclick', (appContext, elementView) =>
    {
        const { model } = elementView;
        if (appContext.graph.getConnectedLinks(model).length > 0) return;
        model.remove();
    });
}

const appContext = { paper, graph };

const viewController = new ViewController(appContext);
const editController = new EditController(appContext);

let editMode = false;
function toggleEditMode(canEdit = !editMode) {
    editMode = canEdit;
    if (editMode) {
        editController.startListening();
        viewController.stopListening();
    } else {
        viewController.startListening();
        editController.stopListening();
    }
}

// start app in view mode
toggleEditMode(false);

```

mvc.Listener.prototype.callbackArguments

listener.callbackArguments

An array of constructor arguments.

```

const listener = new mvc.Listener(callbackArg1, callbackArg2);
listener.callbackArguments; // [callbackArg1, callbackArg2]

```

mvc.Listener.prototype.listenTo

```
listener.listenTo(object, event, callback [, context])
```

Tell the listener to listen to a particular event on an **object**.

```
listener.listenTo(model, 'change', onModelChangeCallback);
```

To supply a **context** value for this when the callback is invoked, pass the optional last argument:

```
listener.listenTo(object, this.update, this);
```

The arguments of the constructor are passed to the **callback** when it is invoked.

```
const listener = new mvc.Listener();  
listener.listenTo(paper, 'element:pointerclick', (elementView) => {});
```

```
const listener = new mvc.Listener(callbackArgument);  
listener.listenTo(paper, 'element:pointerclick', (callbackArgument, elementView) => {});
```

```
const listener = new mvc.Listener(callbackArg1, callbackArg2);  
listener.listenTo(paper, 'element:pointerclick', (callbackArg1, callbackArg2, elementView) => {});
```

```
listener.listenTo(object, eventMap, [, context])
```

Tell the listener to listen to multiple events on an **object** (using event map syntax).

```
const listener = new mvc.Listener(callbackArgument);  
listener.listenTo(graph, {  
  'remove': (callbackArgument, cell) => {},  
  'add': (callbackArgument, cell) => {}  
});
```

mvc.Listener.prototype.stopListening

```
listener.stopListening()
```

Tell the listener to remove all of its registered callbacks.

routers

Routers take an array of link vertices and transform them into an array of route points that the link should go through. The `router` property of a link can be accessed with the `link.router()` [function](#).

The difference between `vertices` and the route is that the vertices are user-defined while the route is computed. The route inserts additional private vertices to complement user vertices as necessary (e.g. to make sure the route is orthogonal).

There are five built-in routers:

- `'manhattan'` - [smart orthogonal router](#)
- `'metro'` - [smart octolinear router](#)
- `'normal'` - [default simple router](#)
- `'orthogonal'` - [basic orthogonal router](#)
- `'rightAngle'` - [orthogonal router that goes in a given direction](#)

Example:

```
link.router('orthogonal', {  
  padding: 10  
});
```

The default router is `'normal'`; this can be changed with the `defaultRouter` [paper option](#). Example:

```
paper.options.defaultRouter = {  
  name: 'orthogonal',  
  args: {  
    padding: 10  
  }  
};
```

The `'manhattan'` and `'metro'` routers are our “smart routers”; they automatically avoid obstacles (elements) in their way.

The `'orthogonal'`, `'manhattan'` and `'rightAngle'` routers generate routes consisting exclusively of vertical and horizontal segments. The `'metro'` router generates routes consisting of orthogonal and diagonal segments.

JointJS also contains mechanisms to define one's own [custom routers](#).

Note that the modular architecture of JointJS allows mixing-and-matching routers with [connectors](#) as desired; for example, a link may be specified to use the `'jumpover'` connector on top of the `'manhattan'` router:

```
var link = new joint.shapes.standard.Link();  
link.source(rect);
```

```
link.target(rect2);
link.router('manhattan');
link.connector('jumpover');
```

route.custom

New routers can be defined in the `joint.routers` namespace (e.g. `joint.routers.myRouter`) or passed directly as a function to the `router` property of a link (or to the `defaultRouter` [option of a paper](#)).

In either case, the router function must return an array of route points that the link should go through (not including the start/end connection points). The function is expected to have the form `function(vertices, args, linkView):`

vertices	<i>Array<g.Point></i>	The vertices of the route.
args	<i>object</i>	An object with additional optional arguments passed to the router method by the user when it was called (the <code>args</code> property).
linkView	<i>dia.LinkView</i>	The LinkView of the connection. The Link model can be accessed as the <code>model</code> property; this may be useful for writing conditional logic based on link attributes.

Example of a router defined in the `joint.routers` namespace:

```
joint.routers.randomWalk = function(vertices, args, linkView) {

  var NUM_BOUNCES = args.numBounces || 20;

  vertices = joint.util.toArray(vertices).map(g.Point);

  for (var i = 0; i < NUM_BOUNCES; i++) {

    var sourceCorner = linkView.sourceBBox.center();
    var targetCorner = linkView.targetBBox.center();

    var randomPoint = g.Point.random(sourceCorner.x, targetCorner.x,
sourceCorner.y, targetCorner.y);
    vertices.push(randomPoint)
  }

  return vertices;
}

var link = new joint.shapes.standard.Link();
link.source(source);
link.target(target);
```

```
link.router('randomWalk', {
  numBounces: 10
});
```

An example of a router passed as a function is provided below. Notice that this approach does not enable passing custom args nor can it be serialized with the `graph.toJSON()` [function](#).

```
var link = new joint.shapes.standard.Link();
link.source(source);
link.target(target);

link.router(function(vertices, args, linkView) {

  var NUM_BOUNCES = 20;

  vertices = joint.util.toArray(vertices).map(g.Point);

  for (var i = 0; i < NUM_BOUNCES; i++) {

    var sourceCorner = linkView.sourceBBox.center();
    var targetCorner = linkView.targetBBox.center();

    var randomPoint = g.Point.random(sourceCorner.x, targetCorner.x,
    sourceCorner.y, targetCorner.y);
    vertices.push(randomPoint)
  }

  return vertices;
});
```

route.manhattan

The `'manhattan'` router is the smart version of the `'orthogonal'` router. It connects vertices with orthogonal line segments, inserting route points when necessary, while avoiding obstacles in its way. The router has useful options that determine how the algorithm behaves. These options can be passed as the `router.args` property:

step	<i>number</i>	Size of the imaginary grid followed by the <code>'manhattan'</code> pathfinding algorithm. Lower number -> higher complexity. The <code>'manhattan'</code> router performs best when <code>step</code> has the same value as <code>dia.Paper.option.gridSize</code> . Default is <code>10</code> .

padding	<i>number object</i>	Padding applied around start/end elements and obstacles. Default is the <code>step</code> value (see above). The <code>util.normalizeSides</code> function is used to understand the provided value. A single number is applied as padding to all sides of the elements. An object may be provided to specify values for <code>left</code>, <code>top</code>, <code>right</code>, <code>bottom</code>, <code>horizontal</code> and/or <code>vertical</code> sides.
maximumLoops	<i>number</i>	The maximum number of iterations of the pathfinding loop. If the number of iterations exceeds this maximum, pathfinding is aborted and the fallback router (<code>'orthogonal'</code>) is used instead. Higher number -> higher complexity. Default is <code>2000</code> .
maxAllowedDirectionChange	<i>number</i>	Maximum change of direction of the <code>'manhattan'</code> route, in degrees. Default is <code>90</code> .
perpendicular	<i>boolean</i>	Should the <code>linkView.perpendicular</code> option be in effect? This causes the router not to link precisely to the anchor of the end element but rather to a point close by that is orthogonal. This creates a clean connection between the element and the first/last vertex of the route. Default is <code>true</code> .
excludeEnds	<i>Array<string></i>	An array with strings <code>'source'</code> and/or <code>'target'</code> that tells the algorithm that the given end element(s) should not be considered as an obstacle. Default is <code>[]</code> (both are considered obstacles).
excludeTypes	<i>Array<string></i>	An array of element types that should not be considered obstacles when calculating the route. Default is <code>['basic.Text']</code> .
startDirections	<i>Array<string></i>	An array that specifies the possible starting directions from an element. Change this in situations where you need the link to, for example, always start at the bottom of the source element (then, the value would be <code>['bottom']</code>). Default is <code>['left', 'right', 'top', 'bottom']</code> (all directions allowed).
endDirections	<i>Array<string></i>	An array that specifies the possible ending directions to an element. Change this in situations where you need the link to, for example, always end at the bottom of the source element (then, the value would be <code>['bottom']</code>). Default is <code>['left', 'right', 'top', 'bottom']</code> (all directions allowed).

isPointObstacle	<i>(point) => boolean</i>	A function that determines whether a given point is an obstacle or not. If used, the <i>padding</i> , <i>excludeEnds</i> and <i>excludeTypes</i> options are ignored.

Example:

```
link.router('manhattan', {
  excludeEnds: ['source'],
  excludeTypes: ['myNamespace.MyCommentElement'],
  startDirections: ['top'],
  endDirections: ['bottom']
});
```

rovers.metro

The **'metro'** router is a modification of the **'manhattan'** router that produces an octolinear route (i.e. a route consisting of orthogonal and diagonal line segments, akin to the London Underground map design). It also avoids obstacles, and accepts the same `router.args` as **'manhattan'**, with a few modifications:

maximumLoops	<i>number</i>	Does not use the 'orthogonal' router as fallback if path cannot to be found in the given number of iterations. Instead, a custom octolinear fallback route is used that does not avoid obstacles.
maxAllowedDirectionChange	<i>number</i>	Default changes to 45 .
startDirection	<i>Array<string></i>	Same as 'manhattan' (i.e. only the four orthogonal directions are accepted as start directions).
endDirection	<i>Array<string></i>	Same as 'manhattan' (i.e. only the four orthogonal directions are accepted as end directions).

Example:

```
link.router('metro', {
  excludeEnds: ['source'],
  excludeTypes: ['myNamespace.MyCommentElement'],
  startDirections: ['top'],
  endDirections: ['bottom']
});
```


routers.normal

The `'normal'` router is the default router for links and it is the simplest router. It does not have any options and it does not do anything; it returns the vertices passed to it without modification as the route to be taken.

Example:

```
link.router('normal');
```

routers.orthogonal

The `'orthogonal'` router returns a route with orthogonal line segments. It generates extra route points in order to create the right angles on the way. It does not avoid obstacles. The router has one optional argument; this can be passed as the `router.args.elementPadding` property:

padding	<i>number object</i>	The minimum distance from element at which the first/last route angle may be placed. Default is <code>20</code> . The <code>util.normalizeSides</code> function is used to understand the provided value. A single number is applied as padding to all sides of the elements. An object may be provided to specify values for <code>left</code> , <code>top</code> , <code>right</code> , <code>bottom</code> , <code>horizontal</code> and/or <code>vertical</code> sides.

Example:

```
link.router('orthogonal', {  
  padding: 10  
});
```

(*Deprecated*) For the purposes of backwards compatibility, the `'orthogonal'` router can also be enabled by setting the `link.manhattan` property to `true`. Note that the meaning of “manhattan” in JointJS has changed over time.

```
// deprecated  
link.set('manhattan', true);
```

routers.rightAngle

The `'rightAngle'` router returns a route with orthogonal line segments just like `'orthogonal'` router, except it chooses the direction of the link based on the position of the source and target anchors (or port position). This router

avoids collisions with source and target elements, but does not avoid other obstacles. The behavior of the router can be modified by passing optional arguments:

This router completely ignores vertices, so using vertex tools such as `linkTools.Vertices` will not result in any change of path.

margin	<i>number</i>	The minimum distance between the route and the element. The default is <code>20</code> .
sourceDirection	<i>rightAngle.Directions</i>	The direction of the link from the source anchor. The default is <code>rightAngle.Directions.AUTO</code> .
targetDirection	<i>rightAngle.Directions</i>	The direction of the link from the target anchor. The default is <code>rightAngle.Directions.AUTO</code> .

Example:

```
link.router('rightAngle', {  
  margin: 10,  
  sourceDirection: routers.rightAngle.Directions.TOP,  
  targetDirection: routers.rightAngle.Directions.BOTTOM  
});
```

routers.rightAngle.Directions

Available values for the `sourceDirection` and `targetDirection` options:

rightAngle.Directions.AUTO	It automatically decides which direction to use. If the link is connected to a port, the <code>rightAngle.Directions.MAGNET_SIDE</code> is used. Otherwise, it uses <code>rightAngle.Directions.ANCHOR_SIDE</code> .
rightAngle.Directions.ANCHOR_SIDE	Determines the direction depending on the side of the magnet where the anchor is located.
rightAngle.Directions.MAGNET_SIDE	Determines the direction depending on the side of the element where the magnet (e.g. port) is located.
rightAngle.Directions.LEFT	Sets the direction of the link to the left side.
rightAngle.Directions.RIGHT	Sets the direction of the link to the right side.

rightAngle.Directions.TOP	Sets the direction of the link to the top side.
rightAngle.Directions.BOTTOM	Sets the direction of the link to the bottom side.

shapes.devs

The Devs plugin provides you with ready-to-use shapes with predefined input & output port groups and simplified API.

joint.shapes.devs.Model

The Model shape implements simple API on top of the JointJS built-in ports. It splits ports into two semantic groups (**in** and **out**) and adds convenient methods for adding and removing ports.

inPorts	an array of all input ports
outPorts	an array of all output ports

shapes.devs.Model.addInPort

`element.addInPort(port, [opt])`

Add a single port into the ``inPorts`` array, where ``port`` is a name of the port.

shapes.devs.Model.addOutPort

`element.addOutPort(port, [opt])`

Add a single port into the ``outPorts`` array, where ``port`` is a name of the port.

shapes.devs.Model.changeInGroup

`element.changeInGroup(properties, [opt])`

Change the settings for the input ports, where ``properties`` is an object with a [group configuration](#). It extends the previous settings with the new configuration by default. Pass `{ rewrite: true }` via `opt` to invalidate the previous settings.

shapes.devs.Model.changeOutGroup

`element.changeOutGroup(properties, [opt])`

Change the settings for the output ports, where `properties` is an object with a [group configuration](#). It extends the previous settings with the new configuration by default. Pass `{ rewrite: true }` via `opt` to invalidate the previous settings.

shapes.devs.Model.removeInPort

`element.removeInPort(port, [opt])`

Remove a port from an element from the `inPorts` array, where `port` is a name of the port.

shapes.devs.Model.removeOutPort

`element.removeOutPort(port, [opt])`

Remove a port from an element from the `outPorts` array, where `port` is a name of the port.

joint.shapes.devs.Link

The `devs.Link` extends the `joint.dia.Link` and changes the link appearance.

Example usage

```
var shape = new joint.shapes.devs.Model({
  position: {
    x: 100,
    y: 100
  },
  inPorts: ['in1', 'in2'],
  outPorts: ['out1', 'out2']
});

shape.addTo(graph);

// adding/removing ports dynamically
shape.addInPort('in3');
shape.removeOutPort('out1').addOutPort('out3');

var link = new joint.shapes.devs.Link({
  source: {
    id: shape.id,
    port: 'out3'
  },
  target: {
    id: 'targetId',
    port: 'targetPort'
  }
});
```

```

    target: {
      x: 200,
      y: 300
    }
  });
  link.addTo(graph);

  // moving the input ports from `left` to `top`
  shape.changeInGroup({ position: 'top' });
  // moving the output ports `right` to `bottom`
  shape.changeOutGroup({ position: 'bottom' });

```

Hierarchical diagrams

There are two more shapes designed for hierarchical diagrams as part of the plugin - `devs.Atomic` and `devs.Coupled`. They inherit from the `devs.Model` and differ only in the color and size. The purpose of these shapes is to enable you to implement a custom logic in your application based on their type. For instance a `devs.Coupled` can embed `devs.Atomic` 's but not the other way round as shown in the demo below.

When working with hierarchical diagrams there is a few methods, that might come in handy.

`coupled.embed(atomic)` that can put the `atomic` shape into the `coupled`.

`coupled.fitToChildren()` that resizes the `coupled` shape such that all embedded elements are visually contained within it.

`link.reparent()` that finds the best parent for the `link` based on the source and target element.

shapes.standard

The Standard plugin provides you with ready-to-use, high-performance versions of the most common shapes. The shapes can be used as they are or can serve as an inspiration for creating custom shapes.

source code

shapes.standard.BorderedImage

An image with a border and label.

Supported attris properties

Selector	Node	Description
root	<i>SVGGElement</i>	Container of all nodes

image	<i>SVGImageElement</i>	Image body of the shape
border	<i>SVGRectElement</i>	Border around the image
background	<i>SVGRectElement</i>	Area behind the image
label	<i>SVGTextElement</i>	Text below the image

```
var borderedImage = new joint.shapes.standard.BorderedImage();
borderedImage.resize(150, 100);
borderedImage.position(225, 410);
borderedImage.attr('root/title', 'joint.shapes.standard.BorderedImage');
borderedImage.attr('label/text', 'Bordered\nImage');
borderedImage.attr('border/rx', 5);
borderedImage.attr('image/xlinkHref', 'image.png');
borderedImage.addTo(graph);
```

shapes.standard.Circle

A circle with a label.

Supported attrs properties

Selector	Node	Description
root	<i>SVGGElement</i>	Container of all nodes
body	<i>SVGCircleElement</i>	Circular body of the shape
label	<i>SVGTextElement</i>	Text inside the body

```
var circle = new joint.shapes.standard.Circle();
circle.resize(100, 100);
circle.position(250, 10);
circle.attr('root/title', 'joint.shapes.standard.Circle');
circle.attr('label/text', 'Circle');
circle.attr('body/fill', 'lightblue');
circle.addTo(graph);
```

shapes.standard.Cylinder

A cylinder with a label.

Supported `attrs` properties

Selector	Node	Description
root	<i>SVGGElement</i>	Container of all nodes
body	<i>SVGPathElement</i>	Lateral area of the cylinder <i>The shape has a custom attribute <code>lateralArea</code> set to <code>10</code> by default. The attribute sets the <i>SVGPathElement</i>'s <code>d</code> in the shape of the lateral area of a tilted cylinder. The attribute expects a <code>topRy</code> value; either a percentage or a number. The value determines the vertical radius of the exposed area of cylinder base; either relative to shape height, or directly. (Refer to the <code>cylinder.topRy</code> function for more detailed specification.)</i>
top	<i>SVGEllipseElement</i>	Top of the cylinder (cylinder base)
label	<i>SVGTextElement</i>	Text inside the body

shapes.standard.Cylinder.prototype.topRy

`cylinder.topRy()`

Return the vertical radius of the exposed area of the cylinder base (the value of the `body/lateralArea` attribute; `10` by default).

`cylinder.topRy(t, [opt])`

Set the cylinder vertical radius of the exposed area of the cylinder base.

If the provided value is a percentage, it is relative to the `refBBox.height` of the shape. In practice, only values between `'0%'` and `'50%'` make sense. If the provided value is a number, it determines the vertical radius directly. Only values between `0` and half of `refBBox.height` make sense.

The function automatically sets the value of several attributes: `body/lateralArea`; and `top/ry`, `top/cy`, `top/refRy` and `top/refCy`. If these arguments need to be modified further, make sure to assign them only after calling `cylinder.topRy`.

Example usage:

```

var cylinder = new standard.Cylinder();
cylinder.resize(100, 200);
cylinder.position(525, 75);
cylinder.attr('root/title', 'joint.shapes.standard.Cylinder');
cylinder.attr('body/fill', 'lightgray');
cylinder.attr('top/fill', 'gray');
cylinder.attr('label/text', 'Cylinder');
cylinder.topRy('10%');
cylinder.addTo(graph);

```

shapes.standard.DoubleLink

A double line link.

Supported attrs properties

Selector	Node	Description
root	<i>SVGGElement</i>	Container of all nodes
line	<i>SVGPathElement</i>	Inner connection
outline	<i>SVGPathElement</i>	Outer connection

```

var doubleLink = new joint.shapes.standard.DoubleLink();
doubleLink.prop('source', { x: 500, y: 600 });
doubleLink.prop('target', { x: 450, y: 750 });
doubleLink.prop('vertices', [{ x: 500, y: 700 }]);
doubleLink.attr('root/title', 'joint.shapes.standard.DoubleLink');
doubleLink.attr('line/stroke', '#30d0c6');
doubleLink.addTo(graph);

```

shapes.standard.Ellipse

An ellipse with a label.

Supported attrs properties

Selector	Node	Description

root	<i>SVGGElement</i>	Container of all nodes
body	<i>SVGEllipseElement</i>	Elliptical body of the shape
label	<i>SVGTextElement</i>	Text inside the body

```
var ellipse = new joint.shapes.standard.Ellipse();
ellipse.resize(150, 100);
ellipse.position(425, 10);
ellipse.attr('root/title', 'joint.shapes.standard.Ellipse');
ellipse.attr('label/text', 'Ellipse');
ellipse.attr('body/fill', 'lightblue');
ellipse.addTo(graph);
```

shapes.standard.EmbeddedImage

An image embedded into a rectangle with a label.

Supported attrs properties

Selector	Node	Description
root	<i>SVGGElement</i>	Container of all nodes
body	<i>SVGRectElement</i>	Rectangular body of the shape
image	<i>SVGImageElement</i>	Image inside the body
label	<i>SVGTextElement</i>	Text next to the image

```
var embeddedImage = new joint.shapes.standard.EmbeddedImage();
embeddedImage.resize(150, 100);
embeddedImage.position(425, 410);
embeddedImage.attr('root/title', 'joint.shapes.standard.EmbeddedImage');
embeddedImage.attr('label/text', 'Embedded\nImage');
embeddedImage.attr('image/xlinkHref', 'image.png');
embeddedImage.addTo(graph);
```

shapes.standard.HeaderedRectangle

A rectangle with header.

Supported attrs properties

Selector	Node	Description
root	<i>SVGGElement</i>	Container of all nodes
body	<i>SVGRectElement</i>	Rectangular body of the shape
header	<i>SVGRectElement</i>	Rectangular header of the shape
headerText	<i>SVGTextElement</i>	Text inside the header
bodyText	<i>SVGTextElement</i>	Text inside the body

```
var headeredRectangle = new joint.shapes.standard.HeaderedRectangle();
headeredRectangle.resize(150, 100);
headeredRectangle.position(25, 610);
headeredRectangle.attr('root/title', 'joint.shapes.standard.HeaderedRectangle');
headeredRectangle.attr('header/fill', 'lightgray');
headeredRectangle.attr('headerText/text', 'Header');
headeredRectangle.attr('bodyText/text', 'Headered\nRectangle');
headeredRectangle.addTo(graph);
```

shapes.standard.Image

An image with a label.

Supported attrs properties

Selector	Node	Description
root	<i>SVGGElement</i>	Container of all nodes
image	<i>SVGImageElement</i>	Image body of the shape

label	<i>SVGTextElement</i>	Text below the image

```
var image = new joint.shapes.standard.Image();
image.resize(150, 100);
image.position(25, 410);
image.attr('root/title', 'joint.shapes.standard.Image');
image.attr('label/text', 'Image');
image.attr('image/xlinkHref', 'image.png');
image.addTo(graph);
```

shapes.standard.InscribedImage

An image inscribed in an ellipse with a label.

Supported attrs properties

Selector	Node	Description
root	<i>SVGGElement</i>	Container of all nodes
image	<i>SVGImageElement</i>	Image inscribed in the ellipse
border	<i>SVGEllipseElement</i>	Border around the ellipse
background	<i>SVGEllipseElement</i>	Area of the ellipse
label	<i>SVGTextElement</i>	Text below the shape

```
var inscribedImage = new joint.shapes.standard.InscribedImage();
inscribedImage.resize(150, 100);
inscribedImage.position(225, 410);
inscribedImage.attr('root/title', 'joint.shapes.standard.InscribedImage');
inscribedImage.attr('label/text', 'Inscribed Image');
inscribedImage.attr('border/strokeWidth', 5);
inscribedImage.attr('background/fill', 'lightgray');
inscribedImage.attr('image/xlinkHref', 'image.png');
inscribedImage.addTo(graph);
```

shapes.standard.Link

A single line link.

Supported attrs properties

Selector	Node	Description
root	<i>SVGGElement</i>	Container of all nodes
line	<i>SVGPathElement</i>	Visible connection of the link
wrapper	<i>SVGPathElement</i>	Invisible wrapper around the connection to make the line thicker, so the user can interact with the link more easily.

```
var link = new joint.shapes.standard.Link();
link.prop('source', { x: 450, y: 600 });
link.prop('target', { x: 400, y: 750 });
link.prop('vertices', [{ x: 450, y: 700 }]);
link.attr('root/title', 'joint.shapes.standard.Link');
link.attr('line/stroke', '#fe854f');
link.addTo(graph);
```

shapes.standard.Path

A path with a label.

Supported attrs properties

Selector	Node	Description
root	<i>SVGGElement</i>	Container of all nodes
body	<i>SVGPathElement</i>	Generic body of the shape
label	<i>SVGTextElement</i>	Text inside the body

```
var path = new joint.shapes.standard.Path();
path.resize(100, 100);
```

```

path.position(50, 210);
path.attr('root/title', 'joint.shapes.standard.Path');
path.attr('label/text', 'Path');
path.attr('body/refD', 'M 0 5 10 0 C 20 0 20 20 10 20 L 0 15 Z');
path.addTo(graph);

```

shapes.standard.Polygon

A polygon with a label.

Supported attrs properties

Selector	Node	Description
root	<i>SVGGElement</i>	Container of all nodes
body	<i>SVGPolygonElement</i>	Polygonal body of the shape
label	<i>SVGTextElement</i>	Text inside the body

```

var polygon = new joint.shapes.standard.Polygon();
polygon.resize(100, 100);
polygon.position(250, 210);
polygon.attr('root/title', 'joint.shapes.standard.Polygon');
polygon.attr('label/text', 'Polygon');
polygon.attr('body/refPoints', '0,10 10,0 20,10 10,20');
polygon.addTo(graph);

```

shapes.standard.Polyline

A polyline with a label.

Supported attrs properties

Selector	Node	Description
root	<i>SVGGElement</i>	Container of all nodes
body	<i>SVGPolylineElement</i>	Generic body of the shape

label	<i>SVGTextElement</i>	Text inside the body

```
var polyline = new joint.shapes.standard.Polyline();
polyline.resize(100, 100);
polyline.position(450, 210);
polyline.attr('root/title', 'joint.shapes.standard.Polyline');
polyline.attr('label/text', 'Polyline');
polyline.attr('body/refPoints', '0,0 0,10 10,10 10,0');
polyline.addTo(graph);
```

shapes.standard.Rectangle

A rectangle with a label.

Supported attrs properties

Selector	Node	Description
root	<i>SVGGElement</i>	Container of all nodes
body	<i>SVGRectElement</i>	Rectangular body of the shape
label	<i>SVGTextElement</i>	Text inside the body

```
var rectangle = new joint.shapes.standard.Rectangle();
rectangle.resize(100, 100);
rectangle.position(50, 10);
rectangle.attr('root/title', 'joint.shapes.standard.Rectangle');
rectangle.attr('label/text', 'Rectangle');
rectangle.attr('body/fill', 'lightblue');
rectangle.addTo(graph);
```

shapes.standard.ShadowLink

A thicker line with shadow.

Supported attrs properties

Selector	Node	Description

root	<i>SVGGElement</i>	Container of all nodes
line	<i>SVGPathElement</i>	Visible connection of the link
shadow	<i>SVGPathElement</i>	Shadow of the connection

```
var shadowLink = new joint.shapes.standard.ShadowLink();
shadowLink.prop('source', { x: 550, y: 600 });
shadowLink.prop('target', { x: 500, y: 750 });
shadowLink.prop('vertices', [{ x: 550, y: 700 }]);
shadowLink.attr('root/title', 'joint.shapes.standard.ShadowLink');
shadowLink.attr('line/stroke', '#5654a0');
shadowLink.addTo(graph);
```

shapes.standard.TextBlock

A rectangle with an HTML label (*with a fallback to SVG label for IE*).

Supported attrs properties

Selector	Node	Description
root	<i>SVGGElement</i>	Container of all nodes
body	<i>SVGRectElement</i>	Rectangular body of the shape
label	<i>HTMLDivElement</i>	Text inside the body

```
var textBlock = new joint.shapes.standard.TextBlock();
textBlock.resize(100, 100);
textBlock.position(250, 610);
textBlock.attr('root/title', 'joint.shapes.standard.TextBlock');
textBlock.attr('body/fill', 'lightgray');
textBlock.attr('label/text', 'Hyper Text Markup Language');
// Styling of the label via `style` presentation attribute (i.e. CSS).
textBlock.attr('label/style/color', 'red');
textBlock.addTo(graph);
```

util.assign

`util.assign(destinationObject, [...sourceObjects])`

Assigns own enumerable string keyed properties of source objects to the destination object. Source objects are applied from left to right. Subsequent sources overwrite property assignments of previous sources.

destinationObject	Object	The destination object
[sourceObjects]	Object[]	The source objects

util.bindAll

`util.bindAll(object, methodNames)`

Binds methods of an object to the object itself, overwriting the existing method.

object	Object	The object to bind and assign the bound methods to
methodNames	string string[]	The object method names to bind

util.breakText

`util.breakText(text, size [, attrs, opt])`

Break the provided `text` into lines so that it can fit into the bounding box defined by `size.width` and (optionally) `size.height`.

The function creates a temporary SVG `<text>` element and adds words to it one by one, measuring the element's actual rendered width and height at every step. If a word causes a line to overflow `size.width`, a newline character (`'\n'`) is generated. If a newline causes the text to overflow `size.height`, the string is cut off.

The returned string is a (possibly truncated) copy of `text` with newline characters at appropriate locations.

```
joint.util.breakText('lorem ipsum dolor sit amet consectetur adipiscing elit', {
  width: 100 })
// 'lorem ipsum\ndolor sit amet\nconsectetur\nadipiscing elit'
```

```
joint.util.breakText('lorem ipsum dolor sit amet consectetur adipiscing elit', {
  width: 100, height: 50 })
```



```
// 'lorem ipsum\ndolor sit amet'
```

The `attrs` parameter is an object with SVG attributes that should be set on the temporary SVG text element while it is being measured (such as `'font-weight'`, `'font-size'`, `'font-family'`, etc.). For example, an area of fixed size can obviously fit more words of font size `12px` than it can fit words of font size `36px`. If nothing can fit, an empty string is returned. (If an `attrs` object is not provided, the browser's default style is used.)

```
joint.util.breakText('lorem ipsum dolor sit amet consectetur adipiscing elit', {  
  width: 100, height: 50 }, { 'font-size': 12 })  
// 'lorem ipsum dolor\nsit amet consectetur\nadipiscing elit'
```

```
joint.util.breakText('lorem ipsum dolor sit amet consectetur adipiscing elit', {  
  width: 100, height: 50 }, { 'font-size': 16 })  
// 'lorem ipsum\ndolor sit amet'
```

```
joint.util.breakText('lorem ipsum dolor sit amet consectetur adipiscing elit', {  
  width: 100, height: 50 }, { 'font-size': 36 })  
// 'lorem'
```

```
joint.util.breakText('lorem ipsum dolor sit amet consectetur adipiscing elit', {  
  width: 100, height: 50 }, { 'font-size': 72 })  
// ''
```

The method also accepts the following additional options:

- `opt.separator` - a string or a regular expression which denotes how to split the text into words. Defaults to `' '`.
- `opt.eol` - a string representing the end-of-line symbol. Defaults to `'\n'`.
- `opt.ellipsis` - a boolean that specifies whether the ellipsis symbol (`'...'`, U+2026 HORIZONTAL ELLIPSIS) should be displayed when the text overflows. Defaults to `false`. If you provide a string, that string will be used as the ellipsis symbol instead.
- `opt.svgDocument` - an SVG document to which the temporary SVG text element should be added. By default, an SVG document is created automatically for you.
- `opt.hyphen` - a string or regex representing the hyphen symbol. If required, the method tries to break long words at hyphens first.
- `opt.maxLineCount` - a number to limit the maximum number of lines.
- `opt.preserveSpaces` - preserve the text spaces (avoid all consecutive spaces being deleted and replaced by one space, and preserve a space at the beginning and the end of the text).

Examples of text with the `ellipsis` option specified:

```
joint.util.breakText('lorem ipsum dolor sit amet consectetur adipiscing elit', {  
  width: 100, height: 50 }, { 'font-size': 12 }, { ellipsis: true })
```

```
// 'lorem ipsum dolor\nsit amet consectetur\nadipiscing elit'
```

```
joint.util.breakText('lorem ipsum dolor sit amet consectetur adipiscing elit', {  
width: 100, height: 50 }, { 'font-size': 16 }, { ellipsis: true })  
// 'lorem ipsum\ndolor sit ame...'
```

```
joint.util.breakText('lorem ipsum dolor sit amet consectetur adipiscing elit', {  
width: 100, height: 50 }, { 'font-size': 16 }, { ellipsis: '...!?' })  
// 'lorem ipsum\ndolor sit a...!?'
```

```
joint.util.breakText('lorem ipsum dolor sit amet consectetur adipiscing elit', {  
width: 100, height: 50 }, { 'font-size': 36 }, { ellipsis: true })  
// 'lore...'
```

```
joint.util.breakText('lorem ipsum dolor sit amet consectetur adipiscing elit', {  
width: 100, height: 50 }, { 'font-size': 36 }, { ellipsis: true })  
// ''
```

If you need to wrap text inside a `text` [attribute](#) of an Element, you should use the `textWrap` [attribute](#) instead. It does not require you to provide explicit size measurements, and it automatically responds to element resizing.

util.camelCase

```
util.camelCase([string=''])
```

Converts string to camel case.

<code>[string=""]</code>	string	The string to convert

util.cancelFrame

```
util.cancelFrame(requestId)
```

Cancels an animation frame request identified by `requestId` previously scheduled through a call to [joint.util.nextFrame](#).

util.clone

`util.clone(value)`

Creates a shallow clone of value.

value	any	The value to clone

util.cloneDeep

`util.cloneDeep(value)`

This method is like `util.clone` except that it recursively clones value.

value	any	The value to recursively clone

util.dataUriToBlob

`util.dataUriToBlob(dataUri)`

Convert a Data URI string into a Blob object.

util.debounce

`util.debounce(func, [wait=0], [options={}])`

Creates a debounced function that delays invoking `func` until after `wait` milliseconds have elapsed since the last time the debounced function was invoked. The debounced function comes with a `cancel` method to cancel delayed `func` invocations and a `flush` method to immediately invoke them. Provide options to indicate whether `func` should be invoked on the leading and/or trailing edge of the wait timeout. The `func` is invoked with the last arguments provided to the debounced function. Subsequent calls to the debounced function return the result of the last `func` invocation.

func	Function	The function to debounce
[wait=0]	number	The number of milliseconds to delay
[options={}]	Object	The options object
[options.leading=false]	boolean	Specify invoking on the leading edge of the timeout
[options.maxWait]	number	The maximum time <code>func</code> is allowed to be delayed before it's invoked

<code>[options.trailing=true]</code>	boolean	Specify invoking on the trailing edge of the timeout

util.deepMixin

`util.deepMixin(destinationObject, [...sourceObjects])`

(Deprecated) An alias for `util.assign`, use it instead.

util.deepSupplement

`util.deepSupplement(destinationObject [...sourceObjects])`

(Deprecated) An alias for `util.defaultsDeep`, use it instead.

util.defaults

`util.defaults(destinationObject [...sourceObjects])`

Assigns own and inherited enumerable string keyed properties of source objects to the destination object for all destination properties that resolve to undefined. Source objects are applied from left to right. Once a property is set, additional values of the same property are ignored.

destinationObject	Object	The destination object
[sourceObjects]	Object[]	The source objects

util.defaultsDeep

`util.defaultsDeep(destinationObject [...sourceObjects])`

This method is like `util.defaults` except that it recursively assigns default properties.

destinationObject	Object	The destination object
[sourceObjects]	Object[]	The source objects

util.difference

util.difference(array, [...values])

Creates an array of array values not included in the other given arrays using SameValueZero for equality comparisons. The order and references of result values are determined by the first array.

array	any[]	The array to inspect
[values]	any[]	The values to exclude

util.downloadBlob

util.downloadBlob(blob, fileName)

Download provided Blob object as a file with given fileName .

util.downloadDataUri

util.downloadDataUri(dataUri, fileName)

Download provided Data URI string as a file with given fileName .

util.flattenDeep

util.flattenDeep(array)

Recursively flattens array.

array	any[]	The array to flatten

util.flattenObject

util.flattenObject(object, delim, stop)

Flatten a nested object up until the stop function returns true . The stop function takes the value of the node currently traversed. delim is a delimiter for the combined keys in the resulting object. Example:

```

joint.util.flattenObject({
  a: {
    a1: 1,
    a2: 2,
    a3: {
      a31: 5,
      a32: {
        a321: { a3211: 5 }
      }
    }
  },
  b: 6
}, '/', function(v) { return !!v.a321; });

/*
{
  "a/a1": 1,
  "a/a2": 2,
  "a/a3/a31": 5,
  "a/a3/a32": {
    "a321": {
      "a3211": 5
    }
  },
  "b": 6
}
*/

```

util.forIn

util.forIn(object, [iteratee])

Iterates over elements of collection and invokes iteratee for each element. The iteratee is invoked with three arguments: (value, index|key, collection). Iteratee functions may exit iteration early by explicitly returning false.

collection	any[] Object	The collection to iterate over
[iteratee]	Function	The function invoked per iteration

util.getByPath

util.getByPath(object, path, delim)

Return a value at the `path` in a nested `object`. `delim` is the delimiter used in the `path` Example:

```
joint.util.getByPath({ a: { aa: { aaa: 3 } } }, 'a/aa/aaa', '/');  
// 3
```

util.getElementBBox

`util.getElementBBox(el)`

Return a bounding box of the element `el`. The advantage of this method is that it can handle both HTML and SVG elements. The resulting object is of the form `{ x: Number, y: Number, width: Number, height: Number }`.

util.getRectPoint

`util.getRectPoint(rect, positionName)`

Returns a `g.Point` on the rectangle specified by the keyword `positionName`.

The `rect` object has the form `{ x: Number, y: Number, width: Number, height: Number }`.

The `positionName` keyword can be one of the following:

- `'top-left'`
- `'top'`
- `'top-right'`
- `'bottom-left'`
- `'bottom'`
- `'bottom-right'`
- `'left'`
- `'right'`
- `'center'`

util.groupBy

`util.groupBy(collection, [iteratee])`

Creates an object composed of keys generated from the results of running each element of `collection` thru `iteratee`. The order of grouped values is determined by the order they occur in `collection`. The corresponding value of each key is an array of elements responsible for generating the key. The `iteratee` is invoked with one argument: *(value)*.

collection	<code>any[] Object</code>	The collection to iterate over

<code>[iteratee]</code>	Function	The function invoked per iteration

util.guid

`util.guid()`

Return an identifier unique for the page.

util.has

`util.has(object, propertyPath)`

Checks if path is a direct property of object.

object	Object	The object to query
propertyPath	string[] string	The path to check

util.hashCode

`util.hashCode(str)`

Return a simple hash code from a string.

util.imageToDataUri

`util.imageToDataUri(url, callback)`

Convert an image at `url` to the [Data URI scheme](#). The function is able to handle PNG, JPG and SVG formats. Useful if you need to embed images directly into your diagram instead of having them referenced externally.

The `callback` function has the following signature: `function(err, dataUri) {}`.

util.intersection

`util.intersection([...arrays])`

Creates an array of unique values that are included in all given arrays using [SameValueZero](#) for equality comparisons. The order and references of result values are determined by the first array.

[arrays]	any[][]	The arrays to inspect

util.invoke

util.[invoke](#)(collection, path, [args])

Invokes the method at path of each element in collection, returning an array of the results of each invoked method. Any additional arguments are provided to each invoked method. If path is a function, it's invoked for, and this bound to, each element in collection.

collection	any[] Object	The collection to iterate over
path	string[] string Function	The path of the method to invoke or the function invoked per iteration
[args]	any[]	The arguments to invoke each method with

util.invokeProperty

util.[invokeProperty](#)(object, path, [args])

Invokes the method at path of object.

object	Object	The object to query
path	string[] string	The path of the method to invoke
[args]	any[]	The arguments to invoke the method with

util.isBoolean

util.[isBoolean](#)(value)

Return [true](#) when value is a boolean.

util.isEmpty

util.isEmpty(value)

Checks if value is an empty object, collection, map, or set.

Objects are considered empty if they have no own enumerable string keyed properties.

Array-like values such as arguments objects, arrays, strings, or jQuery-like collections are considered empty if they have a length of 0. Similarly, maps and sets are considered empty if they have a size of 0.

value	any	The value to check

util.isEqual

util.isEqual(value, otherValue)

Performs a deep comparison between two values to determine if they are equivalent.

value	any	The value to compare
otherValue	any	The other value to compare

util.isFunction

util.isFunction(value)

Checks if value is classified as a Function object.

value	any	The value to check

util.isNumber

util.isNumber(value)

Return true when value is a number.

util.isObject

util.isObject(value)

Return **true** when `value` is an object (plain object or a function).

util.isPercentage

util.isPercentage(value)

Return **true** when `value` is a string that holds a percentage value, e.g. `'10%'`.

util.isPlainObject

util.isPlainObject(value)

Checks if `value` is a plain object, that is, an object created by the `Object` constructor or one with a `[[Prototype]]` of null.

value	any	The value to check

util.isString

util.isString(value)

Return **true** when `value` is a string.

util.merge

util.merge(destinationObject, [...sourceObjects], [customizer])

This method is like `joint.util.assign` except that it recursively merges own and inherited enumerable string keyed properties of source objects into the destination object. Source properties that resolve to undefined are skipped if a destination value exists. Array and plain object properties are merged recursively. Other objects and value types are overridden by assignment. Source objects are applied from left to right. Subsequent sources overwrite property assignments of previous sources. In addition this method accepts a customizer which is invoked to produce the merged values of the destination and source properties. The customizer is invoked with six arguments: (*objValue*, *srcValue*, *key*, *object*, *source*, *stack*)

destinationObject	Object	The destination object
[sourceObjects]	Object[]	The source objects
customizer	Function	The function to customize assigned values

util.mixin

`util.mixin(destinationObject, [...sourceObjects])`

(Deprecated) An alias for `util.assign`, use it instead.

util.nextFrame

`util.nextFrame(callback [, context, ...args])`

Tell the browser to schedule the `callback` function to be called before the next repaint.

This is a cross-browser version of the `window.requestAnimationFrame` function. It returns an ID of the frame request.

You can optionally pass a `context` for your callback function. Any further arguments are passed as arguments to the callback function.

util.noop

`util.noop()`

An empty function with no return statement (`void` in Typescript).

util.normalizeEvent

`util.normalizeEvent(evt)`

Normalize the provided event.

For touch events, the normalized event receives a copy of all properties from `evt.originalEvent` that are not included in `evt`.

Additionally, if `evt.target.correspondingUseElement` is defined (as in IE), that element is assigned as `target` in the normalized event.

util.normalizeSides

`util.normalizeSides(box)`

Return a new object of the form `{ top: Number, right: Number, bottom: Number, left: Number }`.

If `box` is a number, the value of all four sides will be this number. If `box` is an object with some/all of the `top`, `right`, `bottom`, `left` properties defined, the values of these properties will be used.

Composite properties `horizontal` (for right and left side) and `vertical` (for top and bottom side) can be used, as well; they will be destructured appropriately. When two properties clash (e.g. `horizontal: 10` and `left: 5`), the more specific one prevails (here the returned object would contain `right: 10` and `left: 5`).

If any property is missing, its side will be set to `0` in the resulting object.

```
joint.util.normalizeSides() // { top: 0, right: 0, bottom: 0, left: 0 }
joint.util.normalizeSides(5) // { top: 5, right: 5, bottom: 5, left: 5 }
joint.util.normalizeSides({ horizontal: 5 }) // { top: 0, right: 5, bottom: 0, left: 5 }
joint.util.normalizeSides({ left: 5 }) // { top: 0, right: 0, bottom: 0, left: 5 }
joint.util.normalizeSides({ horizontal: 10, left: 5 }) // { top: 0, right: 10, bottom: 0, left: 5 }
joint.util.normalizeSides({ horizontal: 0, left: 5 }) // { top: 0, left: 5, right: 0, bottom: 0 }
```

JointJS and Rappid use this method internally whenever there is an option object that can be specified either by a number or a (possibly incomplete) object with sides (for example, the `padding` option).

util.omit

`util.omit(object, [...paths])`

Opposite of `util.pick`, this method creates an object composed of the own and inherited enumerable property paths of object that are not omitted.

object	Object	The source object
[paths]	string[] string[][]	The property paths to omit

util.parseCssNumeric

util.parseCssNumeric(value)

Parse the provided value as a float and return an object with the number as the `value` parameter. If the value cannot be converted into a number, return `null`.

If the number is followed by a unit, assign it as the `unit` parameter in the return object. If there is no unit, assign an empty string. Examples:

```
joint.util.parseCssNumeric('hello'); // => null

joint.util.parseCssNumeric(1.1); // => { value: 1.1, unit: '' }
joint.util.parseCssNumeric('1.1'); // => { value: 1.1, unit: '' }
joint.util.parseCssNumeric('1.1px'); // => { value: 1.1, unit: 'px' }
joint.util.parseCssNumeric('1.1em'); // => { value: 1.1, unit: 'em' }
```

util.parseCssNumeric(value, restrictUnits)

Parse the provided value and only accept it if its unit is part of the `restrictUnits` parameter; otherwise return `null`.

The `restrictUnits` parameter can be a string or an array of strings. If you provide an empty string, that means you are expecting `value` to have no unit. Providing an empty array always returns `null`. Examples:

```
joint.util.parseCssNumeric(1.1, ''); // => { value: 1.1, unit: '' }
joint.util.parseCssNumeric('1.1', ''); // => { value: 1.1, unit: '' }
joint.util.parseCssNumeric('1.1px', ''); // => null
joint.util.parseCssNumeric('1.1px', 'px'); // => { value: 1.1, unit: 'px' };
joint.util.parseCssNumeric('1.1px', 'em'); // => null

joint.util.parseCssNumeric(1.1, []); // => null
joint.util.parseCssNumeric('1.1', []); // => null
joint.util.parseCssNumeric('1.1px', []); // => null

joint.util.parseCssNumeric(1.1, ['']); // => { value: 1.1, unit: '' }
joint.util.parseCssNumeric('1.1', ['']); // => { value: 1.1, unit: '' }
joint.util.parseCssNumeric('1.1px', ['px']); // => { value: 1.1, unit: 'px' };
joint.util.parseCssNumeric('1.1px', ['em']); // => null

joint.util.parseCssNumeric(1.1, ['', 'px']); // => { value: 1.1, unit: '' }
joint.util.parseCssNumeric('1.1', ['', 'px']); // => { value: 1.1, unit: '' }
joint.util.parseCssNumeric('1.1px', ['', 'px']); // => { value: 1.1, unit: 'px' }
joint.util.parseCssNumeric('1.1em', ['', 'px']); // => null
```

util.pick

`util.pick(object, [...paths])`

Creates an object composed of the picked object properties.

object	Object	The source object
[paths]	string[] string[][]	The property paths to pick

util.result

`util.result(object, path, [defaultValue])`

Gets the value at path of object. If the resolved value is undefined, the defaultValue is returned in its place. If the resolved value is a function it's invoked with the this binding of its parent object and its result is returned.

object	Object	The source object
path	string string[]	The path of the property to get
[defaultValue]	any	The value returned for undefined resolved values

util.sanitizeHTML

`util.sanitizeHTML(html)`

Sanitize the provided HTML (string) to protect against XSS attacks. The algorithm has several steps:

- Wrap the provided HTML inside a `<div>` tag. This will remove tags that are invalid in that context (e.g. `<body>` and `<head>`).
- Parse the provided HTML in a new document context (using `jQuery.parseHTML()`). This prevents inline events from firing and also prevents image GET requests from being sent.
- Discard all `<script>` tags.
- Iterate through all DOM nodes and remove all `on...` attributes (e.g. `onload` , `onerror`).
- Iterate through all attributes of the nodes and remove all that use the `javascript:` pseudo-protocol as value.
- Return the sanitized HTML back as a string.

The six simple steps protect against the most common XSS attacks; however, we cannot guarantee bulletproof security here. If you need stronger security, you should always keep an eye on a [list XSS attacks](#) and replace the

`joint.util.sanitizeHTML()` function with your own, more secure version.

Examples:

```
joint.util.sanitizeHTML('<html><body><p>Hello</p></body></html>'); // =>
'<p>Hello</p>'
joint.util.sanitizeHTML('<p>Hello</p><script>alert("Hacked");</script>'); // =>
'<p>Hello</p>'
joint.util.sanitizeHTML('<p>Hello</p><img onload="alert(&quot;Hacked&quot;);">'); //
=> '<p>Hello</p><img>'
joint.util.sanitizeHTML('<p>Hello</p>'); // => '<p>Hello</p><img>'
```

util.setAttributesBySelector

`util.setAttributesBySelector(el, attrs)`

Set attributes on the DOM element (SVG or HTML) `el` and its descendants based on the selector in the `attrs` object. The `attrs` object is of the form: `{ [selector]: [attributes] }`. For example:

```
var myEl = document.querySelector('.mydiv');
joint.util.setAttributesBySelector(myEl, {
  '.myDiv': { 'data-foo': 'bar' },    // Note the reference to the myEl element
  itself.
  'input': { 'value': 'my value' }
});
```

The code above sets `'data-foo'` attribute of the `myEl` element to the value `'bar'` and `'value'` attribute of all the descendant inputs to the value `'my value'`. This is a convenient way of setting attributes on elements (including themselves) and their descendants. Note that `'class'` attribute is treated as a special case and its value does not override the previous value of the `'class'` attribute. Instead, it adds this new class to the list of classes for the element referenced in the selector.

util.setByPath

`util.setByPath(object, path, value, delim)`

Set a `value` at the `path` in a nested `object`. `delim` is the delimiter used in the `path`. Returns the augmented `object`.

```
joint.util.setByPath({ a: 1 }, 'b/bb/bbb', 2, '/');
/*
{
```



```

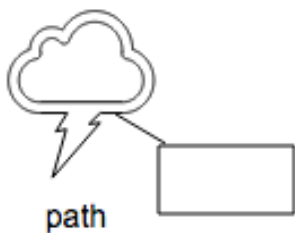
    "a": 1,
    "b": {
      "bb": {
        "bbb": 2
      }
    }
  }
  */

```

util.shapePerimeterConnectionPoint

util.[shapePerimeterConnectionPoint](#)(linkView, view, magnet, ref)

(*Deprecated*) This function can be directly used in the [dia.Paper](#) linkConnectionPoint parameter. When used, links will try to find the best connection point right on the perimeter of the connected shape rather than only on the bounding box. See the image below.



```

var paper = new joint.dia.Paper({
  // ...
  linkConnectionPoint: joint.util.shapePerimeterConnectionPoint
  // ...
})

```

util.sortBy

util.[sortBy](#)(collection, [iteratees])

Creates an array of elements, sorted in ascending order by the results of running each element in a collection thru each iteratee. This method performs a stable sort, that is, it preserves the original sort order of equal elements. The iteratees are invoked with one argument: (*value*).

collection	any[] Object	The collection to iterate over
[iteratees]	Funtion Funtion[]	The iteratees to sort by

util.sortedIndex

util.sortedIndex(array, value, [iteratee])

Uses a binary search to determine the lowest index at which value should be inserted into array in order to maintain its sort order. In addition this method accepts iteratee which is invoked for value and each element of array to compute their sort ranking. The iteratee is invoked with one argument: (*value*)

array	any[]	The sorted array to inspect
value	any	The value to evaluate
[iteratee]	Function	The iteratee invoked per element

util.sortElements

util.sortElements(elements, comparator)

Change the order of elements (a collection of HTML elements or a selector or jQuery object) in the DOM according to the comparator(elementA, elementB) function. The comparator function has the exact same meaning as in [Array.prototype.sort\(comparator\)](#).

util.supplement

util.supplement(destinationObject [...sourceObjects])

(Deprecated) An [alias](#) for util.defaults, use it instead.

util.svg

svg(strings)

This is the tagged template which converts a string into a [markup](#) object while declaring a [Cell](#). The resulting object contains fields which are described in [this part of the documentation](#). SVG is converted into JSON markup as follows:

Property	Representation
tagName	SVG tag name.

selector	@selector attribute.
groupSelector	@group-selector attribute. Accepts comma-separated lists e.g. @group-selector="group1, group2".
namespaceURI	The namespace URI of the element. It defaults to the SVG namespace "http://www.w3.org/2000/svg" .
attributes	Attributes of the element.
style	The style attribute of the element parsed as key-value pairs.
className	The class attribute of the element.
children	The children of the element.

Text nodes are placed into the children array as strings. For example, the markup `<span>a<span>b</span></span>` is converted to `{ tagName: 'span', children: ['a', { tagName: 'span', children: ['b']}]}`.

util.template

`util.template(html)`

Return a template function that returns pre-compiled `html` when called.

util.toArray

`util.toArray(value)`

Converts value to an array.

value	any	The value to convert

util.toggleFullScreen

`util.toggleFullScreen([element])`

Make (or cancel) an `element` to be displayed full-screen. The default element is `window.top.document.body`.

```
joint.util.toggleFullScreen(paper.el);
```

util.toKebabCase

util.toKebabCase(str)

Convert the string to kebab-case. Example:

```
joint.util.toKebabCase('strokeWidth'); // => 'stroke-width'
```

util.union

util.union(...arrays)

Creates an array of unique values, in order, from all given arrays using [SameValueZero](#) for equality comparisons.

[arrays]	any[]	The arrays to inspect

util.uniq

util.uniq(array, [iteratee])

Creates a duplicate-free version of an array, using [SameValueZero](#) for equality comparisons, in which only the first occurrence of each element is kept. The order of result values is determined by the order they occur in the array. In addition this method accepts iteratee which is invoked for each element in array to generate the criterion by which uniqueness is computed. The order of result values is determined by the order they occur in the array. The iteratee is invoked with one argument: (*value*).

array	any[]	The array to inspect
[iteratee]	Function	The iteratee invoked per element

util.uniqueId

util.uniqueId([prefix])

Generates a unique ID. If prefix is given, the ID is appended to it.

[prefix]	string	The prefix of the ID

util.unsetByPath

util.[unsetByPath](#)(object, path, delim)

Unset (delete) a property at the `path` in a nested `object`. `delim` is the delimiter used in the `path`. Returns the augmented `object`.

```
joint.util.unsetByPath({ a: { aa: { aaa: 3 } } }, 'a/aa/aaa', '/');
// { a: { aa: {} } }
```

util.uuid

util.[uuid](#)()

Return a pseudo-UUID.

util.without

util.[without](#)(array, [...values])

Creates an array excluding all given values using [SameValueZero](#) for equality comparisons.

array	any[]	The array to inspect
[values]	any[]	The values to exclude

util.format.number

util.format.[number](#)(specifier, value)

Format number `value` according to the `specifier` defined via the [Python Format Specification Mini-language](#).

```
joint.util.format.number('.2f', 5)    // 5.00
joint.util.format.number('03d', 5)    // 005
```

```
joint.util.format.number('.1%', .205)    // 20.5%  
joint.util.format.number('*^9', 5)      // ***5***
```

Geometry API

g

Together with [Vectorizer](#), Geometry is another lightweight library built-in to JointJS. This library implements many useful **geometry operations**. The geometry library does not have any dependencies and can be used standalone. Please see the download page that contains both the [development and minified versions](#) of this library.

g.normalizeAngle

`g.normalizeAngle(angle)`

Convert the `angle` to the range `[0, 360]`.

g.random

`g.random(min, max)`

Return a random integer between `min` and `max`, inclusive.

`g.random(max)`

Return a random integer between `0` and `max`, inclusive.

`g.random()`

Return `0` or `1`, chosen randomly.

g.snapToGrid

`g.snapToGrid(val, gridSize)`

Snap the value `val` to a grid of size `gridSize`.

g.toDeg

`g.toDeg(rad)`

Convert radians `rad` to degrees.

g.toRad

`g.toRad(deg, over360)`

Convert degrees `deg` to radians. If `over360` is `true`, do not modulate on 360 degrees.

g.bezier.curveThroughPoints

`g.bezier.curveThroughPoints(points)`

Deprecated. Use the `g.Curve.throughPoints` [function](#) instead.

Return the SVG path that defines a cubic bezier curve passing through `points`.

This method automatically computes the cubic bezier control points necessary to create a smooth curve with `points` as intermediary endpoints.

g.bezier.getCurveControlPoints

`g.bezier.getCurveControlPoints(knots)`

Deprecated. Use the `g.Curve.throughPoints` [function](#) instead.

Get open-ended Bezier Spline Control Points.

The `knots` argument should be an array of (at least two) Bezier spline points. Returns an array where the first item is an array of the first control points and the second item is an array of second control points.

g.bezier.getCurveDivider

`g.bezier.getCurveDivider(p0, p1, p2, p3)`

Deprecated. Use the `curve.divide` [function](#) instead.

Returns a function that divides a Bezier curve into two at point defined by value t between 0 and 1. Uses the deCasteljau algorithm.

g.bezier.getFirstControlPoints

`g.bezier.getFirstControlPoints(rhs)`

Deprecated. Use the `g.Curve.throughPoints` function instead.

Solves a tridiagonal system for one of coordinates (x or y) of first Bezier control points.

The `rhs` argument is a right hand side vector. Returns a solution vector.

g.bezier.getInversionSolver

`g.bezier.getInversionSolver(p0, p1, p2, p3)`

Deprecated. Use the `curve.closestPointT` function instead.

Solves an inversion problem -- Given the (x, y) coordinates of a point which lies on a parametric curve $x = x(t)/w(t)$, $y = y(t)/w(t)$, find the parameter value t which corresponds to that point. Returns a function that accepts a point and returns t .

g.Curve.constructor

`g.Curve(p1 [, p2, p3, p4])`

Return a new curve object with start at point `p1`, control points at `p2` and `p3`, and end at `p4`. All points are passed through the Point constructor so they can also be passed in string form. Examples:

```
var c = new g.Curve(new g.Point(10, 10), new g.Point(10, 40), new g.Point(50, 40),
new g.Point(50, 10));
var c = new g.Curve('10 10', '10 40', '50 40', '50 10');
var c = new g.Curve('10@10', '10@40', '50@40', '50@10');
```

The constructor also accepts a single Curve object as an argument; it creates a new curve with points cloned from the provided curve.

g.Curve.throughPoints

`g.Curve.throughPoints(points)`

Return an array of cuic bezier curves that defines a curve passing through provided `points`.

This method automatically computes the cubic bezier control points necessary to create a smooth curve with `points` as intermediary endpoints. An error is thrown if an array with fewer than two points is provided.

The result of this function may be provided directly to the `g.Path()` [constructor](#) in order to create a Path object from these curves.

g.Curve.prototype.bbox

`curve.bbox()`

Return a rectangle that is the tight bounding box of the curve (i.e. not including the curve control points).

g.Curve.prototype.clone

`curve.clone()`

Return another curve which is a clone of the curve.

g.Curve.prototype.closestPoint

`curve.closestPoint(point [, opt])`

Return the point on the curve that lies closest to `point`.

The function uses the same algorithm as the `curve.closestPointT()` [function](#). The point at `t` closest to `point` is returned.

The default value for `opt.precision` is 3; this corresponds to maximum observed error of 0.1%.

The `opt.subdivisions` property may be specified, directly providing an array of pre-computed curve subdivisions with which to begin the algorithm. Use the `curve.getSubdivisions()` [function](#) to obtain an array of curve subdivisions. The `opt.precision` property is still necessary, however; it determines the precision of the point search algorithm.

g.Curve.prototype.closestPointLength

`curve.closestPointLength(point [, opt])`

Return the length of the curve up to the point that lies closest to `point`.

The function uses the same algorithm as the `curve.closestPointT()` [function](#). The length of the curve at `t` closest to `point` is returned.

The default value for `opt.precision` is 3; this corresponds to maximum observed error of 0.1%.

The `opt.subdivisions` property may be specified, directly providing an array of pre-computed curve subdivisions with which to begin the algorithm. Use the `curve.getSubdivisions()` [function](#) to obtain an array of curve subdivisions. The `opt.precision` property is still necessary, however; it determines the precision of the point search algorithm.

g.Curve.prototype.closestPointNormalizedLength

`curve.closestPointNormalizedLength(point [, opt])`

Return the normalized length (distance from the start of the curve / total curve length) of the curve up to the point that lies closest to `point`.

The function uses the same algorithm as the `curve.closestPointT()` [function](#). The normalized length of the curve at `t` closest to `point` is returned.

The default value for `opt.precision` is 3; this corresponds to maximum observed error of 0.1%.

The `opt.subdivisions` property may be specified, directly providing an array of pre-computed curve subdivisions with which to begin the algorithm. Use the `curve.getSubdivisions()` [function](#) to obtain an array of curve subdivisions. The `opt.precision` property is still necessary, however; it determines the precision of the point search algorithm.

g.Curve.prototype.closestPointT

`curve.closestPointT(point [, opt])`

Return the `t` value of the point on the curve that lies closest to `point`.

The curve is first subdivided, according to `opt.precision` (refer to `curve.length()` [documentation](#) for more information about precision and curve flattening). Then, one subdivision is identified whose endpoints are the closest to `point`. A binary search is then performed on that subdivision, until a curve is found whose difference between endpoints' distance to `point` lies within `opt.precision`. The `t` value of the closest endpoint is returned by the function.

The default value for `opt.precision` is 3; this corresponds to maximum observed error of 0.1%.

The `opt.subdivisions` property may be specified, directly providing an array of pre-computed curve subdivisions with which to begin the algorithm. Use the `curve.getSubdivisions()` [function](#) to obtain an array of curve subdivisions. The `opt.precision` property is still necessary, however; it determines the precision of the point search algorithm.

g.Curve.prototype.closestPointTangent

`curve.closestPointTangent(point [, opt])`

Return a line that is tangent to the curve at the point that lies closest to `point`.

The tangent line starts at the identified closest point. The direction from `start` to `end` is the same as the direction of the curve at the closest point.

If the control points of the curve all lie at the same coordinates, `null` is returned (it is impossible to determine the slope of a point). The `curve.isDifferentiable()` [function](#) may be used in advance to determine whether tangents can exist for a given curve.

The function uses the same algorithm as the `curve.closestPointT()` [function](#). The tangent at `t` closest to `point` is returned.

The default value for `opt.precision` is 3; this corresponds to maximum observed error of 0.1%.

The `opt.subdivisions` property may be specified, directly providing an array of pre-computed curve subdivisions with which to begin the algorithm. Use the `curve.getSubdivisions()` [function](#) to obtain an array of curve subdivisions. The `opt.precision` property is still necessary, however; it determines the precision of the point search algorithm.

g.Curve.prototype.containsPoint

`curve.containsPoint(p [, opt])`

Return `true` if the point `p` is inside the area surrounded by the curve (inclusive). Return `false` otherwise. Uses the [even-odd algorithm](#) to resolve self-intersections.

Uses the `curve.toPolyline()` [function](#) in the background. The precision of the calculation may be adjusted by passing the `opt.precision` and `opt.subdivisions` properties.

g.Curve.prototype.divideAt

`curve.divideAt(ratio [, opt])`

Divide the curve into two curves at the point that lies `ratio` (normalized length) away from the beginning of the curve.

Returns an array with two new curves without modifying the original curve. The function expects `ratio` to lie between 0 and 1; values outside the range are constrained to 0 and 1, respectively.

This function mirrors the functionality of the `line.divideAt()` [function](#). If it is necessary to divide the curve at a certain `t` value instead, use the `curve.divideAtT()` [function](#).

The function uses the same algorithm as the `curve.divideAtLength()` [function](#).

The default value for `opt.precision` is 3; this corresponds to maximum observed error of 0.1%.

The `opt.subdivisions` property may be specified, directly providing an array of pre-computed curve subdivisions from which to calculate curve length. Use the `curve.getSubdivisions()` [function](#) to obtain an array of curve

subdivisions. The `opt.precision` property is still necessary, however; it determines the precision of the point search algorithm.

g.Curve.prototype.divideAtLength

`curve.divideAtLength(length [, opt])`

Divide the curve into two curves at the point that lies `length` away from the beginning of the curve.

Returns an array with two new curves without modifying the original curve. If negative `length` is provided, the algorithm starts looking from the end of the curve. If `length` is higher than curve length, the curve is divided at the closest endpoint instead.

The curve is first subdivided, according to `opt.precision` (refer to `curve.length()` [documentation](#) for more information about precision and curve flattening). Then, one subdivision is identified which contains the point at `length`. A binary search is then performed on that subdivision, until a curve is found whose endpoint lies within `opt.precision` away from `length`. That endpoint is used by the function to divide the curve.

The default value for `opt.precision` is 3; this corresponds to maximum observed error of 0.1%.

As a rule of thumb, increasing precision by 1 doubles the number of operations needed to find the point to be returned (this is on top of the cost of curve subdivision); exact numbers vary for every individual curve, however.

The `opt.subdivisions` property may be specified, directly providing an array of pre-computed curve subdivisions from which to calculate curve length. Use the `curve.getSubdivisions()` [function](#) to obtain an array of curve subdivisions. The `opt.precision` property is still necessary, however; it determines the precision of the point search algorithm.

g.Curve.prototype.divideAtT

`curve.divideAtT(t)`

Divide the curve into two curves at the point specified by `t`.

Returns an array with two new curves without modifying the original curve. The function expects `t` to lie between 0 and 1; values outside the range are constrained to 0 and 1, respectively.

This function is not a counterpart to the `line.divideAt()` [function](#). The relationship between `t` and distance along a curve is complex and non-linear. If it is necessary to divide the curve at a certain `ratio` (normalized length) away from beginning of the curve, use the `curve.divideAt()` [function](#).

g.Curve.prototype.endpointDistance

`curve.endpointDistance()`

Return the distance between curve `start` and `end` points.

g.Curve.prototype.equals

`curve.equals(otherCurve)`

Return `true` if the curve exactly equals the other curve.

g.Curve.prototype.getSkeletonPoints

`curve.getSkeletonPoints(t)`

Return an object that contains five points necessary for dividing curves.

The function expects `t` to lie between 0 and 1; values outside the range are constrained to 0 and 1, respectively.

The returned object has the following properties:

<code>startControlPoint1</code>	First curve's <code>controlPoint1</code> .
<code>startControlPoint2</code>	First curve's <code>controlPoint2</code> .
<code>divider</code>	The point at <code>t</code> ; <code>end</code> point of the first curve and <code>start</code> point of the second curve.
<code>dividerControlPoint1</code>	Second curve's <code>controlPoint1</code> .
<code>dividerControlPoint2</code>	Second curve's <code>controlPoint2</code>

If only the point at `t` is needed, use the `curve.pointAtT()` function.

If the two divided curves are needed, rather than their control points, use the `curve.divide()` function.

g.Curve.prototype.getSubdivisions

`curve.getSubdivisions([opt])`

Return an array of curves obtained by recursive halving of the curve up to a given precision.

Halving is not defined in terms of length, but in terms of the `t` parameter. The curves are subdivided at `t = 0.5` .

This is an intermediary function. Curve functions that rely on length calculations must work with flattened curves, with points obtained by curve subdivision at an arbitrary precision level. Refer to `curve.length()` [documentation](#) for more information about precision and curve flattening.

This function makes it possible to avoid expensive re-subdivisions of the curve when several operations need to be performed at the same level of precision (for example, obtaining the length of the curve and then finding the point at 10% length). The returned array may be passed to all such functions as the `opt.subdivisions` property.

The default value for `opt.precision` is 3; this corresponds to maximum observed error of 0.1%.

g.Curve.prototype.isDifferentiable

`curve.isDifferentiable()`

Return `true` if a tangent line can be found for the curve.

Tangents cannot be found if all of the curve control points are coincident (the curve appears to be a point).

g.Curve.prototype.length

`curve.length([opt])`

Return the length of the curve.

The curve length is a flattened length. This means that there will always be a difference between the reported length and the real length of the curve. (The returned curve length is always lower than the actual curve length.) That being said, the observed error can be constrained to an arbitrary threshold - here determined by `opt.precision`.

The `opt.precision` property is logarithmic, which means that increasing precision by 1 decreases observed error in length by a factor of 10. (For example, precision 3 leads to an observed error of less than 0.1%, precision 4 guarantees an observed error of less than 0.01%, etc.)

The default value for `opt.precision` is 3; this corresponds to maximum observed error of 0.1%.

As a rule of thumb, increasing precision by 1 quadruples the number of operations needed to determine the length; exact numbers vary for every individual curve, however. Precision 3 is considered good enough when drawing curve approximations, and precision 4 is considered good enough for mapping curve `t` values to curve length. (Precision 4 should be the highest necessary for any practical use.)

The measure used, observed error (difference between subsequent observed lengths), is not a measure of actual error (difference between observed length and the actual length); it is a necessary substitution, however. The actual curve length cannot be precisely determined in the general case, and obtaining flattened length at maximum precision is not feasible for every single length calculation. Still, actual error is generally 2-3 times lower than observed error, so `opt.precision` can be seen as an upper bound on the length error with a high degree of confidence.

Instead of `opt.precision`, the `opt.subdivisions` property may be specified, directly providing an array of pre-computed curve subdivisions from which to calculate curve length. This is useful when several operations need to be performed with the same curve at the same level of precision (for example, obtaining the length of the curve and then

finding the point at 10% length). Use the `curve.getSubdivisions()` [function](#) to obtain an array of curve subdivisions.

g.Curve.prototype.lengthAtT

`curve.lengthAtT(t [, opt])`

Return the length of the curve up to the point specified by `t`.

The curve is divided at `t` and the length of the first subcurve is returned. For more information about curve length calculation refer to `curve.length()` [documentation](#).

The default value for `opt.precision` is 3; this corresponds to maximum observed error of 0.1%.

This method does make use of the `opt.subdivisions` property.

g.Curve.prototype.pointAt

`curve.pointAt(ratio [, opt])`

Return a point on the curve that lies `ratio` (normalized length) away from the beginning of the curve.

The function expects `ratio` to lie between 0 and 1; values outside the range are constrained to 0 and 1, respectively.

This function mirrors the functionality of the `line.pointAt()` [function](#). If the point at a given `t` value is needed instead, use the `curve.pointAtT()` [function](#).

The function uses the same algorithm as the `curve.pointAtLength()` [function](#).

The default value for `opt.precision` is 3; this corresponds to maximum observed error of 0.1%.

The `opt.subdivisions` property may be specified, directly providing an array of pre-computed curve subdivisions from which to calculate curve length. Use the `curve.getSubdivisions()` [function](#) to obtain an array of curve subdivisions. The `opt.precision` property is still necessary, however; it determines the precision of the point search algorithm.

g.Curve.prototype.pointAtLength

`curve.pointAtLength(length [, opt])`

Return a point on the curve that lies `length` away from the beginning of the curve.

If negative `length` is provided, the algorithm starts looking from the end of the curve. If `length` is higher than curve length, the closest curve endpoint is returned instead.

The curve is first subdivided, according to `opt.precision` (refer to `curve.length()` [documentation](#) for more information about precision and curve flattening). Then, one subdivision is identified which contains the point at `length`. A binary search is then performed on that subdivision, until a curve is found whose endpoint lies within `opt.precision` away from `length`. That endpoint is returned by the function.

The default value for `opt.precision` is 3; this corresponds to maximum observed error of 0.1%.

As a rule of thumb, increasing precision by 1 doubles the number of operations needed to find the point to be returned (this is on top of the cost of curve subdivision); exact numbers vary for every individual curve, however.

The `opt.subdivisions` property may be specified, directly providing an array of pre-computed curve subdivisions from which to calculate curve length. Use the `curve.getSubdivisions()` [function](#) to obtain an array of curve subdivisions. The `opt.precision` property is still necessary, however; it determines the precision of the point search algorithm.

g.Curve.prototype.pointAtT

`curve.pointAtT(t)`

Return a point on the curve at a point specified by `t`.

The function expects `t` to lie between 0 and 1; values outside the range are constrained to 0 and 1, respectively.

This function is not a counterpart to the `line.pointAt()` [function](#). The relationship between `t` and distance along a curve is complex and non-linear. If a point at a certain `ratio` (normalized length) away from beginning of the curve is needed, use the `curve.pointAt()` [function](#).

g.Curve.prototype.round

`curve.round([precision])`

Round all coordinates of the curve to the given precision.

Default precision is 0.

Modifies this curve in-place, and returns it.

g.Curve.prototype.scale

`curve.scale(sx, sy [, origin])`

Scale the curve by `sx` and `sy` about the given origin.

If origin is not specified, the curve is scaled around 0,0.

g.Curve.prototype.tangentAt

`curve.tangentAt(ratio [, opt])`

Return a line that is tangent to the curve at a point that lies `ratio` (normalized length) away from the beginning of the curve.

The function expects `ratio` to lie between 0 and 1; values outside the range are constrained to 0 and 1, respectively.

The tangent line starts at the specified point. The direction from `start` to `end` is the same as the direction of the curve at the specified point.

If the control points of the curve all lie at the same coordinates, `null` is returned (it is impossible to determine the slope of a point). The `curve.isDifferentiable()` [function](#) may be used in advance to determine whether tangents can exist for a given curve.

The function uses the same algorithm as the `curve.tangentAtLength()` [function](#).

The default value for `opt.precision` is 3; this corresponds to maximum observed error of 0.1%.

The `opt.subdivisions` property may be specified, directly providing an array of pre-computed curve subdivisions from which to calculate curve length. Use the `curve.getSubdivisions()` [function](#) to obtain an array of curve subdivisions. The `opt.precision` property is still necessary, however; it determines the precision of the point search algorithm.

g.Curve.prototype.tangentAtLength

`curve.tangentAtLength(length [, opt])`

Return a line that is tangent to the curve at a point that lies `length` away from the beginning of the curve.

If negative `length` is provided, the algorithm starts looking from the end of the curve. If `length` is higher than curve length, a line tangent to the closest curve endpoint is returned instead.

The tangent line starts at the specified point. The direction from `start` to `end` is the same as the direction of the curve at the specified point.

If the control points of the curve all lie at the same coordinates, `null` is returned (it is impossible to determine the slope of a point). The `curve.isDifferentiable()` [function](#) may be used in advance to determine whether tangents can exist for a given curve.

The curve is first subdivided, according to `opt.precision` (refer to `curve.length()` [documentation](#) for more information about precision and curve flattening). Then, one subdivision is identified which contains the point at `length`. A binary search is then performed on that subdivision, until a curve is found whose endpoint lies within `opt.precision` away from `length`. That endpoint is used as the start of the tangent line.

The default value for `opt.precision` is 3; this corresponds to maximum observed error of 0.1%.

The `opt.subdivisions` property may be specified, directly providing an array of pre-computed curve subdivisions from which to calculate curve length. Use the `curve.getSubdivisions()` [function](#) to obtain an array of curve subdivisions. The `opt.precision` property is still necessary, however; it determines the precision of the point search algorithm.

g.Curve.prototype.tangentAtT

`curve.tangentAtT(t)`

Return a line that is tangent to the curve at point specified by `t`.

The function expects `t` to lie between 0 and 1; values outside the range are constrained to 0 and 1, respectively.

The tangent line starts at the specified point. The direction from `start` to `end` is the same as the direction of the curve at the specified point.

If the control points of the curve all lie at the same coordinates, `null` is returned (it is impossible to determine the slope of a point). The `curve.isDifferentiable()` [function](#) may be used in advance to determine whether tangents can exist for a given curve.

g.Curve.prototype.tAt

`curve.tAt(ratio [, opt])`

Return the `t` value of the point that lies `ratio` (normalized length) away from the beginning of the curve.

The function expects `ratio` to lie between 0 and 1; values outside the range are constrained to 0 and 1, respectively.

The function uses the same algorithm as the `curve.tAtLength()` [function](#).

The default value for `opt.precision` is 3; this corresponds to maximum observed error of 0.1%.

The `opt.subdivisions` property may be specified, directly providing an array of pre-computed curve subdivisions from which to calculate curve length. Use the `curve.getSubdivisions()` [function](#) to obtain an array of curve subdivisions. The `opt.precision` property is still necessary, however; it determines the precision of the point search algorithm.

g.Curve.prototype.tAtLength

`curve.tAtLength(length [, opt])`

Return the `t` value of the point that lies `length` away from the beginning of the curve.

If negative `length` is provided, the algorithm starts looking from the end of the curve. If `length` requested is higher than curve length, the `t` of the closest curve endpoint (0 or 1) is returned instead.

The curve is first subdivided, according to `opt.precision` (refer to `curve.length()` [documentation](#) for more information about precision and curve flattening). Then, one subdivision is identified which contains the point at `length`. A binary search is then performed on that subdivision, until a curve is found whose endpoint lies within `opt.precision` away from `length`. The `t` value of that endpoint is returned by the function.

The default value for `opt.precision` is 3; this corresponds to maximum observed error of 0.1%.

The `opt.subdivisions` property may be specified, directly providing an array of pre-computed curve subdivisions from which to calculate curve length. Use the `curve.getSubdivisions()` [function](#) to obtain an array of curve subdivisions. The `opt.precision` property is still necessary, however; it determines the precision of the point search algorithm.

g.Curve.prototype.toPoints

`curve.toPoints([opt])`

Return an array of points that approximate the curve at given precision.

The points obtained are endpoints of curve subdivisions whose flattened length is up to the specified precision level away from actual curve length. Refer to `curve.length()` [documentation](#) for more information about precision and curve flattening.

The default value for `opt.precision` is 3; this corresponds to maximum observed error of 0.1% in flattened curve length.

Instead of `opt.precision`, the `opt.subdivisions` property may be specified, directly providing an array of pre-computed curve subdivisions from which to obtain the array of points. Use the `curve.getSubdivisions()` [function](#) to obtain an array of curve subdivisions.

g.Curve.prototype.toPolyline

`curve.toPolyline([opt])`

Return a polyline that approximates the curve at given precision.

The polyline points are found as endpoints of curve subdivisions whose flattened length is up to the specified precision level away from actual curve length. Refer to `curve.length()` [documentation](#) for more information about precision and curve flattening.

The default value for `opt.precision` is 3; this corresponds to maximum observed error of 0.1% in flattened curve length.

Instead of `opt.precision`, the `opt.subdivisions` property may be specified, directly providing an array of pre-computed curve subdivisions from which to obtain the polyline. Use the `curve.getSubdivisions()` [function](#) to obtain an array of curve subdivisions.

g.Curve.prototype.toString

curve.toString()

Return the curve represented as a string.

```
new g.Curve('10 10', '10 20', '20 20', '20 10').toString() = "10@10 10@20 20@20 20@10"
```

g.Curve.prototype.translate

curve.translate(tx [, ty])

Translate the curve by `tx` on the x-axis and by `ty` on the y-axis.

If only `tx` is specified and is a number, `ty` is considered to be zero. If only `tx` is specified and is an object, it is considered to be a point or an object in the form `{ x: [number], y: [number] }`

g.Ellipse.constructor

g.Ellipse(c, rx, ry)

Return a new ellipse object with center at point `c` and parameters `rx` and `ry`.

g.Ellipse.fromRect

g.Ellipse.fromRect(rect)

Return a new ellipse object from the given rect.

g.Ellipse.prototype.bbox

ellipse.bbox()

Return a rectangle that is the bounding box of the ellipse.

g.Ellipse.prototype.center

ellipse.**center**()

Return a point that is the center of the ellipse.

g.Ellipse.prototype.clone

ellipse.**clone**()

Return another ellipse which is a clone of the ellipse.

g.Ellipse.prototype.containsPoint

ellipse.**containsPoint**(p)

Return **true** if the point *p* is inside the ellipse (inclusive). Return **false** otherwise.

g.Ellipse.prototype.equals

ellipse.**equals**(otherEllipse)

Return **true** if the ellipse equals the other ellipse.

g.Ellipse.prototype.inflate

ellipse.**inflate**(dx [, dy])

Return an ellipse inflated in axis *x* by $2 * dx$ and in axis *y* by $2 * dy$. When method is called with a single parameter, the resulting ellipse is inflated by $2 * dx$ in both *x* and *y* axis.

g.Ellipse.prototype.intersectionWithLine

ellipse.**intersectionWithLine**(line)

Return an array of the intersection points of the ellipse and the line. Return **null** if no intersection exists.

g.Ellipse.prototype.intersectionWithLineFromCenterToPoint

ellipse.intersectionWithLineFromCenterToPoint(p, angle)

Return the point on the boundary of the ellipse that is the intersection of the ellipse with a line starting in the center of the ellipse ending in the point `p`. If `angle` is specified, the intersection will take into account the rotation of the ellipse by `angle` degrees around its center.

g.Ellipse.prototype.normalizedDistance

ellipse.normalizedDistance(p)

Return a normalized distance from the ellipse center to point `p`.

Returns `n < 1` for points inside the ellipse, `n = 1` for points lying on the ellipse boundary and `n > 1` for points outside the ellipse.

g.Ellipse.prototype.round

ellipse.round([precision])

Round all coordinates of the ellipse to the given precision.

Default precision is `0`.

Modifies this ellipse in-place, and returns it.

g.Ellipse.prototype.tangentTheta

ellipse.tangentTheta(point)

Returns the angle between the x axis and the tangent from a `point`. It is valid for points lying on the ellipse boundary only.

g.Ellipse.prototype.toString

ellipse.toString()

Returns the ellipse represented as a string.

```
new g.Ellipse({ x: 10, y: 10}, 20, 30).toString() = "10@10 20 30"
```

g.intersection.exists

`g.intersection.exists(shape1, shape2 [, shape1Options, shape2Options])`

Returns `true` if two shapes intersect each other. The shape must be one of the following types:

- [Line](#)
- [Ellipse](#)
- [Rect](#)
- [Polyline](#)
- [Polygon](#)
- [Path](#)

```
g.intersection.exists(new g.Rect(0, 0, 10, 10), new g.Line({ x: 5, y: 5 }, { x: 30, y: 30 })); // true
```

Using the shape options is valid for [Path](#) only. The path is internally converted into polylines with [toPolylines\(\)](#) method. The options control the conversion of the path.

```
g.intersection.exists(new g.Rect(), new g.Path(), null, { precision: 2 });
g.intersection.exists(new g.Path(), new g.Rect(), { precision: 2 });

g.intersection.exists(path1, path2, {
  precision: 2
  segmentSubdivisions: path1SegmentSubdivisions
}, {
  precision: 2
  segmentSubdivisions: path2SegmentSubdivisions
});
```

g.Line.constructor

`g.Line(p1, p2)`

Return a new line object with start at point `p1` and end at point `p2`. `p1` and `p2` are first passed through the [Point](#) constructor so they can also be passed in string form. Examples:

```
var l = new g.Line(new g.Point(10, 20), new g.Point(50, 60));
var l = new g.Line('10 20', '50 60');
```



```
var l = new g.Line('10@20', '50@60');
```

The constructor also accepts a single Line object as an argument; it creates a new line with points cloned from the provided line.

g.Line.prototype.angle

line.[angle\(\)](#)

Return the angle of incline of the line.

The function returns `NaN` if the two endpoints of the line both lie at the same coordinates (it is impossible to determine the angle of incline of a line that appears to be a point). The `line.isDifferentiable()` [function](#) may be used in advance to determine whether the angle of incline can be computed for a given line.

g.Line.prototype.bbox

line.[bbox\(\)](#)

Return a rectangle that is the bounding box of the line.

g.Line.prototype.bearing

line.[bearing\(\)](#)

Return the bearing (cardinal direction) of the line. The return value is one of the following strings: `'NE'`, `'E'`, `'SE'`, `'S'`, `'SW'`, `'W'`, `'NW'` and `'N'`.

The function returns `'N'` if the two endpoints of the line are coincident.

g.Line.prototype.clone

line.[clone\(\)](#)

Return another line which is a clone of the line.

g.Line.prototype.closestPoint

line.[closestPoint](#)(point)

Return the point on the line that lies closest to `point` .

g.Line.prototype.closestPointLength

`line.closestPointLength(point)`

Return the length of the line up to the point that lies closest to `point` .

g.Line.prototype.closestPointNormalizedLength

`line.closestPointNormalizedLength(point)`

Return the normalized length (distance from the start of the line / total line length) of the line up to the point that lies closest to `point` .

g.Line.prototype.closestPointTangent

`line.closestPointTangent(point)`

Return a line that is tangent to the line at the point that lies closest to `point` .

The tangent line starts at the identified closest point. The direction from `start` to `end` is the same as the direction of the line at the closest point.

If the two endpoints of the line both lie at the same coordinates, `null` is returned (it is impossible to determine the slope of a point). The `line.isDifferentiable()` [function](#) may be used in advance to determine whether tangents can exist for a given line.

g.Line.prototype.containsPoint

`line.containsPoint(p)`

Return `true` if the point `p` lies on the line. Return `false` otherwise.

g.Line.prototype.divideAt

`line.divideAt(t)`

Divide the line into two lines at the point that lies `t` (normalized length) away from the beginning of the line.

Returns an array with two new lines without modifying the original line. For example, if `t` equals `0.5`, the function returns an array of two lines that subdivide the original line at its midpoint.

The function expects `t` to lie between 0 and 1; values outside the range are constrained to 0 and 1, respectively.

g.Line.prototype.divideAtLength

`line.divideAtLength(length)`

Divide the line into two lines at the point that lies `length` away from the beginning of the line.

Returns an array with two new lines without modifying the original line. If negative `length` is provided, the algorithm starts looking from the end of the curve. If `length` is higher than line length, the line is divided at the closest line endpoint instead.

g.Line.prototype.equals

`line.equals(otherLine)`

Return `true` if the line equals the other line.

g.Line.prototype.intersect

`line.intersect(shape)`

Return an array of the intersection points of the line with another geometry shape.

A single intersection point is returned, when the shape is a line.

Return `null` if no intersections are found.

g.Line.prototype.intersection

`line.intersection(l)`

Deprecated. Use the `line.intersect()` [function](#) instead.

Return the intersection point of the line with another line `l`.

A rectangle can also be passed in as the parameter. In that case, an array of intersection points with the line is returned.

Return `null` if no intersections are found.

g.Line.prototype.intersectionWithLine

`line.intersectionWithLine(line)`

Return an array of the intersection points of the line and another line `line`. Return `null` if no intersection exists.

g.Line.prototype.isDifferentiable

`line.isDifferentiable()`

Return `true` if a tangent line can be found for the line.

Tangents cannot be found if both of the line endpoints are coincident (the line appears to be a point).

g.Line.prototype.length

`line.length()`

Return the length of the line.

g.Line.prototype.midpoint

`line.midpoint()`

Return the point that is in the middle of the line.

g.Line.prototype.parallel

`line.parallel(distance)`

Return a new line which is parallel with this line. The new line is placed `distance` from this line in clockwise direction.

```
var line = new g.Line('0 0', '10 0');
line.parallel(10).toString(); // "0@10 10@10"
line.parallel(-5).toString(); // "0@-5 10@-5"
```

g.Line.prototype.pointAt

`line.pointAt(t)`

Return a point on the line that lies `t` (normalized length) away from the beginning of the line.

For example, if `t` equals `0.5`, the function returns the midpoint of the line.

The function expects `t` to lie between 0 and 1; values outside the range are constrained to 0 and 1, respectively.

g.Line.prototype.pointAtLength

`line.pointAtLength(length)`

Return a point on the line that lies `length` away from the beginning of the line.

If negative `length` is provided, the algorithm starts looking from the end of the line. If `length` is higher than line length, the closest line endpoint is returned instead.

g.Line.prototype.pointOffset

`line.pointOffset(point)`

Return the perpendicular distance between the line and `point`. The distance is positive if the point lies to the right of the line, negative if the point lies to the left of the line, and 0 if the point lies on the line.

g.Line.prototype.rotate

`line.scale(origin, angle)`

Rotate the line by `angle` around `origin`.

g.Line.prototype.round

`line.round([precision])`

Round all coordinates of the line to the given precision.

Default precision is `0`.

Modifies this line in-place, and returns it.

g.Line.prototype.scale

```
line.scale(sx, sy [, origin])
```

Scale the line by `sx` and `sy` about the given origin.

If origin is not specified, the line is scaled around 0,0.

g.Line.prototype.serialize

```
line.serialize()
```

Returns the line represented as an SVG points string.

```
new g.Line('10 10', '20 20').serialize() = "10,10 20,20"
```

g.Line.prototype.setLength

```
line.setLength(length)
```

Scale the line so that it has the requested `length`. The start point of the line is preserved.

g.Line.prototype.squaredLength

```
line.squaredLength()
```

Return the squared length of the line.

Useful for distance comparisons in which real length is not necessary (saves one `Math.sqrt()` operation).

g.Line.prototype.tangentAt

```
line.tangentAt(t)
```

Return a line tangent to the line at point that lies `t` (normalized length) away from the beginning of the line.

The function expects `t` to lie between 0 and 1; values outside the range are constrained to 0 and 1, respectively.

The tangent line starts at the specified point. The direction from `start` to `end` is the same as the direction of the line.

If the two endpoints of the line both lie at the same coordinates, `null` is returned (it is impossible to determine the slope of a point). The `line.isDifferentiable()` [function](#) may be used in advance to determine whether tangents can exist for a given line.

g.Line.prototype.tangentAtLength

`line.tangentAtLength(length)`

Return a line tangent to the line at point that lies `length` away from the beginning of the line.

If negative `length` is provided, the algorithm starts looking from the end of the line. If `length` is higher than line length, a line tangent to the closest line endpoint is returned instead.

The tangent line starts at the specified point. The direction from `start` to `end` is the same as the direction of the line.

If the two endpoints of the line both lie at the same coordinates, `null` is returned (it is impossible to determine the slope of a point). The `line.isDifferentiable()` [function](#) may be used in advance to determine whether tangents can exist for a given line.

g.Line.prototype.toString

`line.toString()`

Returns the line represented as a string.

```
new g.Line('10 10', '20 20').toString() = "10@10 20@20"
```

g.Line.prototype.translate

`line.translate(tx [, ty])`

Translate the line by `tx` on the x-axis and by `ty` on the y-axis.

If only `tx` is specified and is a number, `ty` is considered to be zero. If only `tx` is specified and is an object, it is considered to be a point or an object in the form `{ x: [number], y: [number] }`

g.Line.prototype.vector

`line.vector()`

Return the vector (point) of the line with length equal to length of the line.

This function effectively translates the line so that its start lies at 0,0.

g.Path.constructor

`g.Path(segments)`

Return a new path object consisting of path segments provided.

The path returned is not guaranteed to be a valid path (i.e. its path data might not be immediately usable as a `d` attribute of an SVG DOM element). This happens when the `segments` array does not start with a Moveto segment. The `path.isValid()` [function](#) may be used to test whether the path is valid. Use the `path.validate()` [function](#) to fix invalid paths.

The constructor also accepts an array of Line and/or Curve objects as an argument. Then, path segments are generated using this array. An initial Moveto segment is added, and additional Moveto segments are inserted between any two disconnected objects (the end point of one is not the start point of another).

It is not necessary to pass single-element arrays to the constructor; a single Segment, Line or Curve object is accepted as well.

Alternatively, the constructor accepts a Polyline object as an argument; Lineto path segments are generated to connect the polyline's points.

Finally, the constructor accepts a path data string as an argument; then the `g.Path.parse()` [function](#) is used to generate path segments.

g.Path.createSegment

`g.Path.createSegment(type [, ...args])`

Return a new path segment with specified `type` and (optionally) any further arguments.

The function throws an error if `type` is not recognized; only a limited subset of SVG path commands is supported (absolute versions of Moveto, Lineto, Curveto and Closepath).

Every path segment type expects a different number of arguments. Moveto expects 1 point, Lineto expects 1 point, Curveto expects 3 points, and Closepath expects no arguments. Chaining of arguments is allowed, so multiples of these numbers are accepted (e.g. 3 points for Lineto and 9 points for Curveto). An error is thrown if an incorrect number of points is provided (e.g. 2 points for Curveto). Examples:

```
var segment = g.Path.createSegment('M', new g.Point(100, 0));
var segment = g.Path.createSegment('L', new g.Point(100, 100), new g.Point(200, 200),
new g.Point(300, 300));
var segment = g.Path.createSegment('C', new g.Point(10, 10), new g.Point(20, 10), new
```



```
g.Point(20, 0), new g.Point(20, -10), new g.Point(30, -10), new g.Point(30, 0));
var segment = g.Path.createSegment('Z');
```

Instead of points, segments may be created with pairs of coordinates. That is, instead of providing 1 point to construct a Lineto segment, 2 strings or numbers may be provided. An error is thrown if an incorrect number of coordinates is provided. Examples:

```
var segment = g.Path.createSegment('M', 100, 0);
var segment = g.Path.createSegment('L', 100, 100, 200, 200, 300, 300);
var segment = g.Path.createSegment('C', 10, 10, 20, 10, 20, 0, 20, -10, 30, -10, 30, 0);
var segment = g.Path.createSegment('Z');
```

g.Path.isDataSupported

`g.Path.isDataSupported(pathData)`

Return `true` if the path data string provided consists of supported SVG Path commands (absolute versions of Moveto, Lineto, Curveto and Closepath).

g.Path.parse

`g.Path.parse(pathData)`

Return a new path object with path segments created from provided path data string.

The path data string does not require normalization, but it is restricted to subset of SVG path commands (absolute versions of Moveto, Lineto, Curveto and Closepath).

The string does not need to start with a Moveto command. (Empty string is accepted and creates an empty path.) Chaining of coordinates (e.g. providing a Moveto command with 4 parameters) is allowed.

The function throws an error if an unrecognized path command is encountered. Additionally, an error is thrown when an incorrect number of arguments is found with a command (e.g. a Curveto command with 5 coordinates).

g.Path.prototype.appendSegment

`path.appendSegment(segment)`

Append `segment` to the path.

Also accepts an array of segments as an argument. The segments are appended in the order they appear in the provided array.

The function does not return anything.

g.Path.prototype.bbox

`path.bbox()`

Return a rectangle that is the tight bounding box of the path (i.e. without curve control points).

Invisible path segments do not affect the bounding box. If the path contains no visible segments, a bounding box with `0` width and `0` height is returned, with the coordinates of the `end` point of the last path segment. If the path has no segments at all, `null` is returned.

g.Path.prototype.clone

`path.clone()`

Return another path which is a clone of the path.

g.Path.prototype.closestPoint

`path.closestPoint(point [, opt])`

Return the point on the path that lies closest to `point`.

Invisible segments (e.g. Move to segments) have no length and are therefore skipped by the algorithm. If the path contains no visible segments, the `end` point of the last segment is returned. If the path has no segments at all, `null` is returned.

The function uses the same algorithm as the `path.closestPointLength()` [function](#). It finds a visible segment whose identified closest point lies at the lowest distance from `point`.

The `opt` argument is optional. Two properties may be specified, `opt.precision` and `opt.segmentSubdivisions`, which determine maximum error allowed in segment `closestPoint` calculations for curved segments (default precision is 3; this corresponds to maximum observed error of 0.1%). The `opt.segmentSubdivisions` property is an array of individual segments' [subdivision arrays](#). The `path.getSegmentSubdivisions()` [function](#) may be used to obtain the `segmentSubdivisions` array. The `opt.precision` property is still necessary, however; it determines the precision of the point search algorithm in curved segments.

g.Path.prototype.closestPointLength

`path.closestPointLength(point [, opt])`

Return the length of the path up to the point that lies closest to `point` .

Invisible segments (e.g. MoveTo segments) have no length and are therefore skipped by the algorithm; if the path contains no visible segments, `0` is returned. If the path has no segments at all, `0` is returned.

The function finds a visible segment whose identified closest point lies at the lowest distance from `point` . It then determines the length of the path up to the identified closest point.

The `opt` argument is optional. Two properties may be specified, `opt.precision` and `opt.segmentSubdivisions` , which determine maximum error allowed in segment `closestPoint` calculations for curved segments (default precision is 3; this corresponds to maximum observed error of 0.1%). The `opt.segmentSubdivisions` property is an array of individual segments' subdivision arrays. The `path.getSegmentSubdivisions()` function may be used to obtain the `segmentSubdivisions` array. The `opt.precision` property is still necessary, however; it determines the precision of the point search algorithm in curved segments.

g.Path.prototype.closestPointNormalizedLength

`path.closestPointNormalizedLength(point [, opt])`

Return the normalized length (distance from the start of the path / total path length) of the path up to the point that lies closest to `point` .

Invisible segments (e.g. MoveTo segments) have no length and are therefore skipped by the algorithm; if the path contains no visible segments, `0` is returned. If the path has no segments at all, `0` is returned.

The function uses the same algorithm as the `path.closestPointLength()` function. It finds a visible segment whose identified closest point lies at the lowest distance from `point` . It then determines the normalized length of the path up to the identified closest point.

The `opt` argument is optional. Two properties may be specified, `opt.precision` and `opt.segmentSubdivisions` , which determine maximum error allowed in segment `closestPoint` calculations for curved segments (default precision is 3; this corresponds to maximum observed error of 0.1%). The `opt.segmentSubdivisions` property is an array of individual segments' subdivision arrays. The `path.getSegmentSubdivisions()` function may be used to obtain the `segmentSubdivisions` array. The `opt.precision` property is still necessary, however; it determines the precision of the point search algorithm in curved segments.

g.Path.prototype.closestPointT

`path.closestPointT(point [, opt])`

Private helper method.

Returns a `t` object that simplifies calculations for other `path.closestPoint` functions.

g.Path.prototype.closestPointTangent

path.[closestPointTangent](#)(point [, opt])

Return a line that is tangent to the path at the point that lies closest to `point`.

If the identified closest point is a point of discontinuity (e.g. it is a point shared by two Lineto segments with different slopes), the tangent line is constructed for the earlier segment (i.e. the segment closer to the beginning of the path).

The tangent line starts at the identified closest point. The direction from `start` to `end` is the same as the direction of the curve at the closest point.

The algorithm skips over segments that are not differentiable. This includes all invisible segments (e.g. Moveto segments) and visible segments with zero length. If the path contains no valid segments, `null` is returned. If the path has no segments at all, `null` is returned, as well. The `segment.isDifferentiable()` functions may be used to determine whether a given segment is valid; the `path.isDifferentiable()` [function](#) may be used to determine whether the path contains at least one valid segment.

The function finds a valid segment whose identified closest point lies at the lowest distance from `point`. It then finds a tangent to the segment at the identified closest point.

The `opt` argument is optional. Two properties may be specified, `opt.precision` and `opt.segmentSubdivisions`, which determine maximum error allowed in segment `closestPoint` calculations for curved segments (default precision is 3; this corresponds to maximum observed error of 0.1%). The `opt.segmentSubdivisions` property is an array of individual segments' [subdivision arrays](#). The `path.getSegmentSubdivisions()` [function](#) may be used to obtain the `segmentSubdivisions` array. The `opt.precision` property is still necessary, however; it determines the precision of the point search algorithm in curved segments.

g.Path.prototype.containsPoint

path.[containsPoint](#)(p [, opt])

Return `true` if the point `p` is inside the area surrounded by the path (inclusive). Return `false` otherwise. Uses the [even-odd algorithm](#) to resolve self-intersections.

Uses the `path.toPolylines()` [function](#) in the background. The precision of the calculation may be adjusted by passing the `opt.precision` and `opt.subdivisions` properties.

g.Path.prototype.divideAt

path.[divideAt](#)(ratio [, opt])

Divide the path into two paths at the point that lies `ratio` (normalized length) away from the beginning of the path.

Returns an array with two new paths without modifying the original path. The returned paths are valid; that is, they both start with an appropriate Moveto segment. Additionally, Closepath segments are converted into Lineto segments if necessary. The function expects `ratio` to lie between 0 and 1; values outside the range are constrained to 0 and 1, respectively. Invisible segments (e.g. Moveto segments) have no length and are therefore skipped by the algorithm. If the path contains no visible segments, `null` is returned.

The function uses the same algorithm as the `path.divideAtLength()` [function](#). It finds a visible segment which contains the point at length that corresponds to given `ratio` and then calls the segment's `divideAtLength()` function.

The `opt` argument is optional. Two properties may be specified, `opt.precision` and `opt.segmentSubdivisions`, which determine maximum error allowed in `divideAtLength` [calculations](#) for curved segments (default precision is 3; this corresponds to maximum observed error of 0.1%). The `opt.segmentSubdivisions` property is an array of individual segments' [subdivision arrays](#). The `path.getSegmentSubdivisions()` [function](#) may be used to obtain the `segmentSubdivisions` array. The `opt.precision` property is still necessary, however; it determines the precision of the point search algorithm in curved segments.

g.Path.prototype.divideAtLength

`path.pointAtLength(length [, opt])`

Divide the path into two paths at the point that lies `length` away from the beginning of the path.

Returns an array with two new paths without modifying the original path. The returned paths are valid; that is, they both start with an appropriate Moveto segment. Additionally, Closepath segments are converted into Lineto segments if necessary. If negative `length` is provided, the algorithm starts looking from the end of the path. If `length` is higher than path length, the path is divided at the closest visible path endpoint instead. Invisible segments (e.g. Moveto segments) have no length and are therefore skipped by the algorithm. If the path contains no visible segments, the end point of the last segment is returned. If the path has no segments at all, `null` is returned.

One visible segment is identified which contains the point at `length`. Finding the desired point is straightforward for linear segments (see `line.pointAtLength()` [for reference](#)). Finding the desired point in curved segments is more complex, as illustrated by the `curve.pointAtLength()` [function](#).

The `opt` argument is optional. Two properties may be specified, `opt.precision` and `opt.segmentSubdivisions`, which determine maximum error allowed in `pointAtLength` [calculations](#) for curved segments (default precision is 3; this corresponds to maximum observed error of 0.1%). The `opt.segmentSubdivisions` property is an array of individual segments' [subdivision arrays](#). The `path.getSegmentSubdivisions()` [function](#) may be used to obtain the `segmentSubdivisions` array. The `opt.precision` property is still necessary, however; it determines the precision of the point search algorithm in curved segments.

g.Path.prototype.equals

`path.equals(otherPath)`

Return `true` if the path exactly equals the other path.

The two paths are equal only if every segment of one path exactly equals the corresponding segment of the other path. The function returns `true` if `segments` arrays of both paths have no elements.

g.Path.prototype.getSegment

`path.getSegment(index)`

Return the path segment at `index`.

Negative indices are accepted; they instruct the algorithm to start looking from the end of the segments array.

Throws an error if the path has no segments. Also throws an error if the index is out of range.

g.Path.prototype.getSegmentSubdivisions

`path.getSegmentSubdivisions([opt])`

Return an array of segment subdivision arrays.

In Curveto segments, subdivisions are obtained by recursive halving up to given precision. Other types of segments do not have subdivisions; `[]` placeholders are used in their place.

This is an intermediary function. Path functions that rely on length calculations may need to work with flattened curves, with points obtained by curve subdivision at an arbitrary precision level. Refer to `curve.length()` documentation for more information about precision and curve flattening.

This function makes it possible to avoid expensive re-subdivisions of curved segments when several operations need to be performed at the same level of precision (for example, obtaining the length of the path and then finding the point at 10% length). The returned array may be passed to all such functions as the `opt.segmentSubdivisions` property.

The default value for `opt.precision` is 3; this corresponds to maximum observed error of 0.1% in the flattened length of curved segments.

g.Path.prototype.insertSegment

`path.insertSegment(index, segment)`

Insert `segment` to the path at `index` provided.

Negative indices are accepted; they instruct the algorithm to start looking from the end of the segments array.

Also accepts an array of segments as an argument. The segments are inserted at `index` as a group, in the order they appear in the provided array.

The function does not return anything.

g.Path.prototype.intersectionWithLine

`path.intersectionWithLine(line [, opt])`

Return an array of the intersection points of the path and the line. Return `null` if no intersection exists.

The `opt` argument is optional. Two properties may be specified, `opt.precision` and `opt.segmentSubdivisions`, which determine maximum error allowed in `pointAtLength` calculations for curved segments (default precision is 3; this corresponds to maximum observed error of 0.1%). The `opt.segmentSubdivisions` property is an array of individual segments' subdivision arrays. The `path.getSegmentSubdivisions()` function may be used to obtain the `segmentSubdivisions` array. The `opt.precision` property is still necessary, however; it determines the precision of the point search algorithm in curved segments.

g.Path.prototype.isDifferentiable

`path.isDifferentiable()`

Return `true` if a tangent line can be found for at least one segment of the path.

Invisible segments (e.g. Moveto segments) are never differentiable. In Line-based segments (e.g. Lineto segments), tangents cannot be found both line endpoints are coincident. In Curve-based segments (e.g. Curveto segments), tangents cannot be found if all control points are coincident. If the path contains no segments, return `false`.

g.Path.prototype.isValid

`path.isValid()`

Return `true` if the path has valid path data (and thus may be used as a `d` attribute of an SVG DOM element).

Return `true` if the path has no segments. Return `false` if the path has segments but does not start with a Moveto.

You may use the `path.validate()` function to add an initial `'M 0 0'` segment to invalid paths; the method leaves valid paths unchanged.

g.Path.prototype.length

path.[length](#)([opt])

Return the length of the path.

The path length is a sum of the lengths of visible path segments; invisible segments (e.g. Moveto segments) have a length of 0. If the path has no segments at all, `0` is returned.

Although calculating the length of linear segments (e.g. Lineto and Closepath segments) is straightforward, determining the length of curved segments is complex. Refer to `curve.length()` [documentation](#) for more information.

The `opt` argument is optional. Two properties may be specified, `opt.precision` and `opt.segmentSubdivisions`, which determine maximum error allowed for the length calculation (default precision is 3; this corresponds to maximum observed error of 0.1%). The `opt.segmentSubdivisions` property is an array of individual segments' [subdivision arrays](#). The `path.getSegmentSubdivisions()` [function](#) may be used to obtain the `segmentSubdivisions` array.

g.Path.prototype.lengthAtT

path.[lengthAtT](#)(t [, opt])

Private helper method.

Makes use of `t` objects obtained from the `path.closestPointT()` [function](#). Used as an intermediary by other `path.closestPoint` functions.

g.Path.prototype.pointAt

path.[pointAt](#)(ratio [, opt])

Return a point on the path that lies `ratio` (normalized length) away from the beginning of the path.

The function expects `ratio` to lie between 0 and 1; values outside the range are constrained to 0 and 1, respectively. Invisible segments (e.g. Moveto segments) have no length and are therefore skipped by the algorithm. If the path contains no visible segments, the `end` point of the last segment is returned. If the path has no segments at all, `null` is returned.

The function uses the same algorithm as the `path.pointAtLength()` [function](#). It finds a visible segment which contains the point at length that corresponds to given `ratio` and then calls the segment's `pointAtLength()` function.

The `opt` argument is optional. Two properties may be specified, `opt.precision` and `opt.segmentSubdivisions`, which determine maximum error allowed in `pointAtLength` [calculations](#) for curved

segments (default precision is 3; this corresponds to maximum observed error of 0.1%). The `opt.segmentSubdivisions` property is an array of individual segments' [subdivision arrays](#). The `path.getSegmentSubdivisions()` [function](#) may be used to obtain the `segmentSubdivisions` array. The `opt.precision` property is still necessary, however; it determines the precision of the point search algorithm in curved segments.

g.Path.prototype.pointAtLength

`path.pointAtLength(length [, opt])`

Return a point on the path that lies `length` away from the beginning of the path.

If negative `length` is provided, the algorithm starts looking from the end of the path. If `length` is higher than path length, the closest visible path endpoint is returned instead. Invisible segments (e.g. `Moveto` segments) have no length and are therefore skipped by the algorithm. If the path contains no visible segments, the `end` point of the last segment is returned. If the path has no segments at all, `null` is returned.

One visible segment is identified which contains the point at `length`. Finding the desired point is straightforward for linear segments (see `Line.pointAtLength()` [for reference](#)). Finding the desired point in curved segments is more complex, as illustrated by the `curve.pointAtLength()` [function](#).

The `opt` argument is optional. Two properties may be specified, `opt.precision` and `opt.segmentSubdivisions`, which determine maximum error allowed in `pointAtLength` [calculations](#) for curved segments (default precision is 3; this corresponds to maximum observed error of 0.1%). The `opt.segmentSubdivisions` property is an array of individual segments' [subdivision arrays](#). The `path.getSegmentSubdivisions()` [function](#) may be used to obtain the `segmentSubdivisions` array. The `opt.precision` property is still necessary, however; it determines the precision of the point search algorithm in curved segments.

g.Path.prototype.pointAtT

`path.pointAtT(t)`

Private helper method.

Makes use of `t` objects obtained from the `path.closestPointT()` [function](#). Used as an intermediary by other `path.closestPoint` functions.

g.Path.prototype.removeSegment

`path.removeSegment(index)`

Remove the path segment at `index`.

Negative indices are accepted; they instruct the algorithm to start looking from the end of the segments array.

The function does not return anything.

g.Path.prototype.replaceSegment

path.[replaceSegment](#)(index, segment)

Replace the path segment at `index` with the `segment` provided.

Negative indices are accepted; they instruct the algorithm to start looking from the end of the segments array.

Also accepts an array of segments as an argument. The segments replace the segment at `index` as a group, in the order they appear in the provided array.

The function does not return anything.

g.Path.prototype.round

path.[round](#)([precision])

Round all coordinates of the path to the given precision.

Default precision is `0`.

Modifies this path in-place, and returns it.

g.Path.prototype.scale

path.[scale](#)(sx, sy [, origin])

Scale the path by `sx` and `sy` about the given origin.

If origin is not specified, the path is scaled around 0,0.

Modifies this path in-place, and returns it.

g.Path.prototype.segmentAt

path.[segmentAt](#)(ratio [, opt])

Return the path segment that contains a point that lies `ratio` (normalized length) away from the beginning of the path.

The function expects `ratio` to lie between 0 and 1; values outside the range are constrained to 0 and 1, respectively. Invisible segments (e.g. Moveto segments) have no length and are therefore skipped by the algorithm. If the path contains no visible segments, `null` is returned. If the path has no segments at all, `null` is returned, as well.

The `opt` argument is optional. Two properties may be specified, `opt.precision` and `opt.segmentSubdivisions`, which determine maximum error allowed in `pointAtLength` [calculations](#) for curved segments (default precision is 3; this corresponds to maximum observed error of 0.1%). The `opt.segmentSubdivisions` property is an array of individual segments' [subdivision arrays](#). The `path.getSegmentSubdivisions()` [function](#) may be used to obtain the `segmentSubdivisions` array. The `opt.precision` property is still necessary, however; it determines the precision of the point search algorithm in curved segments.

g.Path.prototype.segmentAtLength

`path.segmentAtLength(length [, opt])`

Return the path segment that contains a point that lies `length` away from the beginning of the path.

If negative `length` is provided, the algorithm starts looking from the end of the path. If `length` is higher than path length, the closest visible path segment is returned. Invisible segments (e.g. Moveto segments) have no length and are therefore skipped by the algorithm. If the path contains no visible segments, `null` is returned. If the path has no segments at all, `null` is returned, as well.

The `opt` argument is optional. Two properties may be specified, `opt.precision` and `opt.segmentSubdivisions`, which determine maximum error allowed in `pointAtLength` [calculations](#) for curved segments (default precision is 3; this corresponds to maximum observed error of 0.1%). The `opt.segmentSubdivisions` property is an array of individual segments' [subdivision arrays](#). The `path.getSegmentSubdivisions()` [function](#) may be used to obtain the `segmentSubdivisions` array. The `opt.precision` property is still necessary, however; it determines the precision of the point search algorithm in curved segments.

g.Path.prototype.segmentIndexAt

`path.segmentIndexAt(ratio [, opt])`

Return the index of the path segment that contains a point that lies `ratio` (normalized length) away from the beginning of the path.

The function expects `ratio` to lie between 0 and 1; values outside the range are constrained to 0 and 1, respectively. Invisible segments (e.g. Moveto segments) have no length and are therefore skipped by the algorithm. If the path contains no visible segments, `null` is returned. If the path has no segments at all, `null` is returned, as well.

The `opt` argument is optional. Two properties may be specified, `opt.precision` and `opt.segmentSubdivisions`, which determine maximum error allowed in `pointAtLength` [calculations](#) for curved segments (default precision is 3; this corresponds to maximum observed error of 0.1%). The `opt.segmentSubdivisions` property is an array of individual segments' [subdivision arrays](#). The

`path.getSegmentSubdivisions()` [function](#) may be used to obtain the `segmentSubdivisions` array. The `opt.precision` property is still necessary, however; it determines the precision of the point search algorithm in curved segments.

g.Path.prototype.segmentIndexAtLength

`path.segmentIndexAtLength(length [, opt])`

Return the index of the path segment that contains a point that lies `length` away from the beginning of the path.

If negative `length` is provided, the algorithm starts looking from the end of the path. If `length` is higher than path length, the closest visible path segment is returned. Invisible segments (e.g. `Moveto` segments) have no length and are therefore skipped by the algorithm. If the path contains no visible segments, `null` is returned. If the path has no segments at all, `null` is returned, as well.

The `opt` argument is optional. Two properties may be specified, `opt.precision` and `opt.segmentSubdivisions`, which determine maximum error allowed in `pointAtLength` [calculations](#) for curved segments (default precision is 3; this corresponds to maximum observed error of 0.1%). The `opt.segmentSubdivisions` property is an array of individual segments' [subdivision arrays](#). The `path.getSegmentSubdivisions()` [function](#) may be used to obtain the `segmentSubdivisions` array. The `opt.precision` property is still necessary, however; it determines the precision of the point search algorithm in curved segments.

g.Path.prototype.serialize

`path.serialize()`

Return the path represented as a path data string.

Whenever a Path object needs to be converted to a string for the SVG `d` attribute, this function should be used. Unlike the `path.toString()` [function](#), this function checks whether the path data string is [valid](#). The function throws an error if the path is not valid. This happens if the path has segments but does not start with a `Moveto` segment. Note that an empty string is valid, according to the SVG specification.

```
new g.Path('M 150 100 L 100 100 C 100 100 0 150 100 200 Z').serialize() = "M 150 100
L 100 100 C 100 100 0 150 100 200 Z"
```

g.Path.prototype.tangentAt

`path.tangentAt(ratio [, opt])`

Return a line tangent to the path at point that lies `ratio` (normalized length) away from the beginning of the path.

The function expects `ratio` to lie between 0 and 1; values outside the range are constrained to 0 and 1, respectively. If point at `ratio` is a point of discontinuity (e.g. it is a point shared by two Lineto segments with different slopes), the tangent line is constructed for the earlier segment (i.e. the segment closer to the beginning of the path).

The tangent line starts at the specified point. The direction from `start` to `end` is the same as the direction of the path segment at the specified point.

The algorithm skips over segments that are not differentiable. This includes all invisible segments (e.g. Moveto segments) and visible segments with zero length. If the path contains no valid segments, `null` is returned. If the path has no segments at all, `null` is returned, as well. The `segment.isDifferentiable()` functions may be used to determine whether a given segment is valid; the `path.isDifferentiable()` [function](#) may be used to determine whether the path contains at least one valid segment.

The function uses the same algorithm as the `path.tangentAtLength()` [function](#). It finds a valid segment which contains the point at length that corresponds to given `ratio` and then calls the segment's `segment.tangentAtLength()` function.

The `opt` argument is optional. Two properties may be specified, `opt.precision` and `opt.segmentSubdivisions`, which determine maximum error allowed in `pointAtLength` [calculations](#) for curved segments (default precision is 3; this corresponds to maximum observed error of 0.1%). The `opt.segmentSubdivisions` property is an array of individual segments' [subdivision arrays](#). The `path.getSegmentSubdivisions()` [function](#) may be used to obtain the `segmentSubdivisions` array. The `opt.precision` property is still necessary, however; it determines the precision of the point search algorithm in curved segments.

g.Path.prototype.tangentAtLength

`path.tangentAtLength(length [, opt])`

Return a line tangent to the path at point that lies `length` away from the beginning of the path.

If negative `length` is provided, the algorithm starts looking from the end of the path. If `length` is higher than path length, a line tangent to the closest valid path endpoint is returned instead. If point at `length` is a point of discontinuity (e.g. it is a point shared by two Lineto segments with different slopes), the tangent line is constructed for the earlier segment (i.e. the segment closer to the beginning of the path).

The tangent line starts at the specified point. The direction from `start` to `end` is the same as the direction of the path segment at the specified point.

The algorithm skips over segments that are not differentiable. This includes all invisible segments (e.g. Moveto segments) and visible segments with zero length. If the path contains no valid segments, `null` is returned. If the path has no segments at all, `null` is returned, as well. The `segment.isDifferentiable()` functions may be used to determine whether a given segment is valid; the `path.isDifferentiable()` [function](#) may be used to determine whether the path contains at least one valid segment.

One valid segment is identified which contains the point at `length`. Finding the desired point is straightforward for linear segments (see `line.pointAtLength()` [for reference](#)). Finding the desired point in curved segments is more complex, as illustrated by the `curve.pointAtLength()` [function](#). A tangent line is then obtained that touches the path at the identified point (see `curve.tangentAtLength()` [for reference](#)).

The `opt` argument is optional. Two properties may be specified, `opt.precision` and `opt.segmentSubdivisions`, which determine maximum error allowed in `pointAtLength` [calculations](#) for curved segments (default precision is 3; this corresponds to maximum observed error of 0.1%). The `opt.segmentSubdivisions` property is an array of individual segments' [subdivision arrays](#). The `path.getSegmentSubdivisions()` [function](#) may be used to obtain the `segmentSubdivisions` array. The `opt.precision` property is still necessary, however; it determines the precision of the point search algorithm in curved segments.

g.Path.prototype.tangentAtT

`path.tangentAtT(t)`

Private helper method.

Makes use of `t` objects obtained from the `path.closestPointT()` [function](#). Used as an intermediary by other `path.closestPoint` functions.

g.Path.prototype.toPoints

`path.toPoints([opt])`

Return an array of arrays of points that approximate the path at given precision. If there are no segments, `null` is returned.

Every Moveto segment starts a new subpath within the path. Every subpath has a separate array of points in the returned array. If there is only one subpath, an array with one array of points is returned.

The points are found as endpoints of Lineto and Closepath segments and as endpoints of curve subdivisions whose flattened length is up to the specified precision level away from actual curved length. Refer to `path.length()` [documentation](#) for more information about precision and curve flattening.

The `opt` argument is optional. Two properties may be specified, `opt.precision` and `opt.segmentSubdivisions`, which determine maximum error allowed in the calculation (default precision is 3; this corresponds to maximum observed error of 0.1% in flattened curve length). The `opt.segmentSubdivisions` property is an array of individual segments' [subdivision arrays](#). The `path.getSegmentSubdivisions()` [function](#) may be used to obtain the `segmentSubdivisions` array.

g.Path.prototype.toPolylines

`path.toPolylines([opt])`

Return an array of polylines that approximates the path at given precision. If there are no segments, `null` is returned.

Every Moveto segment starts a new subpath within the path. Every subpath has a separate entry in the returned array. If there is only one subpath, an array with one element is returned.

The polyline points are found as endpoints of Lineto and Closepath segments and as endpoints of curve subdivisions whose flattened length is up to the specified precision level away from actual curved length. Refer to `path.length()` [documentation](#) for more information about precision and curve flattening.

The `opt` argument is optional. Two properties may be specified, `opt.precision` and `opt.segmentSubdivisions`, which determine maximum error allowed in the calculation (default precision is 3; this corresponds to maximum observed error of 0.1% in flattened curve length). The `opt.segmentSubdivisions` property is an array of individual segments' [subdivision arrays](#). The `path.getSegmentSubdivisions()` [function](#) may be used to obtain the `segmentSubdivisions` array.

g.Path.prototype.toString

`path.toString()`

Return the path represented as a string.

Whenever a Path object needs to be converted to a string for the SVG `d` attribute, the `path.serialize()` [function](#) should be used. It provides additional error checking to make sure that the path data string is valid.

```
new g.Path('M 150 100 L 100 100 C 100 100 0 150 100 200 Z').toString() = "M 150 100 L 100 100 C 100 100 0 150 100 200 Z"
```

g.Path.prototype.translate

`path.translate(tx [, ty])`

Translate the path by `tx` on the x-axis and by `ty` on the y-axis.

If only `tx` is specified and is a number, `ty` is considered to be zero. If only `tx` is specified and is an object, it is considered to be a point or an object in the form `{ x: [number], y: [number] }`

Modifies this path in-place, and returns it.

g.Path.prototype.validate

`path.validate()`

If the path is not valid (see `path.isValid()` [for reference](#)), make it valid by inserting an initial `'M 0 0'` segment.

Valid paths are not changed. Empty paths are considered valid, so they are left unchanged by this function.

Modifies this path in-place, and returns it.

g.Point.constructor

`g.Point(x [, y])`

Return a new Point object with `x` and `y` coordinates. If `x` is a string, it is considered to be in the form "`[number]` `[number]`" or "`[number]`@`[number]`" where the first number is the x coordinate and the second is the y coordinate.

Examples:

```
var p = g.Point(10, 20);
var p = new g.Point(10, 20);
var p = g.Point('10 20');
var p = g.Point('10@20');
var p = g.Point(g.Point(10, 20));
```

g.Point.fromPolar

`g.Point.fromPolar(distance, angle, origin)`

Returns a new Point object from the given polar coordinates.

g.Point.random

`g.Point.random(x1, x2, y1, y2)`

Returns a new point object with random coordinates that fall within the range `[x1, x2]` and `[y1, y2]` .

g.Point.prototype.adhereToRect

`point.adhereToRect(r)`

If the point lies outside the rectangle `r` , adjust the point so that it becomes the nearest point on the boundary of `r` .

g.Point.prototype.angleBetween

`point.angleBetween(p1, p2)`

Compute the angle (in degrees) between the line from this point to `p1` and the line from this point to `p2` .

The ordering of points `p1` and `p2` is important. The function returns a value between 0 and 180 when the angle (in the direction from `p1` to `p2`) is clockwise, and a value between 180 and 360 when the angle is counterclockwise. The function returns `NaN` if either of the points `p1` and `p2` is coincident with this point.

g.Point.prototype.bearing

`point.bearing(p)`

Return the bearing (cardinal direction) of the line from this point to `p`. The return value is one of the following strings: `'NE'`, `'E'`, `'SE'`, `'S'`, `'SW'`, `'W'`, `'NW'` and `'N'`.

The function returns `'N'` if this point is coincident with the point `p`.

g.Point.prototype.changeInAngle

`point.changeInAngle(dx, dy, ref)`

Return the change in angle (in degrees) that is the result of moving the point from its previous position to its current position.

More specifically, this function computes the angle between the line from the `ref` point to the previous position of this point (i.e. current position `-dx`, `-dy`) and the line from the `ref` point to the current position of this point.

The function returns a positive value between 0 and 180 when the angle (in the direction from previous position of this point to its current position) is clockwise, and a negative value between 0 and -180 when the angle is counterclockwise. The function returns 0 if the previous and current positions of this point are the same (i.e. both `dx` and `dy` are `0`).

g.Point.prototype.chooseClosest

`point.chooseClosest(points)`

Choose the point closest to this point from among `points`. If `points` is an empty array, `null` is returned.

g.Point.prototype.clone

`point.clone()`

Return another point which is a clone of the point.

g.Point.prototype.cross

`point.cross(p1, p2)`

Return the cross product of the vector from the point passing through `p1` and the vector from the point passing through `p2`.

The left-hand rule is used because the coordinate system is left-handed.

g.Point.prototype.difference

`point.difference(dx [, dy])`

Return a point that has coordinates computed as a difference between the point and another point with coordinates `dx` and `dy`.

If only `dx` is specified and is a number, `dy` is considered to be zero. If only `dx` is specified and is an object, it is considered to be another point or an object in the form `{ x: [number], y: [number] }`

g.Point.prototype.distance

`point.distance(p)`

Return the distance between the point and another point `p`.

g.Point.prototype.dot

`point.dot(p)`

Return the dot product of this vector (point) and vector `p`.

g.Point.prototype.equals

`point.equals(p)`

Return `true` if the point equals another point `p`. Return `false` otherwise.

g.Point.prototype.lerp

point.lerp(p, t)

Return an interpolation between the point and point `p` for a parameter `t` in the closed interval $[0, 1]$

g.Point.prototype.magnitude

point.magnitude()

Return the magnitude of the point vector.

g.Point.prototype.manhattanDistance

point.manhattanDistance(p)

Return the manhattan distance between the point and another point `p`.

g.Point.prototype.move

point.move(ref, distance)

Move the point on a line that leads to another point `ref` by a certain `distance`.

g.Point.prototype.normalize

point.normalize(length)

Normalize the point vector and return the point itself.

In other words, scale the line segment between (0, 0) and the point in order for it to have the given `length`. If length is not specified, it is considered to be `1`; in that case, a unit vector is computed.

g.Point.prototype.offset

point.offset(dx [, dy])

Offset the point (change its `x` and `y` coordinates) by `dx` in x-axis and `dy` in y-axis.

If only `dx` is specified and is a number, `dy` is considered to be zero. If only `dx` is specified and is an object, it is considered to be another point or an object in the form `{ x: [number], y: [number] }`

g.Point.prototype.reflection

```
point.reflection(p)
```

Return a point that is a reflection of the point with the center of reflection at point `p`.

g.Point.prototype.rotate

```
point.rotate(origin, angle)
```

Rotate the point by `angle` around `origin`.

g.Point.prototype.round

```
point.round([precision])
```

Round the coordinates of the point to the given precision.

Default precision is `0`.

Modifies this point in-place, and returns it.

g.Point.prototype.scale

```
point.scale(sx, sy [, origin])
```

Scale point by `sx` and `sy` about the given origin.

If origin is not specified, the point is scaled around 0,0.

g.Point.prototype.serialize

```
point.serialize()
```

Returns the point represented as an SVG point string.

```
new g.Point('10 10').serialize() = "10,10"
```

g.Point.prototype.snapToGrid

point.[snapToGrid](#)(gridSize [, gridSizeY])

Snap the point (change its `x` and `y` coordinates) to a grid of size `gridSize` (or `gridSize` x `gridSizeY` for non-uniform grid).

g.Point.prototype.squaredDistance

point.[squaredDistance](#)(p)

Return the squared distance between the point and another point `p`.

Useful for distance comparisons in which real distance is not necessary (saves one `Math.sqrt()` operation).

g.Point.prototype.theta

point.[theta](#)(p)

Return the angle (in degrees) between the point, another point `p` and the x-axis.

g.Point.prototype.toJSON

point.[toJSON](#)()

Return the point as a simple JSON object. For example: { `"x": 0`, `"y": 0` }.

g.Point.prototype.toPolar

point.[toPolar](#)([origin])

Convert rectangular to polar coordinates.

If `origin` is not specified, it is considered to be 0,0.

g.Point.prototype.toString

`point.toString()`

Returns the point as a string.

```
new g.Point('10 10').toString() = "10@10"
```

g.Point.prototype.translate

`point.translate(tx [, ty])`

Translate the point by `tx` on the x-axis and by `ty` on the y-axis.

If only `tx` is specified and is a number, `ty` is considered to be zero. If only `tx` is specified and is an object, it is considered to be another point or an object in the form `{ x: [number], y: [number] }`

g.Point.prototype.update

`point.update(x, y)`

`point.update(p)`

Update the point's `x` and `y` coordinates with new values and return the point itself. Useful for chaining.

Also accepts a single argument in the form of an object `{ x: [number], y: [number] }`.

g.Point.prototype.vectorAngle

`point.vectorAngle(p)`

Compute the angle (in degrees) between the line from (0,0) and this point and the line from (0,0) to `p`.

The function returns a value between 0 and 180 when the angle (in the direction from this point to `p`) is clockwise, and a value between 180 and 360 when the angle is counterclockwise. The function returns `NaN` if called from point (0,0) or if `p` is (0,0).

g.Polygon

While a [Polyline](#) is open ended, the polygon defines a closed area and includes the interior. It's a shape consisting of a series of coordinates in an ordered sequence.

It has the same API as Polyline, but the difference is:

- The interior is taken into account when used as an argument to find an [intersection](#).
- It implicitly adds the final closing segment (e.g. when the length is calculated).

g.Polyline.constructor

`g.Polyline(points)`

Return a new polyline object with points given by the `points` array.

The array can hold anything that can be understood by the Point constructor; Point objects, objects with `x` and `y` attributes, or strings in one of the two valid formats (`"x y"` or `"x@y"`). Array elements that cannot be understood as Points are replaced by 0,0 points.

If `points` are not provided, an empty Polyline is created.

Also accepts a polyline SVG string as an argument; then the `g.Polyline.parse()` [function](#) is used to generate points for the polyline.

g.Polyline.fromRect

`g.Polyline.fromRect(rect)`

Return a new polyline object with points created from provided `rect` .

g.Polyline.parse

`g.Polyline.parse(svgString)`

Return a new polyline object with points created from provided polyline SVG string.

Empty string is accepted and creates an empty polyline.

g.Polyline.prototype.bbox

`polyline.bbox()`

Return a rectangle that is the bounding box of the polyline.

If the polyline contains no points, `null` is returned.

g.Polyline.prototype.clone

`polyline.clone()`

Return another polyline which is a clone of the polyline.

g.Polyline.prototype.closestPoint

`polyline.closestPoint(point)`

Return the point on the path that lies closest to `point`.

If the polyline contains no points, `null` is returned.

g.Polyline.prototype.closestPointLength

`polyline.closestPointLength(point)`

Return the length of the polyline up to the point that lies closest to `point`.

If the polyline contains no points, `null` is returned.

g.Polyline.prototype.closestPointNormalizedLength

`polyline.closestPointNormalizedLength(point)`

Return the normalized length (distance from the start of the path / total path length) of the polyline up to the point that lies closest to `point`.

If the polyline contains no points, `null` is returned.

g.Polyline.prototype.closestPointTangent

`polyline.closestPointTangent(point)`

Return a line tangent to the polyline at the point that lies closest to `point`.

The tangent line starts at the identified closest point. The direction from `start` to `end` is the same as the direction of the polyline segment at the identified point. If the identified closest point is a point of discontinuity (e.g. it is a point shared by two polyline segments with different slopes), the tangent line is constructed for the earlier segment (i.e. the segment closer to the beginning of the polyline).

The algorithm ignores polyline segments that have zero length. If the polyline contains no valid segments (i.e. all polyline points are coincident), `null` is returned. If the polyline has fewer than two points (exclusive), `null` is returned, as well. The `polyline.isDifferentiable()` function may be used in advance to determine whether the polyline contains at least one valid segment for which a tangent may be found.

g.Polyline.prototype.containsPoint

`polyline.containsPoint(p)`

Return `true` if the point `p` is inside the area surrounded by the polyline (inclusive). Return `false` otherwise. Uses the even-odd algorithm to resolve self-intersections.

g.Polyline.prototype.convexHull

`polyline.convexHull()`

Returns a polyline containing the convex hull of this polyline's points.

The output polyline begins with the first point of this polyline that is on the hull. The convex hull points then follow in clockwise order.

The minimum number of points is reported. This means that collinear points lying in the middle of hull edges are not reported. If a group of coincident points is encountered, only the first point (in order of the original polyline) is reported.

If the polyline contains no points, `null` is returned.

g.Polyline.prototype.equals

`polyline.equals(otherPolyline)`

Return `true` if the polyline exactly equals the other polyline.

The two polylines are equal only if every point of one polyline exactly equals the corresponding point of the other polyline. The function returns `true` if `points` arrays of both polylines have no elements.

g.Polyline.prototype.intersectionWithLine

polyline.`intersectionWithLine`(line)

Return an array of the intersection points of the polyline and the line. Return `null` if no intersection exists.

g.Polyline.prototype.isDifferentiable

polyline.`isDifferentiable`()

Return `true` if a tangent line can be found for at least one line of the polyline.

Tangents cannot be found if all points of the polyline are coincident. Additionally, return `false` if the polyline contains only one point or if it contains no points at all.

g.Polyline.prototype.length

polyline.`length`()

Return the length of the polyline.

If the polyline contains no points, `0` is returned.

g.Polyline.prototype.pointAt

polyline.`pointAt`(ratio)

Return a point on the polyline that lies `ratio` (normalized length) away from the beginning of the polyline.

The function expects `ratio` to lie between 0 and 1; values outside the range are constrained to 0 and 1, respectively. If the polyline contains no points, `null` is returned.

g.Polyline.prototype.pointAtLength

polyline.`pointAtLength`(length)

Return a point on the polyline that lies `length` away from the beginning of the polyline.

If negative `length` is provided, the algorithm starts looking from the end of the polyline. If `length` is higher than the length of the polyline, the closest endpoint is returned instead. If the polyline contains no points, `null` is returned.

g.Polyline.prototype.round

`polyline.round([precision])`

Round all coordinates of the polyline to the given precision.

Default precision is `0`.

Modifies this polyline in-place, and returns it.

g.Polyline.prototype.scale

`polyline.scale(sx, sy [, origin])`

Scale the polyline by `sx` and `sy` about the given origin.

If origin is not specified, the polyline is scaled around 0,0.

g.Polyline.prototype.serialize

`polyline.serialize()`

Returns the polyline represented as an SVG points string.

```
new g.Polyline('10,10 20,20 30,30').serialize() = "10,10 20,20 30,30"
```

g.Polyline.prototype.simplify

`polyline.simplify[opt])`

Simplify the polyline by removing non-essential points (i.e. points lying along straight lines within the polyline).

This function modifies the original polyline and returns self. If the polyline has fewer than 3 points, the polyline is not modified.

By default, a point is considered non-essential only if it lies directly on the connection line between the previous and the following point. You can specify a tolerance range by providing a `threshold` value within the `opt` object. Polyline

points that are closer to the connection line than this value (inclusive) are removed; points that are farther from the connection line (exclusive) are kept.

g.Polyline.prototype.tangentAt

`polyline.tangentAt(ratio)`

Return a line tangent to the polyline at point that lies `ratio` (normalized length) away from the beginning of the polyline.

The function expects `ratio` to lie between 0 and 1; values outside the range are constrained to 0 and 1, respectively.

The tangent line starts at the point at `ratio`. The direction from `start` to `end` is the same as the direction of the polyline segment at the specified point. If the specified point is a point of discontinuity (e.g. it is a point shared by two polyline segments with different slopes), the tangent line is constructed for the earlier segment (i.e. the segment closer to the beginning of the polyline).

The algorithm ignores polyline segments that have zero length. If the polyline contains no valid segments (i.e. all polyline points are coincident), `null` is returned. If the polyline has fewer than two points (exclusive), `null` is returned, as well. The `polyline.isDifferentiable()` [function](#) may be used in advance to determine whether the polyline contains at least one valid segment for which a tangent may be found.

g.Polyline.prototype.tangentAtLength

`polyline.tangentAtLength(length)`

Return a line tangent to the polyline at point that lies `length` away from the beginning of the polyline.

If negative `length` is provided, the algorithm starts looking from the end of the polyline. If `length` is higher than the length of the polyline, a line tangent to the closest valid polyline endpoint is returned instead.

The tangent line starts at the point at `length`. The direction from `start` to `end` is the same as the direction of the polyline segment at the specified point. If the specified point is a point of discontinuity (e.g. it is a point shared by two polyline segments with different slopes), the tangent line is constructed for the earlier segment (i.e. the segment closer to the beginning of the polyline).

The algorithm ignores polyline segments that have zero length. If the polyline contains no valid segments (i.e. all polyline points are coincident), `null` is returned. If the polyline has fewer than two points (exclusive), `null` is returned, as well. The `polyline.isDifferentiable()` [function](#) may be used in advance to determine whether the polyline contains at least one valid segment for which a tangent may be found.

g.Polyline.prototype.toString

`polyline.toString()`

Returns the polyline as a string.

```
new g.Polyline('10,10 20,20 30,30').toString() = "10@10,20@20,30@30"
```

g.Polyline.prototype.translate

polyline.**translate**(tx [, ty])

Translate the polyline by `tx` on the x-axis and by `ty` on the y-axis.

If only `tx` is specified and is a number, `ty` is considered to be zero. If only `tx` is specified and is an object, it is considered to be a point or an object in the form `{ x: [number], y: [number] }`

g.Rect.constructor

g.**Rect**(x, y, width, height)

Return a new rectangle object with top left corner at point with coordinates `x`, `y` and dimensions `width` and `height`.

Also accepts a single argument in the form of an object `{ x: [number], y: [number], width: [number], height: [number] }`.

g.Rect.fromEllipse

g.Rect.**fromEllipse**(ellipse)

Returns a new rectangle object from the given ellipse.

g.Rect.fromPointUnion

g.Rect.**fromPointUnion**(...points)

Returns a new rectangle object that is large enough to contain given `points`, where each point is an object with `x` and `y` properties. Examples:

```
const rect = g.Rect.fromPointUnion(new g.Point(0, 0), new g.Point(20, 20));
const rect2 = g.Rect.fromPointUnion({ x: 0, y: 0 }, { x: 20, y: 20 });
// { x: 0, y: 0, width: 20, height: 20 }
```

g.Rect.fromRectUnion

`g.Rect.fromRectUnion(...rects)`

Returns a new rectangle object that is large enough to contain given `rects`, where each `rect` is an object with `x`, `y`, `width` and `height` properties. Examples:

```
const rect = g.Rect.fromRectUnion(new g.Rect(0, 0, 10, 10), new g.Rect(10, 10, 10, 10));
const rect2 = g.Rect.fromRectUnion({ x: 0, y: 0, height: 10, width: 10 }, { x: 10, y: 10, height: 10, width: 10 });
// { x: 0, y: 0, width: 20, height: 20 }
```

g.Rect.prototype.bbox

`rect.bbox([angle])`

Return a rectangle that is the bounding box of the rectangle.

If `angle` is specified, the bounding box calculation will take into account the rotation of the rectangle by `angle` degrees around its center.

g.Rect.prototype.bottomLeft

`rect.bottomLeft()`

Return the point that is the bottom left corner of the rectangle.

g.Rect.prototype.bottomLine

`rect.bottomLine()`

Return the bottom line of the rectangle.

g.Rect.prototype.bottomMiddle

`rect.bottomMiddle()`

Return the point that is at the bottom middle of the rectangle.

g.Rect.prototype.bottomRight

```
rect.bottomRight()
```

Return the point that is the bottom right corner of the rectangle.

g.Rect.prototype.center

```
rect.center()
```

Return the point that is the center of the rectangle.

g.Rect.prototype.clone

```
rect.clone()
```

Return another rectangle which is a clone of the rectangle.

g.Rect.prototype.containsPoint

```
rect.containsPoint(p)
```

Return **true** if the point `p` is inside the rectangle (inclusive). Return **false** otherwise.

g.Rect.prototype.containsRect

```
rect.containsRect(r)
```

Return **true** if the rectangle `r` is (completely) inside the rectangle (inclusive). Return **false** otherwise.

g.Rect.prototype.corner

```
rect.corner()
```

Return the point that is the bottom right corner of the rectangle.

g.Rect.prototype.equals

`rect.equals(r)`

Return `true` if the rectangle equals another rectangle `r`. Return `false` otherwise.

g.Rect.prototype.inflate

`rect.inflate(dx [, dy])`

Return a rectangle inflated in axis `x` by `2 * dx` and in axis `y` by `2 * dy`.

When the function is called with a single parameter, the resulting rectangle is inflated by `2 * dx` in both `x` and `y` axis.

g.Rect.prototype.intersect

`rect.intersect(r)`

Return a rectangle object that is a subtraction of the two rectangles if such an object exists (the two rectangles intersect). Return `null` otherwise.

g.Rect.prototype.intersectionWithLine

`rect.intersectionWithLine(line)`

Return an array of the intersection points of the rectangle and the line. Return `null` if no intersection exists.

g.Rect.prototype.intersectionWithLineFromCenterToPoint

`rect.intersectionWithLineFromCenterToPoint(p [, angle])`

Return the point on the boundary of the rectangle that is the intersection of the rectangle with a line starting in the center of the rectangle ending in the point `p`.

If `angle` is specified, the intersection will take into account the rotation of the rectangle by `angle` degrees around its center.

g.Rect.prototype.leftLine

```
rect.leftLine()
```

Return the left line of the rectangle.

g.Rect.prototype.leftMiddle

```
rect.leftMiddle()
```

Return the point that is at the left middle of the rectangle.

g.Rect.prototype.maxRectScaleToFit

```
rect.maxRectScaleToFit(limitRectangle [, origin])
```

Returns an object where `sx` and `sy` give the maximum scaling that can be applied to the rectangle so that it would still fit into `limitRectangle`.

If `origin` is specified, the rectangle is scaled around it; otherwise, it is scaled around its center.

g.Rect.prototype.maxRectUniformScaleToFit

```
rect.maxRectUniformScaleToFit(limitRectangle [, origin])
```

Returns a number that specifies the maximum scaling that can be applied to the rectangle along both axes so that it would still fit into `limitRectangle`.

If `origin` is specified, the rectangle is scaled around it; otherwise, it is scaled around its center.

g.Rect.prototype.moveAndExpand

```
rect.moveAndExpand(r)
```

Offset the rectangle by `r.x` and `r.y` and expand it by `r.width` and `r.height`.

g.Rect.prototype.normalize

```
rect.normalize()
```

Normalize the rectangle, i.e. make it so that it has non-negative width and height.

If width is less than 0, the function swaps left and right corners and if height is less than 0, the top and bottom corners are swapped.

g.Rect.prototype.offset

```
rect.offset(dx [, dy])
```

Offset the rectangle (change its x and y coordinates) by dx in x-axis and dy in y-axis.

If only dx is specified and is a number, dy is considered to be zero. If only dx is specified and is an object, it is considered to be another point or an object in the form { x: [number], y: [number] }

g.Rect.prototype.origin

```
rect.origin()
```

Return the point that is the top left corner of the rectangle.

g.Rect.prototype.pointNearestToPoint

```
rect.pointNearestToPoint(p)
```

Return the point on the boundary of the rectangle nearest to the point p.

g.Rect.prototype.rightLine

```
rect.rightLine()
```

Return the right line of the rectangle.

g.Rect.prototype.rightMiddle

```
rect.rightMiddle()
```

Return the point that is at the right middle of the rectangle.

g.Rect.prototype.rotateAroundCenter

```
rect.rotateAroundCenter(angle)
```

Rotates the rectangle around its center. The method updates the size and the position to match the bounding box of the rotated rectangle.

g.Rect.prototype.round

```
rect.round([precision])
```

Round all coordinates of the rectangle to the given precision.

Default precision is 0 .

Modifies this rectangle in-place, and returns it.

g.Rect.prototype.scale

```
rect.scale(sx, sy [, origin])
```

Scale the rectangle by `sx` , `sy` around the given `origin` .

If `origin` is not specified, the rectangle is scaled around the point 0,0.

g.Rect.prototype.sideNearestToPoint

```
rect.sideNearestToPoint(p)
```

Return a string ("top" , "left" , "right" or "bottom") denoting the side of the rectangle which is nearest to the point `p` .

g.Rect.prototype.snapToGrid

`rect.snapToGrid(gx, gy)`

Adjust the position and dimensions of the rectangle such that its edges are on the nearest increment of `gx` on the x-axis and `gy` on the y-axis.

g.Rect.prototype.toJSON

`rect.toJSON()`

Return the rectangle as a simple JSON object. For example: { `"x": 0`, `"y": 0`, `"width": 40`, `"height": 40` } .

g.Rect.prototype.topLeft

`rect.topLeft()`

Return the point that is the top left corner of the rectangle.

g.Rect.prototype.topLine

`rect.topLine()`

Return the top line of the rectangle.

g.Rect.prototype.topMiddle

`rect.topMiddle()`

Return the point that is at the top middle of the rectangle.

g.Rect.prototype.topRight

`rect.topRight()`

Return the point that is the top right corner of the rectangle.

g.Rect.prototype.toString

`rect.toString()`

Returns the rectangle as a string.

```
new g.Rect(10, 20, 30, 40).toString() = "10@20 40@60"
```

g.Rect.prototype.translate

`rect.translate(tx [, ty])`

Translate the rectangle by `tx` on the x-axis and `ty` on the y-axis.

If only `tx` is specified and is a number, `ty` is considered to be zero. If only `tx` is specified and is an object, it is considered to be a point or an object in the form `{ x: [number], y: [number] }`

g.Rect.prototype.union

`rect.union(r)`

Return a rectangle that is a union of this rectangle and rectangle `r`.

g.Rect.prototype.update

`rect.update(x, y, width, height)`

`rect.update(r)`

Update the rect's `x`, `y`, `width` and `height` properties with new values and return the rect itself. Useful for chaining.

Also accepts a single argument in the form of an object `{ x: [number], y: [number], width: [number], height: [number] }`.

g.scale.linear

```
g.scale.linear(domain, range, value)
```

Return the `value` from the `domain` interval linearly scaled to the `range` interval.

Both `domain` and `range` intervals must be specified as arrays with two numbers specifying start and end of the interval.

Vectorizer API

V

JointJS exports three global variables, the `joint` namespace, the `V` variable and the `g` variable (that is described later in the [Geometry](#) section). The `V` variable is a function of a little SVG helper library that we call “Vectorizer”. The reason why this library sits in its own namespace and not inside the `joint` namespace is that it can be used completely standalone, even without JointJS. It is a very helpful library making life easier when dealing with SVG. You can think of it as of a very lightweight jQuery for SVG. If you want to use Vectorizer as a standalone library, see the download page that contains both the [development and minified versions](#).

V.constructor

`V(svg)`

Return a Vectorizer object. If `svg` parameter is a string, construct SVG DOM elements from the string markup. If `svg` is an SVG DOM element, just wrap that element by the Vectorizer object and return it. You can think of this function as of the `jQuery.$` function except that the `V` function does not accept selectors. The Vectorizer object contains a reference to the original SVG DOM element in its `node` property.

```
var vel = V('<g><rect/><text/></g>');
console.log(vel.node);
```

V.annotateString

`V.annotateString(string, annotations [, opt])`

Apply the annotations (see [v.text\(\)](#)) to the string given.

V.createCDATASection

`V.createCDATASection(data)`

Return an [XML CDATA Section](#) node containing the `data` string.

V.createSVGMatrix

`V.createSVGMatrix(extension)`

Return the [SVG transformation matrix](#) initialized with the matrix `extension`. `extension` is an object of the form: {
`a`: [number], `b`: [number], `c`: [number], `d`: [number], `e`: [number], `f`: [number]}.

V.createSVGPoint

`V.createSVGPoint(x, y)`

Return the [SVG point](#) object initialized with the `x` and `y` coordinates.

V.createSVGStyle

`V.createSVGStyle(stylesheets)`

Return an [SVGStyleElement](#) containing the CSS `stylesheet`.

```
const style = V.createSVGStyle(`
  rect, circle {
    fill: white;
    stroke: black;
  }
`);
document.getElementById('my-svg').appendChild(style);
```

V.createSVGTransform

`V.createSVGTransform([matrix])`

Returns a [SVG transform](#) object.

V.decomposeMatrix

`V.decomposeMatrix(matrix)`

Decompose the SVG transformation matrix into separate transformations. Return an object of the form: { translateX: [number], translateY: [number], scaleX: [number], scaleY: [number], skewX: [number], skewY: [number], rotation: [number] }.

V.findAnnotationsAtIndex

V.findAnnotationsAtIndex(annotations, index)

Find all the annotations (see v.text()) that apply to the character at `index`.

V.findAnnotationsBetweenIndexes

V.findAnnotationsBetweenIndexes(annotations, start, end)

Find all the annotations (see v.text()) that apply to all the characters in the `start` and `end` range.

V.isSVGGraphicsElement

V.isSVGGraphicsElement(object)

Return `true` if `object` is an instance of SVGGraphicsElement.

V.isVElement

V.isVElement(object)

Return `true` if `object` is a vectorizer element.

V.normalizePathData

V.normalizePathData(pathData)

Convert provided SVG path data string into a normalized path data string. The normalization uses a restricted subset of path commands; all segments are translated into lineto, curveto, moveto, and closepath segments. Relative path commands are changed into their absolute counterparts, and chaining of coordinates is disallowed.

The function will always return a valid path data string; if an input string cannot be normalized, `'M 0 0'` is returned.

Command translations:

```

V.normalizePathData('M 10 10 H 20') === 'M 10 10 L 20 10';
V.normalizePathData('M 10 10 V 20') === 'M 10 10 L 10 20';
V.normalizePathData('M 10 20 C 10 10 25 10 25 20 S 40 30 40 20') === 'M 10 20 C 10 10
25 10 25 20 C 25 30 40 30 40 20';
V.normalizePathData('M 20 20 Q 40 0 60 20') === 'M 20 20 C 33.33333333333333
6.666666666666666 46.666666666666664 6.666666666666666 60 20';
V.normalizePathData('M 20 20 Q 40 0 60 20 T 100 20') === 'M 20 20 C 33.33333333333333
6.666666666666666 46.666666666666664 6.666666666666666 60 20 C 73.33333333333333
33.33333333333333 86.66666666666666 33.33333333333333 100 20';
V.normalizePathData('M 30 15 A 15 15 0 0 0 15 30') === 'M 30 15 C 21.715728752538098
15.000000000000002 14.999999999999998 21.715728752538098 15 30';

```

Relative-to-absolute normalizations:

```

V.normalizePathData('m 10 10') === 'M 10 10';
V.normalizePathData('M 10 10 m 10 10') === 'M 10 10 M 20 20';
V.normalizePathData('M 10 10 l 10 10') === 'M 10 10 L 20 20';
V.normalizePathData('M 10 10 c 0 10 10 10 10 0') === 'M 10 10 C 10 20 20 20 20 10';
V.normalizePathData('M 10 10 z') === 'M 10 10 Z'

```

Transcription of chained coordinates:

```

V.normalizePathData('M 10 10 20 20') === 'M 10 10 L 20 20';
V.normalizePathData('M 10 10 L 20 20 30 30') === 'M 10 10 L 20 20 L 30 30';
V.normalizePathData('M 10 10 C 10 20 20 20 20 10 20 0 30 0 30 10') === 'M 10 10 C 10
20 20 20 20 10 C 20 0 30 0 30 10';

```

Edge cases:

```

V.normalizePathData('L 10 10') === 'M 0 0 L 10 10';
V.normalizePathData('C 0 10 10 10 10 0') === 'M 0 0 C 0 10 10 10 10 0';
V.normalizePathData('Z') === 'M 0 0 Z';

V.normalizePathData('M 10 10 Z L 20 20') === 'M 10 10 Z L 20 20';
V.normalizePathData('M 10 10 Z C 10 20 20 20 20 10') === 'M 10 10 Z C 10 20 20 20 20
10';
V.normalizePathData('M 10 10 Z Z') === 'M 10 10 Z Z';

V.normalizePathData('') === 'M 0 0'; // empty string
V.normalizePathData('X') === 'M 0 0'; // invalid command

V.normalizePathData('M') === 'M 0 0'; // no arguments for a command that needs them
V.normalizePathData('M 10') === 'M 0 0'; // too few arguments
V.normalizePathData('M 10 10 20') === 'M 10 10'; // too many arguments

```

```
V.normalizePathData('X M 10 10') === 'M 10 10'; // mixing invalid and valid commands
V.normalizePathData('X M 10 10 X L 20 20') === 'M 10 10 L 20 20'; // invalid commands
interspersed with valid commands
```

V.rectToPath

V.rectToPath(r)

Convert a rectangle `r` to SVG path commands. `r` is an object of the form { `x`: [number], `y`: [number], `width`: [number], `height`: [number], `top-ry`: [number], `top-ry`: [number], `bottom-rx`: [number], `bottom-ry`: [number] } where `x`, `y`, `width`, `height` are the usual rectangle attributes and `[top-/bottom-]rx/ry` allows for specifying radius of the rectangle for all its sides (as opposed to the built-in SVG rectangle that has only `rx` and `ry` attributes).

V.shiftAnnotations

V.shiftAnnotations(annotations, index, offset)

Shift all the annotations (see [v.text\(\)](#)) after character at `index` by `offset` positions.

V.transformLine

V.transformLine(line, matrix)

Transform a line by an SVG transformation represented by `matrix`.

```
var line = new g.Line({ x: 10, y: 20 }, { x: 100, y: 200 });
var rotatedLine = V.transformLine(line, V.createSVGMatrix().rotate(45));
```

V.transformPoint

V.transformPoint(p, matrix)

Transform a point specified by `p` (an object with `x` and `y` coordinates) by an SVG transformation represented by `matrix`. This is a convenient shortcut to calling:

```
var p = mySVGDocument.createSVGPoint();
p.x = x;
```

```
p.y = y;  
p.matrixTransform(matrix)
```

V.transformPolyline

`V.transformPolyline(polyline, matrix)`

Transform a polyline (or an array of points) by an SVG transformation represented by `matrix`.

```
var polyline = new g.Polyline([ { x: 10, y: 20 }, { x: 20, y: 30 }, { x: 100, y: 30 } ]);  
var rotatedPolyline = V.transformPolyline(polyline, V.createSVGMatrix().rotate(45));
```

V.transformRect

`V.transformRect(r, matrix)`

Transform a rectangle specified by `r` (an object with `x`, `y`, `width` and `height` properties) by an SVG transformation represented by `matrix`. This function is used internally in the [V.prototype.bbox](#) method to return a bounding box of an SVG element relative to another SVG parent element. However, this can be used elsewhere too, that's why it is publicly exposed. Whenever you need to transform a bounding box to a coordinate system of another element, use this function. To get the transformation `matrix` of the target element relative to which you want to transform your rectangle, use e.g. the native SVG method: `var matrix = svgSourceElement.getTransformToElement(svgTargetElement)`.

V.transformStringToMatrix

`V.transformStringToMatrix(transformString)`

Return the [SVG transformation matrix](#) built from the `transformString`. E.g. `'translate(10,10) scale(2,2)'` will result in matrix `{ a: 2, b: 0, c: 0, d: 2, e: 10, f: 10 }`.

V.prototype.addClass

`vel.addClass(className)`

Append `className` to the element `class` attribute if it doesn't contain it already. Return the Vectorizer object for easy chaining.

V.prototype.animateAlongPath

vel.[animateAlongPath](#)(attrs, path)

Animate the element along the `path` SVG element (or Vectorizer object). `attrs` contain [Animation Timing attributes](#) describing the animation. The following example shows how to send a token along a JointJS link element:

```
var c = V('circle', { r: 8, fill: 'red' });
c.animateAlongPath({ dur: '4s', repeatCount: 'indefinite' },
paper.findViewByModel(myLink).$('.connection')[0]);
V(paper.svg).append(c);
```

V.prototype.append

vel.[append](#)(els)

Append another element (or elements) `els` as the last child of the element. `els` can be Vectorizer object(s) or SVG DOM element(s).

V.prototype.appendTo

vel.[appendTo](#)(node)

Append `vel.node` as a child to `node`.

V.prototype.attr

vel.[attr](#)(name, value)

Set SVG attribute with `name` and `value` on the element. If `name` is an object of the form `{ [name]: [value] }`, more attributes will be set in one go.

V.prototype.bbox

vel.[bbox](#)([withoutTransformations, target])

Return the bounding box of the element after transformations are applied. If `withoutTransformations` is `true`, transformations of the element will not be considered when computing the bounding box. If `target` is specified, bounding box will be computed relatively to the `target` element.

V.prototype.before

`vel.before(els)`

Adds the given element (or elements) `els` before the Vectorizer element `vel` as a child of the parent node. `els` can be Vectorizer object(s) or SVG DOM element(s).

V.prototype.children

`vel.children()`

Returns an array of Vectorizer-wrapped children of this element's node.

V.prototype.clone

`vel.clone()`

Clone the Vectorizer object creating a brand new copy of the element. This clone is not automatically added to the DOM.

V.prototype.contains

`vel.contains(element)`

Return `true` if `vel.node` is the parent/ancestor of `element` node in the DOM. Also, return `true` if `vel.node` is `element` node.

V.prototype.convertToPath

`vel.convertToPath()`

Convert an SVG element to the SVG path element. It currently works for `<path>`, `<line>`, `<polygon>`, `<polyline>`, `<ellipse>`, `<circle>` and `<rect>`.

V.prototype.convertToPathData

`vel.convertToPathData()`

Convert an SVG element to the SVG path data (string of SVG path commands). It currently works for `<path>` , `<line>` , `<polygon>` , `<polyline>` , `<ellipse>` , `<circle>` and `<rect>` .

V.prototype.defs

`vel.defs()`

Return the Vectorizer object for the first `<defs>` element of the root SVG element. Note that the `<defs>` element is a good place to put referenced SVG elements like gradients, clip paths, filters and others.

V.prototype.empty

`vel.empty()`

Removes all the child nodes from the Vectorizer element.

V.prototype.find

`vel.find(selector)`

Return all elements wrapped in the Vectorizer object matching the `selector` .

V.prototype.findIntersection

`vel.findIntersection(ref, target)`

Find the intersection of a line starting in the center of the SVG node ending in the point `ref` (an object of the form `{ x: Number, y: Number }`). `target` is an SVG element to which this node's transformations are relative to. In JointJS, `target` is usually the `paper.viewport` SVG group element. Note that `ref` point must be in the coordinate system of the `target` for this function to work properly. This function returns a point in the `target` coordinte system (the same system as `ref` is in) if an intersection is found. Returns `undefined` otherwise.

V.prototype.findOne

`vel.findOne(selector)`

Return the first element wrapped in the Vectorizer object matching the `selector` Return `undefined` if not such element was found.

V.prototype.findParentByClass

`vel.findParentByClass(className [, terminator])`

Traversing the DOM hierarchy outwards, up to optional `terminator` node (exclusive), find the first ancestor node that has the specified class. If no ancestor before terminator has `className`, return `null`.

V.prototype.getBBox

`vel.getBBox([options])`

Return the bounding box of the element after transformations are applied. Unlike `vel.bbox()`, this function fixes a browser implementation bug to return the correct bounding box if this element is a group of svg elements (if `options.recursive` is specified). The `options` argument is optional. Two attributes may be provided inside the object. If `options.target` is not undefined, bounding box will be computed relatively to the transformations of the `target` element; if it is undefined, the bounding box will be computed relatively to the element calling the function. If `options.recursive` is true, the bounding box is computed as a union of all individual descendants' bounding boxes.

V.prototype.getTransformToElement

`vel.getTransformToElement(toElement)`

Return an `SVGMatrix` that specifies the transformation necessary to convert vel coordinate system into `toElement` coordinate system.

V.prototype.hasClass

`vel.hasClass(className)`

Return `true` if the element contains `className` in its `class` attribute. Return `false` otherwise.

V.prototype.id

`vel.id`

Element's unique identifier. A property of the element for getting/setting `vel.node.id`.

V.prototype.index

`vel.index()`

Return the index of the SVG element amongst its siblings.

V.prototype.normalizePath

`vel.normalizePath()`

Normalize this element's `d` attribute. SVGPathElements without a path data attribute obtain a value of `'M 0 0'`. If this element is not an SVGPathElement, there are no changes. Return `this`, to allow easy chaining of V methods.

V.prototype.parent

`vel.parent()`

Returns the Vectorizer-wrapped parent of the specified node `vel` in the DOM tree. It returns `null` if the node has no parent.

V.prototype.prepend

`vel.prepend(els)`

Prepend another element (or elements) `els` as the first child of the element. `els` can be Vectorizer object(s) or SVG DOM element(s).

V.prototype.remove

`vel.remove()`

Remove the element from the DOM.

V.prototype.removeAttr

`vel.removeAttribute(attribute)`

Remove the specified attribute from the `vel.node`.

V.prototype.removeClass

```
vel.removeClass(className)
```

Remove `className` from the element `class` attribute if it contains it. Return the Vectorizer object for easy chaining.

V.prototype.rotate

```
vel.rotate(angle [, cx, cy])
```

Rotate the element by `angle` degrees. If the optional `cx` and `cy` coordinates are passed, they will be used as an origin for the rotation.

V.prototype.sample

```
vel.sample(interval)
```

Sample the underlying SVG element (it currently works only on paths - where it is most useful anyway). Return an array of objects of the form `{ x: Number, y: Number, distance: Number }`. Each of these objects represent a point on the path. This basically creates a discrete representation of the path (which is possible a curve). The sampling `interval` defines the accuracy of the sampling. In other words, we travel from the beginning of the path to the end by `interval` distance (on the path, not between the resulting points) and collect the discrete points on the path. This is very useful in many situations. For example, SVG does not provide a built-in mechanism to find intersections between two paths. Using sampling, we can just generate bunch of points for each of the path and find the closest ones from each set.

V.prototype.scale

```
vel.scale(sx [, sy])
```

Scale the element by `sx` and `sy` factors. If `sy` is not passed, it will be considered the same as `sx`.

V.prototype.setAttribute

```
vel.setAttribute(name, value)
```

Assign `value` to the attribute with `name`. Attribute names can be namespaced; for example, `<image>` elements have a `xlink:href` attribute to set the source of the image.

V.prototype.setAttributes

`vel.setAttributes(attributes)`

Take an `attributes` object with `{ attribute: value }` pairs and assign them to `vel`.

V.prototype.svg

`vel.svg()`

Return the Vectorizer object for the root SVG element of the element.

V.prototype.tagName

`vel.tagName()`

Return the tag name of this element, in uppercase.

V.prototype.text

`vel.text(content [, opt])`

Set the `text content` of the element. This only makes sense for the `<text>` element. This method can deal with multi-line text in case the `content` string contains newline characters (`\n`).

```
t.text('my text');
t.text('my multiline\ntext');
```

Alignment

`opt.lineHeight` can be optionally used to set the line height of the text. It defaults to `'1em'`. The `opt.lineHeight` can also be set to `'auto'` in which case it is left on Vectorizer to set the best possible line height. This is useful if you annotate the text making it a rich-text (see below) and you don't want to set the line height to a fixed value for all the lines.

Use `opt.textVerticalAnchor` to control the y-axis alignment relatively to the initial text position. The anchor position can be set by a keyword or a value in `px` or `em`.

- `'top'` - at the top of the text
- `'bottom'` - at the bottom of the text
- `'middle'` - in the middle of the text

- `'0.3em'` - in the middle of the first line
- `'0.8em'` - at the top edge of the first line (*default*)
- `'-0.3em'` - at the bottom edge of the first line
- `0` or `'0em'` - at the baseline of the first line
- `20` or `'20px'` - 20 pixels up from the baseline of the first line

```
t.text('my text\nwith custom line height', {
  lineHeight: '2em'
});
t.attr('font-size', 20).text('vertically aligned text', {
  textVerticalAnchor: 'middle'
});
```

Note: The method uses a heuristic algorithm to align the text. To improve the results, define the text `font-size` attribute in `px` and `lineHeight` option either in `px` or `em`. If a keyword or annotations are used and no `font-size` attribute is set on the text element, the font size defaults to `16`.

Annotations

If `opt.annotations` array is set, the text will be annotated by the attributes defined in the annotations array. This means that you can easily render rich text. Each annotation in the annotations array is an object of the form `{ start: Number, end: Number, attrs: Object }` where `start` (inclusive) and `end` (exclusive) define the range of the text content where the `attrs` SVG Attributes will be applied. If there are overlapping annotations, they will be smartly merged (classes concatenated, attributes merged, `style` can be always defined either as a string or an object).

```
var text = V('text', { x: 250, y: 150, fill: 'black' });
text.text('This is a rich text.\nThis text goes to multiple lines.', {
  lineHeight: 'auto',
  annotations: [
    { start: 5, end: 10, attrs: { fill: 'red', 'font-size': 30, rotate: '20' }},
    { start: 7, end: 15, attrs: { fill: 'blue' }},
    { start: 20, end: 30, attrs: { fill: 'blue', 'class': 'text-link', style: 'text-decoration: underline' }}
  ]
});
svg.append(text);
```

If `opt.includeAnnotationIndices` is set to `true`, each `<tspan>` will contain the `'annotations'` attribute with comma-separated indices of annotations that apply to that piece of text. Vectorizer provides some useful functions for working with annotations. Those are [V.findAnnotationsAtIndex\(\)](#), [V.findAnnotationsBetweenIndexes\(\)](#), [V.shiftAnnotations\(\)](#), and [V.annotateString\(\)](#).

Text Along Path

To make the text go along a path, use `opt.textPath`. It can be either string or an object.

string

The provided `textPath` specifies the path data of the path the text should go along.

```
t.text('text that goes along a path', { textPath: 'M 0 100 Q 30 10 100 0' });
```

object

The provided `textPath` can contain a `d` property that specifies the path data of the path the text should go along, and optionally other attributes that will be set on the automatically created `<textPath>` SVG element, such as `startOffset`. If `textPath` contains `xlink:href` property, an existing path is used as reference.

```
t.text('another text that goes along a path', {
  textPath: {
    d: 'M 0 100 Q 30 10 100 0',
    startOffset: 50
  }
});
t.text('text that goes along an existing path', {
  textPath: { 'xlink:href': '#path1' }
});
```

Misc

By default, the `SVGTextElement` without the content is not displayed (It is not clickable and occupies no space). To prevent this, use `opt.displayEmpty` and set it to `true`.

V.prototype.toGeometryShape

`vel.toGeometryShape()`

Convert the `SVGElement` to an equivalent geometric shape. The element's transformations are not taken into account.

SVGElement	Shape
SVGRectElement	<code>g.Rect</code>
SVGLineElement	<code>g.Line</code>
SVGCircleElement	<code>g.Ellipse</code>
SVGEllipseElement	<code>g.Ellipse</code>

SVGPolygonElement	g.Polyline
SVGPolylineElement	g.Polyline
SVGPathElement	g.Path
<i>others</i>	g.Rect

V.prototype.toggleClass

`vel.toggleClass(className, switch)`

Add or remove `className` from the element `class` attribute depending on either the class's presence or the value of the switch argument.

V.prototype.toLocalPoint

`vel.toLocalPoint(x, y)`

Convert a global point with coordinates `x` and `y` into the coordinate space of the element.

V.prototype.transform

`vel.transform([matrix], [opt])`

When `matrix` is **not** provided, returns the current transformation matrix of the Vectorizer element.

When `matrix` is provided, applies the provided transformation matrix to the Vectorizer element.

You can clear previous transformations by passing `opt` with property `absolute:true`

V.prototype.translate

`vel.translate(tx [, ty])`

Translate the element by `tx` pixels in x axis and `ty` pixels in y axis. `ty` is optional in which case the translation in y axis is considered zero.

V.prototype.translateAndAutoOrient

```
vel.translateAndAutoOrient(position, reference, target)
```

Auto-orient the element. This basically implements the `orient=auto` attribute of [markers](#). The easiest way of understanding on what this does is to imagine the element is an arrowhead. Calling this method on the arrowhead makes it point to the `position` point while being auto-oriented (properly rotated) towards the `reference` point. `target` is the element relative to which the transformations are applied. Usually the root SVG element which is also default if not `target` is passed.

V.prototype.translateCenterToPoint

```
vel.translateCenterToPoint(p)
```

Translate the element so that its new center will be at point `p`. `p` is an object of the form `{ x: [number], y: [number] }`.

JointJS+

JointJS+ is a powerful and modern toolkit for building **visual tools** of various kinds. This documentation gives you an overview and many examples on how you can use the plugins of the JointJS+ toolkit to build groundbreaking applications in a fraction of the time.

If you're looking for documentation on the **core JointJS library**, go to the [JointJS API reference page](#).

For more information about the **JointJS+ toolkit, licensing and online purchase**, see the [About JointJS+ page](#).

Browse through the menu on the left to see the documentation for the JointJS+ plugins.

The JointJS+ toolkit is split into modules. Every plugin has a module identifier in its name. This helps you in orientating around the framework. Here is the list of modules with a short description:

- **ui** - User Interface Widgets. These widgets are user facing components that the user can interact with.
- **dia** - Programmer's toolkit. This includes undo/redo, graph manipulation, validation and more.
- **shapes** - Different kinds of diagram shapes including a wide range of charts, BPMN shapes, and Record shapes.
- **format** - Exports to vector and raster formats, imports and print.
- **storage** - Storage.
- **alg** - Algorithms and data structures.
- **layout** - Automatic layouts.
- **elementTools** - Render a certain type of control element on top of an ElementView.
- **graphUtils** - Utilities for algorithms and data structures.

Installation

Include `rappid.min.css` and `rappid.min.js` files in your HTML together with all the JointJS+ dependencies (jQuery, Lodash and Backbone).

```
<link rel="stylesheet" type="text/css" href="dist/rappid.min.css">
<script src="lib/jquery/jquery.min.js"></script>
<script src="lib/lodash/dist/lodash.min.js"></script>
<script src="lib/backbone/backbone-min.js"></script>
<script src="dist/rappid.min.js"></script>
```

It is highly recommended to use the exact same versions of all the dependencies as they are included in the package.

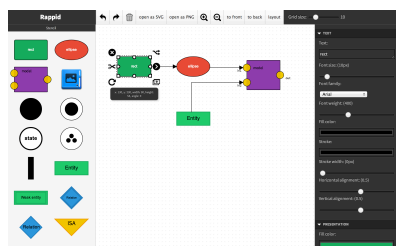
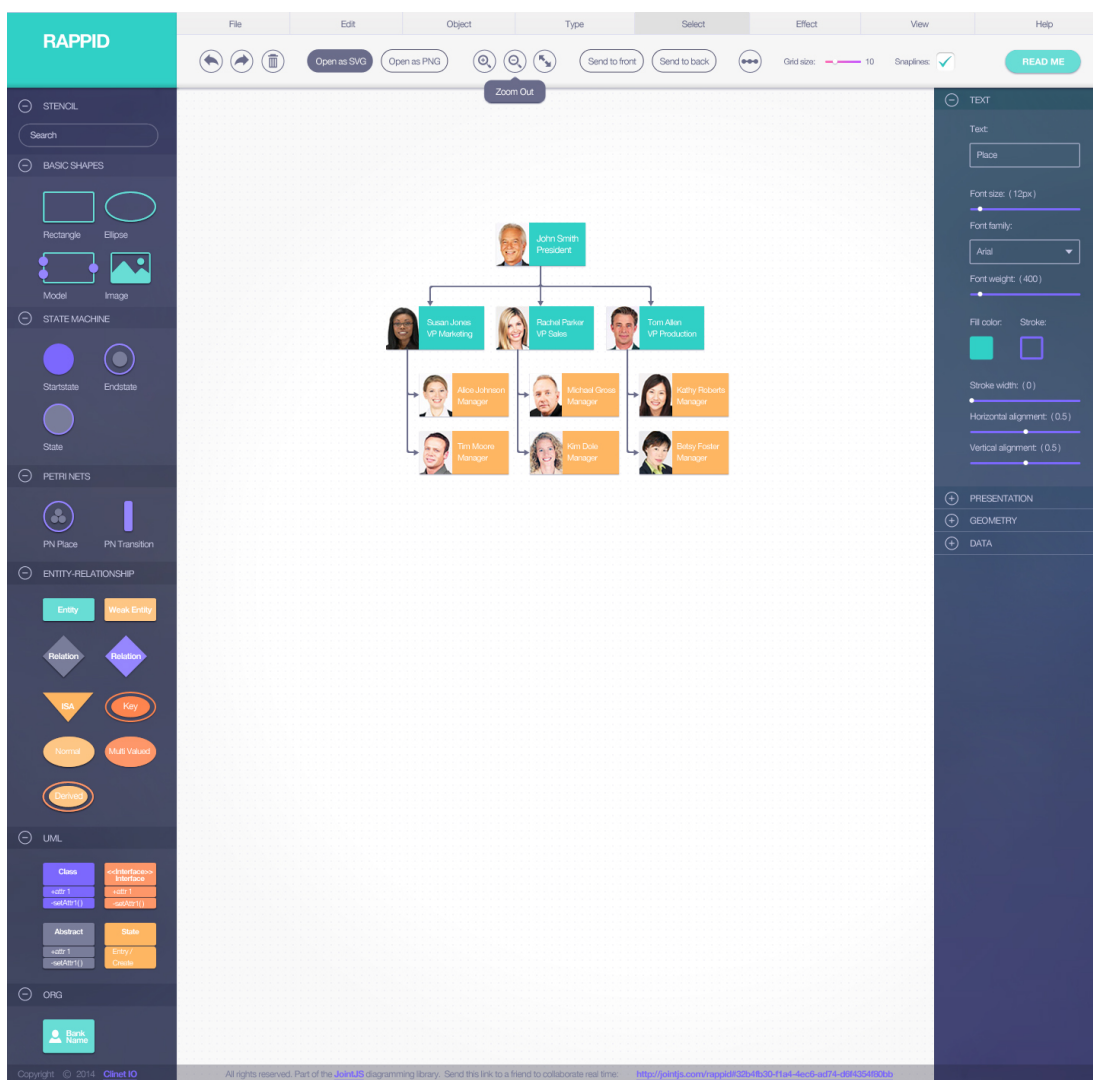
Note that each component documentation page has an *Installation* section. You do not need to follow the instructions from the section if you installed JointJS+ as described above (with `rappid.min.js`, `rappid.min.css` and all the dependencies). The Installation section of the components is only useful if you do not want to include the whole framework but only some of its components. In this case, you have to include all the dependencies, the JointJS Core library and then only the components you want to use:

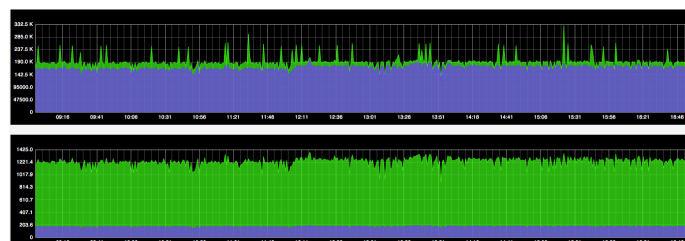
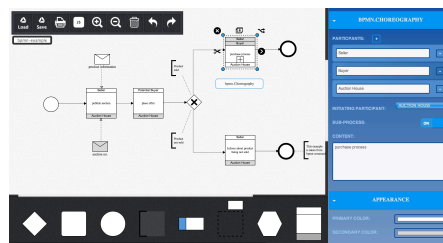
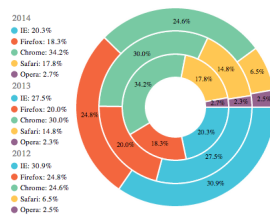
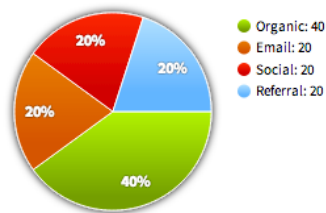
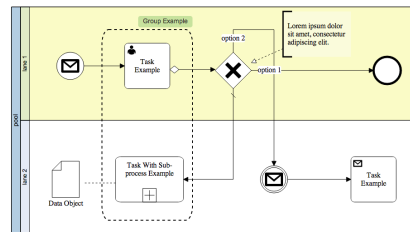
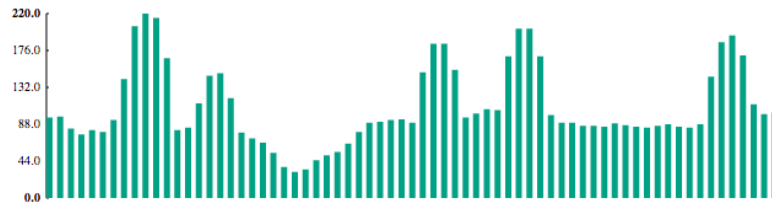

```

<link rel="stylesheet" type="text/css" href="dist/joint.css">
<script src="lib/jquery/jquery.min.js"></script>
<script src="lib/lodash/dist/lodash.min.js"></script>
<script src="lib/backbone/backbone-min.js"></script>
<script src="dist/joint.js"></script>

// And now, for example, only the ui.Stencil component:
<link href="dist/joint.ui.stencil.css" rel="stylesheet" type="text/css">
<script src="dist/joint.ui.stencil.js"></script>

```





JointJS+ Dijkstra

alg.Dijkstra

`alg.Dijkstra` is an implementation of the Dijkstra's shortest path algorithm. It can efficiently find the shortest path in both directed and undirected graphs. This implementation uses a binary heap based priority queue. The **time complexity** of this algorithm is $O(|E| + |V| \log |V|)$, where $|E|$ is the number of edges in the graph and $|V|$ is the number of nodes.

Install

Include both `joint.alg.dijkstra.js` and its priority queue dependency files into your HTML:

```
<script src="joint.alg.priorityQueue.js"></script>
<script src="joint.alg.dijkstra.js"></script>
```

Usage

Even though `alg.Dijkstra` is listed as a separate plugin, you usually don't use it directly. Instead, you can use the graphUtils plugin which extends the `joint.dia.Graph` with a convenience method for finding the shortest path between two nodes `shortestPath(source, target [, opt])`. However, it could be useful to use `alg.Dijkstra` directly in some situations. The reason is that `alg.Dijkstra` calculates not only the shortest path between two nodes but the shortest path between a node and *all* the other nodes in the graph.

`joint.alg.Dijkstra` is a function that takes a graph represented as an adjacency list, a source node and optionally a `weight` function. The adjacency list is an object where a key is a node ID and value is an array of IDs of the neighbouring nodes of that node. If you want to use `alg.Dijkstra` directly on a JointJS graph, you first have to convert the graph into the adjacency list. (This is what the `shortestPath()` method from the `alg.GraphUtils` plugin does for you internally.) `weight` function is a function that takes two nodes and returns a distance between them. How you define distance between two nodes is completely on you. By default, the distance is always `1`. Only keep in mind that the Dijkstra's algorithm requires the weight to be a positive integer.

The function returns a special object that encodes the shortest paths between the node provided and all the other nodes in the graph.

```
var graph = {
  a: ['b', 'c'],
  b: ['d', 'e'],
  c: ['f', 'g'],
  f: ['b'],
  e: ['c'],
  h: ['f', 'g'],
  i: ['h', 'a', 'd', 'g'],
  j: ['a']
```

```
};

var previous = joint.alg.Dijkstra(graph, 'a');
// { b: "a", c: "a", e: "b", f: "c" }
```

The above example shows the result of the `joint.alg.Dijkstra` function. This special object allows us to get the shortest path to all the nodes in the graph starting at node `'a'`. For example, the shortest path to the node `'f'` is a path `['a', 'c', 'f']`. How did we get there? You simply start from your target node and keep asking the returned object node by node till you reach your source node. In other words, we start by `previous['f']` which gives us `'c'`. Then we ask for `previous['c']` which gives us `'a'` and we're done since we reached our source node.

API

`joint.alg.Dijkstra(adjacencyList, source [, weight])`

Find the shortest path between the node `source` and all the other nodes in the graph represented as `adjacencyList`. See the [Usage](#) section for more info. `weight` can optionally contain a function that takes two nodes and returns the distance between them. This function defaults to:

```
function(u, v) { return 1; }
```

JointJS+ PriorityQueue

alg.PriorityQueue

`alg.PriorityQueue` is an implementation of the Priority Queue abstract data type. It is like a normal stack or queue, but where each item has assigned a priority (a number). Items with higher priority are served before items with lower priority. This implementation uses binary heap as an internal representation of the queue. The time complexity of all the methods is as follows:

- `create`: $O(n)$
- `insert`: $O(\log n)$
- `peek`: $O(1)$
- `peekPriority`: $O(1)$
- `remove`: $O(\log n)$
- `isEmpty`: $O(1)$

`alg.PriorityQueue` is used internally by the alg.Dijkstra algorithm for finding the shortest path in a graph. It is however useful on its own, that's why it is listed as a separate plugin.

Install

Include `joint.alg.priorityQueue.js` file into your HTML:

```
<script src="joint.alg.priorityQueue.js"></script>
```

Usage

The usage of `alg.PriorityQueue` is pretty simple. You just have to create an object of the `joint.alg.PriorityQueue` type. Then you can insert, remove or retrieve elements from the queue similarly as you would do with a normal JavaScript array.

```
var q = new joint.alg.PriorityQueue;
q.insert(1, 'one');
q.insert(3, 'three');
q.insert(2, 'two');

q.peek() // the first value is 'one'
q.peekPriority() // the first priority is 1
q.remove() // the first value was 'one'
q.remove() // the second value was 'two'
q.remove() // the third value was 'three'
q.isEmpty() // true
```

Moreover, this implementation of the priority queue allows you to update priorities of any item that you have inserted into the queue. This is also known as the “ decreaseKey ” operation. For this to work, you have to give each item you insert into the queue a unique ID. This is because in JavaScript, objects do not have unique IDs by default. Therefore, there is no way the `alg.PriorityQueue` can know which item you'd like to update priority for. Here is an example of priority queue that uses the `updatePriority()` operation:

```
var q = new joint.alg.PriorityQueue;
q.insert(1, 'one', 'id1');
q.insert(3, 'three', 'id3');
q.insert(2, 'two', 'id2');

q.peek() // the first value is 'one'
q.updatePriority('id1', 5);
q.peek() // now the first value is 'two'
q.updatePriority('id1', 1);
q.peek() // the first value 'one' got again back to the top
```

API

joint.alg.PriorityQueue([opt])	Create a priority queue object. If <code>opt.data</code> array is passed, it must be an array of items of the form <code>{ priority: Number, value: Object }</code> . In this case, the priority queue will be initialized with this array. It's like calling <code>insert(priority, value)</code> for each item of this array. <code>opt.comparator</code> can optionally be a function that will be used to compare two priorities. The signature of this function is <code>function(a, b)</code> . The function should return a value less than <code>0</code> if priority <code>a</code> is lower than priority <code>b</code> , value equal to <code>0</code> if the priorities are the same and value bigger than <code>0</code> if priority <code>a</code> is higher than priority <code>b</code> . The comparator function defaults to: <code>function(a, b) { return a - b }</code> . This function effectively allows you to use any object as a priority in which case it is on you to tell the priority queue how to compare two priorities.
isEmpty()	Return <code>true</code> if the priority queue is empty, <code>false</code> otherwise.
insert(priority, value [, id])	Insert a value with priority to the queue. Optionally pass a unique <code>id</code> of this item. Passing unique IDs for each item you insert allows you to use the <code>updatePriority()</code> operation. See the Usage section for details .
peek()	Return the value of an item with the highest priority.
peekPriority()	Return the highest priority in the queue.
remove()	Return the value of an item with the highest priority and remove the item from the queue.

updatePriority(id, priority)

Update priority of an item identified by a unique `id` . You can only use this operation if all the items you inserted to the queue had a unique ID assigned. See the [Usage section for details](#).

JointJS+ CommandManager

dia.CommandManager

CommandManager keeps track of graph changes and allows you to **travel the history** of those changes back and forth. There is no limitation put into the number of levels one can **undo/redo**.

Installation

Include `joint.dia.command.js` to your HTML:

```
<script src="joint.dia.command.js"></script>
```

Creating CommandManager

```
var graph = new joint.dia.Graph;
var paper = new joint.dia.Paper({ el: $('#paper'), width: 500, height: 500, model:
graph });

var commandManager = new joint.dia.CommandManager({ graph: graph });

$('#undo-button').click(function() { commandManager.undo(); });
$('#redo-button').click(function() { commandManager.redo(); });
```

How does CommandManager work?

CommandManager listens to graph changes and when any cell's attribute is changed it stores the difference (command). CommandManager stores every change made on graph's cells into `undoStack` and every reverted change into `redoStack`. It allows you to revert the changes stored in those stacks.

CommandManager API

constructor

The `joint.dia.CommandManager` constructor takes up to three parameters.

graph

Is the graph the CommandManager listens to.

cmdBeforeAdd

A function evaluated before any command is added. If the function returns `false`, the command does not get stored. This way you can control which commands do not get registered for undo/redo.


```

var commandManager = new joint.dia.CommandManager({
  graph: graph,
  cmdBeforeAdd: function(cmdName, cell, graph, options) {
    options = options || {};
    return !options.ignoreCommandManager;
  }
});

// ...

// Note that the last argument to set() is an options object that gets passed to the
cmdBeforeAdd() function.
element.set({ foo: 'bar' }, { ignoreCommandManager: true });

```

cmdNameRegex

A regular expression specifying what cell's attributes the CommandManager listens to. Default regex is `/^(?:add|remove|change:\w+)\$/`.

revertOptionsList

An array of options property names that are meant to be passed in undo actions. It defaults to `['propertyPath']`.

```

commandManager = new joint.dia.CommandManager({
  revertOptionsList: ['myOptionAttribute1']
})
element.set('attribute', 'value', { myOptionAttribute1: 5, myOptionAttribute2: 6 });
commandManager.undo(); // -> calls element.set('attribute', 'prevValue', {
myOptionAttribute1: 5 });

```

Alternatively the `revertOptionList` can be defined as a function.

```

revertOptionsList: function(attrValue, attrName) {
  return attrName !== 'doNotPassMe'; /* pass over everything except `doNotPassMe`
attribute */
}

```

applyOptionsList

An array of options property names that are meant to be passed in redo actions. It defaults to `['propertyPath']`.

```

commandManager = new joint.dia.CommandManager({
  applyOptionsList: ['myOptionAttribute1']
})
element.set('attribute', 'value', { myOptionAttribute1: 5, myOptionAttribute2: 6 });

```

```
commandManager.undo();
commandManager.redo(); // -> calls element.set('attribute', 'value', {
myOptionAttribute1: 5 });
```

Alternatively the `applyOptionList` can be defined as a function.

```
applyOptionsList: function(attrValue, attrName) {
    return attrName !== 'doNotPassMe'; /* pass over everything except `doNotPassMe`
attribute */
}
```

stackLimit

The maximum number of commands in the stack. If the number of commands exceeds the `stackLimit`, the oldest commands in the `undoStack` are forgotten. It defaults to Infinity.

undo

Function `undo(opt)` travels the history one command back. It accepts an option parameter (`opt`) that is applied when function makes changes to the graph. e.g `commandManager.undo({ undoneBy: 'user123' })`

redo

Function `redo(opt)` go back to whatever was previously `undo()` ed.

cancel

Function `cancel(opt)` same as `undo()` but does not store the undo-ed command to the `redoStack`. Canceled command therefore cannot be redo-ed.

hasUndo

Function `hasUndo()` `true` if there is something in the `undoStack`.

hasRedo

Function `hasRedo()` `true` if there is something in the `redoStack`

squashUndo

The method `squashUndo([n])` merges `n` number of commands in the `undo` stack into a single batch command. If no `n` is provided all commands are merged.

squashRedo

The method `squashRedo([n])` merges `n` number of commands in the `redo` stack into a single batch command. If no `n` is provided all commands are merged.

reset

Function `reset()` clears both `undoStack` and `redoStack`.

toJSON

The method exports the current state of the undo and redo stack to JSON. The export depends on the current state of graph and should be always accompanied by the graph export.

```
const graphJSON = graph.toJSON();
const historyJSON = commandManager.toJSON();
// a user-defined method to store data
await saveToDB({ historyJSON, graphJSON });
```

fromJSON

The method `fromJSON(json, [opt])` imports an existing undo and redo stack.

```
// a user-defined method to retrieve data
const { historyJSON, graphJSON } = await loadFromDB();
graph.fromJSON(graphJSON);
commandManager.fromJSON(historyJSON);
```

initBatchCommand

Function `initBatchCommand()` gives you the ability to gather multiple changes into a single command. These commands could be revert with single `undo()` call.

From the moment the function is called every change made on graph is not stored into the `undoStack`. Changes are temporarily kept in the `CommandManager` until `storeBatchCommand()` is called.

storeBatchCommand

Calling function `storeBatchCommand()` tells the `CommandManager` to store all changes temporarily kept in the `undoStack`. In order to store changes you have to call this function as many times as `initBatchCommand()` had been called.

Events

The plugin fires the following events.

Event Name	Handler Signature	Description
<code>stack:undo</code>	<code>(commandUndone, opt)</code>	Triggered when a command was undone.
<code>stack:redo</code>	<code>(commandRedone, opt)</code>	Triggered when a command were redone.

<i>stack:push</i>	(commandPushedToUndoStack, opt)	Triggered when a command were added to the stack.
<i>stack:cancel</i>	(commandCanceled, opt)	Triggered when a command was canceled.
<i>stack:reset</i>	(opt)	Triggered when all commands were reset or loaded from JSON.
<i>stack</i>	(opt)	Triggered when any change was made to stacks.

```
commandManager.on('stack', function(opt) {
  if (this.hasUndo()) { /* do something */ }
  if (this.hasRedo()) { /* do something else */ }
});
```

JointJS+ Validator

dia.Validator

Validator runs a set of callbacks to determine if a command is valid. This is useful for checking if a certain action in your application does lead to an invalid state of the diagram (from your application specific perspective). You can not only cancel such a command but e.g. also give the user a hint that this action is not allowed).

Installation

Include `joint.dia.command.js` and `joint.dia.validator.js` to your HTML:

```
<script src="joint.dia.command.js"></script>
<script src="joint.dia.validator.js"></script>
```

Creating Validator

```
const graph = new joint.dia.Graph;
const paper = new joint.dia.Paper({ el: document.getElementById('paper'), width: 500,
height: 500, model: graph });

const commandManager = new joint.dia.CommandManager({ graph: graph });
const validator = new joint.dia.Validator({ commandManager: commandManager });
validator.validate('remove',
  function(err, command, next) {
    if (command.data.type === 'standard.Rectangle') return next('Rectangles
cannot be removed. ');
    return next();
  },
  function(err, command, next) {
    if (err) console.log(err);
    return next(err);
  }
);
```

(Those that know the great [ExpressJS](#) application framework might have recognized a similar API to what ExpressJS provides for registering URL route handlers.)

Validator API

constructor

The `joint.dia.Validator` constructor takes two parameters.

commandManager

The CommandManager the validator listens to.

cancelInvalid

Determine whether to cancel an invalid command or not. If set to `false`, only the `invalid` event is triggered.

The default is `true`.

validate

Function `validate(action [, callback]*)` registers callbacks for a given action.

The validator listens to commands added to the CommandManager and runs this set of callbacks registered for the action. If the last callback returns an error, the command is canceled (see `joint.dia.CommandManager.cancel()`). This behaviour can be suppressed by setting the `cancelInvalid` to `false` in options passed to the Validator constructor.

action

Action is the name of the event triggered on the graph (see [List of triggered events](#)). Multiple actions may be given separated by whitespaces.

```
validate("change:source change:target", function() { .. });
```

callback

```
callback(err, command, next)
```

The validation function. An arbitrary number of validation functions can be passed to `validate()`. Every callback has the following signature:

Where `err` is an error from the previous callback.

The `command` parameter is a record from the CommandManager stack. It holds information about an action in the graph.

See below for the structure of a `command`. This command says that a `standard.Rectangle` element has been moved by 50px down.

```
{
  "action": "change:position"
  "data": {
    "id": "e0552cf2-fb05-4444-a9eb-3636a4589d64", // id of the cell it was
    manipulated with
    "type": "standard.Rectangle", // type of the cell

    // change:attribute events always have these two attributes filled
```

```

    "previous": { "position": { "x": 50, "y": 50 }},
    "next": { "position": { "x": 50, "y": 100 }},
  },
  "batch": true, // is command part of a batch command?
  "options": {} // Backbone options that are passed through the listeners when passed
as the last argument to the Backbone Model set() method.
}

```

The add and remove actions have empty previous and next objects. They hold all the cell attributes in the attributes object so that the CommandManager is able to reconstruct the whole cell if it has to. See below for an example of such a command structure:

```

{
  "action": "add",
  "data": {
    "id": "28de715b-62a7-4130-a729-1bcf7dbb1f2b",
    "type": "standard.Circle",

    // empty attributes
    "previous": {},
    "next": {},

    // all cells attributes
    "attributes": {
      "type": "standard.Circle",
      "size": {
        "width": 50,
        "height": 30
      },
      "position": { "x": 1220, "y": 680 },
      "id": "28de715b-62a7-4130-a729-1bcf7dbb1f2b",
      "attrs": {
        "circle": {
          "width": 50,
          "height": 30
        }
      }
    }
  }
}

"batch": true,
"options": {
  "add": true,
  "merge": false,
  "remove": false
}
}

```

The `next` parameter is a function accepting one argument - an error passed to the next callback in a row.

Calling the `next()` function means going to the next callback in the order passed to the `validate()` method. If a call to the `next()` function is omitted, the validation for the specified action stops.

JointJS+ RecordScrollbar

elementTools.RecordScrollbar

The `RecordScrollbar` element tool renders a scrollbar thumb on a `standard.Record` shape. The scrollbar is not visible if the record's size is large enough to show all items.

It accepts the following arguments which can be passed as an object to the element tool constructor:

width	<i>number</i>	The width of the scrollbar thumb. The default value is 8.
margin	<i>number</i>	A gap between the scrollbar thumb and sides of the record's content. The default value is 2.
rightAlign	<i>boolean</i>	Determines whether the scrollbar should be shown on the right side of the record's content. It is aligned to the left by default.

Example:

```
record1.setScrollTop(0);
const scrollbar = new joint.elementTools.RecordScrollbar({ width: 8 });
record1.findView(paper).addTools(new joint.dia.ToolsView({
  tools: [scrollbar]
}));
```

JointJS+ SwimlaneBoundary

elementTools.SwimlaneBoundary

The `SwimlaneBoundary` element tool renders a rectangular border to show the bounding box of a [bpmn2.Pool](#) and [bpmn2.HeaderedPool](#) pool lane. It accepts the following arguments which can be passed as an object to the element tool constructor:

laneId	<i>string</i>	Specifies the lane group for which the tool should be rendered.
padding	<i>number object</i>	Determines whether the boundary area should be visually inflated and if so, by how much. Default is <code>10</code> ({ <code>left: 10</code> , <code>top: 10</code> , <code>right: 10</code> , <code>bottom: 10</code> }).

Example:

```
const boundaryTool = new joint.elementTools.SwimlaneBoundary({
  laneId: 'customLaneId',
  padding: 20
});
```

JointJS+ SwimlaneTransform

elementTools.SwimlaneTransform

The `SwimlaneTransform` element tool allows you to resize lanes in `bpmn2.Pool` or `bpmn2.HeaderedPool` shape, from the UI. It renders 4 handles around selected lane which you drag and use to resize lane in particular direction. It accepts a few additional arguments, which can be passed as an object to the element tool constructor:

laneld	<i>string</i>	Specifies the lane group for which the tool should be rendered.									
padding	<i>number object</i>	Specifies how much the boundary area of handles should be visually inflated. Default is <code>10</code> ({ <code>left: 10</code> , <code>top: 10</code> , <code>right: 10</code> , <code>bottom: 10</code> }).									
minSize	<i>number</i>	Specifies the minimal height of the lane after which the tool will not allow it to shrink further. Default is <code>30</code> .									
constraintsPadding	<i>number</i>	Specifies the padding around constraint points. Default is <code>10</code> .									
minSizeConstraints	<i>function</i>	Callback function that is expected to return an array of minimal constraint points (in the form: { <code>x: number</code>, <code>y: number</code> }), relative to the paper. Minimal constraint points are points that should always be kept inside selected lane when resizing. If this option is left undefined, the default <code>minSizeConstraints</code> function is used instead which creates those points automatically from embedded shapes. Callback is called each time a handle starts to be dragged, the arguments passed are: <table><tr><td>model</td><td><i>dia.Element</i></td><td>Pool shape that the tool was added to.</td></tr><tr><td>laneld</td><td><i>string</i></td><td>Id of the lane that the tool was initialized with.</td></tr><tr><td>handleSide</td><td><i>"left" "top" "right" "bottom"</i></td><td>Position of the dragged handle (relative to the lane).</td></tr></table>	model	<i>dia.Element</i>	Pool shape that the tool was added to.	laneld	<i>string</i>	Id of the lane that the tool was initialized with.	handleSide	<i>"left" "top" "right" "bottom"</i>	Position of the dragged handle (relative to the lane).
model	<i>dia.Element</i>	Pool shape that the tool was added to.									
laneld	<i>string</i>	Id of the lane that the tool was initialized with.									
handleSide	<i>"left" "top" "right" "bottom"</i>	Position of the dragged handle (relative to the lane).									
maxSizeConstraints	<i>function</i>	Callback function that is expected to return an array of maximum constraint points (in the form: { <code>x: number</code> , <code>y: number</code> }), relative to the paper. Maximum constraint points are points that should									

		always be kept outside selected lane when resizing. If this option is left undefined, the default <code>maxSizeConstraints</code> function is used instead which creates those points automatically from embedded shapes. Callback is called each time a handle starts to be dragged, the arguments passed are: <table><tr><td>model</td><td><i>dia.Element</i></td><td>Pool shape that the tool was added to.</td></tr><tr><td>laneId</td><td><i>string</i></td><td>Id of the lane that the tool was initialized with.</td></tr><tr><td>handleSide</td><td><i>"left" "top" "right" "bottom"</i></td><td>Position of the dragged handle (relative to the lane).</td></tr></table>	model	<i>dia.Element</i>	Pool shape that the tool was added to.	laneId	<i>string</i>	Id of the lane that the tool was initialized with.	handleSide	<i>"left" "top" "right" "bottom"</i>	Position of the dragged handle (relative to the lane).
model	<i>dia.Element</i>	Pool shape that the tool was added to.									
laneId	<i>string</i>	Id of the lane that the tool was initialized with.									
handleSide	<i>"left" "top" "right" "bottom"</i>	Position of the dragged handle (relative to the lane).									
stopPropagation	<i>boolean</i>	Should be events stopped from propagating to the paper? Default is true .									

Example:

```
const transformTool = new joint.elementTools.SwimlaneTransform({
  laneId: 'customLaneId',
  padding: 8,
  minSize: 50,
  focusOpacity: 0.5,
  constraintsPadding: 20,
});
```

Example with custom constraint points:

```
graph.on('change:parent', function(element, parentId) {
  if (parentId) {
    const pool = graph.getCell(parentId);
    const [laneId] = pool.getLanesFromPoint(element.getBBox().center());
    element.prop('laneId', laneId);
  } else {
    element.prop('laneId', null);
  }
});

const transformTool = new joint.elementTools.SwimlaneTransform({
  laneId: 'customLaneId',
  constraintsPadding: 30,
  minSizeConstraints: (model, laneId, handleSide) => {
```

```

const minPoints = [];
const embedCells = model.getEmbeddedCells();

if (embedCells.length === 0) return;

embedCells.forEach(cell => {
  if (cell.prop('laneId') === laneId) {
    const cellBBox = cell.getBBox();
    minPoints.push(cellBBox.topLeft());
    minPoints.push(cellBBox.topRight());
    minPoints.push(cellBBox.bottomLeft());
    minPoints.push(cellBBox.bottomRight());
  }
});

if (handleSide === 'left' || handleSide === 'right') {
  const embedsBBox = model.graph.getCellsBBox(embedCells);
  // add left most and right most points
  minPoints.push(embedsBBox.origin());
  minPoints.push(embedsBBox.corner());
}

return minPoints;
},
maxSizeConstraints: (model, laneId) => {
  const maxPoints = [];
  const embedCells = model.getEmbeddedCells();

  if (embedCells.length === 0) return;

  embedCells.forEach(cell => {
    if (cell.prop('laneId') !== laneId) {
      const cellBBox = cell.getBBox();
      maxPoints.push(cellBBox.topLeft());
      maxPoints.push(cellBBox.topRight());
      maxPoints.push(cellBBox.bottomLeft());
      maxPoints.push(cellBBox.bottomRight());
    }
  });

  return maxPoints;
}
});

```

JointJS+ GEXF

format.GEXF

This plugin imports a graph represented as an XML string in the [GEXF format](#) and makes it a JointJS graph.

The plugin contains only one function `joint.format.gexf.toCellsArray(xmlString, makeElement, makeLink)`, where:

argument	description
xmlString	an XML string representing a graph in GEXF format
makeElement(data)	a function that takes an object with <code>id</code> and <code>label</code> (and <code>width</code> , <code>height</code> , <code>x</code> , <code>y</code> , <code>z</code> , <code>color</code> , <code>shape</code> if present in the XML) properties and returns a JointJS element for this node.
makeLink(data)	a function that takes an object with <code>source</code> and <code>target</code> properties and returns a JointJS link for that edge.

Installation

Include `joint.format.gexf.js` file into your HTML:

```
<script src="joint.format.gexf.js"></script>
```

Usage

```
const xmlString = `<?xml version="1.0" encoding="UTF-8"?>
<gexf xmlns="http://www.gexf.net/1.2draft"
xmlns:viz="http://www.gexf.net/1.1draft/viz" version="1.2">
  <graph mode="static" defaultedgetype="directed">
    <nodes>
      <node id="0" label="Hello">
        <viz:position x="15.0" y="40.0" z="0.0"/>
        <viz:size value="50" />
        <viz:color r="239" g="173" b="66" a="0.6"/>
      </node>
      <node id="1" label="World">
        <viz:position x="15.0" y="140.0" z="0.0"/>
        <viz:size value="50" />
        <viz:color r="239" g="173" b="66" a="0.6"/>
      </node>
    </nodes>
  </graph>
</gexf>
```

```

        </node>
    </nodes>
    <edges>
        <edge id="0" source="0" target="1" />
    </edges>
</graph>
</gexf>`;

const cells = joint.format.gexf.toCellsArray(
    xmlString,
    function makeElement(data) {
        return new joint.shapes.standard.Rectangle({
            id: data.id,
            position: { x: data.x, y: data.y },
            size: { width: data.width, height: data.height },
            attrs: {
                label: { text: data.label },
                body: { fill: data.color }
            }
        });
    },
    function makeLink(data) {
        return new joint.shapes.standard.Link({
            source: { id: data.source },
            target: { id: data.target }
        });
    }
);

graph.resetCells(cells);

```

JointJS+ Print

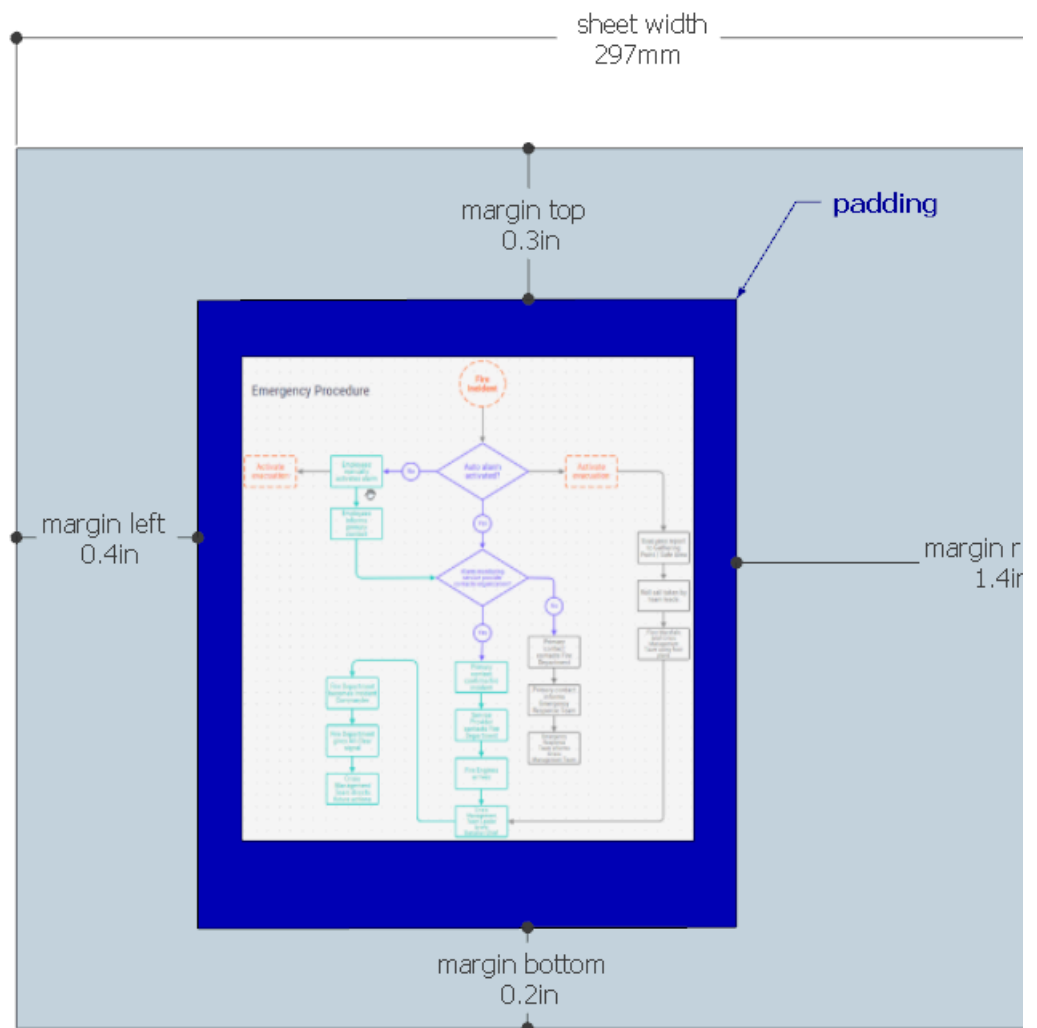
format.Print

The Print plugin prepares a paper for printing and initiates printing using the browser print wizard.

- `print(paper, [options])` - prints the paper using the browser print dialog box.

Where `paper` is an instance of `dia.Paper` and `options` is an object with the following possible values:

sheet	object	Optional. Along with the <code>sheetUnit</code>, <code>margin</code> and <code>marginUnit</code> it defines the output an object: { <code>width</code>: [number], <code>height</code>: [number] }. It defaults to size of the A4 page orientation - <i>landscape</i> or <i>portrait</i>. For example A4 - landscape would be set as { <code>w</code> Example: <pre>const printOptions = { sheet { width: 297, height: 210 }, // A4 - landscape sheetUnit: 'mm', margin: { left: 0.4, top: 0.3, bottom: 0.2, right: 1.4 }, marginUnit: 'in', padding: 20 }</pre>



Please note, the padding is ignored when the area is defined.

sheetUnit	<i>string</i>	Optional. Size unit for the <code>sheet</code> option. Supported units are <code>mm</code> , <code>cm</code> , <code>in</code> , <code>pt</code> , <code>pc</code> . It
area	<i>object</i>	Optional. Custom area to be printed. Defined as <code>{width: [number]: height: [numk [number]] }</code> . If the <code>area</code> is not defined, the whole graph is printed.
ready	<i>function</i>	Optional. This callback function is triggered when the print content is ready for the actual p update the print content element (e.g. set some additional css styles) and pass this throug print process. The prepared print content is a HTML element (multiple elements in case of t print process pass falsy value into the <code>readyToPrint</code> callback (see the example below). watermark or have a custom print preview UI widget.

```
// adjust the print content before it outputs to the printer
var options = {
  /**
   * @param {Array<jQuery>} pages
   * @param {function} readyToPrint
   * @param {{sheetSizePx: {width: number, height: number, css
string}}}} opt
```

		<pre> */ ready: function(pages, send, opt) { pages.forEach(function(\$page) { // custom action - e.g. remove border on every page: \$page.find('#paper').css('border', 'none'); }); // pass `false` into the readyToPrint to stop printing send(pages); }, } </pre> • <code>printAreaElements</code> - array of ready-to-print elements. • <code>readyToPrint</code> - pass <code>printAreaElements</code> into this callback to start printing process. • <code>opt</code> - additional useful options from the print plugin. <pre> { // sheet size convert to pixels sheetSizePx: { width: number, height: number, cssWidth: string, cssHeight: string } } </pre>
poster	boolean object	Optional. Splits the area to be printed to pieces. Each piece will be printed on a separate page. If <code>rows</code> / <code>columns</code> is a "table-style". It treats the print area to be a table and it divides it to desired width/height variant allows you to define the <code>width</code> and <code>height</code> of the piece. It default
margin	number object	Optional. The space on all sides of the paper area to be printed. The <code>margin</code> could be also an object with properties for finer grained control of the margin on each side: { <code>left</code> : [number], <code>top</code> : [number], <code>bottom</code> : [number] }
marginUnit	string	Optional. Size unit for the <code>margin</code> option. Supported units are <code>mm</code> , <code>cm</code> , <code>in</code> , <code>pt</code> , <code>pc</code> .
padding	number object	Optional. This is applicable only if the option <code>area</code> is not set - when the whole graph is printed. The <code>padding</code> could be also an object with the following properties: <code>padding</code> on each side: { <code>left</code> : [number], <code>top</code> : [number], <code>right</code> : [number],

Install

Include `print.css` and `joint.format.print.js` files into your HTML:

```
<link rel="stylesheet" type="text/css" href="print.css">
<script src="joint.format.print.js"></script>
```

Usage

```
document.getElementById('print-btn').addEventListener('click', function() {

    joint.format.print(paper, {
        padding: 10,
        sheet: { width: 297, height: 210 },
        poster: false,
        margin: 1,
        marginUnit: 'in',
        ready: function(printAreaElements, readyToPrint) {
            printAreaElements.forEach(function($el) {
                $el.find('#paper').css('border', 'none');
            });
            readyToPrint(printAreaElements)
        }
    });

}, false);
```

Demo

Adding a custom logo

If you'd like to add your custom text or logo to all the printed papers, you can use the following method in your CSS:

```
@media print {
    body:before {
        content: 'My Cool Company';
        top: -5mm;
    }
}
```

This adds the text *My Cool Company* to the top of the printed document. For adding a logo image, just set `width`, `height` and `background-image` properties on the `:before` pseudo-element.

JointJS+ Raster

format.Raster

This plugin adds new methods to the `joint.dia.Paper` object for exporting paper to PNG and JPEG raster formats on the client side:

toPNG(callback[, options])	Executes the callback function with a PNG <u>data URI</u> representing the content of the paper as the first parameter. The second parameter is an optional error in the case something happened during the processing.
toJPEG(callback[, options])	Executes the callback function with a JPEG data URI string representing the content of the paper as the first parameter. The second parameter is an optional error in the case something happened during the processing.
toDataURL(callback[, options])	The general version of two previous methods. Executes the callback function with a data URI string representing the content of the paper in specified MIME type as the first parameter. The second parameter is an optional error in the case something happened during the processing.
toCanvas(callback[, options])	Executes the callback function with a HTMLCanvasElement of applicable parameters from the paper as the first parameter. The second parameter is an optional error in the case something happened during the processing.

The available options are:

width	<i>number</i>	Set the width of the image in pixels.
height	<i>number</i>	Set the height of the image in pixels.
size	<i>number</i>	Size multiplier for generating raster images in a higher resolution. It expects a string in the form of <code>[Number] + 'x'</code> describing how many times the image width and height should be multiplied. It defaults to <code>'1x'</code> .
backgroundColor	<i>string</i>	The image background color.
padding	<i>number</i> <i>object</i>	The space between the image border and the image content. It can also be an object with <code>left</code> , <code>top</code> , <code>right</code> and <code>bottom</code> properties.

type	<i>string</i>	The standard <u>MIME type</u> for the image format to return (only applies to <code>toDataURL()</code>).
quality	<i>number</i>	The quality level of a JPEG image compression in the range of 0.0 to 1.0 (only applies to <code>toJPEG()</code>).
area	<i>g.Rect</i>	An area which should be displayed in the resulting raster image. The value is an object with <code>x</code>, <code>y</code>, <code>width</code> and <code>height</code> properties, describing the origin and size of a rectangular area on the paper in the local coordinates. It defaults to the paper content bounding box - <code>paper.getContentBBox()</code> transformed into the local coordinates. This is the same behaviour as <u>area</u> in <code>paper.prototype.toSVG()</code>. <pre>// Export selected elements only and add a padding var padding = 20; paper.toPNG(callback, { area: graph.getCellsBBox(selection.collection.toArray()).inflate(padding) });</pre>
useComputedStyles	<i>boolean</i>	The same behaviour as <u>useComputedStyles</u> in <code>paper.prototype.toSVG()</code> .
stylesheet	<i>string</i>	The same behaviour as <u>stylesheet</u> in <code>paper.prototype.toSVG()</code> .
beforeSerialize(SVGDocument, paper)	<i>function</i>	The same behaviour as <u>beforeSerialize</u> in <code>paper.prototype.toSVG()</code> .
fillFormControls	<i>boolean</i>	The same behaviour as <u>fillFormControls</u> in <code>paper.prototype.toSVG()</code> .

Note an exception `'dia.Paper: raster size exceeded'` can be thrown if the raster size reaches the browser limits. In that case the raster size / resolution can be lowered through the `size` option e.g `{ size: '.5x' }`.

Installation

Include `joint.format.svg.js` and `joint.format.raster.js` file into your HTML:

```
<script src="joint.format.raster.js"></script>
```

(Note that this plugin requires the SVG Export.)

To get the plugin to work across all modern browsers include also [canvg](#) library and its dependencies. All these files are part of the JointJS+ toolkit, in the Raster/lib directory:

```
<script src="format/Raster/lib/rgbcolor.js"></script>
<script src="format/Raster/lib/StackBlur.js"></script>
<script src="format/Raster/lib/canvg.js"></script>
```

Usage

```
paper.toPNG(function(imageData) { sendToServer(imageData); });

paper.toJPEG(function(imageData) { offerForDownload(imageData); }, {
  width: 640,
  height: 320,
  quality: 0.7
});
```

JointJS+ SVG

format.SVG

This plugin adds two new methods to the `joint.dia.Paper` object:

- `toSVG(callback[, opt])` - Converts the content of the paper to an SVG string and calls the callback with the SVG string as the first parameter. The second parameter is an optional error in the case something happened during the processing.
- `openAsSVG()` - Opens a new window with SVG representing the content of the paper.

Installation

Include `joint.format.svg.js` file into your HTML:

```
<script src="joint.format.svg.js"></script>
```

Usage

```
paper.toSVG(function(svgString, error) {  
    sendToServer(svgString);  
    offerForDownload(svgString);  
});
```

Configuration

The `toSVG(callback[, opt])` method takes an option object `opt` with the following parameters:

preserveDimensions	<i>boolean object</i>	By default, the resulting SVG document has set width and height to 100%. If you'd like to have the dimensions to be set to the actual content width and height, set <code>preserveDimensions</code> to <code>true</code> . An object with <code>width</code> and <code>height</code> properties can be also used here if you need to define the export size explicitly.
area	<i>g.Rect</i>	An area which should be displayed in the resulting raster image. The value is an object with <code>x</code> , <code>y</code> , <code>width</code> and <code>height</code> properties, describing the origin and size of a rectangular area on the paper in the local coordinates. It defaults to the paper content bounding box - <code>paper.getContentBBox()</code> transformed into the local coordinates. <pre>// Export selected elements only and add a padding</pre>

		<pre> var padding = 20; paper.toSVG(callback, { area: graph.getCellsBBBox(selection.collection.toArray()).inflate(padding) }); </pre>
convertImagesToDataUris	<i>boolean</i>	Converts all contained images into Data URI format. It defaults to false.
useComputedStyles	<i>boolean</i>	When set to true all the styles from external stylesheets are copied to the resulting SVG export. Note this requires a lot of computations and it might significantly affect the export time. <pre> // External stylesheet.css .joint-link .connection { fill: none } .joint-element rect { stroke: red } </pre> When set to false no styles from external stylesheets are copied to the resulting SVG export. Using this option requires you to make sure all the elements and links styling is inlined. <pre> // Store SVG Attributes right on the DOM elements link.attr('.connection', { fill: 'none' }); element.attr('rect', { stroke: 'red' }); paper.toSVG(callback, { useComputedStyles: false }); </pre>
stylesheet	<i>string</i>	A stylesheet (as string) can be provided and appended to the resulting SVG export. <pre> paper.toSVG(callback, { stylesheet: ['.joint-link .connection { fill: none }', '.joint-element rect { stroke: red }'].join('') }); </pre> It defaults to true.

beforeSerialize(SVGDocument, paper)	<i>function</i>	A function called before the XML serialization. It may be used to modify the exported SVG before it is converted to a string. The function can also return a new SVGDocument. <pre> paper.toSVG(callback, { beforeSerialize: function(doc) { // remove all links from the exported XML Array.from(doc.querySelectorAll('.joint-link')).forEach(function(node) { node.parentNode.removeChild(node); }); } }) </pre>
fillFormControls	<i>boolean</i>	Fill in the values of all form controls (input, select, textarea) so that they are displayed in the export (<i>form controls can be part of the export if they are inserted via <u>foreignObject</u></i>). It defaults to true.

JointJS+ Visio

format.Visio

This plugin allows for import of a Microsoft Visio 2013 VSDX file and converting Visio Shapes to JointJS Cells, as well as exporting a JointJS Paper as a VSDX archive. Visio plugin is an independent module, and it supports both ES5 as well as newer, non-ES5 project environments.

ES5 installation guide

Include all required dependencies into your HTML:

```
<!-- JointJS+/JointJS dependencies: -->
<script src="lodash.js"></script>
<script src="jquery.js"></script>
<script src="backbone.js"></script>
<script src="rappid.js"></script>

<!-- plugin dependencies: -->
<script src="jszip.js"></script>
<script src="formula-parser.js"></script>
<script src="joint.format.visio.js"></script>
```

At this point plugin is ready to use. Following example imports a vsdx project using default converter:

```
joint.format.Visio.VisioArchive.fromURL('network.vsd').then(function(archive) {
  var page = archive.document.getPages()[0];
  var graph = new joint.dia.Graph();
  var paper = new joint.dia.Paper({
    el: document.getElementById('paper'),
    model: graph,
    interactive: false,
    width: page.width,
    height: page.height,
    sorting: joint.dia.Paper.sorting.APPROX,
    async: true,
    frozen: true
  });
  page.getContent().then(function(content) {
    var cells = content.toGraphCells();
    graph.resetCells(cells);
    paper.unfreeze();
  });
});
```

```
});  
});
```

Module import guide

For non ES5 projects there's no additional installation required, simply import from `@clientio/rappid-visio` plugin module.

```
import { VisioArchive, types, VisioPageContent } from '@clientio/rappid-visio';
```

Usage

Following documentation examples use a non-ES5 code.

- To import a VSDX file use `VisioArchive` and a chosen method, suited for the source type:
 - fromURL - using a local or remote vsdx file
 - fromBase64 - using a vsdx file in a form of a Base64 string
 - fromArrayBuffer - using a vsdx file in a form of an ArrayBuffer
- To export a JointJS paper as a VSDX archive use VisioPage.fromPaper method to update the archive, and then export it using VisioArchive.toVSDX.

Import (basic)

This basic example of how to import a Visio project and use it in JointJS application uses default import process.

As the default converter translates Visio Shapes as-is, the result might differ in JointJS in comparison to Visio.

```
// load a vsdx file and pick a page to work with  
const { document: vsdDocument } = await VisioArchive.fromURL('./project.vsd');  
const [page0] = vsdDocument.getPages();  
  
// load objects present on the VisioPage  
const pageContent = await page0.getContent();  
  
// use default converter to create JointJS Cells from Visio Shapes  
const cells = pageContent.toGraphCells();  
  
graph.resetCells(cells);  
paper.setDimensions(page0.width, page0.height);
```

For an example with mapping, which gives more control over how Visio Shapes are imported, see the [Visio/demo/bpmn-import](#).

Export (basic)

A basic export example, which uses default converter to translate JointJS Paper as-is.

Do mind, that results might differ between JointJS application and Visio (i.e. unmapped icons might not display properly).

```
// load Visio archive that will be used for export
const archive = await VisioArchive.fromURL('./project.vsd');
const { document: vsdDocument } = archive;

// pick a page that will be the target for the export
const [page0] = vsdDocument.getPages();

// update page in the archive with JointJS Paper data using the default converter
await page0.fromPaper(paper);

// download archive using chosen method
const blob = await archive.toVSDX({ type: 'blob' });
util.downloadBlob(blob, 'project.vsd');
```

format.Visio.config

Config contains globally used flags controlling certain functionalities of the plugin.

To modify the plugin behavior, import `config` and toggle contained flags.

evaluateFormulas	<i>boolean</i>	Boolean controlling if formulas are evaluated when looking up a VisioCell value. Defaults to false . Formula parser is currently in early beta. Some parsed values might be different in comparison to value calculated by Visio.

format.Visio.types.VisioCellName

An enum containing all cell names supported by Visio 2013. This is useful i.e. when adding new cells to `VisioShape`.

format.Visio.types.VisioRowType

An enum containing row types supported by Visio 2013. This is useful i.e. when creating own `Geometry` type section.

format.Visio.types.VisioSectionType

An enum containing section types supported by Visio 2013. This is useful when adding, modifying or removing a section.

format.Visio.types.VisioUnitType

An enum containing cell unit types supported by Visio 2013. When modifying or adding a cell, remember to set `units` for the `value` , otherwise a project exported and opened in Visio might not work properly. Visio uses `units` to properly evaluate the cell `value` .

format.Visio.util.fromPixels

`util.fromPixels(value, units)`

Convert a given `value` from pixels to the provided `units` . `units` should be a valid numeric type supported by Visio ([types.VisioUnitType](#)).

value	<i>number</i>	Numeric value that will be converted.
units	<i>VisioUnitType</i>	Unit type of the target value. Supported numeric unit types: <code>IN</code> , <code>IN_F</code> , <code>DL</code> , <code>DP</code> , <code>DT</code> , <code>PT</code> , <code>MM</code> , <code>CM</code>

```
const inches = vsdUtil.fromPixels(96, VisioUnitType.IN); // 1
```

format.Visio.util.isFontAvailable

`util.isFontAvailable(font)`

Return `true` if provided font is available for the application.

font	<i>string array</i>	Function accepts either a <code>string</code> name of a single font or an <code>array</code> of font names. If a string was provided, function will return <code>true</code> if the font is available in the browser. Otherwise it will return <code>false</code> . If an <code>array</code> of font names was provided, an <code>array</code> of <code>true/false</code> results will be returned in order of the provided collection of font names.

format.Visio.util.toPixels

`util.toPixels(value, units)`

Convert the given numeric `value` to pixels. Function takes `opt` object in which a `units` property can be defined. If no `units` is provided, `types.VisioUnitType.IN` is used by default.

value	<i>number</i>	Numeric value that will be converted.
units	<i>VisioUnitType</i>	Unit type of the source value. Defaults to <code>VisioUnitType.IN</code> . Supported numeric unit types: <code>IN</code> , <code>IN_F</code> , <code>DL</code> , <code>DP</code> , <code>DT</code> , <code>PT</code> , <code>MM</code> , <code>CM</code>

```
const px = vsdUtil.toPixels(1, VisioUnitType.IN); // 96
```

format.Visio.VisioArchive

At the core of the plugin lies `VisioArchive` class, which is a bridge between Microsoft Visio and JointJS. It's main purpose is to translate Visio vsdx archive into a `VisioDocument` instance.

A more complex example showcasing mapping to/from Visio Shapes can be found in the `plugins/demo` directory.

format.Visio.VisioArchive.fromArrayBuffer

`async fromArrayBuffer(buffer)`

Load a Visio vsdx archive from an `ArrayBuffer`. This method returns a Promise.

```
const url = './project.vsdx';
const contentType = 'application/vnd-ms-visio.drawing;charset=utf-8';

const file = await fetch(url, { headers: { 'Content-Type': contentType } });
const arrayBuffer = await file.arrayBuffer();

const archive = await VisioArchive.fromArrayBuffer(arrayBuffer);
```

format.Visio.VisioArchive.fromBase64

async `fromBase64`(base64)

Load a Visio vsdx archive from a `base64` string. This method returns a Promise.

format.Visio.VisioArchive.fromURL

async `fromURL`(url)

Load a Visio vsdx archive fetching it from a provided URL. This method returns a Promise.

```
const url = './project.vsdX';  
const archive = await VisioArchive.fromURL(url);
```

A blank project that can be used to test export process can be fetched from [JointJS resources](#) page.

format.Visio.VisioArchive.prototype.document

The `document` property holds an instance of [VisioDocument](#) of an imported vsdx archive.

format.Visio.VisioArchive.prototype.source

The `source` property holds the original archive used to create the `VisioArchive` instance. It's being stored as an `ArrayBuffer`.

format.Visio.VisioArchive.prototype.toVSDX

async `visioArchive.toVSDX`([opt])

Use data stored in the `VisioArchive` instance to generate a vsdx file. This method will return a promise resulting in a vsdx file in a chosen format.

The `toVSDX([opt])` method takes an object `opt` with the following parameters:

type	<i>string</i>	Format type in which the vsdx will be returned. By default <code>'blob'</code> type is used. Other accepted types are: <code>'base64'</code> , <code>'binarystring'</code> , <code>'array'</code> , <code>'uint8array'</code> , <code>'arraybuffer'</code>
compression	<i>string</i>	Compression method to use. Available options are <code>'STORE'</code> and <code>'DEFLATE'</code> . Defaults to <code>'STORE'</code> .
compressionOptions	<i>object</i>	Additional compression options, ignored when using <code>'STORE'</code> method. Only available option is compression <code>level</code> , which is a number between 1 (best speed) and 9 (best compression). <pre>const blob = await archive.toVSDX({ type: 'blob' compression: 'DEFLATE', compressionOptions: { level: 9 } });</pre>

Usage

```
const blob = await visioArchive.toVSDX({ type: 'blob' });
joint.util.downloadBlob(blob, 'project.vsd');
```

A basic example of export process using default converter can be found [in the plugin usage section](#).

format.Visio.VisioCell

`VisioCell` describes attributes of a single cell, and its API allows only for the lookup actions.

setCell method should be used to perform an update.

format.Visio.VisioCell.prototype.eval

`visioCell.eval()`

Return a calculated result based on `cell.value`, `units` and `formula`.

format.Visio.VisioCell.prototype.formula

Return a `formula` that the cell uses to calculate the `value`.

format.Visio.VisioCell.prototype.name

Return a cell name (one of [VisioCellName](#) enum values).

format.Visio.VisioCell.prototype.units

Return a unit type of the cell. Used by Visio and [visioCell.eval\(\)](#) to properly evaluate the value.

format.Visio.VisioCell.prototype.value

Return a raw cell value. Raw cell value is given in the `cell.units` and returned as a string.

format.Visio.VisioConnect

VisioConnect represents a connection between two Visio Shapes. It contains metadata describing source and target including which part of the Shape is used as an anchor.

An important concept to understand is that in Visio projects, a connection between two shapes consists of two entities: Connect and a Shape, represented by [VisioConnect](#) and [VisioShape](#) within the plugin. VisioConnect contains metadata about the connection. It also points to the shape that visually represents the connection, as well as source and target shapes.

format.Visio.VisioConnect.prototype.fromCell

String name of the `VisioCell` from which the connection originates.

format.Visio.VisioConnect.prototype.fromPart

Identifies the part of the Visio Shape from which a connection is made.

format.Visio.VisioConnect.prototype.fromSheet

INT id of the source `VisioShape` .

format.Visio.VisioConnect.prototype.getShape

```
visioConnect.getShape()
```

Return a VisioShape representing the connection itself.

format.Visio.VisioConnect.prototype.getSource

```
visioConnect.getSource()
```

Return a VisioShape representing the connection source shape. In case there is no source shape, method will return `null` .

format.Visio.VisioConnect.prototype.getTarget

`visioConnect.getTarget()`

Return a [VisioShape](#) representing the connection [target](#) shape. In case there is no source shape, method will return `null`.

format.Visio.VisioConnect.prototype.toCell

String name of the `VisioCell` to which the connection is made.

format.Visio.VisioConnect.prototype.toLinkAttributes

`visioConnect.toLinkAttributes(sourceElement, targetElement)`

Return JointJS `Link` `attributes`, based on the `VisioConnect` properties and a provided `sourceElement` and `targetElement`.

format.Visio.VisioConnect.prototype.toPart

Identifies the [part](#) of the `VisioShape` to which a connection is made.

format.Visio.VisioConnect.prototype.toSheet

INT id of the target `VisioShape`.

format.Visio.VisioDocument

The top-level model in the Visio archive structure, containing all the project file data.

`VisioDocument` is a starting point for both importing a Visio vsdx project, as well as exporting a JointJS project to a vsdx file. When importing a vsdx project, `VisioDocument` is used to navigate pages, page contents, master shapes etc. In export process, `VisioDocument` is helpful in instantiating `VisioShapes` based on `VisioMaster` shapes and finally building a vsdx file from the document contents.

format.Visio.VisioDocument.prototype.getMasters

visioDocument.[getMasters\(\)](#)

Return an array of [VisioMaster](#) instances present in the document.

format.Visio.VisioDocument.prototype.getMastersIdMap

visioDocument.[getMastersIdMap\(\)](#)

Return a Map of [VisioMaster](#) instances present in the document, using master `id` as the key.

format.Visio.VisioDocument.prototype.getMastersNameMap

visioDocument.[getMastersMameMap\(\)](#)

Return a Map of [VisioMaster](#) instances present in the document, using master `name` as the key.

format.Visio.VisioDocument.prototype.getPage

visioDocument.[getPage](#)(pageId)

Return a [VisioPage](#) metadata object describing a single page with the given `pageId` . If page with given id doesn't exist, method will return [undefined](#) .

Metadata does not contain page contents. To load contents, use [page.getContent\(\)](#) method.

pageId	<i>int</i>	A unique integer page id number.

Examples

To fetch metadata of a particular page do:

```
const archive = await VisioArchive.fromURL('./assets/vsdX/archive.vsdX');
const { document: vsdDocument } = archive;
```

```
const page1 = vsdDocument.getPage(0);
```

To fetch a page that's used as a background for a particular page, simply do:

```
const archive = await VisioArchive.fromURL('./assets/vsdX/archive.vsdX');  
const { document: vsdDocument } = archive;  
  
const page1 = vsdDocument.getPage(0);  
const bgPage = vsdDocument.getPage(page1.backPage);
```

format.Visio.VisioDocument.prototype.getPages

`visioDocument.getPages()`

Return an array of [VisioPage](#) metadata objects present in the loaded vsdx document. Metadata does not contain page contents. To load contents of a particular page, use [page.getContent\(\)](#) method.

`getPages()` can be useful i.e. for building project page listing.

format.Visio.VisioGeometrySection

`VisioGeometrySection` inherits from [VisioIndexedSection](#). Indexed section structures hold ordered rows collection.

format.Visio.VisioGeometrySection.prototype.addRow

`visioGeometrySection.addRow(geometryRowType)`

Add a named `VisioRow` to a geometry section rows collection. Provided `geometryRowType` must be one of `VisioRowType` enum values. If no `geometryRowType` is provided, a row of type `VisioRowType.RelMoveTo` will be added in case it's the first row, or `VisioRowType.RelLineTo` for any consecutive rows.

Usage

```
const [geometry0] = vsdShape.getGeometry();  
geometry0.addRow(VisioRowType.MoveTo);
```

format.Visio.VisioIndexedSection

`VisioIndexedSection` inherits from `VisioSection`. Indexed section structures order rows using an integer index. Order of rows is important.

Following sections use indexed row structure: Character Connection Field FillGradient Geometry Layer LineGradient Paragraph Reviewer Scratch Tabs

format.Visio.VisioIndexedSection.prototype.addRow

`visioIndexedSection.addRow()`

Appends a `VisioRow` to an indexed section rows collection.

Usage

```
const scratch = vsdShape.getSection(VisioSectionType.Scratch);
scratch.addRow(); // add a row with next available index
```

format.Visio.VisioIndexedSection.prototype.getRow

`visioIndexedSection.getRow(index)`

Return a `VisioRow` with a given `index` in a section with indexed structure. If row doesn't exist, return `null`.

format.Visio.VisioIndexedSection.prototype.removeRow

`visioIndexedSection.removeRow(index)`

Remove a row with the given `index` from an indexed section.

format.Visio.VisioMaster

`VisioMaster` contains metadata of a Visio stencil item.

format.Visio.VisioMaster.prototype.getIconBase64

`visioMaster.getIconBase64()`

Return a `base64` data string containing a stencil item icon.

format.Visio.VisioMaster.prototype.id

Return a unique, integer master `id`.

format.Visio.VisioMaster.prototype.name

Return a `name` that by design in Visio is unique. Multiple instances of a same name in a stencil receive an additional ID appended to the name (i.e. `"Master.12"`).

format.Visio.VisioNamedSection

`VisioIndexedSection` inherits from `VisioSection`. Named section structures hold unordered rows collection. Row names have to be unique.

Following sections use named row structure: `Actions` `Control` `Hyperlink` `Property` `ActionTag` `User`

format.Visio.VisioNamedSection.prototype.addRow

`visioNamedSection.addRow(rowName)`

Add a `VisioRow` to a named section rows collection. Can not add a row with a given `rowName` if a row with such name already exists.

Usage

```
const properties = vsdShape.getSection(VisioSectionType.Property);
properties.addRow('NewPropertyRow');
```

format.Visio.VisioNamedSection.prototype.getRow

`visioNamedSection.getRow(rowName)`

Return a `VisioRow` in a named section with a given `rowName` . If row doesn't exist, return `null` .

format.Visio.VisioNamedSection.prototype.removeRow

`visioNamedSection.removeRow(rowName)`

Remove a row with the given `rowName` from a named section.

format.Visio.VisioPage

VisioPage contains metadata describing a page in a document. Its API allows to load contents of the page, as well as update the page data with JointJS Paper when exporting a project to vsdx format.

format.Visio.VisioPage.prototype.background

Return `true` if the page is marked as a background page. Background pages are oftentimes used to show repetitive content on many foreground pages.

format.Visio.VisioPage.prototype.backPage

If the page uses a background page, return its `id`

format.Visio.VisioPage.prototype.fromPaper

`async visioPage.fromPaper(paper [,opt])`

`page.fromPaper()` method is used when exporting a JointJS project to a vsdx file format. Before the vsdx archive is built, its pages should be updated with new content. This new content is built using JointJS paper.

Example

The below example updates visio archive with the contents of JointJS paper and uses default converter to create a `VisioShape` for each JointJS Cell, as well as update project configuration. Lastly, the updated archive is exported as a vsdx file.

```
const archive = await VisioArchive.fromURL('./project.vsd');
const { document: vsdDocument } = archive;
const [page1] = vsdDocument.getPages();

await page1.fromPaper(paper);
const blob = await archive.toVSDX({ type: 'blob' });

util.downloadBlob(blob, 'project.vsd');
```

For a more robust example on the **export flow** with mapping, view [demos/bpmn-export](#) demo files.

Options

The `fromPaper(paper, [opt])` method takes JointJS paper, and an object `opt` with the following parameters:

exportElement	function
	An <code>async</code> function run for each <code>dia.Element</code> present on the provided <code>paper</code>. The function is passed <code>elementView</code> and expects either an instance of <code>VisioShape</code> to be returned or a <code>null</code> if element is to be skipped in export process. <pre>exportElement: async(elementView) => { const { model, paper } = elementView; let master; // perform relevant logic to find chosen master shape // ... // create a raw shape based on a stencil Master const vsdShape = await VisioShape.fromMaster(master, vsdDocument);</pre>

		<pre> // perform vsdShape manipulations based on the given elementView // ... // return modified vsdShape return vsdShape; } </pre>
exportLink	function	An <code>async</code> function run for each <code>dia.Link</code> present on the provided <code>paper</code>. The function is passed <code>linkView</code> and expects either an instance of <code>VisioShape</code> to be returned or a <code>null</code> if links is to be skipped in export process. Connect objects required by Visio are created under the hood. The <code>VisioShape</code> returned by this function is the visual representation of connection in Visio. <pre> exportLink: async(linkView) => { let master; // perform relevant logic to find chosen master shape // ... return await VisioShape.fromMaster(master, vsdDocument); } </pre>

format.Visio.VisioPage.prototype.getContent

`async visioPage.getContent()`

Return a `VisioPageContent` instance containing page shapes.

format.Visio.VisioPage.prototype.height

Return page `height` in pixels.

format.Visio.VisioPage.prototype.id

Return a unique, integer page `id` .

format.Visio.VisioPage.prototype.name

Return page name as configured in Visio project.

format.Visio.VisioPage.prototype.nameU

Return page universal name as configured in Visio project.

format.Visio.VisioPage.prototype.width

Return page `height` in pixels.

format.Visio.VisioPageContent

`VisioPageContent` gives access to all instances of `VisioShapes` and `VisioConnects` present on the given page , as well exposes an API useful in converting page contents to JointJS Cells using `pageContent.toGraphCells([,opt])` method.

To work with page content, modifying shapes etc., loading `VisioPageContent` is always the first step. Provided API allows fetching any kind of entity that is part of the page content.

format.Visio.VisioPageContent.prototype.getConnect

`visioPageContent.getConnect(shapeId)`

Return a `VisioConnect` associated with the provided `shapeId` . If no such shape exists, return `undefined` .

As `VisioConnect` contains connection metadata, it always points to the actual `VisioShape` that is a visual representation of the connection.

format.Visio.VisioPageContent.prototype.getConnects

```
visioPageContent.getConnects()
```

Return all `VisioConnects` present in the page content.

format.Visio.VisioPageContent.prototype.getElementShapes

```
visioPageContent.getElementShapes()
```

Return all `VisioShape` 's that would be identified as `Element` in JointJS.

format.Visio.VisioPageContent.prototype.getFonts

```
visioPageContent.getFonts()
```

Return an `array` of fonts used by shapes on the page. If no fonts are defined, return an empty `array` .

format.Visio.VisioPageContent.prototype.getForeignShape

```
visioPageContent.getForeignShape(path)
```

Return `VisioShape` considered foreign (i.e. image), with the given `path` within the `VisioArchive` document. If there is no shape for the given path, method will return `undefined`

format.Visio.VisioPageContent.prototype.getForeignShapes

```
visioPageContent.getForeignShapes()
```

Return all `VisioShape` considered foreign (i.e. images) from the page content.

format.Visio.VisioPageContent.prototype.getLinkShapes

`visioPageContent.getLinkShapes()`

Return all `VisioShape` 's that would be identified as `Link` in JointJS.

format.Visio.VisioPageContent.prototype.getRootShape

`visioPageContent.getRootShape(id)`

Return a top-level `VisioShape` with the given `id` from the page content. Return `undefined` if shape does not exist or is not a root shape.

format.Visio.VisioPageContent.prototype.getRootShapes

`visioPageContent.getRootShapes()`

Return all top-level `VisioShape` instances from the page content. Omit all the sub-shapes.

format.Visio.VisioPageContent.prototype.getShape

`visioPageContent.getShape(id)`

Return a `VisioShape` with the given `id`.

format.Visio.VisioPageContent.prototype.getShapes

`visioPageContent.getShapes()`

Return all `VisioShape` instances from the page content.

format.Visio.VisioPageContent.prototype.getUnsupportedFonts

visioPageContent.[getUnsupportedFonts](#)()

Return an `array` of fonts that are used by shapes on the page, however are not supported by the application. If all fonts are supported, return an empty `array` .

format.Visio.VisioPageContent.prototype.toGraphCells

visioPageContent.[toGraphCells](#)([, opt])

Method used to convert contents of a Visio project page to JointJS Graph `Cells` . Its main usage is in the import process of a vsdx file into a JointJS application.

Options

The [toGraphCells](#)([, opt]) method takes an object `opt` with the following parameters:

ignoreNonPrinting	<i>boolean</i>	Control if shapes marked in Visio project as <code>nonPrinting</code> should be taken into account during import or not. Defaults to <code>true</code> .
ignoreSubShapeConnects	<i>boolean</i>	Control if connections originating from or targeting nested shapes should be imported or not. Defaults to <code>true</code> .
importShape	<i>function</i>	For each <pre>importShape: (vsdShape) => { let element; // .. perform logic creating an Element // using the vsdShape as the source of information return element; }</pre>

importConnect	function	<pre> importConnect: (vsdConnect, sourceElement, targetElement) => { let link; // .. perform logic creating a Link from vsdConnect return link; } </pre>
importLabels	function	If VisioShape identified as a connection contains a text, it is considered a label for that connection. If this function is not used, the default converter will be used to create link labels. Labels can be handled using <code>importLabels</code> function, which is passed <code>vsdShape</code> and <code>link</code>, and expects an array of label <code>dia.Elements</code> to be returned. If no new elements are being created for the links (i.e. we append labels to link directly), the function does not have to return anything. <pre> importLabels: (vsdShape, link) => { const data = vsdShape.getSection(VisioSectionType.Property); const name = data.getProperty('BpmnName'); if (name) { link.appendLabel({ attrs: { body: { fill: '#F4F7F6' }, label: { text: name, fontWeight: 'bold', fontSize: 10 } }, position: 0.5 }); } } </pre>

importImage	<i>function</i>	Foreign shapes, i.e. images, can be handled separately with <code>importImage</code> function. The function is passed <code>vsdShape</code> , <code>element</code> , <code>vsdImage</code> . Function can be used i.e. to set <code>xlinkHref</code> attribute on an element. <pre>importImage: (vsdShape, element, image) => { element.attr('image/xlinkHref', image.base64); }</pre>
onImagesLoad	<i>callback</i>	Although <code>toGraphCells</code> is run synchronously, images take time to load. To catch the moment where all images are ready, <code>onImageLoad</code> callback can be used. Callback function is passed the images collection. <pre>onImagesLoad: (images) => { // i.e. finish preloading }</pre>

A full example of usage can be found in `demo/bpmn-import` directory.

format.Visio.VisioRow

`VisioRow` inherits from `VisioSheetObject` abstract class, exposing an API used in cells manipulation: `getOwnCellNames` , `getCell` , `setCell` and `removeCell` .

`VisioRow` is contained within a `VisioSection`. Each row contains cells describing a `VisioSection` that the row is contained in.

Available cells are defined by a section type row. More information can be found [in the Visio documentation](#).

format.Visio.VisioRow.prototype.index

If the row is part of a indexed `VisioSection`, returns an integer `index` of the row.

format.Visio.VisioRow.prototype.name

If the row is part of a named VisioSection, returns a `name` of the row.

format.Visio.VisioRow.prototype.type

Return `type` of the row or `undefined` if there is no type set.

As an example, `type` is used by Geometry section rows to indicate what kind if drawing action it describes, i.e. `'LineTo'`.

format.Visio.VisioSection

`VisioShapes` and `VisioRows` contains `VisioSections` that describe them using cells.

`VisioSection` is an abstract class that `VisioNamedSection` and `VisioIndexedSection` inherit from. It provides an API to look-up `VisioRows`. Classes that inherit from it expose API to fetch and modify rows.

Section row structures

<u>Named</u>	Named structures identify rows by a <code>name</code> that is unique for given section. <code>Actions</code> <code>Control</code> <code>Hyperlink</code> <code>Property</code> <code>ActionTag</code> <code>User</code>
<u>Indexed</u>	Indexed structures order rows using an integer index. Order of indexed rows is important. <code>Character</code> <code>Connection</code> <code>Field</code> <code>FillGradient</code> <code>Geometry</code> <code>Layer</code> <code>LineGradient</code> <code>Paragraph</code> <code>Reviewer</code> <code>Scratch</code> <code>Tabs</code>

format.Visio.VisioSection.prototype.getRows

`visioSection.getRows()`

Return an `array` of `VisioRows` the section contains. If none exist, return an empty `array`.

format.Vio.VioSection.prototype.type

Return `type` of the section. Each section type has one of the two row structures, either `'Indexed'` or `'Named'` .

format.Vio.VioShape

`VioShape` inherits from `VioSheetObject` abstract class, exposing an API used in cells manipulation:
`getOwnCellNames` , `getCell` , `setCell` and `removeCell` .

format.Vio.VioShape.fromMaster

`vioShape.fromMaster(master, page)`

Return a new `VioShape` created from a provided `master` and on a provided `page` .

Example below shows typical case when loaded archive is used as an export target, so it's used both as a source of masters and the target for the created `VioShapes` .

```
const archive = await VioArchive.fromURL('./project.vsd');
const { document: vsdDocument } = archive;
const [page0] = vsdDocument.getPages();

// pick a stencil master
const masters = vsdDocument.getMastersNameMap();
const [master0] = masters['Master Name'];

// create a raw shape based on a stencil Master
const vsdShape = await VioShape.fromMaster(master, page0);
```

format.Vio.VioShape.prototype.addSection

`vioShape.addSection(sectionType)`

Add a new `VioSection` (either `VioIndexedSection` or `VioNamedSection`) based on the given `sectionType` . In case it is a named section that already exists, new one will not be added. New indexed section will be appended at the end of the given collection with an index incremented from the previously highest index.

```
// add a custom properties section  
vsdShape.addSection(types.VisioSectionType.Property);
```

format.Visio.VisioShape.prototype.getAncestorShapes

visioShape.getAncestorShapes()

Return an array of ancestor VisioShapes of the shape, ordered starting with highest level.

format.Visio.VisioShape.prototype.getComputedGeometry

visioShape.getComputedGeometry()

Return an array of sections merged from **own** and **inherited** geometries. If there is no geometry sections defined on either shape or the master, return an empty array.

format.Visio.VisioShape.prototype.getComputedSection

visioShape.getComputedSection(type)

Return a section merged from **own** and **inherited** from master section data. If section of the `type` does not exist on either, return `null`.

If the given `type` is `VisioSectionType.Geometry`, return results of `visioShape.getComputedGeometry()`.

format.Visio.VisioShape.prototype.getConnect

visioShape.getConnect()

Return a `VisioConnect` of the shape. If shape does not have a connect pointing to it, return `null`.

format.Visio.VisioShape.prototype.getGeometry

`visioShape.getGeometry()`

Return an `array` shapes **own** geometry sections. If none is defined, return an empty `array` .

format.Visio.VisioShape.prototype.getImage

async `visioShape.getImage()`

Return an `image` object containing following properties:

absolutePath	string	File path within the visio archive.
file	string	File name of the image.
extension	string	File extension of the image.
base64	string	A <code>base64</code> representation of the image.
selector	string	Image element selector.

format.Visio.VisioShape.prototype.getMaster

`visioShape.getMaster()`

Return a `VisioMaster` for a shape. Return `null` if the shape was not created from a stencil master.

format.Visio.VisioShape.prototype.getOwnSectionNames

`visioShape.getOwnSectionNames()`

Return an `array` of section names defined for the shape.

format.Visio.VisioShape.prototype.getPageAngle

`visioShape.getPageAngle()`

Return a `number` describing shape angle to the page.

format.Visio.VisioShape.prototype.getPageBBox

`visioShape.getPageBBox()`

Return a `g.Rect` describing the shape bbox on the page.

format.Visio.VisioShape.prototype.getPagePosition

`visioShape.getPagePosition()`

Return a `g.Point` describing position of the shape on the page.

format.Visio.VisioShape.prototype.getPageZIndex

`visioShape.getPageZIndex()`

Return an `int` describing the shape z-index on the page.

format.Visio.VisioShape.prototype.getRootShape

`visioShape.getRootShape()`

Return the `VisioShape` that is a root shape for the given shape, or the `visioShape` itself if it is a root shape.

format.Visio.VisioShape.prototype.getSection

`visioShape.getSection(type)`

If an **own** section of `type` exists on the shape, return it. If a section does not exist on the shape, return `null`

Own sections are defined for the shape and set values override sections inherited from master shape.

If the given `type` is `VisioSectionType.Geometry`, return results of `visioShape.getGeometry()`.

format.VioShape.prototype.getSectionNames

`visioShape.getSectionNames()`

Return an `array` of all section names available for the shape, including inherited ones.

format.VioShape.prototype.getSubShapes

`visioShape.getSubShapes()`

Return an `array` of all shapes nested within the given `visioShape`.

format.VioShape.prototype.getText

`visioShape.getText()`

Return a `string` representing shape text if one exists. If none is defined, return empty `''`.

format.VioShape.prototype.hasImage

`visioShape.hasImage()`

Return `true` if the shape contains an image.

format.VioShape.prototype.id

A unique, integer shape `id`.

format.Visio.VisioShape.prototype.isOneDimensional

visioShape.**isOneDimensional**()

Return **true** if the shape is considered 1-D (one-dimensional) in Visio. Such shapes contain `beginX`, `beginY`, `endX`, `endY` cells. Return **false** if shape is considered 2-D.

format.Visio.VisioShape.prototype.masterId

`masterId` points to a master given shape was created from.

format.Visio.VisioShape.prototype.name

A string `name` of the shape.

format.Visio.VisioShape.prototype.removeSection

visioShape.**removeSection**(sectionName, sectionIndex)

If a named section is being removed, only the `sectionName` parameter is required. For indexed section, a second parameter, `sectionIndex` has to be provided. `sectionIndex` defaults to **0** if none is provided.

```
// remove property section
vsdShape.removeSection(types.VisioSectionType.Property);

// remove geometry section with index 2
vsdShape.removeSection(types.VisioSectionType.Geometry, 2);
```

format.Visio.VisioShape.prototype.setText

```
visioShape.setText(text)
```

Update shape text with given `text` . Do mind this will override the existing text.

format.Visio.VisioShape.prototype.toElementAttributes

```
visioShape.toElementAttributes()
```

Return JointJS `dia.Element` `attributes` created from the shape. It's particularly useful when using custom `Element` instead of the one used by the default converter.

format.Visio.VisioShape.prototype.type

A string `type` of the shape, i.e. `'Shape'` or `'Group'` .

format.Visio.VisioSheetObject

`VisioSheetObject` is an abstract class that allows for manipulation of cells within an object with sheet.

Classes inheriting from `VisioSheetObject` are `VisioShape` and `VisioRow`.

Any cells manipulation should be done **only** using the `VisioSheetObject` API.

Cell names

Cell names are case-sensitive and should follow camel-case naming convention, while the `name` property in the cell object uses pascal-case naming convention. Having that difference in mind, when using `getCell` method, always use camel-case names, otherwise `undefined` will be returned.

Cell structure

Each cell is an object with `name` , `value` properties. Optionally `formula` and `units` can be defined.

format.Visio.VisioSheetObject.prototype.getCell

```
sheetObject.getCell(cellName)
```


Return a cell object with the given `cellName` . If cell with given name does not exist, return `undefined` . `cellName` is case sensitive and should use camel-case naming convention.

Usage

```
console.log(vsdShape.getCell(VisioCellName.PinX));  
// {value: "1", units: "IN", name: "PinX"}
```

format.Visio.VisioSheetObject.prototype.getOwnCellNames

`sheetObject.getOwnCellNames()`

Return an `array` of cell names defined as own for the sheet object. Does not include inherited cell names.

format.Visio.VisioSheetObject.prototype.removeCell

`sheetObject.removeCell(cellName)`

Remove cell with the given `cellName` from the sheet object. `cellName` is case sensitive and should use camel-case naming convention.

```
vsdShape.removeCell(VisioCellName.PinX)
```

format.Visio.VisioSheetObject.prototype.setCell

`sheetObject.setCell(cellName, cellAttributes)`

`setCell` allows for adding or updating a cell within the sheet object. Important thing to note is that knowledge about Visio Cells is required to properly set cells in certain scenarios, i.e. `units` might be required to properly evaluate the `value` in Visio.

Inheriting and overriding attributes

Passing `null` as any of the `cellAttributes` will remove the attribute from the cell, which will in return fallback to inherited value. If the intention is to override any attribute, set it to an empty string, i.e. `{formula: ''}` . Not passing any of the `attributes` will skip them, and existing ones will be kept.

Parameters

cellName	<i>string</i>	cellName should use camel-case naming of cells. Full list of accepted cell names can be found in Visio documentation .		
cellAttributes	<i>object</i>	value	<i>string</i> <i>null</i>	value should be passed as strings (case-sensitive).
		formula	<i>string</i> <i>null</i>	formula are case-sensitive and should be passed as strings. Note, that exporting project to vsdx using visioArchive.toVSDX() will trigger re-evaluation of all formulas on first startup of the project in Visio.
		units	<i>string</i> <i>null</i>	units should be set as string and are case-sensitive. For the list of units refer to Visio.VisioUnitType .

Usage

Below code will update width cell (or create it if it doesn't exist), setting value to '1', units to 'IN' and inheriting the formula.

```
// shape cell
vsdShape.setCell(VisioCellName.Width, { value: '1', units: VisioUnitType.IN });

// property section row cell
vsdRow.setCell(VisioCellName.Value, { value: '1', formula: null, units:
VisioUnitType.BOOL });
```

JointJS+ graphUtils

graphUtils.intro

Installation

Include `joint.graphUtils.js` file to your HTML:

```
<script src="joint.graphUtils.js"></script>
```

If you plan to use `shortestPath()` method, you're going to also need to include `joint.alg.priorityQueue.js` and `joint.alg.dijkstra.js` plugins:

```
<script src="joint.alg.priorityQueue.js"></script>  
<script src="joint.alg.dijkstra.js"></script>
```

JointJS+ constructTree

graphUtils.constructTree

`constructTree`(parent, opt)

Construct a tree from a JSON object (parent , i.e. the top level node). This method returns an array of cells (elements and links) that constitute the tree. The parent parameter must be a JSON of the form:

```
{
  name: 'my label',
  children: [ { name: 'my label 2', children: [...] }, ...]
}
```

opt.children is the property specifying the children array (default is 'children'). If opt.children is a function, it will be called with the current node as an argument and must return an array of its child nodes.

opt.makeElement() is a function that is passed the current tree node and returns a JointJS element for it.

opt.makeLink() is a function that is passed a parent and child nodes and returns a JointJS link for the edge.

Example:

```
const cells = joint.graphUtils.constructTree(
  {
    name: 'my top',
    children: [{
      name: 'my child 1'
    }, {
      name: 'my child 2'
    }]
  },
  {
    makeElement: function(node) {
      return new joint.shapes.standard.Rectangle({
        size: { width: 80, height: 30 },
        attrs: { label: { text: node.name } }
      });
    },
    makeLink: function(parentElement, childElement) {
      return new joint.shapes.standard.Link({
        source: { id: parentElement.id },
        target: { id: childElement.id }
      });
    }
  }
)
```

```
);  
graph.addCell(cells);
```

Example of an adjacency list:

```
const adjacencyList = {  
  'a': ['b', 'c'],  
  'b': ['d', 'e'],  
  'c': [],  
  'd': ['f'],  
  'e': [],  
  'f': []  
};  
  
const cells = joint.graphUtils.constructTree(  
  'a', // root  
  {  
    makeElement: function(name) {  
      return new joint.shapes.standard.Rectangle({  
        size: { width: 80, height: 30 },  
        attrs: { label: { text: name } }  
      });  
    },  
    makeLink: function(parentElement, childElement) {  
      return new joint.shapes.standard.Link({  
        source: { id: parentElement.id },  
        target: { id: childElement.id }  
      });  
    },  
    children: (name) => adjacencyList[name]  
  }  
);  
graph.addCell(cells);
```

JointJS+ shortestPath

graphUtils.shortestPath

`shortestPath(graph, source, target [, opt])`

Return an array of IDs of nodes on the shortest path between `source` and `target`. `source` and `target` can either be elements or IDs of elements. `opt.weight` is an optional function returning a distance between two nodes. It defaults to `function(u, v) { return 1 }`. If `opt.directed` is `true`, the algorithm will take link direction into account. The implementation uses the [joint.alg.Dijkstra](#) plugin internally. Please refer to the plugin documentation for more information.

JointJS+ toAdjacencyList

graphUtils.toAdjacencyList

`toAdjacencyList(graph)`

Converts `graph` to an adjacency list. The method returns an object with element ids as properties and an array of connected element ids as values (the array includes indirect connections i.e. the path between connected elements includes a link-link connection).

Example:

```
const graph = new joint.dia.Graph;

const a = new joint.shapes.standard.Circle({ id: 'a' });
const b = new joint.shapes.standard.Circle({ id: 'b' });
const c = new joint.shapes.standard.Circle({ id: 'c' });

const ab = new joint.dia.Link({
  id: `a-b`,
  source: { id: 'a' },
  target: { id: 'b' },
});
const bc = new joint.dia.Link({
  id: `b-c`,
  source: { id: 'b' },
  target: { id: 'c' },
});
const indirectLink = new joint.dia.Link({
  id: `indirectLink`,
  source: { id: 'a' },
  target: { id: 'b-c' },
});

graph.addCell(a, b, c, ab, bc, indirectLink);

// Returns an object:
// {
//   "a": ["b", "c"],
//   "b": ["c"],
//   "c": []
// }
joint.graphUtils.toAdjacencyList(graph);
```

JointJS+ ForceDirected

layout.ForceDirected

ForceDirected plugin implements automatic layouts for graphs using a [force-directed approach](#). This is useful for, usually larger, undirected graphs.

Installation

Include the `joint.layout.ForceDirected.js` file into your HTML:

```
<script src="joint.layout.ForceDirected.js"></script>
```

Usage

ForceDirected layout plugin auto-lays out graphs based on three forces: repulsive force (forces nodes to move out of each other), attractive force (tries to get nodes together like a spring) and a gravity force (tendency of the nodes to move to a certain point).

The `ForceDirected` constructor accepts a couple of parameters for configuring the layout. The `gravityCenter` parameter is the point the nodes tend to move to. This is usually set to the center of the area where the nodes are to be laid out. `charge` parameter affects the repulsive force. Bigger the parameter is, bigger the repulsive force.

`linkDistance` and `linkStrength` both affect the attractive force (you can think of it as parameters of a spring that tightens together two nodes). Bigger the `linkStrength`, bigger the attractive force between two connected nodes. On the other hand, smaller the `linkDistance`, shorter links you allow and so the attractive force is bigger between two connected nodes.

There is no one-fits-all set of parameters that would give great result for all possible graphs. Therefore, it is suggested that you to play with the parameters, try to set different values so that it gives a good result for your use case.

```
var graphLayout = new joint.layout.ForceDirected({
  graph: graph // or array of selected cells
  x: 0,
  y: 0,
  width: 600,
  height: 400,
  gravityCenter: { x: 300, y: 200 },
  charge: 180,
  linkDistance: 30
});

graphLayout.start();

Array.from({ length: 100 }).forEach(function() { graphLayout.step(); });
```


Note: It is recommended to use the [requestAnimationFrame](#) for stepping the layout.

JointJS+ GridLayout

layout.GridLayout

GridLayout implements automatic layouts for graphs. This is useful in many scenarios, one of them being e.g. automatically positioning elements in [Stencil](#).

Installation

Include the `joint.layout.GridLayout.js` file into your HTML:

```
<script src="joint.layout.GridLayout.js"></script>
```

Usage

GridLayout plugin exposes one function `joint.layout.GridLayout.layout(graphOrElements, options)`. The first parameter `graphOrElements` is the `joint.dia.Graph` or an array of `joint.dia.Elements` we want to layout. The second parameter `options` is an object that contains various options for configuring the layout.

```
graph.addCell([
  new joint.shapes.standard.Rectangle({ size: { width: 80, height: 50 }}),
  new joint.shapes.standard.Rectangle({ size: { width: 50, height: 50 }}),
  new joint.shapes.standard.Ellipse({ size: { width: 80, height: 50 }}),
  new joint.shapes.standard.Ellipse({ size: { width: 50, height: 50 }})
]);

// Layout the entire graph
joint.layout.GridLayout.layout(graph, {
  columns: 2,
  columnWidth: 100,
  rowHeight: 70
});

// Layout the ellipses with minimal resulting `y` coordinate equals 100.
var ellipses = graph.getElements().filter(function(el) {
  return el instanceof joint.shapes.standard.Ellipse;
});
joint.layout.GridLayout.layout(ellipses, {
  columns: 2,
  marginY: 100
});
```

Here are the options for configuring the GridLayout:

- `columns` - Number of columns. It defaults to `1`.
- `columnWidth`
 - `'auto'` - width of a column is equal to the widest element amongst all elements
 - `'compact'` - width of a column is equal to the widest element in the column
 - `number` - value of a column width (e.g. 200)

It defaults to `'auto'`.
- `rowHeight`
 - `'auto'` - height of a row is equal to the highest element amongst all elements
 - `'compact'` - height of a row is equal to the highest element in the row
 - `number` - value of a row height (e.g. 100)

It defaults to `'auto'`.
- `verticalAlign`
 - `'top'` - the elements in the grid are aligned to the top of a grid cell
 - `'middle'` - the elements in the grid are aligned to the center of a grid cell
 - `'bottom'` - the elements in the grid are aligned to the bottom of a grid cell

It defaults to `'top'` (if `centre`) is disabled).
- `horizontalAlign`
 - `'left'` - the elements in the grid are aligned to the left side of a grid cell
 - `'middle'` - the elements in the grid are aligned to the center of a grid cell
 - `'right'` - the elements in the grid are aligned to the right side of a grid cell

It defaults to `'left'` (if `centre`) is disabled).
- `columnGap` - the horizontal gap between columns. It defaults to `0`.
- `rowGap` - the vertical gap between rows. It defaults to `0`.
- `resizeToFit` - Resizes the elements to fit a grid cell, preserving the aspect ratio (default: `false`)
- `marginX` - Sets the origin (`x` coordinate) of the most top-left element. It defaults to `0`.
- `marginY` - Sets the origin (`y` coordinate) of the most top-left element. It defaults to `0`.
- `deep` - Uses deep positioning for elements when set to `true` .i.e. moves not only the elements but also their descendants. It defaults to `false`.
- `parentRelative` - Position the elements relatively to their parents.

```
var padding = 20;
// layout the children of the element `el`
joint.layout.GridLayout.layout(el.getEmbeddedCells(), {
  parentRelative: true,
  marginX: padding,
  marginY: padding
});
// resize the element `el` to fit all children
el.fitEmbeds({ padding: padding });
```

- `setAttributes`(element, { `position`, `size` }, opt) - a callback function to customize how the positions (and sizes when `resizeToFit` enabled) are set to the elements.
- `dx` *deprecated* - Shifts the elements horizontally by a given amount. It defaults to `0`.

- `dy` *deprecated* - Shifts the elements vertically by a given amount. It defaults to `0`.
- `centre` *deprecated* - Positions the elements in the centre of a grid cell. It defaults to `true`.

The function returns an object with various geometry information.

- `bbox` - a tight bounding box around the elements
- `gridX` - an array of x-coordinates of vertical separators between columns.
- `gridY` - an array of y-coordinates of horizontal separators between rows.
- `rowHeights` - an array of row heights
- `columnWidths` - an array of column widths

JointJS+ StackLayout

layout.StackLayout

StackLayout implements automatic stack layouts for elements. This is useful in many scenarios, for example in creating kanban-styled diagrams or yamazumi charts.

Installation

Include the `joint.layout.StackLayout.js` file in your HTML:

```
<script src="joint.layout.StackLayout.js"></script>
```

Usage

The StackLayout plugin exposes the function `joint.layout.StackLayout.layout(model, options)`, `joint.layout.StackLayout.Directions` enum, and `joint.layout.StackLayout.Alignments` enum.

[Source Code](#)

layout.StackLayout.Alignments

alignment	value	description
Start	"start"	For TopBottom and BottomTop directions, the elements in the stack are aligned to the left side of the stack. For the other directions, they are aligned to the top side.
Middle	"middle"	The elements in the stack are always aligned to the middle of the stack.
End	"end"	For TopBottom and BottomTop directions, the elements in the stack are aligned to the right side of the stack. For the other directions, they are aligned to the bottom side.

layout.StackLayout.Directions

direction	value	description
TopBottom	"TB"	The elements in stacks are laid out from top to bottom.

BottomTop	"BT"	The elements in stacks are laid out from bottom to top.
LeftRight	"LR"	The elements in stacks are laid out from left to right.
RightLeft	"RL"	The elements in stacks are laid out from right to left.

layout.StackLayout.layout

The `layout` function takes two parameters: `model` and `options`. The `model` is the `joint.dia.Graph` or an array of `joint.dia.Elements` we want to lay out. The second parameter `options` is an object that contains various layout configuration options.

```
joint.layout.StackLayout.layout(graph, {
  direction: joint.layout.StackLayout.Directions.TopBottom,
  alignment: joint.layout.StackLayout.Alignments.Middle
});
```

Layout function options

direction	<u>StackLayout.Directions</u>	The direction of the stack layout elements. The default is <code>Directions.TopBottom</code> .
alignment	<u>StackLayout.Alignments</u>	The alignment of the stack layout elements. The default is <code>Alignments.Middle</code> .
stackSize	<i>number</i>	The size of the stack (width for vertical and height for horizontal layouts). The default is <code>100</code> .
stackGap	<i>number</i>	The gap between stacks. The default is <code>10</code> .
stackElementGap	<i>number</i>	The gap between elements in the stack. The default is <code>10</code> .
stackCount	<i>number</i>	The number of stacks in the layout. By default the number of stacks is calculated from the <code>stackIndex</code> property of elements.
topLeft	<i>object</i>	The object representing the top-left point of the layout (contains <code>x</code> and <code>y</code> properties).

bottomLeft	<i>object</i>	The object representing the bottom-left point of the layout (contains x and y properties).
topRight	<i>object</i>	The object representing the top-right point of the layout (contains x and y properties).
bottomRight	<i>object</i>	The object representing the bottom-right point of the layout (contains x and y properties).
setAttributes	<i>function</i>	A callback function to customize how the positions are set in the elements. <pre> setAttributes(element, { position }, opt) { element.set('position', position, opt); } </pre>
stackIndexAttributeName	<i>string</i>	The name of the element attribute specifying the index of the stack. The default is <code>"stackIndex"</code> .
stackElementIndexAttributeName	<i>string</i>	The name of the element attribute that specifies the position of the element in the stack. The default is <code>"stackElementIndex"</code> .

The function returns an object with various geometrical information.

bbox	<i>object</i>	A bounding box around elements.									
stacks	<i>Array<object></i>	The array of stack objects with these properties. <table> <tr> <td>bbox</td><td><i>object</i></td><td>A bounding box around the stack.</td></tr> <tr> <td>elements</td><td><i>Array<object></i></td><td>The array of elements of the stack.</td></tr> <tr> <td>index</td><td><i>number</i></td><td>The index of the stack inside the layout.</td></tr> </table>	bbox	<i>object</i>	A bounding box around the stack.	elements	<i>Array<object></i>	The array of elements of the stack.	index	<i>number</i>	The index of the stack inside the layout.
bbox	<i>object</i>	A bounding box around the stack.									
elements	<i>Array<object></i>	The array of elements of the stack.									
index	<i>number</i>	The index of the stack inside the layout.									

For fine-grained control of the stack layout, you can also set the following attributes on the element models.

stackIndex (specified by stackIndexAttributeName)	<i>number</i>	The attribute specifying the index of the stack (0-based indexing) in which the element is located. The default is 0 .
stackElementIndex (specified by stackElementIndexAttributeName)	<i>number</i>	The attribute specifying the position of the element in the stack. The element with the lower number is lower in the stack. For elements with the same index, the order of the elements in the graph matters. The default is 0 .

JointJS+ TreeLayout

layout.TreeLayout

`layout.TreeLayout` is a layout algorithm for **tree-like graphs**. It's great for displaying org charts, mind maps, class hierarchies and other tree structures. The `layout.TreeLayout` is strong in combination with the [ui.TreeLayoutView](#) which implements drag&drop functionality useful when editing a tree structure.

Note: The TreeLayout is meant to be used for tree graph structures only. Each node has a maximum of 1 parent, and there are no cycles. For generic graphs, you would need to use other layouts (DirectedGraph). You could also make the TreeLayout ignore the links that go back (with the filter option), and draw them as a curve for instance (a different style than other links, not affecting the position of other elements), or style the links with stubs as seen in the Links demo.

Installation

Include `joint.layout.treeLayout.js` file to your HTML:

```
<script src="joint.layout.treeLayout.js"></script>
```

Usage

Create an object of `layout.TreeLayout` type passing your graph (object of type `joint.dia.Graph`) as a parameter. Then you can just call the `layout()` method and the graph will be automatically laid out. It's important to note that the position of the root element of the tree will stay the same as it was before the layout. This allows you to move the tree to a position you want or have your tree be laid out “around” the root.

```
var graphLayout = new joint.layout.TreeLayout({
  graph: graph,
  parentGap: 20,
  siblingGap: 20
});

// Root element position stays the same after the layout.
// In this demo, we assume the root is the first element.
var root = graph.getElements()[0].position(200, 200);

graphLayout.layout();
```

Configuration

The following table lists options that you can pass to the `layout.TreeLayout` constructor function:

graph	The JointJS graph object on which you want to perform the layout.

parentGap	The distance between a parent element and its children. Defaults to 20 .
siblingGap	The distance between siblings. Defaults to 20 .
firstChildGap	The distance between the first sibling and its parent. It's applicable only for diagonal layouts ('TL' , 'TR' , 'BL' , 'BR'). It defaults to 20 .
direction	The default direction of the layout. It can be set to one of the following values: 'L' , 'R' , 'T' , 'B' (for left-to-right, right-to-left, top-to-bottom and bottom-to-top layouts) or diagonal variants 'TL' , 'TR' , 'BL' , 'BR' (for top-left, top-right, bottom-left and bottom-right). It defaults to 'R'. The <code>direction</code> defines how an element connects its parent. For instance <code>element.set('direction', 'R')</code> will appear on the right side to the parent element. Note that, this option is only used for nodes that do not have a <code>direction</code> property explicitly set.
filter(children, parent, opt)	An arbitrary element or sub-tree can be skipped by the layout. By providing a filter function, one can control, which elements will be laid out and which not. The filter function returns an array of elements to be laid out and runs for every single parent element in the graph. Parameters: <code>children</code> is an array of siblings <code>parent</code> is the parent of the <code>children</code> or null, when <code>children</code> are roots <code>opt</code> is the option object passed to <code>layout()</code> method when calling the layout. <pre>// e.g. a collapse branch is not visible in the paper but resides in the graph. function(children) { return children.filter(function(child) { return child.get('visible'); }); }</pre> No element is filtered out by default.
updatePosition(element, position, opt)	A function responsible for setting the resulting position on the elements. It calls <code>element.set('position', position, opt)</code> ; by default. It can be used for positioning the elements in an animated manner. For instance:

	<pre> element.transition('position', position, { delay: 300, valueFunction: joint.util.interpolate.object, }); </pre>
updateVertices(link, vertices, opt)	A function responsible for setting the resulting vertices on the links. It calls <code>link.set('vertices', vertices, opt);</code> by default. If the option is defined as <code>null</code> no vertices will be set by the layout.
attributeNames	An object that maps <u>element attributes</u> accepted by the layout to user defined attribute names. Useful for resolving conflicts when an attribute is already in use by the application logic. e.g Setting <code>{ siblingRank: 'index' }</code> will make the layout look for the <code>index</code> attribute instead of <code>siblingRank</code> when trying to figure out the order of siblings.
symmetrical	Enabling this layout option ensures that the distance between children is the same. This feature only works either in the horizontal or vertical direction.

The tree layout also reads some element properties allowing for a fine control of the layout engine. These are:

direction	The direction of the layout for this specific node. Note the tree layout can handle even mixed layout directions on a per-node basis. See the globally defined <code>direction</code> in the TreeLayout constructor function configuration.
siblingRank	The index of the element among its siblings.
offset	The additional distance from the parent (the real distance is a sum of <code>parentGap</code> - defined globally in the TreeLayout constructor function configuration . - and the <code>offset</code>).
margin	Extends the area occupied by the element on both sibling sides. <pre> element.set({ direction: 'R', size: { width: 100, height: 100 }, margin: 20 }); // as the `element` siblings are located above and under the // element, // it will in fact occupied an area of { width: 100, height: 140 } </pre>

prevSiblingGap	The additional gap next to the element sub-tree on the side of the previous sibling. The previous sibling is defined as the sibling with the closest smaller <code>siblingRank</code> .
nextSiblingGap	The additional gap next to the element sub-tree on the side of the next sibling. The next sibling is defined as the sibling with the closest larger <code>siblingRank</code> .
firstChildGap	The distance between the element and the first child. It's applicable only for diagonal layouts (<code>'TL'</code> , <code>'TR'</code> , <code>'BL'</code> , <code>'BR'</code>). It overrides the global <code>firstChildGap</code> option.

API

layout([opt])	Layout the graph in a tree. It iterates over all graph sources (elements without a parent) and layout them as separate trees. The option object <code>opt</code> , which will be passed down to all operations on the graph made by the layout call. e.g. <code>treeLayout.layout({ myFlag: true })</code> will internally set an element position as <code>element.position(x, y, { myFlag: true })</code>; .
getLayoutBBox()	Returns the bounding box of all affected elements from the last layout run (i.e. elements, which have been filtered are not taken into account when calculating the bounding box).
removeElement(element, [opt])	Removes element from graph and reconnects children to the parent of the removed element. The option object <code>opt</code> , will be passed down to all operations on the graph made by the <code>treeLayout.removeElement(e1, [opt])</code> . Additionally this method accepts <code>opt.layout = false</code> , this will omit calling layout function after removing the element.

JointJS+ bpmn

shapes.bpmn

The BPMN plugin provides you with set of **Business Process Model and Notation 2.0** shapes. They are designed to be easily configurable with the [ui.Inspector](#) plugin.

The plugin consists of flow objects (Activities, Events, Gateways), connection objects (Flows), swim lanes (Pools) and artifacts (DataObjects, Groups, Annotation). It also covers models introduced in version 2.0 as Conversations, Choreographies and Messages.

Install

Include `joint.shapes.bpmn.js` file to your HTML:

```
<script src="joint.shapes.bpmn.js"></script>
```

joint.shapes.bpmn.Activity

The Activity attributes:

activityType	Changes the type of the shape. 'task' 'transaction' 'event-sub-process' 'call-activity'
content	A text inside the element (i.e 'My Content').
icon	An icon name (list of icons).
subProcess	Any truthy value makes the sub-process icon visible (sub-process handling).
attrs	Allows to apply custom SVG attributes (fill, stroke, opacity, etc.) ' .body ' (rect), ' .label ' (text)

Usage:

```
var activity = new joint.shapes.bpmn.Activity({  
  attrs: {  
    '.body': { fill: 'gold', stroke: 'black' },  
    '.label': { text: 'My Activity' }  
  },  
  content: 'A content of the Activity',  
});
```

```

    activityType: 'transaction'
  });

```

joint.shapes.bpmn.Annotation

The Annotation attributes:

content	A text inside the element (i.e 'My Content').
attrs	Allows to apply custom SVG attributes (fill, stroke, opacity, etc.) ' .body ' (rect), ' .stroke ' (path)

Usage:

```

var annotation = new joint.shapes.bpmn.Annotation({
  attrs: {
    '.body': { fill: 'gold' },
    '.stroke': { stroke: 'silver' }
  },
  content: 'A content of the Annotation'
});

```

joint.shapes.bpmn.Choreography

The Choreography attributes:

content	A text inside the element (i.e 'My Content').
initiatingParticipant	An index or name of the participant (case-sensitive). (i.e ' participant 1 ')
participants	An array of participant names (i.e [' participant 1 ', ' participant 2 ']).
subProcess	Any truthy value makes the sub-process icon visible (<u>sub-process handling</u>).
attrs	Allows to apply custom SVG attributes (fill, stroke, opacity, etc.) ' .body ' (rect), ' .label ' (text), ' .participant-rect ' (rect), ' .participant-label ' (text)

Usage:

```

var choreography = new joint.shapes.bpmn.Choreography({
  attrs: {
    '.body': { fill: 'gold', stroke: 'black' },
    '.label': { fill: 'gray', text: 'My Choreography' },

```

```

        '.participant-rect': { fill: 'silver' }
    },
    participants: ['partic1', 'partic2', 'partic3'],
    initiatingParticipant: 2, // ='partic3',
    content: 'A content of the Choreography'
});

```

joint.shapes.bpmn.Conversation

The Conversation attributes:

conversationType	Changes the type of the shape. 'conversation' 'call-conversation'
attrs	Allows to apply custom SVG attributes (fill, stroke, opacity, etc.) ' .body ' (rect), ' .label ' (text)

Usage:

```

var conversation = new joint.shapes.bpmn.Conversation({
  attrs: {
    '.body': { fill: 'gold', stroke: 'black' },
    '.label': { fill: 'gray', text: 'My Conversation' }
  },
  conversationType: 'call-conversation'
});

```

joint.shapes.bpmn.DataObject

The Data Object attributes:

attrs	Allows to apply custom SVG attributes (fill, stroke, opacity, etc.) ' .body ' (rect), ' .label ' (text)

Usage:

```

var dataObject = new joint.shapes.bpmn.DataObject({
  attrs: {
    '.body': { fill: 'gold', stroke: 'black' },
    '.label': { fill: 'gray', text: 'My Data Object' }
  }
});

```

joint.shapes.bpmn.Event

The Event attributes:

eventType	Changes the type of the shape. 'start' 'end' 'intermediate'
icon	An icon name (list of icons).
attrs	Allows to apply custom SVG attributes (fill, stroke, opacity, etc.) ' .body ' (circle), ' .label ' (text)

Usage:

```
var event = new joint.shapes.bpmn.Event({
  attrs: {
    '.body': { fill: 'gold', stroke: 'black' },
    '.label': { fill: 'gray', text: 'My Event' }
  },
  icon: 'message',
  eventType: 'end'
});
```

joint.shapes.bpmn.Flow

The Flow as the only one inherits from `joint.dia.Link`.

The Annotation attributes:

flowType	Changes the type of the link. 'default' 'conditional' 'normal' 'association' 'conversation' 'message'

Usage:

```
var flow = new joint.shapes.bpmn.Flow({
  source: { id: task.id },
  target: { id: annotation.id },
  flowType: 'association'
});
```

joint.shapes.bpmn.Gateway

The Gateway attributes:

icon	An icon name (list of icons).

attrs	Allows to apply custom SVG attributes (fill, stroke, opacity, etc.) ' .body ' (polygon), ' .label ' (text)

Usage:

```
var gateway = new joint.shapes.bpmn.Gateway({
  attrs: {
    '.body': { fill: 'gold', stroke: 'black' },
    '.label': { fill: 'gray', text: 'My Gateway' }
  },
  icon: 'cross'
});
```

joint.shapes.bpmn.Group

The Group attributes:

attrs	Allows to apply custom SVG attributes (fill, stroke, opacity, etc.) ' .body ', ' .label ', ' .label-rect '

Usage:

```
var group = new joint.shapes.bpmn.Group({
  attrs: {
    '.label-rect': { fill: 'gold', stroke: 'black' },
    '.label': { fill: 'gray' }
  }
});
```

joint.shapes.bpmn.Message

The Message attributes:

attrs	' .body ' (polygon), ' .label ' (text)

Usage:

```
var message = new joint.shapes.bpmn.Message({
  attrs: {
    '.body': { fill: 'gold', stroke: 'black' },
    '.label': { fill: 'gray' }
  }
});
```

joint.shapes.bpmn.Pool

The Pool attributes:

lanes	An object consist of a label , headerWidth (optional, defaults to 20) and an array of sublanes (optional). A sublane is an object consist of a label , a name (optional), headerWidth (optional), ratio (number $<0-1>$, optional) and an array of sublanes (optional).
attrs	Allows to apply custom SVG attributes (fill, stroke, opacity, etc.) <code>'.body'</code> (rect), <code>'.header'</code> (rect), <code>'.lane-body'</code> (rect), <code>'.lane-header'</code> (rect), <code>'.blackbox-label'</code> (text)

Usage:

```
var pool = new joint.shapes.bpmn.Pool({
  attrs: {
    '.header': { fill: 'gold' },
    '.lane-header': { fill: 'silver' },
    '.myClass .lane-body': { fill: 'goldenrod' } // customize a specific lane body
  },
  color:
  },
  lanes: {
    label: 'My Pool',
    sublanes: [
      {
        label: 'lane 1',
        name: 'myClass', // this lane has an unique css class and therefore
        // in attrs above
        ratio: 0.6 // this lane will take up 60% of width of the parent lane
        // the other lanes (in this case `lane 2`) will take up
        the rest i.e. 40%
      },
      {
        label: 'lane 2',
        headerWidth: 40, // sets the header size for `lane2` to 40px
        sublanes: [] // it may consist of another sublanes
      }
    ]
  }
});
```

Each rendered lane DOM element has class name `lane` and attribute `data-lane-path`, that references corresponding lane configuration inside the model's `lanes` attribute. e.g. `data-lane-path="lanes/sublanes/1"` for the ``lane 2`` in the example above. It's useful when one needs to know, what lane a user just clicked on.

```
paper.on('element:pointerup', function(elementView, evt) {
  var pool = elementView.model;
  if (pool.get('type') === 'bpmn.Pool') {
    var lanePath = $(evt.target).closest('.lane').data('lane-path') || 'lanes';
    console.log(pool.prop(lanePath + '/label')); // log the lane label
  }
});
```

joint.shapes.bpmn.icons

The `joint.shapes.bpmn.icons` namespace includes several icons that can be used within the shapes.

none	No image.
message	
user	
plus	
cross	
circle	
service	

Usage:

```
gateway.set('icon', 'circle');
var task = new joint.shapes.bpmn.Activity({ icon: 'user' });
```

Adding custom icons can be made by inserting them into the `joint.shapes.bpmn.icons` namespace.

```
joint.shapes.bpmn.icons['myIcon'] = 'pathToImage'; // or data URI

// later on in the code
gateway.set('icon', 'myIcon');
```

Sub-process handling

The `subProcess` value is stored as a data attribute on the sub-process icon (SVG path) element. An interaction with the sub-process icon can be therefore caught and handled on the `joint.dia.Paper`. In the example bellow the model's `subProcess` value is shown after an user doubleclicks the icon.

```
conversation.set('subProcess', 'My sub-process ID');

paper.on('cell:pointerdblclick', function(cellView, evt) {

    var subProcess = $(evt.target).data('sub-process');

    if (subProcess) {
        alert(subProcess);
    }
});

// a doubleclick on conversation's sub-process icon will alert 'My sub-process ID'
```

JointJS+ bpmn2

shapes.bpmn2

The BPMN plugin provides you with a set of **Business Process Model and Notation 2.0** shapes. They are designed to be easily configurable with the [ui.inspector](#) plugin.

The plugin consists of flow shapes (Activities, Events, Gateways), connection objects (Flows), several icons and artifacts (DataObjects, Groups, Annotation). It also covers models introduced in version 2.0: Conversations and Connections.

Install

Include `joint.shapes.bpmn.js` file to your HTML:

```
<script src="joint.shapes.bpmn2.js"></script>
```

shapes.bpmn2.Activity

A rectangular Activity shape with icons.

source code

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
background	<i>SVGRectElement</i>	Rectangle background of the shape
border	<i>SVGPathElement</i>	Border of the shape (see built-in border styles and border types)
icon	<i>SVGImageElement</i>	Image containing the icon (see list of available icons)
label	<i>SVGTextElement</i>	Text inside the shape
markers	<i>SVGGElement</i>	Container for marker icons (see list of available markers)

Usage:

```
const activity = new joint.shapes.bpmn2.Activity({
  attrs: {
    'background': { fill: '#6764A7' },
    'icon': { iconType: 'receive' },
    'label': { text: 'My Activity' },
    'markers': {
      iconTypes: ['ad-hoc', 'loop'],
      iconsFlow: 'column',
      iconColor: '#ffffff'
    }
  }
});
```

shapes.bpmn2.Activity.ACTIVITY_MARKER_ICONS

Static property containing several marker icons that can be used with the Activity shape.

none	No image.
parallel	
sequential	≡
sub-process	⊕
compensation	⏮
ad-hoc	~
loop	↺

Add custom icons by defining them in the `joint.shapes.bpmn2.Activity.ACTIVITY_MARKER_ICONS` static property.

```
// define myIcon
joint.shapes.bpmn2.Activity.ACTIVITY_MARKER_ICONS['myIcon'] = 'pathToImage'; // or
data URI
```

```
// later on in the code
activity.attr('markers/iconTypes', ['myIcon']);
```

Markers handling

The marker value is stored as a data attribute on the marker icon (SVG path) element. An interaction with the marker icon can be caught and handled on the `joint.dia.Paper`. In the example bellow the model's marker name is displayed on interaction with the icon.

```
paper.on('element:marker:pointerdown', function(cellView, evt) {
  const markerName = evt.target.dataset.iconType;

  if (markerName) {
    alert(markerName);
  }
});
```

shapes.bpmn2.Activity.ACTIVITY_TYPE_ICONS

Static property including Activity type icons that can be used within the shapes.

none	No image.
receive	
send	
business-rule	
service	
script	
manual	
user	

Add custom icons by defining them in the `joint.shapes.bpmn2.Activity.ACTIVITY_TYPE_ICONS` static property.

```
// define myIcon
joint.shapes.bpmn2.Activity.ACTIVITY_TYPE_ICONS['myIcon'] = 'pathToImage'; // or data
URI

// later on in the code
activity.attr('icon/iconType', 'myIcon');
```

shapes.bpmn2.Activity.attributes.borderStyle

Set the style of the border.

solid	Solid style border (default)
dashed	Dashed style border
dotted	Dotted style border

Usage:

```
activity.attr('border/borderStyle', 'dashed');

const task = new joint.shapes.bpmn2.Activity({
  attrs: {
    border: {
      borderStyle: 'dashed'
    }
  }
});
```

shapes.bpmn2.Activity.attributes.borderType

Set the border type.

single	Single line border (default)

thick	Single line border but thicker
double	Double line border

Usage:

```
activity.attr('border/borderType', 'double');

const task = new joint.shapes.bpmn2.Activity({
  attrs: {
    border: {
      borderType: 'double'
    }
  }
});
```

shapes.bpmn2.Activity.attributes.iconColor

Set the icon or marker icons color.

Usage:

```
activity.attr('icon/iconColor', '#6764A7');

const task = new joint.shapes.bpmn2.Activity({
  attrs: {
    icon: {
      iconType: 'send',
      iconColor: '#6764A7'
    }
  }
});

activity.attr('markers/iconColor', '#6764A7');

const task = new joint.shapes.bpmn2.Activity({
  attrs: {
    markers: {
      iconTypes: ['ad-hoc', 'loop'],
      iconColor: '#6764A7'
    }
  }
});
```

```
    }  
  });
```

shapes.bpmn2.Activity.attributes.iconsFlow

Set the marker icons flow, available values are:

row	Marker icons will be displayed in a row (default)
column	Marker icons will be displayed in a column

Usage:

```
activity.attr('markers/iconsFlow', 'column');  
  
const task = new joint.shapes.bpmn2.Activity({  
  attrs: {  
    markers: {  
      iconTypes: ['ad-hoc', 'loop'],  
      iconsFlow: 'column'  
    }  
  }  
});
```

shapes.bpmn2.Activity.attributes.iconSize

Set the markers icon size.

Usage:

```
activity.attr('markers/iconSize', 24);  
  
const task = new joint.shapes.bpmn2.Activity({  
  attrs: {  
    markers: {  
      iconTypes: ['ad-hoc', 'loop'],  
      iconSize: 24  
    }  
  }  
});
```

```
    }  
  });
```

shapes.bpmn2.Activity.attributes.iconsOrigin

Set the marker icons origin:

left-top	Sets the origin to top left
left-bottom	Sets the origin to left bottom
right-top	Sets the origin to right top
right-bottom	Sets the origin to right bottom
top	Sets the origin to top middle
bottom	Sets the origin to bottom middle (default)
right	Sets the origin to right middle
left	Sets the origin to left middle
center	Sets the origin to center

Usage:

```
activity.attr('markers/iconsOrigin', 'center');  
  
const task = new joint.shapes.bpmn2.Activity({  
  attrs: {  
    markers: {  
      iconTypes: ['ad-hoc', 'loop'],  
      iconsOrigin: 'center'  
    }  
  }  
});
```

shapes.bpmn2.Activity.attributes.iconType

Set the main icon.

Usage:

```
activity.attr('icon/iconType', 'send');

const task = new joint.shapes.bpmn2.Activity({
  attrs: {
    icon: {
      iconType: 'send'
    }
  }
});
```

shapes.bpmn2.Activity.attributes.iconTypes

Set the marker icons. Property **has to be** an array of strings. **Note:** an array is required.

Usage:

```
activity.attr('markers/iconTypes', ['parallel', 'sequential'], { rewrite: true }); //
pass 'rewrite' option to replace markers, instead of merging them with existing ones

const task = new joint.shapes.bpmn2.Activity({
  attrs: {
    markers: {
      iconTypes: ['parallel', 'sequential']
    }
  }
});
```

shapes.bpmn2.Annotation

A rectangular shape with label.

source code

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
body	<i>SVGRectElement</i>	Rectangle body of the shape
border	<i>SVGPathElement</i>	Border of the shape (see border configuration).
label	<i>SVGTextElement</i>	Annotation text

Usage:

```
const group = new joint.shapes.bpmn2.Annotation({
  attrs: {
    'body': {
      stroke: '#fe854f'
    },
    'label': {
      text: 'Group'
    }
  }
});
```

shapes.bpmn2.Annotation.attributes.annotationD

Change the annotation border length. `annotationD` is an object with the following properties:

Property	Type	Description
size	Number	Thickness of Annotation border
side	String ('left' 'top' 'right' 'bottom')	Side on which to render Annotation border

Usage:

```
annotation.attr('border/annotationD', { size: 30, side: 'bottom' });

const task = new joint.shapes.bpmn2.Annotation({
  attrs: {
    border: {
      annotationD: {
        size: 30,
        side: 'bottom'
      }
    }
  }
});
```

shapes.bpmn2.AnnotationLink

A dashed link.

source code

Supported attrs properties:

Selector	Node	Description
line	<i>SVGPathElement</i>	Visible connection of the link
wrapper	<i>SVGPathElement</i>	An invisible, interactable wrapper around the connection used to make clicking on the link easier.

Usage:

```
const annotationLink = new joint.shapes.bpmn2.AnnotationLink({});
annotationLink.prop('source', { x: 450, y: 600 });
annotationLink.prop('target', { x: 400, y: 750 });
annotationLink.attr('line/stroke', '#fe854f');
```

shapes.bpmn2.Conversation

A polygon shape with label and markers.

source code

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
body	<i>SVGPolygonElement</i>	Polygon body of the shape
markers	<i>SVGGElement</i>	Container for marker icons (see list of available markers)
label	<i>SVGTextElement</i>	Shape text

Usage:

```
const conversation = new joint.shapes.bpmn2.Conversation({
  attrs: {
    'body': {
      fill: '#ffc2a7',
      stroke: '#fe854f'
    },
    'label': {
      text: 'My Conversation'
    }
  }
});
```

shapes.bpmn2.Conversation.CONVERSATION_MARKER_ICONS

The `joint.shapes.bpmn2.Conversation.CONVERSATION_MARKER_ICONS` static property includes icons that can be used within the shapes.

none	No image.
parallel	
sequential	
sub-process	
compensation	
ad-hoc	
loop	

Add custom icons by defining them in the `joint.shapes.bpmn2.Conversation.CONVERSATION_MARKER_ICONS` static property.

```
joint.shapes.bpmn2.Conversation.CONVERSATION_MARKER_ICONS['myIcon'] = 'pathToImage';  
// or data URI  
  
// later on in the code  
conversation.attr('markers/iconTypes', ['myIcon']);
```

Markers handling

The marker value is stored as a data attribute on the marker icon (SVG path) element. An interaction with the marker icon can be caught and handled on the `joint.dia.Paper`. In the example bellow the model's marker name is displayed on interaction with the icon.

```
paper.on('element:marker:pointerdown', function(cellView, evt) {  
  
  const markerName = evt.target.dataset.iconType;  
  
  if (markerName) {  
    alert(markerName);  
  }  
});
```



```
    }  
  });
```

shapes.bpmn2.Conversation.attributes.iconColor

Set the marker icons color.

Usage:

```
conversation.attr('markers/iconColor', '#6764A7');  
  
const task = new joint.shapes.bpmn2.Conversation({  
  attrs: {  
    markers: {  
      iconTypes: ['ad-hoc', 'loop'],  
      iconColor: '#6764A7'  
    }  
  }  
});
```

shapes.bpmn2.Conversation.attributes.iconsFlow

Set the marker icons flow, available values are:

row	Marker icons will be displayed in a row (default)
column	Marker icons will be displayed in a column

Usage:

```
conversation.attr('markers/iconsFlow', 'column');  
  
const task = new joint.shapes.bpmn2.Conversation({  
  attrs: {  
    markers: {  
      iconTypes: ['ad-hoc', 'loop'],  
      iconsFlow: 'column'  
    }  
  }  
});
```

```
    }  
  });
```

shapes.bpmn2.Conversation.attributes.iconSize

Set the marker icons size.

Usage:

```
conversation.attr('markers/iconSize', 24);  
  
const task = new joint.shapes.bpmn2.Conversation({  
  attrs: {  
    markers: {  
      iconTypes: ['ad-hoc', 'loop'],  
      iconSize: 24  
    }  
  }  
});
```

shapes.bpmn2.Conversation.attributes.iconsOrigin

Set the marker icons origin, available values are:

left-top	Sets the origin to top left
left-bottom	Sets the origin to left bottom
right-top	Sets the origin to right top
right-bottom	Sets the origin to right bottom
top	Sets the origin to top middle
bottom	Sets the origin to bottom middle (default)
right	Sets the origin to right middle

left	Sets the origin to left middle
center	Sets the origin to center

Usage:

```
conversation.attr('markers/iconsOrigin', 'center');

const task = new joint.shapes.bpmn2.Conversation({
  attrs: {
    markers: {
      iconTypes: ['ad-hoc', 'loop'],
      iconsOrigin: 'center'
    }
  }
});
```

shapes.bpmn2.Conversation.attributes.iconTypes

Set the marker icons. Property **has to be** an array of strings. **Note:** an array is required.

Usage:

```
conversation.attr('markers/iconTypes', ['ad-hoc', 'loop'], { rewrite: true }); //
pass 'rewrite' option to replace markers, instead of merging them with existing ones

const task = new joint.shapes.bpmn2.Conversation({
  attrs: {
    markers: {
      iconTypes: ['ad-hoc', 'loop']
    }
  }
});
```

shapes.bpmn2.ConversationLink

A double lined link.

source code

Supported `attrs` properties:

Selector	Node	Description
line	<i>SVGPathElement</i>	Visible connection of the link
outline	<i>SVGPathElement</i>	Outline of the link
wrapper	<i>SVGPathElement</i>	An invisible, interactive wrapper around the connection used to make clicking on the link easier.

Usage:

```
const conversationLink = new joint.shapes.bpmn2.ConversationLink({
  source: { x: 250, y: 260 },
  target: { x: 700, y: 260 },
  attrs: {
    line: {
      stroke: '#30d0c6',
    },
    outline: {
      stroke: '#5654a0'
    }
  }
});
```

shapes.bpmn2.DataAssociation

A dashed link.

source code

Supported `attrs` properties:

Selector	Node	Description
line	<i>SVGPathElement</i>	Visible connection of the link
wrapper	<i>SVGPathElement</i>	An invisible, interactive wrapper around the connection used to make clicking on the link easier.

Usage:

```
const dataAssociation1 = new joint.shapes.bpmn2.DataAssociation({
  source: { x: 250, y: 260 },
  target: { x: 700, y: 260 },
  attrs: {
    line: {
      stroke: '#30d0c6',
    }
  }
});
```

shapes.bpmn2.DataObject

A polygon shape with label.

source code

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
body	<i>SVGPathElement</i>	Path body of the shape (see body configuration options)
label	<i>SVGTextElement</i>	Shape text
dataTypelcon	<i>SVGImageElement</i>	Image containing the icon (see list of available icons)

collectionIcon	<i>SVGImageElement</i>	Image containing the collection icon (see list of available icons)

Usage:

```
const dataObject = new joint.shapes.bpmn2.DataObject({
  attrs: {
    'body': {
      fill: '#ffc2a7',
      stroke: '#fe854f'
    },
    'label': {
      text: 'Data Store'
    }
  }
});
```

shapes.bpmn2.DataObject.DATA_OBJECT_COLLECTION_ICONS

The `joint.shapes.bpmn2.DataObject.DATA_OBJECT_COLLECTION_ICONS` static property includes one icon that can be used within the shapes.

false	No image.
true	

Add custom icons by defining them in the

`joint.shapes.bpmn2.DataObject.DATA_OBJECT_COLLECTION_ICONS` static property.

```
joint.shapes.bpmn2.DataObject.DATA_OBJECT_COLLECTION_ICONS['myIcon'] = 'pathToImage';
// or data URI

// later on in the code
dataObject.attr('collectionIcon/iconType', 'myIcon');
```

shapes.bpmn2.DataObject.DATA_OBJECT_TYPE_ICONS

The `joint.shapes.bpmn2.DataObject.DATA_OBJECT_TYPE_ICONS` static property includes icons that can be used within the shapes.

none	No image.
input	
output	

Add custom icons by defining them in the `joint.shapes.bpmn2.DataObject.DATA_OBJECT_TYPE_ICONS` static property.

```
joint.shapes.bpmn2.DataObject.DATA_OBJECT_TYPE_ICONS['myIcon'] = 'pathToImage'; // or
data URI

// later on in the code
dataObject.attr('dataTypeInfo/iconType', 'myIcon');
```

shapes.bpmn2.DataObject.attributes.collection

Set the collection icon.

Usage:

```
dataObject.attr('collectionIcon/collection', 'true');

const task = new joint.shapes.bpmn2.DataObject({
  attrs: {
    collectionIcon: {
      collection: 'true'
    }
  }
});
```

shapes.bpmn2.DataObject.attributes.iconColor

Set the icon or collection icon color.

Usage:

```
dataObject.attr('collectionIcon/iconColor', '#6764A7');

const task = new joint.shapes.bpmn2.DataObject({
  attrs: {
    collectionIcon: {
      collection: 'true',
      iconColor: '#6764A7'
    }
  }
});

dataObject.attr('dataTypeIcon/iconColor', '#6764A7');

const task = new joint.shapes.bpmn2.DataObject({
  attrs: {
    dataTypeIcon: {
      iconType: 'input',
      iconColor: '#6764A7'
    }
  }
});
```

shapes.bpmn2.DataObject.attributes.iconType

Set the main icon.

Usage:

```
dataObject.attr('dataTypeIcon/iconType', 'input');

const task = new joint.shapes.bpmn2.DataObject({
  attrs: {
    dataTypeIcon: {
      iconType: 'input'
    }
  }
});
```


shapes.bpmn2.DataObject.attributes.objectD

Changes the fold of the shape.

Usage:

```
dataObject.attr('body/objectD', 30);

const task = new joint.shapes.bpmn2.DataObject({
  attrs: {
    body: {
      objectD: 30
    }
  }
});
```

shapes.bpmn2.DataStore

A polygon shape with label.

source code

Supported attrs properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
body	<i>SVGPathElement</i>	Path body of the shape
top	<i>SVGEllipseElement</i>	Top ellipse element
label	<i>SVGTextElement</i>	Shape text

Usage:

```
const datastore = new joint.shapes.bpmn2.DataStore({
  attrs: {
    'body': {
      fill: '#ffc2a7',
      stroke: '#fe854f'
    },
    'top': {
      fill: '#ffc2a7',
      stroke: '#fe854f'
    },
    'label': {
      text: 'Data Store'
    }
  }
});
```

shapes.bpmn2.DataStore.prototype.topRy

`dataStore.topRy()`

Return the vertical radius of the exposed area of the DataStore base (the value of the `body/lateralArea` attribute; `10` by default).

`dataStore.topRy(t, [opt])`

Set the datastore vertical radius of the exposed area of the cylinder base.

If the provided value is a percentage, it is relative to the `refBBox.height` of the shape. In practice, only values between `'0%'` and `'50%'` make sense. If the provided value is a number, it determines the vertical radius directly. Only values between `0` and half of `refBBox.height` make sense.

The function automatically sets the value of several attributes: `body/lateralArea`; and `top/ry`, `top/cy`, `top/refRy` and `top/refCy`. If these arguments need to be modified further, make sure to assign them only after calling `dataStore.topRy`.

Example usage:

```
const datastore = new bpmn2.DataStore();
datastore.resize(100, 200);
datastore.position(525, 75);
datastore.attr('root/title', 'joint.shapes.bpmn2.DataStore');
datastore.attr('body/fill', 'lightgray');
```

```
dataStore.attr('top/fill', 'gray');
dataStore.attr('label/text', 'DataStore');
dataStore.topRy('10%');
dataStore.addTo(graph);
```

shapes.bpmn2.Event

A circular shape with icon and label.

source code

Supported attrs properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
background	<i>SVGEllipseElement</i>	Ellipse background of the shape
border	<i>SVGPathElement</i>	Border of the shape (see built-in border types)
icon	<i>SVGImageElement</i>	Image containing the icon (see list of available icons)
label	<i>SVGTextElement</i>	Shape text

Usage:

```
const group = new joint.shapes.bpmn2.Event({
  attrs: {
    'body': {
      stroke: '#fe854f'
    },
    'label': {
      text: 'Group'
    }
  }
});
```

shapes.bpmn2.Event.EVENT_ICONS

The `joint.shapes.bpmn2.Event.EVENT_ICONS` static property includes several icons that can be used within the shapes.

none	No image.
message1	
message2	
timer1	
conditional1	
link1	
link2	
signal1	
signal2	
error1	
error2	
escalation1	
escalation2	
termination1	
termination2	

compensation1	
compensation2	
cancel1	
cancel2	
multiple1	
multiple2	
parallel1	
parallel2	

Add custom icons by defining them in the `joint.shapes.bpmn2.Event.EVENT_ICONS` static property.

```
joint.shapes.bpmn2.Event.EVENT_ICONS['myIcon'] = 'pathToImage'; // or data URI

// later on in the code
event.attr('icon/iconType', 'myIcon');
```

shapes.bpmn2.Event.attributes.borderStyle

Set the style of the border.

solid	Solid style border (default)
dashed	Dashed style border
dotted	Dotted style border

Usage:

```
event.attr('border/borderStyle', 'dashed');

const task = new joint.shapes.bpmn2.Event({
  attrs: {
    border: {
      borderStyle: 'dashed'
    }
  }
});
```

shapes.bpmn2.Event.attributes.borderType

Set the border type.

single	Single line border (default)
thick	Single line border but thicker
double	Double line border

Usage:

```
event.attr('border/borderType', 'double');

const task = new joint.shapes.bpmn2.Event({
  attrs: {
    border: {
      borderType: 'double'
    }
  }
});
```

shapes.bpmn2.Event.attributes.iconColor

Set the icon color.

Usage:

```

event.attr('icon/iconColor', '#6764A7');

const task = new joint.shapes.bpmn2.Event({
  attrs: {
    icon: {
      iconType: 'message1',
      iconColor: '#6764A7'
    }
  }
});

```

shapes.bpmn2.Event.attributes.iconType

Set the icon.

Usage:

```

event.attr('icon/iconType', 'message1');

const task = new joint.shapes.bpmn2.Event({
  attrs: {
    icon: {
      iconType: 'message1'
    }
  }
});

```

shapes.bpmn2.Flow

A link with configurable flow.

source code

Supported attrs properties:

Selector	Node	Description

line	<i>SVGPathElement</i>	Visible connection of the link (see configuration options)
wrapper	<i>SVGPathElement</i>	An invisible, interactive wrapper around the connection used to make clicking on the link easier.

Usage:

```
const flow = new joint.shapes.bpmn2.Flow({
  source: { x: 250, y: 340 },
  target: { x: 700, y: 340 },
  attrs: {
    line: {
      flowType: 'message'
    }
  }
});
```

shapes.bpmn2.Flow.attributes.flowType

The `joint.shapes.bpmn2.Flow` link comes with 4 flow types:

sequence	A simple link with no additional elements
default	Adds crossed line at the source of the link
conditional	Adds a rhombus at the source of the link
message	Dashed link with circle at the source of the link

Usage:

```
const flow = new joint.shapes.bpmn2.Flow({
  source: { x: 100, y: 200 },
  target: { x: 500, y: 200 },
  attrs: {
    line: {
      flowType: 'message'
    }
  }
});
```



```
    }
  });
```

shapes.bpmn2.Flow.attributes.markerFill

Flow link accepts additional attribute for styling the fill of the markers.

Note: markerFill applies only to message and conditional flowTypes.

Usage:

```
flow.attr('line/markerFill', '#fe854f');

const flow = new joint.shapes.bpmn2.Flow({
  source: { x: 100, y: 200 },
  target: { x: 500, y: 200 },
  attrs: {
    line: {
      markerFill: '#fe854f',
      flowType: 'message'
    }
  }
});
```

shapes.bpmn2.Gateway

A polygon shape with icon.

source code

Supported attrs properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes

body	<i>SVGPolygonElement</i>	Polygon body of the shape
label	<i>SVGTextElement</i>	Shape text
icon	<i>SVGImageElement</i>	Image containing the icon (see list of available icons)

Usage:

```
const gateway = new joint.shapes.bpmn2.Gateway({
  attrs: {
    'body': { fill: 'gold', stroke: 'black' },
    'icon': { iconType: 'exclusive' },
    'label': { text: 'My Activity' }
  }
});
```

shapes.bpmn2.Gateway.GATEWAY_ICONS

The `joint.shapes.bpmn2.Gateway.GATEWAY_ICONS` static property includes several icons that can be used within the shapes.

exclusive_blank	No image.
exclusive	
inclusive	
parallel	
event	
exclusive_event	
parallel_event	
complex	

Add custom icons by defining them in the `joint.shapes.bpmn2.Gateway.GATEWAY_ICONS` static property.

```
joint.shapes.bpmn2.Gateway.GATEWAY_ICONS['myIcon'] = 'pathToImage'; // or data URI

// later on in the code
gateway.attr('icon/iconType', 'myIcon');
```

shapes.bpmn2.Gateway.attributes.iconColor

Set the icon color.

Usage:

```
gateway.attr('icon/iconColor', '#6764A7');

const task = new joint.shapes.bpmn2.Gateway({
  attrs: {
    icon: {
      iconType: 'exclusive',
      iconColor: '#6764A7'
    }
  }
});
```

shapes.bpmn2.Gateway.attributes.iconType

Set the icon.

Usage:

```
gateway.attr('icon/iconType', 'exclusive');

const task = new joint.shapes.bpmn2.Gateway({
  attrs: {
    icon: {
      iconType: 'exclusive'
    }
  }
});
```

```
}  
});
```

shapes.bpmn2.Group

A rectangle shape with label.

source code

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
body	<i>SVGRectElement</i>	Rectangle body of the shape
wrapper	<i>SVGRectElement</i>	Rectangle wrapper of the shape
label	<i>SVGTextElement</i>	Shape text

Usage:

```
const group = new joint.shapes.bpmn2.Group({  
  attrs: {  
    'body': {  
      stroke: '#fe854f'  
    },  
    'label': {  
      text: 'Group'  
    }  
  }  
});
```

shapes.bpmn2.HeaderedPool

A rectangle shape with nested lanes and an additional header. Inherits from `joint.shapes.bpmn2.Pool` [shape](#) and adds a `header` rectangle and `headerLabel` text label.

[source code](#)

Additional `attrs` properties

Selector	Node	Description
header	<i>SVGRectElement</i>	Rectangular header of the shape, positioned to the left of the shape's body
headerLabel	<i>SVGTextElement</i>	Text label inside the header

To adjust the appearance of a `HeaderedPool` element, use the `Element.attr` [function](#):

```
const headeredPool = new joint.shapes.bpmn2.HeaderedPool();
headeredPool.resize(700, 400);
headeredPool.position(25, 200);
headeredPool.attr('root/magnet', false);
headeredPool.attr('laneHeaders/fill', '#3cb371');
headeredPool.attr('milestoneHeaders/fill', '#00fa9a');
headeredPool.attr('milestoneHeaders/stroke', '#333333');
headeredPool.attr('milestoneLines/stroke', '#3cb371');
headeredPool.attr('milestoneLines/strokeWidth', 2);
headeredPool.attr('milestoneLabels/fill', '#333333');
headeredPool.attr('milestoneLabels/fontStyle', 'italic');
headeredPool.attr('header/fill', '#00fa9a');
headeredPool.attr('headerLabel/text', 'Headered Pool');
headeredPool.attr('headerLabel/fill', '#ffffff');
headeredPool.addTo(graph);
```

shapes.bpmn2.HeaderedPool.prototype

All methods are inherited from the `joint.shapes.bpmn2.Pool` [shape](#).

shapes.bpmn2.HeaderedPool.attributes

All configuration properties are inherited from the `joint.shapes.bpmn2.Pool` shape. They are set during shape definition with the `Cell.define` function, or anytime later with the `Cell.set` function:

```
headeredPool.set('padding', 20);
headeredPool.set('padding', { top: 10, right: 20, bottom: 5, left: 10 });
headeredPool.set('lanes', [
  {
    label: 'fixed lane',
    size: 80,
    headerSize: 40
  },
  {
    label: 'lane with sublanes',
    sublanes: [
      {
        // omit the label property or make it undefined to hide the header
      },
      {
        label: 'sublane 1'
      }
    ]
  }
]);
headeredPool.set('milestones', [
  {
    label: 'milestone'
  },
  {
    label: 'milestone with fixed width',
    size: 350
  }
]);
```

shapes.bpmn2.Pool

A rectangle shape with nested lanes.

source code

Supported attrs properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
body	<i>SVGRectElement</i>	Rectangle body of the shape
laneGroups	<i>SVGGElement</i>	Selector for all lane groups
laneHeaders	<i>SVGRectElement</i>	Selector for all lane rect headers
laneLabels	<i>SVGTextElement</i>	Selector for all lane text labels
lanes	<i>SVGRectElement</i>	Selector for all lane rect bodies
milestoneGroups	<i>SVGGElement</i>	Selector for all milestone groups
milestoneHeaders	<i>SVGRectElement</i>	Selector for all milestone rect headers
milestoneLabels	<i>SVGTextElement</i>	Selector for all milestone text labels
milestoneLines	<i>SVGLineElement</i>	Selector for all milestone lines

Usage:

```
const pool = new joint.shapes.bpmn2.Pool({
  position: { x: 100, y: 100 },
  attrs: {
    laneHeaders: {
      fill: '#3cb371'
    },
    milestoneHeaders: {
      fill: '#00fa9a',
      stroke: '#333333'
    },
    milestoneLines: {
      stroke: '#3cb371',
      strokeWidth: 1
    },
    milestoneLabels: {
      fill: '#333333',
      fontStyle: 'italic'
    }
  }
},
```

```

milestonesSize: 30,
milestones: [
  {
    label: 'milestone',
  },
  {
    label: 'fixed milestone',
    size: 200
  },
  {
    label: 'milestone',
  }
],
headerSize: 30,
lanes: [
  {
    label: 'lane',
  },
  {
    label: 'fixed lane',
    size: 100
  },
  {
    label: 'lane'
  }
]
});

```

To adjust the appearance of a Pool element, use the `Element.attr` [function](#):

```

const pool = new joint.shapes.bpmn2.Pool();
pool.resize(700, 400);
pool.position(25, 200);
pool.attr('root/magnet', false);
pool.attr('laneHeaders/fill', '#3cb371');
pool.attr('milestoneHeaders/fill', '#00fa9a');
pool.attr('milestoneHeaders/stroke', '#333333');
pool.attr('milestoneLines/stroke', '#3cb371');
pool.attr('milestoneLines/strokeWidth', 2);
pool.attr('milestoneLabels/fill', '#333333');
pool.attr('milestoneLabels/fontStyle', 'italic');
pool.addTo(graph);

```


shapes.bpmn2.Pool.prototype

The Pool shape exposes functions that enable working with lanes and milestones:

shapes.bpmn2.Pool.prototype.getLaneBBox

`pool.getLaneBBox(laneGroupId)`

Get the bounding box of a lane group (header + lane) with `laneGroupId` id.

shapes.bpmn2.Pool.prototype.getLanePath

`pool.getLanePath(laneGroupId)`

Returns the path to the lane in attributes with `laneGroupId` id.

Example:

```
const path = pool.getLanePath('customId') + '/label';
pool.prop(path, 'new label', { rewrite: true });
```

shapes.bpmn2.Pool.prototype.getLanesFromPoint

`pool.getLanesFromPoint(point)`

Find lanes from a given `point`, where `point` is an object with `x` and `y` coordinates in the paper local coordinate system. Returns a sorted array of lane ids (from the most nested lane to the top parent) which bounding box contains the `point`.

Note: if lane has a custom id assigned, it will be returned instead.

shapes.bpmn2.Pool.prototype.getLaneIds

`pool.getLaneIds()`

Returns an array of all lane identifiers (in non-specific order).

shapes.bpmn2.Pool.prototype.getMilestoneBBox

`pool.getMilestoneBBox(milestoneGroupId)`

Get the bounding box of a milestone group (including its header) by `milestoneGroupId` id.

shapes.bpmn2.Pool.prototype.getMilestoneFromPoint

`pool.getMilestoneFromPoint(point)`

Find a milestone from a given `point`, where `point` is an object with `x` and `y` coordinates in the paper local coordinate system. Returns a string id of the milestone which bounding box contains `point`.

Note: if milestone has a custom id assigned, it will be returned instead.

shapes.bpmn2.Pool.prototype.getMinimalSize

`pool.getMinimalSize()`

Return the width and height of the minimal bounding box necessary to encompass all of the shape's content.

shapes.bpmn2.Pool.prototype.getParentLaneId

`pool.getParentLaneId(laneGroupId)`

Returns a parent lane id or null if the lane is a top-level lane.

shapes.bpmn2.Pool.attributes

Configuration properties

Property	Type	Description																		
lanes	<i>object[]</i>	Lanes in the Pool. Each lane is contained in a group along with the header and has a <u>unique id</u>. Each lane may have optional properties: <table><tr><th>Property</th><th>Type</th><th>Description</th></tr><tr><td>label</td><td>string</td><td>Text content of the lane's label (when left undefined the header with label will not be shown)</td></tr><tr><td>id</td><td>string</td><td>Custom id of the lane group</td></tr><tr><td>size</td><td>number</td><td>Size of the lane, when defined it will fix the lane height to the specified value</td></tr><tr><td>headerSize</td><td>number</td><td>Size of the lane header</td></tr><tr><td>sublanes</td><td><i>object[]</i></td><td>Nested lanes</td></tr></table>	Property	Type	Description	label	string	Text content of the lane's label (when left undefined the header with label will not be shown)	id	string	Custom id of the lane group	size	number	Size of the lane, when defined it will fix the lane height to the specified value	headerSize	number	Size of the lane header	sublanes	<i>object[]</i>	Nested lanes
Property	Type	Description																		
label	string	Text content of the lane's label (when left undefined the header with label will not be shown)																		
id	string	Custom id of the lane group																		
size	number	Size of the lane, when defined it will fix the lane height to the specified value																		
headerSize	number	Size of the lane header																		
sublanes	<i>object[]</i>	Nested lanes																		
milestones	<i>object[]</i>	Milestones in the Pool. Each milestone may have optional properties: <table><tr><th>Property</th><th>Type</th><th>Description</th></tr><tr><td>label</td><td>string</td><td>Text content of the milestone's label</td></tr><tr><td>id</td><td>string</td><td>Custom id of the milestone group</td></tr><tr><td>size</td><td>number</td><td>Size of the milestone, when defined it will fix the milestone width to the specified value</td></tr></table>	Property	Type	Description	label	string	Text content of the milestone's label	id	string	Custom id of the milestone group	size	number	Size of the milestone, when defined it will fix the milestone width to the specified value						
Property	Type	Description																		
label	string	Text content of the milestone's label																		
id	string	Custom id of the milestone group																		
size	number	Size of the milestone, when defined it will fix the milestone width to the specified value																		
milestonesSize	number	Height of all milestones																		
headerSize	number	Default size of all lane headers																		

padding	object	Padding within the Pool's body element, useful when creating another shapes based on Pool

Configuration properties are set during shape definition with the `Cell.define` function, or anytime later with the `Cell.set` function:

```
pool.set('padding', 20);
pool.set('padding', { top: 10, right: 20, bottom: 5, left: 10 });
pool.set('lanes', [
  {
    label: 'fixed lane',
    size: 80,
    headerSize: 40
  },
  {
    label: 'lane with sublanes',
    sublanes: [
      {
        id: 'customLane'
        // omit the label property or make it undefined to hide the header
      },
      {
        label: 'sublane 1'
      }
    ]
  }
]);
pool.set('milestones', [
  {
    label: 'milestone'
  },
  {
    label: 'milestone with fixed width',
    id: 'customMilestone',
    size: 350,
  }
]);

// change the attrs using custom id
pool.attr('lane_customLane/fill', '#3cb371');
pool.attr('milestoneLabel_customMilestone/fontSize', '18');
```

Id of lanes and milestones

Each lane and milestone group has a unique id that can be used to target those groups or individual elements within groups. Passing string to `id` property of a lane or milestone will create an alias to that given group. **Note:** custom id must be unique and the same id cannot be assigned for more than 1 milestone or 1 lane group.

Lane group id starts with a 'lanes' prefix and a combination of numbers that is based on position within an array and nest level (sublanes). Each lane group contains the following elements:

- lane rectangle (prefix `lane`)
- header rectangle (prefix `header` , note that header won't be added if label is undefined)
- label text (prefix `label` , note that label won't be added if it is undefined)

For example let's consider this lanes structure:

```
lanes: [  
  {  
    label: 'lane' // group id: 'lanes_0', lane id: 'lane_0', header id:  
'header_0', label id: 'label_0'  
  },  
  {  
    label: 'lane with sublanes', // group id: 'lanes_1', lane id: 'lane_1',  
    header id: 'header_1', label id: 'label_1'  
    sublanes: [  
      {  
        label: 'sublane 1' // group id: 'lanes_1_0', lane id: 'lane_1_0',  
        header id: 'header_1_0', label id: 'label_1_0'  
      },  
      {  
        label: 'sublane 2' // group id: 'lanes_1_1', lane id: 'lane_1_1',  
        header id: 'header_1_1', label id: 'label_1_1'  
      }  
    ]  
  },  
  // example with custom id  
  {  
    id: 'customId',  
    label: 'lane with custom id', // group id: 'lanes_customId', lane id:  
'lane_customId', header id: 'header_customId', label id: 'label_customId'  
    sublanes: [  
      {  
        label: 'sublane 1' // group id: 'lanes_2_0', lane id: 'lane_2_0',  
        header id: 'header_2_0', label id: 'label_2_0'  
      },  
      {  
        id: 'sublaneId',  
        label: 'sublane 2' // group id: 'lanes_sublaneId', lane id:  
'lane_sublaneId', header id: 'header_sublaneId', label id: 'label_sublaneId'  
      }  
    ]  
  }  
]
```

Similarly, each milestone group id starts with a 'milestone' prefix. Milestone group contains the following elements:

- header rectangle (prefix `milestoneHeader`)
- label text (prefix `milestoneLabel`)
- milestone line (prefix `milestoneLine`)

Example milestones structure:

```
milestones: [  
  {  
    label: 'milestone 1' // group id: 'milestone_0', header id:  
    'milestoneHeader_0', label id: 'milestoneLabel_0', line id: 'milestoneLine_0'  
  },  
  {  
    label: 'milestone 2' // group id: 'milestone_1', header id:  
    'milestoneHeader_1', label id: 'milestoneLabel_1', line id: 'milestoneLine_1'  
  },  
  // example with custom id  
  {  
    id: 'customId',  
    label: 'milestone 3' // group id: 'milestone_customId', header id:  
    'milestoneHeader_customId', label id: 'milestoneLabel_customId', line id:  
    'milestoneLine_customId'  
  }  
]
```

shapes.bpmn2.Pool.attributes.labelAlignment

Aligns the label to the specified side, relatively to the header:

left-top	Sets the alignment to top left
left-bottom	Sets the alignment to left bottom
right-top	Sets the alignment to right top
right-bottom	Sets the alignment to right bottom
top	Sets the alignment to top middle
bottom	Sets the alignment to bottom middle

right	Sets the alignment to right middle (default for milestone labels)
left	Sets the alignment to left middle
center	Sets the alignment to center (default for lane labels)

Usage:

```
const pool = new joint.shapes.bpmn2.Pool({
  lanes: [
    {
      label: 'pool lane',
      id: 'firstLane'
    },
    {
      label: 'pool lane'
    }
  ],
  milestones: [
    {
      label: 'pool milestone',
      id: 'firstMilestone'
    },
    {
      label: 'pool milestone'
    }
  ],
  attrs: {
    laneLabels: {
      labelAlignment: 'left' // aligns all lane labels to the left side
    },
    milestoneLabels: {
      labelAlignment: 'center' // aligns all milestone labels in the center
    },

    label_firstLane: {
      labelAlignment: 'right' // align lane label with custom id
    },
    milestoneLabel_firstMilestone: {
      labelAlignment: 'right' // align milestone label with custom id
    }
  }
});

// align all labels
```

```

pool.attr('laneLabels/labelAlignment', 'left');
pool.attr('milestoneLabels/labelAlignment', 'center');

// align individual labels with custom id
pool.attr('label_firstLane/labelAlignment', 'right');
pool.attr('milestoneLabel_firstMilestone/labelAlignment', 'right');

```

shapes.bpmn2.Pool.attributes.labelMargin

Applies margin around the label. The `util.normalizeSides` is used to understand the provided value. A single number is applied as padding to all sides of the elements. An object may be provided to specify values for `left`, `top`, `right`, `bottom`, `horizontal` and/or `vertical` sides.

Usage:

```

const pool = new joint.shapes.bpmn2.Pool({
  headerSize: 32,
  milestonesSize: 32,
  lanes: [
    {
      label: 'pool lane',
      id: 'firstLane'
    },
    {
      label: 'pool lane'
    }
  ],
  milestones: [
    {
      label: 'pool milestone',
      id: 'firstMilestone'
    },
    {
      label: 'pool milestone'
    }
  ],
  attrs: {
    laneLabels: {
      labelMargin: 4 // adds 4px margin on all sides to all lane labels
    },
    milestoneLabels: {
      labelMargin: { vertical: 10 } // adds 10px top and 10px bottom margin to
    all milestone labels
    },
  },

```



```
// control margin of individual labels with custom id
label_firstLane: {
    labelMargin: 0
},
milestoneLabel_firstMilestone: {
    labelMargin: { horizontal: 8 }
}
}
});

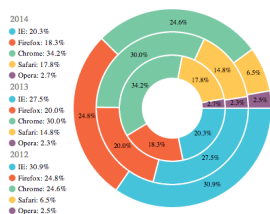
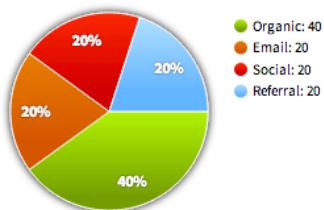
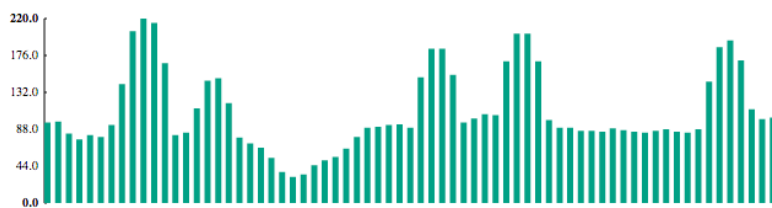
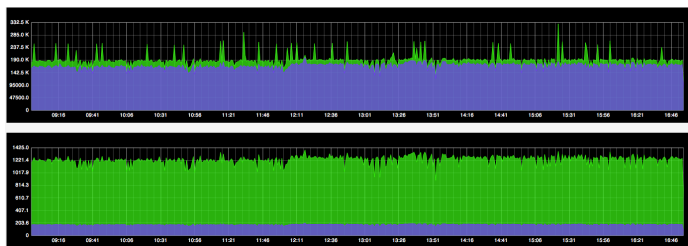
// add margin to all labels
pool.attr('laneLabels/labelMargin', 4);
pool.attr('milestoneLabels/labelMargin', { vertical: 6 });

// control the margin of individual labels with custom id
pool.attr('label_firstLane/labelMargin', 0);
pool.attr('milestoneLabel_firstMilestone/labelMargin', { vertical: 4, horizontal: 20
});
```

JointJS+ chart

shapes.chart

The Chart plugin provides you with a set of flexible, good looking and **interactive charts** that you can use in your applications. Note that from the JointJS/JointJS+ perspective, a chart is **just another JointJS cell**. This means that you can add it to your diagram, resize it, rotate it, clone it, connect it with other elements, serialize/deserialize it to/from JSON format or export it to SVG, PNG or JPEG. No other library can give you this flexibility!



Install

Include `joint.shapes.chart.js` file to your HTML:

```
<script src="joint.shapes.chart.js"></script>
```

Chart Types

The following is a list of currently supported chart types:

- [Line chart](#)
- [Bar chart](#)
- [Area chart](#)
- [Combo chart](#)
- [Time series](#)
- [Pie chart](#)
- [Donut chart](#)
- [Knob chart](#)

shapes.chart.knob

chart.Knob is a great component for displaying numbers in a fancy way. They can also be animated to better visualize changes in values.



Create a basic Knob

The code snippet below creates a basic Knob. The Knob contains only one value.

```
var graph = new joint.dia.Graph;
var paper = new joint.dia.Paper({ el: $('#paper'), width: 200, height: 200, model:
graph });

var knob = new joint.shapes.chart.Knob({
  position: { x: 20, y: 20 },
  size: { width: 100, height: 100 },
  min: 0, max: 100, value: 80,
  fill: '#2c97de'
});
graph.addCell(knob);
```

Styling the Knob

To set color of the knob, simply pass `fill` attribute with color of your choice to the knob constructor. Moreover, as it is common in JointJS/JointJS+, you can style any graphical element of the knob shape. For example, you might want to change the font family or other font properties of the legend. For this, you can customize the knob using the usual JointJS approach, via the `attrs` object or the `attr()` method. This means that by just knowing the structure of the SVG elements the knob is built from, you have full flexibility in styling your knob. Only SVG and CSS standards are your limits.

Let's have a look at some examples demonstrating different knob styles. For the full SVG structure of the knob, please see the section Chart SVG Structure in this document. Once you know the structure, it is easy to make CSS selectors referencing the SVG elements that you want to style.

In the example above, we showed how to pass styling in the `attrs` object when creating a new knob but you can also change attributes of an existing knob:

```
knob.attr('.slice .slice-fill/stroke', 'black')
knob.attr('.slice .slice-fill/stroke-width', 2)
knob.attr('.legend-slice text/font-size', 16)
```

Note that you can use not only `fill` but a whole range of other SVG attributes to style your knobs. Moreover, JointJS provides a facility to use gradients and filters. You can take advantage of them to make your charts even prettier:

```
var knob = new joint.shapes.chart.Knob({
  ...
  attrs: {
    '.data': {
      filter: { name: 'dropShadow', args: { dx: 0, dy: 0, blur: 3, color:
'black' } }
    }
  }
  ...
});
```

Multiple values

The Knob chart supports multiple series. The `value`, `fill`, `min` and `max` properties of the Knob constructor accept arrays. Simply pass more values into the arrays and the Knob chart will automatically render multiple series as in the following example:

```
var chart = new joint.shapes.chart.Knob({
  ...
  value: [30, 60, 90],
  fill: ['#F2C500', '#4CC3D9', '#E94B35'],
  min: 0, max: 100
});
```

Tooltips

Tooltips are a great way to show contextual information in your charts. The following demo shows how to use the `ui.Tooltip` plugin with combination with Knob chart:

IMPORTANT: For tooltips to work, you must have the `ui.Tooltip` plugin installed.

When setting up tooltips, we no longer work with the chart model. Instead, we request the paper for the view that represents the chart. This view triggers `mousemove`, and `mouseout` events that we use to hide and show our tooltip. The `mousemove` event handler gets passed two important arguments: `slice` and `event`. These two arguments mean the following:

- `slice` - an object with useful information about the slice the event was triggered on. This object contains the following properties: `sliceIndex`, `serieIndex`, `value`, `percentage`, `fill`, `offset`, `angle`, `startAngle`, `endAngle`, `outerRadius`, `innerRadius`, and `label`. Most of the time, for tooltips, the important ones are `value` and `label`.
- `event` - This is the DOM event object. For tooltips, the two important properties are `clientX` and `clientY`. We use these to render the tooltip at a particular position.

All these three arguments help us render our tooltip.

Animation

In general, you can animate any property of the Knob chart (change its value over a certain period of time). In the following demo, we will show how to animate some properties of the Knobs. Using the same technique, you can animate an arbitrary property.

```
knob.transition('value', 100, { duration: 1000, timingFunction:
joint.util.timing.exponential });
knob.transition('pieHole', .4, { duration: 1000, timingFunction:
joint.util.timing.exponential });
```

You can find more about JointJS transitions in the [API reference](#).

Knob Options

This is the full list of options that you can pass to the Knob constructor function.

- `value` - a number or an array of numbers (in case of multiple series) representing the value of the knob.
- `fill` - a color or an array of colors (in case of multiple series) representing the color of the knob. If multiple series are used, the number of items in the `value` array and `fill` array should match. If they don't match, the first color from the `fill` array will be taken wherever a color on the index matching the current serie value is missing.
- `min` - a minimum value of the knob. `min` together with `max` specify the range for the `value`. The same logic for multiple series as described above for `fill` applies.
- `max` - a maximum value of the knob. The same logic for multiple series as described above for `fill` applies.
- `serieDefaults` - an object with various properties specifying how series should be rendered. As Knob inherits from Pie chart, all of the properties described in [Pie serie object](#) applies here as well.
- `sliceDefaults` - an object with various properties specifying how slices should be rendered. As Knob inherits from Pie chart, all of the properties described in [Pie slice object](#) applies here as well.

Knob API

chart.Knob

As `chart.Knob` inherits from `chart.Pie` all of the `chart.Pie` API methods apply here as well.

chart.KnobView

As `chart.Knob` inherits from `chart.Pie` all of the `chart.PieView` API methods apply here as well.

chart.KnobView events

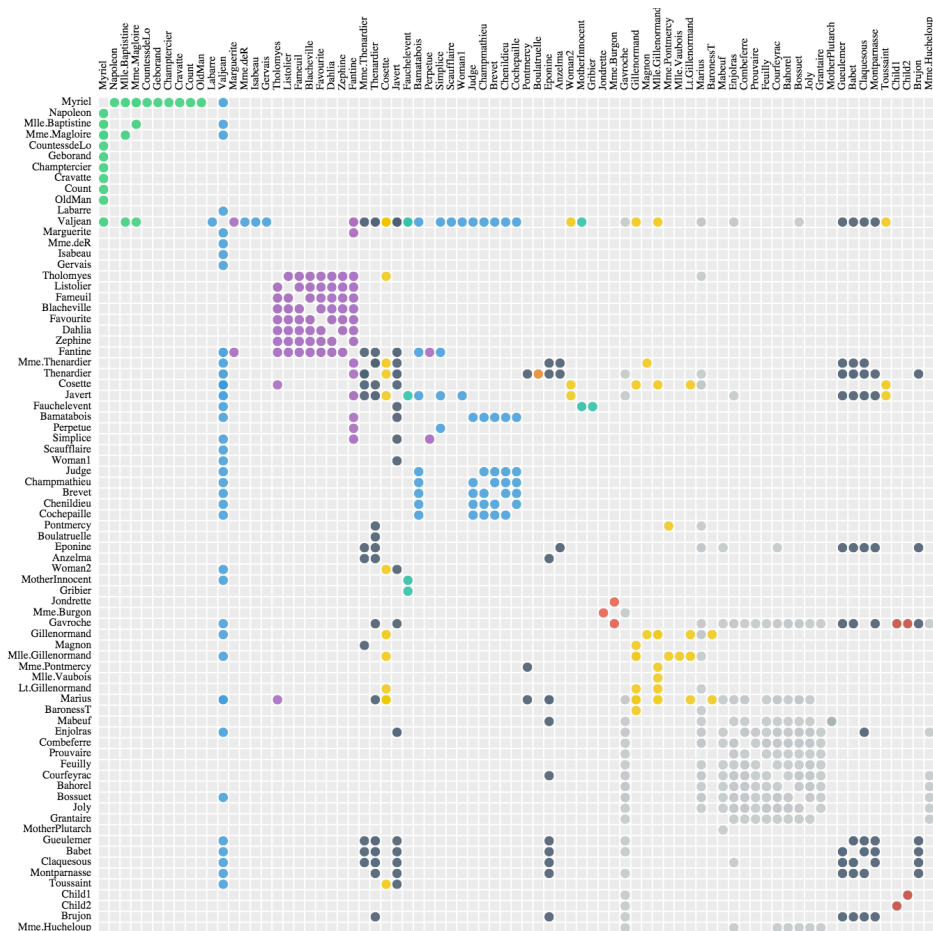
As `chart.Knob` inherits from `chart.Pie` all of the `chart.Pie` events apply here as well.

Knob SVG structure

As `chart.Knob` inherits from `chart.Pie`, the SVG structure described in [chart.Pie SVG structure](#) applies here as well.

shapes.chart.matrix

chart.Matrix is great for displaying co-occurrences in a matrix.



Create a Matrix Chart

The code snippet below creates a basic Matrix chart with three rows and three columns.

```
var graph = new joint.dia.Graph;
var paper = new joint.dia.Paper({ el: $('#paper'), width: 500, height: 500, model:
graph });

var matrix = new joint.shapes.chart.Matrix({
  position: { x: 50, y: 50 },
  size: { width: 170, height: 170 },
  cells: [
    [{ fill: 'red' }, { fill: 'black' }, { fill: 'black' }],
    [{ fill: 'black' }, { fill: 'red', rx: 50, ry: 50 }, { fill: 'black' }],
    [{ fill: 'black' }, { fill: 'black' }, { fill: 'red' }]
  ],
  labels: {
    rows: [{ text: '1' }, { text: '2' }, { text: '3' }],
    columns: [{ text: 'A' }, { text: 'B' }, { text: 'C' }],
  }
});
graph.addCell(matrix);
```

Styling the Matrix

All the cells in the matrix are SVG rectangles. The objects in the `cells` array can contain any SVG attributes that will be used to style those rectangles. The most common one is `fill` which allows you to set the fill color of the rectangle. Another useful ones are radii `rx` and `ry` which can effectively turn the rectangles into circles if the radii specified are big enough (as we saw in the above example).

However, there is more we can do to make our matrices much prettier. For example, we can use `opacity` to make our cells semi-transparent, set `stroke`, style labels, grid lines or the background of the matrix.

Real-world example

In our previous examples, we were trying to show the basics of working with Matrix chart. However, the real power of Matrix charts is to show co-occurrences. The following chart is somewhat bigger and shows the relationships from the *Les Misérables* book by Victor Hugo. The demo also integrates the [ui.Tooltip](#) plugin to show how we can display tooltips when cells are hovered.

Conway's Game of Life example

The following example shows the great Conway's [Game of Life](#) cellular automaton.

Matrix Options

This is the full list of options that you can pass to the Matrix constructor function.

- `cells` - an array of cells each being an object that can contain SVG attributes for the cell rectangle. If a cell is undefined, it is not rendered. If the matrix is sparse, setting empty cells to an undefined value significantly improves performance when rendering large matrices.

- `labels` - an object containing `rows` and `columns` arrays. These arrays contain objects defining the text for the labels on the respective axis. The most important ones are `text` (the actual label), `font-size`, `font-family` and `fill` but you can use any other SVG attribute for styling the labels.

Matrix SVG structure

Knowing the SVG structure of the matrix allows you to further style the matrix via the `attrs` object or [`attr\(\)`](#) method.

shapes.chart.pie

`chart.Pie` is a great chart type for displaying numerical proportions. Donut charts, multiple series, animation, changing look & feel of any part of the chart, drop shadows and other filters and gradients are all supported.

Create a basic Pie chart

The code snippet below creates a basic Pie chart. The pie chart contains only one serie and its legend is displayed by default.

```
var graph = new joint.dia.Graph;
var paper = new joint.dia.Paper({ el: $('#paper'), width: 500, height: 500, model: graph });

var chart = new joint.shapes.chart.Pie({
  position: { x: 50, y: 10 },
  size: { width: 100, height: 100 },
  series: [ { data: [
    { value: 40, label: 'Organic', fill: '#8bce5d' },
    { value: 20, label: 'Email', fill: '#53abdd' },
    { value: 20, label: 'Social', fill: '#c377b1' },
    { value: 20, label: 'Referral', fill: '#ffe891' }
  ] }
]);
graph.addCell(chart);
```

Styling Pie Slices

Normally, you can just pass a color of each slice in the `fill` property of the slice object as you can see in the example above. However, as it is common in JointJS/JointJS+, you can style any graphical element of the chart shape. For example, you might want to change the font family or other font properties of the slice labels, or the legend. For this, you can customize the chart using the usual JointJS approach, via the `attrs` object or the [`attr\(\)`](#) method. This means that by just knowing the structure of the [SVG elements the chart is built from](#), you have full flexibility in styling your chart. Only SVG and CSS standards are your limits. Following the JointJS philosophy, we don't try to invent special parameters. Instead, we try to follow SVG and CSS standards as much as we can.

Let's have a look at one snippet of the SVG structure of the charts that is used to render a slice. For the full SVG structure of charts, please see the section [Chart SVG Structure](#) in this document. Once you know the structure, it is easy to make CSS selectors referencing the SVG elements that you want to style. We'll show an example of that below. the SVG structure of one series is the following:

```
<g class="slice serie-[index of the serie] slice-[index of the slice]">
  <path class="slice-fill">
  <path class="slice-border">
  <text class="slice-inner-label">
</g>
```

Knowing that, we can, for example, change color and size of the text label of the first slice:

```
var chart = new joint.shapes.chart.Pie({
  ... /* the same as before */ ...
  attrs: {
    '.slice-0 .slice-inner-label': { fill: 'black', 'font-size': 20 }
  }
});
```

In the example above, we showed how to pass styling in the `attrs` object when creating a new chart but you can also change attributes of an existing chart:

```
chart.attr('.slice-0 .slice-inner-label/fill', 'blue')
```

Note that you can use not only `fill` but a whole range of [other SVG attributes](#) to style your charts. Moreover, JointJS provides a facility to use gradients and filters. You can take advantage of them to make your charts even prettier:

```
var chart = new joint.shapes.chart.Pie({
  ...
  series: [ { data: [
    { value: 40, label: 'Organic', fill: {
      type: 'linearGradient', stops: [
        { offset: '0%', color: '#b4f200' },
        { offset: '80%', color: '#759d00' }
      ],
      attrs: { x1: '0%', y1: '0%', x2: '0%', y2: '100%' }
    } },
    ...
  ]}],
  ...
});
```

Multiple series

The Pie chart supports multiple series. Simply add more series to the `series` array. This is useful for creating pie charts that, for example, compare data (from different sources, periods, ...).

```
var chart = new joint.shapes.chart.Pie({
  ...
  series: [
    { label: '2014', data: [
      { value: 20.3, label: 'IE', fill: '#4CC3D9' },
      { value: 18.3, label: 'Firefox', fill: '#F16745' },
      { value: 34.2, label: 'Chrome', fill: '#7BC8A4' }
    ]},
    { label: '2013', data: [
      { value: 27.5, label: 'IE', fill: '#4CC3D9' },
      { value: 20, label: 'Firefox', fill: '#F16745' },
      { value: 30, label: 'Chrome', fill: '#7BC8A4' }
    ]},
    ...
  ],
  ...
});
```

All series have automatically generated legends. This **legend is interactive**. Based on the effect you defined, the user can either hover or click on the labels in the legend and offset (or enlarge) the associated slices in the chart.

Exploding a Slice

Slices in the Pie chart can be separated from the rest (exploded) by setting the `offset` property of the slice object:

Donut Charts

The difference between Donut and Pie charts is very small, the only thing that differs is the hole inside the pie. While there is no hole for Pie chart, Donut charts have a hole that can be set arbitrarily. The `pieHole` property can either be a floating point number between `0` and `1`, in which case the size of the hole is proportional to the radius of the pie chart. If the `pieHole` property is bigger than `1`, it is considered to be an absolute size of the hole in pixels:

Tooltips

Tooltips are a great way to show contextual information in your charts. The following demo shows how to use the `ui.Tooltip` plugin with combination with Pie chart:

IMPORTANT: For tooltips to work, you must have the `ui.Tooltip` plugin installed.

When setting up tooltips, we no longer work with the chart model. Instead, we request the paper for the view that represents the chart. This view triggers `mousemove`, and `mouseout` events that we use to hide and show our tooltip. The `mousemove` event handler gets passed two important arguments: `slice` and `event`. These two arguments mean the following:

- `slice` - an object with useful information about the slice the event was triggered on. This object contains the following properties: `sliceIndex`, `serieIndex`, `value`, `percentage`, `fill`, `offset`, `angle`,

`startAngle`, `endAngle`, `outerRadius`, `innerRadius`, and `label`. Most of the time, for tooltips, the important ones are `value` and `label`.

- `event` - This is the DOM event object. For tooltips, the two important properties are `clientX` and `clientY`. We use these to render the tooltip at a particular position.

Legend

The chart legend is auto-generated and contains labels and colors of all the data series. The legend is interactive by default and allows the user to highlight the slices displayed in the chart while hovering the legend items. This section shows you how you can **customize the legend and its position**.

The legend is rendered as an SVG group element with the class `Legend`. Inside this group, we have `Legend-item`s each containing an SVG circle and text elements. Please refer to the [Chart SVG structure](#) section for the full SVG structure of the chart. This is all you need to know to be able to style the legend. By default, the legend is positioned on the top right corner of the pie chart.

The styling of the legend item text (name of the associated slice) and the legend item circle can be done the usual `JointJS` way, through standard SVG attributes as you can see below. Another useful properties for styling the legend items are `legendLabelLineHeight`, `legendLabelMargin` and `labelLineHeight`. The first two can be defined in the [slice objects](#) (or in the `sliceDefaults` for all slices) while the third one is specific to the legend item for the series and therefore is defined in the [serie object](#) (or in `serieDefaults`).

```
// Positioning to the top (80px above the chart and centered).
chart3.attr('.legend/ref-y', -80);
chart3.attr('.legend/ref-x', .5);
chart3.attr('.legend/x-alignment', -.5);

// Styling of the legend text and circle.
chart3.prop('sliceDefaults/legendLabel', '{label} is {value}%');
chart3.prop('sliceDefaults/legendLabelLineHeight', 8);
chart3.prop('sliceDefaults/legendLabelMargin', 25);
chart3.attr('.legend-slice text/fill', '#336699');
chart3.attr('.legend-slice text/font-weight', 'bold');
chart3.attr('.legend-slice text/text-decoration', 'underline');
chart3.attr('.legend-slice text/font-size', 13);
chart3.attr('.legend-slice circle/r', 7);
chart3.attr('.legend-slice circle/stroke', 'black');
chart3.attr('.legend-slice circle/stroke-width', 1);
```

Animation

In general, you can animate any property of the Pie chart (change its value over a certain period of time). In the following demo, we will show how to animate some properties of the Pie chart. Using the same technique, you can animate an arbitrary property.

```
chart.transition('pieHole', .6, { duration: 700 });
chart.transition('series/0/data/0/offset', 50, { duration: 700 });
chart.transition('attrs/.slice-inner-label/font-size', 30, { duration: 700 });
```

```
chart.transition('series/0/data/0/fill', '#ff0000', {
  duration: 700,
  valueFunction: joint.util.interpolate.hexColor
});
```

You can find more about JointJS transitions in the [API reference](#).

Effects

The effect applied when the user either clicks or hover a slice in the pie can be customized. There are two actions that trigger an effect (`onClickEffect` and `onHoverEffect`) and two types of effects: `enlarge` and `offset` . Try to interact (hover/click slices) with the pie chart in the demo below to see the different effects applied.

```
var chart = new joint.shapes.chart.Pie({
  ...
  sliceDefaults: {
    onClickEffect: { type: 'offset', offset: 20 },
    onHoverEffect: { type: 'enlarge', scale: 1.5 }
  }
});
```

Chart Options

This is the full list of the options that you can pass to the chart constructor function.

- `series` - an array of [series](#).
- `pieHole` - the size of the hole in the center of the pie. If the number is in the `0..1` interval, it is considered to be proportional to the pie radius. If it is bigger than `1` , it is considered to be the absolute radius in pixels.
- `serieDefaults` - an object with various properties specifying how series should be rendered. All properties of this object are overruled by properties of the same names in the [serie object](#).
- `sliceDefaults` - an object with various properties specifying how slices should be rendered. All properties of this object are overruled by properties of the same names in the [slice object](#).

Serie

These are items of the `series` array.

- `name` - the name of the serie. This name is added as a CSS class to the serie SVG element. You can take advantage of this CSS class for further [styling](#).
- `label` - the serie label (used in the auto-generated legend)
- `data` - an array of [slices](#)
- `startAngle` - the angle in degrees the serie slices start to draw.
- `degree` - the total degree of the serie in the pie.
- `showLegend` - `true` if the legend for the serie should be shown, `false` otherwise.
- `labelLineHeight` - the line height of the serie label in the auto-generated legend

When the above properties are defined in the `serieDefaults` options object, they will be applied to all the series unless overruled by properties in the serie object.

Slice

These are items of the `data` array of the [serie object](#).

- `value` - a number specifying the value for the slice data point.
- `label` - the label for the slice. This label is used in the legend.
- `fill` - the fill color (or gradient) of the slice.
- `innerLabel` - the template for the label rendered inside the slice. The format for the template is derived from the [Python `format\(\)` function](#). Properties are enclosed in curly brackets and formats can be specified using the [Python Format Specification Mini-Language](#). For example:

```
'{label}: {value:.2f}' // 'My Slice: 2.26', i.e. value is rounded to two decimal numbers.
```

See the [API reference](#) for more examples on number formatting. You can use the following properties in your template: `value`, `label`, `percentage`, `angle`, `startAngle` and `endAngle`.

- `innerLabelMargin` - the gap between the middle of the text label and the serie border in the pie. If value between `0...1` is used, it is considered to be proportional to the outer radius. If the number is bigger than `1`, it is considered to be an absolute distance in pixels.
- `legendLabel` - the template for the label for the slice used in the legend. Uses the same formatting as `innerLabel`.
- `legendLabelLineHeight` - the line height of the slice label in the legend.
- `legendLabelMargin` - the gap between the circle representing the slice color and the slice label in the legend.
- `offset` - the offset of the slice. This option is useful for “exploding” slices and makes sense only for the outermost slices of the pie. If `offset` is between `0...1` is used, it is considered to be proportional to the outer radius. If the number is bigger than `1`, it is considered to be an absolute distance in pixels.
- `onClickEffect` - the effect applied when the user clicks on the slice. This is an object with the property `type` that can be either `'enlarge'` or `'offset'`. If `'enlarge'` type is used, the object can have an additional property `scale` which is the scaling factor for the slice enlargement. If `'offset'` type is used, the object can have an additional property `offset` which is the offset of the slice.
- `onHoverEffect` - the effect applied when the user hovers with his mouse cursor over the slice. This is an object with the property `type` that can be either `'enlarge'` or `'offset'`. If `'enlarge'` type is used, the object can have an additional property `scale` which is the scaling factor for the slice enlargement. If `'offset'` type is used, the object can have an additional property `offset` which is the offset of the slice.

When the above properties are defined in the `sliceDefaults` options object, they will be applied to all the slices unless overruled by properties in the slice object.

Chart API

chart.Pie

addSlice(slice, serieIndex [, opt])	Append a slice <code>slice</code> to the serie identified by <code>serieIndex</code> . <code>opt</code> can be an arbitrary object that is passed to the <code>onchange</code> handlers of the JointJS element.
editSlice(slice, sliceIndex, serieIndex [, opt])	Replace a slice in the pie by a new <code>slice</code> . The serie is identified by <code>serieIndex</code> in which the slice at index <code>sliceIndex</code> will be replaced. <code>opt</code> can be an arbitrary object that is passed to the <code>onchange</code> handlers of the JointJS element.

chart.PieView

To access the view for a pie chart, use: `paper.findViewByModel(chart)`.

clickSlice(sliceIndex, serieIndex)	Simulate a click on the slice. Note that this triggers the effects that you defined in the <code>onClickEffect</code> in the slice object .
mouseoverSlice(sliceIndex, serieIndex)	Simulate a mouse over on the slice. Note that this triggers the effects that you defined in the <code>onHoverEffect</code> in the slice object .

chart.PieView events

To set a listener for the pie view events, use: `paper.findViewByModel(chart).on('mousemove', myFunction)`.

mousemove	Continuously triggered when the user hovers over a slice. The handler is passed <code>slice</code> and <code>event</code> objects.
mouseover	Triggered when the user hovers over a slice. The handler is passed <code>slice</code> and <code>event</code> objects.
mouseout	Triggered when the user leaves a slice. The handler is passed <code>slice</code> and <code>event</code> objects.
click	Triggered when the user clicks a slice. The handler is passed <code>slice</code> and <code>event</code> objects.

Chart SVG structure

The following is the chart SVG structure that you can take as a reference when styling chart elements.

```
<g class="background">
  <rect/>
  <text/>
</g>
<g class="data">
  <g class="slice serie-[index of the serie] slice-[index of the slice] [serie name?] [slice name?] [outer?] [hover?] [clicked?]" data-serie="[index of the serie]" data-slice="[index of the slice]" data-value="[value of the slice]">
    <path class="slice-fill"/>
```

```

        <path class="slice-border"/>
        <text class="slice-inner-label"/>
    </g>
    <!-- ... possibly more slices -->
</g>
<g class="foreground">
    <rect/>
    <text class="caption"/>
    <text class="subcaption"/>
    <g class="legend">
        <g class="legend-items">
            <g class="legend-serie [serie name?]" data-serie="[index of the
associated serie]">
                <text/>
            </g>
            <g class="legend-slice [slice name?]" data-serie="[index of the
associated serie]" data-slice="[index of the associated slice]">
                <circle/>
                <text/>
            </g>
            <!-- ... possibly more legend items -->
        </g>
    </g>
</g>

```

Selector examples

```

chart.attr('.legend-slice text/text-decoration' , 'underline') // Underline a legend
label.
chart.attr('.legend-slice text/font-size' , 13) // Set a font size of all the legend
texts for slices.
chart.attr('.slice-fill/stroke' , 'gray') // Set a stroke color of a slice.
chart.attr('.slice-fill/stroke-width' , 1) // Set a stroke width of a slice.
chart.attr('.slice-3 .slice-fill/filter', { name: 'dropShadow', args: { dx: 2, dy: 2,
blur: 3 } })

```

shapes.chart.plot

Plot

The **chart.Plot** chart allows you to create **Line**, **Bar** and **Area** charts and their combinations.

```

var graph = new joint.dia.Graph;
var paper = new joint.dia.Paper({ el: $('#paper'), width: 500, height: 500, model:

```

```
graph });

var chart = new joint.shapes.chart.Plot({
  position: { x: 50, y: 50 },
  size: { width: 300, height: 100 },
  series: [
    {
      name: 'myserie',
      data: [{ x: 1, y: 2 }, { x: 2, y: 3 }, { x: 3, y: 2 }, { x: 4, y: 3 }]
    }
  ]
});
graph.addCell(chart);
```

Series styling

All the elements of the chart can be customized using the usual JointJS approach, via the `attrs` object or the `attr()` method. This means that by just knowing the structure of the SVG elements the chart is built from, you have full flexibility in styling your chart. Following the JointJS philosophy, we don't try to invent special parameters. Instead, we try to follow SVG and CSS standards as much as we can in the way things are styled.

Let's have a look at one snippet of the SVG structure of the charts that is used to render a serie. For the full SVG structure of charts, please see the section [Chart SVG Structure](#) in this document. Once you know the structure, it is easy to make CSS selectors referencing the SVG elements that you want to style. We'll show an example of that below. the SVG structure of one serie is the following:

```
<g class="serie [name of the serie]">
  <path>
    <g class="points">
      <g class="point">
        <circle>
        <text>
      </g>
      <!-- ... possibly more points -->
    </g>
  </g>
```

Knowing that, we can, for example, set the stroke color of the serie SVG path like this:

```
var chart = new joint.shapes.chart.Plot({
  position: { x: 50, y: 50 },
  size: { width: 300, height: 100 },
  series: [
    {
      name: 'myserie',
      data: [{ x: 1, y: 2 }, { x: 2, y: 3 }, { x: 3, y: 2 }, { x: 4, y: 3 }]
    }
  ]
});
```



```

    ],
    attrs: {
      '.myserie path': {
        stroke: 'green', 'stroke-width': 3
      }
    }
  });

```

In the example above, we showed how to pass styling when creating a new chart but you can also change attributes of an existing chart:

```

chart.attr('.myserie path/stroke', 'red');

```

Note that not only you can change strokes but you can also use [other SVG presentational attributes](#). An interesting one is the `fill` attribute. By setting the `fill` attribute on our serie path, we effectively render our serie as an **Area chart**. In the following demos, we even use the JointJS gradients for the `fill` attribute making our charts even prettier:

Multiple series

The Chart plugin supports multiple series. Simply add more series to the `series` array. As you might have noticed, series objects have the `name` property. This `name` is then set as a class name on the serie SVG grouping element. This allows you to style series based on their names as we've already shown in the previous demos. In the following example, we plot multiple series and style them differently based on their names.

All series have automatically generated legends. This **legend is interactive**. You can click on the labels in the legend and turn off and on the rendering of the associated serie in the chart.

Markings

Markings (a.k.a trend lines) are vertical or horizontal lines or rectangles highlighting a certain area in the chart. You can have an unlimited number of markings in your charts. Markings are defined in the `markings` array with coordinates of the start and end of the marking.

The following is a snippet from the options passed to the chart constructor that defines four markings and styles them. For the full source code, please follow the link below the demo.

```

markings: [
  {
    name: 'target',
    start: { y: 3.2 },
    label: 'My Target'
  },
  {
    name: 'current',
    start: { x: 4 }
  },
  {
    name: 'previous',

```

```

        start: { x: 2.8 },
        end: { x: 3.2 }
    },
    {
        name: 'next',
        start: { x: 4.8, y: 3.5 },
        end: { x: 5.2, y: 2.5 }
    }
],
attrs: {
    '.marking.target rect': {
        fill: 'red'
    },
    '.marking.next rect': {
        fill: '#3498DB', rx: 1000, ry: 1000, 'fill-opacity': .6, stroke: 'black'
    },
    '.marking.previous rect': {
        fill: '#27AE60', 'fill-opacity': .8
    }
}
}

```

Guidelines

Guidelines are lines that appear when you hover over the chart with your mouse pointer. They help the user orient himself in the chart. Guidelines are disabled by default but can be very easily turned on.

Here is the snippet from the `attrs` object that enables guidelines:

```

attrs: {
    '.guideline': {
        'stroke-dasharray': '3,1', 'stroke-width': .8, display: 'block'
    },
    '.x-guideline': {
        stroke: 'red'
    },
    '.y-guideline': {
        stroke: 'green'
    }
}
}

```

Hover over the chart to see the guidelines following your mouse cursor.

Tooltips

Tooltips are a great way to show contextual information in your charts. The JointJS Chart plugin also provides a facility to get to the closest points to the mouse cursor from all the series data points. Moreover, the chart view offers the

`renderPoint()` method that makes it easy to display a point marker at the place of any data point from any serie. Let's start with a demo and then see how this can be implemented.

IMPORTANT: For tooltips to work, you must have the [ui.Tooltip](#) plugin installed.

When setting up tooltips, we no longer work with the chart model. Instead, we request the paper for the view that represents the chart. This view triggers `mouseover` and `mouseout` events that we use to hide and show our tooltip. The `mouseover` event handler gets passed three important arguments: `dataPoint`, `clientPoint` and `closestPoints`. These three arguments mean the following:

- `dataPoint` - the point in the chart area. This point is interpolated based on the minimum and maximum x and y values and the position of the mouse cursor in the chart.
- `clientPoint` - basically the mouse event `clientX` and `clientY` coordinates but transformed to the coordinate system of the chart view.
- `closestPoints` - an array of from each data serie that are the closest points to the mouse cursor. Each object in this array is of the form: `{ x: [number], y: [number], serie: [serie object] }`.

Legend

The chart legend is auto-generated and contains labels and colors of all the data series. The legend is also interactive by default and allows the user to turn on and off data series displayed in the chart. This sections shows you how you can **customize the legend and its position** and also how to turn series on and off programmatically.

The legend is rendered as an SVG group element with the class `Legend`. Inside this group, we have `Legend-item`s each containing an SVG circle and text elements. Please refer to the [Chart SVG structure](#) section for the full SVG structure of the chart. This is all you need to know to be able to style the legend. Moreover, we have prepared a syntactic sugar method (`legendPosition()`) for positioning the legend so that you don't have to deal with the low-level positioning using the [JointJS special attributes](#):

```
chart.legendPosition('nw');
```

Also, the styling of the legend item text (name of the associated serie), the legend item circle and the their look in the disabled state (when the serie is turned off) can be done the usual JointJS way:

```
chart.attr('.legend-item text/fill', 'red');
chart.attr('.legend-item circle/r', 6);
chart.attr('.legend-item circle/stroke', 'black');
chart.attr('.legend-item.disabled text/fill', 'blue');
```

Real-time data

Every change to the `series` array triggers an update of the chart view and new series are rendered. However, the Chart plugin provides two methods that are handy when dealing with real-time data: `addPoint(p, serieName, op)` and `lastPoint(serieName)`. Use `addPoint()` to add a new data point to a serie identified by `serieName`. This method automatically triggers an update and the associated view is re-rendered:

```
setInterval(function() {
  chart.addPoint({ x: chart.lastPoint('myserie').x + 1, y: Math.random() * 2 + 2 },
```

```
'myserie', { maxLen: 6 });
}, 600);
```

Fixed axis

By default, the chart plugin automatically chooses the minimum and maximum values on each of the axis based on the minimum/maximum data points. By specifying `min` and `max` values in the axis object, you can define the precise minimum and maximum values for the axis:

```
var chart = new joint.shapes.chart.Plot({
  // ...
  axis: {
    'y-axis': {
      min: -3,
      max: 10
    },
    'x-axis': {
      min: -5,
      max: 10
    }
  }
  // ...
});
```

Time series

By going through the previous sections, you already know all you need to be able to plot time series. There is nothing special about plotting time series except of being able to format the data/times on the x axis. For that, the Chart plugin allows you to define your own formatting functions for the tick labels. This means that for representing time series data, your data point x coordinates should be in the UNIX time format. Then you just provide a function that formats this timestamp into a more readable representation:

```
var chart = new joint.shapes.chart.Plot({
  // ...
  series: [
    { name: 'temp', data: [{ x: 1397560472344, y: 21.5 }, ... ] }
  ],
  axis: {
    'x-axis': {
      tickFormat: function(t) {
        var d = new Date(t);
        return (d.getMonth() + 1) + '/' + d.getDate() + '/' + d.getHours() +
        '/' + d.getMinutes()
      }
    },
    'y-axis': {
```

```

        tickFormat: function(v) { return joint.util.format.number('.1f', v) + '
dgC'}
    }
    // ...
});

```

TIP: We recommend using the great [Moment.js](#) JavaScript library for parsing, validating, manipulating and formatting dates.

Bar series

The chart plugin supports bar series. To turn a serie to a bar serie, just set the `bars` option to `true`:

```

var chart = new joint.shapes.chart.Plot({
  // ...
  series: [
    { name: 'myserie', bars: true, data: [{ x: 2, y: 20 }, ... ] }
  ]
  // ...
});

```

There is couple of recommended settings when working with bar charts, namely `padding`, `align` and `barWidth`. `padding` is not specific to bar charts but is especially useful when using this kind of chart. This is because when bars are rendered, they must be aligned to a certain value on the x axis. By default, the top-left corner of each bar is at the position of the associated `x` data point value. That means that without any padding, the last bar won't be visible (only a very tiny fraction of its left border). Therefore, we have to set at least `right` padding to make the bar visible. The `align` property of `bars` object controls the alignment of the bars with respect to the associated `x` data point value. Possible values are `'left'` (the default), `'right'` and `'middle'`. `barWidth` controls the width of the bars. If the value is a floating point number between `0` and `1`, it is used to set the final width of each bar proportionally to the calculated width. If the value is an integer higher than `1`, it is considered to be an absolute width in pixels. For a full description of the bar options, please see the [Serie options object](#), especially the `bars` object.

Bar & Line series intermixed

Bar and Line series can be rendered together as part of one chart.

```

var chart = new joint.shapes.chart.Plot({
  // ...
  series: [
    { name: 'mybars', bars: { barWidth: .7 }, data: [{ x: 1, y: 15 } /* ... */ ] },
    { name: 'myotherbars', bars: { barWidth: .7 }, data: [{ x: 1, y: 12 } /* ... */ ] },
    { name: 'myline', interpolate: 'bezier', data: [{ x: 1, y: 20 } /* ... */ ] }
  ]
});

```

```
// ...  
});
```

As you can see in the code above, we have two bar series and one line serie as part of one chart. It is important to note that if there are more than one bar serie in the chart, by default, the bars of the different series will be rendered one over another. For cases where this is not desirable, you can offset the bars of one of the series as shown in the demo below:

As you might have noticed in the demo above, **negative values** for the series **are fully supported**.

Chart options

This is the full list of the options that you can pass to the chart constructor function.

- `series` - an array of series of data points.
- `markings` - an array of markings. Markings can be both vertical or horizontal lines or rectangles with optional label text.
- `axis` - axis definition. Currently, the object can contain only `'x-axis'` and `'y-axis'` properties representing the two axes of the chart.
- `padding` - an object with `top`, `bottom`, `left`, and `right` attributes that specify the paddings between the chart contents and the chart boundaries.
- `attrs` - an array of selectors with their associated CSS attributes.

In addition, since a chart is just a normal JointJS object, you can also pass relevant Element attributes.

- `position` - an object with `x` and `y` coordinates that specify the position of the chart within the paper.
- `size` - an object with `width` and `height` dimensions that specify the size of the chart within the paper.

Series

These are items of the `series` array.

- `name` - the name of the serie. This name is added as a CSS class to the serie container (the SVG group element).
- `label` - the serie label (used in the auto-generated legend)
- `data` - an array of data points of the form `{ x: [number], y: [number] }`
- `interpolate` - (*line charts*) the interpolation method. The interpolation method can be one of:
 - `'linear'` - the default interpolation.
 - `'bezier'` - interpolation with a bezier curve.
 - `'step'` - step function
 - `'stepBefore'` - step function
 - `'stepAfter'` - step function
- `bars` - (*bar charts*) if set to `true`, the series will be rendered as a bar serie using default options. You can override the default options by instead setting `bars` to an object with the following properties (all are optional):
 - `align` - the alignment of the bars with respect to the x data point values. Possible values are `'middle'` (default), `'left'` and `'right'`.
 - `barWidth` - controls the width of the bars. If the value is a floating point number between `0` and `1`, it is used to set the final width of each bar proportionally to the calculated width. The calculated width is computed such as, for a consecutive x values on the axis, the sum of the widths of all the bars equals the

width of the chart area. If the value is an integer higher than `1`, it is considered to be an absolute width in pixels.

- `'top-ix'` - the top x radius of the bars. Together with the `top-iy` value, this allows you to define border radius for the bars.
- `'top-iy'` - the top y radius of the bars.
- `showLegend` - controls whether a legend item for the serie should be shown or not. If `false`, the legend item will be skipped. If it is a function, it should return `true` if the legend item should be shown, `false` otherwise. The function has the following signature: `function(serie, stats)`, where `serie` is the serie object and `stats` is an object that holds some interesting statistics for the serie useful for deciding whether to show the legend item or not. The statistics are: `decreasingX`, `decreasingY`, `maxX`, `maxY`, `minX`, `minY`, `nonDecreasingX` and `nonDecreasingY`.
- `legendLabelLineHeight` - controls the legend label line height. It defaults to `16`.
- `hideFillBoundaries` - (*line charts*) option to hide both fill boundaries. Default is `false`. Improves the look of line charts, but note that fill boundaries are necessary for the `'.data.mySeries path/fill'` attribute to correctly support area charts.
- `showRightFillBoundary` - (*area charts*) option to show the right fill boundary. Default is `false`.
- `fillPadding` - (*area charts*) an object whose attributes (`left`, `right` and `bottom`) determine the offsets of fill boundaries from the respective extreme data points (or the x-axis - for `bottom`). Default is `{ left: 0, right: 0, bottom: 10 }`.

Markings

These are items of the `markings` array.

- `name` - the name of the marking. This name is added as a CSS class to the marking container (the SVG group element). This makes it easier to style it.
- `label` - the marking label. This is an auto-positioned text element.
- `start` - the start coordinate of the marking. An object of the form `{ x: [number], y: [number] }`. If only `x` is given, a vertical marking is rendered at that x-coordinate. If only `y` is given, a horizontal marking is rendered at that `y` coordinate.
- `end` - the end coordinate of the marking. An object of the form `{ x: [number], y: [number] }`. If only `x` is given, a vertical marking is rendered at that x-coordinate. If only `y` is given, a horizontal marking is rendered at that `y` coordinate. If both `start` and `end` objects are defined, a rectangular marking starting at `start` coordinates ending at `end` coordinates is rendered. (TIP: if you style your marking with high `ix` and `iy` values, you can make the marking look like a circle.)
- `attrs` - additional SVG attributes that will be set on the marking container SVG element. This is sometimes useful if you want to store additional data associated with the marking in its DOM element. For example, when using tooltips.

Axis

These are properties of the `axis` object.

- `min` - the precise minimum value for the scale. Specifying `min` and `max` values for an axis prevents the chart plugin from automatically choosing the minimum and maximum values, which is the default behavior. Note that for the y-axis, the `bottom` chart padding is considered as well - to ensure that the desired min value is actually shown at the bottom of the axis, set bottom padding to `0`. See the [Fixed axis](#) section for more information.

- `max` - the precise maximum value for the axis. Note that for the y-axis, the `top` chart padding is considered as well - to ensure that the desired min value is actually shown at the top of the axis, set top padding to `0`. See the [Fixed axis](#) section for more information.
- `tickFormat` - a format of the tick labels. This can be either a string in the [Python Format Specification Mini-Language](#) or a function that takes the tick value and returns the label for that tick. Example:

```
axis: {
  'x-axis': { // floating point value with one digit after the decimal point
    tickFormat: '.1f'
  },
  'y-axis': { // currency
    tickFormat: function(v) {
      return '$' + joint.util.format.number('.2f', v)
    }
  }
}
```

- `tickSuffix` - a string or a function that will be used as a suffix of the resulting formatted tick label. If it is a function, it is passed the tick value as an argument.
- `ticks` - the number of ticks on the axis. On the y-axis, the default is `10`. On the x-axis, the default is the number of discrete x-values in the data set.
- `tickStep` - (x-axis) the modulus for tick display. Default is `1` (all ticks are displayed). Set this value to a higher number to filter out ticks. For example, setting this number to `10` makes the axis render only every 10th x-value encountered in the data set. Note that this option only works for the x-axis. This option is ignored if the axis has a `ticks` attribute.

Chart API

legendPosition(position [, opt])	Set position of the legend. <code>position</code> is a string that can be one of <code>'n'</code> , <code>'nw'</code> , <code>'w'</code> , <code>'sw'</code> , <code>'s'</code> , <code>'se'</code> , <code>'e'</code> , <code>'ne'</code> , <code>'nne'</code> , <code>'nn'</code> , <code>'nnw'</code> , <code>'nnww'</code> , <code>'nww'</code> , <code>'ww'</code> , <code>'sww'</code> , <code>'ssww'</code> , <code>'ssw'</code> , <code>'ss'</code> , <code>'sse'</code> , <code>'ssee'</code> , <code>'see'</code> , <code>'ee'</code> , <code>'nee'</code> , <code>'nnee'</code> . See the Legend section for more info. <code>opt.padding</code> can be used to set the distance between the legend and the chart edges. Default is <code>10</code> px.
addPoint(p, serieName [, opt])	Append point <code>p</code> to the serie identified by <code>serieName</code> . If <code>opt.maxLen</code> is set and the number of points in the serie is higher than <code>maxLen</code> , shift the data in the serie.
lastPoint(serieName)	Return the last point in the serie identified by <code>serieName</code> .
firstPoint(serieName)	Return the first point in the serie identified by <code>serieName</code> .

chart.PlotView

To access the view for a plot chart, use: `paper.findViewById(chart)`.

renderPoint(p, serie)	Render a point at the coordinates defined in <code>p</code> . <code>serie</code> is the <u>object</u> from the <code>chart.Plot</code> <code>series</code> array and defines the serie for which the point should be rendered. Return the SVG DOM element for the point. See the <u>Tooltips</u> section on why this method is useful.
getSerieColor(serieName)	Return the color used to draw the serie identified by <code>serieName</code> .
hideSerie(serieName)	Hide serie identified by <code>serieName</code> .
showSerie(serieName)	Show serie identified by <code>serieName</code> .
closestPoints(x)	Return an array of the closest points from all the series in the chart for a given <code>x</code> coordinate.

Selector examples

```

chart.attr('.caption/text', 'My Chart') // Set chart caption.
chart.attr('.subcaption/text', '2014') // Set chart sub caption.
chart.attr('.caption/fill', 'red') // Set chart caption text color.
chart.attr('.caption/font-size', 20) // Set chart caption font size.
chart.attr('.caption/ref-x', 10) // Move chart caption 10px from the left edge of the
chart.
chart.attr('.guideline/display', 'block') // Display guidelines.
chart.attr('.x-axis .tick line/stroke', 'green') // Set the color of the ticks on the
x axis.
chart.attr('.y-axis .tick text/fill', 'blue') // Set the color of the tick labels on
the y axis.
chart.attr('.legend-item[data-serie="myserie"]/text-decoration', 'underline') //
Underline a legend label.
chart.attr('.point/display', 'none') // Hide the data points.
chart.attr('.data .myserie .bar[data-x="5"]/fill', 'blue') // Set blue fill color for
only the bar with x value 5.

```

JointJS+ measurement

shapes.measurement

Set of shapes for displaying dimensions of objects, the distances between them, and their relative angles.

source code

shapes.measurement.Angle

A line with angle arcs drawn at its ends.

Supported `attrs` properties

Selector	Node	Description									
root	<i>SVGGElement</i>	Container of all nodes									
line	<i>SVGPathElement</i>	Visible connection of the link									
wrapper	<i>SVGPathElement</i>	Invisible wrapper around the connection to make the line thicker, so the user can interact with the link more easily.									
angles	<i>group of SVGPathElements</i>	Group of both angle arcs (<code>sourceAngle</code> and <code>targetAngle</code>). Each angle is an arc between this link and another cell (element or link). There are several custom presentation attributes, which further control the angle behavior. <table><tr><th>Property</th><th>Type</th><th>Description</th></tr><tr><td><code>angleRadius</code></td><td>number</td><td>Specifies the radius of the angle arc (in pixels). Default is <code>40</code> .</td></tr><tr><td><code>angleStart</code></td><td>string enum</td><td> Specifies the starting point of the angle arc. <code>"self"</code> (default) - The angle arc begins at this link, then moves in the specified <code>angleDirection</code> (see </td></tr></table>	Property	Type	Description	<code>angleRadius</code>	number	Specifies the radius of the angle arc (in pixels). Default is <code>40</code> .	<code>angleStart</code>	string enum	Specifies the starting point of the angle arc. <code>"self"</code> (default) - The angle arc begins at this link, then moves in the specified <code>angleDirection</code> (see
Property	Type	Description									
<code>angleRadius</code>	number	Specifies the radius of the angle arc (in pixels). Default is <code>40</code> .									
<code>angleStart</code>	string enum	Specifies the starting point of the angle arc. <code>"self"</code> (default) - The angle arc begins at this link, then moves in the specified <code>angleDirection</code> (see									

					below), and finally ends at the first encountered side of the end element (or the first encountered segment of the end link). • "source" <i>(applicable only when connected to a link)</i> - The angle arc begins at the segment of the connected link that is closest to the connected link's source, then moves in the specified <code>angleDirection</code> (see below), and finally ends at this link. Depending on <code>angleDirection</code>, this may mean that the angle arc crosses through the other ray of the connected link on its way to this link. • "target" <i>(applicable only when connected to a link)</i> - The angle arc begins at the segment of the connected link that is closest to the connected link's target, then moves in the specified <code>angleDirection</code> (see below), and finally ends at this link. Depending on <code>angleDirection</code>, this may mean that the angle arc crosses through the other segment of the connected link on its way to this link.
				angleDirection string enum	Specifies the direction of the angle arc from the starting point (see <code>angleStart</code> above). • "clockwise" - Draw the angle arc in the direction that is clockwise from <code>angleStart</code>. • "anticlockwise" - Draw the angle arc in the direction that is anticlockwise from <code>angleStart</code>.

				• "small" (<i>default</i>) - Draw the angle arc with the smaller angle of the two options ("clockwise" and "anticlockwise"). • "large" - Draw the angle arc with the larger angle of the two options ("clockwise" and "anticlockwise").
		anglePie	boolean	Should the angle be drawn as a pie (filled arc)? Default is false .
		angle	number	If specified, the angle arc is set to the given value. It will not automatically recalculate based on the position of the connected cells.

sourceAngle	SVGPathElement	Angle arc between the link and the source cell (element or link). The node can specify all <code>angles</code> custom presentation attributes; the values set here will overwrite the more generic values from <code>angles</code> .									
targetAngle	SVGPathElement	Angle arc between the link and the target cell (element or link). The node can specify all <code>angles</code> custom presentation attributes; the values set here will overwrite the more generic values from <code>angles</code> .									
angleLabels	group of SVGTextElements	Group of both angle labels (<code>sourceAngleLabel</code> and <code>targetAngleLabel</code>). Each label is a text element positioned within the angle arc; it shows the current angle value, in degrees. There are several custom presentation attributes, which further control the angle label behavior. <table><tr><th>Property</th><th>Type</th><th>Description</th></tr><tr><td>angleTextDecimalPoints</td><td>object</td><td>The number of digits to show after the decimal point of the angle value. This may be a value between 0 (default) and 20.</td></tr><tr><td>angleTextDistance</td><td>number</td><td>The distance from the vertex of the angle to the anchor of the angle label</td></tr></table>	Property	Type	Description	angleTextDecimalPoints	object	The number of digits to show after the decimal point of the angle value. This may be a value between 0 (default) and 20.	angleTextDistance	number	The distance from the vertex of the angle to the anchor of the angle label
Property	Type	Description									
angleTextDecimalPoints	object	The number of digits to show after the decimal point of the angle value. This may be a value between 0 (default) and 20.									
angleTextDistance	number	The distance from the vertex of the angle to the anchor of the angle label									

				text element. Default is 23 .
targetAngleLabel	SVGTextElement	Label of the source angle; it shows the current angle value, in degrees. The node can specify all angleLabels custom presentation attributes; the values set here will overwrite the more generic values from angleLabels.		
sourceAngleLabel	SVGTextElement	Label of the target angle; it shows the current angle value, in degrees. The node can specify all angleLabels custom presentation attributes; the values set here will overwrite the more generic values from angleLabels.		

```

var angle = new joint.shapes.measurement.Angle();
angle.attr('root/title', 'joint.shapes.measurement.Angle');
angle.attr('line/stroke', '#fe854f');
angle.attr('angleLabels/stroke', '#fe854f');
// Source Connected To An Element
angle.source(element, {
  anchor: { name: 'left' },
  connectionPoint: { name: 'anchor' }
});
angle.attr('sourceAngle/fill', 'lightgray');
angle.attr('sourceAngle/anglePie', true);
angle.attr('sourceAngleLabel/fill', '#333333');
// Target Connected To A Link
angle.target(link, {
  anchor: { name: 'connectionRatio', { args: { ratio: 0.2 } } },
});
angle.attr('targetAngle/angleStart', 'source');
angle.attr('targetAngle/angleDirection', 'anticlockwise');
angled.addTo(graph);

```

shapes.measurement.Distance

A line with a label that shows the distance between the source and the target of the link (the length of the connection path).

Supported attrs properties

Selector	Node	Description
root	SVGGElement	Container of all nodes

line	<i>SVGPathElement</i>	Visible connection of the link									
wrapper	<i>SVGPathElement</i>	Invisible wrapper around the connection to make the line thicker, so the user can interact with the link more easily.									
anchorLines	<i>group of SVGPathElements</i>	Lines between anchor points and connection points of this link.									
sourceAnchorLine	<i>SVGPathElement</i>	Line between the source anchor and the source connection point of this link.									
targetAnchorLine	<i>SVGPathElement</i>	Line between the target anchor and the target connection point of this link.									
distanceLabel	<i>SVGTextElement</i>	Label with the distance between the source and target connection point. There are several custom presentation attributes, which further control the label behavior. <table><tr><th>Property</th><th>Type</th><th>Description</th></tr><tr><td>distanceText</td><td>object</td><td> Specifies the format of the label. It may have two properties: <code>unit</code> (<i>string</i>) - The unit to show after the number. It defaults to <code>"px"</code>. <code>fixed</code> (<i>number</i>) - The number of digits to show after the decimal point. This may be a value between 0 (default) and 20. </td></tr><tr><td>labelPosition</td><td>object</td><td> Specifies the position of the label on the link. It may have two properties: <code>ratio</code> (<i>number</i>) - Move and auto-orient the label to the point at the given ratio of the link's total length. <code>offset</code> (<i>number</i>) - If the offset is a positive number, displace the label perpendicularly to the right (in the direction of the line path) in the paper local coordinate </td></tr></table>	Property	Type	Description	distanceText	object	Specifies the format of the label. It may have two properties: <code>unit</code> (<i>string</i>) - The unit to show after the number. It defaults to <code>"px"</code>. <code>fixed</code> (<i>number</i>) - The number of digits to show after the decimal point. This may be a value between 0 (default) and 20.	labelPosition	object	Specifies the position of the label on the link. It may have two properties: <code>ratio</code> (<i>number</i>) - Move and auto-orient the label to the point at the given ratio of the link's total length. <code>offset</code> (<i>number</i>) - If the offset is a positive number, displace the label perpendicularly to the right (in the direction of the line path) in the paper local coordinate
Property	Type	Description									
distanceText	object	Specifies the format of the label. It may have two properties: <code>unit</code> (<i>string</i>) - The unit to show after the number. It defaults to <code>"px"</code>. <code>fixed</code> (<i>number</i>) - The number of digits to show after the decimal point. This may be a value between 0 (default) and 20.									
labelPosition	object	Specifies the position of the label on the link. It may have two properties: <code>ratio</code> (<i>number</i>) - Move and auto-orient the label to the point at the given ratio of the link's total length. <code>offset</code> (<i>number</i>) - If the offset is a positive number, displace the label perpendicularly to the right (in the direction of the line path) in the paper local coordinate									

system. (An offset of 0 is the default; it means no offset in either direction.)

- If the offset is a negative number, displace the label perpendicularly to the left (in the direction of the line path) in the paper local coordinate system.

```
var distance = new joint.shapes.measurement.Distance();
distance.attr('root/title', 'joint.shapes.measurement.Distance');
distance.attr('line/stroke', '#fe854f');
distance.attr('distanceLabel/stroke', '#fe854f');
distance.attr('anchorLines/strokeDasharray', '1,1');
distance.source(el2, {
  anchor: { name: 'bottomRight' },
  connectionPoint: { name: 'anchor', args: { align: 'right', alignOffset: 30 }}
});
distance.target(el1, {
  anchor: { name: 'topRight' },
  connectionPoint: { name: 'anchor', args: { align: 'right', alignOffset: 30 }}
});
distance.addTo(graph);
```

JointJS+ standard

shapes.standard

The Standard shapes plugin provides extra shapes with advanced functionality to extend the [JointJS Standard shapes library](#). The shapes can be used as they are or can serve as an inspiration for creating custom shapes.

source code

shapes.standard.BorderedRecord

An object that displays structured/hierarchical data. Inherits from `joint.standard.Record` [shape](#) and adds a background `body` rectangle.

Additional `attrs` properties

Selector	Node	Description
<code>body</code>	<i>SVGRectElement</i>	Rectangular body of the shape

To adjust the appearance of a BorderedRecord element, use the `Element.attr` [function](#):

```
var borderedRecord = new joint.shapes.standard.BorderedRecord();
borderedRecord.resize(150, 100);
borderedRecord.position(25, 610);
borderedRecord.attr('root/magnet', false);
borderedRecord.attr('body', { fill: 'white', strokeWidth: 2 });
borderedRecord.attr('buttonsGroups/stroke', 'black');
borderedRecord.attr('forksGroups/stroke', 'lightgray');
borderedRecord.attr('itemBodies/stroke', 'black');
borderedRecord.attr('itemBodies_0/fill', '#feb663');
borderedRecord.attr('itemLabels', { fontSize: 12, fontFamily: 'sans-serif' });
borderedRecord.attr('itemLabels_1/magnet', true);
borderedRecord.addTo(graph);
```

Custom presentation attributes:

All custom presentation attributes are inherited from the `joint.standard.Record` [shape](#).


```

var borderedRecord = new joint.shapes.standard.BorderedRecord();
borderedRecord.attr('itemBodies/itemHighlight', { fill: 'lightgray', stroke: 'gray'
});
borderedRecord.attr('itemLabels/itemHighlight', { fill: 'orange' });
if (borderedRecord.isItemHighlighted('my_item_id') === false) {
    // `my_item_id` item' background switches to gray and label to orange.
    borderedRecord.toggleItemHighlight('my_item_id');
}
borderedRecord.attr('itemLabels/itemText', { textWrap: true, ellipsis: '*' });

```

Configuration properties

All configuration properties are inherited from the `joint.standard.Record` [shape](#). They are set during shape definition with the `Cell.define` [function](#), or anytime later with the `Cell.set` function:

```

borderedRecord.set('itemHeight', 25);
borderedRecord.set('padding', { top: 10, right: 20, bottom: 5, left: 10 });
borderedRecord.set('items', [
    [{ id: 'value1', label: 'Hello' }]
]);
borderedRecord.set('items', [
    [{ id: 'value1', label: 'Hello', items: [
        { id: 'value2', label: 'World!' }]
    }]
]);

```

Functions

All methods are inherited from the `joint.standard.Record` [shape](#).

shapes.standard.HeaderedRecord

An object that displays structured/hierarchical data. Inherits from `joint.standard.Record` [shape](#) and adds a background `body` rectangle, as well as a `header` rectangle and `headerLabel` text label.

Additional attrs properties

Selector	Node	Description
body	<i>SVGRectElement</i>	Rectangular body of the shape
header	<i>SVGRectElement</i>	Rectangular header of the shape, positioned above shape's body
headerLabel	<i>SVGTextElement</i>	Text label inside the header

To adjust the appearance of a HeaderedRecord element, use the `Element.attr` [function](#):

```
var headeredRecord = new joint.shapes.standard.HeaderedRecord();
headeredRecord.resize(150, 100);
headeredRecord.position(25, 610);
headeredRecord.attr('root/magnet', false);
headeredRecord.attr('body', { fill: 'white', strokeWidth: 2 });
headeredRecord.attr('header', { fill: 'cornflowerblue', strokeWidth: 2 });
headeredRecord.attr('headerLabel', { fontSize: 24, fontFamily: 'serif' });
headeredRecord.attr('buttonsGroups/stroke', 'black');
headeredRecord.attr('forksGroups/stroke', 'lightgray');
headeredRecord.attr('itemBodies/stroke', 'black');
headeredRecord.attr('itemBodies_0/fill', '#feb663');
headeredRecord.attr('itemLabels', { fontSize: 12, fontFamily: 'sans-serif' });
headeredRecord.attr('itemLabels_1/magnet', true);
headeredRecord.addTo(graph);
```

Custom presentation attributes:

All custom presentation attributes are inherited from the `joint.standard.Record` [shape](#).

```
var headeredRecord = new joint.shapes.standard.HeaderedRecord();
headeredRecord.attr('itemBodies/itemHighlight', { fill: 'lightgray', stroke: 'gray'
});
headeredRecord.attr('itemLabels/itemHighlight', { fill: 'orange' });
if (headeredRecord.isItemHighlighted('my_item_id') === false) {
    // `my_item_id` item' background switches to gray and label to orange.
    headeredRecord.toggleItemHighlight('my_item_id');
}
headeredRecord.attr('itemLabels/itemText', { textWrap: true, ellipsis: '*' });
```

Configuration properties

All configuration properties are inherited from the `joint.standard.Record` [shape](#). They are set during shape definition with the `Cell.define` [function](#), or anytime later with the `Cell.set` function:

```
headeredRecord.set('itemHeight', 25);
headeredRecord.set('padding', { top: 10, right: 20, bottom: 5, left: 10 });
headeredRecord.set('items', [
    [{ id: 'value1', label: 'Hello' }]
]);
headeredRecord.set('items', [
    [{ id: 'value1', label: 'Hello', items: [
        { id: 'value2', label: 'World!' }
    ]}
]);
```

Functions

All methods are inherited from the `joint.standard.Record` [shape](#).

shapes.standard.Record

An object that displays structured/hierarchical data.

If you are extending the behaviour of `Record` shapes, and creating custom element definitions, it's necessary to use the [RecordView](#).

Supported `attrs` properties

Selector	Node	Description
<code>root</code>	<i>SVGGElement</i>	Container of all nodes
<code>itemBodies</code>	<i>group of SVGRectElements</i>	All item bodies across all columns. The node(s) can have a custom attribute itemHighlight .
<code>itemBodies_[group_index]</code> <code>itemBodies_[group_name]</code>	<i>group of SVGRectElements</i>	Item bodies in the given group (column). <i>For example, item bodies in the first column are accessed as <code>itemBodies_0</code>.</i> <i>Alternatively, item bodies in a group named <code>headers</code> are accessed as <code>itemBodies_headers</code>.</i> The node(s) can have a custom attribute itemHighlight .
<code>itemBody_[item_id]</code>	<i>SVGRectElement</i>	Body of the given item. <i>For example, item body for an item with id <code>my_item</code> is accessed as <code>itemBody_my_item</code>.</i> The node can have a custom attribute itemHighlight .
<code>itemLabels</code>	<i>group of SVGTextElements</i>	All item labels across all columns The node(s) can have custom attributes itemHighlight and itemText .
<code>itemLabels_[group_index]</code> <code>itemLabels_[group_name]</code>	<i>group of SVGTextElements</i>	Item labels in the given group (column). <i>For example, item labels in the first column are accessed as <code>itemLabels_0</code>.</i>

		Alternatively, item labels in a group named <code>headers</code> are accessed as <code>itemLabels_headers</code>. The node(s) can have custom attributes <u><code>itemHighlight</code></u> and <u><code>itemText</code></u>.
<code>itemLabel_[item_id]</code>	<i>SVGTextElement</i>	Label of the given item. For example, item label for an item with id <code>my_item</code> is accessed as <code>itemLabel_my_item</code>. The node can have custom attributes <u><code>itemHighlight</code></u> and <u><code>itemText</code></u>.
<code>itemIcons</code>	<i>group of SVGImageElements</i>	All item icons across all columns
<code>itemIcons_[group_index]</code> <code>itemIcons_[group_name]</code>	<i>group of SVGImageElements</i>	Item icons in the given group (column). For example, item icons in the first column are accessed as <code>itemIcons_0</code>. Alternatively, item icons in a group named <code>alerts</code> are accessed as <code>itemIcons_alerts</code>.
<code>itemIcon_[item_id]</code>	<i>SVGImageElement</i>	Icon of the given item. For example, item icon for an item with id <code>my_item</code> is accessed as <code>itemIcon_my_item</code>.
<code>bodiesGroups</code>	<i>group of SVGGElements</i>	All groups of item bodies
<code>labelsGroups</code>	<i>group of SVGGElements</i>	All groups of item labels
<code>forksGroups</code>	<i>group of SVGGElements</i>	All groups of hierarchy fork lines
<code>buttonsGroups</code>	<i>group of SVGGElements</i>	All groups of collapse buttons
<code>iconsGroups</code>	<i>group of SVGGElements</i>	All groups of item icons

To adjust the appearance of a Record element, use the `Element.attr` [function](#):

```

var record = new joint.shapes.standard.Record();
record.resize(150, 100);
record.position(25, 610);
record.attr('root/magnet', false);
record.attr('buttonsGroups/stroke', 'black');
record.attr('forksGroups/stroke', 'lightgray');
record.attr('itemBodies/stroke', 'black');
record.attr('itemBodies_0/fill', '#feb663');
record.attr('itemLabels', { fontSize: 12, fontFamily: 'sans-serif' });
record.attr('itemLabels_1/magnet', true);
record.addTo(graph);

```

Custom presentation attributes:

Attribute	Selectors	Description												
itemHighlight	<i>itemBodies</i> <i>itemBodies_[group_index]</i> <i>itemBodies_[group_name]</i> <i>itemBody_[item_id]</i> <i>itemLabels</i> <i>itemLabels_[group_index]</i> <i>itemLabels_[group_name]</i> <i>itemLabel_[item_id]</i>	Object with SVG attributes. Applied on the selected node(s) when the item is highlighted.												
itemText	<i>itemLabels</i> <i>itemLabels_[group_index]</i> <i>itemLabels_[group_name]</i> <i>itemLabel_[item_id]</i>	An object with two properties that modify the behavior of the selected SVGTextNode(s): <table> <tr> <th>Property</th><th>Type</th><th>Description</th></tr> <tr> <td>textWrap</td><td>boolean</td><td>Enable text wrapping of item label (true by default)</td></tr> <tr> <td>ellipsis</td><td>boolean string</td><td>Enable label ellipsis (true by default)</td></tr> <tr> <td>maxLineCount</td><td>number</td><td>A number to limit the maximum number of lines.</td></tr> </table>	Property	Type	Description	textWrap	boolean	Enable text wrapping of item label (true by default)	ellipsis	boolean string	Enable label ellipsis (true by default)	maxLineCount	number	A number to limit the maximum number of lines.
Property	Type	Description												
textWrap	boolean	Enable text wrapping of item label (true by default)												
ellipsis	boolean string	Enable label ellipsis (true by default)												
maxLineCount	number	A number to limit the maximum number of lines.												

```

var record = new joint.shapes.standard.Record();
record.attr('itemBodies/itemHighlight', { fill: 'lightgray', stroke: 'gray' });
record.attr('itemLabels/itemHighlight', { fill: 'orange' });
if (record.isItemHighlighted('my_item_id') === false) {

```

```
// `my_item_id` item' background switches to gray and label to orange.
record.toggleItemHighlight('my_item_id');
}
record.attr('itemLabels/itemText', { textWrap: true, ellipsis: '*' });
```

Configuration properties

Property	Type	Description																											
items	<i>object[]</i> []	Items in the Record. Index in outer array specifies the column, index in inner array specifies the order within the column. Each item may have several optional properties: <table> <tr> <th>Property</th><th>Type</th><th>Description</th></tr> <tr> <td>id</td><td>string</td><td> (Required. Must be unique.) Item's identifier in the DOM and in JointJS. Use the <code>cellView.findAttribute('item-id', id)</code> <i>function</i> to find an item by given <i>id</i>. </td></tr> <tr> <td>label</td><td>string</td><td>Text content of the item's label</td></tr> <tr> <td>icon</td><td>string</td><td>Path to item's icon</td></tr> <tr> <td>collapsed</td><td>boolean</td><td>Is the item collapsed?</td></tr> <tr> <td>highlighted</td><td>boolean</td><td>Is the item highlighted?</td></tr> <tr> <td>height</td><td>number</td><td>Height of the item</td></tr> <tr> <td>span</td><td>number</td><td>How many columns does this item span across?</td></tr> <tr> <td>group</td><td>string string[]</td><td> <u>Selector</u> postfix adding the item into a group with the given name. An item may be added to multiple groups by providing multiple group names in a string array. For example, passing <code>'headers'</code> as the group name would make the item accessible via the </td></tr> </table>	Property	Type	Description	id	string	(Required. Must be unique.) Item's identifier in the DOM and in JointJS. Use the <code>cellView.findAttribute('item-id', id)</code> <i>function</i> to find an item by given <i>id</i>.	label	string	Text content of the item's label	icon	string	Path to item's icon	collapsed	boolean	Is the item collapsed?	highlighted	boolean	Is the item highlighted?	height	number	Height of the item	span	number	How many columns does this item span across?	group	string string[]	<u>Selector</u> postfix adding the item into a group with the given name. An item may be added to multiple groups by providing multiple group names in a string array. For example, passing <code>'headers'</code> as the group name would make the item accessible via the
Property	Type	Description																											
id	string	(Required. Must be unique.) Item's identifier in the DOM and in JointJS. Use the <code>cellView.findAttribute('item-id', id)</code> <i>function</i> to find an item by given <i>id</i>.																											
label	string	Text content of the item's label																											
icon	string	Path to item's icon																											
collapsed	boolean	Is the item collapsed?																											
highlighted	boolean	Is the item highlighted?																											
height	number	Height of the item																											
span	number	How many columns does this item span across?																											
group	string string[]	<u>Selector</u> postfix adding the item into a group with the given name. An item may be added to multiple groups by providing multiple group names in a string array. For example, passing <code>'headers'</code> as the group name would make the item accessible via the																											

		<table> <tr> <td></td><td></td><td>itemBodies_headers and itemLabels_headers selectors.</td></tr> <tr> <td>items</td><td>object[]</td><td>Nested items</td></tr> </table>			itemBodies_headers and itemLabels_headers selectors.	items	object[]	Nested items
		itemBodies_headers and itemLabels_headers selectors.						
items	object[]	Nested items						
itemHeight	number	Height of all items						
itemOffset	number	Offset of nested items from parent item						
itemMinLabelWidth	number	Ensure minimum width of item labels						
itemButtonSize	number	Size of collapse buttons						
itemIcon	object	May specify width, height, and padding of item icons						
itemOverflow	boolean	Should item cells be stretched horizontally when model is widened? i.e. Make items overlap the left and right padding.						
itemAboveViewSelector	string	If an item is not in view (it's scrolled out above the record) into which SVGElement (determined by selector) should the links be visually reconnected.						
itemBelowViewSelector	string	If an item is not in view (it's scrolled out below the record) into which SVGElement (determined by selector) should the links be visually reconnected.						
padding	object	Padding within the Record object						
scrollTop	number null	The number of pixels that a record's content (without the padding) is scrolled vertically. A record's scrollTop value is a measurement of the distance from the content's top to its topmost visible content. To enable scrolling set the value to a number. If the value is <code>null</code> (default), the record is automatically resized to the <u>minimal size</u> effectively preventing the need for scrolling. <pre>new Record({ scrollTop: 0 });</pre>						

Configuration properties are set during shape definition with the `Cell.define` [function](#), or anytime later with the `Cell.set` function:

```
record.set('itemHeight', 25);
record.set('padding', { top: 10, right: 20, bottom: 5, left: 10 });
```

```

record.set('items', [
  [{ id: 'value1', label: 'Hello' }]
]);
record.set('items', [
  [{ id: 'value1', label: 'Hello', items: [
    { id: 'value2', label: 'World!' }]}
  ]
]);

```

Functions

The Record shape exposes several functions that enable working with the embedded hierarchical data (items):

shapes.standard.Record.prototype.addItemAtIndex

```
record.addItemAtIndex(parentId, index, item [, opt])
```

Insert the provided `item` into the Record as a child of `parentId` (string) at given `index`.

```
record.addItemAtIndex(groupIndex, index, item [, opt])
```

Insert the provided `item` into the group (column) with provided `groupIndex` (number) at given `index`.

shapes.standard.Record.prototype.addNextSibling

```
record.addNextSibling(siblingId, item [, opt])
```

Insert the provided `item` into the Record after the item identified by `siblingId`.

shapes.standard.Record.prototype.addPrevSibling

```
record.addPrevSibling(siblingId, item [, opt])
```

Insert the provided `item` into the Record before the item identified by `siblingId`.

shapes.standard.Record.prototype.getItemGroupIndex

```
record.getItemGroupIndex(itemId)
```


Return the group (column) index of the item with the provided `itemId` . Return `null` if the item does not exist.

shapes.standard.Record.prototype.getItemParentId

```
record.getItemParentId(itemId)
```

Return the itemId of the parent of the item with the provided `itemId` . Return `null` if the item does not exist.

shapes.standard.Record.prototype.getItemPathArray

```
record.getItemPathArray(itemId)
```

Return the path to the item with the provided `itemId` , as an array of strings. Return `null` if the item does not exist.

shapes.standard.Record.prototype.getItemSide

```
record.getItemSide(itemId)
```

Return the side on which lies the item with the provided `itemId` .

Return `'left'` if the item lies within the leftmost column of the Record. Return `'right'` if the item lies within the rightmost column of the Record. Return `'middle'` if the item does not touch either side of the Record - or if there is only one column. Return `null` if the item does not exist.

shapes.standard.Record.prototype.getPadding

```
record.getPadding()
```

Return the normalized padding in the form of { `top`: Number, `right`: Number, `bottom`: Number, `left`: Number } .

shapes.standard.Record.prototype.getMinimalSize

```
record.getMinimalSize()
```

Return the width and height of the minimal bounding box necessary to encompass all of the shape's content.

shapes.standard.Record.prototype.isItemCollapsed

```
record.isItemCollapsed(itemId)
```

Return **true** if the item is collapsed. Return **null** if the item does not exist.

shapes.standard.Record.prototype.isItemHighlighted

```
record.isItemHighlighted(itemId)
```

Return **true** if the item is highlighted. Return **null** if the item does not exist.

shapes.standard.Record.prototype.isItemVisible

```
record.isItemVisible(itemId)
```

Return **true** if the item is visible (i.e. it is not a child of a collapsed item). Return **null** if the item does not exist.

shapes.standard.Record.prototype.item

```
record.item(itemId)
```

Return the item with the provided `itemId`. Return **null** if the item does not exist.

```
record.item(itemId, item [, opt])
```

Add/merge the provided `item` properties into the Record for the given `itemId`.

shapes.standard.Record.prototype.removeItem

```
record.removeItem(itemId)
```

Remove the item with the provided `itemId` and all its children. Return the removed item, or `null` if the item does not exist.

shapes.standard.Record.prototype.toggleItemCollapse

```
record.toggleItemCollapse(itemId [, opt])
```

Collapse/expand the item with the provided `itemId`.

shapes.standard.Record.prototype.toggleItemHighlight

```
record.toggleItemHighlight(itemId [, opt])
```

Highlight/unhighlight the item with the provided `itemId`.

shapes.standard.Record.prototype.getItemBBox

```
record.getItemBBox(itemId)
```

Return the bounding box (g.Rect) of the item in the element's coordinate system. Returns `null` if no item with such an id exists.

shapes.standard.Record.prototype.getItemViewSign

```
record.getItemViewSign(itemId)
```

For the item with the provided `itemId`, the method is taking the current `scrollTop` value into account and

- return `0` if the item is visible in view.
- return `-1` if the item is scrolled out above the view.
- return `1` if the item is scrolled out below the view.

The method throws an error if the item is hidden within a collapsed ancestor.

shapes.standard.Record.prototype.isItemInView

```
record.isItemInView(itemId)
```

Is the item with the provided `itemId` visible in view and not scrolled out?

The method throws an error if the item is hidden within a collapsed ancestor.

shapes.standard.Record.prototype.isEveryItemInView

```
record.isEveryItemInView()
```

Are all the items visible in view?

shapes.standard.Record.prototype.getScrollTop

```
record.getScrollTop()
```

Return the normalized value of model's `scrollTop` property. It is the current number of pixels the record's content is scrolled vertically.

shapes.standard.Record.prototype.setScrollTop

```
record.setScrollTop(number, [opt])
```

Normalize and set the `scrollTop` property of the model.

shapes.standard.RecordView

The view for `shapes.standard.Record`. It inherits from `joint.dia.ElementView`. If you are extending the behaviour of `Record` shapes, and creating custom element definitions, it's necessary to use the `RecordView`. The `RecordView` is responsible for interactivity among other things, so it ensures your custom `Record` behaves as expected.

```
class CustomRecord extends shapes.standard.HeaderedRecord {
  defaults() {
    return util.defaultsDeep({
      type: 'customNamespace.CustomRecord',
      attrs: {
```

```

        // attributes
    }
    }, super.defaults);
}

preinitialize() {
    this.markup = util.svg/* xml */ `
        // markup
    `;
}
}

const CustomRecordView = shapes.standard.RecordView;

Object.assign(shapes, {
    customNamespace: {
        CustomRecord,
        CustomRecordView
    }
});

```

JointJS+ vsm

shapes.vsm

The VSM plugin provides you with a set of **Value Stream Mapping** shapes. The VSM plugin is an independent module, and needs to be installed as an addition to the main JointJS+ package.

VSM Demo source code

Installation

Place the VSM plugin installable package in your project. The following `package.json` example places the package in the root of the project. Next, run `npm install` to install the dependency.

```
{
  "dependencies": {
    "@joint/vsm-shapes": "file:joint-vsm-shapes.tgz", // path to file
  }
}
```

Usage

```
import * as VSMShapes from '@joint/vsm-shapes';
import { dia, shapes as defaultShapes } from 'jointjs';

const shapes = { ...VSMShapes, ...defaultShapes };

const graph = new dia.Graph({}, { cellNamespace: shapes });

const paper = new dia.Paper({
  model: graph,
  el: document.getElementById('app') as HTMLDivElement,
  width: 300,
  height: 300,
  cellViewNamespace: shapes
});

graph.fromJSON({ cells: [
  { type: "VSMTruck" },
  { type: "standard.Rectangle", size: { width: 100, height: 20 }, position: { x:
```

```
10, y: 120 } }  
  ]});
```

The `type` attribute of each vsm shape is prefixed with `VSM`, so remember to include this when creating an instance.

```
const truck = new VSMTruck();  
truck.addTo(graph);  
  
const operator = new VSMOperator();  
operator.addTo(graph);
```

shapes.vsm.CustomerSupplier

The CustomerSupplier shape can represent both the customer or the supplier depending on its placement in a value stream map. The input for the value stream is the supplier, and the endpoint of the value stream is the customer.

The supplier is denoted if the shape is placed at the upper-left of the diagram, and upper-right placement denotes the customer.

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
body	<i>SVGPathElement</i>	Path body of shape
label	<i>SVGTextElement</i>	Shape text

shapes.vsm.DataBox

The DataBox shape represents the significant data in a value stream map. Placement of the DataBox under other icons provides the additional information required to analyze the system.

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes

body	<i>SVGRectElement</i>	Rectangle body of shape
boxes	<i>SVGRectElement</i>	Selector for all rect boxes
labels	<i>SVGTextElement</i>	Selector for all box labels

API

setLabelAttr	<i>function</i>	Function to set the label attributes of a box. The function parameters are:		
		index	<i>number</i>	The number by index of the box label to target.
		attrs	<i>attributes.SVGTextAttributes</i>	Text attributes for the label.
		opt?	<i>dia.Cell.Options</i>	Cell options.
setBoxAttr	<i>function</i>	Function to set the box attributes of a box. The function parameters are:		
		index	<i>number</i>	The number by index of the box to target.
		attrs	<i>attributes.SVGRectAttributes</i>	Rect attributes for the box.
		opt?	<i>dia.Cell.Options</i>	Cell options.

Usage



```
// Set the label text of the first box to be 'Data Box'
dataBox.setLabelAttr(0, { text: 'Data Box' });

// Set the amount of boxes to 4
dataBox.set('count', 4);

// Set the fill color of the first box
dataBox.setBoxAttr(0, { fill: 'lightgray' });
```


shapes.vsm.DedicatedProcess

The DedicatedProcess shape represents fixed and continuous internal material flow through a department, process, operation, or machine in a value stream map.

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
body	<i>SVGRectElement</i>	Rectangle body of shape
header	<i>SVGRectElement</i>	Rect header of shape
label	<i>SVGTextElement</i>	Shape text

shapes.vsm.ElectronicInformationFlow

The ElectronicInformationFlow shape shows a digital information flow such as the internet in a value stream map.

Supported `attrs` properties:

Selector	Node	Description
line	<i>SVGPathElement</i>	Visible connection of the link
wrapper	<i>SVGPathElement</i>	An invisible, interactive wrapper around the connection used to make clicking on the link easier

shapes.vsm.FIFOLane

The FIFOLane shape shows a First-in-First-Out inventory system in a value stream map. The max capacity can be written underneath the icon.

Supported `attrs` properties:

Selector	Node	Description	GroupSelector
root	<i>SVGGElement</i>	Container for all shape nodes	
body	<i>SVGRectElement</i>	Rectangle body of shape	
outline	<i>SVGPathElement</i>	Path outline of shape	
rectIcon	<i>SVGRectElement</i>	Rectangle icon of shape	icons
triangleIcon	<i>SVGPathElement</i>	Triangle icon of shape	icons
ellipseIcon	<i>SVGEllipseElement</i>	Ellipse icon of shape	icons
label	<i>SVGTextElement</i>	Shape text	

shapes.vsm.GoSee

The GoSee shape indicates when information is being collected through physical observation in a value stream map. For example, information collected by a supervisor.

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
body	<i>SVGPathElement</i>	Path body of shape
label	<i>SVGTextElement</i>	Shape text

shapes.vsm.InformationFlow

The InformationFlow shape represents a general information flow in a value stream map.

Supported `attrs` properties:

Selector	Node	Description

line	<i>SVGPathElement</i>	Visible connection of the link
outline	<i>SVGPathElement</i>	Path outline of shape

shapes.vsm.KaizenBurst

The KaizenBurst shape represents the problem areas in a value stream map.

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
body	<i>SVGPathElement</i>	Path body of shape
label	<i>SVGTextElement</i>	Shape text

shapes.vsm.KanbanPost

The KanbanPost shape is used to show the location for collecting Kanban signals in a value stream map. It is usually placed near a supermarket.

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
body	<i>SVGPathElement</i>	Path body of shape
label	<i>SVGTextElement</i>	Shape text

shapes.vsm.LoadLevelling

The LoadLevelling shape can batch Kanbans over time in a value stream map. It levels out the volume.

Supported `attrs` properties:

Selector	Node	Description	GroupSelector
root	<i>SVGGElement</i>	Container for all shape nodes	
body	<i>SVGRectElement</i>	Rectangle body of shape	
outline	<i>SVGPathElement</i>	Path outline of shape	
cross1Icon	<i>SVGPathElement</i>	Path icon of shape	icons
cross2Icon	<i>SVGPathElement</i>	Path icon of shape	icons
circle1Icon	<i>SVGCircleElement</i>	Circle icon of shape	icons
circle2Icon	<i>SVGCircleElement</i>	Circle icon of shape	icons
label	<i>SVGTextElement</i>	Shape text	

shapes.vsm.ManualInfo

The ManualInfo shape represents manual information flow in a value stream map. For example, discussions, memos, reports, etc.

Supported `attrs` properties:

Selector	Node	Description
line	<i>SVGPathElement</i>	Visible connection of the link
wrapper	<i>SVGPathElement</i>	An invisible, interactive wrapper around the connection used to make clicking on the link easier

shapes.vsm.MaterialBatchKanban

The MaterialBatchKanban shape represents Kanban cards arriving or being sent in batches in a value stream map.

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
body	<i>SVGRectElement</i>	Rectangle body of shape
label	<i>SVGTextElement</i>	Shape text

shapes.vsm.MaterialFlow

The MaterialFlow shape represents a material flow in a value stream map.

Supported attrs properties:

Selector	Node	Description
line	<i>SVGPathElement</i>	Visible connection of the link
outline	<i>SVGPathElement</i>	Path outline of shape
material	<i>SVGPathElement</i>	Path material of shape

shapes.vsm.MaterialKanban

The MaterialKanban shape represents a Kanban card arriving or being sent in a value stream map.

Supported attrs properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
body	<i>SVGPathElement</i>	Path body of shape
label	<i>SVGTextElement</i>	Shape text

shapes.vsm.MaterialPull

The MaterialPull shape is used when supermarkets connect to a downstream process in a value stream map. The objective is to remove the materials from the supermarket physically.

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
body	<i>SVGEllipseElement</i>	Ellipse body of shape
arrow	<i>SVGPathElement</i>	Path arrow of shape
label	<i>SVGTextElement</i>	Shape text

shapes.vsm.Operator

The Operator shape represents a person or worker in a value stream map. The amount of operators can also be noted in the ellipse.

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
body	<i>SVGEllipseElement</i>	Ellipse body of shape
cap	<i>SVGPathElement</i>	Path cap of shape
label	<i>SVGTextElement</i>	Shape text

shapes.vsm.ProductionBatchKanban

The ProductionBatchKanban shape represents the production needed in batches to supply parts to customers in a value stream map.

Supported `attrs` properties:

Selector	Node	Description	GroupSelector
root	<i>SVGGElement</i>	Container for all shape nodes	
bodyBottom	<i>SVGPathElement</i>	Path bottom of shape	bodies
bodyMiddle	<i>SVGPathElement</i>	Path middle of shape	bodies
body	<i>SVGPathElement</i>	Path body of shape	bodies
label	<i>SVGTextElement</i>	Shape text	

shapes.vsm.ProductionControl

The ProductionControl shape represents a central production scheduling or control department in a value stream map.

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
body	<i>SVGRectElement</i>	Rectangle body of shape
label	<i>SVGTextElement</i>	Shape text

shapes.vsm.ProductionKanban

The ProductionKanban shape represents the production needed to supply parts to customers in a value stream map.

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes

body	<i>SVGPathElement</i>	Path body of shape
label	<i>SVGTextElement</i>	Shape text

shapes.vsm.PullArrow

The PullArrow shape represents physical removal of stored inventory from supermarkets in a value stream map.

Supported `attrs` properties:

Selector	Node	Description
wrapper	<i>SVGPathElement</i>	An invisible, interactive wrapper around the connection used to make clicking on the link easier
line	<i>SVGPathElement</i>	Visible connection of the link

shapes.vsm.ResourcePlanning

The ResourcePlanning shape represents an inventory scheduling symbol in a value stream map.

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
body	<i>SVGPathElement</i>	Path body of shape
top	<i>SVGEllipseElement</i>	Ellipse top of shape
label	<i>SVGTextElement</i>	Shape text

shapes.vsm.RoundedInventory

The RoundedInventory shape can represent basic inventory between 2 processes and stored inventory in a value stream map. Additional details such as the count can also be placed underneath the symbol.

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
body	<i>SVGPathElement</i>	Path body of shape
icon	<i>SVGPathElement</i>	Icon of shape
label	<i>SVGTextElement</i>	Shape text

shapes.vsm.SafetyStock

The SafetyStock shape represents temporary safety stock that is used to prevent shortages in the case of system failure in a value stream map.

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
body	<i>SVGPathElement</i>	Path body of shape
label	<i>SVGTextElement</i>	Shape text

shapes.vsm.SequencePullBall

The SequencePullBall shape represents a pull system that offers instructions to processes to manufacture a required product in a value stream map. The objective is to eliminate the need for supermarket storage.

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
outer	<i>SVGEllipseElement</i>	Outer ellipse of shape

inner	<i>SVGEllipseElement</i>	Inner ellipse of shape
label	<i>SVGTextElement</i>	Shape text

shapes.vsm.SharedProcess

The SharedProcess shape represents a process, department, operation, or work center shared by other value stream families in a value stream map.

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
body	<i>SVGRectElement</i>	Rectangle body of shape
header	<i>SVGRectElement</i>	Rect header of shape
label	<i>SVGTextElement</i>	Shape text

shapes.vsm.Shipment

The Shipment shape shows the transport of materials from the supplier to the customer in a value stream map.

Supported `attrs` properties:

Selector	Node	Description
line	<i>SVGPathElement</i>	Visible connection of the link

shapes.vsm.SignalKanban

The SignalKanban shape represents when a reorder point is reached, and another batch of parts must be produced in a value stream map.

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
body	<i>SVGPathElement</i>	Path body of shape
icon	<i>SVGTextElement</i>	Icon text
label	<i>SVGTextElement</i>	Shape text

shapes.vsm.Subprocess

The Subprocess shape represents a manufacturing subprocess in a value stream map.

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
body	<i>SVGRectElement</i>	Rectangle body of shape
stripes	<i>SVGPathElement</i>	Path stripes of shape
label	<i>SVGTextElement</i>	Shape text

shapes.vsm.Supermarket

The Supermarket shape represents the end point of a kanban loop in a value stream map. It is also known as a Kanban stock point. It shows where downstream customers can get the Inventory they need as the supplier provides it.

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
body	<i>SVGPathElement</i>	Path body of shape

label	<i>SVGTextElement</i>	Shape text

shapes.vsm.SupermarketParts

The SupermarketParts shape represents the parts of a supermarket in a value stream map.

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
body	<i>SVGRectElement</i>	Rectangle body of shape
label	<i>SVGTextElement</i>	Shape text

shapes.vsm.TimelineProcessing

The TimelineWaiting shape shows processing times in a value stream map.

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
line	<i>SVGPathElement</i>	Path line of shape
label	<i>SVGTextElement</i>	Shape text

shapes.vsm.TimelineTotal

The TimelineTotal shape shows waiting times and processing times in a value stream map. It also helps calculation of lead time and total cycle time.

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
body	<i>SVGRectElement</i>	Rectangle body of shape
line	<i>SVGPathElement</i>	Path line of shape
label	<i>SVGTextElement</i>	Shape text
labelTotalWaiting	<i>SVGTextElement</i>	Total waiting text
labelTotalProcessing	<i>SVGTextElement</i>	Total processing text

shapes.vsm.TimelineWaiting

The TimelineWaiting shape shows waiting times in a value stream map.

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
line	<i>SVGPathElement</i>	Path line of shape
label	<i>SVGTextElement</i>	Shape text

shapes.vsm.TriangleInventory

The TriangleInventory shape can represent basic inventory between 2 processes and stored inventory in a value stream map. Additional details such as the count can also be placed underneath the symbol.

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes

body	<i>SVGPathElement</i>	Path body of shape
icon	<i>SVGPathElement</i>	Icon of shape
label	<i>SVGTextElement</i>	Shape text

shapes.vsm.Truck

The Truck shape represents external shipments to customers or from suppliers in a value stream map. The shipment frequency may be noted.

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
background	<i>SVGPathElement</i>	Path background of shape
body	<i>SVGPathElement</i>	Path body of shape
label	<i>SVGTextElement</i>	Shape text

shapes.vsm.Workcell

The Workcell shape represents the integration of multiple processes into one manufacturing work cell in a value stream map.

Supported `attrs` properties:

Selector	Node	Description
root	<i>SVGGElement</i>	Container for all shape nodes
body	<i>SVGPathElement</i>	Path body of shape
label	<i>SVGTextElement</i>	Shape text

JointJS+ Local

storage.Local

`storage.Local` plugin gives you a convenient way to **store documents** (especially JointJS graphs and cells) to the **client-side [HTML 5 localStorage](#)**. This browser storage allows you to store data in the browser similar to cookies but with a greatly enhanced capacity and no information stored in the HTTP request header.

The standard HTML 5 `localStorage` is a simple key-value store where both keys and values are strings. Therefore, it is on the programmer to invent a way of storing and retrieving more complex data structures. `storage.Local` make this abstraction for you and its API, though overly simplified, resembles APIs for common NoSQL document object stores such as MongoDB.

Install

Include `joint.storage.local.js` file into your HTML:

```
<script src="joint.storage.local.js"></script>
```

Usage

`joint.storage.Local` is a singleton object with three basic methods: `insert()`, `find()` and `remove()`. It behaves like a single database with collections of objects. You can add, remove and search objects in each collection. Objects can be any JavaScript objects or JointJS elements, links or graphs. Currently, `joint.storage.Local` indexes objects in collections by their IDs only. It is important to note that all the methods are asynchronous. This means that the callback is deferred until the current JavaScript call stack has cleared (similar to `setTimeout` with a delay of 0). This is because we want to keep the API similar to other NoSQL databases and so if you later decide to store your documents in e.g. MongoDB instead, you don't have to make drastic changes to your application. Instead, you can just write a simple replacement for `joint.storage.Local` and store your documents e.g. via AJAX on the server.

```
const graph = new joint.dia.Graph;
(new joint.shapes.standard.Rectangle).addTo(graph);

// Insert a graph into the collection 'graphs'.
joint.storage.Local.insert('graphs', graph);
// Find all the graphs in the collection 'graphs'.
joint.storage.Local.find('graphs', {}, function(err, graphs) {});
// Find a graph with ID 'mygraph' in the collection 'graphs'.
joint.storage.Local.find('graphs', { id: 'mygraph' }, function(err, graphs) {});
// Remove all the graphs from the collection 'graphs'.
joint.storage.Local.remove('graphs', {}, function(err) {});
// Remove a graph from the collection 'graphs'.
joint.storage.Local.remove('graphs', { id: 'mygraph' }, function(err) {});
```

API

insert(collection, doc [, callback])	Insert a document <code>doc</code> into the <code>collection</code> . Optionally pass a <code>callback(err, doc)</code> . If the document <code>doc</code> didn't have an <code>id</code> property, one is automatically created and will be part of the <code>doc</code> object returned in the callback.
find(collection, query, callback)	Find a document in <code>collection</code> . <code>query</code> can currently be either empty in which case all the documents from the <code>collection</code> are returned or it can contain <code>id</code> of a document in which case only a document with that <code>id</code> is returned. <code>callback</code> signature is: <code>callback(err, docs)</code> .
remove(collection, query [, callback])	Remove a document from the <code>collection</code> . <code>query</code> can currently be either empty in which case all the documents from the <code>collection</code> are removed or it can contain an <code>id</code> of the document to be removed. <code>callback</code> signature is: <code>callback(err)</code> .

JointJS+ ui

ui.intro

JointJS+ now comes with a variety of themes ("material", "modern", "dark", and the default theme). It is possible to set the theme globally (applied to all UI components) and to set a different theme for each UI component individually.

Setting the theme globally:

```
joint.setTheme('material');
```

This will update the theme for all the components currently rendered, as well as for any components that are rendered from now on.

Setting the theme for an individual component:

```
var colorPalette = new joint.ui.ColorPalette({
  theme: 'material',
  options: [
    { content: '#90C3D4' },
    { content: '#D4A190' },
    { content: '#A1D490' },
    { content: '#ffffff' }
  ]
});
document.body.appendChild(colorPalette.render().el);
```

And to change the theme of a component that has already been rendered:

```
colorPalette.setTheme('modern');
```

Here you can view all the JointJS+ UI plugins for every theme (you can also open it in a separate window) :

JointJS+ Clipboard

ui.Clipboard

Clipboard implements copy-pasting of cells. Optionally, clipboard also supports storing cells in an HTML 5 localStorage so that copy-pasting of cells works despite a browser tab being refreshed or closed/reopened. Additionally, clipboard is also able to copy cells from one paper and paste them to another. However, the determination of the targeted paper is left to the application layer.

Installation

Include `joint.ui.clipboard.js` to your HTML:

```
<script src="joint.ui.clipboard.js"></script>
```

Clipboard Configuration

Instantiating a clipboard is a matter of creating a `joint.ui.Clipboard` object:

```
var clipboard = new joint.ui.Clipboard({ useLocalStorage: false });
```

Options passed to a clipboard used for clipboard operations but can be overridden for the specific method call.

link	Set of attributes to be set on each copied link on every <code>pasteCells()</code> or <code>pasteCellsAtPoint()</code> call. It is defined as an object. e.g. <code>{ z: 1 }</code> .
translate	An increment that is added to the pasted elements position on every <code>pasteCells()</code> call. It is an object with <code>dx</code> and <code>dy</code> attributes. It defaults to <code>{ dx: 20, dy: 20 }</code> .
origin	A position in a cells' bbox which is used for determining the origin on every <code>pasteCellsAtPoint()</code> call. It is a string representing bbox positions (e.g. 'top-left', 'bottom' etc.). It defaults to <code>'center'</code> .
useLocalStorage	Set to <code>false</code> if you don't want to use <code>window.localStorage</code> for storing copied cells. It defaults to <code>true</code> .
addCellOptions	Options that will be passed to the <code>graph.addCell</code> method during paste operations.
removeCellOptions	Options that will be passed to the cell's <code>remove</code> method during the cut operation.

deep	Set to <code>true</code> if you want the clipboard to automatically add all embedded cells to the provided elements for copying.

Clipboard API

copyElements(collection, graph, opt)	Store the elements from the <code>collection</code> along with all links, which are connected by both source and target to these elements. Collection can be either a <code>Backbone.Collection</code> or an array of cells. The options <code>opt</code> passed as a parameter can override the global options.
cutElements(collection, graph, opt)	Similar to <code>copyElements()</code> but it also removes all copied cells from the <code>graph</code> .
pasteCells(graph, opt)	Add copies of stored cells to the <code>graph</code> . The method can be called multiple times and it always produces new copies.
pasteCellsAtPoint(graph, point, opt)	Add copies of stored cells to the <code>graph</code> at specific point. The method can be called multiple times and it always produces new copies. The <code>origin</code> option indicates which point on the cells bbox should be used for pasting. The default value is <code>'center'</code> .
isEmpty(opt)	Return <code>true</code> when there is no cells stored on the Clipboard instance. Returns <code>false</code> otherwise. Checks local storage if the <code>useLocalStorage</code> option is <code>true</code> .
clear(opt)	Clear all stored cells. Also clears <code>window.localStorage</code> if <code>useLocalStorage</code> is <code>true</code> .

Clipboard Setup

The next step is hooking these methods to relevant user events.

```
var selection = new joint.ui.Selection({ paper: paper });
var keyboard = new joint.ui.Keyboard;

keyboard.on('ctrl+c', function() { clipboard.copyElements(selection.collection,
paper.model); });
keyboard.on('ctrl+x', function() { clipboard.cutElements(selection.collection,
paper.model); });
keyboard.on('ctrl+v', function() { clipboard.pasteCells(paper.model); });
```

Note that the `selection.collection` is a `Backbone.Collection` that stores the selected elements. The clipboard is probably the best when used in combination with the **Selection** JointJS plugin. We suggest you to look at the documentation to the [Selection](#) plugin for further details on using selections.

Also, it is advisable to look at our Kitchen Sink application to see how the Selection and Clipboard work together.

JointJS+ ColorPalette

ui.ColorPalette

`ui.ColorPalette` is a UI widget for displaying dropdowns with color palette.

Installation

Include `joint.ui.colorPalette.js` and `joint.ui.colorPalette.css` files to your HTML:

```
<link rel="stylesheet" type="text/css" href="joint.ui.colorPalette.css">
<script src="joint.ui.colorPalette.js"></script>
```

Usage

Create an object of `ui.ColorPalette` type, render it with the `render()` method and append the generated HTML element anywhere in your DOM:

```
var colorPalette = new joint.ui.ColorPalette({
  options: [
    { content: '#000000' },
    { content: '#FFFFFF' },
    { content: 'transparent', icon:
'data:image/png;base64,iVBORw0KGgoAAAANSUHEUgAAAGoAAABrCAYAAACffRcyAAAACXBIWXMAAAAsTAA
ALEwEAmpwYAAAAAGXRFWHRTb2Z0d2FyZQBBZG9iZSBjbWFnZVJlYWR5ccllPAAABPxJREFUeNrsnc9rU1kUx+/
LS2LTBtI4VMiqjYjiQgQ3IoIUuhvGUmYzUBgV8SdWLS7sooLgxm0XgjKg0P+gMDC7ggjDzGwG0t1psS2WBpti
oiYVmrxbxnviiSc17eaPvvdzz7vcLlyTk5vVyPtzzzv3mtDWq1arYrVwuNyAfxuUYkaNfQEFoWY4Z0aYymczS7
jeNRLASUC9NlOMs4tZRTdNgkcCKX4GyID2V4yjiPITm5Bisw4oAkrIiFk8tNp9AWek0kNSENvVLfaurq1Q4LC
ImSisbtao7R5mmWTQMYxPx8l6yRohvb2/3tpk2HrVKcFtA6XT6WSwWKY0k/qLSqfQUcOVTDsBGIk7nJEAKRhR
jirXDLP6I024CpGBhUczt3rcFhXtS8HKKeQTh4aHot35Q3vy0bW1tpdvNSyQSi8lk8iWuiyFQtGg59rWbJyuZ
NVzv+4XUx0QABVCa68+/0mJi8oj4/Y99Xlwuioj6B0n8LSHx4U0s9vr0j2vYUYrJ/G++6z0kRKIihn9aQepTz
V14viB6Lo5LP0N68mhWnDxRACjFIP1w9YYwNjZMryEBLNeQSmVR7e7e9hoSib44rLasMqLRtb6+vlm7D5ZKpf
3ycNfj4qCY2tnZ2dM2t5tmSY4NbtejexKL09pJB0ndbw/ym4cPtd0AriRymfD4/ZHeo/uaqz6iN4vTDGxWPx9d
SqdQ8q+trdXfhGhU0tXRnPh44u3nwwDE/HAYkPi9KcB/uSQDFEBJAMYEEUegAdT30A4BQgIoBRwHgGLi0AAU
E8fBrXzvmAATPR0W3TgEdPhU5Xo1x+Hy9WbHQR5m5QnZk/XJ0a8DAeW2h0A3x8Ht+pD6GJfgAMUcEkDZQSLdm
fhhbFUGAZQfp9q1/xK+jKyotUXtQTY5DHdLYLZeqrVNrUE20g8KQtAbV5DgoDklbUI20AwdIJO16Jhp7H0h1eX
LizcbPw1tBrw89E24dB2snSUjZTqwPPRP/pwRXPN3pB4o5JD1AhQBS+EGFBFkoQXFxHLQGxcLx0BYUN8dBs1C
R9yWRUneflePgVqHpmTCKb83kxWvZy0JSV4PjkJU0KwK62txgNawZyKfj4vRc0Pi1UpXpx0H9EzYRzAuRn4hS
L1hS3fhAaUJJN6gNILEF5RmkFiCoup0N0jsQNE5KXnmQLY3SKxAEaS9V2+K+jlJJ0hsQNUhXV4sCB0h1VK+6j
0TNcdBprtdjkPgPQ7omQiZ46Bfz4SGJTg/UIDEABQgMQAFS0qD0tVxYAVKZ8eBDSjdHQCWo0A4uFfHeia49Tj
o2TMRYschPD0TKMEZgAikBqAAiQEoQFIffBwHBqDg0DAABceBASg4Dj7cQrzumQhrj004eibgODDomUAJzuAe
BUhMQF0a0w5IHECVy3FA8lfe/HvXu5P/1h4V+husANVKAMTXmYAAcqnPD9EJ3M3hzm0PgW7XCwyU29/6wPWQ+
nCPghQAVa1W4whPsHKKuS0oeUPsrVQqPQhfMKJYU8ydQC3bvVkoFE4BVjCQKNY0U5ap6puR46bdrLpfXx82Tb
NoGMYmQupPunPaSZZm6BveAflkESFTWtLIjPnZkk+mEQtLNU2M6sXEuBxziiLymrPYfKr6JLGifBgEL0UgDVps
```

```

vpTnDbCQBhVId42QSIas0r6alcvlBqwtNyJHP+IWiJatCnzKqhua1BIUpJ4+CjAAVnYzLhKE5pcAAAAASUVOR
K5CYII=' },
    { content: '#B3B3B3' },
    { content: '#808080' },
    { content: '#4D4D4D' },
    { content: '#E6E6E6' },
    { content: '#FFC7C9' },
    { content: '#FFA0A4' },
    { content: '#E3686D' },
    { content: '#D71920' },
    { content: '#FFE3D1' },
    { content: '#FFCBA8' },
    { content: '#FFAB73' },
    { content: '#F58235' }
  ]
});
document.body.appendChild(colorPalette.render().el);

```

Configuration

The following table lists options that you can pass to the `ui.ColorPalette` constructor function. You might notice that many of the options are similar to the `ui.SelectBox` widget. This is because the `ui.ColorPalette` inherits from this widget.

width	The width of the color palette in pixels.						
options	An array of items. Each item is an object with the following properties: <table> <tr> <td>content</td><td>The color in hex or other color formats (rgb, rgba).</td></tr> <tr> <td>icon</td><td>A path to an image (note that it can also be an image embedded via the Data URI Scheme). The icon can be handy for representing the transparent color.</td></tr> <tr> <td>selected</td><td><code>true</code> if the item should be selected by default. If non of the items are selected, the first item is selected by default.</td></tr> </table>	content	The color in hex or other color formats (rgb, rgba).	icon	A path to an image (note that it can also be an image embedded via the Data URI Scheme). The icon can be handy for representing the transparent color.	selected	<code>true</code> if the item should be selected by default. If non of the items are selected, the first item is selected by default.
content	The color in hex or other color formats (rgb, rgba).						
icon	A path to an image (note that it can also be an image embedded via the Data URI Scheme). The icon can be handy for representing the transparent color.						
selected	<code>true</code> if the item should be selected by default. If non of the items are selected, the first item is selected by default.						
selected	The index of the item which should be selected by default. If this value is undefined (which it is by default), the <code>ui.ColorPalette</code> widget looks up the selected item from the <code>options</code> array (by using the <code>selected</code> boolean property). If nothing is selected, the first item in the <code>options</code> array is selected by default.						
keyboardNavigation	<code>true</code> (the default) if you want the <code>ui.ColorPalette</code> to provide a keyboard navigation (up/down/left/right/enter/escape keys are supported).						

selectBoxOptionsClass	When the color palette is opened, the <code>ui.ColorPalette</code> generates an HTML container that contains all the items. This container is then appended to the document body (or <code>target</code> if provided). Sometimes, you want to provide custom styling to this color palette container. <code>selectBoxOptionsClass</code> gives you a chance to define a CSS class that will be set on this container so that you can style it in your CSS.
target	<code>ui.ColorPalette</code> generates an HTML container that contains all the color palette items. This container is by default appended to the document body. In some situations (e.g. if you want the color palette to scroll with some other container), you want to append this container to a different DOM element. The <code>target</code> option is exactly for that. It accepts an HTML element that will be used as a container for the generated color palette items.
placeholder	In some cases, you want to “deselect” the color palette and show a placeholder in place of the selected item. This is when you don't want any of the items to be selected. In this case, set the <code>selected</code> index to <code>-1</code> and the <code>placeholder</code> to any HTML you want to display in the selected item window. This placeholder has the <code>selected-box-placeholder</code> CSS class that you can use for further styling.

API

render()	Render the color palette. Note that once you render the color palette, you can use the <code>el</code> property that points to the container HTML element and append it anywhere in the DOM (e.g. <code>\$(document.body).append(colorPalette.el)</code>).
getSelection()	Get the current selection. The selection points to one of the items from the <code>options</code> array.
getSelectionValue()	Get the current selection value (the color).
getSelectionIndex()	Get the index in the <code>options</code> array of the item that is currently selected.
select(index, opt)	Select an item. <code>index</code> is the index to the <code>options</code> array. <code>opt</code> is optional and can be an arbitrary object that will be passed to the <code>option:select</code> event handler.
selectByValue(color)	Select an item by color.
isOpen()	Returns <code>true</code> if the color palette is currently opened. <code>false</code> otherwise.
toggle()	Programmatically toggle the color palette.
open()	Programmatically open the color palette.

close()	Programmatically close the color palette.
remove()	Remove the color palette from the DOM.
on(event, handler [, context])	Register a <code>handler</code> function that will be called whenever <code>event</code> is triggered. The optional <code>context</code> is an object that will be the context of the handler function (the <code>this</code>).

Events

The `ui.ColorPalette` object triggers various events that you can react on. Events can be handled by using the `on()` method (see above).

option: hover	Triggered when the user hovers over one of the color items. The handler is passed the option object and the index of the option in the <code>options</code> array.
option: select	Triggered when the user selects an item. The handler is passed the option object and the index of the option in the <code>options</code> array. If you selected the item with the <code>select(index, opt)</code> method, the third argument to the event handler will be your <code>opt</code> object.
option: mouseout	Triggered when the user leaves an item with mouse cursor. The handler is passed an event object as the only argument.
close	Triggered when the color palette gets closed.

JointJS+ ContextToolbar

ui.ContextToolbar

`ui.ContextToolbar` is a UI widget for displaying context toolbars (usually a set of buttons) below a certain target element (HTML or SVG). There can always be only one context toolbar opened at a time. This is automatically handled by the `ui.ContextToolbar` component which simplifies the process and makes sure you don't have to keep track of opened context toolbars yourself.

Installation

Include `joint.ui.contextToolbar.js` and `joint.ui.contextToolbar.css` files in your HTML:

```
<link rel="stylesheet" type="text/css" href="joint.ui.contextToolbar.css">
<script src="joint.ui.contextToolbar.js"></script>
```

Usage

Create an object of `joint.ui.ContextToolbar` type, add tool buttons and call the `render()` method in order to open the context toolbar.

```
var ct = new joint.ui.ContextToolbar({
  tools: [
    { action: 'yes', content: 'Yes' },
    { action: 'no', content: 'No' },
    { action: 'maybe', content: 'Maybe' },
    { action: 'sure', content: 'Sure' }
  ],
  target: this
});

ct.on('action:yes', function() { alert('Yes') });
ct.on('action:no', function() { alert('No') });
ct.on('action:maybe', function() { alert('Maybe') });
ct.on('action:sure', function() { alert('Sure') });

ct.render();
```

A common usage is to open the context toolbar on click on a certain element (either HTML or SVG):

```
$('#circle').on('click', function() {

  var circle = this;
```

```

var ct = new joint.ui.ContextToolbar({
  tools: [
    { action: 'hide', content: 'Hide' },
    { action: 'info', content: 'Info' },
    { action: 'remove', content: 'Remove' }
  ],
  target: circle
});

ct.on('action:hide', ct.remove, ct);
ct.on('action:remove', function() {
  V(circle).remove();
});
ct.on('action:info', function() {
  alert('This is an SVG circle');
});

ct.render();
});

```

Configuration

The following table lists options that you can pass to the `ui.ContextToolbar` constructor function:

tools	An array of tools. Each tool is an object that can have the following properties:	
	action	The name of the action. This action is then triggered on the context toolbar object so that you can react to it later.
	content	The content of the button. It can be a text or an arbitrary HTML.
	icon	Optionally, instead of passing <code>content</code> , you can just pass a path to an image.
	attrs	An object with attributes that will be applied on the generated button. Useful for adding your own custom attributes such as <code>{ 'data-tooltip': 'My Tooltip' }</code> .
target	The target element (either HTML or SVG) or an object representing a point with properties <code>x</code> and <code>y</code> representing client coordinates.	
padding	The gap between the anchor of the target element and the context toolbar. The default is <code>20</code> .	

position	The name of the position on the target element to which the toolbar is attached. The default is <code>'bottom'</code> .
anchor	The name of the position on the toolbar, where it is attached to the target. The default is <code>'top'</code> .
autoClose	Determines if the context toolbar should be closed if the user clicked outside of the area of the context toolbar. The default is <code>true</code> .
vertical	When set to <code>true</code> the toolbar arranges tools from top to bottom instead of left to right. The default is <code>false</code> .
scale	Optional parameter that sets the scale of the context toolbar.

API

render()	Render the context toolbar onto the screen.
remove()	Remove the context toolbar. Usually, you don't have to call this method as the context toolbar is closed automatically if the user clicked outside of the area of the context toolbar (unless you set the <code>autoClose</code> option to <code>false</code>).
on(event, handler [, context])	Register a <code>handler</code> function that will be called whenever <code>event</code> is triggered. The optional <code>context</code> is an object that will be the context of the handler function (the <code>this</code>).
joint.ui.ContextToolbar.close()	This is a static method that closes any context toolbar that is currently open.
joint.ui.ContextToolbar.update()	This is a static method that updates the position of the opened context toolbar (if any). Call this method if your <code>target</code> element changes its position while you still want to keep the context toolbar opened.

Events

The `ui.ContextToolbar` object triggers events that correspond to the actions that you defined in the `tools` array. Events can be handled by using the `on()` method (see above).

action:[action]	Triggered when the user clicks on one of the buttons in the context toolbar.
close	Triggered when the context toolbar gets closed.

JointJS+ Dialog

ui.Dialog

`ui.Dialog` is a UI widget for displaying both modal and non-modal dialog boxes. It's a little window that overlays all other elements on the page. `ui.Dialog` supports an arbitrary HTML content, allows you to define buttons in an easy way and can even be inlined on the page in any HTML container. The dialog can be optionally draggable.

Installation

Include `joint.ui.dialog.js` and `joint.ui.dialog.css` files to your HTML:

```
<link rel="stylesheet" type="text/css" href="joint.ui.dialog.css">
<script src="joint.ui.dialog.js"></script>
```

Usage

Simply create an object of `ui.Dialog` type and call `open()` method on it:

```
var dialog = new joint.ui.Dialog({
  width: 300,
  title: 'My title',
  content: 'This is my dialog box.'
});
dialog.open();
```

Configuration

The following table lists options that you can pass to the `ui.Dialog` constructor function:

width	The width of the dialog in pixels.
title	The title of the dialog.
content	The content of the dialog. It can be a string or HTML or even a DOM element.
type	The type of the dialog. This effectively sets a CSS class on the dialog HTML container. The predefined dialog types are: <code>'info'</code> , <code>'alert'</code> , <code>'warning'</code> , <code>'success'</code> and <code>'neutral'</code> .
buttons	Buttons of the dialog. <code>buttons</code> is an array of objects of the form <code>{ action: String, content: String [, position: String] }</code> .

	<table> <tr> <td>action</td><td>The name of the action the button represents. When the button is clicked, the dialog object triggers an event <code>action:[name]</code>, allowing you to react.</td></tr> <tr> <td>content</td><td>The label of the button (it can be HTML, as well).</td></tr> <tr> <td>position</td><td>Optional. It can be <code>'left'</code>, <code>'right'</code>, or <code>'center'</code>. If position is not specified, it is considered to be <code>'right'</code>. If multiple buttons are provided with the same <code>position</code>, they are placed left-to-right according to the order in which they were provided.</td></tr> </table>	action	The name of the action the button represents. When the button is clicked, the dialog object triggers an event <code>action:[name]</code> , allowing you to react.	content	The label of the button (it can be HTML, as well).	position	Optional. It can be <code>'left'</code> , <code>'right'</code> , or <code>'center'</code> . If position is not specified, it is considered to be <code>'right'</code> . If multiple buttons are provided with the same <code>position</code> , they are placed left-to-right according to the order in which they were provided.
action	The name of the action the button represents. When the button is clicked, the dialog object triggers an event <code>action:[name]</code> , allowing you to react.						
content	The label of the button (it can be HTML, as well).						
position	Optional. It can be <code>'left'</code> , <code>'right'</code> , or <code>'center'</code> . If position is not specified, it is considered to be <code>'right'</code> . If multiple buttons are provided with the same <code>position</code> , they are placed left-to-right according to the order in which they were provided.						
draggable	If <code>draggable</code> is <code>true</code> , the dialog window will be draggable by the user. It defaults to <code>false</code> .						
closeButton	If <code>closeButton</code> is <code>false</code> , the dialog window will not display the default “cross” button in the top right corner allowing the user to close the dialog. It defaults to <code>true</code> .						
closeButtonContent	The HTML content of the little close button in the top right corner of the dialog window. It defaults to a “cross” (×).						
modal	If <code>false</code> , the dialog window will not be made modal. In other words, the user will still be able to interact with other elements on the page. It defaults to <code>true</code> .						
inlined	If <code>true</code> , the dialog window will be inlined in a container that you can pass to the <code>open()</code> method.						

API

open()	Open the dialog window.
close()	Close the dialog window.
on(event, handler [, context])	Register a <code>handler</code> function that will be called whenever <code>event</code> is triggered. The optional <code>context</code> is an object that will be the context of the handler function (the <code>this</code>).

Events

The `ui.Dialog` object triggers events when the user clicks one of its buttons. Events can be handled by using the `on()` method.

action:[name]	Triggered when the user clicks a button in the dialog with action named <code>[name]</code> .

close	Triggered when the dialog gets closed.

JointJS+ FlashMessage

ui.FlashMessage

`ui.FlashMessage` is a UI component for displaying flash messages. This is very useful for informing the user that something has happened. As you will see later on on this page, `ui.FlashMessage` plugin provides a quick way of displaying messages without you having to worry about closing the message boxes or cascading them on top of each other.

Installation

Include `joint.ui.flashMessage.js` and `joint.ui.flashMessage.css` files and `joint.ui.dialog.js` and `joint.ui.dialog.css` dependencies (`ui.FlashMessage` inherits from `ui.Dialog`) to your HTML:

```
<link rel="stylesheet" type="text/css" href="joint.ui.dialog.css">
<link rel="stylesheet" type="text/css" href="joint.ui.flashMessage.css">
<script src="joint.ui.dialog.js"></script>
<script src="joint.ui.flashMessage.js"></script>
```

Usage

There are two ways to display flash messages. One is by creating an object of `joint.ui.FlashMessage` type and call the `open()` method on it. However, for convenience, there is also a static method `joint.ui.FlashMessage.open()` that is much shorter:

```
// The longer method of displaying a flash message.
(new joint.ui.FlashMessage({
  title: 'Message',
  type: 'alert',
  content: 'An error occurred. Try again later.'
})).open();
```

```
// A shorter alternative.
joint.ui.FlashMessage.open('I am a flash message.');
```

Couple more examples of common usage:

```
$('#btn-display').on('click', function() {
  (new joint.ui.FlashMessage({
    title: 'Message',
    type: 'alert',
    content: 'An error occurred. Try again later.'
```

```

    })).open();
});

$('#btn-display-width').on('click', function() {
    (new joint.ui.FlashMessage({
        width: 150,
        title: 'Message',
        content: 'An error occurred. Try again later.'
    })).open();
});

$('#btn-display-modal').on('click', function() {
    (new joint.ui.FlashMessage({
        type: 'alert',
        closeAnimation: false,
        modal: true,
        title: 'Modal Message',
        content: 'This is a modal Flash message requiring the user to close the
message manually.'
    })).open();
});

$('#btn-close-all').on('click', function() {
    joint.ui.FlashMessage.close();
});

joint.ui.FlashMessage.open('ui.FlashMessage 1');
joint.ui.FlashMessage.open('ui.FlashMessage alert', '', { type: 'alert',
closeAnimation: { delay: 1000 } });
joint.ui.FlashMessage.open('ui.FlashMessage warning', '', { type: 'warning',
closeAnimation: { delay: 2000 } });
joint.ui.FlashMessage.open('ui.FlashMessage success', '', { type: 'success',
closeAnimation: { delay: 3000 } });
joint.ui.FlashMessage.open('ui.FlashMessage neutral', '', { type: 'neutral' });
joint.ui.FlashMessage.open('ui.FlashMessage info', '', { type: 'info' });
joint.ui.FlashMessage.open('ui.FlashMessage close delay 3s', '', { type: 'neutral',
closeAnimation: { delay: 3000 } });
joint.ui.FlashMessage.open('ui.FlashMessage with title', 'Title');
joint.ui.FlashMessage.open('ui.FlashMessage without close animation', '', {
closeAnimation: false });

```

Configuration

The following table lists options that you can pass to the `ui.FlashMessage` constructor function:

title	The title of the flash message.

type	The type of the flash message. One of <code>'alert'</code> (default), <code>'warning'</code> , <code>'success'</code> , <code>'neutral'</code> .
content	The content of the flash message. It can be either a text or an arbitrary HTML.
modal	Set to <code>true</code> if you want your flash message to be modal.
closeAnimation	Set to <code>false</code> if you don't want the flash message to be closed in an animated fashion. It can also be an object of the form <code>{ delay: Number, duration: Number }</code> specifying properties of the close animation.

API

open()	Open the flash message.
joint.ui.FlashMessage.open(content, title, opt)	Open a flash message (the shorter way). <code>opt</code> can contain any option that you can pass to the <code>joint.ui.FlashMessage</code> constructor function (see the Configuration section for details).
on(event, handler [, context])	Register a <code>handler</code> function that will be called whenever <code>event</code> is triggered. The optional <code>context</code> is an object that will be the context of the handler function (the <code>this</code>).

Events

The `ui.FlashMessage` object triggers events that you can react on. Events can be handled by using the `on()` method (see above).

close:animation:complete	Triggered when the flash message is actually removed from the DOM, i.e. when the close animation finishes.
close	Triggered when the flash message gets closed.

JointJS+ FreeTransform

ui.FreeTransform

FreeTransform plugin creates a control panel above an element giving a user the ability to resize an element in any direction or rotate it.

Installation

Include `joint.ui.freetransform.css`, `joint.ui.freetransform.js` and `joint.dia.freetransform.js` files to your HTML:

```
<link rel="stylesheet" type="text/css" href="joint.ui.freeTransform.css">
<script src="joint.dia.freeTransform.js"></script>
<script src="joint.ui.freeTransform.js"></script>
```

Usage

The usage of the `ui.FreeTransform` plugin is pretty straightforward. Just instantiate an object of the `joint.ui.FreeTransform` type and pass in the view for the element you want to have the free transform handles hooked onto:

```
var graph = new joint.dia.Graph;
var paper = new joint.dia.Paper({ el: $('#paper'), width: 500, height: 500, model:
graph });

paper.on('cell:pointerup', function(cellView) {
  // We don't want to transform links.
  if (cellView.model instanceof joint.dia.Link) return;

  var freeTransform = new joint.ui.FreeTransform({ cellView: cellView });
  freeTransform.render();
});
```

Configuration

The `joint.ui.FreeTransform` constructor accepts the following options:

cellView	The view for the element you want the free transform handles to be displayed for.
preserveAspectRatio	Set to <code>true</code> if you want the resizing to preserve the aspect ratio of the element. The default is <code>false</code> .

resizeGrid	Set to <code>{ width: number, height: number }</code> if you want the resizing to snap to grid by specified dimensions. The default is <code>undefined</code> .
allowRotation	Set to <code>false</code> if you want the rotating handle to be omitted from the handles. The default is <code>true</code> .
minWidth	The minimum width allowed for the element when resizing. The default is <code>0</code> . It can be a number or a function returning a number <code>(cell) => number;</code> .
maxWidth	The maximum width allowed for the element when resizing. The default is <code>Infinity</code> . It can be a number or a function returning a number <code>(cell) => number;</code> .
minHeight	The minimum height allowed for the element when resizing. The default is <code>0</code> . It can be a number or a function returning a number <code>(cell) => number;</code> .
maxHeight	The maximum height allowed for the element when resizing. The default is <code>Infinity</code> . It can be a number or a function returning a number <code>(cell) => number;</code> .
rotateAngleGrid	The steps in angle when rotating the element. The default is <code>15</code> .
allowOrthogonalResize	Set to <code>false</code> if you only want the four corner handles to be displayed. The default is <code>true</code> .
resizeDirections	The list of the resize handles to be displayed referenced by their direction. It defaults to <code>['top', 'bottom', 'left', 'right', 'top-left', 'top-right', 'bottom-left', 'bottom-right']</code> i.e. all resize handles are displayed.
clearAll	If set to <code>true</code> (the default value), clear all the existing freeTransforms from the page when a new freeTransform is created. This is the most common behavior as it is assumed that there is only one freeTransform visible on the page at a time. However, some applications might need to have more than one freeTransform visible. In this case, set <code>clearAll</code> to <code>false</code> (and make sure to call <code>remove()</code> once you don't need a freeTransform anymore)
clearOnBlankPointerdown	If set to <code>true</code> (the default value), clear the freeTransform when a user clicks the blank area of the paper.
usePaperScale	If set to <code>true</code> , the freeTransform frame and its handles scale with the paper.
padding	The gap between the element and the freeTransform frame. It defaults to 3px.

API

The `joint.ui.FreeTransform` objects provide the following methods:

<code>render()</code>	Render the free transform widget. You should call this right after you instantiate the free transform object.
<code>remove()</code>	Remove the free transform widget.
<code>joint.ui.FreeTransform.clear(paper)</code>	Clear all the free transform widgets from a JointJS paper.

Events

The plugin fires the following events.

Event Name	Handler Signature	Description
<i>resize:start</i>	<code>(evt: dia.Event)</code>	Triggered when the user starts resizing the element.
<i>resize</i>	<code>(evt: dia.Event)</code>	Triggered while the element is being resized.
<i>resize:stop</i>	<code>(evt: dia.Event)</code>	Triggered when the user finishes resizing the element.
<i>rotate:start</i>	<code>(evt: dia.Event)</code>	Triggered when the user starts rotating the element.
<i>rotate</i>	<code>(evt: dia.Event)</code>	Triggered while the element is being rotated.
<i>rotate:stop</i>	<code>(evt: dia.Event)</code>	Triggered when the user finishes rotating the element.

JointJS+ Halo

ui.Halo

Halo creates a control panel above an element with various tools. This gives the user control over your elements with tools located right at hand.

Installation

Include `joint.ui.halo.css` and `joint.ui.halo.js` files to your HTML:

```
<link rel="stylesheet" type="text/css" href="joint.ui.halo.css">
<script src="joint.ui.halo.js"></script>
```

Creating a Halo

```
var graph = new joint.dia.Graph;
var paper = new joint.dia.Paper({ el: $('#paper'), width: 500, height: 500,
model: graph });

paper.on('cell:pointerup', function(cellView) {
  // We don't want a Halo for links.
  if (cellView.model instanceof joint.dia.Link) return;

  var halo = new joint.ui.Halo({ cellView: cellView });
  halo.render();
});
```

The `joint.ui.Halo` constructor takes a graph and paper objects + it also requires the view of a cell we want to display the halo above.

Default handles (tools)

The `ui.Halo` control panel has the following built-in tools. To add custom tools, please see the section [Customizing halo tools](#).

remove	Remove the element.
clone	Drag to clone the element.
link	Drag to link the element with another element. Note that the new link will be created based on the <code>defaultLink</code> option from the <code>joint.dia.Paper</code> object.

fork	Drag to clone the element and connect it with cloned element in one go.
unlink	Remove all the links coming in/out of the element.
resize	Resize the element.
rotate	Rotate the element.

Configuration

The `joint.ui.Halo` constructor function takes an options object as an argument. The options object can have the following parameters:

cellView	an element or link view. Previously, Halo accepted also <code>paper</code> and <code>graph</code> options but this is now deprecated as those can always be inferred from <code>cellView.paper</code> and <code>cellView.paper.model</code> properties.
type	the type of the Halo control panel. The default value is <code>'surrounding'</code> . Another possible values are <code>'pie'</code> in which case the Halo will be displayed as a <u>pie menu</u> and <code>'toolbar'</code> in which case the Halo tools are displayed above the element in a small toolbar. See the demos below.
loopLinkPreferredSide	the preferred side for a self-loop link created from Halo (<code>"top" "bottom" "left" "right"</code>), default is <code>"top"</code>
loopLinkWidth	the self-loop link width in pixels, default is <code>40</code>
rotateAngleGrid	the angle increments the rotate action snaps to, default is <code>15</code>
rotateEmbeds	should be elements embedded inside the <i>cellView</i> element rotated as well? Default is <code>false</code>
boxContent	a string (text or HTML) or a function (of the form <code>boxContent(cellView, boxDOMElement)</code>) that returns an HTML string with the content that will be used in the information box below an element. By default, the box displays the x, y coordinates and width and height dimensions of the element. If <code>boxContent</code> is set to <code>false</code> (or an empty string), the information box will be hidden
clearAll	if set to <code>true</code> (the default value), clear all the existing halos from the page when a new halo is created. This is the most common behavior as it is assumed that there is only one halo visible on the page at a time. However, some applications might need to have more than one halo visible. In this case, set <code>clearAll</code> to <code>false</code> (and make sure to call <code>remove()</code> once you don't need a halo anymore)

clearOnBlankPointerdown	if set to <code>true</code> (the default value), clear the halo when a user clicks the blank area of the paper.
useModelGeometry	if set to <code>true</code> , the model position and dimensions will be used as a basis for the Halo tools position. By default, this is set to <code>false</code> which causes the Halo tools position be based on the bounding box of the element view. Sometimes though, your shapes can have certain SVG sub elements that “stick out” of the view and you don't want these sub elements to affect the Halo tools position. In this case, set the <code>useModelGeometry</code> to <code>true</code> .
clone	a function with signature <code>function(cell, opt)</code>. This function will be called when cloning or forking actions take place and it should return a clone of the original cell. This is useful e.g. if you want the clone to be moved by an offset after the user clicks the clone handle. The default function is: <code>function(cell, opt) { return cell.clone().unset('z') }</code>. The callback may return an array of cells. This is useful if the element has nested cells that also need to be cloned. In this case the user drags the first cell in the array. <pre> new ui.Halo({ /* ... */ clone: (cell) => { return cell.clone({ deep: true }); // const [cellClone, ...embedsClones] = cell.clone({ // deep: true }); // return [cellClone, ...embedsClones]; } }); </pre>
pieSliceAngle	<i>(only valid for the 'pie' type of Halo)</i> The angle of one slice in the pie menu. It defaults to <code>45</code> .
pieStartAngleOffset	<i>(only valid for the 'pie' type of Halo)</i> The angular offset of the first handle in the pie menu. It defaults to <code>0</code> .
pieIconSize	<i>(only valid for the 'pie' type of Halo)</i> The size in pixels of the icon in the pie menu. It defaults to <code>14</code> .
pieToggles	<i>(only valid for the 'pie' type of Halo)</i> An array of pie toggle buttons. Usually, there's only one (the default) but you can have as many as you want. The default value is <code>[{ name: 'default', position: 'e' }]</code> . Each item in the array is an object of the form <code>{ name: [name of you toggle], position: [one of e/w/s/n] }</code> . The <code>name</code> is passed in the options object in the <code>state:close</code> and <code>state:open</code> events triggered by the Halo when the pie toggle is clicked.

bbox	A bounding box within which the Halo view will be rendered.
tinyThreshold	if the set number value is higher than the Halo <code>bbox</code> width and height, a <i>'tiny'</i> CSS class is applied to the Halo. This reduces the size of the Halo. For elements, the default value is <code>40</code> . For links, the default Halo <code>bbox</code> width and height both equal <code>1</code> . This means a value of <code>2</code> or higher is needed to apply the class.
smallThreshold	if the set number value is higher than the Halo <code>bbox</code> width and height, a <i>'small'</i> CSS class is applied to the Halo. This reduces the size of the Halo. A <i>'small'</i> class will not be applied, if a <i>'tiny'</i> class is already present. For elements, the default value is <code>80</code> . For links, the default Halo <code>bbox</code> width and height both equal <code>1</code> . This means a value of <code>2</code> or higher is needed to apply the class.
magnet	A function accepting an <code>elementView</code> returning an <code>SVGElement</code> used as a magnet for links created via linking or forking handles. <pre> new joint.dia.Halo({ cellView: elementView, magnet: function(elementView, end, evt) { // where `end` is either "source" (linking and forking) // or "target" (forking only) // and `evt` is a `mousedown` event // connect the link directly to a rectangle element as // opposed to the element group return elementView.el.querySelector('rect') elementView.el; } }) </pre>

Disabling halo tools

Built-in tools are `remove` , `resize` , `rotate` , `clone` , `fork` , `link` and `unlink` . You can call the `removeHandle()` method to remove any of these tools just by passing its name:

```
halo.removeHandle('clone');
```

Alternatively, you can disable tools via CSS:

```
.halo .handle.clone { display: none; } /* disables the clone tool */
```

Also, the same way you would hide halo tools, as explained above, you can reposition them. For example, to reposition the `remove` icon tool to the bottom left corner, you could do:


```
.halo .handle.remove {
  left: -25px;
  bottom: -25px;
  top: auto;
  right: auto;
}
```

Note the `auto` values for the top/right coordinates are important in order to cancel out the default values defined in the Halo plugin.

Another way to remove/reposition tool handles in the Halo is in JavaScript:

```
halo.removeHandle('clone');
halo.changeHandle('remove', { position: 'se' });
```

The code above removes the clone tool from the Halo and re-positions the remove tool to the bottom right (south-east).

Each Halo can also be styled in CSS (disabling tools, repositioning, changing icons, ...) based on the type of element the Halo is displayed for. The Halo `<div>` container stores the type of the element in its `data-type` attribute. This makes it easy to target a halo for a certain type of element in your CSS. For instance, let's say we want to hide the remove tool only for an element of type `"standard.Rectangle"`. We can do this:

```
.halo[data-type="standard.Rectangle"] .remove { display: none; }
```

Customizing halo tools

Halo provides three methods for adding, removing and changing custom tools: `addHandle()`, `removeHandle()` and `changeHandle()`. Use the `addHandle()` method to add new tools to your halo:

```
halo.addHandle({ name: 'myaction', position: 's', icon: 'myaction.png' });
halo.on('action:myaction:pointerdown', function(evt) {
  evt.stopPropagation();
  alert('My custom action.');
```

```
});
```

In the example above, we added a new tool named `myaction`, positioned the tool to the south (bottom-center) and used our own icon `myaction.png`. When the user clicks on our tool, Halo triggers an event named `action:[name]:pointerdown`. We can handle the event by listening on the halo object. Similarly, the Halo triggers `action:[name]:pointermove` and `action:[name]:pointerup` events. This gives us a high flexibility in implementing our own actions.

For removing a halo tool, use the `removeHandle(name)` method:

```
halo.removeHandle('myaction')
```

`changeHandle()` method allows us to change halo tool handles. For instance, if we want to change a position of the clone tool, we could use:

```
halo.changeHandle('clone', { position: 'se' })
```

Pie menu type of Halo

To display the Halo control panel as a pie menu, just set the `type` option to `'pie'`:

```
new joint.ui.Halo({ cellView: myElementView, type: 'pie' })
```

You can still add your own actions as you would normally do with the default `'surrounding'` type of Halo.

Toolbar type of Halo

To display the Halo control panel as a small toolbar above an element, just set the `type` option to `'toolbar'`:

```
new joint.ui.Halo({ cellView: myElementView, type: 'toolbar' })
```

You can still add your own actions as you would normally do with the default `'surrounding'` type of Halo.

Links with Halo

It is also possible to use the Halo plugin with links, as the following demo illustrates.

API

addHandle(opt)	Add a custom tool to the Halo. <code>opt.name</code> is the name of the custom tool. This name will be also set as a CSS class to the handle DOM element making it easy to select it your CSS stylesheet. <code>opt.position</code> is a string that specifies a position of the tool handle. Possible values are <code>n</code> , <code>nw</code> , <code>w</code> , <code>sw</code> , <code>s</code> , <code>se</code> , <code>e</code> and <code>ne</code> . <code>opt.icon</code> is a URL of the icon used to render the tool. This icons is set as a background image on the tool handle DOM element.
addHandles(handles)	Add multiple handles in one go. This calls <code>addHandle()</code> internally for each item of the <code>handles</code> array.
removeHandle(name)	Remove a tool handle named <code>name</code> from the Halo.
removeHandles()	Remove all handles from the Halo.
changeHandle(name, opt)	Change a tool handle named <code>name</code> in the Halo. The <code>opt</code> object can contain the same parameters as with the <code>addHandle()</code> method except of the <code>name</code> property. <code>opt</code> parameters will be merged with those defined previously.

render()	Render the Halo. This must be called after the Halo object is instantiated.
on(event, callback)	Register a handler (<code>callback</code>) for an <code>event</code> . See the Halo Events section for the list of events the Halo object triggers.
toggleState([toggleName])	Toggle (open/close) the state of the Halo (applicable for the <i>pie</i> type of Halo). <code>toggleName</code> is the name of the pie toggle as defined in the <code>pieToggles</code> option. If <code>toggleName</code> is not passed, all pie menus change the state (the most common usage as there is usually only one pie toggle).
isOpen([toggleName])	Return <code>true</code> if the Halo is open (applicable for the <i>pie</i> type of Halo). <code>toggleName</code> is the name of the pie toggle as defined in the <code>pieToggles</code> option. If <code>toggleName</code> is not passed, return <code>true</code> if any of the pie menus is open.
remove()	Remove/destroy the Halo. Note that by default, the Halo removes itself when the user clicks on a blank area in the paper. In some cases, however, it is useful to remove the Halo programmatically.
joint.ui.Halo.clear(paper)	Remove all the Halo panels (without having to have a reference to a particular halo object) from the <code>paper</code> . Note that this is a static function in the <code>joint.ui.Halo</code> namespace rather than a method of a halo object.

Events

The Halo object triggers events when the user manipulates its tools. These events can be handled by using the Halo `on()` method.

action: [name]:pointerdown	Triggered when the user clicks (touches) on a tool handle named <code>[name]</code> . The handler is called with the mousedown event object, and <code>x</code> and <code>y</code> of the pointer in local coordinates.
action: [name]:pointermove	Triggered when the user moves with mouse cursor after a tool handle named <code>[name]</code> was mousedown (touched). The handler is called with the mousemove event object, and <code>x</code> and <code>y</code> of the pointer in local coordinates.
action: [name]:pointerup	Triggered when the user releases his mouse cursor after a tool handle named <code>[name]</code> was mousedown (touched). The handler is called with the mouseup event object, and <code>x</code> and <code>y</code> of the pointer in local coordinates.
action: [name]:contextmenu	Triggered when the user invokes contextmenu on a tool handle named <code>[name]</code> . The handler is called with the mousedown event object, and <code>x</code> and <code>y</code> of the pointer in local coordinates.

action:link:add	Triggered after the user finished creating a link with the default link tool. This is useful if you want to do something with the link after it has been created. For example, to prevent the user from creating loose links (links that do not have both source and target set), you can do: <pre> halo.on('action:link:add', function(link) { if (!link.get('source').id !link.get('target').id) { link.remove(); } }); </pre>
state:open	Triggered when the user opens the Halo (applicable for the <i>pie</i> type of Halo). The handler is passed the pie toggle name as defined in <code>pieToggles</code> option.
state:close	Triggered when the user closes the Halo (applicable for the <i>pie</i> type of Halo). The handler is passed the pie toggle name as defined in <code>pieToggles</code> option.
close	Triggered when the halo gets closed.

JointJS+ Inspector

ui.Inspector

Inspector plugin creates an editor and viewer of cell model properties. It creates a two-way data-binding between the cell model and a generated HTML form with input fields. These input fields can be defined in a declarative fashion using a plain JavaScript object. Not only visual attributes can be configured for editing in the inspector but also custom data that will be set on the cell model! This all makes the inspector an extremely flexible and useful widget for use in applications.

Install

Include `joint.ui.inspector.css` and `joint.ui.inspector.js` files to your HTML:

```
<link rel="stylesheet" type="text/css" href="joint.ui.inspector.css"/>
<script src="joint.ui.inspector.js"></script>
```

Create an inspector

You can create the Inspector instance as follows:

```
var inspector = new joint.ui.Inspector(options).render();
inspector.$el.appendTo($('body'))
```

Usually the applications display only one inspector at the time, which is used for various application models and differs in the configuration of input fields only. For this common use-case we've introduced a static helper -

`joint.ui.Inspector.create()`. It makes sure the previous instance (if there is any) is properly removed, it creates a new one and renders it into the DOM. It also keeps track of open/closed groups and restores them based on the last used state. It can be used as follows:

```
joint.ui.Inspector.create('#inspector', options);
```

For more information about the `create` method visit the [Inspector API chapter](#).

```
paper.on('element:pointerdown', function(elementView) {
  // open the inspector when the user interacts with an element
  joint.ui.Inspector.create('#inspector', {
    cell: cellView.model,
    inputs: {
      attrs: {
        circle: {
          fill: {
            type: 'color-palette',
            options: [
```

```
        { content: '#FFFFFF' },
        { content: '#FF0000' },
        { content: '#00FF00' },
        { content: '#0000FF' },
        { content: '#000000' }
    ],
    label: 'Fill color',
    group: 'presentation',
    index: 1
},
stroke: {
    type: 'color-palette',
    options: [
        { content: '#FFFFFF' },
        { content: '#FF0000' },
        { content: '#00FF00' },
        { content: '#0000FF' },
        { content: '#000000' }
    ],
    label: 'Outline color',
    group: 'presentation',
    index: 2
},
'stroke-width': {
    type: 'range',
    min: 0,
    max: 50,
    unit: 'px',
    label: 'Outline thickness',
    group: 'presentation',
    index: 3
}
},
text: {
    text: {
        type: 'textarea',
        label: 'Text',
        group: 'text',
        index: 1
    },
    'font-size': {
        type: 'range',
        min: 5,
        max: 30,
        label: 'Font size',
        group: 'text',
        index: 2
    }
}
```

```

    },
    'font-family': {
      type: 'select',
      options: ['Arial', 'Times New Roman', 'Courier New'],
      label: 'Font family',
      group: 'text',
      index: 3
    }
  }
},
groups: {
  presentation: {
    label: 'Presentation',
    index: 1
  },
  text: {
    label: 'Text',
    index: 2
  }
}
});
});

```

Configuration

There are two ways to create an instance of the inspector. The first option is to use the `joint.ui.Inspector.create(container, options)` static function (where `container` is a CSS selector of an HTML element on your page). The second option is to directly create an instance with `new joint.ui.Inspector(options)`. The inspector can be configured by the `options` object with the following properties:

cellView	<i>joint.dia.CellView</i>	(Mandatory - alternative 1) An ElementView or LinkView which you want to inspect. (Mutually exclusive with <code>cell</code> option.)
cell	<i>Backbone.Model</i>	(Mandatory - alternative 2) An arbitrary Backbone model which you want to inspect (i.e. an Element, Link or Graph). (Mutually exclusive with the <code>cellView</code> option.)
inputs	<i>object</i>	An object that mimics the structure of a cell model. Instead of the final values, it contains definitions of input fields. (The input field is in charge of setting the user input on the specified cell model.) <u>See below for further explanation.</u>
groups	<i>object</i>	An object that contains group identifiers as keys and group options as values. Each group may contain any number of

		input fields in the inspector panel. The user can show/hide the whole group by clicking a toggle button. <u>See below for further explanation.</u>
live	<i>boolean</i>	Should the Inspector update the cell properties immediately (in reaction to an <code>"onchange"</code> event triggered on a form input)? The default is <code>true</code>. If you need to prevent the Inspector from updating cell properties immediately when the user leaves their corresponding input field (for example, if you need to update cell properties only in reaction to the user pressing an <i>Update</i> button somewhere in your application), set this option to <code>false</code> instead. You then need to call the <code>`updateCell()`</code> <u>function</u> manually when appropriate.
multiOpenGroups	<i>boolean</i>	Is the user allowed to have multiple groups opened in the Inspector at the same time? The default is <code>true</code>. For the classical “accordion” type of Inspector (only one open group at a time), set this option to <code>false</code>.
container	<i>string Element jQuery</i>	A CSS selector, DOM element or jQuery object that is the container element which the <code>select-box</code> or <code>'color-palette'</code> options is appended to.
validateInput(element, path, type, inspector)	<i>function</i>	A callback function, called by Inspector fields to check whether user input was valid. The function should return <code>true</code> if the input had a valid value, and <code>false</code> if it did not. If <code>false</code> is returned, the input field does not save the input value to the cell model. See the <u>Validation chapter</u> for more information. The callback function is passed four arguments: <code>element</code> - a reference to the <code><input></code> HTML element the user interacted with <code>path</code> - path to the input field within the Inspector's <code>inputs</code> object (<code>'/'</code> separated) <code>type</code> - the type of the input field, as defined in its <code>options</code> object <code>inspector</code> - a reference to the current Inspector instance (for context)

		The default function checks the <code>validity</code> property of <code>element</code> : <pre>function(element, path, type, inspector) { return (element.validity ? element.validity.valid : true); }</pre>
renderFieldContent(options, path, value, inspector)	<i>function</i>	A callback function that returns an HTML, DOM element or jQuery object that will be appended by the Inspector into the space reserved for the field. In other words, this function allows you to define <u>custom fields</u>. The function is passed four arguments: <code>options</code> - the object provided in the <code>inputs</code> option of the Inspector as a definition of the field <code>path</code> - path to the input field within the Inspector's <code>inputs</code> object (<code>'/'</code> separated) <code>value</code> - the value read from the cell property at the corresponding path at the time of the rendering of the field. It also takes into account <code>defaultValue</code> and <code>valueRegExp</code> from the <code>options</code> object defined for this field <code>inspector</code> - a reference to the current Inspector instance (for context) If the function is defined but returns <code>undefined</code> in some cases, the Inspector will try to understand the field as if it were one of the built-in field types.
getFieldValue(attribute, type, inspector)	<i>function</i>	A callback function that returns an object of the form <code>{ value: [value read from a custom field] }</code>. This function is especially useful in combination with the <code>renderFieldContent()</code> option; it allows the Inspector to understand <u>custom fields</u>. The function is passed three arguments: <code>attribute</code> - the DOM element container of the Inspector field (i.e. the value returned by <code>renderFieldContent()</code> , if used) <code>type</code> - the type of the field, as defined in the field's <code>options</code> object

		<code>inspector</code> - a reference to the current Inspector instance (for context)
renderLabel(options, path, inspector)	<i>function</i>	A callback function that returns an HTML, DOM element or jQuery object that will be appended by the Inspector into the space reserved for field label. In other words, this function allows you to define <u>custom labels</u>. The function is passed three arguments: <code>options</code> - the object provided in the <code>inputs</code> option of the Inspector as a definition of the field <code>path</code> - path to the input field within the Inspector's <code>inputs</code> object (<code>'/'</code> separated) <code>inspector</code> - a reference to the current Inspector instance (for context)
focusField(options, path, element, inspector)	<i>function</i>	A callback function called by Inspector when trying to focus an element. This function allows you to define the focus behaviour for <u>custom fields</u>. The function is passed three arguments: <code>options</code> - the object provided in the <code>inputs</code> option of the Inspector as a definition of the field <code>path</code> - path to the input field within the Inspector's <code>inputs</code> object (<code>'/'</code> separated) <code>element</code> - the rendered <code>HTMLElement</code> of this field <code>inspector</code> - a reference to the current Inspector instance (for context)
stateKey(model)	<i>function</i>	A callback function that should return a unique identifier for saving/restoring the state of the groups (i.e. which groups are opened and which ones are closed). The default function returns the <code>id</code> of the current <code>Element</code>, i.e. every <code>Element</code> instance has its own group state. (So, Inspectors opened on different instances of one <code>Element</code> type will not remember previously opened/closed groups.) An alternative method, <code>function(model) { return model.get('type'); }</code> would store state per element type. Then, Inspectors of all <code>standard.Rectangle</code> <code>Elements</code> would share the same group state.

storeGroupsState	<i>boolean</i>	(Applicable only when used with the <i>create</i> method) Should group state be saved? (That is, should the Inspector remember which groups were opened and which groups were closed, when it is reopened?) The default is true . (Group state can be restored by the <code>restoreGroupsState</code> option when the Inspector is created. It defaults to true . See below.)
restoreGroupsState	<i>boolean</i>	(Applicable only when used with the <i>create</i> method) Should previous group state be restored (if any group state had been saved)? The default is true .
updateCellOnClose	<i>boolean</i>	(Applicable only when used with the <i>create</i> method) Should the current inspector values be saved to the <code>cell</code> when a new inspector is about to be created? The default is true .

Options properties `storeGroupsState` / `restoreGroupsState` are applicable only if they are passed into the static `create()` method. Otherwise, you can use the API methods `storeGroupsState()` and `restoreGroupsState()` to manually manipulate group states.

Inputs Configuration

The `inputs` object is extremely important. Its structure mimics the structure of properties of the cell model. Instead of the final values, it contains definitions of input fields. (The input field is in charge of setting the user input on the specified cell model.)

The options object of inputs can contain the following parameters:

type	string	(Mandatory) The type of the input field. The supported types are:													
		<table><tr><td rowspan="4">'number'</td><td colspan="3">creates an HTML 5 number input field. Special properties are:</td></tr><tr><td>property</td><td>default</td><td>description</td></tr><tr><td>min</td><td>- Infinity</td><td>The minimum value that is acceptable and valid for the input.</td></tr><tr><td>max</td><td>Infinity</td><td>The maximum value that is acceptable and valid for the input.</td></tr></table>	'number'	creates an HTML 5 number input field. Special properties are:			property	default	description	min	- Infinity	The minimum value that is acceptable and valid for the input.	max	Infinity	The maximum value that is acceptable and valid for the input.
'number'	creates an HTML 5 number input field. Special properties are:														
	property	default		description											
	min	- Infinity		The minimum value that is acceptable and valid for the input.											
	max	Infinity	The maximum value that is acceptable and valid for the input.												

'text'	creates a text input field.															
'textarea'	creates a textarea.															
'content-editable'	creates a content-editable div (resize automatically as user types). Special properties are: <table><tr><th>property</th><th>default</th><th>description</th></tr><tr><td>html</td><td>true</td><td>Is the content HTML or plain text?</td></tr><tr><td>readonly</td><td>false</td><td>Makes the field not mutable. The user can not edit the content.</td></tr></table>	property	default	description	html	true	Is the content HTML or plain text?	readonly	false	Makes the field not mutable. The user can not edit the content.						
property	default	description														
html	true	Is the content HTML or plain text?														
readonly	false	Makes the field not mutable. The user can not edit the content.														
'range'	creates an HTML 5 range input field. Special properties are: <table><tr><th>property</th><th>default</th><th>description</th></tr><tr><td>min</td><td>- Infinity</td><td>The minimum value that is acceptable and valid for the input.</td></tr><tr><td>max</td><td>Infinity</td><td>The maximum value that is acceptable and valid for the input.</td></tr><tr><td>step</td><td>1</td><td>A number that specifies the granularity that the value must adhere to.</td></tr><tr><td>unit</td><td>""</td><td>The name of the unit displayed next to the value.</td></tr></table>	property	default	description	min	- Infinity	The minimum value that is acceptable and valid for the input.	max	Infinity	The maximum value that is acceptable and valid for the input.	step	1	A number that specifies the granularity that the value must adhere to.	unit	""	The name of the unit displayed next to the value.
property	default	description														
min	- Infinity	The minimum value that is acceptable and valid for the input.														
max	Infinity	The maximum value that is acceptable and valid for the input.														
step	1	A number that specifies the granularity that the value must adhere to.														
unit	""	The name of the unit displayed next to the value.														
'color'	creates an HTML 5 color input field.															
'select'	creates a select box. Special properties are: <table><tr><th>property</th><th>default</th><th>description</th></tr></table>	property	default	description												
property	default	description														

options	[]	An array that contains the options for the select box
multiple	false	One or more values can be selected.
size	options.length	When <i>multiple</i> is specified, it is the number of rows in the list that should be visible at one time.

```
workdays: {
  type: 'select',
  options: ['Sun', 'Mon', 'Tue', 'Wed',
    'Thu', 'Fri', 'Sat'],
  multiple: true,
  overwrite: true,
  group: 'occupation'
}
```

'toggle'

creates a toggle (checkbox).

'select-box'

creates a [SelectBox component](#)

'color-
palette'

creates a [ColorPalette component](#)

'select-
button-
group'

creates a [SelectButtonGroup component](#)

'radio-
group'

creates a [RadioGroup component](#)

'object'

creates inputs for the properties of an object.

'list'

creates a widget for adding/removing items from an array.

		<table> <tr> <th>property</th><th>default</th><th>description</th></tr> <tr> <td>addButtonLabel</td><td>"+"</td><td>The label of the add list item button.</td></tr> <tr> <td>removeButtonLabel</td><td>"-"</td><td>The label of the remove list item button.</td></tr> </table>	property	default	description	addButtonLabel	"+"	The label of the add list item button.	removeButtonLabel	"-"	The label of the remove list item button.
property	default	description									
addButtonLabel	"+"	The label of the add list item button.									
removeButtonLabel	"-"	The label of the remove list item button.									
label	string	Label for the form input.									
group	string	Group the input belongs to.									
index	number	Index of the input within its group.									
defaultValue	any	The value that will be used in the input field in case the associated cell property is undefined .									
valueRegExp	string	A regular expression used to extract (and set) a property value on the cell. Use in combination with the <code>defaultValue</code> option to make sure the Inspector does not try to extract something from an undefined value.									
overwrite	boolean	Should the input value overwrite the current contents of the cell? The default is false, which means that the input attributes are merged with the model's current attributes. Example: <pre>// overwrite: false (default) Model: { existingProperty: 'value1' } Input: { newProperty: 'value2' } => Model: { newProperty: 'value2', existingProperty: 'value1' } // overwrite: true Model: { existingProperty: 'value1' } Input: { newProperty: 'value2' } => Model: { newProperty: 'value2' }</pre>									
options		Specific to several input types ('select' , 'select-box' , 'select-button-group' , 'color-palette' , 'radio-group'). Can be defined in several different ways:									

array<string>

A simple list of string values. For example:

```
['option1', 'option2']
```

array<object>

A list of value/content objects. For example:

```
// select & select-box
[
  { value: 'value1', content: 'option1'
},
  { value: 'value2', content: 'option2'
}
]
// color-palette
[
  { content: 'red', icon: 'image1.svg'
},
  { content: 'blue' }
]
// select-button-group
[
  {
    value: 'value1',
    content: 'option1',
    buttonWidth: 20,
    icon: 'image.png',
    iconSelected: 'image2.png',
    iconWidth: 20,
    iconHeight: 20
  }
]
```

string

A string path pointing to a `cell` property (which contains an array of options). For example:

```
'path'
=>
cell.prop('path')
=>
['option1', 'option2']
```

		object A specific object which provides options data for the field. See Source.
min	number	The minimum value of a range input. Specific to the 'range' input type. The minimum number of items in an array. Specific to the 'list' input type.
max	number	The maximum value of a range input. Specific to the 'range' input type. The maximum number of items in an array. Specific to the 'list' input type.
properties	object	An object containing definitions of the properties of an object. Specific to the 'object' type.
item	object	A definition of a generic item of a list. Specific for the 'list' type.
attrs	object	An object of the form <code><selector>: { <attributeName>: <attributeValue>, ... }</code> . This object allows you to set arbitrary HTML attributes on the generated HTML for this input field. Useful if you want to <i>mark</i> certain input fields or store some additional content in them (for example, to display a tooltip).
when	object	An object containing conditions that determine whether this input should be shown or hidden based on the values of other inputs. For more information see chapter expressions .
previewMode	boolean	Should “preview mode” be enabled on the widget? The default is <code>false</code> . Specific to several input types ('select-box' , 'select-button-group' , 'color-palette').

In preview mode, when the user hovers over one of the items in the widget (e.g. it being a dropdown item in the case of the 'select-box' type), the Inspector still changes the connected model but triggers the change with the `dry` flag set to `true`. You can then react to this by using:

```
myCell.on('change:myProperty', function(cell, change, opt) {
  if (opt.dry) {
    /* do something when in preview mode */
  } else {
    sendToDatabase(JSON.stringify(graph));
  }
});
```



```
    }  
  });
```

This is useful, for example, if your application wants to reflect the values of the hovered items in the diagram but does not want to store the change to the database.

Note that the `'list'` and `'object'` types allow you to create input fields in the Inspector for arbitrary nested structures. If your cell contains an object as a property (`cell.set('myobject', { first: 'John', last: 'Good' })`), you can instruct the inspector to use the `'object'` type for `myobject` property and then define types for each of the nested properties of that object:

```
var inspector = new joint.ui.Inspector({  
  cellView: cellView,  
  inputs: {  
    myobject: {  
      type: 'object',  
      properties: {  
        first: { type: 'text' },  
        last: { type: 'text' }  
      }  
    }  
  },  
  groups: {}  
});
```

Similarly, if your cell contains a list as a property (`cell.set('mylist', [{ first: 'John', last: 'Good' }, { first: 'Jane', last: 'Good' }])`), you can instruct the inspector to use the `'list'` type for `mylist` property and then define the type of the list item. Importantly, the `'list'` input type enables users to add and remove list items. In our example, the `mylist` item contains a nested object, so we need to define its properties as well:

```
var inspector = new joint.ui.Inspector({  
  cellView: cellView,  
  inputs: {  
    mylist: {  
      type: 'list',  
      item: {  
        type: 'object',  
        properties: {  
          first: { type: 'text' },  
          last: { type: 'text' }  
        }  
      }  
    }  
  },  
  groups: {}  
});
```

Options source configuration

You can specify `options` parameter of the field as an source object which contains following properties:

dependencies	string[]	An array of dependencies. Each dependency is the name of a cell property. When one of the properties changes, new data is loaded from the source callback. You can use wildcards for dependency path inside lists. To do that, a wildcard (<code>'\${index}'</code>) has to be placed within the path - it will be dynamically substituted for the actual index of the item inside which the dependency is being evaluated. Dependencies array can be omitted and in that case you can use <code>refreshSources()</code> or <code>refreshSource(path)</code> functions to update the options array.																															
source	function	A callback which takes object with callback data and returns array of options or a Promise which returns array of options (each option is an object specific for each field). The callback object parameter is following: <table><tr><td>model</td><td>dia.Cell</td><td colspan="2">The cell instance.</td></tr><tr><td>inspector</td><td>ui.Inspector</td><td colspan="2">The current ui.Inspector instance.</td></tr><tr><td>initialized</td><td>boolean</td><td colspan="2">true if it is the first source call of the field.</td></tr><tr><td>path</td><td>string</td><td colspan="2">The current field path.</td></tr><tr><td>dependencies</td><td>object</td><td colspan="2"> The key-value object with keys as dependencies specified in the <code>dependencies</code> field and value as an object with following properties: <table><tr><td>path</td><td>string</td><td>The path to the cell's property (wildcard replaced).</td></tr><tr><td>changedPath</td><td>string</td><td>An actual path that invoked source call (for example in the case of nested fields of an object or list).</td></tr><tr><td>value</td><td>any</td><td>A value of the</td></tr></table></td></tr></table>			model	dia.Cell	The cell instance.		inspector	ui.Inspector	The current ui.Inspector instance.		initialized	boolean	true if it is the first source call of the field.		path	string	The current field path.		dependencies	object	The key-value object with keys as dependencies specified in the <code>dependencies</code> field and value as an object with following properties: <table><tr><td>path</td><td>string</td><td>The path to the cell's property (wildcard replaced).</td></tr><tr><td>changedPath</td><td>string</td><td>An actual path that invoked source call (for example in the case of nested fields of an object or list).</td></tr><tr><td>value</td><td>any</td><td>A value of the</td></tr></table>		path	string	The path to the cell's property (wildcard replaced).	changedPath	string	An actual path that invoked source call (for example in the case of nested fields of an object or list).	value	any	A value of the
model	dia.Cell	The cell instance.																															
inspector	ui.Inspector	The current ui.Inspector instance.																															
initialized	boolean	true if it is the first source call of the field.																															
path	string	The current field path.																															
dependencies	object	The key-value object with keys as dependencies specified in the <code>dependencies</code> field and value as an object with following properties: <table><tr><td>path</td><td>string</td><td>The path to the cell's property (wildcard replaced).</td></tr><tr><td>changedPath</td><td>string</td><td>An actual path that invoked source call (for example in the case of nested fields of an object or list).</td></tr><tr><td>value</td><td>any</td><td>A value of the</td></tr></table>		path	string	The path to the cell's property (wildcard replaced).	changedPath	string	An actual path that invoked source call (for example in the case of nested fields of an object or list).	value	any	A value of the																					
path	string	The path to the cell's property (wildcard replaced).																															
changedPath	string	An actual path that invoked source call (for example in the case of nested fields of an object or list).																															
value	any	A value of the																															

Here is the example of a field declaration using source object

```
inputs: {
  array: {
    type: 'list',
    group: 'data',
    item: {
      type: 'object',
      properties: {
        value: {
          type: 'radio-group',
          options: {
            dependencies: ['array/${index}/options'],
            source: (data) => {
              const { path, value, changedPath } =
data.dependencies['array/${index}/options'];
              const options = value;
              if (Array.isArray(options)) {
                return options.map(name => {
                  return { value: name, content: name };
                });
              }
              return [];
            }
          }
        },
        options: {
          type: 'list',
          item: {
            type: 'text'
          }
        }
      }
    }
  }
}
```

In this example we are defining two inputs inside of an element's `array` property. The first one is the `value` property with source options parameter. It depends on the sibling `options` field which is a list of textboxes. The source function of `value` property triggers on any change inside of the `options` list. So in the end radio group will be populated with options from the `options` field array.

Groups Configuration

Each group options object in the `groups` object can contain the following parameters:

label	<i>string</i>	A label for the group. This label will be displayed as a header of the group section in the accordion-like inspector.
index	<i>number</i>	An index of the group relative to other groups. Use this to put the groups in a certain order.
closed	<i>boolean</i>	If set to <code>true</code> , the group will be closed by default.
when	<i>object</i>	An object containing conditions that determine if this group should be shown or hidden based on the values of other inputs. For more information see chapter expressions .

Expressions

The inspector relies on expressions defined in the `when` parameter to switch the visibility of an input field based on the values of other inputs. Whenever an input field's expression is evaluated to `false` (meaning “the condition is not met”), the input field is hidden. Otherwise, the input field is shown.

Definition of Expressions

When evaluated in a model context, expressions return a boolean (`true` / `false`) based on the value of specified model attributes. Expressions are defined recursively as follows:

- `{ <primitive>: { <path>: <value> }, <*options> }` is an expression
- `{ <unary-operator>: expression, <*options> }` is an expression
- `{ <multiary-operator>: [expression-1, expression-2, ..., expression-n], <*options> }` for $n > 0$ is an expression

...where:

path	Is a string determining a property of the model (e.g. <code>'attrs/text/text'</code> , <code>'property'</code> , <code>'myobject/nestedProperty'</code> , <code>'mylist/\${index}'</code> , <code>'mylist/\${index}/nestedProperty'</code>)			
value	Is a number, string or an array (e.g. <code>13</code> , <code>'jointjs'</code> , <code>[1, 3, 5]</code>)			
*options	(Optional) Additional options that affect the evaluation of the expression: <table><tr><td>otherwise</td><td>(Optional) What should happen if the expression evaluates to <code>false</code> ?</td></tr></table>		otherwise	(Optional) What should happen if the expression evaluates to <code>false</code> ?
otherwise	(Optional) What should happen if the expression evaluates to <code>false</code> ?			

		unset	If true , the input field is cleared when hidden. If the field has a <code>defaultValue</code> specified, the field is reverted to that value. (By default, when an input field is hidden, it remembers its user-submitted content and presents it again whenever it becomes visible.)
	dependencies	(Optional) An array of property paths on <code>inspector cell</code> which are necessary for the evaluation of custom operators in the expression. See the relevant chapter for more information.	

primitive	Can be one of the following simple operators:		
	eq	returns true if the value at <code><path></code> equals <code><value></code> (using <code>==</code> internally)...	
	equal	returns true if the value at <code><path></code> equals <code><value></code> (using <code>_.isEqual()</code> internally)...	
	ne	returns true if the value at <code><path></code> doesn't equal <code><value></code> (using <code>!=</code> internally)...	
	regex	returns true if the value at <code><path></code> matches a regular expression described in <code><value></code> ...	
	text	returns true if the value at <code><path></code> contains <code><value></code> as a substring...	
	lt	returns true if the value at <code><path></code> is less than <code><value></code> ...	
	lte	returns true if the value at <code><path></code> is less than or equal to <code><value></code> ...	
	gt	returns true if the value at <code><path></code> is greater than <code><value></code> ...	
	gte	returns true if the value at <code><path></code> is greater than or equal to <code><value></code> ...	
	in	returns true (only if <code><value></code> is an array) if the value at <code><path></code> is an element in <code><value></code> array...	
	nin	returns true (only if <code><value></code> is an array) if the value at <code><path></code> is not an element in <code><value></code> array...	
	...and returns false otherwise.		

unary_operator	Accepts exactly one expression <table border="1" data-bbox="400 268 1075 360"> <tr> <td data-bbox="400 268 493 360">not</td><td data-bbox="493 268 1075 360">returns the negation of the provided expression</td></tr> </table>	not	returns the negation of the provided expression				
not	returns the negation of the provided expression						
multiary_operator	Accepts an array of at least one expression <table border="1" data-bbox="400 506 1455 907"> <tr> <td data-bbox="400 506 493 640">and</td><td data-bbox="493 506 1455 640">returns a conjunction of all the provided expressions (i.e. <code>expr1 && expr2 && expr3</code>)</td></tr> <tr> <td data-bbox="400 640 493 775">or</td><td data-bbox="493 640 1455 775">returns a disjunction of all the provided expressions (i.e. <code>expr1 expr2 expr3</code>)</td></tr> <tr> <td data-bbox="400 775 493 907">nor</td><td data-bbox="493 775 1455 907">returns the negation of a disjunction of all the provided expressions (i.e. <code>!(expr1 expr2 expr3)</code>)</td></tr> </table>	and	returns a conjunction of all the provided expressions (i.e. <code>expr1 && expr2 && expr3</code>)	or	returns a disjunction of all the provided expressions (i.e. <code>expr1 expr2 expr3</code>)	nor	returns the negation of a disjunction of all the provided expressions (i.e. <code>!(expr1 expr2 expr3)</code>)
and	returns a conjunction of all the provided expressions (i.e. <code>expr1 && expr2 && expr3</code>)						
or	returns a disjunction of all the provided expressions (i.e. <code>expr1 expr2 expr3</code>)						
nor	returns the negation of a disjunction of all the provided expressions (i.e. <code>!(expr1 expr2 expr3)</code>)						

Examples on using Expressions

Here are a few valid expressions:

```
{ eq: { 'size/width': 300 }}
{ regex: { 'attrs/text/text' : 'JointJS|Rappid' }}
{ lt: { 'count': 10 }}
{ in: { 'index': [0,2,4] }}
{ not: { eq: { 'day': 'Monday' }}}
{ and: [{ gte: { 'position/x': 100 }}, { lte: { 'position/x': 400 } }]}
```

Imagine a scenario where you have a 'select' input field with options 'email' and 'tel'. Below this input field, you want to show either a text field or a number input field, based on the selected option. Assuming your cell properties structure is as follows: { contact_option: 'email', contact_email: '', contact_tel: '' }, your inspector could look like this:

```
var inspector = new joint.ui.Inspector({
  cell: mycell,
  inputs: {
    contact_option: { type: 'select', options: ['email', 'tel'] },
    contact_email: { type: 'text', when: { eq: { 'contact_option': 'email' }}}},
    contact_tel: { type: 'number', when: { eq: { 'contact_option': 'tel' }}}
  }
});
```

It is also possible to refer to input fields inside nested objects, by using more complicated paths:

```

var inspector = new joint.ui.Inspector({
  cell: mycell,
  inputs: {
    user_info: {
      type: 'object',
      properties: {
        contact_option: { type: 'select', options: ['email', 'tel'] },
        name: { type: 'text' }
      }
    },
    contact_email: { type: 'text', when: { eq: { 'user_info/contact_option': 'email' } } },
    contact_tel: { type: 'number', when: { eq: { 'user_info/contact_option': 'tel' } } }
  }
});

```

It does not make sense to reference list items from the outside, but it does make sense to reference sibling input fields within a list item's `when` clause. To do that, a wildcard (`'${index}'`) has to be placed within the path - it will be dynamically substituted for the actual index of the item inside which the `when` clause is being evaluated:

```

var inspector = new joint.ui.Inspector({
  cell: mycell,
  inputs: {
    user_list: {
      type: 'list',
      item: {
        type: 'object',
        properties: {
          contact_option: { type: 'select', options: ['email', 'tel'] },
          contact_email: { type: 'text', when: { eq: { 'user_list/${index}/contact_option': 'email' } } },
          contact_tel: { type: 'number', when: { eq: { 'user_list/${index}/contact_option': 'tel' } } }
        }
      }
    }
  }
});

```

It is also possible to toggle groups with `when` expressions.

```

var inspector = new joint.ui.Inspector({
  groups: {
    first: { label: 'F' },
    second: { label: 'S', when: { eq: { 'attribute2': true } } }
  }
});

```

```
}  
});
```

Custom Operators in Expressions

As you can see [above](#), `ui.Inspector` provides a good list of useful built-in primitive operators (`eq` , `lt` , `in` , ...). However, sometimes this is not enough and applications have special requirements for when fields in the inspector should be hidden/displayed based on other information. To meet this requirement while still taking advantage of the inspector configurability through expressions, `ui.Inspector` provides a way to define your own custom operators.

First, the custom primitive operator has to be defined inside the `operators` array option on the Inspector. Each operator definition is an object of the form: `{ custom-primitive: function(cell, value, *arguments) }`. The provided callback function should return `true` when the operator condition is successful, and `false` otherwise. The `cell` parameter is the cell associated with the Inspector, `value` is the value of the field at the `path` specified in the `when` clause (see below), and `*arguments` is anything that was passed to the operator (see below).

Second, in the `when` clause of a definition within the Inspector's `inputs` object, an expression with a custom operator has to be defined according to the following format:

- `{ <custom-primitive>: { <path>: <*arguments> }, <*options> }`

The custom primitive expression can be used as an operand in unary and multiary operators, the same as if it was a built-in [primitive expression](#).

For example, let's say you want to show an inspector field only when a value of another input field is longer (has more characters) than the value of a numeric property set on the associated inspector cell:

```
var inspector = new joint.ui.Inspector({  
  cell: mycell,  
  inputs: {  
    title: { type: 'text' },  
    description: { type: 'text', when: { longerThan: { 'title': 'titleThreshold' } } },  
  },  
  operators: {  
    longerThan: function(cell, value, prop, valuePath) {  
      // value === contents of 'title' input field  
      // prop === 'titleThreshold'  
      // valuePath === 'title'  
      return (value ? (value.length > cell.prop(prop)) : false);  
    }  
  }  
});
```

The example above displays the `description` field only when the content of `title` is longer than a numeric threshold which we have stored in a property on the cell model named `titleThreshold`. Now whenever the user types within the `'title'` input field in the inspector and the text becomes longer than `cell.get('titleThreshold')`, the `description` field appears (and vice versa, if the text becomes shorter than `titleThreshold`, the `description` field gets hidden).

However, the example above has a small problem. If the value of the `titleThreshold` property changes on the cell model (e.g. due to some other change somewhere else in the application), that change is not taken into account by the expression. In order to fix this, we have to tell the inspector that there are prop dependencies that could affect the resolution of the expression in the `when` clause - we do that by providing a `dependencies` list inside the `when` clause. Here's the fixed version of the code provided above:

```
var inspector = new joint.ui.Inspector({
  cell: mycell,
  inputs: {
    title: { type: 'text' },
    description: {
      type: 'text',
      when: {
        longerThan: { 'title': 'titleThreshold' },
        dependencies: ['titleThreshold']
      }
    },
  },
  operators: {
    longerThan: function(cell, value, prop, valuePath) {
      // value === contents of 'title' input field
      // prop === 'titleThreshold'
      // valuePath === 'title'
      return (value ? (value.length > cell.prop(prop)) : false);
    }
  }
});
```

Validation

The following example shows how to reflect a custom shape's validation functions inside your Inspector. The inspector definition provides a custom `validateInput` method. That method then refers to Shape's custom `validateProperty` method, which uses some common regex validators. Notice that this architecture makes the Shape (the model instance) responsible for accepting/rejecting user input data, not the Inspector. As such, this arrangement manages to fulfill one of the goals of the JointJS framework, namely the separation of Model-View-Controller components from each other.

```
(function(joint, Shape) {

  joint.setTheme('modern');

  var paper = new joint.dia.Paper({
    el: document.getElementById('paper'),
    width: 500,
    height: 500
  });
```

```

var shape = new Shape();
shape.position(150,50);
shape.size(200,200);
shape.addTo(paper.model);

var inspector = joint.ui.Inspector.create('#inspector', {
  cell: shape,
  inputs: shape.getInspectorDefinition(),
  validateInput: function(el, path, type, inspector) {
    var $el = $(el);
    var value = inspector.parse(type, inspector.getFieldValue(el, type), el);
    $el.removeClass('error').parent().find('error').remove();
    var error = shape.validateProperty(path, value);
    if (error) {
      var $error = $('<error/>').text(error);
      $el.addClass('error').before($error);
    }
    return !error;
  }
});

// run the first validity check
inspector.updateCell();

})(joint, joint.dia.Element.define('Shape', {

  attrs: {
    body: {
      refWidth: '100%',
      refHeight: '100%',
      fill: '#dddddd',
      stroke: 'lightblue',
      strokeWidth: 2
    }
  },

  phoneNumber: '',
  emailAddress: 'org@client.io'

}, {

  markup: [{
    tagName: 'rect',
    selector: 'body'
  }],

  REGEX_PHONE_NUMBER: /^(\\+\\d{1,2}\\s)?\\((?\\d{3}\\)?[\\s.-]\\d{3}[\\s.-]\\d{4}$/,

```

```

REGEX_HEXCOLOR: /^#([a-f0-9]{3}){1,2}\b/i,
REGEX_EMAIL_ADDRESS: /^\\w+@[a-zA-Z_]+?\\. [a-zA-Z]{2,3}$/,

validateProperty: function(path, value) {
  switch (path) {
    case 'attrs/body/stroke':
    case 'attrs/body/fill':
      if (this.REGEX_HEXCOLOR.test(value)) break;
      return 'Invalid Color (e.g. #ff0000)';
    case 'attrs/body/strokeWidth':
      if (!_.isNumber(value) || value <= 0) break;
      return 'Invalid Stroke Width (A positive number)';
    case 'phoneNumber':
      if (this.REGEX_PHONE_NUMBER.test(value)) break;
      return 'Invalid Phone Number (e.g. 123-456-7890)';
    case 'emailAddress':
      if (this.REGEX_EMAIL_ADDRESS.test(value)) break;
      return 'Invalid Email Address.';
  }
  return null;
},

getInspectorDefinition: function() {
  return {
    'attrs/body/fill': {
      type: 'text',
      label: 'Fill Color'
    },
    'attrs/body/stroke': {
      type: 'text',
      label: 'Stroke Color'
    },
    'attrs/body/strokeWidth': {
      type: 'number',
      label: 'Stroke Width'
    },
    'phoneNumber': {
      type: 'text',
      label: 'Phone Number'
    },
    'emailAddress': {
      type: 'text',
      label: 'Email Address'
    }
  };
}

```

```
}});
```

Custom Fields

The Inspector has a useful built-in set of ready-to-use [field types](#). However, in some cases, you might want to render your own custom fields (or to integrate a third party widget) while still taking advantage of the two-way data binding and the configuration options provided by Inspector. This can be done with two [Inspector options](#):

[renderFieldContent](#)(options, path, value) and [getFieldValue](#)(attribute, type).

The following example shows how to render two buttons in a single custom field and how to integrate the [Select2 widget](#) for advanced select boxes:

```
function createInspector(cellView) {

    joint.ui.Inspector.create('.inspector-container', {
        cellView: cellView,
        inputs: {
            attrs: {
                label: {
                    style: {
                        textDecoration: {
                            label: 'Text Style',
                            type: 'select2',
                            group: 'text-decoration',
                            options: ['none', 'underline', 'overline', 'line-
through']
                        }
                    },
                },
                text: {
                    label: 'Text Content',
                    type: 'my-button-set',
                    group: 'text'
                }
            }
        },
        groups: {
            'text-decoration': { label: 'Text Decoration (Select2)' },
            'text': { label: 'Text' }
        },

        renderFieldContent: function(options, path, value, inspector) {

            switch (options.type) {

                case 'my-button-set':
```

field attribute

```
var $buttonSet = $('
```

```

    }

    if (type === 'select2') {
        const $select = $(attribute).find('.select2').data('element');
        return { value: $select.val() };
    }
}
});
}

```

source code

Custom Labels

It is also possible to customize the appearance and behavior of field labels in your Inspector. This can be used to create labels with custom HTML - as demonstrated by the `myList` label (an `<a>` tag with a `<label>` tag inside), which links to an address specified in `myList.url`. Additionally, our templating functionality can be used to define dynamic labels. We use this for `myList.item` elements; when an element is added to the array with the `+` button, the `'{{index}}'` placeholder is replaced with the element's actual index within `myList`.

```

joint.ui.Inspector.create('#container', {
  cell: model,
  inputs: {
    myList: {
      type: 'list',
      url: 'https://jointjs.com',
      item: {
        type: 'text',
        label: 'Item {{index}}'
      }
    }
  },
  renderLabel: function(opt, path) {
    // returns an HTMLElement (`$el`) as a custom label
    // this method is called for every element inside inspector when being
    rendered
    // in this case, it is called when the example loads, to create a label for
    the whole `myList`
    // it is also called whenever the + button is pressed, to create a label for
    each element added to `myList`

    var $label = document.createElement('label');
    var $el = $label;

    // List numbering:
    var text = opt.label;
    var indexPlaceholder = '{{index}}';
  }
});

```

```

        // is this an inspector element with a `label` text specified (i.e. one of
`myList` items)?
        // does this inspector element contain a substring to replace?
        if (text && text.indexOf(indexPlaceholder) > -1) {

            // every input field in the inspector is addressed via a `path`
            // elements added to `myList` are addressed as 'myList/0', 'myList/1',
etc.

            // we can use this in a regex to identify list elements
            // we do this by checking if there is a '/' followed by a digit at the
end of the path
            var match = path.match(/\/(\d+)$/);
            if (match) {

                // if this is a list element, use its index as index
                // (+1 to make sure the rendered labels start from 1)
                var index = parseInt(match[1], 10) + 1;
                // actually add the human-readable index to the label
                text = text.replace(indexPlaceholder, index);
            }
        }
        // then, set the text of the label to the text we generated

        // else: set it to the raw `path` string
        // (this happens for `myList` itself - so it gets a label that says 'myList')
        $label.textContent = text || path;

        // item labels are hidden via CSS by default, we need to unhide them
        $label.style.cssText = 'display:block !important;';

        // Clickable label:
        // is this an inspector element with an `url` specified (i.e. `myList`
itself)?
        if (opt.url) {
            // then create a new <a> element and add <label> to it
            var $a = document.createElement('a');
            $a.href = opt.url;
            $a.target = 'blank';
            $a.appendChild($el);
            $el = $a;
        }

        // we have created a custom label
        return $el;
    }
});

```

Inspector Events

The Inspector object triggers events when the user changes its inputs or when the Inspector needs to re-render itself partially. These events can be handled by using the Inspector `on()` method.

render	Triggered when the Inspector re-renders itself partially. If you're adding event listeners or your own custom HTML into the inspector, you should always do it inside the <code>`render`</code> event handler.
change:[path to the changed property]	Triggered when the user changes a property of a cell through the inspector. The handler is passed the value at the property path and the input DOM element as arguments.
close	Triggered when the inspector gets closed.

Inspector API

static

create(container, options)	A helper for creating the inspector, where <code>container</code> is an <code>HTMLElement</code> or a selector (<code>container</code> is a DOM placeholder into which the Inspector will be rendered). For more information about <code>options</code> , see the Configuration chapter . An instance of <code>joint.ui.Inspector</code> is returned.
close()	A helper for closing Inspector instances which were created via the <code>create()</code> method above.

public

render()	Render the Inspector based on the options passed to the constructor function. Note that this does not add the inspector to the live DOM tree. This must be done manually by appending the Inspector DOM element (accessible as its <code>el</code> property) to a live DOM element on the page.
updateCell()	Manually update the associated cell based on the current values in the Inspector. This is especially useful if you use the inspector with <code>live</code> mode disabled. See the Configuration chapter for more information.
remove()	Remove the Inspector from the DOM. This should be called once you're finished with using the Inspector.
focusField(path)	Focuses the field identified by the <code>'/'</code> separated <code>path</code> .
openGroup(name)	Open the group identified by <code>name</code> .

closeGroup(name)	Close the group identified by <code>name</code> .
toggleGroup(name)	Toggle the group identified by <code>name</code> .
openGroups()	Open all groups.
closeGroups()	Close all groups.
storeGroupsState()	Save the current group state - which groups are opened and which are closed. The key for storing the state is determined by the <code>stateKey</code> Inspector option .
restoreGroupsState()	Apply the stored group state - open and close groups according this state. The state information is looked up by the current <code>stateKey</code> Inspector option .
getGroupsState()	Get the current group state - array of closed groups.
refreshSources()	Refresh sources of all controls.
refreshSource(path)	Refresh source of a control by its path.

JointJS+ Keyboard

ui.Keyboard

The `ui.Keyboard` is an essential plugin for creating keyboard shortcuts in your JointJS+ application. It allows you to bind custom handlers (functions) to arbitrary keyboard events (e.g a single keystroke, key combination).

Configuration

The keyboard constructor accepts the following options:

<code>filter(event, keyboard)</code>	Prevent keyboard events from triggering. The callback will be invoked whenever the keyboard <code>DOM event</code> is fired. If the function returns <code>false</code>, the keyboard event will not be not fired. The default filter is <code>isConsumerElement()</code>. <pre>const keyboard = new ui.Keyboard({ filter: (evt, keyboard) => { const { target } = evt; if (target.tagName === 'INPUT' && target.type === 'range') { // Prevent range inputs from consuming events. return true; } // Otherwise return the default behavior return !keyboard.isConsumerElement(evt); } });</pre>

API

<code>on(event, callback, [context])</code>	Bind a callback function to an object. The callback will be invoked whenever the keyboard <code>event</code> is fired. It is the same logic as Backbone.Events.on . More information about <code>event</code> can be found in the Event description section.
<code>off(event, callback, [context])</code>	Remove a previously-bound callback function from an object. It is the same logic as Backbone.Events.off . More information about <code>event</code> can be found in the Event description section.
<code>enable()</code>	Start tracking keyboard events (enabled by default).

disable()	Stop tracking keyboard events.
isActive(name, event)	Check if key <code>name</code> is currently pressed. <code>event</code> is the DOM Event <code>KeyboardEvent</code> . It is available for modifiers only - <code>alt</code> , <code>ctrl</code> , <code>shift</code> , <code>command</code>
isKeyPrintable(event)	Check if the key pressed is a printable character. <code>event</code> is the DOM Event <code>KeyboardEvent</code> .
isConsumerElement(event)	Checks if the element onto which the <code>event</code> was dispatched should consume this event and thus prevent the <code>ui.Keyboard</code> event from firing. <code>event</code> is the DOM Event <code>KeyboardEvent</code> . The method returns <code>true</code> for <i>inputs</i> , <i>selects</i> , <i>text-areas</i> and elements defined as <i>content-editable</i> .

Event description

The shortcut can be defined as a `string`, either a single keystroke e.g. `enter`, `esc`, `up`, `down`, `left` or a combination like `ctrl+c`, `ctrl+alt+t` etc. If you need to be more specific - in terms of `keydown`, `keyup`, `keypress` events - you can also define the shortcut as `keydown:a`, `keyup:enter` or even `keyup:ctrl+x`.

Multiple shortcuts bound to a single handler can be defined as a list of shortcuts, separated by a space.

```
keyboard.on('ctrl+v shift+insert', pasteHandler);
```

Please note that all Backbone event methods also support an event map syntax.

```
keyboard.on({
  'enter': addNewHandler,
  'ctrl+enter' : createNewHandler,
  'up down left right': moveHandler
});
```

Demo

JointJS+ Lightbox

ui.Lightbox

`ui.Lightbox` is a UI component for displaying images by filling the screen and dimming out the rest of the application. `ui.Lightbox` component nicely auto-adjusts based on the size of the screen.

Installation

Include `joint.ui.lightbox.js` and `joint.ui.lightbox.css` files and `joint.ui.dialog.js` and `joint.ui.dialog.css` dependencies (`ui.Lightbox` inherits from `ui.Dialog`) to your HTML:

```
<link rel="stylesheet" type="text/css" href="joint.ui.dialog.css">
<link rel="stylesheet" type="text/css" href="joint.ui.lightbox.css">
<script src="joint.ui.dialog.js"></script>
<script src="joint.ui.lightbox.js"></script>
```

Usage

Create an object of `joint.ui.Lightbox` type and call the `open()` method in order to open the lightbox.

```
var lightbox = new joint.ui.Lightbox({
  title: 'Image caption goes here.',
  image: 'images/lightbox.png'
});
lightbox.open();
```

Another common usage is opening images in lightbox on click:

```
$('#img').on('click', function(evt) {
  new joint.ui.Lightbox({
    title: $(evt.target).attr('title'),
    image: $(evt.target).attr('src')
  }).open();
});
```

The `downloadable` option offers an easy way to download an image:

```
$('#img').on('click', function(evt) {
  new joint.ui.Lightbox({
    title: $(evt.target).attr('title'),
    image: $(evt.target).attr('src'),
    downloadable: true
  }).open();
});
```

```

    }).open();
  });

```

Configuration

The following table lists options that you can pass to the `ui.Lightbox` constructor function:

title	The image caption.
image	The path or URL to the image.
top	The distance from the top edge of the image to the top of the screen. Default is <code>100</code> .
windowArea	The maximum percentage of the screen covered by lightbox. Default is <code>0.8</code> .
closeAnimation	Set to <code>false</code> if you don't want the lightbox to be closed in an animated fashion.
closeButton	Set to <code>false</code> if you don't want to display the small close button in the top left corner of the lightbox. Note that the lightbox will still be closeable by clicking anywhere outside the lightbox area.
downloadable	If <code>true</code> , a default download button is added to the lightbox (after any <code>buttons</code> specified). When clicked, this button triggers the <code>downloadAction</code> (<code>'download'</code> by default).
fileName	The filename of downloaded images. By default, the downloaded images are named <code>'Image'</code> .
downloadAction	If the action <code>action:[downloadAction]</code> is triggered on the lightbox, the lightbox <code>image</code> is downloaded to the user computer. By default, the download action is <code>'download'</code> . Downloaded images will have <code>fileName</code> as filename.
buttons	Buttons to show under the lightbox <code>title</code> . Explained in ui.Dialog configuration . Specifically, a custom download button can be created by providing a button that has <code>downloadAction</code> as its <code>action</code> (<code>'download'</code> by default).

API

open()	Open the lightbox.
close()	Close the lightbox.
on(event, handler [, context])	Register a <code>handler</code> function that will be called whenever <code>event</code> is triggered. The optional <code>context</code> is an object that will be the context of the handler function (the <code>this</code>).

Events

The `ui.Lightbox` object triggers various events that you can react on. Events can be handled by using the `on()` method (see above).

close:animation:complete	Triggered when the lightbox is actually removed from the DOM, i.e. when the close animation finishes.
close	Triggered when the lightbox gets closed.

JointJS+ Navigator

ui.Navigator

`ui.Navigator` is a UI widget that displays a smaller view into a larger diagram. It's a quick way for users to navigate in diagrams with pannable and resizable rectangle.

Installation

Include `joint.ui.navigator.js` and `joint.ui.navigator.css` files to your HTML:

```
<link rel="stylesheet" type="text/css" href="joint.ui.navigator.css">
<script src="joint.ui.navigator.js"></script>
```

Usage

Simply create an object of `ui.Navigator` type, append the navigator HTML DOM element to any container and call the `render()` method. The navigator requires a [ui.PaperScroller](#) object as a parameter.

```
var nav = new joint.ui.Navigator({
  paperScroller: paperScroller,
  width: 300,
  height: 200,
  padding: 10,
  zoomOptions: { max: 2, min: 0.2 }
});
nav.$el.appendTo('#navigator');
nav.render();
```

Configuration

The following table lists options that you can pass to the `ui.Navigator` constructor function:

paperScroller	The paper scroller. See <u>ui.PaperScroller</u> plugin for more details.
width	The width of the navigator view.
height	The height of the navigator view.
padding	The initial padding between the small paper inside the navigator and the navigator view rectangle edges.

zoomOptions	An object with parameters controlling the zoom functions. <code>zoomOptions.min</code> and <code>zoomOptions.max</code> determine the minimum and the maximum zoom allowed when zooming via dragging the rectangle corner inside the navigator.
paperConstructor	The type of the paper. Defaults to <code>joint.dia.Paper</code> . This is only useful if you use a different paper type for rendering your graphics. In case you inherit from <code>joint.dia.Paper</code> to implement some specific rendering, you can just pass your constructor function to this parameter and use the <code>paperOptions</code> parameter to pass additional options to your special paper.
paperOptions	Options object that will be passed to the internal paper used for rendering graphics.
useContentBBox	When enabled the navigator shows only the content of the paper rather than entire scrollable area. An object with a single option can be passed here to calculate the content size from the model <code>{ useModelGeometry: true }</code> . It defaults to <code>false</code> .

API

render()	Render the navigator. You should call this method after you have appended the navigator view root HTML element to your container. See the Usage section.
remove()	Remove the navigator and clean up all its registered event handlers. Call this once you do not need the navigator anymore.
updateCurrentView()	Updates the navigator's viewport based on the associated paperScroller state. The method is debounced (the invoking is delayed until after 0 milliseconds have elapsed since the last time the method was invoked) and is called automatically when the paperScroller is scrolled or zoomed.
freeze(opt)	Freeze the paper of the navigator. In this state, the paper does not automatically re-render upon changes in the graph. This is useful when for instance the navigator is collapsed (not visible on the screen). For more information see paper.freeze() .
unfreeze(opt)	Unfreeze the paper of the navigator. For more information see paper.unfreeze() .

Events

The `joint.ui.Navigator` object triggers various events that you can react on:

pan:start	Triggered when the user starts panning the viewport. The handler is passed an <code>evt</code> (the mousedown event).

pan:stop	Triggered when the user stops panning the viewport. The handler is passed an <code>evt</code> (the mouseup event).
zoom:start	Triggered when the user starts zooming the viewport. The handler is passed an <code>evt</code> (the mousedown event).
zoom:stop	Triggered when the user stops zooming the viewport. The handler is passed an <code>evt</code> (the mouseup event).

JointJS+ PaperScroller

ui.PaperScroller

Paper scroller wraps the paper element and implements scrolling, panning, centering and auto-resizing of the paper content.

Installation

Include `joint.ui.paperScroller.js` and `joint.ui.paperScroller.css` files to your HTML:

```
<link rel="stylesheet" type="text/css" href="joint.ui.paperScroller.css">
<script src="joint.ui.paperScroller.js"></script>
```

Create a PaperScroller

PaperScroller creates a window to the paper area. This way, the actual paper can be large but the user will see only the PaperScroller area which he can scroll & pan. First, we create a PaperScroller object. Then we pass the automatically created PaperScroller element as a container for our paper. Once we have our paper object created, we pass it back to the PaperScroller as it needs a reference to the actual paper.

```
var paper = new joint.dia.Paper({
  width: 2000,
  height: 2000,
  model: graph
});

var paperScroller = new joint.ui.PaperScroller({
  paper: paper
});

$('#paper-container').append(paperScroller.render().el);
```

The next step is to hook the `startPanning` method of the paper scroller on the `blank:pointerdown` paper event which will cause panning whenever the user drags a blank area of the paper:

```
paper.on('blank:pointerdown', paperScroller.startPanning);
```

In order to help users understand what action will happen when they drag the paperScroller, use `setCursor(cursor)` method. It sets the mouse cursor, which is displayed when the mouse pointer is over the PaperScroller. It accepts any valid CSS *cursor* property value. Alternatively the cursor can be passed to the constructor. It defaults to `'default'` (an arrow).

```
var paperScroller = new joint.ui.PaperScroller({
  paper: paper,
  cursor: 'grab'
});
// Or dynamically change cursor e.g. when user is about to select elements.
paperScroller.setCursor('crosshair');
```

Padding

The default padding around the paper's content. The padding ensures the paper inside the PaperScroller is always pannable. It also makes sure that there is always at least a fragment of the paper visible. Padding can be a number, an object, or a function.

```
const scroller = new joint.ui.PaperScroller({
  paper: paper,
  // Default Padding function
  padding: function() {

    var clientSize = this.getClientSize();
    var minVisibleSize = Math.max(this.options.minVisiblePaperSize, 1) || 1;
    var padding = {};

    padding.left = padding.right = Math.max(clientSize.width - minVisibleSize, 0);
    padding.top = padding.bottom = Math.max(clientSize.height - minVisibleSize, 0);

    return padding;
  }
});

/*
  Example Usage
  padding: 10
  padding: { top: 20, left: 20 }
  padding: function() { return 10; }
*/
```

Scrolling the paper

User scrolling of the paper is allowed by default. Two methods are provided to toggle this functionality:

lock()	Lock the current viewport by disabling user scrolling.

unlock()	Enable user scrolling if previously locked.

PaperScroller also provides methods for programmatic scrolling. These methods try scroll the paper so that a specified point on the paper lies at the center of the paper scroller viewport. If this is not possible (e.g. if the requested point lies in a corner of the paper), the paper is scrolled as far as possible without requiring paddings. This means that the requested point will not actually move into the center of the viewport if there is not enough room for it. (Use the `center` [functions](#) to add paddings if necessary and force the requested point to move to the center of paper scroller viewport.)

All `scroll` methods optionally support animation if an `opt.animation` object is provided. The object may contain jQuery animation properties, most notably `opt.animation.duration` - a number specifying the time, in milliseconds, determining how long the animation will run. (Note that animated transitions are better supported with the `transition` [functions](#).)

scroll(x, y [, opt])	Try to scroll the paper scroller so that the position (x , y) on the paper (in local coordinates) is at the center of the paper scroller viewport. If only one of the coordinates is specified, only scroll in the specified dimension and keep the other coordinate unchanged. <pre> paperScroller.scroll(100, 200); // scroll to the point (100,200) paperScroller.scroll(100); // scroll to the point (100,[y of center of visible area]) paperScroller.scroll(null, 200); // center at the point ([x of center of visible area],200) paperScroller.scroll(100, 200, { animation: { duration: 400 }}); paperScroller.scroll(100, null, { animation: { duration: 200, easing: 'linear' }}); paperScroller.scroll(null, 200, { animation: { duration: 600 }}); </pre>
scrollToContent([opt])	Try to scroll the paper scroller so that the center of paper content is at the center of the paper scroller viewport. <pre> paperScroller.scrollToContent(); paperScroller.scrollToContent({ animation: { duration: 600 }}); </pre>

<code>scrollToElement(element [, opt])</code>	Try to scroll the paper scroller so that the center of <code>element</code> (an instance of <code>joint.dia.Element</code>) is at the center of the paper scroller viewport. <pre>paperScroller.scrollToElement(element); paperScroller.scrollToElement(element, { animation: { duration: 600 }});</pre>

The `PaperScroller` constructor takes an optional parameter `scrollWhileDragging`. When this parameter is set to `true`, the `PaperScroller` automatically scrolls the paper to ensure that the dragged shape stays within the viewport. This scrolling behavior can be adjusted further by providing the following options:

ScrollWhileDragging			
option	type	default	description
interval	<i>number (milliseconds)</i>	25	A debounce interval between changes in the <code>scrollLeft</code> and <code>scrollTop</code> properties of the <code>PaperScroller</code> , in <i>ms</i> .
padding	<i>number (pixels)</i>	-20	Additional padding to add to the viewport area until scrolling is triggered. A negative value means that scrolling is triggered when the dragging pointer approaches the viewport boundary from the inside. A value of <code>0</code> means that scrolling is triggered when the dragging pointer crosses the viewport boundary. A positive value means that scrolling is triggered when the dragging pointer is that much farther from the viewport boundary.
scrollingFunction	<i>function</i>	<pre>(distance) => ((distance < 20) ? 5 : 20)</pre>	A function that translates the distance between the pointer and the viewport boundary into the number of pixels by which to scroll. The default function scrolls slowly while the pointer is within the viewport and goes faster when the pointer leaves the viewport.

Another optional parameter is `inertia`. If set to `true`, panning of the `PaperScroller` will have an inertia effect where the scroller builds momentum while dragging and continues moving in the same direction for some time. The parameter can be also enabled with an object containing the following properties.

inertia			
option	type	default	description
friction	<i>number</i>	<i>0.92</i>	Rate at which values slow down after the user releases the pointer. A value between 0 and 1.

The optional parameter `borderless` changes the default behavior of the padding. If set to `true`, the paper inside the `paperScroller` extends beyond the padding, i.e. the blank space around the paper is converted to the paper itself making the borders of the paper visible no more.

Centering and positioning paper content

The `center` methods are more aggressive than the `scroll` methods. These methods position the paper so that a specific point on the paper lies at the center of the paper scroller viewport, adding paddings around the paper if necessary (e.g. if the requested point lies in a corner of the paper). This means that the requested point will always move into the center of the viewport. (Use the `scroll` [functions](#) to avoid adding paddings and only scroll the paper scroller viewport as far as the paper boundary.)

The `position` methods are a more general version of the `center` methods. They position the paper so that a specific point on the paper lies at requested coordinates inside the paper scroller viewport.

Padding can be added with `opt.padding`, to offset positioned elements away from the edges of client viewport. This stacks up with the `x` and `y` coordinates; it also affects the center calculations. The padding value is normalized using the `joint.util.normalizeSides` function; either a single number may be passed (e.g. `padding: 10` applies padding of 10px to all edges), or an object with numeric properties (e.g. `padding: { top: 20, horizontal: 10 }` applies padding of 20px to the top edge and 10px to the right and left edges).

<code>center([opt])</code>	If no coordinate is specified, position the center of paper to the center of the paper scroller viewport. <pre>paperScroller.center();</pre> Adding 100px of left padding causes the effective center of the paper scroller window to shift by 50px to the right. That is where the center of paper will be positioned. <pre>paperScroller.center({ padding: { left: 100 } });</pre>
<code>center(x, y [, opt])</code>	Position the point (<code>x</code> , <code>y</code>) on the paper (in local coordinates) to the center of the paper scroller viewport. If only one of the coordinates is specified, only center along the specified dimension and keep the other coordinate unchanged.

	<pre>paperScroller.center(100, 200); // center at the point (100,200) paperScroller.center(100); // center at the point (100, [y of center of visible area]) paperScroller.center(null, 200); // center at the point ([x of center of visible area],200)</pre> Adding 100px of left padding causes the effective center of the paper scroller window to shift by 50px to the right. That is where the point will be positioned. <pre>paperScroller.center(100, 200, { padding: { left: 100 } }); paperScroller.center(100, null, { padding: { left: 100 } }); paperScroller.center(null, 200, { padding: { left: 100 } });</pre>
centerContent([opt])	Position the center of paper content to the center of the paper scroller viewport. <pre>paperScroller.centerContent(); paperScroller.centerContent({ padding: { left: 100 } });</pre>
centerElement(element [, opt])	Position the center of <code>element</code> (an instance of <code>joint.dia.Element</code>) to the center of the paper scroller viewport. <pre>paperScroller.centerElement(element); paperScroller.centerContent(element, { padding: { left: 100 } });</pre>
positionContent(positionName [, opt])	Align paper content inside the paper scroller viewport as specified by <code>positionName</code>. There are 9 possible <code>positionName</code> values: <code>'top'</code>, <code>'top-right'</code>, <code>'right'</code>, <code>'bottom-right'</code>, <code>'bottom'</code>, <code>'bottom-left'</code>, <code>'left'</code>, <code>'top-left'</code>, <code>'center'</code>. An appropriate point of the paper content area is positioned to the corresponding point inside the paper scroller viewport. For example, if <code>positionName</code> was <code>'bottom-left'</code>, the bottom-left corner of paper content area is positioned to the bottom-left corner of the paper scroller viewport (and inside requested paper scroller paddings, if provided).

	<pre>paperScroller.positionContent('bottom-left'); paperScroller.centerContent('bottom-left', { padding: { horizontal: 20, vertical: 10 }});</pre>
positionElement(element, positionName [, opt])	Align <code>element</code> (an instance of <code>joint.dia.Element</code>) inside the paper scroller viewport as specified by <code>positionName</code> (detailed above). An appropriate point of the element <code>bbox</code> is positioned to the corresponding point inside the paper scroller viewport, similar to <code>positionContent</code> method. <pre>paperScroller.positionElement(element, 'right'); paperScroller.centerContent('right', { padding: { right: 50 }});</pre>
positionRect(rect, positionName [, opt])	Align a custom <code>rect</code> inside the paper scroller viewport as specified by <code>positionName</code> (detailed above). An appropriate point of the rectangle is positioned to the corresponding point inside the paper scroller viewport, similar to <code>positionContent</code> method. <pre>paperScroller.positionElement(element1.getBBox().union(element2.getBBox()), 'left-top'); paperScroller.centerContent(path.bbox(), 'right-top', { padding: { vertical: 10, right: 50 }});</pre>
positionPoint(point, x, y [, opt])	Position a custom <code>point</code> on the paper (in local coordinates) to the point <code>(x, y)</code> inside the paper scroller viewport (in client coordinates). This is useful when you need complete control over what to position where. Percentage values are allowed for <code>x</code> and <code>y</code>; the percentages are relative to the paperScroller area (only the area inside padding, if padding is provided). Negative values/percentages cause the algorithm to look from opposite edge (i.e. from right/bottom). The following example positions the origin of the paper to the origin (top-left point) of the paper scroller viewport: <pre>paperScroller.positionPoint(new g.Point(0,0), 0, 0); paperScroller.positionPoint(new g.Point(0,0), '-100%', '-100%'); // same result</pre> The following example positions <code>point</code> to the point 25% of the way between the right and the left edge of the paper scroller viewport and 40px above the

bottom edge:

```
paperScroller.positionPoint(point, '25%', -40);
```

The following example positions `point` to the point 30px away from right edge of the paper scroller viewport (10px + 20px padding) and 20px above bottom edge (10px + 10px padding):

```
paperScroller.positionPoint(point, 10, -10, { padding: { horizontal: 20, vertical: 10 } })
```

Paper visibility

`ui.PaperScroller` provides methods for checking whether elements or points are visible and not scrolled outside the viewport.

<code>getVisibleArea()</code>	Returns the size and origin of the current <code>paperScroller</code> viewport. The returned value is a <code>g.Rect</code> object, which contains <code>x</code>, <code>y</code>, <code>width</code> and <code>height</code> describing the visible area of the paper ignoring paper transformations (as would no scale, translate or rotate be applied). <pre>// return elements currently visible in the paperScroller viewport. var visibleElements = graph.findModelsInArea(paperScroller.getVisibleArea());</pre>
<code>isElementVisible(element [, opt])</code>	Returns <code>true</code> if the <code>element</code> is visible in the current <code>paperScroller</code> viewport. It returns <code>false</code> otherwise. Use <code>{ strict: [Boolean] }</code> option for toggling complete <code>true</code> vs. partial <code>false</code> visibility. It's non-strict by default. <pre>// Scroll to an element only if it is not completely in the viewport. if (!paperScroller.isElementVisible(rect, { strict: true }))) { paperScroller.scrollToElement(rect); }</pre>
<code>isPointVisible(point)</code>	Returns <code>true</code> if the <code>point</code> (e.g. <code>{ x: 100, y: 200 }</code>) is visible in the current <code>paperScroller</code> viewport. It returns <code>false</code> otherwise.

Zooming the paper

The PaperScroller exposes an API to scale the contained paper more easily:

```
paperScroller.zoom();
```

Return the current zoom level of the paper. Technically speaking, the “zoom level” is the scaling factor (`sx`) of the paper's `SVGElement`.

```
paperScroller.zoom(value [, opt]);
```

The `zoom()` method accepts a value by which we want to increase/decrease the scaling factor of the paper. A positive value increases the scaling factor, which causes the paper to be zoomed **in**. A negative value decreases the scaling factor, which causes the paper to be zoomed **out**.

You can pass two options to cap the range of possible scaling factors as part of an `opt` object - `min` and `max`. The scaling factor cannot then be decreased below `min` (if provided) or increased above `max` (if provided). These options are often useful in practice, with UI buttons bound to the paperScroller's zooming operations:

```
$('#zoom-in').on('click', function() {
  paperScroller.zoom(0.2, { max: 4 });
});

$('#zoom-out').on('click', function() {
  paperScroller.zoom(-0.2, { min: 0.2 });
});
```

Options of `ox` and `oy` are also available to transform the origin of the zoom. By default, the origin is the center of the viewport.

```
paper.on('blank:mousewheel', (evt, ox, oy, delta) => {
  evt.preventDefault();
  paperScroller.zoom(delta * 0.2, { min: 0.4, max: 3, grid: 0.2, ox, oy });
});
```

Additionally, passing a value for the `grid` option causes the resulting scaling factor to be rounded to the closest multiple of the `grid` value.

Passing `absolute: true` as an option sets the scaling factor directly to the provided `value` (instead of treating it as a relative value). The constraints imposed by the `min`, `max`, and `grid` are still in effect, however.

```
paperScroller.zoomToFit([opt]);
```

Scale the paper so that all of the contents of its graph fit into the user's viewport. For a list of available options, refer to `paper.scaleContentToFit()` [documentation](#).

```
paperScroller.zoomToRect(rect [, opt]);
```

Zoom and reposition the paper so that the area defined by `rect` (in the paper local coordinates) fits into the user's viewport. For a list of available options, refer to `paper.scaleContentToFit()` [documentation](#).

Animated transitions

The `paperScroller` provides few methods for panning and zooming in animated fashion. These are:

```
paperScroller.transitionToPoint(x, y [, opt])
paperScroller.transitionToPoint(point [, opt])
```

The method pans and optionally zooms the `paperScroller` to a given point. It accepts a point defined in the paper local coordinate system and an option object with several properties to control the transition.

scale	The zoom level to reach at the end of the transition. It defaults to the current <code>paperScroller</code> zoom level.
duration	A string representing the number of seconds or milliseconds a transition animation should take to complete. E.g. <code>'500ms'</code> , <code>'2s'</code> . It defaults to <code>'1s'</code>
delay	A string representing the amount of time to wait before a transition starts. It defaults to <code>'0s'</code>
timingFunction	The option to describe how the intermediate pan positions and zoom values are calculated. For all available values visit CSS transition-timing-function property documentation . It defaults to <code>'ease'</code> .
onTransitionEnd	A callback function fired with <code>transitionend</code> event as its first parameter when the transition ends.

```
// Move paper point (x=100, y=100) to the center of the viewport.
paperScroller.transitionToPoint(100, 100);
// Move and scale paper, so the point { x: 100, y: 100 } will appear in the center of
the viewport.
paperScroller.transitionToPoint(100, 100, { scale: 2 });
// Move the the center of the `element` to the center of the viewport.
paperScroller.transitionToPoint(element.getBBox().center(), {
  duration: '500ms',
  onTransitionEnd: function(transitionEvent) {
    // do something when the animation ends
  }
});
```

```
paperScroller.transitionToRect(rect [, opt]);
```

The method pans and zooms the paperScroller over a specific period so the given rectangular area is at the end of the transition entirely visible in the viewport. The rectangular area is an object with `x`, `y`, `width` and `height` properties and defined in the paper local coordinate system. The method accepts all options from `transitionToPoint()` (but `scale`) and the following options from the table below.

center	An object with <code>x</code> and <code>y</code> properties, which defines the point to be in the center of the viewport when the transition ends. By default the center of the rectangular area is used.
visibility	A number to change the percentage coverage of the viewport. The rectangular area covers 100% of the viewport by default (i.e. <code>{ visibility: 1 }</code>). For instance 80% coverage could be set with <code>{ visibility: .8 }</code>
minScale	A number to make the resulting scale always greater than or equal to <code>minScale</code> value.
maxScale	A number to make the resulting scale always less than or equal to <code>maxScale</code> value.
scaleGrid	A number to make the resulting scale the largest multiple of <code>scaleGrid</code> value less than or equal to the calculated scale.

```
// Zoom the paperScroller so the given area covers 90% of the viewport.
// Also do not let the paperScroller exceed the zoom level 3.
paperScroller.transitionToRect({
  x: 100,
  y: 100
  width: 200,
  height: 200
}, {
  visibility: 0.9,
  maxScale: 3,
  onTransitionEnd: function(transitionEvent) {
    // do something when the animation ends
  }
});
// Zoom the paperScroller in a way the elements `el1`, `el2`, `el3` cover the entire
viewport
// and the center of the `el1` moves to the center of the viewport.
var rect = graph.getCellsBBBox([el1, el2, el3]);
paperScroller.transitionToRect(rect, {
  duration: '500ms',
  center: el1.getBBox().center()
});

paperScroller.removeTransition();
```

When method called, the ongoing (if any) paperScroller transition is removed and the associated `onTransitionEnd` callback is not fired.

Paper auto-resize/shrink

The PaperScroller constructor takes an optional parameter `autoResizePaper`. When this parameter is set to `true`, the PaperScroller automatically resizes the paper so that it fits the content inside it. This makes it possible to have a fixed (even smaller) size of the paper and when the user drags an element outside this area, the PaperScroller extends this paper in order to accommodate for the new element. By default, the `ui.PaperScroller` resizes the paper by the `paper width / height`. If you want the paper to extend by a different amount, pass the `baseWidth / baseHeight` options to the `ui.PaperScroller` constructor function. The paper adjustment is internally handled by calculating proper parameters for the `joint.dia.Paper.fitToContent` method. You can further adjust the behavior by setting the `contentOptions` object to the `ui.PaperScroller` constructor function. This object can contain additional parameters that will be mixed with the calculated ones before calling the `joint.dia.Paper.fitToContent` method. This is handy if you, for example, want to set the maximum width and height of the paper. In this case, you can just set `contentOptions: { maxWidth: 3000, maxHeight: 3000 }` to the `ui.PaperScroller` constructor function. `contentOptions` can be also defined as a function returning the options.

```
new joint.ui.PaperScroller({
  padding: 0,
  contentOptions: function(paperScroller) {
    var visibleArea = paperScroller.getVisibleArea();
    return {
      padding: {
        bottom: visibleArea.height / 2,
        top: visibleArea.height / 2,
        left: visibleArea.width / 2,
        right: visibleArea.width / 2
      },
      allowNewOrigin: 'any'
    };
  }
});
```

Try how this works in the following demo by zooming the paper out so that you see its borders and move your element outside the paper area. You should see how the paper gets automatically resized. When you drag the element back to its original location, the PaperScroller resizes the paper back to its original size.

```
var paperScroller = new joint.ui.PaperScroller({ autoResizePaper: true });
```

Events

The plugin fires the following events.

Event Name	Handler Signature	Description

<i>pan:start</i>	(evt)	Triggered when the user starts panning.
<i>pan:stop</i>	(evt)	Triggered when the user stops panning.
<i>scroll</i>	(evt)	Triggered when the paper has been scrolled.

```
paperScroller.on('scroll', function(evt) {  
  console.log('The visible area has been changed:', this.getVisibleArea());  
});
```

JointJS+ PathDrawer

ui.PathDrawer

Path drawer allows users to draw `<path>` elements.

Installation

Include `joint.ui.pathDrawer.css` and `joint.ui.pathDrawer.js` files to your HTML.

```
<link rel="stylesheet" type="text/css" href="joint.ui.pathDrawer.css">
<script src="joint.ui.pathDrawer.js"></script>
```

Create a PathDrawer

First, we create a PathDrawer object:

```
new joint.ui.PathDrawer({
  target: document.getElementById('example-canvas'),
  pathAttributes: {
    'class': 'example-path'
  }
});
```

The constructor needs a `target`, a reference to an `<svg>` `SVGSVGElement` on the page, which will act as a drawing area. We can use one we created previously in the DOM, for example:

```
<svg id="example-canvas" width="800" height="800"></svg>
```

We can also use the `.svg` property of an existing Paper.

Configuration

The `joint.ui.PathDrawer` constructor accepts the following options:

target	<i>SVGSVGElement</i>	Required. The drawing area. Any paths drawn by the user are appended to this <code><svg></code> element. They are not removed when the PathDrawer is removed.
pathAttributes	<i>Object</i>	Optional. An object with CSS attributes of the new path. Default values are:

		<pre> { 'class': null, 'fill': '#ffffff', 'stroke': '#000000', 'stroke-width': 1, 'pointer-events': 'none' } </pre> If your <code>pathAttributes</code> object sets the <code>'class'</code> attribute (e.g. to <code>'example-path'</code>), you can access the path for further styling through the <code>#example-canvas .example-path</code> CSS selector.
startPointMarkup	<i>string</i>	Optional. The SVG markup of control point overlay elements. Default is <code>'<circle r="5"/>'</code> . (CSS styling should use the <code>.joint-path-drawer .start-point</code> CSS selector.)
snapRadius	<i>number</i>	Optional. If set to a number greater than zero, the endpoint of the current segment snaps to the x- and y-values of previously drawn segment endpoints. The number determines the distance for snapping.
enableCurves	<i>boolean</i>	Optional, <code>true</code> by default. If set to <code>false</code> , only straight paths can be drawn.

API

The `joint.ui.PathEditor` object provides the following methods:

render()	Renders the path drawer. Called automatically after object is instantiated.
remove()	Removes the path drawer.

For more fine-tuned programmatic control, you can emulate user interaction with delegated or synthetic mouse events. For example, to start drawing a path immediately after a `pathDrawer` is initialized:

```

paper.on('blank.pointerdown', function(event) {
  var vCanvas = V('svg', {
    width: 400,
    height: 400
  }).appendTo(this.paper.el);

  var pathDrawer = this.pathDrawer = new joint.ui.PathDrawer({
    el: vCanvas.node
  });
}

```



```

    pathDrawer.onPointerDown(event);
  }, this)

```

Interaction

Users can interact with `joint.ui.PathDrawer` objects by clicking/touching its canvas:

mousedown touchstart	.start-point	Calls <code>onStartPointPointerDown(event)</code> . Closes the path.
		Calls <code>onPointerDown(event)</code> . If this is the first click of a path, creates a new path with a new start point. Otherwise, finishes previous action and starts a new one. If the user does not release the mouse, triggers a bound <code>mousemove / touchmove</code> event and the option <code>enableCurves</code> is set to <code>true</code> , this function starts drawing a <code>curveto</code> control point until a <code>mouseup / touchend</code> event is registered. Otherwise, starts drawing a <code>lineto</code> path segment.
mousemove		Calls <code>onPointerMove(event)</code> . If the user is currently drawing a path segment, the path endpoint moves with user pointer.
dblclick		Calls <code>onDoubleClick(event)</code> . Stops path drawing at the location of the double click. The path is not finished with a <code>closepath</code> segment.
contextmenu		Calls <code>onContextMenu(event)</code> . Stops path drawing at the location of the previous anchor point. The path is not finished with a <code>closepath</code> segment.

Events

The `joint.ui.PathDrawer` object triggers various events that you can react on. Events can be handled by using the `on()` method.

clear	Triggered when the pathDrawer is cleared.		
path:create	Triggered when a new path is created. The handler is passed a reference to the <code>svgPath</code> <code>SVGPathElement</code> .		
path:finish	Triggered when a path is finished in a valid manner. The handler is passed a reference to the <code>svgPath</code> <code>SVGPathElement</code>. The following specific events are triggered alongside this generic event: <table> <tr> <td> path:close </td><td> Triggered when a path is finished by being reconnected with the start point. The handler is passed a reference to the <code>svgPath</code> <code>SVGPathElement</code>. </td></tr> </table>	path:close	Triggered when a path is finished by being reconnected with the start point. The handler is passed a reference to the <code>svgPath</code> <code>SVGPathElement</code> .
path:close	Triggered when a path is finished by being reconnected with the start point. The handler is passed a reference to the <code>svgPath</code> <code>SVGPathElement</code> .		

	<table> <tr> <td>path:stop</td><td>Triggered when a path is finished by being stopped without reconnecting with the start point (e.g. due to a double click or a right click). The handler is passed a reference to the <code>svgPath SVGPathElement</code>.</td></tr> </table>	path:stop	Triggered when a path is finished by being stopped without reconnecting with the start point (e.g. due to a double click or a right click). The handler is passed a reference to the <code>svgPath SVGPathElement</code> .				
path:stop	Triggered when a path is finished by being stopped without reconnecting with the start point (e.g. due to a double click or a right click). The handler is passed a reference to the <code>svgPath SVGPathElement</code> .						
path:abort	Triggered when a path is finished in an invalid manner (e.g. closed with only one anchor point). The handler is passed a reference to the <code>svgPath SVGPathElement</code> .						
path:segment:add	Triggered when the user adds a new segment to path segment list. The handler is passed a reference to the <code>svgPath SVGPathElement</code>. The following specific events are triggered alongside this generic event: <table> <tr> <td>path:move-segment:add</td><td>Triggered when the user adds a <code>moveto</code> segment to path segment list. The handler is passed a reference to the <code>svgPath SVGPathElement</code>.</td></tr> <tr> <td>path:line-segment:add</td><td>Triggered when the user adds a <code>lineto</code> segment to path segment list. The handler is passed a reference to the <code>svgPath SVGPathElement</code>.</td></tr> <tr> <td>path:curve-segment:add</td><td>Triggered when the user adds a <code>curveto</code> segment to path segment list. The handler is passed a reference to the <code>svgPath SVGPathElement</code>.</td></tr> </table>	path:move-segment:add	Triggered when the user adds a <code>moveto</code> segment to path segment list. The handler is passed a reference to the <code>svgPath SVGPathElement</code> .	path:line-segment:add	Triggered when the user adds a <code>lineto</code> segment to path segment list. The handler is passed a reference to the <code>svgPath SVGPathElement</code> .	path:curve-segment:add	Triggered when the user adds a <code>curveto</code> segment to path segment list. The handler is passed a reference to the <code>svgPath SVGPathElement</code> .
path:move-segment:add	Triggered when the user adds a <code>moveto</code> segment to path segment list. The handler is passed a reference to the <code>svgPath SVGPathElement</code> .						
path:line-segment:add	Triggered when the user adds a <code>lineto</code> segment to path segment list. The handler is passed a reference to the <code>svgPath SVGPathElement</code> .						
path:curve-segment:add	Triggered when the user adds a <code>curveto</code> segment to path segment list. The handler is passed a reference to the <code>svgPath SVGPathElement</code> .						
path:edit	Triggered when the user edits the path. The handler is passed a reference to the <code>svgPath SVGPathElement</code>. The following specific events are triggered alongside this generic event: <table> <tr> <td>path:last-segment:adjust</td><td>Triggered when the user adjusts last added segment. (This happens, for example, when the user specifies a control point after having added a <code>curveto</code> segment.) The handler is passed a reference to the <code>svgPath SVGPathElement</code>.</td></tr> <tr> <td>path:last-segment:replace-with-curve</td><td>Triggered when the user replaces last added line segment with a curved segment. (This happens, for example, when the user begins to specify a control point after having added a <code>lineto</code> segment.) The handler is passed a reference to the <code>svgPath SVGPathElement</code>.</td></tr> <tr> <td>path:last-segment:remove</td><td>Triggered when the user removes last added segment. (This happens, for example, when the user stops drawing with a right-click). The handler is passed a reference to the <code>svgPath SVGPathElement</code>.</td></tr> </table>	path:last-segment:adjust	Triggered when the user adjusts last added segment. (This happens, for example, when the user specifies a control point after having added a <code>curveto</code> segment.) The handler is passed a reference to the <code>svgPath SVGPathElement</code> .	path:last-segment:replace-with-curve	Triggered when the user replaces last added line segment with a curved segment. (This happens, for example, when the user begins to specify a control point after having added a <code>lineto</code> segment.) The handler is passed a reference to the <code>svgPath SVGPathElement</code> .	path:last-segment:remove	Triggered when the user removes last added segment. (This happens, for example, when the user stops drawing with a right-click). The handler is passed a reference to the <code>svgPath SVGPathElement</code> .
path:last-segment:adjust	Triggered when the user adjusts last added segment. (This happens, for example, when the user specifies a control point after having added a <code>curveto</code> segment.) The handler is passed a reference to the <code>svgPath SVGPathElement</code> .						
path:last-segment:replace-with-curve	Triggered when the user replaces last added line segment with a curved segment. (This happens, for example, when the user begins to specify a control point after having added a <code>lineto</code> segment.) The handler is passed a reference to the <code>svgPath SVGPathElement</code> .						
path:last-segment:remove	Triggered when the user removes last added segment. (This happens, for example, when the user stops drawing with a right-click). The handler is passed a reference to the <code>svgPath SVGPathElement</code> .						
path:interact	Triggered when the user interacts with drawer overlay elements. The handler is passed a reference to the <code>svgPath SVGPathElement</code> . The following specific events are triggered						

alongside this generic event:

path:control:adjust	Triggered when the user adjusts a control path. (This happens when the user specifies a control for a <code>curveTo</code> segment.) The handler is passed a reference to the <code>svgPath SVGPathElement</code> .
path:control:remove	Triggered when a control path is removed from the drawer interface. (This happens when the user is finished drawing a curve.) The handler is passed a reference to the <code>svgPath SVGPathElement</code> .

JointJS+ PathEditor

ui.PathEditor

Path editor allows users to edit `<path>` elements via an editing overlay. Users can interact with the path's segments, anchor points and curve control points.

Installation

Include `joint.ui.pathEditor.css` and `joint.ui.pathEditor.js` files to your HTML.

```
<link rel="stylesheet" type="text/css" href="joint.ui.pathEditor.css">
<script src="joint.ui.pathEditor.js"></script>
```

Create a PathEditor

First, we create a PathEditor object. The constructor needs a reference to a `<path>` SVGPathElement on the page.

```
new joint.dia.PathEditor({
  pathElement: document.querySelector('path')
});
```

Configuration

The `joint.ui.PathEditor` constructor accepts the following options:

pathElement	<i>SVGPathElement</i>	Required. The path element to be edited. The PathEditor will be rendered on top of it.
anchorPointMarkup	<i>string</i>	Optional. The SVG markup of anchor point overlay elements. Default is <code>'<circle r="2.5"/>'</code> . (CSS styling should use the <code>.joint-path-editor .anchor-point</code> CSS selector.)
controlPointMarkup	<i>string</i>	Optional. The SVG markup of control point overlay elements. Default is <code>'<circle r="2.5"/>'</code> . (CSS styling should use the <code>.joint-path-editor .control-point</code> CSS selector.)

API

The `joint.ui.PathEditor` object provides the following methods:

render()	Render the path editor overlay. Called automatically after object is instantiated.
remove()	Remove the path editor.
adjustAnchorPoint(index, dx, dy, evt [, opt])	Adjust the position of an anchor point. Identified by <code>index</code>, the index of the anchor point within path segment list (starting from <code>0</code> for the endpoint of the first moveto segment). The anchor point is translated by (<code>dx</code> , <code>dy</code>). The <code>evt</code> is the original user interaction event. The <code>opt</code> parameter is optional; it is used by internal methods. It can have one property: <code>dry</code> (indicating that corresponding <code>'path:editing'</code> and <code>'path:edit'</code> events will be suppressed).
adjustControlPoint(index, controlPointIndex, dx, dy, evt [, opt])	Adjust the position of a control point. Identified by <code>index</code>, the index of the path segment it is attached to within path segment list (starting from <code>1</code> for the first non-moveto segment), and <code>controlPointIndex</code>, the index of the control point within the segment (<code>1</code> for the “first” control point - the one visually attached to the previous anchor point, or <code>2</code> for the “second” control point - the one visually attached to this segment's anchor point). The control point is translated by (<code>dx</code> , <code>dy</code>). The <code>evt</code> is the original user interaction event. The <code>opt</code> parameter is optional; it is used by internal methods. It can have one property: <code>dry</code> (indicating that corresponding <code>'path:editing'</code> / <code>'path:edit'</code> events will be suppressed).
adjustSegment(index, dx, dy, evt)	Adjust the position of a segment. Identified by its <code>index</code> within path segment list (starting from <code>1</code> for the first non-moveto segment). The <code>evt</code> is the original user interaction event.
getControlPointLockedStates()	Return an array of arrays that records, for every control point (identified as <code>[index][controlPointIndex]</code>), whether it is currently locked with another point (<code>true</code>) or not (<code>false</code>). In tandem with <code>setControlPointLockedStates(lockedStates)</code> , the function can be used to preserve locking information across multiple path editor sessions for a given element.
setControlPointLockedStates(lockedStates)	Change the <code>Locked</code> status of the editor's control points, so they correspond to the information in the given <code>lockedStates</code> list (e.g. the output of the <code>getControlPointLockedStates()</code>

function). A **true** value adds the 'Locked' class to the control point element; a **false** value removes the class.

Interaction API

The following methods are meant to be called in response to user interaction events (e.g. click, touch, dblclick). Some of them are bound with user interactions by default (see the [Interaction](#) section), but others are not. Those may be called upon by delegated events from your application. For example:

```
pathEditor.delegate('contextmenu', '.segment-path', function(event) {  
    event.stopPropagation();  
    event.preventDefault();  
  
    this.convertSegmentPath(event);  
}).bind(pathEditor);
```

startMoving(event)	Bound with 'mousedown' / 'touchstart' events by default. The selected anchor point / control point / segment path starts moving with user pointer. Trigger a corresponding 'path:interact' event. To start listening for the <code>move()</code> and <code>stopMoving()</code> methods, call <code>delegateDocumentEvents()</code>.
move(event)	Bound with 'mousemove' / 'touchmove' events by default (after <code>delegateDocumentEvents()</code> is called). The selected anchor point / control point / segment path moves with user pointer. Trigger corresponding 'path:editing' events.
stopMoving(event)	Bound with 'mouseup' / 'touchend' / 'touchcancel' events by default (after <code>delegateDocumentEvents()</code> is called). The selected anchor point / control point / segment path stops moving with user pointer. Trigger a corresponding 'path:edit' event. To stop listening for the <code>move()</code> and <code>stopMoving()</code> methods, call <code>undelegateDocumentEvents()</code>.
createAnchorPoint(event)	Bound with 'dblclick .segment-path' event by default. If <code>event</code> is targeted on a segment path, create an anchor point at the location of the user click and insert it into the path segment list. Trigger the 'path:anchor-point:create' event. Note: Due to current limitations, the function does not work properly for

	closepath segments in paths that have 2 or more closepath segments (the anchor points are created at wrong position).
removeAnchorPoint(event)	Bound with <code>'dblclick .anchor-point'</code> event by default. If <code>event</code> is targeted on an anchor point, remove it. Trigger the <code>'path:anchor-point:remove'</code> event. If only one anchor point remains in the path after the removal, also trigger the <code>'path:invalid'</code> event.
lockControlPoint(event)	Bound with <code>'dblclick .control-point'</code> event by default. If <code>event</code> is targeted on a control point of a <code>curveto</code> segment, and the control point is immediately adjacent to a control point of another <code>curveto</code> segment, toggle its <code>locked</code> status. Trigger the corresponding <code>'path:control-point:lock'</code> / <code>'path:control-point:unlock'</code> event. Note that the locking action necessarily causes a change in the path markup which does not trigger an <code>'path:edit'</code> event!
addClosePathSegment(event)	Not bound with any user interaction by default. If <code>event</code> is targeted on the first or last anchor point, and the path segment list contains no <code>closepath</code> segment, append a <code>closepath</code> segment to the path segment list.
removeClosePathSegment(event)	Not bound with any user interaction by default. If <code>event</code> is targeted on a <code>closepath</code> segment, remove the segment from the path segment list.
convertSegmentPath(event)	Not bound with any user interaction by default. If <code>event</code> is targeted on a path segment, convert the segment from linear to curved or vice versa. A <code>curveto</code> segment is replaced by a <code>lineto</code> segment. A <code>lineto</code> / <code>closepath</code> segment is replaced by a <code>curveto</code> segment whose control points are coincident with anchor points; a replacement <code>closepath</code> segment is appended to the path segment list if the converted segment was a <code>closepath</code> segment. Note: Due to current limitations, the function does not work properly for closepath segments in paths that have 2 or more closepath segments (the anchor points are created at wrong position).

Interaction

Users can interact with `joint.ui.PathEditor` objects by clicking/touching editor overlay elements:

mousedown touchstart	.anchor-point	Call <code>onAnchorPointPointerDown(event)</code> , which calls <code>startMoving()</code> followed by <code>delegateDocumentEvents()</code> by default.
	.control-point	Calls <code>onControlPointPointerDown(event)</code> , which calls <code>startMoving()</code> followed by <code>delegateDocumentEvents()</code> by default.
	.segment-path	Calls <code>onSegmentPathPointerDown(event)</code> , which calls <code>startMoving()</code> followed by <code>delegateDocumentEvents()</code> by default.

Double-clicking editor overlay elements triggers additional actions:

dblclick	.anchor-point	Calls <code>onAnchorPointDoubleClick(event)</code> , which calls <code>removeAnchorPoint()</code> by default.
	.control-point	Calls <code>onControlPointDoubleClick(event)</code> , which calls <code>lockControlPoint()</code> by default.
	.segment-path	Calls <code>onSegmentPathDoubleClick(event)</code> , which calls <code>createAnchorPoint()</code> by default.

For the documentation of the called functions, see the [Interaction API](#) section.

Events

The `joint.ui.PathEditor` object triggers various events that you can react on:

clear	Triggered when the pathEditor is cleared.	
path:interact	Triggered when the user interacts with (clicks/touches) the path. The following specific events are triggered alongside this generic event:	
	<table> <tr> <td>path:anchor-point:select</td><td> Triggered when the user interacts with an anchor point. The handler is passed the <code>index</code> of the segment that this anchor point is bound to in the path segment list. The first anchor point on the path has an index of 0. </td></tr> </table>	path:anchor-point:select
path:anchor-point:select	Triggered when the user interacts with an anchor point. The handler is passed the <code>index</code> of the segment that this anchor point is bound to in the path segment list. The first anchor point on the path has an index of 0.	

	path:control-point:select	Triggered when the user interacts with a control point. The handler is passed two values. The first value is the <code>index</code> of the segment that this control point is bound to in the path segment list. (Since the first <code>moveto</code> segment cannot have control points, the minimal index is 1.) The second value is the <code>controlPointIndex</code> of the control point. It can be either <code>1</code> or <code>2</code>.		
	path:segment:select	Triggered when the user interacts with a path segment. The handler is passed the <code>index</code> of the segment in the path segment list. (Since the first <code>moveto</code> segment is not clickable, the minimal index is 1.)		
	path:control-point:lock	Triggered when the user successfully locks a control segment (binds it with a control point in an adjacent <code>curveto</code> path). The handler is passed two values. The first value is the <code>index</code> of the segment that this control point is bound to in the path segment list. (Since the first <code>moveto</code> segment cannot have control points, the minimal index is 1.) The second value is the <code>controlPointIndex</code> of the control point. It can be either <code>1</code> or <code>2</code>.		
	path:control-point:unlock	Triggered when the user unlocks a control segment (stops its binding with a control point in an adjacent <code>curveto</code> path). The handler is passed two values. The first value is the <code>index</code> of the segment that this control point is bound to in the path segment list. (Since the first <code>moveto</code> segment cannot have control points, the minimal index is 1.) The second value is the <code>controlPointIndex</code> of the control point. It can be either <code>1</code> or <code>2</code>.		
path:editing	Triggered while the user is editing the path. The handler is passed a reference to the updated <code>svgPath SVGPathElement</code>, and <code>evt</code> (the user interaction event). The following specific events are triggered alongside this generic event: <table><tr><td>path:anchor-point:adjusting</td><td> Triggered while the user is adjusting the position of an anchor point. The handler is passed a reference to the updated <code>svgPath SVGPathElement</code>, <code>evt</code> (the user interaction event), and an </td></tr></table>		path:anchor-point:adjusting	Triggered while the user is adjusting the position of an anchor point. The handler is passed a reference to the updated <code>svgPath SVGPathElement</code>, <code>evt</code> (the user interaction event), and an
path:anchor-point:adjusting	Triggered while the user is adjusting the position of an anchor point. The handler is passed a reference to the updated <code>svgPath SVGPathElement</code>, <code>evt</code> (the user interaction event), and an			

		<code>opt</code> object with two properties: <code>index</code> (the index of the adjusted anchor point), and <code>segPoint</code> (the new position of the anchor point).						
	path:control-point:adjusting	Triggered while the user is adjusting the position of a control point. The handler is passed a reference to the updated <code>svgPath SVGPathElement</code>, <code>evt</code> (the user interaction event), and an <code>opt</code> object with three properties: <code>index</code> (the index of the associated anchor point), <code>controlPointIndex</code> (the index of the adjusted control point), and <code>segPoint</code> (the new position of the anchor point).						
	path:segment:adjusting	Triggered while the user is adjusting the position of a segment. The handler is passed a reference to the updated <code>svgPath SVGPathElement</code>, <code>evt</code> (the user interaction event), and an <code>opt</code> object with one property: <code>index</code> (the index of the adjusted segment).						
path:edit	Triggered when the user edits the path. The handler is passed a reference to the updated <code>svgPath SVGPathElement</code>, and <code>evt</code> (the user interaction event). The following specific events are triggered alongside this generic event: <table><tr><td>path:anchor-point:adjust</td><td> Triggered when the user adjusts the position of an anchor point. The handler is passed a reference to the updated <code>svgPath SVGPathElement</code>, <code>evt</code> (the user interaction event), and an <code>opt</code> object with two properties: <code>index</code> (the index of the adjusted anchor point), and <code>segPoint</code> (the new position of the anchor point). </td></tr><tr><td>path:control-point:adjust</td><td> Triggered when the user adjusts the position of a control point. The handler is passed a reference to the updated <code>svgPath SVGPathElement</code>, <code>evt</code> (the user interaction event), and an <code>opt</code> object with three properties: <code>index</code> (the index of the associated anchor point), <code>controlPointIndex</code> (the index of the adjusted control point), and <code>segPoint</code> (the new position of the anchor point). </td></tr><tr><td>path:segment:adjust</td><td> Triggered when the user adjusts the position of a segment. </td></tr></table>		path:anchor-point:adjust	Triggered when the user adjusts the position of an anchor point. The handler is passed a reference to the updated <code>svgPath SVGPathElement</code>, <code>evt</code> (the user interaction event), and an <code>opt</code> object with two properties: <code>index</code> (the index of the adjusted anchor point), and <code>segPoint</code> (the new position of the anchor point).	path:control-point:adjust	Triggered when the user adjusts the position of a control point. The handler is passed a reference to the updated <code>svgPath SVGPathElement</code>, <code>evt</code> (the user interaction event), and an <code>opt</code> object with three properties: <code>index</code> (the index of the associated anchor point), <code>controlPointIndex</code> (the index of the adjusted control point), and <code>segPoint</code> (the new position of the anchor point).	path:segment:adjust	Triggered when the user adjusts the position of a segment.
path:anchor-point:adjust	Triggered when the user adjusts the position of an anchor point. The handler is passed a reference to the updated <code>svgPath SVGPathElement</code>, <code>evt</code> (the user interaction event), and an <code>opt</code> object with two properties: <code>index</code> (the index of the adjusted anchor point), and <code>segPoint</code> (the new position of the anchor point).							
path:control-point:adjust	Triggered when the user adjusts the position of a control point. The handler is passed a reference to the updated <code>svgPath SVGPathElement</code>, <code>evt</code> (the user interaction event), and an <code>opt</code> object with three properties: <code>index</code> (the index of the associated anchor point), <code>controlPointIndex</code> (the index of the adjusted control point), and <code>segPoint</code> (the new position of the anchor point).							
path:segment:adjust	Triggered when the user adjusts the position of a segment.							

		The handler is passed a reference to the updated <code>svgPath SVGPathElement</code> , <code>evt</code> (the user interaction event), and an <code>opt</code> object with one property: <code>index</code> (the index of the adjusted segment).
	path:anchor-point:create	Triggered when the user adds a new anchor point to the path. The handler is passed a reference to the updated <code>svgPath SVGPathElement</code>, and <code>evt</code> (the user interaction event).
	path:anchor-point:remove	Triggered when the user removes an anchor point from the path. The handler is passed a reference to the updated <code>svgPath SVGPathElement</code>, and <code>evt</code> (the user interaction event).
	path:segment:convert	Triggered when the user converts a <code>lineto</code> path segment into a <code>curveto</code> segment or vice versa. The handler is passed a reference to the updated <code>svgPath SVGPathElement</code>, and <code>evt</code> (the user interaction event).
	path:closepath-segment:add	Triggered when the user adds a <code>closepath</code> segment to the path. The handler is passed a reference to the updated <code>svgPath SVGPathElement</code>, and <code>evt</code> (the user interaction event).
	path:closepath-segment:remove	Triggered when the user removes a <code>closepath</code> segment from the path. The handler is passed a reference to the updated <code>svgPath SVGPathElement</code>, and <code>evt</code> (the user interaction event).
path:invalid	Triggered when user modifications result in an invalid path (e.g. after removing, a path that has only one anchor point). The handler is passed a reference to the <code>svgPath SVGPathElement</code>, and <code>evt</code> (the user interaction event).	

You can react to these events by using the `on()` method:

```
pathEditor.on({
  'path:control-point:select': console.log('started moving control point'),
  'path:control-point:adjusting': console.log('moving control point'),
```

```
    'path:control-point:adjust': console.log('stopped moving control point')
  });
```

It is also possible to specify a custom context for the `on()` method. This is useful when you need to handle user interaction on your PathEditor from within another Backbone object, for instance:

```
pathEditor.on({
  'path:edit': this.onPathEdit.bind(this),
}, this);
```

JointJS+ Popup

ui.Popup

`ui.Popup` is a UI widget for displaying contextual information (or even other diagrams) below a certain target element (HTML or SVG). There can always be only one popup opened at a time. This is automatically handled by the `ui.Popup` component which simplifies the process and makes sure you don't have to keep track of opened popups yourself. This component is very similar to the `ui.ContextToolbar` component (in fact, it inherits from it) with the only difference that `ui.Popup` does not contain a set of buttons but an arbitrary HTML (which might even contain another diagram).

Installation

Include `joint.ui.popup.js` and `joint.ui.popup.css` files and the `joint.ui.contextToolbar.js` and `joint.ui.contextToolbar.css` dependencies to your HTML:

```
<link rel="stylesheet" type="text/css" href="joint.ui.contextToolbar.css">
<link rel="stylesheet" type="text/css" href="joint.ui.popup.css">
<script src="joint.ui.contextToolbar.js"></script>
<script src="joint.ui.popup.js"></script>
```

Usage

Create an object of `joint.ui.Popup` type, specify content and call the `render()` method in order to open the popup.

```
const element = new joint.shapes.standard.Rectangle({
  position: { x: 10, y: 10 },
  size: { width: 80, height: 50 },
  attrs: { label: { text: 'Rect' } }
});

const elementView = element.findView(paper);

new joint.ui.Popup({
  content: '<b>I am a ui.Popup</b>',
  target: elementView.el
}).render();
```

A common usage is to open the popup on click on a certain element (either HTML or SVG) and display either an image or another diagram:

```
paper.on('element:pointerdown', (elementView) => {
  const popup = new joint.ui.Popup({
```

```

        content: '',
        target: elementView.el
    });
    popup.render();
});

paper.on('element:pointerdown', (elementView, evt) => {
    evt.preventDefault();
    const popup = new joint.ui.Popup({
        content: 'I am a ui.Popup!',
        target: elementView.el,
        anchor: 'bottom-left',
        position: 'right',
        arrowPosition: 'none'
    });
    popup.render();
});

paper.on('element:pointerdown', (elementView) => {
    const popup = new joint.ui.Popup({
        content: function(el) {
            const graph = new joint.dia.Graph;
            const paper = new joint.dia.Paper({
                width: 200,
                height: 100,
                gridSize: 1,
                model: graph
            });
            el.appendChild(paper.el);
            const r1 = new joint.shapes.standard.Rectangle({ position: { x: 10, y: 10
}, size: { width: 50, height: 30 }, attrs: { body: { fill: '#FE854F' }, label: { text:
'r1' }}}});
            const r2 = new joint.shapes.standard.Rectangle({ position: { x: 90, y: 40
}, size: { width: 50, height: 30 }, attrs: { body: { fill: '#7C68FC' }, label: { text:
'r2' }}}});
            const l = new joint.shapes.standard.Link({ source: { id: r1.id }, target:
{ id: r2.id }});
            graph.addCell([r1, r2, l]);
        },
        target: elementView.el
    }).render();
});

paper.on('element:pointerdown', (elementView) => {
    const popup = new joint.ui.Popup({
        events: {

```

```

        'click .btn-cancel': 'remove',
        'click .btn-change': function() {
            const strokeWidth = parseInt(document.querySelector('.inp-stroke-
width').value, 10);
            const fill = document.querySelector('.inp-fill').value;
            elementView.model.attr({
                body: { fill: fill, strokeWidth: strokeWidth }
            });
        }
    },
    content: [
        '<div>',
        'Fill: <input class="inp-fill" type="color" value="#FEB663"> <br><br>',
        'Stroke width: <input class="inp-stroke-width" type="number" value="5">
<br><br>',
        '<button class="btn-cancel">Cancel</button>',
        '<button class="btn-change">Change</button>',
        '</div>'
    ].join(''),
    target: elementView.el
});

popup.render();
});

```

Appearance

You can alter arrow appearance by customizing css variables inside the `.joint-popup` class. Available variables are `--arrow-width`, `--arrow-mask-width`, `--arrow-color`, `--arrow-mask-color`.

Configuration

The following table lists options that you can pass to the `ui.Popup` constructor function:

content	The content of the popup. It can be a text or an arbitrary HTML as a string, or an HTML element. It can also be specified as a function with <code>el</code> (the parent HTML element of the Popup) exposed as a parameter. The parameter can be used, for example, to dynamically calculate a return string or HTML element to be used as the content of the popup, as specified above. Alternatively, instead of returning a value, the function may append HTML elements directly to <code>el</code> (e.g. by calling <code>el.appendChild(paper.el)</code>; using a JointJS <code>paper</code> element). This approach enables appending of JointJS diagrams directly into the <code>ui.Popup</code> element.
target	The target element (either HTML or SVG).

padding	The gap between the target element and the popup. Default is <code>20</code> .
position	The name of the position on the target element to which the popup is attached. The default is <code>'bottom'</code> .
anchor	The name of the position on the popup, where it is attached to the target. The default is <code>'top'</code> . The arrow is rendered with this position by default. If the anchor is <code>'center'</code> the arrow is not rendered.
arrowPosition	The optional name of the position on the popup, where the arrow is rendered. The default is anchor position. Use <code>'none'</code> if you want the arrow not to be rendered. Accepts all position names except <code>'center'</code> .
autoClose	Determines if the popup should be closed if the user clicked outside of the area of the popup. Default is <code>true</code> .
scale	Optional parameter that sets the scale of the context toolbar.

API

render()	Render the popup onto the screen.
remove()	Remove the popup. Usually, you don't have to call this method as the popup is closed automatically if the user clicked outside of the area of the popup (unless you set the <code>autoClose</code> option to <code>false</code>).
on(event, handler [, context])	Register a <code>handler</code> function that will be called whenever <code>event</code> is triggered. The optional <code>context</code> is an object that will be the context of the handler function (the <code>this</code>).
joint.ui.Popup.close()	This is a static method that closes any popup that is currently open.
joint.ui.Popup.update()	This is a static method that updates the position of the opened popup (if any). Call this method if your <code>target</code> element changes its position while you still want to keep the popup opened.

Events

close	Triggered when the popup gets closed.

JointJS+ RadioGroup

ui.RadioGroup

`ui.RadioGroup` is a UI widget for displaying radio groups.

Installation

Include `joint.ui.radioGroup.js` and `joint.ui.radioGroup.css` files to your HTML:

```
<link rel="stylesheet" type="text/css" href="joint.ui.radioGroup.css">
<script src="joint.ui.radioGroup.js"></script>
```

Usage

Create an object of `ui.RadioGroup` type, render it with the `render()` method and append the generated HTML element anywhere in your DOM:

```
const radioGroup = new joint.ui.RadioGroup({
  name: 'myRadioGroup',
  options: [{
    content: 'Value 1',
    value: 1
  }, {
    content: 'Value 2',
    value: 2
  }, {
    content: 'Value 2',
    value: 2
  }]
});
document.body.appendChild(radioGroup.render().el);
```

Configuration

The following table lists options that you can pass to the `ui.RadioGroup` constructor function.

options	An array of options. Each option is an object with the following properties:	
	content	The content of the radio button label. Can be a simple string or a string with a custom html.
	value	The value of the option. It can be a string, a number or any other JavaScript object.
name	Name of the inner input element. By default it is set to the component's <code>cid</code> .	

API

render()	Render the radio group. Note that once you render the color palette, you can use the <code>el</code> property that points to the container HTML element and append it anywhere in the DOM (e.g. <code>\$(document.body).append(radioGroup.el)</code>).
selectByValue(value)	Select an item in the radio group by its value.
select(index)	Select an item in the radio group by its index in the options array.
getCurrentValue()	Returns the current value of the radio group.
getSelectionIndex()	Returns the index value of the current value.
setOptions(options)	Sets new set of options to be rendered inside the radio group component.
remove()	Remove the radio group from the DOM.
on(event, handler [, context])	Register a <code>handler</code> function that will be called whenever <code>event</code> is triggered. The optional <code>context</code> is an object that will be the context of the handler function (the <code>this</code>).

Events

The `ui.RadioGroup` object triggers events that you can react on. Events can be handled by using the `on()` method (see above).

options:select	Triggered when the value of the radio group is selected. The handler passed the current value of the radio group as the first parameter. The second parameter is the RadioGroup object.

JointJS+ SelectBox

ui.SelectBox

`ui.SelectBox` is a UI widget for displaying dropdowns. Items in the dropdown can contain arbitrary HTML. Because items in dropdowns often contain images, `ui.SelectBox` provides an easy way to add icons to your dropdown items. The `ui.SelectBox` also provides different open policies and support for keyboard navigation.

Installation

Include `joint.ui.selectBox.js` and `joint.ui.selectBox.css` files to your HTML:

```
<link rel="stylesheet" type="text/css" href="joint.ui.selectBox.css">
<script src="joint.ui.selectBox.js"></script>
```

Usage

Create an object of `ui.SelectBox` type, render it with the `render()` method and append the generated HTML element anywhere in your DOM:

```
var selectBox = new joint.ui.SelectBox({
  width: 200,
  options: [
    { content: 'Amsterdam' },
    { content: 'Prague', selected: true },
    { content: 'London' },
    { content: 'San Francisco' },
    { content: 'New York' }
  ]
});

document.body.appendChild(selectBox.render().el);
```

ui.SelectBox with icons

```
var selectBox = new joint.ui.SelectBox({
  width: 250,
  options: [
    { icon: '/path/to/some-image.png', content: 'Some text content',
selected: true },
    { icon: '/path/to/some-other-image.png', content: 'Some other text' }
  ]
});
```

```
document.body.appendChild(selectBox.render().el);
```

Configuration

The following table lists options that you can pass to the `ui.SelectBox` constructor function:

width	The width of the dropdown in pixels.								
options	An array of items. Each item is an object with the following properties: <table><tr><td>content</td><td>The content of the item. It can be either a string or an HTML.</td></tr><tr><td>value</td><td>The value of the item. It can be a string, a number or any other JavaScript object. If the value is not defined, the value is considered to be equal to the <code>content</code> property.</td></tr><tr><td>icon</td><td>A path to an image. The icon is prepended to the item and is given the <code>select-box-option-icon</code> CSS class.</td></tr><tr><td>selected</td><td><code>true</code> if the item should be selected by default. If none of the items are selected, the first item is selected by default.</td></tr></table>	content	The content of the item. It can be either a string or an HTML.	value	The value of the item. It can be a string, a number or any other JavaScript object. If the value is not defined, the value is considered to be equal to the <code>content</code> property.	icon	A path to an image. The icon is prepended to the item and is given the <code>select-box-option-icon</code> CSS class.	selected	<code>true</code> if the item should be selected by default. If none of the items are selected, the first item is selected by default.
content	The content of the item. It can be either a string or an HTML.								
value	The value of the item. It can be a string, a number or any other JavaScript object. If the value is not defined, the value is considered to be equal to the <code>content</code> property.								
icon	A path to an image. The icon is prepended to the item and is given the <code>select-box-option-icon</code> CSS class.								
selected	<code>true</code> if the item should be selected by default. If none of the items are selected, the first item is selected by default.								
selected	The index of the item which should be selected by default. If this value is undefined (which it is by default), the <code>ui.SelectBox</code> widget looks up the selected item from the <code>options</code> array (by using the <code>selected</code> boolean property). If nothing is selected, the first item in the <code>options</code> array is selected by default.								
keyboardNavigation	<code>true</code> (the default) if you want the <code>ui.SelectBox</code> to provide a keyboard navigation (up/down/left/right/enter/escape keys are supported).								
selectBoxOptionsClass	When the dropdown is opened, the <code>ui.SelectBox</code> generates an HTML container that contains all the items. This container is then appended to the document body (or <code>target</code> if provided). Sometimes, you want to provide custom styling to this dropdown container. <code>selectBoxOptionsClass</code> gives you a chance to define a CSS class that will be set on this container so that you can style it in your CSS.								
target	<code>ui.SelectBox</code> generates an HTML container that contains all the dropdown items. This container is by default appended to the document body. In some situations (e.g. if you want the dropdown to scroll with some other container), you want to append this container to a different DOM element. The <code>target</code> option is exactly for that. It accepts an HTML element that will be used as a container for the generated dropdown items.								
openPolicy	<code>openPolicy</code> determines how the dropdown will open when clicked. It can be one of <code>'auto'</code> , <code>'above'</code> , <code>'coverAbove'</code> , <code>'below'</code> , <code>'coverBelow'</code> and								

	<code>'selected' .</code>
placeholder	In some cases, you want to “deselect” the dropdown and show a placeholder in place of the selected item. This is when you don't want any of the items to be selected. In this case, set the <code>selected</code> index to <code>-1</code> and the <code>placeholder</code> to any HTML you want to display in the selected item window. This placeholder has the <code>selected-box-placeholder</code> CSS class that you can use for further styling.

API

render()	Render the dropdown. Note that once you render the dropdown, you can use the <code>el</code> property that points to the container HTML element and append it anywhere in the DOM (e.g. <code>\$(document.body).append(selectBox.el)</code>).
getSelection()	Get the current selection. The selection points to one of the items from the <code>options</code> array.
getSelectionValue()	Get the current selection value.
getSelectionIndex()	Get the index in the <code>options</code> array of the item that is currently selected.
select(index, opt)	Select an item. <code>index</code> is the index to the <code>options</code> array. <code>opt</code> is optional and can be an arbitrary object that will be passed to the <code>option:select</code> event handler.
selectByValue(value)	Select an item by value. The selected item will be chosen based on a strict equality of the <code>value</code> passed an a <code>value (content)</code> of any of the items.
isOpen()	Returns <code>true</code> if the dropdown is currently opened. <code>false</code> otherwise.
toggle()	Programmatically toggle the dropdown.
open()	Programmatically open the dropdown.
close()	Programmatically close the dropdown.
remove()	Remove the dropdown from the DOM.
disable()	Disable the dropdown.
enable()	Enable the dropdown.
isDisabled()	Return <code>true</code> if the dropdown is disabled.

on(event, handler [, context])	Register a <code>handler</code> function that will be called whenever <code>event</code> is triggered. The optional <code>context</code> is an object that will be the context of the handler function (the <code>this</code>).

Events

The `ui.SelectBox` object triggers various events that you can react on. Events can be handled by using the `on()` method (see above).

option: hover	Triggered when the user hovers over one of the items. The handler is passed the option object and the index of the option in the <code>options</code> array.
option: select	Triggered when the user selects an item. The handler is passed the option object and the index of the option in the <code>options</code> array. If you selected the item with the <code>select(index, opt)</code> method, the third argument to the event handler will be your <code>opt</code> object.
option: mouseout	Triggered when the user leaves an item with mouse cursor. The handler is passed an event object as the only argument.
close	Triggered when the dropdown gets closed.

JointJS+ SelectButtonGroup

ui.SelectButtonGroup

`ui.SelectButtonGroup` is a UI widget for displaying groups of buttons with both single and multi selection support. The buttons can contain arbitrary HTML.

Installation

Include `joint.ui.selectButtonGroup.js` and `joint.ui.selectButtonGroup.css` files to your HTML:

```
<link rel="stylesheet" type="text/css" href="joint.ui.selectButtonGroup.css">
<script src="joint.ui.selectButtonGroup.js"></script>
```

Usage

Create an object of `ui.SelectButtonGroup` type, render it with the `render()` method and append the generated HTML element anywhere in your DOM:

```
var buttonGroup = new joint.ui.SelectButtonGroup({
  multi: true,
  selected: [1, 3],
  options: [
    { value: 'line-through', content: 'S' },
    { value: 'underline', content: 'U' },
    { value: 'italic', content: 'I' },
    { value: 'bold', content: 'B' }
  ]
});
document.body.appendChild(buttonGroup.render().el);
```

More examples

`ui.SelectButtonGroup` **with icons**

Configuration

The following table lists options that you can pass to the `ui.SelectButtonGroup` constructor function:

options	<i>Object[]</i>	Optional. An array of items (the buttons). Each item is an object with the following properties:		
		content	string	Optional. The content of the item. It can be either a plain string or a string containing

				HTML code.
		value	<i>any</i>	Optional. The value of the item. It can be a string, a number or any other JavaScript object. By default, the value is considered to be equal to the <code>content</code> property.
		attrs	<i>Object</i>	Optional. A hierarchical object that adds styling to component HTML elements of the option. For example, the following <code>attrs</code> object would add a tooltip to the option: <pre>{ '.select-button-group-button': { 'data-tooltip': 'My tooltip' } }</pre>
		selected	<i>boolean</i>	Optional. Should this item be selected by default? Default is <code>false</code> .
		icon	<i>string</i>	Optional. A path to an image to be added to the option's button.
		iconSelected	<i>string</i>	Optional. A path to an image to be added to the option's button when it is selected. By default, the selected state uses <code>icon</code> (if provided).
		buttonWidth	<i>number</i>	Optional. The width of the button in pixels.
		buttonHeight	<i>number</i>	Optional. The height of the button in pixels.
		iconWidth	<i>number</i>	Optional. The width of <code>icon</code> in pixels (if provided).
		iconHeight	<i>number</i>	Optional. The height of <code>icon</code> in pixels (if provided).

disabled	<i>boolean</i>	Optional. Is the user prevented from interacting with this <code>ui.SelectButtonGroup</code> ? Default is <code>false</code> .
multi	<i>boolean</i>	Optional. Is this a multiple-choice set? (That is, is it allowed for multiple buttons to be selected at the same time?) Default is <code>false</code> .
selected	<i>number number[]</i>	Optional. Which options should be selected by default? If <code>multi === false</code> , expects a number - the index of the selected option. If <code>multi === true</code> , expects an array of numbers - indices of all selected options. By default, this property is <code>undefined</code> . In that case, the widget determines selected items according to individual options' <code>selected</code> property.
singleDeselect	<i>boolean</i>	Optional. If <code>multi === false</code> , is the user allowed to deselect the currently-selected icon by clicking on it? Default is <code>false</code> (radio-button-like behavior).
noSelectionValue	<i>any</i>	Optional. If nothing is selected, what value should be reported by the <code>getSelectionValue()</code> function? Default is <code>undefined</code> .
width	<i>number</i>	Optional. The width of the whole widget in pixels.
buttonWidth	<i>number</i>	Optional. The width of each button in pixels. Overridden by individual options' <code>buttonWidth</code> property (if provided).
buttonHeight	<i>number</i>	Optional. The height of each button in pixels. Overridden by individual options' <code>buttonHeight</code> property (if provided).
iconWidth	<i>number</i>	Optional. The width of each icon in pixels (for options with an <code>icon</code> property). Overridden by individual options' <code>iconWidth</code> property (if provided).
iconHeight	<i>number</i>	Optional. The height of each icon in pixels (for options with an <code>icon</code> property). Overridden by individual options' <code>iconHeight</code> property (if provided).

API

render()	Render the button group. Note that once you render the button group, you can use the <code>el</code> property that points to the container HTML element and append it anywhere in the DOM (e.g. <code>\$(document.body).append(selectButtonGroup.el)</code>).
getSelection()	Get the current selection. The selection points to an item(s) from the <code>options</code> array.

getSelectionValue()	Get the current selection value(s). If nothing is selected, <code>noSelectionValue</code> is returned (default is <code>undefined</code>).
select(index, opt)	Select an item. <code>index</code> is the index to the <code>options</code> array. <code>opt</code> is optional and can be an arbitrary object that will be passed to the <code>option:select</code> event handler.
selectByValue(value)	Select an item by value. The selected item will be chosen based on a strict equality of the <code>value</code> passed an a <code>value (content)</code> of any of the items.
deselect()	Deselect all selected items.
remove()	Remove the button group from the DOM.
on(event, handler [, context])	Register a <code>handler</code> function that will be called whenever <code>event</code> is triggered. The optional <code>context</code> is an object that will be the context of the handler function (the <code>this</code>).
disable()	Prevent to the user's interactions.
enable()	Make the button group available to the user's interactions.
isDisabled()	Check if the button group is prevented to the user's interactions.

Events

The `ui.SelectButtonGroup` object triggers various events that you can react on. Events can be handled by using the `on()` method (see above).

option:hover	Triggered when the user hovers over one of the items. The handler is passed the option object and the index of the option in the <code>options</code> array.
option:select	Triggered when the user selects an item. The handler is passed the option object and the index of the option in the <code>options</code> array. If you selected the item with the <code>select(index, opt)</code> method, the third argument to the event handler will be your <code>opt</code> object.
mouseout	Triggered when the user leaves the entire button group with mouse cursor. The handler is passed an event object as the only argument.

JointJS+ Selection

ui.Selection

Selection implements elements selection, moving the selection in one go and manipulating the selection in terms of selecting/deselecting individual elements.

Installation

Include `joint.ui.selection.js` and `joint.ui.selection.css` files to your HTML:

```
<link rel="stylesheet" type="text/css" href="joint.ui.selection.css">
<script src="joint.ui.selection.js"></script>
```

Initialize Selection

The Selection internally stores selected cells in a collection. This is a normal [Backbone Collection](#). It can be accessed via `collection` attribute. This view takes care of rendering the bounding rectangle during the selection action, determining which elements fall into this rectangle and also provides methods for selecting/deselecting individual elements. The view also listens to the underlying collection and updates selection boxes when an element is removed/added/reset i.e. `selection.collection.add(element)`; finds the view for the element on the paper and render a selection box above it.

```
var selection = new joint.ui.Selection({
  paper: paper,
  collection: new MyCollection // optional, a custom collection that inherits from
  Backbone.Collection
});
```

The next step is to hook the selection actions to the relevant events triggered on the paper and selection:

```
// Initiate selecting when the user grabs the blank area of the paper.
paper.on('blank:pointerdown', selection.startSelecting);

// Select an element if CTRL/Meta key is pressed while the element is clicked.
paper.on('element:pointerup', function(cellView, evt) {
  if (evt.ctrlKey || evt.metaKey) {
    selection.collection.add(cellView.model);
  }
});

// Unselect an element if the CTRL/Meta key is pressed while a selected element is
clicked.
```

```
selection.on('selection-box:pointerdown', function(elementView, evt) {
    if (evt.ctrlKey || evt.metaKey) {
        this.selection.collection.remove(elementView.model);
    }
});
```

Note that the `selection-box:pointerdown` event is triggered on the selection view when the mouse cursor is pressed above a selected element.

Selection Collection

As our selection collection is just a normal Backbone Collection, we can take advantage of the Backbone methods. For instance:

To select an element.

```
selection.collection.add(element);
selection.collection.add(element, { silent: true }); // add element to the
collection, but renders no selection box.
```

To deselect an element.

```
selection.collection.remove(element);
```

To deselect all elements.

```
selection.collection.reset([]);
```

To select multiple elements and deselect all the other elements.

```
selection.collection.reset([element1, element2]);
```

To identify elements that are currently in the selection.

```
selection.collection.on('reset add', function() {
    // Print types of all the elements in the selection.
    console.log(selection.collection.pluck('type'));
});
```

Customizing the look of Selection

The bounding rectangle of the active selection, the rectangles above the selected elements and the rectangle around all the elements once the selection is made can all be styled in CSS. The selection bounding rectangle is a `<div>` element with the `.joint-selection` class. The rectangles above the selected elements are also `<div>` elements having the

.selection-box class and the final rectangle surrounding all the selected element is a <div> with class .selection-wrapper . An example of a custom styling might look like the following:

```
.joint-selection {
  background-color: red;
  border: 2px dashed red;
  opacity: .7;
}
.selection-box {
  border: 1px solid #8e44ad;
  margin-top: -4px;
  margin-left: -4px;
  box-shadow: 2px 2px 5px #8e44ad;
}
.selection-wrapper {
  border: 1px dotted #8e44ad;
}
```

Disabling Selection tools

As you might have noticed, each selection is surrounded by a rectangle and offers three built-in icon tools by default: remove , rotate and resize . To disable any of these tools you may add a line similar to the following to your css:

```
.joint-selection .handle.rotate { display: none; } /* disables the rotate tool */
```

Another way to remove tool handles is via JavaScript:

```
selection.removeHandle('rotate');
```

To quickly disable all the tools (hide them) and also to hide the rectangular box around all the selected elements, you can simply do:

```
.selection-wrapper { display: none; }
```

Customizing Selection tools

Selection provides three methods for adding, removing and changing custom tools: `addHandle()` , `removeHandle()` and `changeHandle()` . Use the `addHandle()` method to add new tools to your selection:

```
selection.addHandle({ name: 'myaction', position: 's', icon: 'myaction.png' });
selection.on('action:myaction:pointerdown', function(evt) {
  evt.stopPropagation();
  alert('My custom action.');
```

```
});
```

In the example above, we added a new tool named `myaction`, positioned the tool to the south (bottom-center) and used our own icon `myaction.png`. When the user clicks on our tool, Selection triggers an event named `action:[name]:pointerdown`. We can handle the event by listening on the Selection object. Similarly, the Selection triggers `action:[name]:pointermove` and `action:[name]:pointerup` events. This gives us a high flexibility in implementing our own actions. You might have noticed that this API is exactly the same as in the [ui.Halo](#) plugin. The difference is that in Selection, we can have actions that manipulate multiple elements in one go.

For removing a tool, use the `removeHandle(name)` method:

```
selection.removeHandle('myaction');
```

`changeHandle()` method allows us to change tools. For instance, if we want to change a position of the remove tool, we could use:

```
selection.changeHandle('remove', { position: 'ne' });
```

Selection Configuration

paper	<i>(required)</i> The <code>joint.dia.Paper</code> or <code>joint.ui.PaperScroller</code> the Selection should operate on. If a <code>PaperScroller</code> object is provided, then its <code>paper</code> option is used as the basis for this Selection; in addition, the behavior specified by the <code>PaperScroller</code> 's <code>scrollWhileDragging</code> option is automatically applied for relevant Selection activities (selecting, translating, resizing).
graph	The JointJS graph the Selection should operate on. If no graph is passed <code>paper.model</code> is used.
collection	A Backbone Collection object used to store selected cells. If no collection is passed <code>new Backbone.Collection</code> is used.
filter	Either an array in which case it can contain (even intermixed) cells or cell types that will be ignored by the selection. This is useful if you have certain elements in the diagram that are not supposed to get selected. It can also be function in which case it will be called with a cell as an argument and should return <code>true</code> if that cell should be filtered out of the selection. The function gives you more flexibility in deciding whether an element can or cannot be selected. Example: <pre>filter: ['standard.Rectangle', 'mymodule.HiddenShape']</pre>
boxContent(cellView, boxDOMElement)	A function that returns an HTML string with the content that will be used in the information box below the selection. By default, the box shows the number of selected elements.

useModelGeometry	If set to true , the cells position and dimensions will be used as a basis for the Selection wrapping rectangle position. By default, this is set to false which causes the Selection wrapping rectangle position be based on the bounding box of the selected element views. Sometimes though, your shapes can have certain SVG sub elements that “stick out” of the view and you don't want these sub elements to affect the Selection wrapping rectangle position. In this case, set the useModelGeometry to true .
strictSelection	If set to true , only elements that are fully contained in the selection rectangle will be selected as opposed to the default behaviour that selects elements whose bounding box intersects the selection rectangle.
allowTranslate	If set to false , users are not allowed to move selected elements around by dragging selection boxes. It defaults to true .
preserveAspectRatio	Set to true if you want the resizing to preserve the aspect ratio of the elements. The default is false .
rotateAngleGrid	When selected elements are rotated the resulting angles are snapped to this value. It defaults to 15 degrees.
handles	A array of objects defining the tools around the selection. This array defaults to: <pre>[{ name: 'remove', position: 'nw', events: { pointerdown: 'removeElements' } }, { name: 'rotate', position: 'sw', events: { pointerdown: 'startRotating', pointermove: 'doRotate', pointerup: 'stopBatch' } }]</pre> Normally, you do not need to set this array. It's better to use the addHandle() , removeHandle() and changeHandle() methods described below in the API section.
translateConnectedLinks	When translating selected elements, should the connected links also be translated? Based on the selected value of the <code>ui.Selection.ConnectedLinksTranslation</code> enumerator, the behavior is as follows: • NONE - none of the links are translated • SUBGRAPH - only links whose both ends are connected to one of the selected elements are translated.

	• ALL - all connected links are translated The default is <code>ui.Selection.ConnectedLinksTranslation.ALL</code>.
allowCellInteraction	Allows interaction with underlying element's features, e. g. buttons or ports. In this case selection-box events will be triggered by browser event from the cell instead of the selection-box html element. It defaults to <code>false</code>.

Selection API

addHandle(opt)	Add a custom tool to the Selection. <code>opt.name</code> is the name of the custom tool. This name will be also set as a CSS class to the handle DOM element making it easy to select it your CSS stylesheet. <code>opt.position</code> is a string that specifies a position of the tool handle. Possible values are <code>n</code>, <code>nw</code>, <code>w</code>, <code>sw</code>, <code>s</code>, <code>se</code>, <code>e</code> and <code>ne</code>. <code>opt.icon</code> is a URL of the icon used to render the tool. This icons is set as a background image on the tool handle DOM element.
removeHandle(name)	Remove a tool handle named <code>name</code> from the Selection.
changeHandle(name, opt)	Change a tool handle named <code>name</code> in the Selection. The <code>opt</code> object can contain the same parameters as with the <code>addHandle()</code> method except of the <code>name</code> property. <code>opt</code> parameters will be merged with those defined previously.
on(event, callback)	Register a handler (<code>callback</code>) for an <code>event</code>. See the Selection Events section for the list of events the Selection object triggers.
startSelecting(evt)	Initiate bulk selection. See below in the Common Setup section how that can be used.
createSelectionBox(element)	Create a single selection box above an element.
destroySelectionBox(element)	Destroy a single selection box, which is rendered above an element.

Selection Events

The Selection object triggers events when the user manipulates its tools. These events can be handled by using the Selection `on()` method.

action: [name]:pointerdown	Triggered when the user clicks (touches) on a tool handle named <code>[name]</code>. The handler is called with the mousedown event object, and <code>x</code> and <code>y</code> of the pointer in local coordinates.

action: [name]:pointermove	Triggered when the user moves with mouse cursor after a tool handle named <code>[name]</code> was mousedowned (touched). The handler is called with the mousemove event object, and <code>x</code> and <code>y</code> of the pointer in local coordinates.
action: [name]:pointerup	Triggered when the user releases his mouse cursor after a tool handle named <code>[name]</code> was mousedowned (touched). The handler is called with the mouseup event object, and <code>x</code> and <code>y</code> of the pointer in local coordinates.
action: [name]:contextmenu	Triggered when the user invokes contextmenu on a tool handle named <code>[name]</code> . The handler is called with the mousedown event object, and <code>x</code> and <code>y</code> of the pointer in local coordinates.
selection-box: pointerdown	Triggered when the user starts interacting with a selected element, an element that has a selection box above it. The handler gets passed the element view, the DOM event object, <code>x</code> and <code>y</code> paper local coordinates. Use <code>elementView.model</code> if you wish to access the actual element model. In the case of <code>allowCellInteraction: true</code> the DOM event will be triggered by the element instead of selection-box. See below in the Common Setup section how that can be used.
selection-box: pointermove	Triggered when the user is moving with a selected element, an element that has a selection box above it. The handler gets passed the element view, the DOM event object, <code>x</code> and <code>y</code> paper local coordinates.
selection-box: pointerup	Triggered when the user stops interacting with a selected element, an element that has a selection box above it. The handler gets passed the element view, the DOM event object, <code>x</code> and <code>y</code> paper local coordinates.

Common Setup of the Selection in Applications Explained

The following is a common setup of the Selection in applications.

Cherry picking elements

For “cherry picking elements” - when the user clicks on an element holding the `[CTRL]` key, we register a handler for the `cell:pointerup` event on the paper, which is triggered when the user releases his mouse above a cell (either element or a link - links get filtered out in our case). In this handler, we add the element to our selection collection and create a selection box around that element.

```
paper.on('element:pointerup', function(elementView, evt) {
  if (evt.ctrlKey || evt.metaKey) {
    selection.collection.add(elementView.model);
  }
});
```

Releasing selection on cherry-picked elements

To implement the reverse action of cherry-picking, i.e. when the user clicks a selected element while holding the `[CTRL]` key, we register a handler for the `selection-box:pointerdown` event on the Selection (see [Selection events](#) for reference). In this handler, we remove the element from our selection collection and destroy the selection box.

```
selection.on('selection-box:pointerdown', function(elementView, evt) {  
    if (evt.ctrlKey || evt.metaKey) {  
        selection.collection.remove(elementView.model);  
    }  
});
```

Initiating bulk selection

Last thing we want to setup is the bulk selection that should happen when the user drags a blank area in the paper:

```
paper.on('blank:pointerdown', selection.startSelecting);
```

TIP: some applications might not want the user to be able to create selections when dragging a blank area in the paper. This is because they might have a different action for this, let's say *panning* the paper. In that case, you can, for example, start bulk selection only when the `[SHIFT]` key is being hold:

```
paper.on('blank:pointerdown', function(evt) {  
    if (evt.shiftKey) selection.startSelecting(evt);  
});
```

JointJS+ Snaplines

ui.Snaplines

Snaplines (a.k.a Guidelines) are great for guiding the user in aligning elements in your application. Snaplines are vertical and horizontal lines that appear when the user is dragging an element that is vertically or horizontally *close enough* to some other element.

Installation

Include `joint.ui.snaplines.css` and `joint.ui.snaplines.js` files to your HTML:

```
<link rel="stylesheet" type="text/css" href="joint.ui.snaplines.css">
<script src="joint.ui.snaplines.js"></script>
```

Usage

Enabling snaplines is as easy as creating a `joint.ui.Snaplines` object:

```
const snaplines = new joint.ui.Snaplines({ paper });
```

Demo

Configuration

The `joint.ui.Snaplines` constructor function takes the following parameters:

paper	The JointJS paper (<code>joint.dia.Paper</code>) object.
distance	The distance in pixels between edges (center) of other elements for snapping applies. The default is 10.
filter	Filter allows you to filter out elements taken into account for snapping. It can either be an array in which case the array is considered to contain cell types (or elements directly, they can even be intermixed) that should be filtered out from snapping. Or it can be a function with the following signature: <code>function(element)</code> in which case the function should return <code>true</code> if the element should not be taken into account for snapping.
usePaperGrid	When an element is snapped to other element, adjust its top-left corner to the paper's grid. Useful when the elements in the graph (or their centers) are not aligned with the grid. The default is <code>false</code> .

canSnap(elementView)	A function to determine whether an element will use snaplines while moving. All elements use snaplines by default.
additionalSnapPoints(elementView, options)	A function that allows the user to specify additional snap points. The first parameter of the callback is the <code>elementView</code> of the dragged/changed element. The second parameter is the <code>options</code> object. The <code>options</code> object contains a single <code>type</code> property that describes what kind of snaplines were invoked (it should snap because the element was moved <code>'move'</code> or resized <code>'resize'</code>). The return value represents an array of points (objects with properties <code>x</code> and <code>y</code>) to which the dragged element should snap. The returned points take precedence over the default snap points generated from the elements.

API

enable()	void	Enable the snaplines. The plugin will start listening to the paper and align elements as they move.
disable()	void	Disable the snaplines.
isDisabled()	boolean	Return <code>true</code> if the snaplines are disabled. Otherwise, return <code>false</code> . The plugin is enabled by default.

Styling Snaplines

Snaplines can be customized in CSS. Snapline is a `<div>` HTML element that has the `snapline` CSS class set. Moreover, you can customize horizontal and vertical snaplines differently by using the `horizontal` and `vertical` CSS class:

```
.snapline.horizontal {
  border-bottom: 2px dashed red;
  box-shadow: 0 0 2px black;
}
.snapline.vertical {
  border-right: 2px dotted blue;
  box-shadow: 0 0 2px black;
}
```

JointJS+ StackLayoutView

ui.StackLayoutView

`ui.StackLayoutView` is a view of a stack layout that implements moving elements between stacks, and provides interfaces for changing the default behaviour.

Installation

Include `joint.ui.StackLayoutView.js` and its dependency `joint.layout.StackLayout.js` in your HTML:

```
<script src="joint.layout.StackLayout.js"></script>
<script src="joint.ui.StackLayoutView.js"></script>
```

Usage

To use stack layout view you need to:

- Create an object from the `layout.StackLayoutView` class.
- Make sure that the elements you want to drag have their interactivity (`elementMove`) set to `false` (otherwise an exception is thrown).
- Every element, which is meant to be involved in the stack layout, should have the `stackIndex` attribute set (otherwise it will be ignored).

Source Code

```
const layoutOptions = {
  direction: 'TB',
  topBottom: {
    x: 20,
    y: 20
  },
  stackCount: 4,
  stackSize: 200
};

paper.setInteractivity(false);

const stackLayoutView = new joint.ui.StackLayoutView({
  layoutOptions,
  paper
});
```

ui.StackLayoutView.API

dragstart(element, x, y)	Start a new drag operation with <code>element</code> set as its source.
drag(element, x, y)	Drag the source element and display preview of the drop location if any.
dragend(element, x, y)	End the drag operation and move the source element to the target stack if there was a valid drop target.
canDrop()	Does the current drag operation have a valid drop target? i.e. an element can be moved to a new location.
cancelDrag()	Stop the current drag operation.

ui.StackLayoutView.configuration

The following table lists options that you can pass to the `ui.StackLayoutView` constructor function:

layoutOptions	<i>object</i>	An object corresponding to layout.StackLayout options.												
paper	<i>dia.Paper</i>	An instance of the paper object.												
preview(dragData, stackLayoutView)	<i>function</i>	A callback function to customize the preview element. It must return an <code>SVGElement</code> object. The preview's position will be set to the middle of the <code>stackSize</code> and to half of the <code>stackElementGap</code> automatically. The <code>dragData</code> has the following properties: <table><tr><td>sourceElement</td><td><i>object</i></td><td>The element being moved.</td></tr><tr><td>sourceStack</td><td><i>object</i></td><td>The source stack of the source element.</td></tr><tr><td>targetStack</td><td><i>object</i></td><td>The target stack.</td></tr><tr><td>insertElementIndex</td><td><i>number</i></td><td>The index in the target stack to</td></tr></table>	sourceElement	<i>object</i>	The element being moved.	sourceStack	<i>object</i>	The source stack of the source element.	targetStack	<i>object</i>	The target stack.	insertElementIndex	<i>number</i>	The index in the target stack to
sourceElement	<i>object</i>	The element being moved.												
sourceStack	<i>object</i>	The source stack of the source element.												
targetStack	<i>object</i>	The target stack.												
insertElementIndex	<i>number</i>	The index in the target stack to												

				which the element is moved.												
		invalid	<i>boolean</i>	Indicates if the current position is invalid (<code>validateMoving</code> returns <code>false</code>).												
		The plugin also adds the <code>stack-layout-preview</code> class name (and <code>stack-layout-preview-invalid</code> when the drop is invalid) for easy styling. <pre><code>.stack-layout-preview { stroke: #4666E5; stroke-width: 3; } .stack-layout-preview-invalid { stroke: #FF4365; }</code></pre>														
validateMoving(dragData, stackLayoutView)	<i>function</i>	A predicate, that returns <code>true</code> if the <code>element</code> , which the user interacts with, can be moved to another stack. It defaults to <code>return true</code>; . The <code>dragData</code> has the following properties: <table><tr><td>sourceElement</td><td><i>object</i></td><td>The element being moved.</td></tr><tr><td>sourceStack</td><td><i>object</i></td><td>The source stack of the source element.</td></tr><tr><td>targetStack</td><td><i>object</i></td><td>The target stack.</td></tr><tr><td>insertElementIndex</td><td><i>number</i></td><td>The index in the target stack to which the element is moved.</td></tr></table>			sourceElement	<i>object</i>	The element being moved.	sourceStack	<i>object</i>	The source stack of the source element.	targetStack	<i>object</i>	The target stack.	insertElementIndex	<i>number</i>	The index in the target stack to which the element is moved.
sourceElement	<i>object</i>	The element being moved.														
sourceStack	<i>object</i>	The source stack of the source element.														
targetStack	<i>object</i>	The target stack.														
insertElementIndex	<i>number</i>	The index in the target stack to which the element is moved.														
canInteract(elementView, stackLayoutView)	<i>function</i>	A predicate, that returns <code>true</code> if the <code>elementView</code> , which the user interacts with, can be dragged. By default, it returns <code>true</code> for elements with a defined <code>stackIndex</code> attribute.														

modifyInsertElementIndex(dragData, point, stackLayoutView)	<i>function</i>	The value of the <code>insertElementIndex</code> is derived from the pointer position and can be further modified using this callback. The callback is triggered before validation and can be used, for example, to find the nearest valid insert index. The <code>dragData</code> has the following properties: <table data-bbox="774 423 1482 981"> <tr> <td>sourceElement</td><td><i>object</i></td><td>The element being moved.</td></tr> <tr> <td>sourceStack</td><td><i>object</i></td><td>The source stack of the source element.</td></tr> <tr> <td>targetStack</td><td><i>object</i></td><td>The target stack.</td></tr> <tr> <td>insertElementIndex</td><td><i>number</i></td><td>The index in the target stack to which the element is moved.</td></tr> </table> The <code>point</code> is the current pointer coordinates.	sourceElement	<i>object</i>	The element being moved.	sourceStack	<i>object</i>	The source stack of the source element.	targetStack	<i>object</i>	The target stack.	insertElementIndex	<i>number</i>	The index in the target stack to which the element is moved.
sourceElement	<i>object</i>	The element being moved.												
sourceStack	<i>object</i>	The source stack of the source element.												
targetStack	<i>object</i>	The target stack.												
insertElementIndex	<i>number</i>	The index in the target stack to which the element is moved.												

When the user drags an element, the plugin creates a copy of the `elementView` DOM (ghost) and adds the class name `stack-layout-ghost` (and `stack-layout-ghost-invalid` if the drop is invalid).

```

.stack-layout-ghost rect {
  stroke: blue;
}
.stack-layout-ghost-invalid rect {
  stroke: blue;
}

```

ui.StackLayoutView.model

The `StackLayoutView` object has the `model` property that can be used to manipulate elements programmatically. The model has an `'update'` event which fires when the layout is updated. The model has following properties:

bbox	<i>object</i>	A bounding box around elements.
stacks	<i>Array<object></i>	The array of stack objects with these properties.

		<table><tr><td>bbbox</td><td><i>object</i></td><td>A bounding box around the stack.</td></tr><tr><td>elements</td><td><i>Array<object></i></td><td>The array of elements of the stack.</td></tr><tr><td>index</td><td><i>number</i></td><td>The index of the stack inside the layout.</td></tr></table>	bbbox	<i>object</i>	A bounding box around the stack.	elements	<i>Array<object></i>	The array of elements of the stack.	index	<i>number</i>	The index of the stack inside the layout.
bbbox	<i>object</i>	A bounding box around the stack.									
elements	<i>Array<object></i>	The array of elements of the stack.									
index	<i>number</i>	The index of the stack inside the layout.									
elements	<i>Array<object></i>	The array of elements to be laid out (elements with the <code>stackIndex</code> attribute defined).									
update()	Recalculates the layout manually. Useful when the elements in stacks are changed programmatically.										
getStackFromPoint(point)	Finds a stack from the point.										
getInsertElementIndexFromPoint(stack, point)	Finds an insert index in the stack from the point.										
getStackFromElement(element)	Finds a stack to which the element belongs.										

**insertElement(element, targetStack,
insertElementIndex)**

Inserts the
element in the
targetStack at the
specified index. It
calls `update()`
internally.

JointJS+ Stencil

ui.Stencil

Stencil plugin implements a palette of JointJS elements. These elements can be dragged onto a paper.

Installation

Include `joint.ui.stencil.js` and `joint.ui.stencil.css` to your HTML:

```
<link href="joint.ui.stencil.css" rel="stylesheet" type="text/css">
<script src="joint.ui.stencil.js"></script>
```

Creating a stencil

```
var graph = new joint.dia.Graph;
var paper = new joint.dia.Paper({
  el: $('#paper'),
  width: 500,
  height: 300,
  model: graph
});

var stencil = new joint.ui.Stencil({
  paper: paper,
  width: 200,
  height: 300
});

$('#stencil-holder').append(stencil.render().el);
```

The `joint.ui.Stencil` constructor takes a paper object and the dimensions of the stencil area. The next thing to do is rendering the stencil widget (`stencil.render()`) and appending the stencil element to a holder which can be any element in your HTML. Please see the demo for reference on how to do that if you're in doubts.

Stencil Configuration

The `joint.ui.Stencil` constructor function takes an options object as an argument. The options object can have the following parameters:

paper	the <code>joint.dia.Paper</code> or <code>joint.ui.PaperScroller</code> object [mandatory]

width	the width of the stencil [mandatory]
height	the height of the stencil [mandatory]
search	<i>object</i> - An object defining what properties of the elements in the stencil will be used for searching. The object has element types as keys (or <code>*</code> wildcard denoting any type) and arrays with element property paths as values. The current element values under those property paths will be used by stencil to find a match between the search term and those values. <pre> search: { '*': ['attrs/text/text'], 'standard.Image': ['description'], 'standard.Path': ['description'] } </pre> <i>function</i> - A function executed on every cell in the stencil returning <code>true</code> (matched) or <code>false</code> (unmatched). <pre> search: function(cell, keyword, group, stencil) { var description = cell.get('description'); return typeof description === 'string' && description.indexOf(keyword) > -1; } </pre> See the Searchable Stencil section for more information.
groups	An object that defines the groups in the stencil (if any). The keys of this object are strings that uniquely identify the groups and values are objects that contain <code>label</code>, <code>index</code> and <code>closed</code> properties. <code>label</code> property specifies the name of the group that will be displayed in the UI, <code>index</code> is a number specifying the position of the group among other groups within the stencil and the optional <code>closed</code> property tells stencil whether this group should be closed or opened by default. An example of this object can look like: <pre> groups: { basic: { label: 'Basic Elements', index: 1 }, text: { label: 'Text', index: 2, closed: true }, advanced: { label: 'Advanced Elements', index: 3, closed: true } } </pre>

groupsToggleButtons	If set to <code>true</code> buttons for expanding/collapsing groups (expand all/collapse all) are rendered into the stencil. It defaults to <code>false</code> .
dropAnimation	An object defining an animation that will be performed on an element that is dropped outside the target paper area. It defaults to <code>undefined</code> meaning that there will be no animation. The <code>dropAnimation</code> object can have <code>duration</code> and <code>easing</code> properties where <code>duration</code> is the duration of the animation in milliseconds (defaults to <code>150</code>) and <code>easing</code> is either <code>"swing"</code> (the default) or <code>"linear"</code>. Example: <pre> dropAnimation: { duration: 300, easing: 'linear' } </pre>
label	A string or HTML Element rendered at the top of the stencil. It defaults to <code>`Stencil`</code> .
layout	If set to <code>true</code>, the stencil elements are automatically laid out using the Grid Layout plugin. To adjust the layout behavior, pass an object defining the layout configuration. For all available options see the aforementioned plugin. <pre> layout: { columnWidth: 100, columns: 3, rowHeight: 100, } </pre> The layout can be defined as a function in which case it will be called for each group on initialization and after the stencil is filtered. The function receives the group graph as the first parameter and the group definition as the second parameter. Note that the graph does not contain elements that don't match the search criteria (if any). Each graph has the <code>`group`</code> attribute set to the group id. <pre> layout: function(graph, group) { const groupName = graph.get('group'); if (groupName === 'my-group-1') { // layout the elements in this group only graph.getElements().forEach(function(el, index) { el.position(0, index * 100); }); } }); </pre>

dragStartClone(element)	A function that produces an element clone when the user starts dragging. It defaults to <code>element.clone()</code> .
dragEndClone(element)	A function that produces an element clone when the user places the dragged element into the paper. It defaults to <code>element.clone()</code> .
canDrag(elementView, evt, groupName)	A function to determine whether an element from the stencil can be dragged by the user or not. All elements can be dragged by default.
snaplines	An instance of the Snapline plugin which is responsible for drawing snaplines while the user drags an element from the stencil.
scaleClones	When set to <code>true</code> dragged clones are automatically scaled based on the current paper transformations. Note, that this option is ignored when <code>snaplines</code> option applied (it always scales the elements). It defaults to <code>false</code> .
paperOptions	An object with <code>joint.dia.Paper</code> options to adjust the look and behavior of the stencil's papers. e.g. <pre> paperOptions: { elementView: CustomElementView, background: { color: 'red' } } </pre> This options can be also used to adjust a single paper (and takes precedence) when defined inside of the group definition. e.g. <pre> groups: { myGroup: { index: 1, label: 'My Group', paperOptions: { color: 'blue' } } } </pre> To adjust the graph in Stencil papers it's also possible to pass a <code>model</code> as well. Please note, the <code>paperOptions</code> has to be a function in such a case: <pre> paperOptions: function() { return { elementView: CustomElementView, model: new joint.dia.Graph([], { cellNamespace: CustomElementNamespace }), background: { color: 'red' } } } </pre>

	<pre> } } </pre>
contentOptions	When the stencil papers are resized to fit their content, the joint.dia.Paper.fitToContent is used internally. Use this option to modify the behavior.
container	A CSS selector or a DOM element is the container element, which the element being dragged is appended to.
paperPadding	<i>(deprecated)</i> A number specifying the padding that will be used in the internal papers that hold the elements in the stencil. It defaults to <code>10</code> .

Populating Stencil with elements

```

const r = new joint.shapes.standard.Rectangle({
  position: { x: 10, y: 10 }, size: { width: 50, height: 30 }
});
const c = new joint.shapes.standard.Circle({
  position: { x: 70, y: 10 }, size: { width: 50, height: 30 }
});

// Stencil is rendered in the DOM now.

// load the elements into the default group
stencil.load([r, c]);

// load the elements into multiple groups
stencil.load({ group1: [r], group2: [c] });

// load the elements into the `group2` group
stencil.loadGroup([r, c], 'group2');

```

As you can see, there is multiple way how to populate the stencil with elements. Use either `stencil.load()` or `stencil.loadGroup()` which make sure the elements are rendered into the stencil.

Note: Populating the stencil with elements needs to be done after the stencil is rendered in the DOM.

Stencil groups (a.k.a. Accordion)

Creating an accordion-like Stencil is as easy as passing the `groups` object to the stencil constructor. `groups` object is a hash table mapping group identifiers to the configuration of the particular group. Each group configuration object contains these properties:

label	group label. Can be either a <code>string</code> or a <code>html</code> .
index	position of the group in the stencil.
closed	<i>[optional]</i> when <code>true</code> , this group will be initially closed.
height	<i>[optional]</i> the height of the paper containing the shapes of the group. If not defined, then the height of the stencil will be used. If none of these heights are defined, the paper will be automatically scaled to fit its content (recommended).
layout	<i>[optional]</i> same as the stencil <u>layout</u> option. It allows you to override the global settings for a particular group.
paperOptions	<i>[optional]</i> same as the stencil <u>paperOptions</u> option. It allows you to override the global settings for a particular group.

```
const stencil = new joint.ui.Stencil({
  graph: graph,
  paper: paper,
  width: 200,
  groups: {
    one: { label: 'First group', index: 1 },
    two: { label: 'Second group', index: 2, closed: true }
  }
});
```

Searchable Stencil

The Stencil allows users to filter the elements by an arbitrary keyword. That is useful especially when the palette contains a lot of elements. To enable this add `search: { /* rules */ }` option to the Stencil constructor.

The rules determine what attributes we match a given keyword with and are defined by an object with `ELEMENT_TYPE`: `[ATTRIBUTE_PATH1, ATTRIBUTE_PATH2, ...]` key-value pairs. Where `ELEMENT_TYPE` is either an element type (e.g `standard.Rectangle`) or `'*'` to match any type. `ATTRIBUTE_PATH` is a path to element's attribute or property (e.g `'attrs/text/text'`, `'description'`).

```
const stencil = new joint.ui.Stencil({
  search: {
    '*': ['attrs/text/text'],
    'standard.Image': ['description'],
    'standard.Path': ['description']
  }
});
```


The rules can be alternatively defined as a function, which, upon search, will be invoked for each element with the element model, string keyword, the group id the element belongs to, and the stencil instance as its arguments.

```
const stencil = new joint.ui.Stencil({
  search: function(element, keyword, groupId, stencil) {
    return element.get('type').includes(keyword) || groupId.includes(keyword);
  }
});
```

By default all the unmatched elements and groups are hidden (`display: none`; is set on them). Alternatively you can add `.joint-stencil .joint-element.unmatched { display: block; }` and `.joint-stencil .group.unmatched { display: block }` rules to your css stylesheet in order to have the unmatched elements translucent.

Here's a list of all CSS classes toggled by filtering.

ClassName	CSS Selector	Description
<i>stencil-filtered</i>	<code>.joint-stencil.stencil-filtered</code>	A keyword was provided. The stencil is filtered.
<i>not-found</i>	<code>.joint-stencil.not-found</code>	There are no results for the given keyword.
<i>unmatched</i>	<code>.joint-stencil .group.unmatched</code>	A stencil group of which none of the elements match the filter.
<i>unmatched</i>	<code>.joint-stencil .joint-element.unmatched</code>	An element, which does not match the filter.

The Stencil triggers the `'filter'` event with a matched subset of elements wrapped in `joint.dia.Graph` as the first argument every time the elements get filtered.

The default text of the Stencil can be modified as shown below.

```
// No Matches Found
stencil.el.dataset.textNoMatchesFound = 'No matches found!';
// Search Placeholder
stencil.el.querySelector('.search').placeholder = 'My Placeholder';
```

API

load(groupedElements)	Accept an object with multiple key-value pairs, where <i>key</i> is the name of a group and <i>value</i> an array of elements to be rendered into that group. For instance:

	<pre> stencil.load({ group1: [{ type: 'standard.Rectangle' }, { type: 'standard.Circle' }], group2: [{ type: 'standard.Image' }] }); </pre> Note that the elements could be also defined as plain javascript objects. That is useful for example when you store the stencil configuration in a JSON, which is received from a DB. If the method is called with an array it behaves like the <code>loadGroup</code> below.
loadGroup(elements [, group])	Accept an array of elements and render them into a single stencil group. If no <code>group</code> provided, the default group is used.
getGraph(groupName)	Get the graph associated with the group identified by <code>groupName</code> . If the stencil does not use groups, just omit the <code>groupName</code> parameter to get the only graph present.
getPaper(groupName)	Get the paper associated with the group identified by <code>groupName</code> . If the stencil does not use groups, just omit the <code>groupName</code> parameter to get the only paper present.
setPaper(paper)	Set the target paper for the stencil. It tells stencil to use a different paper than the one that was passed to it in the initialization through the options object. This is useful if you have a stencil instantiated and want to change the target paper dynamically. For example, if you have tabs each having its own paper with its own diagram but you want to use only one stencil, they you can call <code>setPaper(myTab2)</code> whenever the active tab changes. Note that the <code>paper</code> argument can be both a <code>joint.dia.Paper</code> object or the <code>joint.ui.PaperScroller</code> object, the stencil can handle both.
filter(keyword [, searchOptions])	Filter the stencil elements based on the given <code>keyword</code> and the current (or provided) <u>search options</u> .
openGroup(groupName)	Open group <code>groupName</code> .
closeGroup(groupName)	Close group <code>groupName</code> .
toggleGroup(groupName)	Toggle group <code>groupName</code> .
isGroupOpen(groupName)	Return <code>true</code> if the group with the <code>groupName</code> is open. Return <code>false</code> otherwise.

openGroups()	Open all groups.
closeGroups()	Close all groups.
stopListening()	Disable dragging of elements from the stencil.
startListening()	Enable dragging of elements from the stencil.
freeze(opt)	Freeze all the papers of the stencil. In this state, the paper does not automatically re-render upon changes in the graph. This is useful when for instance the stencil is collapsed (not visible on the screen). For more information see paper.freeze() .
unfreeze(opt)	Unfreeze all the papers of the navigator. For more information see paper.unfreeze() .
startDragging(cell, event)	Start dragging an arbitrary cell (element or link). The cell will be added to the preview paper as is (i.e. the <code>dragStartClone()</code> callback will not be called). The <code>dragEndClone()</code> callback will be called as usual. The <code>event</code> argument is a DOM event object that defines, using <code>clientX</code> and <code>clientY</code>, where the preview is to be displayed. Since the cell does not have to be from the stencil, this method can be used to initiate dragging an element outside the stencil (e.g. from an HTML list). Note that you should not render the stencil if it is never to be attached to the DOM. Calling <code>stencil.render()</code> would create detached DOM elements and take up memory unnecessarily. HTML: <pre> <li data-type="element">Element <li data-type="link">Link </pre> JS: <pre>const stencil = new ui.Stencil({ paper }); const listEl = document.querySelector('ul'); listEl.addEventListener('pointerdown', (evt) => { const { type } = evt.target.dataset; let cell; switch (type) { case 'element':</pre>

	<pre> cell = new shapes.standard.Rectangle({ size: { width: 100, height: 100 } }); break; case 'link': cell = new shapes.standard.Link({ source: { x: 0, y: 0 }, target: { x: 100, y: 100 } }); break; default: throw new Error(`Unknown type: \${type}`); } stencil.startDragging(cell, evt); });</pre>
cancelDrag(opt)	Stop the current drag operation. If <code>opt.dropAnimation</code> is not specified, the default <code>stencil.options.dropAnimation</code> is used.
isDragCanceled()	Was the current drag operation canceled? Returns a boolean value.

Reacting on elements added to the paper

Many times, you might want to perform some action as a reaction on a new element dragged from the stencil to the paper. This can be achieved by usual means, i.e. by reacting on new elements added to the graph:

```
graph.on('add', function(cell, collection, opt) {
    // The stencil adds the `stencil` property to the option object with value
    // set to a client id (`cid`) of the stencil view.
    if (opt.stencil) {
        console.log('A cell with id', cell.id, 'was just added to the paper from
the stencil.');
```

Events

The Stencil object triggers events that you can react on in your applications. These events can be handled by using the Stencil `on(eventName, handler)` method.

Event Name	Handler Signature	Description

element:dragstart	<pre>cloneView: dia.ElementView, evt: dia.Event, dropArea: g.Rect, validDropTarget: boolean</pre>	Triggered when the user starts dragging an element. See <code>element:drag</code> for more information about the parameters passed to the handler function.
element:drag	<pre>cloneView: dia.ElementView, evt: dia.Event, dropArea: g.Rect, validDropTarget: boolean</pre>	Triggered while an element is being dragged. The <code>cloneView</code> is an element view rendered for the model returned by the <code>dragStartClone()</code> method. <code>dropArea</code> is the area (in local coordinates), which the element would occupy if it were a part of the target graph. <code>validDropTarget</code> is a boolean value that answers whether the element is currently over a valid target paper. <pre>stencil.on('element:drag', (cloneView, evt, dropArea, validDropTarget) => { cloneView.vel.attr('opacity', validDropTarget ? 1 : 0.3); });</pre>
element:dragend	<pre>cloneView: dia.ElementView</pre>	Triggered when a drag operation ends (such as by releasing a mouse button or by calling the <code>cancelDrag()</code> method). This event fires regardless of whether the drag completed or was canceled. The dragend event handler can check <code>stencil.isDragCanceled()</code> and the <code>validDropTarget</code> parameter to determine whether the drag

	<pre>iew, evt: dia.Event, dropArea: g.Rect, , validDropTarget: boolean</pre>	operation had succeeded or not. See <code>element:drag</code> for more information about the parameters passed to the handler function.
element:drop	<pre>elementView: dia.ElementView, evt: dia.Event, x: number, y: number</pre>	Triggered when an element is dropped onto a valid target paper. The <code>elementView</code> is an element view rendered for the model returned by the <code>dragEndClone()</code> method. The <code>x</code> and <code>y</code> are coordinates (in local coordinate system) of the user pointer at the moment of the drop event. <pre>stencil.on('element:drop', (elementView) => { const halo = new joint.ui.Halo({ cellView: elementView }); halo.render(); });</pre>
drop:invalid	<pre>evt: dia.Event, element: dia.Element</pre>	Triggered when the user drops an element from the stencil to an invalid area. Invalid area is defined as an area that is outside of the visible paper area so that the dropped element is not accepted by the target paper. Such an element is then either immediately disposed or returned to its original position in stencil in an animated fashion (see the <code>dropAnimation</code> option). This event gives you a chance to react when the invalid drop occurs. The <code>evt</code> is a mouse event object and <code>element</code> is the element model that would have been otherwise dropped into the main paper (and graph for that matter).
filter	<pre>graph: dia.Graph,</pre>	Triggered when the user uses the search input field to filter elements in the stencil. The <code>graph</code> is a set of elements that matched the filter, <code>groupName</code> is

	<pre>groupName: string, keyword: string</pre>	the name of the group in which elements were filtered and keyword is the current search term.
group:open	<pre>groupName: string</pre>	Triggered when the user opens a closed group.
group:close	<pre>groupName: string</pre>	Triggered when the user closes an open group.

JointJS+ TextEditor

ui.TextEditor

`ui.TextEditor` is a powerful (rich-text) inline text editor that can be used on any SVG text element. The `ui.TextEditor` provides the following features:

- rich-text editing of SVG text
- caret & selections
- caret & selections can be styled in CSS
- double-click to select words, triple-click to select the entire text
- keyboard navigation native to the underlying OS
- API for programmatic access
- support for editing scaled & rotated text
- automatic URL hyperlinks detection and styling

Installation

Include `joint.ui.textEditor.js` and `joint.ui.textEditor.css` files to your HTML:

```
<link rel="stylesheet" type="text/css" href="joint.ui.textEditor.css">
<script src="joint.ui.textEditor.js"></script>
```

Usage

In the following example, we will show how to start inline text editing when the user double-clicks a text inside a JointJS element or a link. First we handle the `cell:pointerdblclick` event triggered by the paper. Inside our handler, we call the `joint.ui.TextEditor.edit()` method which starts the text editor on the SVG text under the target of the event. The only parameters we have to pass to this method is the SVG text DOM element, the cell view and the property path that will the text editor use to store the resulting text. Moreover, we also show how easy it is to apply auto-sizing on the text inside our elements so that they stretch based on the bounding box inside it.

```
// RECOMMENDED

paper.on('element:pointerdblclick', function(elementView, evt) {
  joint.ui.TextEditor.edit(evt.target, {
    cellView: elementView,
    textProperty: ['attrs', 'label', 'text'],
    annotationsProperty: ['attrs', 'label', 'annotations']
  });
});

paper.on('link:pointerdblclick', function(linkView, evt) {
```



```

// Editing a link label
var index = Number(linkView.findAttribute('label-idx', evt.target));
joint.ui.TextEditor.edit(evt.target, {
  cellView: linkView,
  textProperty: ['labels', index, 'attrs', 'text', 'text'],
  annotationsProperty: ['labels', index, 'attrs', 'text', 'annotations']
});
});

function autoSize(element) {

  var view = paper.findViewByModel(element);
  var textVel = view.vel.findOne('text');
  // Use bounding box without transformations so that our auto-sizing works
  // even on e.g. rotated element.
  var bbox = textVel.getBBox();
  // 16 = 2*8 which is the translation defined via ref-x ref-y for our rb element.
  element.resize(bbox.width + 16, bbox.height + 16);
}

graph.on('change:attrs', function(cell) { autoSize(cell) });

```

Note that there are two methods for starting up a text editor. The one we showed above (via the `joint.ui.TextEditor.edit()` function) is the easiest and recommended. The `joint.ui.TextEditor.edit()` function is a helper method that does a lot of things for us. First, it finds the nearest SVG `<text>` DOM element. Then it creates an instance of the `joint.ui.TextEditor` type and binds handlers for the `'text:change'` event where it stores the new text content to your JointJS cells (elements or links). It also deals with annotations in case of rich text editing and more.

However, for completeness, we show another way to start text editing and that is by manually creating the instance of the `joint.ui.TextEditor` (not recommended if you don't know what you're doing):

```

// NOT RECOMMENDED

var ed;
paper.on('cell:pointerdblclick', function(cellView, evt) {
  var text = joint.ui.TextEditor.getTextElement(evt.target);
  if (text) {
    if (ed) ed.remove(); // Remove old editor if there was one.
    ed = new joint.ui.TextEditor({ text: text });
    ed.render(paper.el);

    ed.on('text:change', function(newText) {
      // Set the new text to the property that you use to change text in your
      views.
      cellView.model.attr('text/text', newText);
    });
  }
});

```

```
}  
});
```

Note that using the first method, you can still access the actual instance of the text editor via `joint.ui.TextEditor.ed` property.

See the source code of the demo below for a complete example on using the inline text editor, including inline text editing of link labels.

Source Code

Configuration

The following table lists options that you can pass to the `ui.TextEditor.edit(el, opt)` method (**this is the recommended way to start you text editor**):

el	SVG <code><text></code> element or any of its descendants. Usually, you pass the event target object when the user doubleclicks your JointJS element/link (see examples above).
opt.cellView	The view for your cell. Usually, you either have a direct access to this view (in the paper triggered <code>'cell:pointerclick'</code> or <code>'cell:pointerdblclick'</code> events) but you can always find a view for any JointJS element or link by using <code>paper.findViewByModel(cell)</code> .
opt.textProperty	The path to the text property that causes the element/link to re-render the text inside it. (e.g. <code>'attrs/text/text'</code>).
opt.annotationsProperty	The path to the property that holds the text annotations. (e.g. <code>'attrs/text/annotations'</code>). (<i>Rich-text specific.</i>)
opt.placeholder	Set to <code>true</code> to show a placeholder when the text is empty. The placeholder text and visual look can be styled in CSS as follows: <pre>.text-editor .caret.placeholder:before { // The caret. background-color: red; } .text-editor .caret.placeholder:after { // The placeholder text. content: 'Enter text...'; font-style: italic; color: blue; }</pre>
opt.annotations	Annotations that will be set on the <code>ui.TextEditor</code> instance. (<i>Rich-text specific.</i>)

opt.annotateUrls	Set to <code>true</code> to enable the text editor to automatically detect URL hyperlinks and annotate them using the default <code>urlAnnotation</code> .
opt.urlAnnotation	The default annotation for detected URL hyperlinks: <pre> { attrs: { 'class': 'url-annotation', fill: 'lightblue', 'text-decoration': 'underline' } } </pre> It could be defined as an object or a function that returns an object. <pre> function(url) { return { attrs: { 'data-url': url, 'fill': 'blue', 'text-decoration': 'underline' } } } </pre>
opt.useNativeSelection	ui.TextEditor uses <u>native selections</u> for selecting text. This can be disabled by setting <code>useNativeSelection</code> to <code>false</code> in which case ui.TextEditor renders its own selections. This is an advanced feature that is useful only in very rare occasions. For example, if you, for any reason, need to select text in two different elements at the same time, you should set the <code>useNativeSelection</code> to <code>false</code>. This is because if native selections are used, two or more text elements cannot have focus at the same time.
opt.textareaAttributes	Set attributes on the underlying (invisible) textarea that is used internally by the text editor to handle keyboard text input. This is an advanced option and is useful in very rare situations. The default value is: <pre> textareaAttributes: { autocorrect: 'off', autocomplete: 'off', autocapitalize: 'off', spellcheck: 'false', tabindex: '0' } </pre>

	Some jQuery UI users reported an issue with <code>autocomplete</code> attribute being set on the textarea, resulting in the following error thrown in the jQuery UI: <i>Uncaught Error: cannot call methods on autocomplete prior to initialization; attempted to call method 'off'</i>. In this case, you might want to set your own <code>textareaAttributes</code> to overcome this issue (in this specific case, omitting the <code>autocomplete</code> attribute).
opt.onKeyDown	The callback is triggered when a key is pressed. It is possible to stop the event from propagating to the text editor by calling <code>evt.stopPropagation()</code>. <pre> onKeyDown: (evt: KeyboardEvent) => { if (evt.code === 'Enter') { evt.stopPropagation(); TextEditor.close(); } } </pre>
opt.onOutsidePointerdown	The callback is triggered when the user clicks/touches outside the text editor. <pre> onOutsidePointerdown: (evt: PointerEvent) => { TextEditor.close(); } </pre>

The following table lists options that you can pass to the `ui.TextEditor` constructor function (**only for advanced use**):

text	SVG <code><text></code> element. It is recommended to use the <code>joint.ui.TextEditor.getTextElement(el)</code> method to find the container <code><text></code> SVG element wrapping all the lines. You can use this method directly on the <code>evt.target</code> element in your event handlers and just check if the returned value is truthy.
placeholder	Set to <code>true</code> to show a default placeholder (<i>Enter text...</i>) when the text is empty. Set it to a string to show a custom placeholder. The placeholder visual look can be styled in CSS as follows: <pre> .text-editor .caret.placeholder:before { // The caret. background-color: red; } .text-editor .caret.placeholder:after { // The placeholder text. font-style: italic; color: blue; } </pre>

API

The following table lists methods that can be used on the `joint.ui.TextEditor` instance. If you, however, use the recommended way of creating the text editor via the `joint.ui.TextEditor.edit()` function, you usually do not want to deal with the actual instance (even though you can as you can always access the actual instance via `joint.ui.TextEditor.ed` property). To make it easy to work with the instance as with a black-box, `joint.ui.TextEditor` constructor exposes many of the methods directly (internally, it just proxies the methods to the actual instance.) The table below differentiates the instance methods from the class methods by using the full namespace access for the class methods (such as `joint.ui.TextEditor.getAnnotations()` = class method - vs `getAnnotations()` = instance method).

<code>render([root])</code>	Render the Text Editor. Call this method right after you have constructed the text editor instance and you're ready to display the inline text editor. If <code>root</code> (HTML DOM element) is specified, the text editor HTML container will be appended to that element instead of to the <code>document.body</code> . This is useful if you want the text editor to scroll with some other container.
<code>remove()</code>	Remove the Text Editor. Call this when you're done with inline text editing.
<code>selectAll()</code>	Programmatically select all the text inside the text editor.
<code>select(selectionStart, selectionEnd)</code>	Programmatically select portion of the text inside the text editor starting at <code>selectionStart</code> ending at <code>selectionEnd</code> . This method automatically swaps <code>selectionStart</code> and <code>selectionEnd</code> if they are in a wrong order.
<code>deselect()</code>	Programmatically deselect all the selected text inside the text editor.
<code>getSelectionStart()</code>	Return the start character position of the current selection.
<code>getSelectionEnd()</code>	Return the end character position of the current selection.

<code>getSelectionRange()</code>	Return an object of the form <code>{ start: Number, end: Number }</code> containing the start and end position of the current selection. Note that the start and end positions are returned <i>normalized</i> . This means that the start index will always be lower than the end index even though the user started selecting the text from the end back to the start.
<code>getSelectionLength()</code>	Return the number of characters in the current selection.
<code>getSelection()</code>	Return the selected text.
<code>startSelecting()</code>	Starts the text selection (i.e. updates the selection range based on the mouse coordinates until the user releases the mouse).
<code>findAnnotationsUnderCursor(annotations, selectionStart)</code>	Find all the annotations in the <code>`annotations`</code> array that the cursor at <code>`selectionStart`</code> position falls into.
<code>findAnnotationsInSelection(annotations, selectionStart, selectionEnd)</code>	Find all the annotations that fall into the selection range specified by <code>`selectionStart`</code> and <code>`selectionEnd`</code> . This method assumes the selection range is normalized.
<code>setCaret(charNum [, opt])</code>	Programmatically set the caret position. If <code>opt.silent</code> is <code>true</code> , the text editor will not trigger the <code>'caret:change'</code> event.
<code>hideCaret()</code>	Programmatically hide the caret.
<code>updateCaret()</code>	Manually update the visual attributes (e.g. size, color) of the caret. It's useful when the containing SVG was transformed.
<code>getTextContent()</code>	Return the text content (including new line characters) inside the text editor.

getWordBoundary(charNum)	Return the start and end character positions for a word under <code>charNum</code> character position.
getURLBoundary(charNum)	Return the start and end character positions for a URL under <code>charNum</code> character position. Return <code>undefined</code> if there was no URL recognized at the <code>charNum</code> index.
getNumberOfChars()	Return the number of characters in the text.
getCharNumFromEvent(evt)	Return the character position the user clicked on. If there is no such a position found, return the last one.
setCurrentAnnotation(attrs)	This method stores annotation attributes that will be used for the very next insert operation. This is useful, for example, when we have a toolbar and the user changes text to e.g. bold. At this point, we can just call <code>setCurrentAnnotation({ 'font-weight': 'bold' })</code> and let the text editor know that once the user starts typing, the text should be bold. Note that the current annotation will be removed right after the first text operation came. This is because after that, the next inserted character will already inherit properties from the previous character which is our 'bold' text. <i>(Rich-text specific.)</i>
setAnnotations(annotations)	Set annotations of the text inside the text editor. These annotations will be modified during the course of using the text editor. <i>(Rich-text specific.)</i>
getAnnotations()	Return the annotations of the text inside the text editor. <i>(Rich-text specific.)</i>
getCombinedAnnotationAttrsAtIndex(index, annotations)	Get the combined (merged) attributes for a character at the position <code>index</code> taking into account all the

	annotations that apply. (<i>Rich-text specific.</i>)
<code>getSelectionAttrs(range, annotations)</code>	Find a common annotation among all the annotations that fall into the range (an object with start and end properties - <i>normalized</i>). For characters that don't fall into any of the annotations, assume <code>defaultAnnotation</code> (default annotation does not need start and end properties). The common annotation denotes the attributes that all the characters in the range share. If any of the attributes for any character inside range differ, <code>undefined</code> is returned. This is useful e.g. when your toolbar needs to reflect the text attributes of a selection. (<i>Rich-text specific.</i>)
<code>joint.ui.TextEditor.getTextElement(el)</code>	Return the containing SVG <code><text></code> element closest to the SVG element <code>el</code>. This is especially handy in event handlers where you can just pass <code>evt.target</code> element to this function and it will return the element that you can directly use as a parameter to the <code>ui.TextElement</code> constructor function. If no such element exists, the function returns <code>undefined</code>.
<code>joint.ui.TextEditor.edit(el, opt)</code>	See the Configuration section for details.
<code>joint.ui.TextEditor.close()</code>	Close the currently opened text editor (if there was one). Note that before actually removing the editor (and if the <code>annotateUrls</code> option is set to <code>true</code>), the <code>close()</code> method tries to find if there was a URL hyperlink at the current cursor position and annotates it. The reason is that normally, URL hyperlinks are detected after the user types a non-visible character such as SPACE or new line. In this case though, the user closed

	the editor by e.g. clicking outside the element/link and so if there was a URL hyperlink at the end of the text content, it would not have been annotated.
<code>joint.ui.TextEditor.applyAnnotation(annotations)</code>	Apply <code>annotations</code> to the current selection. This is useful if the user has something selected in the text editor and then changes e.g. the font size via a toolbar. In this case, you construct the <code>annotations</code> array from the toolbar changes and apply it using this function on the current selection
<code>joint.ui.TextEditor.setCurrentAnnotation(attributes)</code>	See the instance <code>setCurrentAnnotation()</code> method above.
<code>joint.ui.TextEditor.getAnnotations()</code>	See the instance <code>getAnnotations()</code> method above.
<code>joint.ui.TextEditor.setCaret(charNum, opt)</code>	See the instance <code>setCaret()</code> method above.
<code>joint.ui.TextEditor.deselect()</code>	See the instance <code>deselect()</code> method above.
<code>joint.ui.TextEditor.selectAll()</code>	See the instance <code>selectAll()</code> method above.
<code>joint.ui.TextEditor.select(start, end)</code>	See the instance <code>select()</code> method above.
<code>joint.ui.TextEditor.getNumberOfChars()</code>	See the instance <code>getNumberOfChars()</code> method above.
<code>joint.ui.TextEditor.getCharNumFromEvent(evt)</code>	See the instance <code>getCharNumFromEvent()</code> method above.

<code>joint.ui.TextEditor.getWordBoundary()</code>	See the instance getWordBoundary() method above.
<code>joint.ui.TextEditor.findAnnotationsUnderCursor()</code>	See the instance findAnnotationsUnderCursor() method above except the <code>annotations</code> and <code>index</code> arguments are automatically added so this method does not accept any arguments.
<code>joint.ui.TextEditor.findAnnotationsInSelection()</code>	See the instance findAnnotationsInSelection() method above except the <code>annotations</code> and <code>start</code> and <code>end</code> index of the selection are automatically added so this method does not accept any arguments.
<code>joint.ui.TextEditor.getSelectionAttrs(annotations)</code>	See the instance getSelectionAttrs() method above except the <code>range</code> is automatically added.
<code>joint.ui.TextEditor.getSelectionLength()</code>	See the instance getSelectionLength() method above.
<code>joint.ui.TextEditor.getSelectionRange()</code>	See the instance getSelectionRange() method above.
<code>joint.ui.TextEditor.isLineStart(text, index)</code>	Return <code>true</code> if the character at the position <code>index</code> is the first character of some line.
<code>joint.ui.TextEditor.isLineEnding(text, index)</code>	Return <code>true</code> if the character at the position <code>index</code> is a newline character but does not denote an empty line. In other words, the newline character under <code>index</code> terminates a non-empty line.
<code>joint.ui.TextEditor.isEmptyLine(text, index)</code>	Return <code>true</code> if the character at the position <code>index</code> is a newline

	character that denotes an empty line
<code>joint.ui.TextEditor.normalizeAnnotations(annotations)</code>	Returns an array of annotations that contains no duplicates and is sorted by the <code>start</code> index of each annotation. The method generates annotations containing a maximum number of attributes rather than aiming for the maximum <code>range</code> .
<code>joint.ui.TextEditor.getCombinedAnnotationAttrsAtIndex(annotations, index)</code>	Get the combined (merged) attributes for a character at the position <code>index</code> taking into account all the annotations that apply.
<code>joint.ui.TextEditor.getCombinedAnnotationAttrsBetweenIndexes(annotations, start, end)</code>	Find a common annotation among all the annotations that fall into the range between <code>start</code> and <code>end</code> . Common annotation refers to attributes that all features in the <code>range</code> have in common. If any of the attributes of any character within the range differ, it is omitted from the result.

Events

The following table lists events that are triggered on the `ui.TextEditor` instance but also on the `joint.ui.TextEditor` class (which is useful if you use the recommended way of creating a text editor via the `joint.ui.TextEditor.edit()` function):

text:change	Triggered when the text inside the text editor changes. This is the most important event and the one you want to always react on. The callback has the following signature: <code>function(newText)</code> and is therefore called with the new text as a parameter. You are assumed to set the new text to the property on your JointJS element/link that is used for storing the associated text.

select:change	Triggered when the text inside the text editor is being selected (or if the user double-clicked to select a word or triple-clicked to select the entire text). The callback has the following signature: <code>function(selectionStart, selectionEnd)</code> . For example, you can see the selected text like this: <pre>ed.on('select:change', function(selectionStart, selectionEnd) { var text = ed.getTextContent().substring(selectionStart, selectionEnd); });</pre>
select:changed	Triggered when the user is finished selecting text inside the text editor. The callback has the following signature: <code>function(selectionStart, selectionEnd)</code> . For example, you can see the selected text like this: <pre>ed.on('select:changed', function(selectionStart, selectionEnd) { var text = ed.getTextContent().substring(selectionStart, selectionEnd); });</pre>
caret:change	Triggered when the caret has changed position. The position is passed to the callback as the only argument.
caret:out-of-range	Triggered when the caret is outside the visible text area. This is a very special event that you don't usually deal with. The only situation this event can occur is when <code>ui.TextEditor</code> is used on a text rendered along a path (see <code>Vectorizer#text(str, { textPath: '' })</code>). In this case, if the user moves his cursor outside the visible text area, <code>caret:out-of-range</code> event is triggered so that the programmer has chance to react (if he wants to because these situations are handled seamlessly in <code>ui.TextEditor</code> by hiding the caret).
open	Triggered when a new text editor instance is created. <pre>TextEditor.on('open', function(textNode, cellView) { // Editing of an SVGTextElement `textNode` (from the `cellView`) started. });</pre>
close	Triggered when the text editor instance is destroyed. <pre>TextEditor.on('close', function(textNode, cellView) { // Editing of an SVGTextElement `textNode` (from the</pre>

```
`cellView`) finished.  
});
```

JointJS+ Toolbar

ui.Toolbar

With the Toolbar plugin you can easily enrich your application with various tools. Either use built-in controls to manage other JointJS+ plugins (e.g. `zoom` for `PaperScroller`, `undo / redo` for `CommandManager`) or custom tools of your desire - just choose a type of UI primitive ('slider', 'input', 'button' etc.) and attach an action. Toolbar is a container for this tools - called 'widgets'.

Create a Toolbar

```
var toolbar = new joint.ui.Toolbar({
  tools: [
    { type: 'checkbox' },
    { type: 'range', name: 'slider', min: 0, max: 10, step: 1 },
    { type: 'separator' },
    { type: 'toggle', name: 'toggle', label: '' },
    'separator', // also possible, use defaults
    { type: 'inputText' },
    { type: 'button', name: 'ok', text: 'Ok' },
    { type: 'button', name: 'cancel', text: 'Cancel' },
    { type: 'separator' }
  ]
});

$('body').append(toolbar.render().el);
```

Toolbar configuration

tools	<i>Array<object> Array<string></i>	An array of tools defined as a plain JavaScript object. For more information visit Toolbar configuration section.
references	<i>object</i>	<code>`key => value`</code> storage. Some widgets require additional references for their functionality. Whenever you use such a widget (e.g. <code>`undo/redo`</code> requires <code>`joint.dia.CommandManager`</code> , <code>`zoom`</code> requires <code>`joint.ui.PaperScroller`</code>) you need to pass the required reference into the toolbar as well.
autoToggle	<i>boolean</i>	When set to <code>true</code> , the widgets switch their state (enabled/disabled) automatically based on the their references.

groups	<i>object</i>	<p>Split the widgets defined in <code>tools`</code> option into groups. Groups could be aligned and ordered in the context of the Toolbar.</p> <table> <tr> <td><b>groups.align</b></td><td><i>string</i></td><td>Can be <code>`left`</code> or <code>`right`</code>.</td></tr> <tr> <td>groups.index</td><td><i>number</i></td><td>Set the order of the group.</td></tr> </table>	groups.align	<i>string</i>	Can be <code>`left`</code> or <code>`right`</code> .	groups.index	<i>number</i>	Set the order of the group.
groups.align	<i>string</i>	Can be <code>`left`</code> or <code>`right`</code> .						
groups.index	<i>number</i>	Set the order of the group.						
widgetNamespace	<i>object</i>	A namespace with all available widgets. It points to <code>joint.ui.widgets</code> by default.						

Tool configuration

type	<i>string</i>	Set the type of the widget. Available widgets are listed in the Toolbar widgets section.
name	<i>string</i>	<code>`name`</code> is used to differentiate multiple widgets with the same <code>`type`</code> and has to be unique in the context of the toolbar. Note that the toolbar uses the <code>`name`</code> as a prefix for events triggered on widgets. More information can be found in the Events section.
group	<i>string</i>	Name of the group where a widget belongs to.
attrs	<i>object</i>	JointJS style attribute definition. Additional DOM element attributes can be defined here.
theme	<i>string</i>	Allow to override the global <code>`theme`</code> .

Toolbar API

getWidgetByName(name)	<i><joint.ui.Widget></i>	Return an instance of <code>`joint.ui.Widget`</code> with the matching <code>`name`</code> .
getWidgets()	<i>Array<joint.ui.Widget></i>	Return an array of <code>`joint.ui.Widget`</code> instances included in the toolbar.

Toolbar widgets

checkbox		<table> <tr> <th>name</th><th>type</th><th>default</th></tr> <tr> <td>value</td><td>boolean</td><td></td></tr> <tr> <td>label</td><td>string</td><td></td></tr> </table>	name	type	default	value	boolean		label	string	
name	type	default									
value	boolean										
label	string										

		Events: input , change , pointerdown , pointerclick , pointerup , click										
toggle		<table><tr><td>value</td><td>boolean</td></tr><tr><td>label</td><td>string</td></tr></table> Events: input , change , pointerdown , pointerclick , pointerup , click	value	boolean	label	string						
value	boolean											
label	string											
separator		Vertical line by default. <table><tr><td>width</td><td>number</td></tr></table>	width	number								
width	number											
label		<table><tr><td>text</td><td>string</td></tr></table>	text	string								
text	string											
range		<table><tr><td>value</td><td>boolean</td></tr><tr><td>unit</td><td>string</td></tr><tr><td>min</td><td>number</td></tr><tr><td>max</td><td>number</td></tr><tr><td>step</td><td>number</td></tr></table> Events: change	value	boolean	unit	string	min	number	max	number	step	number
value	boolean											
unit	string											
min	number											
max	number											
step	number											
selectBox		Configuration is described in ui.SelectBox section.										
button		<table><tr><td>text</td><td>string</td></tr></table> Events: pointerclick , pointerdown , pointerup , click	text	string								
text	string											
inputText		<table><tr><td>label</td><td>string</td></tr><tr><td>value</td><td>string</td></tr></table>	label	string	value	string						
label	string											
value	string											

		Events: input , change , pointerclick , pointerdown , pointerup , pointerfocusout , pointerfocusin											
inputNumber	<table><tr><td>label</td><td>string</td></tr><tr><td>value</td><td>number</td></tr><tr><td>min</td><td>number</td></tr><tr><td>max</td><td>number</td></tr></table> Events: input , change , pointerclick , pointerdown , pointerup , pointerfocusout , pointerfocusin	label	string	value	number	min	number	max	number				
label	string												
value	number												
min	number												
max	number												
textarea	<table><tr><td>label</td><td>string</td></tr><tr><td>value</td><td>string</td></tr></table> Events: input , change , pointerclick , pointerdown , pointerup , pointerfocusout , pointerfocusin	label	string	value	string								
label	string												
value	string												
selectButtonGroup	Configuration is described in the ui.SelectButtonGroup section.												
zoomIn zoomOut	<table><tr><td>min</td><td>number</td><td>0.2</td></tr><tr><td>max</td><td>number</td><td>5</td></tr><tr><td>step</td><td>number</td><td>0.2</td></tr><tr><td>autoToggle</td><td>boolean</td><td>false</td></tr></table> Required references: paperScroller ui.PaperScroller instance	min	number	0.2	max	number	5	step	number	0.2	autoToggle	boolean	false
min	number	0.2											
max	number	5											
step	number	0.2											
autoToggle	boolean	false											
zoomToFit	<table><tr><td>min</td><td>number</td><td>0.2</td></tr></table>	min	number	0.2									
min	number	0.2											

		<table><tr><td>max</td><td>number</td><td>5</td></tr><tr><td>step</td><td>number</td><td>0.2</td></tr><tr><td>padding</td><td>number</td><td>20</td></tr><tr><td>useModelGeometry</td><td>boolean</td><td>false</td></tr></table> Required references: paperScroller ui.PaperScroller instance	max	number	5	step	number	0.2	padding	number	20	useModelGeometry	boolean	false			
max	number	5															
step	number	0.2															
padding	number	20															
useModelGeometry	boolean	false															
zoomSlider		<table><tr><td>min</td><td>number</td><td>20</td></tr><tr><td>max</td><td>number</td><td>500</td></tr><tr><td>step</td><td>number</td><td>20</td></tr><tr><td>value</td><td>number</td><td>100</td></tr><tr><td>unit</td><td>string</td><td>'%</td></tr></table> Required references: paperScroller ui.PaperScroller instance	min	number	20	max	number	500	step	number	20	value	number	100	unit	string	'%
min	number	20															
max	number	500															
step	number	20															
value	number	100															
unit	string	'%															
undo redo		<table><tr><td>autoToggle</td><td>boolean</td><td>false</td></tr></table> Required references: commandManager dia.CommandManager instance	autoToggle	boolean	false												
autoToggle	boolean	false															
fullscreen		Presents an element (<i>target</i>) in full-screen mode. The fullscreen feature is available only if the target is not displayed within an iframe (the button is hidden otherwise).															

		<table> <tr> <td>target</td><td>HTMLElement string selector</td><td>window.top.document.body</td></tr> </table>	target	HTMLElement string selector	window.top.document.body
target	HTMLElement string selector	window.top.document.body			
colorPicker		<table> <tr> <td>value</td><td>string</td><td>#FFFFFF</td></tr> </table> Events: <u>change</u>, <u>input</u>	value	string	#FFFFFF
value	string	#FFFFFF			

Every widget has the following API:

enable()	Enable the widget. Example: <pre>toolbar.getWidgetByName('undo').enable();</pre>
disable()	Disable the widget (make the widget not to respond to user actions)
isDisabled()	Return true if the widget is disabled. Return false otherwise.

Events

The toolbar behaves as a proxy for widgets events. An event triggered on a single widget could be also caught on the Toolbar instance. Event names are prefixed with the widget's `name` followed by a colon `:` to recognize from which tool (widget) is the event coming from. In order to listen to a widget, the widget must have a `name` set.

```
var toolbar = new joint.ui.Toolbar({
  tools: [
    { type: 'toggle', name: 'toggle' },
    { type: 'button', name: 'ok', text: 'Ok' },
    { type: 'button', name: 'cancel', text: 'Cancel' },
  ]
});

toolbar.on('toggle:change', function(value, event) {
  console.log(value, event);
});

toolbar.on('ok:pointerclick', function(event) {
  console.log('ok clicked');
});

toolbar.on('cancel:pointerclick', function(event) {
  console.log('cancel clicked');
});
```

```
$('#body').append(toolbar.render().el);
```

Demo

```
var paperScroller = new joint.ui.PaperScroller({
  paper: paper,
  autoResizePaper: true
});

var commandManager = new joint.dia.CommandManager({
  graph: graph
});

var toolbar = new joint.ui.Toolbar({
  // initialize tools with default settings
  tools: ['zoomIn', 'zoomOut', 'zoomToFit', 'zoomSlider', 'undo', 'redo'],
  references: {
    paperScroller: paperScroller,
    commandManager: commandManager
  }
});
```

JointJS+ Tooltip

ui.Tooltip

Use tooltips to display information messages anywhere in the UI.

Installation

Include `joint.ui.tooltip.css` and `joint.ui.tooltip.js` files to your HTML:

```
<link rel="stylesheet" type="text/css" href="joint.ui.tooltip.css">
<script src="joint.ui.tooltip.js"></script>
```

Creating a Tooltip

A tooltip can be created with the `joint.ui.Tooltip` constructor. One instance of `joint.ui.Tooltip` manages multiple tooltips. Tooltips has configuration on html elements, as data attributes. Configuration passed in constructor became global settings for the particular tooltip. Setting defined as a data attribute won't override the global setting defined in the constructor. Newly added elements with a known definition are initialized automatically, right after append.

```
// js part - initialize tooltip
new joint.ui.Tooltip({
  target: '[data-tooltip]',
  direction: 'auto',
  padding: 10
});
```

```
<!-- html part - tooltip configuration -->
<label data-tooltip="Top directed tooltip" data-tooltip-position="top">Hover to see
top tooltip</label>
<label data-tooltip="Left directed tooltip" data-tooltip-position="left">Hover to see
left tooltip</label>
<label data-tooltip="Right directed tooltip" data-tooltip-position="right">Hover to
see right tooltip</label>
<label data-tooltip="Bottom directed tooltip" data-tooltip-position="bottom">Hover to
see bottom tooltip</label>
```

Result

In this demo all the elements with the data attribute `tooltip` will display tooltip when hovered.

Configuration

The `joint.ui.Tooltip` constructor takes the following parameters:

target	<code>string Element jQuery</code>	A CSS selector, DOM element or jQuery object that is the target for the tooltip. When the user hovers over this element with his mouse cursor, the tooltip appears.
rootTarget	<code>string Element jQuery</code>	A CSS selector, DOM element or jQuery object that is the root target for the tooltip. When using the <code>`rootTarget`</code> , the event handlers that trigger the tooltip display will be delegated (as opposed to directly-bound). This is useful when the <code>`target`</code> is a selector pointing to elements that might get removed from the DOM and then put back again. The tooltip tries to find the selected elements specified by the selector in the <code>`target`</code> option when it is initialized (during instantiation) and then uses these element as the target for the tooltip. If later on such elements are removed from the DOM the tooltip loses a reference to the target. Therefore, we can prevent this by using <code>`rootTarget`</code> or selecting the container element and <code>`target`</code> selector for selecting the real target of the tooltip at the time of the <code>`"mouseover"``</code> event.
container	<code>string Element jQuery</code>	A CSS selector, DOM element or jQuery object that is the container element, which the tooltip is appended to. Later when the tooltip is shown it respects the boundary of this container and changes its direction if it would overlap the boundary.
content	<code>string Element jQuery function(element, tooltip)</code>	The content of the tooltip. It can be an arbitrary string/HTML or a function. The <code>element</code> parameter is undefined when the user calls <code>show</code> or <code>toggle</code> API methods with the point as a parameter. If the function returns null or undefined the content is taken from <code>data-tooltip</code> (or arbitrary place using <code>dataAttributePrefix</code> parameter). If the function returns <code>false</code> the tooltip is not displayed.
padding	<code>string Element jQuery function(element)</code>	The distance between <code>`target`</code> element and the tooltip.
position	<code>string function(element)</code>	The position of the element relative to the tooltip. <code>`left`</code> means the element is on the left of the shown tooltip. Can be either <code>`top`</code> , <code>`left`</code> , <code>`bottom`</code> , <code>`right`</code>
direction	<code>string function(element)</code>	The direction of the tooltip arrow. Can be either <code>`top`</code> , <code>`right`</code> , <code>`bottom`</code> , <code>`left`</code> or <code>`auto`</code> . <code>`auto`</code> sets the arrow accordingly to 'position' property. Arrows are disabled if the <code>`direction`</code> is a falsy value or <code>`off`</code>

positionSelector	<i>string</i> <i>Element</i> <i>jQuery</i> <i>function(element)</i>	A CSS selector or DOM element or jQuery object or a function used to compute the position of the edge of the tooltip. It defines bounding box which the tooltip mustn't to get across.						
trigger	<i>string</i>	Event which makes the tooltip show up. Can be either <code>`hover`</code> , <code>`focus`</code> or <code>`click`</code> . This option cannot be set as a data attribute on the DOM element.						
minResizedWidth	<i>number</i>	A minimal width of the tooltip. The tooltip width can be resized down to the <code>`minResizedWidth`</code> . If the available space is smaller than the <code>`minResizedWidth`</code> , the direction of the tooltip is changed to its opposite direction (left tooltip is swapped to right, top to bottom and vice versa). Set <code>`minResizedWidth`</code> to a falsy value if no automatic resizing or direction swapping should take a place. It defaults to <code>`100`</code> .						
hideTrigger	<i>string</i>	Event which makes the tooltip disappear. Can be any of the DOM event. Separate multiple events with <code>` `</code>. <pre><label data-tooltip="tooltip text" data-tooltip-hide-trigger="mouseout mouseover">HIDE ON MOUSEOUT</label></pre> or <pre>new joint.ui.Tooltip({ rootTarget: '.click-tooltip', target: '[data-tooltip]', trigger: 'click', hideTrigger: 'mouseout' });</pre>						
viewport	<i>object</i> <i>function(element)</i>	viewport defines the boundary for the tooltip. It guarantees that the tooltip's content will be rendered inside the <code>`viewport`</code> bounding box. <table> <tr> <td>selector</td><td><i>string</i> <i>jQuery</i> <i>Element</i></td><td>CSS selector, Element or jQuery object defining the <code>`viewport`</code> DOM object. The element triggering tooltip should be rendered inside this element.</td></tr> <tr> <td>padding</td><td><i>number</i></td><td>Minimal distance from the <code>`viewport`</code> boundaries.</td></tr> </table>	selector	<i>string</i> <i>jQuery</i> <i>Element</i>	CSS selector, Element or jQuery object defining the <code>`viewport`</code> DOM object. The element triggering tooltip should be rendered inside this element.	padding	<i>number</i>	Minimal distance from the <code>`viewport`</code> boundaries.
selector	<i>string</i> <i>jQuery</i> <i>Element</i>	CSS selector, Element or jQuery object defining the <code>`viewport`</code> DOM object. The element triggering tooltip should be rendered inside this element.						
padding	<i>number</i>	Minimal distance from the <code>`viewport`</code> boundaries.						

dataAttributePrefix	<i>string function(element)</i>	Defines prefix for HTML data attributes with the tooltip configuration. <pre>new joint.ui.Tooltip({ target: '[custom-def]', dataAttributePrefix: 'custom-def' });</pre> matches with <pre><label data-custom-def="Tooltip content" data- custom-def-position-selector=".box2" data- custom-def-position="bottom">BOTTOM</label></pre>												
animation	<i>boolean object</i>	When set to true a CSS fade-in animation is applied to the tooltip. Passing an object with any of the following properties allows you to make further adjustments to the animation. <table> <tr> <th>Property</th><th>Default</th><th>Description</th></tr> <tr> <td>duration</td><td>'500ms'</td><td>CSS animation-duration</td></tr> <tr> <td>delay</td><td>'400ms'</td><td>CSS animation-delay</td></tr> <tr> <td>timingFunction</td><td>'ease'</td><td>CSS animation-timing-function</td></tr> </table>	Property	Default	Description	duration	'500ms'	CSS animation-duration	delay	'400ms'	CSS animation-delay	timingFunction	'ease'	CSS animation-timing-function
Property	Default	Description												
duration	'500ms'	CSS animation-duration												
delay	'400ms'	CSS animation-delay												
timingFunction	'ease'	CSS animation-timing-function												

Tooltip API

This API allows the user to manually control behaviour of the tooltip. When using the `show` or `toggle` method with the `point` option for closing the tooltip the user needs to call the `hide` or `toggle` method manually (there is no element to bind events).

show(options)	Shows current instance of the tooltip. The method accepts an optional <code>options</code> parameter which may contain a <code>target</code> (CSS selector or DOM Element) and/or coordinates (<code>x</code> and <code>y</code>) at which to create the tooltip.
hide()	Hides current instance of the tooltip.
toggle(options)	Toggles current instance of the tooltip. The method accepts an optional <code>options</code> parameter which may contain a <code>target</code> (CSS selector or DOM Element) and/or coordinates (<code>x</code> and <code>y</code>) at which to create the tooltip.

isVisible()	Returns <code>true</code> if the tooltip is hidden. Otherwise, returns <code>false</code>

Tooltip events

close	Triggered when the tooltip gets closed.

Tooltips in Halo

It's useful to show the user a hint on the [Halo](#) icon tools. Fortunately, it's very easy. For instance:

```
paper.on('cell:pointerup', function(cellView) {
  // We don't want a Halo for links.
  if (cellView.model instanceof joint.dia.Link) return;
  var halo = new joint.ui.Halo({ graph: graph, paper: paper, cellView: cellView });

  halo.changeHandle('remove', {
    attrs: {
      '.handle': {
        'data-tooltip-class-name': 'small',
        'data-tooltip': 'Click to remove the object',
        'data-tooltip-position': 'right'
      }
    }
  });
});

// run this once
new joint.ui.Tooltip({
  rootTarget: document.body,
  target: [data-tooltip],
  padding: 15
});
```

Notice we use the `'small'` `className` in our example. JointJS+ tooltips come with two build-in sizes for tooltips: small and normal. In order to enable the small version use `'small'` as a `className`.

JointJS+ TreeLayoutView

ui.TreeLayoutView

`ui.TreeLayoutView` is a view on a tree-like graph that implements common tree manipulation interface like detaching and attaching branches, disconnecting nodes from the tree to create floating nodes and more.

Installation

Include `joint.ui.treeLayoutView.js` and its dependency `joint.layout.treeLayout.js` to your HTML:

```
<script src="joint.layout.treeLayout.js"></script>
<script src="joint.ui.treeLayoutView.js"></script>
```

Usage

First you need to create an object of `layout.TreeLayout` type. `ui.TreeLayoutView` is dependent on this and expects this object in its `model` parameter:

```
var graphLayout = new joint.layout.TreeLayout({
  graph: graph,
  parentGap: 50,
  siblingGap: 50
});

var treeLayoutView = new joint.ui.TreeLayoutView({
  paper: paper,
  model: graphLayout
});

graphLayout.layout();
```

Source Code

Configuration

The following table lists options that you can pass to the `ui.TreeLayoutView` constructor function:

model	Instance of <code>joint.layout.TreeLayout</code> . See layout.TreeLayout plugin for more details.
paper	JointJS paper object.

useModelGeometry	If set to <code>true</code>, the cells position and dimensions will be used as a basis for the <code>TreeLayoutView</code> wrapping rectangle position. By default, this is set to <code>false</code> which causes the <code>TreeLayoutView</code> wrapping rectangle position be based on the bounding box of the selected element views. Sometimes though, your shapes can have certain SVG sub elements that “stick out” of the view and you don't want these sub elements to affect the <code>TreeLayoutView</code> wrapping rectangle position. In this case, set the <code>useModelGeometry</code> to <code>true</code>.										
clone(element)	A function, that is applied on the <code>element</code> to create a clone for dragging. It defaults to <code>return element.clone();</code>.										
canInteract(element)	A predicate, that returns <code>true</code> if the <code>element</code>, which the user interacts with, can be actually dragged. It defaults to <code>return true;</code>.										
validateConnection(element, candidateParent, treeLayoutView, details)	A predicate, that returns <code>true</code> if the <code>element</code>, which the user interacts with, can be reconnected to the <code>candidateParent</code>. It defaults to <code>return true;</code>. The <code>details</code> object contains additional information about the candidate connection. <table> <tr> <th>Property</th><th>Description</th></tr> <tr> <td>level</td><td>The distance of the <code>element</code> from the tree root.</td></tr> <tr> <td>direction</td><td>The direction of the connection (between the <code>candidateParent</code> and <code>element</code>).</td></tr> <tr> <td>siblingRank</td><td>The index of the element among its siblings.</td></tr> <tr> <td>siblings</td><td>An array of sibling elements (children of <code>candidateParent</code> in a particular <code>direction</code>).</td></tr> </table>	Property	Description	level	The distance of the <code>element</code> from the tree root.	direction	The direction of the connection (between the <code>candidateParent</code> and <code>element</code>).	siblingRank	The index of the element among its siblings.	siblings	An array of sibling elements (children of <code>candidateParent</code> in a particular <code>direction</code>).
Property	Description										
level	The distance of the <code>element</code> from the tree root.										
direction	The direction of the connection (between the <code>candidateParent</code> and <code>element</code>).										
siblingRank	The index of the element among its siblings.										
siblings	An array of sibling elements (children of <code>candidateParent</code> in a particular <code>direction</code>).										
validatePosition(element, x, y, treeLayoutView)	A predicate, that returns <code>true</code> if the <code>element</code>, which the user interacts with, can be placed (disconnected or translated) at the position <code>x</code>, <code>y</code>. It defaults to <code>return true;</code>.										
reconnectElements(elements, parent, siblingRank, direction, treeLayoutView)	A function to override the default connection behaviour (this includes connecting sources and reconnecting children).										
translateElements(element, x, y, treeLayoutView)	A function to override the default translating behaviour (this includes translating sources and disconnecting children).										

previewAttrs	An object with SVG attributes to be applied onto the preview element. The preview consists of 3 components - <code>child</code> , <code>parent</code> and <code>link</code> .e.g. <pre>{ child: { opacity: 0.5 }, link: { opacity: 0.5 }, parent: { rx: 20, ry: 20 } }</pre>
paperConstructor	The type of the paper. Defaults to <code>joint.dia.Paper</code> . This is only useful if you use a different paper type for rendering your graphics. In case you inherit from <code>joint.dia.Paper</code> to implement some specific rendering, you can just pass your constructor function to this parameter and use the <code>paperOptions</code> parameter to pass additional options to your special paper.
paperOptions	Options object that will be passed to the internal paper used for rendering graphics.
layoutFunction(treeLayoutView)	Provides a custom callback function that runs instead of the default layout function to create more complex (e.g multi-step layout - first layout the embedded children using a grid layout and then invoke the tree layout for parents only) layouts. The first parameter of the callback is the current <code>ui.TreeLayoutView</code> instance.

API

startDragging(elementOrElements)	Initiates dragging of a single or multiple elements (<code>dia.Element</code> or an array of <code>dia.Element</code>). Useful when combined with the Selection plugin to drag & drop multiple nodes at once. This is a high-level method, which will start listening to <code>pointermove</code> and <code>pointerup</code> events and executes low-level <code>dragstart</code> , <code>drag</code> and <code>dragend</code> methods accordingly.
cancelDrag()	Stop the current drag operation.
canDrop()	Has the current drag operation valid drop target? i.e. an element can be reconnected or translated.
dragstart(elements, x, y)	Start a new drag operation with <code>elements</code> set as its source.
drag(elements, x, y)	Drag the source elements and display preview of the drop location if any.

dragend(elements, x, y)	End the drag operation and reconnect/translate the source elements if there was a valid drop target.
enable()	Enable the view. The plugin starts listening to the paper and allows the user to reconnect the elements.
disable()	Disable the view. Prevent the user from reconnecting the elements.
isDisabled()	Return true if the view is disabled. Otherwise, return false . The plugin is enabled by default.

An example on how to integrate the [ui.Stencil](#) with TreeLayoutView.

```

stencil.on({
  'element:dragstart': (cloneView, evt) => {
    // An element that will be added to the graph
    const element = cloneView.model.clone();
    evt.data.elements = [element];
  },
  'element:drag': (cloneView, evt, cloneArea, validDropTarget) => {
    const { elements, dragStarted } = evt.data;
    const { x, y } = cloneArea.center();
    if (validDropTarget) {
      if (!dragStarted) {
        treeView.dragstart(elements, x, y);
        evt.data.dragStarted = true;
      }
      treeView.drag(elements, x, y);
      cloneView.el.style.display = 'none';
    } else {
      // Outside the valid drop target
      treeView.cancelDrag();
      evt.data.dragStarted = false;
      cloneView.el.style.display = 'block';
    }
  },
  'element:dragend': (cloneView, evt, cloneArea, validDropTarget) => {
    const { elements } = evt.data;
    const { x, y } = cloneArea.center();
    const canDrop = validDropTarget && treeView.canDrop();
    treeView.dragend(elements, x, y);
    // Show clone in case of drop animation
    cloneView.el.style.display = 'block';
    stencil.cancelDrag({ dropAnimation: !canDrop });
  }
});

```

}
});