Table of Contents

Joint API

This is the API reference to the open source JointJS core library. If you're looking for the Rappid 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 lighweight 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

A link anchor is a point on the paper that the link wants to reach as its endpoint. (In reality, the end element will probably be in the way - then, it will be the job of the connection point function to determine the actual location of the route endpoint taking the respective end element 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).

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

dy	*number*	Offset the anchor by `dy`. Default is `0`.
	string	Percentage strings (e.g. `'40%'`) 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	*boolean*	Should the anchor bbox rotate with the end view? Default is `false`, meaning that the unrotated bbox is used.
dx	*number*	Offset the anchor by `dx`. Default is `0`.
	string	Percentage strings (e.g. `'40%'`) are also accepted.
dy	*number*	Offset the anchor by `dy`. Default is `0`.
	string	Percentage strings (e.g. `'40%'`) 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	*dia.ElementView*	The ElementView to which we are connecting. The Element model can be accessed as `endView.model`; this may be useful for writing conditional logic based on element attributes.
endMagnet	*SVGElement*	The SVGElement in our page that contains the magnet (element/subelement/port) to which we are connecting.
anchorReference	*g.Point*	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...
	SVGElement	...if the anchor in question does not exist (yet), it is that link end's magnet.
args	*object*	An object with additional optional arguments passed to the anchor method by the user when it was called (the `args` 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	*boolean*	Should the anchor bbox rotate with the end view? Default is `false`, meaning that the unrotated bbox is used.
dx	*number*	Offset the anchor by `dx`. Default is `0`.
	string	Percentage strings (e.g. `'40%'`) are also accepted.
dy	*number*	Offset the anchor by `dy`. Default is `0`.

	string	Percentage strings (e.g. `'40%'`) 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	*boolean*	Should the anchor bbox rotate with the end view? Default is `false`, meaning that the unrotated bbox is used.
padding	*number*	Offset the anchor by `padding` away from view bbox. Default is `0`.

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	*number*	Limit the area inside the view bbox available for placing the anchor by `padding`. Default is `0`.

Example:

```
link.source(model, {
    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	*boolean*	Should the anchor bbox rotate with the end view? Default is `false`, meaning that the unrotated bbox is used.
dx	*number*	Offset the anchor by `dx`. Default is `0`.
	string	Percentage strings (e.g. `'40%'`) are also accepted.
dy	*number*	Offset the anchor by `dy`. Default is `0`.

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

dy	*number*	Offset the anchor by `dy`. Default is `0`.
	string	Percentage strings (e.g. `'40%'`) 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	*number*	Offset the connection point from the anchor by the specified distance along the end link path segment. Default is `0`.

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'` connection point function is the simplest connection point function. It places the connection point at the link end's anchor point (determined either by the `anchor` function or by the `defaultAnchor` paper option).

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. It accepts one additional argument, which can be passed within the `connectionPoint.args` property:

stroke	*boolean*	Should the stroke width be included when calculating the connection point? Default is `false`.

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.) It accepts six additional arguments, which can be passed within the `connectionPoint.args` property:

insideout	*boolean*	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	*boolean*	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 `connectionPoint.args.sticky`.
sticky	*boolean*	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	*number*	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	*string*	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.)
stroke	*boolean*	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	*g.Line*	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	*dia.ElementView*	The ElementView to which we are connecting. The Element model can be accessed as `endView.model`; this may be useful for writing conditional logic based on element attributes.
endMagnet	*SVGElement*	The SVGElement in our page that contains the magnet (element/subelement/port) to which we are connecting.

args	*object*	An object with additional optional arguments passed to the connection point method by the user when it was called (the `args` 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. It accepts one additional argument, which can be passed within the `connectionPoint.args` property:

stroke	*boolean*	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 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	*object*	An end definition; the output of the appropriate end function (`link.source()` or `link.target()`). An object containing at least an `id` 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	*dia.ElementView*	The ElementView to which we are connecting. The Element model can be accessed as `endView.model`; this may be useful for writing conditional logic based on element attributes.
endMagnet	*SVGElement*	The SVGElement in our page that contains the magnet (element/subelement/port) to which we are connecting.
coords	*g.Point*	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 four built-in connectors in JointJS:

- `'jumpover'` - connector with bridges over link intersections
- `'normal'` - default simple connector
- `'rounded'` - connector with rounded edges
- `'smooth'` - connector interpolated as a bezier curve

Example:

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

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

```
paper.options.defaultConnector = {
    name: 'rounded',
    args: {
        radius: 20
    }
}
```

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

raw	*boolean*	Should the router return the connection path as a `g.Path` rather than as a string? Default is `false`.

Example:

```
link.connector('normal', {
    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	*g.Point*	The source connection point.
targetPoint	*g.Point*	The target connection point.

routePoints	*Array<g.Point>*	The points of the route, as returned by the router in use.
args	*object*	An object with additional optional arguments passed to the connector method by the user when it was called (the `args` 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	*number*	The size of the jump (the diameter of the arc, or the length of the empty spot on the line). Defaults to `5`.
jump	*string*	The style of the jump. Either `'arc'` (using an Arcto SVG path command), `'cubic'` (using a Curveto path command as a normalized approximation to Arcto), or `'gap'` (leaving a blank space). Defaults to `'arc'`.

Example:

```
link.connector('jumpover', {
    type: 'gap'
```

```
});
```

connectors.normal

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:

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

connectors.rounded

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	*boolean*	The curve radius of the rounded corners. Default is `10`.

Example:

```
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)
```

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 defines new so-called "special" attributes and allows programmers to define their own. 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', { atConnectionRatioKeepGradient: .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

Set the `'d'` attribute based on the current link connection (the result of the link`s connector). It's valid only for `SVGPathElement` within the LinkView context.

```
link.attr('pathSelector', { connection: true, stroke: 'red', fill: 'none' });
```

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 a gradient definition and must have the form defined by the defineGradient() paper method.

```
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' }
    ]
});
```

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 by the filterGradient() paper method.

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

dia.attributes.magnet

When set to `true`, the subelement can become a source/target of a link during link reconnection. Useful for so called 'ports'.

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.ref

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 the 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 the 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 element is automatically rotated based on the path direction. It's defined as an object with `type` property and any other visual attributes. The valid values for `type` are `'path'`, `'circle'`, `'ellipse'`, `'rect'`, `'polyline'` and `'polygon'`.

```
link.attr('.connection/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.

Similar to the `text` attribute, except the provided string is automatically wrapped to fit within the reference bounding box.

Expects an object with a `text` property and optional `width` and `height`, which can adjust the final size of the wrapped text. Negative values decrease the dimension (e.g. to add padding around the wrapped text); positive values increase the dimension. Percentage values are accepted as well.

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. See util.breakText for more info.

```
textWrap: {
  text: 'lorem ipsum dolor sit amet consectetur adipiscing elit',
  width: -10, // reference width minus 10
  height: '50%', // half of the reference height
```

```
  ellipsis: true // could also be a custom string, e.g. '...!?'
}
```

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,
            refRx: .5,
            refRy: .5,
            refCx: .5,
            refCy: .5
        }
    }
});

// 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. 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>'
```

JSON Markup

JSON markup is defined recursively as an array of `JSONElement`, 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"`.
- `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'
    }, {
        tagName: 'circle',
        selector: 'circle2',
        groupSelector: 'circles'
    }]
}]
```

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.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.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.highlight

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

Highlights the cell view.

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 highligher specified by `name`
  - **type** - the kind of highlighting being done (embedding, connecting, magnetAvailability, or elementAvailability)

dia.CellView.prototype.unhighlight

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

Removes the highlighting being done to the cell.

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.

dia.Element

The model for diagram elements. It inherits from joint.dia.Cell with a few additional properties and methods specific to elements. These properties can be put into three groups:

Geometry

Coordinates of an element are stored in the `position` property that is an object with `x` and `y` keys. `position` can be accessed or set directly using the regular Backbone `set()`/`get()` methods or through the translate method.

Rotation angle is stored in the `angle` property. This angle is in degrees and the rotation origin is always considered to be the center of the element. `angle` can be also accessed or set directly using the regular Backbone `set()`/`get()` methods or through the rotate method.

Size of an element is stored in the `size` property that is an object with `width` and `height` keys. Again, `size` can be accessed or set directly using the regular Backbone `set()`/`get()` methods or through the resize method.

Presentation

Another important property is `attrs` which is an object with keys representing **selectors** that match subelements and values which are SVG attributes that will be set on the subelements. One can find a list of **SVG attributes** and their descriptions e.g. on MDN.

It is important to note that each `joint.dia.Element` defines an SVG markup which is then used by `joint.dia.ElementView` to render the element to the paper. For instance, the `joint.shapes.basic.Rect` element (that inherits from `joint.dia.Element`) defines its markup as follows:

```
<g class="rotatable"><g class="scalable"><rect/></g><text/></g>
```

Therefore, in order to set a red fill color for the rectangle subelement, the `attrs` object should contain:

```
rect: { fill: 'red' }
```

Again, it is not recommended to change the `attrs` object directly. Instead, use the attr method.

The `z` property specifies the stack order of the element in the SVG DOM. An element with a higher `z` level is in front of an element with a lower `z` level. (This also stands for links which have the exact same property.)

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` is a list of cell `id`'s that are embedded inside the element. `parent` is an `id` of the parent element of the embedded one. When a parent element is translated, all its children get translated too.

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 transiton 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 then point not 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 Port configuration section.

Port API on `joint.dia.Element`

- `hasPort` / `hasPorts`
- `addPort` / `addPorts`
- `removePort` / `removePorts`
- `getPort` / `getPorts`
- `portProp`
- `getPortPositions`

Port configuration

```
// Single port definition
var 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: 'right',
            args: { y: 6 } // extra arguments for the label layout function, see
`layout.PortLabel` section
        },
        markup: '<text class="label-text" fill="blue"/>'
    },
    attrs: { text: { text: 'port1' } },
    markup: '<rect width="16" height="16" x="-8" strokegit ="red" fill="gray"/>'
};

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

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

id	*string*	It is automatically generated if no `id` provided. IDs must be unique in the context of a single shape - two ports with the same port id are therefore not allowed (`Element: found id duplicities in ports.` error is thrown).
group	*string*	Group name, more info in [groups](#groupssection) section.
args	*object*	Arguments for the port layout function. Available properties depends on the type of layout. More information could be found in [`layout.Port`](#layout.Port).
attrs	*object*	JointJS style attribute definition. The same notation as the `attrs` property on [`Element`](#joint.dia.Element.presentation).

markup	*string*	Custom port markup. Multiple roots are not allowed. Valid notation would be: ``` `  ` ``` It defaults to ``.
label	*object*	Port label layout configuration. E.g. label position, label markup. More information about port label layouts could be found in [`layout.PortLabel`](#layout.PortLabel) section.
• **label.position**	*string* | *object*	Port label position configuration. It could be a `string` for setting the port layout type directly with default settings or an `object` where it's possible to set the layout type and options. ``` { position: 'left'} // or ... { position: { name: 'left', args: { dx: 10 } } } ```
◦ **label.position.name**	*string*	Stands for the layout type, match the layout method name defined in `joint.layout.PortLabel` namespace: `name:'left'` is implemented as `joint.layout.PortLabel.left`.
◦ **label.position.args**	*object*	Additional arguments for the layout method. Available properties depends on the layout type. More information could be found in [`layout.PortLabel`](#layout.PortLabel) section.
• **label.markup**	*string*	Custom port label markup. Multiple roots are not allowed. It defaults to ``.
z	*number* | *string*	Alternative to HTML `z-index`. `z` sets the position of a port in the list of DOM elements within an `ElementView`. Shapes most likely consist of 1 or more DOM elements, ``, `` etc. Ports are placed into the element `rotatable` group (if there is no `rotatable` group in the shape's markup, then the main group element `elementView.el` is used for the port container).

Ports with `z:'auto'` are located right after the last element in the `rotatable` 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) equals to `z`.

```
    <p>For instance an element with the following
markup
<blockquote><pre>`<g class="rotatable">
<g class="scalable"><rect/></g>
<text/>
```

` will be rendered like this:

```
<g model-id="element1">
    <g class="rotatable">
        <g class="port"></g>          <!-- z: 0 -->
        <g class="scalable"><rect/></g>
        <g class="port"></g>          <!-- z: 1 -->
        <text/>
        <g class="port"></g>          <!-- z: 2 -->
    </g>
</g>
```

Another example with simplified markup `` can look as follows:

```
<g model-id="element2">
    <g class="port"></g>          <!-- z: 0 -->
    <circle/>
    <g class="port"></g>          <!-- z: 1 -->
    <g class="port"></g>          <!-- another z: 1
-->
    <text/>
    <g class="port"></g>          <!-- z: 2 -->
    <g class="port"></g>          <!-- z: 'auto' -->
    <g class="port"></g>          <!-- z: 3 -->
    <g class="port"></g>          <!-- z: 'auto' -->
    <g class="port"></g>          <!-- z: 10 -->
</g>
```

```
</td>
```

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

Port groups configuration

`group` attribute comes to play when you're not happy with the default port alignment. It's also handy when you need to define multiple ports with similar properties. `group` defines defaults for ports belonging to the group. Any `group`

property can be overwritten by a port in this group except the type of layout - `position`. 'group' defines the layout and port 'args' are the only way how a port can affect it.

```
    // Define ports and port groups in element constructor.
    var groupA;
    var rect = new joint.shapes.basic.Rect({
         // ...
         ports: {
             groups: {
                 'group1': groupA,
                 // 'group2': ...,
                 // 'group3': ...,
             },
             items: []
         }
     });

     groupA = {
             position: {
                 name: 'string', // layout name
                 args: {}, // arguments for port layout function, properties depends on type of
layout
             },
             label: {
                 // ....
             },
             attrs: {},
             markup: '<rect width="10" height="10" stroke="red"/>'
         };
```

position	*string* | *object*	Port position configuration. Could be `string` to set port layout type directly with default settings or `object` where is possible to set layout type and options.
• **position.name**	*string*	Stands for the layout type, match the layout method name defined in `joint.layout.Port` namespace: `name:'left'` is implemented as `joint.layout.Port.left`.
• **position.args**	*object*	Arguments for the port layout function. Available properties depends on the type of layout. More information could be found in [`layout.Port`](#layout.Port).
attrs	*object*	JointJS style attribute definition. The same notation as the `attrs` property on [`Element`](#joint.dia.Element.presentation).
markup	*string*	Custom port markup. Multiple roots are not allowed. Valid notation would be: `` ` ``

		`` . It defaults to `` ``
label	*object*	Port label layout configuration. E.g. label position, label markup. More information about port label layouts could be found in [`layout.PortLabel`](#layout.PortLabel) section.
• **label.position**	*string* | *object*	Port label position configuration. It could be a `string` for setting the port layout type directly with default settings or an `object` where it's possible to set the layout type and options. `{ position: 'left'}` `// or ...` `{ position: { name: 'left', args: { dx: 10 } } }`
◦ **label.position.name**	*string*	Stands for the layout type, match the layout method name defined in `joint.layout.PortLabel` namespace: `name:'left'` is implemented as `joint.layout.PortLabel.left`.
◦ **label.position.args**	*object*	Additional arguments for the layout method. Available properties depends on the layout type. More information could be found in [`layout.PortLabel`](#layout.PortLabel) section.
• **label.markup**	*string*	Custom port label markup. Multiple roots are not allowed. It defaults to ``.

Custom markup

Both port and port label can have custom markup.

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

    rect.addPort({ markup: '<rect width="10" height="10" fill="blue"/>' })
    rect.addPort({ markup: '<rect width="15" height="15" fill="red"/>', label: { markup: '<text
fill="#000000"/>' }})
```

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: '<rect width="20" height="20" fill="black"/>',
    portLabelMarkup: '<text 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)`

Set SVG attributes (and JointJS special attributes) on subelements. `attr` 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 CSS selectors matching the subelements. The values are objects containing SVG attributes and their values. `attrs` object will be mixined with `attrs` property of

the `element` model. This is a convenient way of rewriting only some of the attributes of the subelements. For overwritting all attributes of all subelements, use `element.set(attrs)`.

```
element.attr({
    rect: { fill: 'blue' },
    text: { fill: 'white', 'font-size': 15 },
    '.myrect2': { fill: 'red' }
});
```

An alternative call using a string path and a value:

```
element.attr('text/font-size', 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 then a single instance.

dia.Element.prototype.embed

```
element.embed(cell)
```

Embed a cell (element or link) 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.

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])
```

Resize the element so that it fits all the embedded elements inside it. If `opt.deep` is `true`, the resizing will be done recursively for any embedded elements that contain embedded elements themselves. Set `opt.padding` if you want certain padding on all the parent elements.

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()
```

Returns an element's bounding box represented as a `g.rect` object (see geometry library).

```
if (element1.getBBox().intersect(element2.getBBox())) {
    // elements intersect
}
```

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

Set the element position to `x` and `y` coordinates. This is almost equivalent to `element.set('position', { x: x, y: y }, opt)`. However, this method provides some additional functionality. If you set `opt.parentRelative` flag to `true`, the `x` and `y` coordinates will be treated relatively to the parent element of this element. If `position()` is called without arguments, it returns the current position. If `position({ parentRealtive: true })` is called without `x` and `y` coordinates and with the `parentRelative` flag set to `true`, the method returns the current position of the element relative to its parent. Setting `opt.deep` to `true` will position not only the element but also all its descendants keeping the original distances from a child to the element origin.

```
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
```

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 } ] } ] })
```

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 then 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 streched 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 easilly 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 get higher `z` index than that of this element in a Breadth-first search (BFS) fashion. 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.

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 get higher `z` index than that of this element in a Breadth-first search (BFS) fashion. 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.

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.

dia.Element.prototype.toJSON

```
element.toJSON()
```

Return a copy of the element's attributes for JSON serialization. This can be used for persistance 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')).getBBox()
})
```

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)
```

Free up an embedded cell 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.getBBox

```
elementView.getBBox([opt])
```

Return a bounding box for an element view. If `opt.useModelGeometry` option is set to `true`, the resulting bounding box will be calculated based on the element model dimensions (so that SVG sub elements "sticking out" of the element will be excluded). The difference from dia.Element.prototype.getBBox is that in this case, the bounding box will be adjusted based on the `joint.dia.Paper` translate and scale.

dia.ElementView.prototype.getNodeBBox

```
elementView.getNodeBBox(magnet)
```

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

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).

dia.Graph.constructor

joint.dia.Graph is the model holding all the 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 an efficient storage for directed graphs but also offers useful algorithms for traversing the graphs.

Additionally, the graph accepts an option object in its constructor function that can contain the `cellNamespace` option. This option can be used to change the default behavior of JointJS which by default reads cell model definitions from the `joint.shapes` namespace. For example, if a cell is of type `'myshapes.MyElement'`, then the graph looks up `joint.shapes.myshapes.MyElement` model when deserializing a graph from the JSON format. If the graph is instantiated as e.g. `var graph = new joint.dia.Graph({}, { cellNamespace: myShapesNamespace })`, then the graph will read the model definition from the `myShapesNamespace.myshapes.MyElement` object instead. This is useful in situations where you don't want to - for any reason - use the `joint.shapes` namespace for defining your own custom shapes. This option is often used in combination with the `cellNamespaceView` 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)
```

Add a new cell to the graph. If `cell` is an array, all the cells in the array will be added to the graph.

```
var rect = new joint.shapes.basic.Rect({
    position: { x: 100, y: 100 },
    size: { width: 70, height: 30 },
    attrs: { text: { text: 'my rectangle' } }
})
var rect2 = rect.clone()
var link = new joint.dia.Link({ source: { id: rect.id }, target: { id: rect2.id }  });
var graph = new joint.dia.Graph
graph.addCell(rect).addCell(rect2).addCell(link)
```

dia.Graph.prototype.addCells

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

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

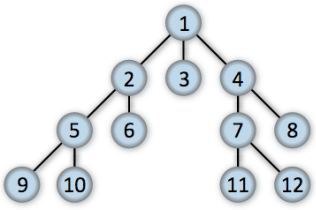
Add new cells to the graph. This is just a convenience method that wraps the addCell method.

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` explicitely 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. swaping 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 were `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 overto 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])
```

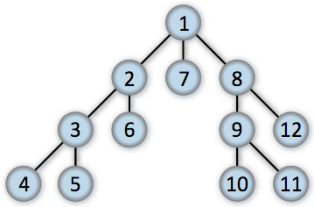
Clone the whole subgraph, including all the connected links whose source/target is in the subgraph. This is equivalent to calling `graph.cloneCells(graph.getSubgraph(cells))`.
If `opt.deep` is `true`, take into account embedded cells of the subgraph cells.
Return an object of the form `{ [original cell 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). `iterate` 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` explicitely 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. swaping 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 were `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 then 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(jsonString)` if you need to convert a JSON string into the object form:

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

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(cells[, opt])
```

Returns the bounding box that surrounds all the given cells. Links are ignored. An example:

```
var bbox = graph.getBBox(graph.getElements());
// { x: Number, y: Number, width: Number, height: Number }
```

dia.Graph.prototype.getCell

```
graph.getCell(id)
```

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

dia.Graph.prototype.getCells

```
graph.getCells()
```

Get all the elements and links in the graph.

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.getConnectedLinks

```
graph.getConnectedLinks(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 were left undefined, or if both of them were set to `true`, return

inbound as well as 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`.

Example use:

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

dia.Graph.prototype.getElements

`graph.getElements()`

Get all the elements in the graph (i.e. omit links).

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()
```

Get all the links in the graph (i.e. omit elements).

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. If `opt.inbound` is `true`, only inbound neighbords (all neighbors connected with a link who's `target` is the `element`) will be returned. Similarly, if `opt.outbound` is `true`, only outbound neighbors will be returned. If `opt.deep` is `true`, return also all the neighbors of all the elements embedded inside `element`.

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, 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.isNeighbor

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

Return `true` if `elementB` is a neighbor of `elementA`. If `opt.deep` is set to `true`, take into account embedded elements.
If `opt.outbound` is set to `true`, return `true` only if `elementB` is a successing neighbor of `elementA`.
Similarly, if `opt.inbound` is set to `true`, return `true` only if `elementB` is a preceeding neighbor of `elementA`.

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 ou want to replace all the cells in one go. `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.Link

The basic model for diagram links. It inherits from joint.dia.Cell with a few additional properties and methods specific to links.

Links have two crucial attributes: `source` and `target`. They define the starting point and the end point of the link. They can be defined as an element id or as 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 }
});
```

These are not the only attributes that may be specified for a newly-created `joint.dia.Link`, however. An individual link instance may provide a value for these attributes:

- `markup` - provide custom link markup. May be specified as an XML string or JSON array.
- `attrs` - provide custom link attributes. These allow you to change the style and size of SVG elements, identified by their selectors.
- `vertices` - provide an array of vertices.
- `vertexMarkup` - provide custom vertex markup (on hover).
- `toolMarkup` - provide custom tool markup (on hover).
- `doubleToolMarkup` - provide custom markup for second set of tools (on hover, if `linkView.doubleLinkTools` is `true`).
- `arrowheadMarkup` - provide custom arrowhead markup (on hover).
- `labels` - provide an array of labels. Each can have its own markup, position, and attrs specified.

It is also possible to pass custom attributes 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, to only enable custom contextmenu interaction for `link1` but not `link2`:

```
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
});
```

Custom Link

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

- `markup` - provide default link markup for all instances of this Link type. May be specified as an XML string or 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.
- `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).
- `defaultLabel` - provide default properties (markup, 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.

Example:

```
var CustomLink = joint.dia.Link.define('examples.CustomLink', {
    defaultLabel: {
        markup: [
            {
                tagName: 'circle',
                selector: 'body'
            }, {
                tagName: 'text',
                selector: 'label'
            }
        ],
        attrs: {
            label: {
                text: '%', // default label text
                fill: '#ff0000', // default text color
                fontSize: 14,
                textAnchor: 'middle',
                yAlignment: 'middle',
```

```
                pointerEvents: 'none'
            },
            body: {
                ref: 'label',
                fill: '#ffffff',
                stroke: '#000000',
                strokeWidth: 1,
                refR: 1,
                refCx: 0,
                refCy: 0
            }
        },
        position: {
            distance: 0.5, // place label at midpoint by default
            offset: {
                y: -20 // offset label by 20px upwards by default
            },
            args: {
                absoluteOffset: true // keep offset absolute when moving by default
            }
        }
    }
});

var link = new CustomLink({
    //...
});
```

Builtin Default Attributes

To ensure backwards compatibility, the `joint.dia.Link` class comes with a private builtin `defaultLabel` attribute. It is reproduced here for reference:

```
defaultLabel: {
    // builtin default markup:
    // used if neither defaultLabel.markup
    // nor label.markup is set
    markup: [
        {
            tagName: 'rect',
            selector: 'rect'
        }, {
            tagName: 'text',
            selector: 'text'
        }
    ],
    // builtin default attributes:
    // applied only if builtin 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
        }
    },
    // builtin default position:
    // used if neither defaultLabel.position
    // nor label.position is set
    position: {
        distance: 0.5
    }
}
```

If custom `position` object is not provided (neither as a type default nor as an instance value), builtin default label position is applied instead (`{ distance: 0.5 }`).

Furthermore, if custom `markup` is not provided (neither as a type default nor as an instance value), builtin default label markup is applied, alongside the builtin default label attrs object. However, it is very highly recommended that you provide both your own `markup` and your own `attrs` - unless you want to use the builtin default precisely as-is.

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` - triggered when the link toggled interpolation
- `change:manhattan` - 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.labels

JointJS supports adding labels on links. One link can have multiple labels, and each label can have different properties.

Properties recognized by JointJS are summarized in the following TypeScript-like schema:

```
{
    markup?: string | Array<{
        tagName: SVGElement,
        selector?: string
    }>,
    attrs?: {
        [key: selector]: {
            [key: SVG attribute | JointJS attribute]: any
        } | null
    },
    position?: number | {
        distance?: number,
        offset?: number | { x: number, y: number },
        args?: {
            absoluteDistance?: boolean,
            reverseDistance?: boolean,
            absoluteOffset?: boolean
        }
    }
}
```

Markup

The `markup` property specifies the markup of the label. It can be provided either as a parsable SVG (e.g. `'<rect /> <text />'`), or as a JSON array (e.g. `[{ tagName: 'rect' }, { tagName: 'text' }]`). The JSON allows the user to specify custom selectors for the SVGElements; these can then be used for targeting elements from within the `attrs` property.

If `markup` is not provided on the label, markup is taken from the Link type's `defaultLabel.markup` property. A custom Link type can be created by the user, providing a `defaultLabel` (see the `Link` documentation for more information). However, if the used link type does not provide `defaultLabel.markup` (this includes `joint.dia.Link` and `joint.shapes.standard.Link`), the builtin default Link markup is used, which defines markup as a JSON array with a `'body'` (a `<rect>` SVGElement) under a `'label'` (a `<text>` SVGElement).

Styling

The `attrs` property is an object where the keys are CSS selectors (referring to custom selectors or SVGElements specified in `markup`, e.g. `body` in the above example). They are expected to contain objects that specify native SVG and/or JointJS special attributes (e.g. `fill`), alongside the value to be assigned (e.g. `'white'`).

If the Link type provides `defaultLabel.attrs`, these attrs are merged with `label.attrs`. This allows you to reference selectors from `defaultLabel.markup` in `label.attrs` and, for example, simply add an attribute (`attrs: { body: { stroke: 'black' } }`).

If the builtin default markup is used (i.e. no custom `label.markup` was provided, and no `defaultLabel.markup`), several builtin default `attrs` are applied for reasons of backwards compatibility. These are merged with `defaultLabel.attrs` (if present on the Link prototype) and then `label.attrs` (if provided). See `Link` documentation for more information.

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. If no `position` is provided, the

builtin default (`{ distance: 0.5 }`) is used to maintain backwards compatibility.

number	If the distance is in the `[0,1]` range (inclusive), then the position of the label is defined as a percentage of the total length of the `.connection` path (the normalized length). For example, passing the number `0.5` positions the label to the middle of the `.connection` path. If the distance is larger than `1` (exclusive), the label will be positioned `distance` pixels away from the beginning of the path along the `.connection` path. If the distance is a negative number, the label will be positioned `distance` pixels away from the end of the path along the `.connection` path.
object	If `position` is specified as an object, it may have three properties: **distance** — A number specifying the distance of the label along the `.connection` path. Same as above. **offset** — Optional. Either a number or an object with `x` and `y` properties. *number*: If the offset is a positive number, displace the label perpendicularly to the right (in the direction of the `.connection` 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 `.connection` path) in the paper local coordinate system. *object*: **x** — Offset the label by `x` in the paper local coordinate system. **y** — Offset the label by `y` in the paper local coordinate system. **args** — Optional. An object with boolean options for the `linkView.getLabelPosition()` function. **absoluteDistance** — If user moves the label, record `distance` as an absolute number. (Relative distances used by default).

reverseDistance	If `absoluteDistane` is set and user moves the label, record `distance` as a negative absolute number. (Positive distances used by default.)
absoluteOffset	If user moves the label, record `offset` as an object with absolute `x` and `y` coordinates. (Relative offsets used by default).

dia.Link.presentation

The shape of a link is determined by five properties - **`source`**, **`target`**, **`vertices`**, **`connector`** and **`router`**.

Source and Target

The `source` and `target` properties have to be provided when creating a Link. Either an Element can be provided (either directly or via an `id` property) or a Point (with `x` and `y` properties).

```
link.source(rect);
link.source({ id: rect.id });

link.source(new g.Point(100, 100));
link.source({ x: 100, y: 100 });
```

If the end is specified as an Element, a specific subelement on that element 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 uses the port id.

For the purposes of link geometry, however, the most important are the `anchor` and `connectionPoint` properties of link end elements. The `connectionStrategy` paper option is also relevant to mention in this context.

Anchor

If the link end (source/target) is specified as an Element (whether or not one of the subelement specifications is in effect), the precise position of the link end's anchor may be specified by the `anchor` property. (If the link end is specified as a Point, its coordinates are used as its anchor directly.) 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 take the end element 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.

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 the 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');
```

Styling

Styling of the link is contained in the `attrs` property which has exactly the same structure as the `attrs` property of elements. Please refer to the joint.dia.Element and joint.dia.Link sections of this page for more information.

Links also have the `z` property which specifies the z-level of the link. It has exactly the same meaning as the `z` property of elements.

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)`

Set SVG attributes on subelements. This is a method analogous to attr method of `Joint.dia.Element`. The keys of the `attrs` object are CSS selectors matching the SVG element the link consists of. The values are objects containing SVG attributes and their values. `attrs` object will be mixined with `attrs` property of the `link` model. This is a convenient way of rewriting only some of the attributes of the SVG elements. For overwritting all attributes of all SVG elements, use `link.set('attrs', attrs)`. Here it is important to mention how the markup of a link looks like:

```
<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 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.

```
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' }
});
```

An alternative call using a string path and a value:

```
link.attr('.marker-source/fill', 'green');
```

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.getSourceElement

```
link.getSourceElement()
```

Return the source element of the link or `null` if there is none.

dia.Link.prototype.getTargetElement

```
link.getTargetElement()
```

Return the target element of the link or `null` if there is none.

dia.Link.prototype.getTransitions

```
link.getTransitions()
```

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

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
        }
    }
});
```

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.

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 an Element is returned (and not a Point), use the `link.getSourceElement` function instead.

```
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 by 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 `dia.Link` 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 an Element is returned (and not a Point), use the `link.getTargetElement` function instead.

`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 by 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 `dia.Link` 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 [, opt])
```

Add a new default label to the link at the coordinates provided. If you need to add a label with custom markup or properties, use the `link.addLabel()` function instead.

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 `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, {
            absoluteDistance: true,
            reverseDistance: true,
            absoluteOffset: true
        });
    }
});

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();
});
```

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.getBBox

`linkView.getBBox()`

Return the bounding box of the linkView.

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.getSegmentSubdivisons()` 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 [, opt])
```

Return a label `position` object based on the `x` and `y` coordinates provided.

The function translates the provided coordinates into an object with two 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`.

By default, `position.distance` is set relative to connection length (as a number in the `[0,1]` range that records the length ratio), and `position.offset` is set 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 `absoluteOffset` is not set (i.e. the default option), label can only be placed/moved in the area that is reachable by lines perpendicular to the link (that is, the label cannot be moved beyond link endpoints).

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 },
    args?: {
        absoluteDistance?: boolean,
        reverseDistance?: boolean,
        absoluteOffset?: boolean
    }
}
```

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>',
            position: this.getLabelPosition(x, y, { absoluteOffset: true }),
            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'
                }
            }
        }
        this.model.addLabel(idx, label);
    }
});

var paper = new joint.dia.Paper({
    //...
    linkView: CustomLinkView,
    interactive: { vertexAdd: false }
});
```

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.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 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 `evt`, `x` and `y` as arguments.

contextmenu

Triggered when pointer is right-clicked on a target (a `contextmenu` event is detected).

The callback function is passed `cellView`, `evt`, `x` and `y` 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 `evt`, `x` and `y` as arguments.

pointerdown

Triggered when pointer is pressed down on a target (a `mousedown` or `touchstart` event is detected). The paper also starts delegating respective `pointermove` and `pointerup` events (including their touch counterparts; see below).

The callback function is passed `cellView`, `evt`, `x` and `y` 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 `evt`, `x` and `y` as arguments.

pointermove

Triggered when pointer is moved over a target while pressed down (a `mousemove` or `touchmove` event is detected).

The callback function is passed `cellView`, `evt`, `x` and `y` as arguments.

cell:pointermove	Triggered when pointer is moved over a cell.
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 `evt`, `x` and `y` as arguments.

<table>
<tr>
<td>pointerup</td>
<td>
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.
</td>
</tr>
<tr>
<td>mouseover</td>
<td>
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>
<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>
</td>
</tr>
<tr>
<td>mouseout</td>
<td>
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>
</table>
</td>
</tr>
</table>

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 `svg` element of the paper outside any cell. The callback function is passed `evt` as argument.

mouseenter

Triggered when pointer enters the area above a target.

The callback function is passed `cellView` and `evt` as arguments.

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 `evt` as argument.

mouseleave

Triggered when pointer leaves the area above a target.

The callback function is passed `cellView` and `evt` as arguments.

cell:mouseleave	Triggered when pointer leaves the area above a cell.
link:mouseleave	Triggered when pointer leaves the area above a link.
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 `evt` as argument.

<table>
<tr><td>mousewheel</td><td>
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>
</td></tr>
<tr><td>magnet</td><td>
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.
<table>
<tr><td>element:magnet:pointerclick</td><td>Triggered when pointer is clicked on an element magnet and no link was created yet from this magnet (controlled by magnetThreshold option). Calling <code>evt.stopPropagation()</code> prevents triggering <code>cell:pointerclick</code> and <code>element:pointerclick</code> events.</td></tr>
<tr><td>element:magnet:pointerdblclick</td><td>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.</td></tr>
<tr><td>element:magnet:contextmenu</td><td>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.</td></tr>
</table>
</td></tr>
<tr><td>cell:highlight</td><td>Triggered when the <code>cellView.highlight</code> method is called on a link or element view.</td></tr>
</table>

	The callback function is passed `cellView` and `el` as arguments. The default handler method adds the `"highlighted"` CSS class to the cell, so that it can be targeted in user stylesheets. To use a different method for highlighting cells, first unregister the default handler with `paper.off('cell:highlight')` and then register a custom handler with `paper.on('cell:highlight', myHandler)`. This method is also called automatically, in two situations. First, if the user reconnects a link and the connection is valid as determined by the `paper.options.validateConnection` method. Second, if embedding mode is enabled on the paper and dragged element is above another element it can be dropped onto as determined by the `paper.options.validateEmbedding` method.
cell:unhighlight	Triggered when the `cellView.unhighlight` method is called on a link or element view. The callback function is passed `cellView` and `el` as arguments. The default handler method removes the `"highlighted"` CSS class from the cell. To use a different method, first unregister the default handler with `paper.off('cell:unhighlight')` and then register a custom handler with `paper.on('cell:unhighlight', myHandler)`. It is important to note that if a cell was highlighted using custom options, those exact same options must be provided to the unhighlight handler.
link:connect	Triggered when a link is connected to a cell. The callback function is passed `linkView`, `evt`, `elementViewConnected`, `magnet` and `arrowhead` as arguments.
link:disconnect	Triggered when a link is disconnected from a cell. The callback function is passed `linkView`, `evt`, `elementViewDisconnected`, `magnet` and `arrowhead` as arguments.
rendered:done	Triggered when the paper has finished rendering all cell views in case async rendering is enabled.
[custom]	Custom cell event can be triggered on pointerdown with Event attribute. Calling `evt.stopPropagation()` 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.cancelRenderViews

`paper.cancelRenderViews()`

When the method is called any asynchronous rendering in progress is canceled.

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 have the following form:

```
{
    type: <type of gradient>,
    stops: <stop colors>,
    attrs: <additional attributes>
}
```

Where `type` is either `'linearGradient'` or `'radialGradient'`, `attrs` is an object containing additional SVG attributes for the SVG gradient element and `stops` is an array of the ramps of color on the gradient. Each stop object is of the form:

```
{
    offset: <offset>,
    color: <color>,
    opacity: <opacity>
}
```

Where `offset` is a string representing the offset of the gradient stop, `color` indicates what color to use at that gradient stop and `opacity` is a number in the [0..1] range representing the transparency of the stop color. Example use:

```
var 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 and the `markerDefinition` is an object with `type` property and any other visual attributes. The valid values for `type` are `'path'`, `'circle'`, `'ellipse'`, `'rect'`, `'polyline'` and `'polygon'`.

```
var markerId = paper.defineMarker({
  type: 'path', // SVG Path
  fill: '#666',
  stroke: '#333',
  // The coordinate system for defining the path data
  // ------------------------------------------------
  // 0,0: marker-start, marker-end or marker-mid
```

```
  // n,0: n > 0 in path direction
  //      n < 0 opposite path direction
  // 0,m: m > 0 left to the path direction
  //      m < 0 right to the path direction
  d: 'M 10 -10 0 0 10 10 z'
});

// Draw an arrow at the start and the end of a path
svgPath.setAttribute('marker-start', 'url(#' + markerId + ')');
svgPath.setAttribute('marker-end', 'url(#' + markerId + ')');
```

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.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 then 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/shrink the paper to fit the content inside it. Snap the resulting width/height to the grid defined by `opt.gridWidth` / `opt.gridHeight`. `opt.padding` adds to the resulting width/height of the paper. The `opt.padding` can either be a number in which case it represents the padding on all the sides or it can be an object of the form `{ top: Number, right: Number, bottom: Number, left: Number }`. By default the method fits the paper to a content with positive coordinates only and sets the origin to (0,0) for the resulting paper. To change this behaviour use the `allowNewOrigin: ['negative'|'positive'|'any']` option. Set `allowNewOrigin` to `'negative'` to account for content with negative coordinates. Set `allowNewOrigin` to `'positive'` to allow origin to be set to the top-left coordinate of the content. Set `allowNewOrigin` to `'any'` to apply both. This method might internally trigger the `"resize"` and `"translate"` events that can be handled by listening on the paper object (`paper.on('resize', myHandler)`). `opt.minWidth` and `opt.minHeight` define the minium width and height of the paper and `opt.maxWidth` and `opt.maxHeight` define the maximum width and height of the paper after fitting it to the content.
You can try many of these options interactively in the paper demo.

Depracated usage: `paper.fitToContent(gridWidth, gridHeight, padding, opt)`

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).

dia.Paper.prototype.getContentBBox

```
paper.getContentBBox()
```

Return the bounding box of the content inside the paper, in client units (as it appears on the screen).

dia.Paper.prototype.hideTools

```
paper.hideTools()
```

Hide all tools attached to all 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.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.
- `viewport` - a reference to the SVG group `<g>` element the paper wraps all the rendered elements and links into. Paper transformations such as scale is performed on this element.
- `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 link views on this paper.

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. If `opt.padding` is set (default is `0`), there will be an additional padding in the resulting, scaled, paper content. If `opt.preserveAspectRatio` is defined (default is `true`), the aspect ratio of the scaled paper content will be preserved. `opt.minScaleX`, `opt.minScaleY`, `opt.maxScaleX` and `opt.maxScaleY` can be optionally used to set the minimum and maximum allowed scale factor for both axis. `opt.scaleGrid` is a number that will be used to as a rounding factor for the resulting scale. For example, if the resulting scale factor is calculated to be `1.15` and your `opt.scaleGrid` is set to `0.2`, then the resulting scale factor will be rounded to `1.2`. Last option is `opt.fittingBBox` which can be an object of the form `{ x: [number], y: [number], width: [number], height: [number] }` and is the area of the paper that should be scaled. By default `opt.fittingBBox` is `{ x: 0, y: y, width: paper.options.width, height: paper.options.height }`, i.e. the whole paper area. If the method called a `"scale"` event can be triggered on the paper. To try it yourself see paper demo.

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 `800`).
String	Dimension as CSS property value (e.g `"100%"`). *Make sure the paper container element has size defined.* `// Responsive Paper` `var paper = new joint.dia.Paper({ width: '100%', height: '100%' });` `window.on('resize', joint.util.debounce(function() {` `  paper.scaleContentToFit({ padding: 10 });` `}), false);`
null	No dimension set. Useful when size of the paper controlled in CSS.

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 link views on this paper.

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 paramater 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.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.async

`async` - when enabled, the paper renders cells added to the graph through `graph.resetCells()` or `graph.addCells()` asynchronously. This is very useful when you want to add a large number of cells into the graph. The rendering performance boost is significant and doesn't block the UI. The option accepts either `true` in which case default values are used or an object of the form `{ batchSize: <value> }` where you can specify the number of cells rendered in each animation frame (default is 50). Normally, the default `batchSize` works great but you might want to experiment with different values in case you encounter performance issues. It is important to note that when asynchronous rendering is used, some of the cell views might not yet be in the paper when you try to access them via the `paper.findViewByModel()`, `element.findView()` or `link.findView()` methods. What you should do in case you use async rendering is to wait for when the paper triggers the `render:done` event.

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 speficying 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.cellViewNamespace

`cellViewNamespace` - this option can be used to change the default behavior of JointJS which by default reads cell view definitions from the `joint.shapes` namespace. For example, if a cell is of type `'myshapes.MyElement'`, then the paper looks up `joint.shapes.myshapes.MyElementView` object type when rendering the cell onto the screen. If `cellViewNamespace` is set, the paper will read the view definition from the `myShapesNamespace.myshapes.MyElementView` object instead. This is useful in situations where you - for any reason - don't want to use the `joint.shapes` namespace for defining your own custom shapes. This option is often used in combination with `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.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 Presentation section of `joint.dia.Link` for more information about connection strategies.

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 Presentation section of `joint.dia.Link` for more information about anchors.

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 Presentation section of `joint.dia.Link` for more information about connection points.

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 Presentation section of `joint.dia.Link` for more information about connectors.

dia.Paper.prototype.options.defaultLink

`defaultLink` - link that should be created when the user clicks and drags and 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.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 Presentation section of `joint.dia.Link` for more information about routers.

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:

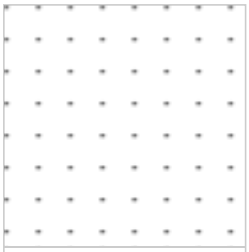
color	*string*	color of the default grid pattern.
thickness	*number*	thickness of the default grid pattern.

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:

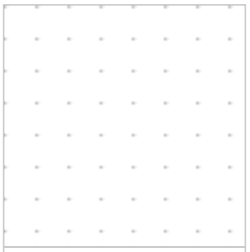
name	*string*	name of the pre-defined pattern. Can be either `dot`, `fixedDot`, `mesh`, `doubleMesh`
args	*object* | *array*	an extra arguments for the pre-defined grid patterns. It can be either an `object` (e.g. `dot`, `mesh`) or an `array` (e.g. `doubleMesh`)

Pre-defined grids with default settings:

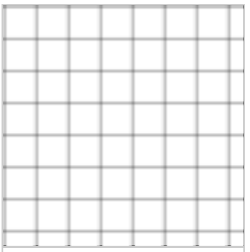
The paper from the images below has been scaled 2 times and has gridSize set to 10.



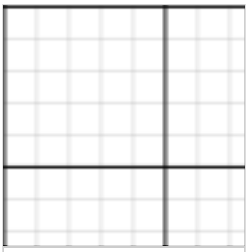
dot



fixedDot



mesh



doubleMesh

```
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.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 `'bbox'` (default), `'center'`, `'origin'`, `'corner'`, `'topRight'`, `'bottomLeft'` and it also can be a `function`. The function accepts an `element` and it returns array of possible candidates.

```
findParentBy: function(element) {
    var bBox = element.getBBox();
    return this.getElements().filter(function(el) {
        return bBox.intersect(el.getBBox());
    });
}
```

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.highlighting

`highlighting` - Configure which highlighter to use (and with which options) for each type of interaction.

List of highlighting interactions

- **default** - the default highlighter to use (and options) when none is specified
- **connecting** - when a valid link connection can be made to an element
- **embedding** - when a cell is dragged over another cell in embedding mode

- **magnetAvailability** - when showing all magnets to which a valid connection can be made
- **elementAvailability** - when showing all elements to which a valid connection can be made

Example usage:

```
new joint.dia.Paper({
    highlighting: {
        'default': {
            name: 'stroke',
            options: {
                padding: 3
            }
        },
        connecting: {
            name: 'addClass',
            options: {
                className: 'highlight-connecting'
            }
        }
    }
})
```

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?
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.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.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.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)` in which case it must return a bounding box (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. For example, to restrict translation of embedded elements by the bounding box defined by their parent element, you can do:

```
restrictTranslate: function(elementView) {
    var parentId = elementView.model.get('parent');
    return parentId && this.model.getCell(parentId).getBBox();
}
```

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.validateConnection

`validateConnection(cellViewS, magnetS, cellViewT, magnetT, end, linkView)` - decide whether to allow or disallow a connection between the source view/magnet (`cellViewS/magnetS`) and target view/magnet (`cellViewT/magnetT`). `end` is either `"source"` or `"target"` and tells which end of the link is being dragged. This is useful for defining whether, for example, a link starting in a port POut of element A can lead to a port PIn of elmement B. By default, all linkings are allowed.

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)` - 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 explicitely set to

"passive" (which is usually the case of input ports).

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	*string*	The name of this tools view. Default is `null` (no name).
tools	*Array<dia.ToolView>*	An array of tools that should be a part of the tools view. Default is `[]` (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.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 by the `toolsView.blurTool` 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.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.

---

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.

In the above demo, we listen to the paper for the `cell:pointerclick` event, then highlight the cell that was clicked.

```
paper.on('cell:pointerclick', function(cellView) {
    cellView.highlight();
});
```

As you can see it is possible to highlight a cell as follows:

```
cellView.highlight();
```

Unhighlighting a cell is simple too:

```
cellView.unhighlight();
```

To highlight a cell using a specific highlighter (and options):

```
cellView.highlight(null/* defaults to cellView.el */, {
    highlighter: {
        name: 'stroke',
        options: {
            width: 3
        }
    }
});
```

Or you can specify just the highlighter's name (its default options will be used):

```
cellView.highlight(null/* defaults to cellView.el */, {
    highlighter: 'stroke'
});
```

highlighters.addClass

Toggles a class name on the cell view's `$el`.

Available options:

- **className** - the class name to toggle on the cell's `$el`

Example usage:

```
cellView.highlight(null/* defaults to cellView.el */, {
    highlighter: {
        name: 'addClass',
        options: {
            className: 'some-custom-class'
        }
```

```
    }
});
```

highlighters.opacity

Changes the opacity of the cell view's `$el`.

When a cell is highlighted with the opacity highlighter, its `$el` 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:

```
cellView.highlight(null/* defaults to cellView.el */, {
    highlighter: {
        name: 'opacity'
    }
});
```

highlighters.stroke

Adds a stroke around the cell view's `$el`.

Available options:

- **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

Example usage:

```
cellView.highlight(null/* defaults to cellView.el */, {
    highlighter: {
        name: 'stroke',
        options: {
            padding: 10,
            rx: 5,
            ry: 5,
            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 graphBBox = joint.layout.DirectedGraph.layout(graph, {
    nodeSep: 50,
    edgeSep: 80,
    rankDir: "TB"
});
console.log('x:', graphBBox.x, 'y:', graphBBox.y)
console.log('width:', graphBBox.width, 'height:', graphBBox.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 `"TB"` (top-to-bottom) / `"BT"` (bottom-to-top) / `"LR"` (left-to-right) / `"RL"` (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: `'network-simplex'` (default), `'tight-tree'` or `'longest-path'`. number of pixels to use as a margin around the top and bottom of the graph.
resizeClusters	set to `false` if you don't want parent elements to stretch in order to fit all their embedded children. Default is `true`.
clusterPadding	A gap between the parent element and the boundary of its embedded children. It could be a number or an object e.g. `{ left: 10, right: 10, top: 30, bottom: 10 }`. It defaults to `10`.
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 `element.set('position', position)` but want to set the position in an animated fashion via transitions.
setVertices(link, vertices)	If set to `true` the layout will adjust the links by setting their vertices. It defaults to `false`. 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 `link.set('vertices', vertices)` but want to set the vertices in an animated fashion via transitions.
setLabels(link, labelPosition, points)	If set to `true` the layout will adjust the labels by setting their position. It defaults to `false`. 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 (`link.label(0);`) is positioned by the layout.*

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 `width` and `height` 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. `'l'` = left, `'c'` = center (default), `'r'` = right.
labelOffset	link	How many pixels to move the label away from the edge. Applies only when `labelPosition` is left or right.
labelSize	link	The width and height of the edge label in pixels. e.g. `{ width: 100, height: 50 }`

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 `joint.dia.Graph` object to the Graphlib graph object. `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`
fromGraphLib(glGraph, opt)	Convert the Graphlib graph representation to JointJS `joint.dia.Graph` object. `opt.importNode` and `opt.importEdge` are functions that accept a Graphlib node and edge objects and should return JointJS element/link. `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 }` `  });` `  }` `});`

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.

```
var rect = new joint.shapes.basic.Rect({
    // ...
    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` value calculated by the layout function **y** number Overrides the `y` value calculated by the layout function **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 en `end` point.

name	*string*	`line`		
args	*object*	**start**	{ x:number, y:number }	The line starting point
		end	{ x:number, y:number }	The line end point

```
{
    name: 'line',
    args: {
        start: { x: 10, y: 10 },
        end: { x: 20, y: 50 }
    }
}
```

Absolute

It lay a port out at the given position (defined as a `x`, `y` coordinates or percentage of the element dimensions).

name	*string*	`absolute`		
args	*object*	**x**	number | string	Sets the port's `x` coordinate. Can be defined as a percentage string ('50%') or as a number
		y	number | string	Sets the port's `y` coordinate. Can be defined as a percentage string ('50%') or as a number
		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 `ellipse`, `ellipseSpread`.		
args	*object*	**x**	number	Overrides the `x` value calculated by the layout function
		y	number	Overrides the `y` value calculated by the layout function
		dx	number	Added to the `x` value calculated by the layout function
		dy	number	Added to the `y` value calculated by the layout function
		dr	number	Added to the port delta rotation
		startAngle	number	Default value is `0`.
		step	number	Default `360 / portsCount` for the `ellipseSpread`, `20` for the `ellipse`
		compensateRotation	boolean	set `compensateRotation:true` 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 positions.

```
/**
* @param {Array<object>} portsArgs
* @param {g.Rect} elBBox shape's bounding box
* @param {object} opt Group options
* @returns {Array<g.Point>}
*/
function(portsArgs, elBBox, opt) {

    // ports on sinusoid
    return _.map(portsArgs, function(portArgs, index) {

        var step = -Math.PI / 8;
        var y = Math.sin(index * step) * 50;
        return g.point({ x: index * 12, y: y + elBBox.height });
    });
}
```

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.

<table>
<tr><td>name</td><td>string</td><td>Can be either `left`, `right`, `bottom`, `top`.</td></tr>
<tr><td>args</td><td>object | string</td><td>
<table>
<tr><td>x</td><td>number</td><td>Overrides the `x` value calculated by the layout function</td></tr>
<tr><td>y</td><td>number</td><td>Overrides the `y` value calculated by the layout function</td></tr>
<tr><td>angle</td><td>number</td><td>The port label rotation angle.</td></tr>
<tr><td>attrs</td><td>number</td><td>JointJS style attribute applied on label's DOM element and it's children. The same notation as the `attrs` property on [`Element`](#joint.dia.Element.presentation).</td></tr>
</table>
</td></tr>
</table>

```
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.

name	*string*	Can be either `inside`, `outside`, `insideOriented`, `outsideOriented`.
args	*object* | *string*	**offset** number Offset in direction from the shape's center. **attrs** number JointJS style attribute applied on label's DOM element and it's children. The same notation as the `attrs` property on [`Element`](#joint.dia.Element.presentation).

```
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.

name	*string*	Can be either `radial`, `radialOriented`.
args	*object* | *string*	**offset** number Offset in direction from the shape's center. **attrs** number JointJS style attribute applied on label's DOM element and it's children. The same notation as the `attrs` property on [`Element`](#joint.dia.Element.presentation).

```
label: {
    position: {
```

```
        name :'radialOriented',
        args: {
            offset: 0,
            attrs: {}
        }
    }
}
```

Manual placement

It allows to set a label position directly.

name	*string*	`manual`
args	*object* | *string*	**x** — number — Sets the label's `x` coordinate. **y** — number — Sets the label's `y` coordinate. **angle** — number — The port label rotation angle. **attrs** — number — JointJS style attribute applied on label's DOM element and it's children. The same notation as the `attrs` property on [`Element`](#joint.dia.Element.presentation).

```
label: {
    position: {
        name: 'manual',
        args: {
            x: 10,
            y: 20,
            angle: 45,
            attrs: {}
        }
    }
}
```

Port label layout demo

---

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	*number*	What should be the opacity of the tool when it is focused (e.g. with the `toolView.focus` 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 one additional argument, which can be passed as an object to the link tool constructor:

padding	*number*	This option determines whether the boundary area should be visually inflated and if so, by how much. Default is `10`.

Example:

```
var boundaryTool = new joint.linkTools.Boundary({
    focusOpacity: 0.5,
    padding: 20
});
```

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	*number*	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`.
	string	Percentage strings (e.g. `'40%'`) are also accepted.
rotate	*boolean*	Should the button rotate according to the slope of the link at the position specified by `distance`? Default is `false`.
offset	*number*	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 `0`.

action	*function*	What should happen when the user clicks the remove button? Default is `undefined` (no interaction). The callback function is expected to have the signature `function(evt)` where `evt` is a `jQuery.Event` object. The button view is available inside the function as `this`; the button model is available as `this.model`.
markup	*JSONMarkup*	The markup of the button, provided in the JointJS JSON format. Default is `undefined` (no content).

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 within the `joint.linkTools` namespace. Then, a new instance of the custom button type can be obtained by calling its constructor:

```
joint.linkTools.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 joint.linkTools.InfoButton();
```

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	*number*	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 `60`.
	string	Percentage strings (e.g. `'40%'`) are also accepted.
rotate	*boolean*	Should the button rotate according to the slope of the link at the position specified by `distance`? Default is `false`.
offset	*number*	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 `0`.
action	*function*	What should happen when the user clicks the remove button? Default: `function(evt) { this.model.remove({ ui: true, tool: this.cid }); }` The callback function is expected to have the signature `function(evt)` where `evt` is a `jQuery.Event` object. The button view is available inside the function as `this`; the button model is available as `this.model`.

markup	*JSONMarkup*	The markup of the button, provided in the JointJS JSON format. Default: `[{ tagName: 'circle', selector: 'button', attributes: { 'r': 7, 'fill': '#FF1D00', 'cursor': 'pointer' } }, { 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' } }]`

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	*boolean*	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 `link.vertices` function, for example.
segmentLengthThreshold	*number*	The minimum segment length for which to display a segment handle (to prevent the handle from overflowing its segment). Default is `40`.

snapRadius	*number*	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	*number*	If the `snapRadius` option is set to `true` 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 `true`, meaning that the handle snaps with the segment.

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 five additional arguments, which can be passed as an object to the link tool constructor:

redundancyRemoval	*boolean*	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 `true`. 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 `link.vertices` function, for example.
restrictArea	*boolean*	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 `true`.
areaPadding	*number*	If the `restrictArea` option is set to `true`, 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 `10`. Note that this is a purely cosmetic setting; regardless of the provided value, the movement stays restricted to the original uninflated bounding box.
snapRadius	*number*	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 `10`. (For link source, the anchor reference is the first vertex. If there are no vertices, it is the target anchor.)
snap	*function*	What snap function should be applied when the user moves the anchor? Default is a simple function that emulates the snapping behavior of Vertices and Segments link tools: If the value of one of the user pointer coordinates is within `snapRadius` 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 `g.Point` and have the signature `function(coords, endView, endMagnet, endType, linkView, toolView)`:

coords	*g.Point*	A Point object recording the x-y coordinates of the user pointer when the snap function was invoked.
endView	*dia.ElementView*	The ElementView which contains the restricted area. The Element model can be accessed as `endView.model`; this may be useful for writing conditional logic based on element attributes.
endMagnet	*SVGElement*	The SVGElement in our page that contains the magnet (element/subelement/port) which serves as the restricted area.
endType	*string*	Is the end view the source (`'source'`) or target (`'target'`) of the link?
linkView	*dia.LinkView*	The LinkView to which the tool is attached. The Link model can be accessed as `linkView.model`; this may be useful for writing conditional logic based on link attributes.
toolView	*dia.ToolView*	This tool. May be useful for writing conditional logic based on tool options (e.g. the default algorithm refers to `toolView.options.snapRadius`).

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).

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 five additional arguments, which can be passed as an object to the link tool constructor:

redundancyRemoval	*boolean*	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 `true`. 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 `link.vertices` function, for example.
restrictArea	*boolean*	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 `true`.
areaPadding	*number*	If the `restrictArea` option is set to `true`, 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 `10`. Note that this is a purely cosmetic setting; regardless of the provided value, the movement stays restricted to the original uninflated bounding box.
snapRadius	*number*	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 `10`. (For link target, the anchor reference is the last vertex. If there are no vertices, it is the source anchor.)
snap	*function*	What snap function should be applied when the user moves the anchor? Default is a simple function that emulates the snapping behavior of Vertices and Segments link tools: If the value of one of the user pointer coordinates is within `snapRadius` 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 `g.Point` and have the signature `function(coords, endView, endMagnet)`:

coords	*g.Point*	A Point object recording the x-y coordinates of the user pointer when the snap function was invoked.
endView	*dia.ElementView*	The ElementView which contains the restricted area. The

		Element model can be accessed as `endView.model`; this may be useful for writing conditional logic based on element attributes.
endMagnet	*SVGElement*	The SVGElement in our page that contains the magnet (element/subelement/port) which serves as the restricted area.
endType	*string*	Is the end view the source (`'source'`) or target (`'target'`) of the link?
linkView	*dia.LinkView*	The LinkView to which the tool is attached. The Link model can be accessed as `linkView.model`; this may be useful for writing conditional logic based on link attributes.
toolView	*dia.ToolView*	This tool. May be useful for writing conditional logic based on tool options (e.g. the default algorithm refers to `toolView.options.snapRadius`).

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).

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	*boolean*	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 `true`. 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 `link.vertices` function, for example.
snapRadius	*number*	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	*boolean*	Can the user add new vertices (by clicking a segment of the link)? Default is `true`.

Example:

```
var verticesTool = new joint.linkTools.Vertices({
    focusOpacity: 0.5,
    redundancyRemoval: false,
    snapRadius: 10,
    vertexAdding: false,
});
```

---

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
- `'oneSide'` - restricted orthogonal router

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 `'oneSide'` 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');
```

routers.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	*Array<g.Point>*	The vertices of the route.
args	*object*	An object with additional optional arguments passed to the router method by the user when it was called (the `args` property).
linkView	*dia.LinkView*	The LinkView of the connection. The Link model can be accessed as the `model` 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;
});
```

routers.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	*number*	Size of the imaginary grid followed by the `'manhattan'` pathfinding algorithm. Lower number -> higher complexity. The `'manhattan'` router performs best when `step` has the same value as `dia.Paper.option.gridSize`. Default is `10`.
padding	*number* | *object*	Padding applied around start/end elements and obstacles. Default is the `step` value (see above). The `util.normalizeSides` 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 `left`, `top`, `right`, `bottom`, `horizontal` and/or `vertical` sides.
maximumLoops	*number*	The maximum number of iterations of the pathfinding loop. If the number of iterations exceeds this maximum, pathfinding is aborted and the fallback router (`'orthogonal'`) is used instead. Higher number -> higher complexity. Default is `2000`.
maxAllowedDirectionChange	*number*	Maximum change of direction of the `'manhattan'` route, in degrees. Default is `90`.
perpendicular	*boolean*	Should the `linkView.perpendicular` 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 `true`.
excludeEnds	*Array<string>*	An array with strings `'source'` and/or `'target'` that tells the algorithm that the given end element(s) should not be considered as an obstacle. Default is `[]` (both are considered obstacles).
excludeTypes	*Array<string>*	An array of element types that should not be considered obstacles when calculating the route. Default is `['basic.Text']`.
startDirections	*Array<string>*	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 `['bottom']`). Default is `['left', 'right', 'top', 'bottom']` (all directions allowed).
endDirections	*Array<string>*	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 `['bottom']`). Default is `['left', 'right', 'top', 'bottom']` (all directions allowed).

Example:

```
link.router('manhattan', {
    excludeEnds: ['source'],
    excludeTypes: ['myNamespace.MyCommentElement'],
    startDirections: ['top'],
```

```
    endDirections: ['bottom']
});
```

routers.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	*number*	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	*number*	Default changes to `45`.
startDirection	*Array<string>*	Same as `'manhattan'` (i.e. only the four orthogonal directions are accepted as start directions).
endDirection	*Array<string>*	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.oneSide

The `'oneSide'` router is a restricted version of the `'orthogonal'` router. Exactly three route segments are generated. The route leaves the start element in a specified direction (the `args.side` property), transitions with a single segment towards the end element, and then enters the end element from the specified direction again. Note that this router does not support link `vertices`. The router does not avoid obstacles. Two arguments are accepted, which can be passed within the `router.args` property.

side	*string*	The direction of the route. Either `'left'`, `'right'`, `'top'` or `'bottom'`. Default is `'bottom'`.
padding	*number* | *object*	The minimum distance from element at which the first/last route angle may be placed. Default is `40`. The `util.normalizeSides` 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 `left`, `top`, `right`, `bottom`, `horizontal` and/or `vertical` sides. Only the side specified in the `side` argument is considered by the router (see above).

Example:

```
link.router('oneSide', {
    side: 'top',
    padding: 30
});
```

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	*number* | *object*	The minimum distance from element at which the first/last route angle may be placed. Default is `20`. The `util.normalizeSides` 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 `left`, `top`, `right`, `bottom`, `horizontal` and/or `vertical` 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);
```

---

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: {
        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.fitEmbeds()` that resizes the `coupled` shape, so it visually contains all shapes embedded in.

`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 `attrs` properties

Selector	Node	Description
root	*SVGGElement*	Container of all nodes
image	*SVGImageElement*	Image body of the shape
border	*SVGRectElement*	Border around the image
background	*SVGRectElement*	Area behind the image
label	*SVGTextElement*	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.BoarderedImage');
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	*SVGGElement*	Container of all nodes
body	*SVGCircleElement*	Circular body of the shape
label	*SVGTextElement*	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	*SVGGElement*	Container of all nodes
body	*SVGPathElement*	Lateral area of the cylinder *The shape has a custom attribute `lateralArea` set to `10` by default. The attribute sets the SVGPathElement's `d` in the shape of the lateral area of a tilted cylinder. The*

		attribute expects a `topRy` 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 `cylinder.topRy` function for more detailed specification.)
top	*SVGEllipseElement*	Top of the cylinder (cylinder base)
label	*SVGTextElement*	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	*SVGGElement*	Container of all nodes
line	*SVGPathElement*	Inner connection
outline	*SVGPathElement*	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	*SVGGElement*	Container of all nodes
body	*SVGEllipseElement*	Elliptical body of the shape
label	*SVGTextElement*	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	*SVGGElement*	Container of all nodes
body	*SVGRectElement*	Rectangular body of the shape
image	*SVGImageElement*	Image inside the body
label	*SVGTextElement*	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	*SVGGElement*	Container of all nodes
body	*SVGRectElement*	Rectangular body of the shape
header	*SVGRectElement*	Rectangular header of the shape
headerText	*SVGTextElement*	Text inside the header
bodyText	*SVGTextElement*	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	*SVGGElement*	Container of all nodes
image	*SVGImageElement*	Image body of the shape
label	*SVGTextElement*	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	*SVGGElement*	Container of all nodes
image	*SVGImageElement*	Image inscribed in the ellipse
border	*SVGEllipseElement*	Border around the ellipse
background	*SVGEllipseElement*	Area of the ellipse
label	*SVGTextElement*	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	*SVGGElement*	Container of all nodes
line	*SVGPathElement*	Visible connection of the link
wrapper	*SVGPathElement*	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	*SVGGElement*	Container of all nodes
body	*SVGPathElement*	Generic body of the shape
label	*SVGTextElement*	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	*SVGGElement*	Container of all nodes
body	*SVGPolygonElement*	Polygonal body of the shape
label	*SVGTextElement*	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	*SVGGElement*	Container of all nodes
body	*SVGPolylineElement*	Generic body of the shape
label	*SVGTextElement*	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	*SVGGElement*	Container of all nodes
body	*SVGRectElement*	Rectangular body of the shape
label	*SVGTextElement*	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	*SVGGElement*	Container of all nodes
line	*SVGPathElement*	Visible connection of the link
shadow	*SVGPathElement*	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	*SVGGElement*	Container of all nodes
body	*SVGRectElement*	Rectangular body of the shape
label	*HTMLDivElement*	Text inside the body

```
var textBlock = new joint.shapes.tandard.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])
```

(*Deprecated*) An alias for `_.assign`. See the Lodash documentation entry for more information.

util.bindAll

```
util.bindAll(object, methodNames)
```

An alias for `_.bindAll`. See the Lodash documentation entry for more information.

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.

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)
```

An alias for `_.camelCase`. See the Lodash documentation entry for more information.

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)
```

An alias for `_.clone`. See the Lodash documentation entry for more information.

util.cloneDeep

```
util.cloneDeep(value)
```

An alias for `_.cloneDeep`. See the Lodash documentation entry for more information.

util.dataUriToBlob

```
util.dataUriToBlob(dataUri)
```

Convert a Data URI string into a Blob object.

util.debounce

```
util.debounce(func [, wait, options])
```

An alias for `_.debounce`. See the Lodash documentation entry for more information.

util.deepMixin

```
util.deepMixin(destinationObject, sourceObject, options)
```

(*Deprecated*) An alias for `_.mixin`. See the Lodash documentation entry for more information.

util.deepSupplement

```
util.deepSupplement(destinationObject [, ...sourceObjects])
```

(*Deprecated*) An alias for `_.defaultsDeep`. See the Lodash documentation entry for more information.

Use the `util.defaultsDeep` function instead.

util.defaults

```
util.defaults(destinationObject [, ...sourceObjects])
```

An alias for `_.defaults`. See the Lodash documentation entry for more information.

util.defaultsDeep

```
util.defaultsDeep(destinationObject [, ...sourceObjects])
```

An alias for `_.defaultsDeep`. See the Lodash documentation entry for more information.

util.difference

```
util.difference(array [, ...excludedValuesArrays])
```

An alias for `_.difference`. See the Lodash documentation entry for more information.

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)
```

An alias for `_.flattenDeep`. See the Lodash documentation entry for more information.

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])
```

An alias for `_.forIn`. See the Lodash documentation entry for more information.

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.groupBy

```
util.groupBy(collection [, iteratee])
```

An alias for `_.groupBy`. See the Lodash documentation entry for more information.

util.guid

```
util.guid()
```

Return an identifier unique for the page.

util.has

```
util.has(object, propertyPath)
```

An alias for `_.has`. See the Lodash documentation entry for more information.

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])
```

An alias for `_.intersection`. See the Lodash documentation entry for more information.

util.invoke

```
util.invoke(collection, methodPath [, args])
```

```
util.invoke(collection, functionToInvokeForAll [, args])
```

An alias for `_.invokeMap`. See the Lodash documentation entry for more information.

util.isBoolean

```
util.isBoolean(value)
```

Return `true` when `value` is a boolean.

util.isEmpty

```
util.isEmpty(value)
```

An alias for `_.isEmpty`. See the Lodash documentation entry for more information.

util.isEqual

```
util.isEqual(value, otherValue)
```

An alias for `_.isEqual`. See the Lodash documentation entry for more information.

util.isFunction

```
util.isFunction(value)
```

An alias for `_.isFunction`. See the Lodash documentation entry for more information.

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)
```

An alias for `_.isPlainObject`. See the Lodash documentation entry for more information.

util.isString

```
util.isString(value)
```

Return `true` when `value` is a string.

util.merge

```
util.merge(destinationObject, ...sourceObjects [, customizer])
```

An alias for `_.mergeWith`. See the Lodash documentation entry for more information.

util.mixin

```
util.mixin(destinationObject [, ...sourceObjects])
```

(*Deprecated*) An alias for `_.assign` (not `_.mixin`). See the Lodash documentation entry for more information.

util.nextFrame

```
util.nextFrame(callback [, context])
```

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. Returns an ID of the frame request. For convenience, you can pass a `context` for your `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 [, ...propertyPathsToOmit])
```

An alias for `_.omit`. See the Lodash documentation entry for more information.

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 [, ...propertyPathsToPick])
```

An alias for `_.pick`. See the Lodash documentation entry for more information.

util.result

```
util.result(object, propertyPath [, defaultValue])
```

An alias for `_.result`. See the Lodash documentation entry for more information.

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><img src="javascript:alert(&quot;Hacked&quot;);">'); // =>
'<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)`

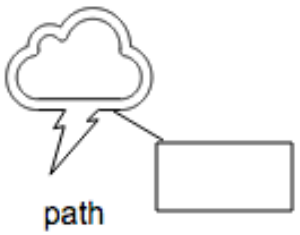
Set a `value` at the `path` in a nested `object`. `delim` is the delimiter used in the `path`. Returns the augmented `object`.

```
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 [, iterateesArray])`

An alias for `_.sortBy`. See the Lodash documentation entry for more information.

util.sortedIndex

`util.sortedIndex(sortedArray, valueToInsert [, iteratee])`

An alias for `_.sortedIndexBy`. See the Lodash documentation entry for more information.

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 `_.defaults`. See the Lodash documentation entry for more information.

Use the `util.defaults` function instead.

util.template

`util.template(html)`

Return a template function that returns pre-compiled `html` when called.

util.toArray

```
util.toArray(value)
```

An alias for `_.toArray`. See the Lodash documentation entry for more information.

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])
```

An alias for `_.union`. See the Lodash documentation entry for more information.

util.uniq

```
util.uniq(array [, iteratee])
```

An alias for `_.uniqBy`. See the Lodash documentation entry for more information.

util.uniqueId

```
util.uniqueId([prefix])
```

An alias for `_.uniqueId`. See the Lodash documentation entry for more information.

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])
```

An alias for `_.without`. See the Lodash documentation entry for more information.

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

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.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 diffrence 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.divide

```
curve.divide(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.

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:

`startControlPoint1`	First curve's `controlPoint1`.
`startControlPoint2`	First curve's `controlPoint2`.
`divider`	The point at `t`; `end` point of the first curve and `start` point of the second curve.
`dividerControlPoint1`	Second curve's `controlPoint1`.
`dividerControlPoint2`	Second curve's `controlPoint2`

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.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.

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.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.

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.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 a one of the following strings: `'NE'`, `'E'`, `'SE'`, `'S'`, `'SW'`, `'W'`, `'NW'` and `'N'`.

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.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.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 curve. If `length` is higher than curve 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])
```

Rounds the line to the given precision.

Default precision is `0`.

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.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.

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.

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 insterted 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. 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.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.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.

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.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.

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.

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.tangent()` 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.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.

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] }`

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)
```

Computes the angle (in degrees) between the line passing through the point and point `p1` and the line passing through the point and point `p2`.

The ordering of points `p1` and `p2` is important. The function returns a value between 0 and 180 degrees when the angle is counterclockwise, and a value between 180 and 360 degrees when the angle is clockwise. The function returns NaN if any of the points `p1` and `p2` are coincident with this point.

g.point.prototype.bearing

```
point.bearing(p)
```

Return the bearing (cardinal direction) of the line between the point and another point `p`.

The return value is a one of the following strings: `'NE', 'E', 'SE', 'S', 'SW', 'W', 'NW' and 'N'`.

g.point.prototype.changeInAngle

```
point.changeInAngle(dx, dy, ref)
```

Return the change in angle that is the result of moving the point from its previous position (`-dx`, `-dy`) to its new position.

This move is relative to the `ref` point and x axis.

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])
```

Rounds the point to the given precision.

Default precision is `0`.

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.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 `x@y`.

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)
```

Update the point's `x` and `y` coordinates with new values and return the point itself. Useful for chaining.

g.point.prototype.vectorAngle

```
point.vectorAngle(p)
```

Computes the angle (in degrees) between the line passing through the point (0,0) and this point and the line passing through the point (0,0) and point `p`.

The function returns a value between 0 and 180 degrees when the angle is counterclockwise, and a value between 180 and 360 degrees when the angle is clockwise. The function returns NaN if called from point (0,0) or if `p` is (0,0).

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.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 wich a tangent may be found.

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.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. Example: `"10,20 40,40 32,29"`

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 wich 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 wich a tangent may be found.

g.polyline.prototype.toString

```
polyline.toString()
```

Returns the polyline as a string in the form `x1@y1,x2@y2,...`.

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 as a single argument an object in the form `{ 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.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.round

```
rect.round([precision])
```

Rounds the rectangle to the given precision.

Default precision is `0`.

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

```
point.toString()
```

Returns the rectangle as a string `x@y x@y`.

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.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

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**.

---

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);
```

---

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.

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);
```

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).

prototype.appendTo

```
vel.appendTo(node)
```

Append `vel.node` as a child to `node`.

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.

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.

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).

prototype.children

```
vel.children()
```

Returns an array of Vectorizer-wrapped children of this element's node.

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.

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.

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>`.

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>`.

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.

prototype.empty

```
vel.empty()
```

Removes all the child nodes from the Vectorizer element.

prototype.find

```
vel.find(selector)
```

Return all elements wrapped in the Vectorizer object matching the `selector`.

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.

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.

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`.

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 elemenent 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.

prototype.getTransformToElement

```
vel.getTransformToElement(toElement)
```

Return an SVGMatrix that specifies the transformation necessary to convert vel coordinate system into toElement coordinate system.

prototype.hasClass

```
vel.hasClass(className)
```

Return `true` if the element contains `className` in its `class` attribute. Return `false` otherwise.

prototype.id

```
vel.id
```

Element's unique identifier. A property of the element for getting/setting `vel.node.id`.

prototype.index

```
vel.index()
```

Return the index of the SVG element amongst its siblings.

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.

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).

prototype.remove

```
vel.remove()
```

Remove the element from the DOM.

prototype.removeAttr

```
vel.removeAttr(attribute)
```

Remove the specified attribute from the `vel.node`.

prototype.removeClass

```
vel.removeClass(className)
```

Remove `className` from the element `class` attribute if it contains it. Return the Vectorizer object for easy chaining.

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.

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.

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`.

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 og the image.

prototype.setAttributes

```
vel.setAttributes(attributes)
```

Take an `attributes` object with `{ attribute: value }` pairs and assign them to vel.

prototype.svg

```
vel.svg()
```

Return the Vectorizer object for the root SVG element of the element.

prototype.tagName

```
vel.tagName()
```

Return the tag name of this element, in uppercase.

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' }
});
```

prototype.toGemetryShape

```
vel.toGeometryShape()
```

Convert the SVGElement to an equivalent geometric shape. The element's transformations are not taken into account.

SVGElement	Shape
SVGRectElement	g.Rect
SVGLineElement	g.Line
SVGCircleElement	g.Ellipse
SVGEllipseElement	g.Ellipse
SVGPolygonElement	g.Polyline
SVGPolylineElement	g.Polyline
SVGPathElement	g.Path
others	g.Rect

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.

prototype.toLocalPoint

```
vel.toLocalPoint(x, y)
```

Convert a global point with coordinates `x` and `y` into the coordinate space of the element.

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`

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.

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.

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] }`.

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.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 tranform 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}`.

Rappid

Rappid 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 Rappid toolkit to build groundbreaking applications in a fraction of a time.

If you're looking for a documentation to the **core JointJS library**, go to the JointJS API reference page.

For more information about the **Rappid toolkit, licensing and online purchase**, see the About Rappid page.

Browse through the menu on the left to see the documentation to the Rappid plugins.

The Rappid 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.
- **com** - Communication with the server. It currently only contains a module for Real-time collaboration.
- **shapes** - Different kinds of diagrams shapes including a wide range of charts.
- **format** - Exports to vector and raster formats, imports and print.
- **storage** - Storage.
- **alg** - Algorithms and data structures.
- **layout** - Automatic layouts.

Installation

Include `rappid.min.css` and `rappid.min.js` files to your HTML together with all the Rappid dependencies (jQuery, Lodash and Backbone).

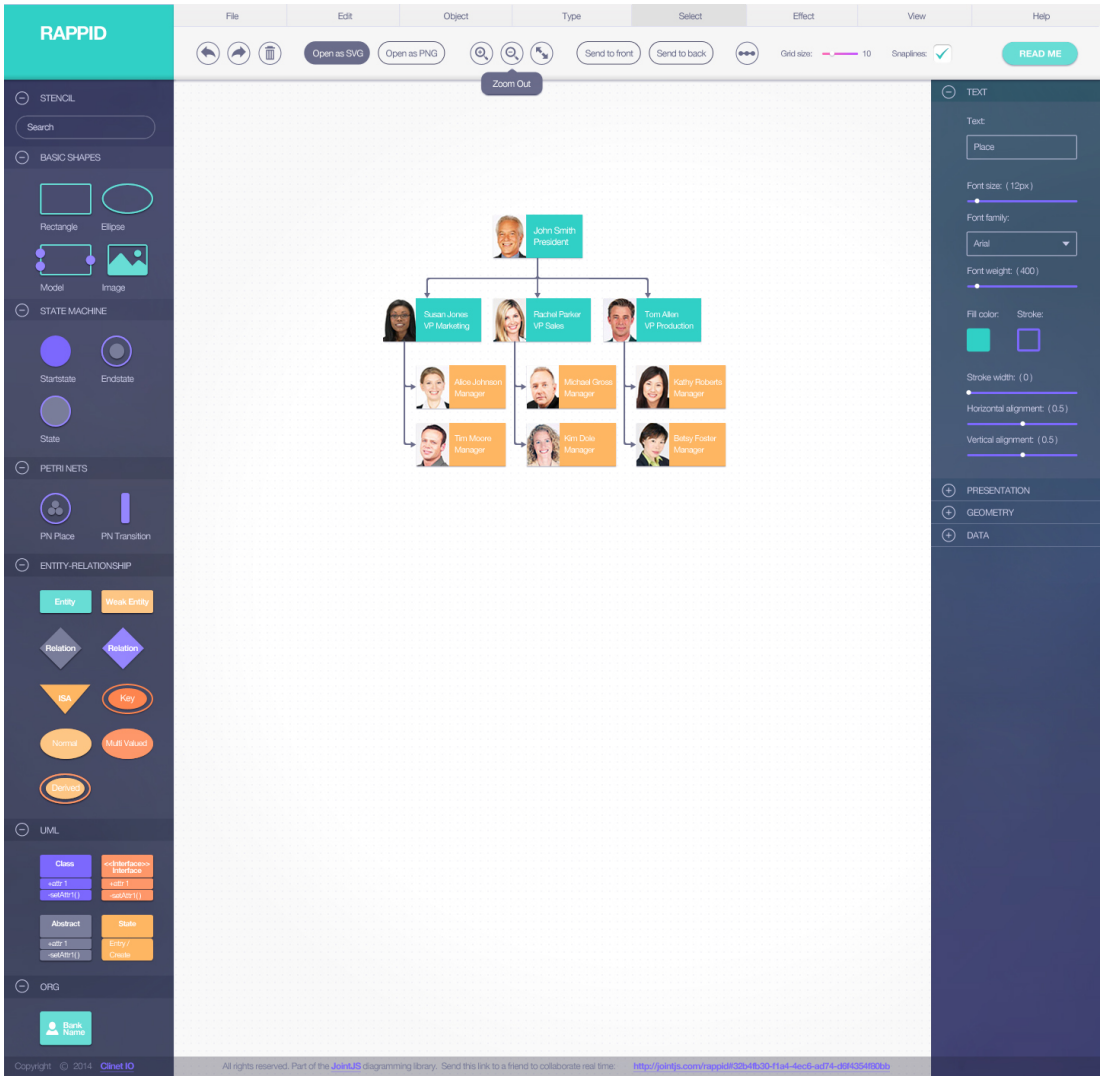
```
<link rel="stylesheet" type="text/css" href="dist/rappid.min.css">
<script src="lib/jquery/jquery.min.js"></script>
<script src="lib/lodash/dist/lodash.min.js"></script>
<script src="lib/backbone/backbone-min.js"></script>
<script src="dist/rappid.min.js"></script>
```

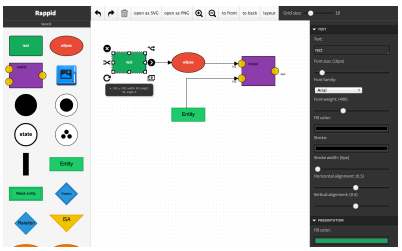
It is highly recommended to use the exact same versions of all the dependencies as they are included in the package.

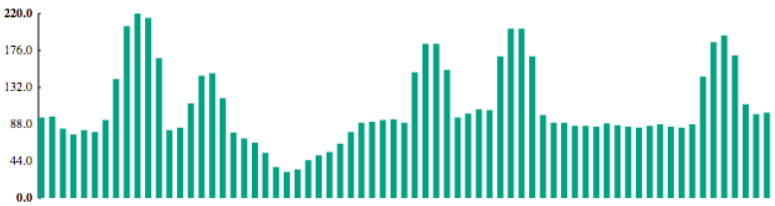
Note that each component documentation page has an *Installation* section. You do not need to follow the instructions from the section if you installed Rappid as described above (with `rappid.min.js`, `rappid.min.css` and all the dependencies). The Installation section of the components is only useful 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:

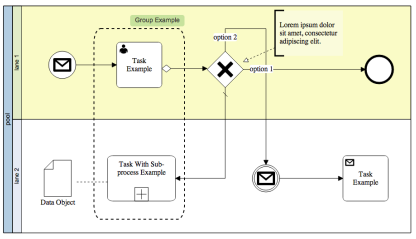
```
<link rel="stylesheet" type="text/css" href="dist/joint.css">
<script src="lib/jquery/jquery.min.js"></script>
<script src="lib/lodash/dist/lodash.min.js"></script>
```

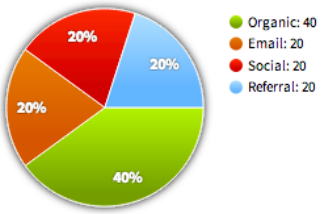
```
<script src="lib/backbone/backbone-min.js"></script>
<script src="dist/joint.js"></script>

// 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>
```

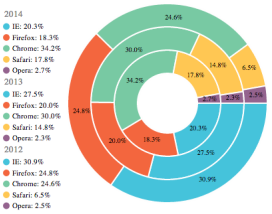




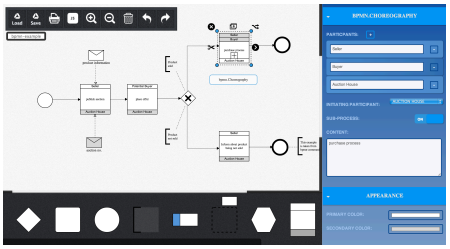


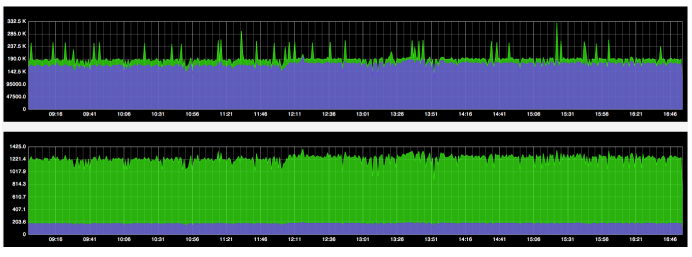

Group Example
Task Example
option 2
option 1
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
pool
lane 1
lane 2
Data Object
Task With Sub-process Example
Task Example


20%
20%
20%
40%
Organic: 40
Email: 20
Social: 20
Referral: 20


2014
IE: 20.3%
Firefox: 18.3%
Chrome: 34.2%
Safari: 17.8%
Opera: 2.7%
2013
IE: 27.5%
Firefox: 20.0%
Chrome: 30.0%
Safari: 14.8%
Opera: 2.3%
2012
IE: 30.9%
Firefox: 24.8%
Chrome: 24.6%
Safari: 6.5%
Opera: 2.5%


CPU
80%
70
100
30%
Mem
70%





Rappid - alg

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 `dia.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 or 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; }`

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 `opt.data` array is passed, it must be an array of items of the form `{ priority: Number, value: Object }`. In this case, the priority queue will be initialized with this array. It's like calling `insert(priority, value)` for each item of this array. `opt.comparator` can optionally be a function that will be used to compare two priorities. The signature of this function is `function(a, b)`. The function should return a value less then `0` if priority `a` is lower than priority `b`, value equal to `0` if the priorities are the same and value bigger than `0` if priority `a` is higher than priority `b`. The comparator function defaults to: `function(a, b) { return a - b }`. 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 `true` if the priority queue is empty, `false` otherwise.
insert(priority, value [, id])	Insert a `value` with `priority` to the queue. Optionally pass a unique `id` of this item. Passing unique IDs for each item you insert allows you to use the `updatePriority()` 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.

Rappid - com

com.Channel

The Channel plugin is a very powerful tool that brings **real-time collaboration** capability to your applications. The plugin **synchronises JointJS graphs between clients and server** and automatically resolves merge conflicts. Real-time collaboration is only one application of this plugin. Others include push updates, easy persistence, monitoring and more.

The Channel plugin uses Operational transformations to maintain consistency across concurrent updates. This means that even though there are different updates to the same JointJS graph that happened at the same time at different sites (across clients and even server), the Channel plugin makes sure the result, after applying those updates, is exactly the same at all sites while preserving intention.

The Channel plugin runs both in browsers and NodeJS environment and requires a NodeJS server running in order to synchronize clients. The procedure to get the channel plugin up and running is quite straightforward and is described in this document. However, if you, for any reason, do not want to maintain a NodeJS application on your backend, please let us know. Thanks to the cross-domain nature of the Channel plugin, we can run the NodeJS application for you which means that the only thing you have to do is to use the client-side part of the Channel plugin. This allows you to have real-time collaboration in your applications without any backend configuration!

Install

In a browser

Include `joint.com.channel.js` file into your HTML:

```
<script src="joint.com.channel.js"></script>
```

On a server

Make sure you have NodeJS installed. NodeJS installation already comes with NPM (Node Package Manager) that we'll use to install all the dependencies. Go to the Rappid package root directory (where the `package.json` file is located) and run:

```
npm install
```

This will install all the necessary dependencies needed by the server side Channel.

Usage

In a browser

In a browser, the only thing necessary is to instantiate the `joint.com.Channel` constructor function and pass it a **`url`** of the server-side channel and the **`graph`** we want to be synchronising:

```
var graph = new joint.dia.Graph;
var channel = new joint.com.Channel({ url: 'ws://localhost:4141', graph: graph });
```

Note we're using the *ws* protocol as the Channel plugin uses WebSockets for real-time communication between server and client. If you plan to take advantage of us maintaining the NodeJS server for you, you'll get a dedicated URL that you can use in your channel and you can skip the rest of the *Usage* section.

On a server

On a server, we use the exact same `joint.com.Channel` constructor function but this time we pass it a `port` we'd like the WebSocket server to be listening on:

```
var joint = require('./index');
var Channel = require('./plugins/com/Channel/joint.com.Channel').Channel;

var graph = new joint.dia.Graph;
var channel = new Channel({ port: 4141, graph: graph });
```

In this example, we assume the server side script is located in the root directory of the Rappid package so that the paths used in the `require()`s are valid.

Rooms

The problem with the previous approach is that if you have more clients that should be divided into groups (rooms) each room sharing its own JointJS graph, you'd have to run as many servers as you have rooms each on a different port. This limits the number of rooms you can serve by the number of ports that your OS allows you to allocate. In general, this is not an ideal approach. Instead, we'd rather have just one server running that is able to maintain more graphs and route the communication between clients based on the room they're in. This is exactly what `joint.com.ChannelHub` does.

ChannelHub runs a WebSocket server and handles client connections. When a client wants to get connected, ChannelHub decides to which Channel it should route the requests to. In this case, the only server running is the ChannelHub. The server side Channels don't create their servers internally. Your server side script changes to something like this:

```
var channels = {};
var channelHub = new ChannelHub({ port: 4141 });
channelHub.route(function(req, callback) {

    var query = JSON.parse(req.query.query);
    var channel = channels[query.room] || null;

    if (!channel) {
       channel = channels[query.room] = new joint.com.Channel({ graph: new joint.dia.Graph });
    }

    // If an error occurred, call the callback function with the error as the first argument.
    // callback(new Error('Some error has occurred.'));

    // Otherwise, call the callback function with the channel as the second argument.
    callback(null, channel);
});
```

Notice how we instantiated the Channel. In this case, we don't pass neither a `url` nor a `port`. This defines a Channel that is not a client (no `url`) and it also does not create a WebSocket server internally (no `port`). Instead, it is the ChannelHub that handles requests and decides to which Channel it should route messages to.

Also, your client side initialization of Channel changes a little bit in order to enable rooms:

```
var channel = new joint.com.Channel({ url: 'ws://localhost:4141', graph: graph, query: { room: 'A' } });
```

We pass the `query` object that contains some data that help the ChannelHub decide to which channel to route the messages to. In this case, we use `room: 'A'` and on the server, we saw `query.room` which we base our decision on. This data can be arbitrary. Feel free to use any kind of decision making mechanism.

Configuration

Channel

- `url` - URL of the server-side channel to which we want the client to be connected to. If this parameter is passed, the channel is considered to be a client. (Note that you can have a client running in NodeJS too.)
- `port` - port number of the channel server. If this parameter is passed, the channel is considered to be a server.
- `reconnectInterval` - if a client looses a connection to the server, it tries to reconnect every `reconnectInterval` milliseconds. The default is 10000ms = 10s. This settings makes sense only for clients.
- `healthCheckInterval` - Interval in milliseconds in which a health check is performed on all the sites of a server. Each health check decreases the `ttl` value of a site if the site socket is disconnected. If the site socket gets connected again, its `ttl` returns to its original value. The default is a health check performed every 1 minute. Considering the default `ttl` is 60, this means that if every minute of an hour a site didn't respond (its socket was closed) then such a site is considered dead (not responding anymore) and therefore removed from the channel register. The reason for health checks is the following. Consider you have ChannelHub running your server. This ChannelHub contains couple of Channels each representing a different room. Consider there is 3 clients connected to one room simultaneously editing the same graph. One of the clients doesn't want to work anymore and shuts down his browser or closes the tab with your application. Now the channel of the room still keeps this client in its register as the client might reconnect anytime (the user closed his browser, went for a coffee and came back in ten minutes.) The `healthCheckInterval` and `ttl` allows you to configure the maximum waiting time in which the channel should still keep track of the client. This setting makes sense only for server.
- `ttl` - time-to-live of a site. Explained above. The actual time to live of a site in milliseconds is `ttl * healthCheckInterval`. The default `ttl` is 60. This setting makes sense only for server.
- `query` - An object that is send with the initial handshake of a client with a server. This can be an arbitrary JSON object. The `query` object is especially useful when working with rooms (see above for documentation on using the Channel plugin with rooms). This setting makes sense only for clients.
- `debugLevel` - the verbosity of debugging logs. The default 0 meaning no debug messages are printed out. Set to a higher number if you want to get some inside on how the Channel works.

ChannelHub

- `port` - port number of the channelHub server.
- `route(req)` - this is actually not an option that you pass to the ChannelHub constructor function but a method that you define directly on the channelHub object. The `route(req)` method receives a connect request from a client and must return a channel object that is supposed to further communicate with that client. The `req` parameter contains the request object. The `req.query.query` contains the `query` object that the client sent during handshake (see above the Channel configuration section). Note that `req.query.query` is a JSON string so you should convert it to an object with `JSON.parse()` before accessing its inner properties. Use this object to determine what channel to return.

Rappid - dia

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 */
}
```

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

reset

Function `reset()` clears both undoStack and redoStack.

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.

dia.GraphUtils

dia.GraphUtils adds additional methods to the `joint.dia.Graph`. Currently, it adds only two methods: `constructTree()` and `shortestpath()`.

Installation

Include `joint.dia.graphUtils.js` file to your HTML:

```
<script src="joint.dia.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>
```

This will add more methods to the `joint.dia.Graph` object:

Methods

constructTree(parent, opt, parentElement)	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: `var cells = graph.constructTree({ name: 'my top, children: [{ name: 'my child 1' }, { name: 'my child 2' }] }, { makeElement: function(node) { return new joint.shapes.basic.Rect({ size: { width: 80, height: 30 }, attrs: { text: { text: node.name } } }); }, makeLink: function(parentElement, childElement) { return new joint.dia.Link({ source: { id: parentElement.id }, target: { id: childElement.id } }); } }); graph.addCells(cells);` This method is used in the JavaScript AST visualizer demo.
shortestPath(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.

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 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.vaidator.js"></script>
```

Creating Validator

```
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 });
var validator = new joint.dia.Validator({ commandManager: commandManager });
validator.validate('remove',
    function(err, command, next) {
        if (command.data.type === 'basic.Rect') 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.

Default is `true`.

validate

Function `validate(action [, callback]*)` registers callbacks for a given action.

The validator listens on 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 `basic.Rect` 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":"basic.Rect", // type of the cell

    // change:attribute events have always 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 `previous` and `next` object empty. 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":"basic.Circle",

    //empty attributes
```

```
    "previous":{},
    "next":{},

    // all cells attributes
    "attributes": {
      "type":"basic.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 `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.

Rappid - format

format.GEXF

This plugins 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 `xmlString` is an XML string representing a graph in GEXF format, `makeElement` is a function that takes an object with `id`, `width`, `height` and `label` properties and returns a JointJS shape for this node. `makeLink` is a function that takes an object with `source` and `target` 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

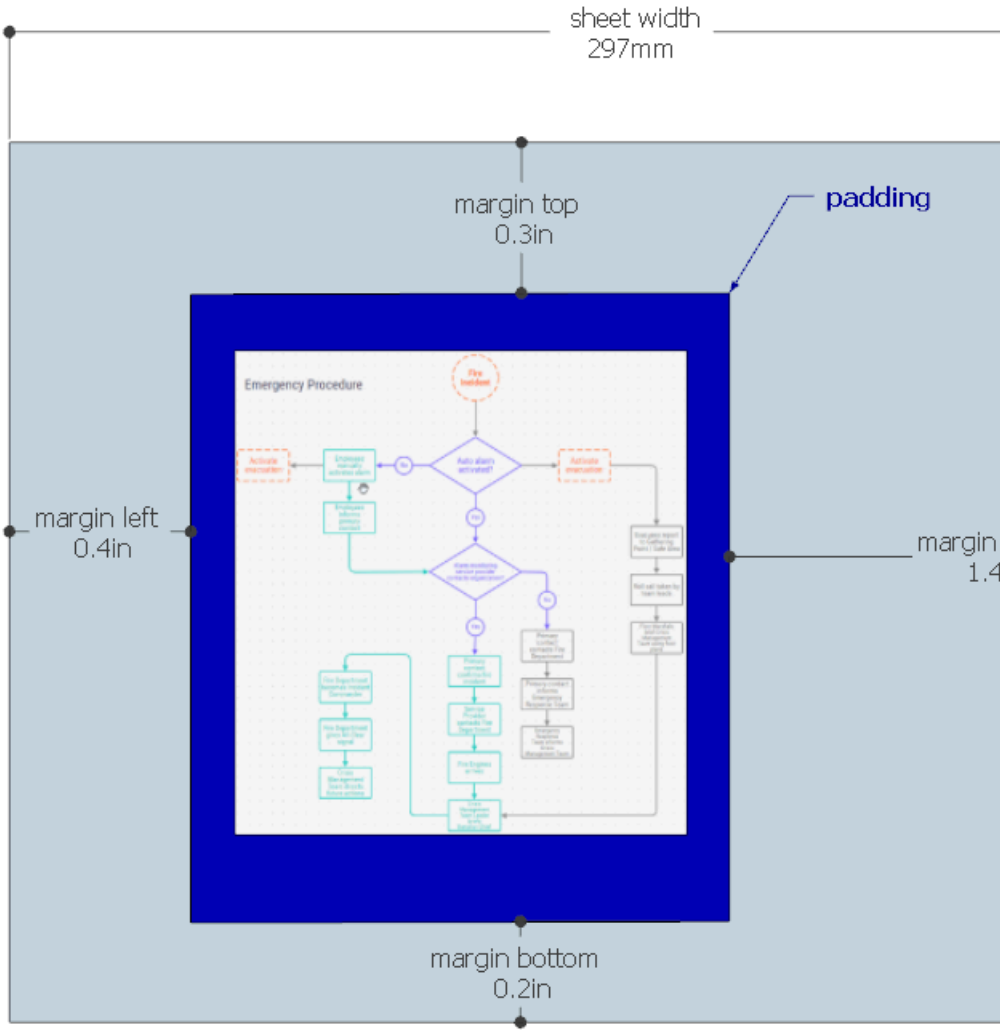
```
var cells = joint.format.gexf.toCellsArray(
    someGEXF_XML_String,
    function makeElement(attrs) {
        return new joint.shapes.basic.Rect({
           id: attrs.id,
           size: { width: attrs.width, height: attrs.height },
           attrs: { text: { text: attrs.label } }
        });
    },
    function makeLink(attrs) {
        return new joint.dia.Link({
            source: { id: attrs.source },
            target: { id: attrs.target }
        });
    }
);

graph.resetCells(cells);
```

format.Print

The Print plugin extends the `joint.dia.Paper` with the following method that, when called, prepares the paper for printing and initiates printing using the browser print wizard:

- `print([options])` - prints the paper using the browser print dialog box.

Where `options` is an object with the following possible values:

sheet	*object*	Optional. Along with the `sheetUnit`, `margin` and `marginUnit` it defines the output pape object: `{ width: [number], height: [number] }`. It defaults to size of the A4 - 210x297. orientation - *landscape* or *portrait*. For example A4 - landscape would be set as `{ width` Example: `var 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 }`  Please note, the `padding` is ignored when the `area` is defined.
sheetUnit	*string*	Optional. Size unit for the `sheet` option. Supported units are `mm`, `cm`, `in`, `pt`, `pc`. It de
area	*object*	Optional. Custom area to be printed. Defined as `{width: [number]: height: [number],` `area` is not defined, the whole graph is printed.

ready	*function*	Optional. This callback function is triggered when the print content is ready for the actu update the print content element (e.g. set some additional css styles) and pass this thro print process. The prepared print content is a HTML element (multiple elements in case print process pass falsy value into the `readyToPrint` callback (see the example below). T 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}}} opt` `  */` `  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);` `  },` `}` • `printAreaElements` - array of ready-to-print elements. • `readyToPrint` - pass `printAreaElements` into this callback to start printing. Pass `f` • `opt` - additional useful options from the print plugin. `{` `  // sheet size convert to pixels` `  sheetSizePx: {` `  width: number,` `  height: number` `  }` `}`
poster	*boolean* | *object*	Optional. Splits the area to be printed to pieces. Each piece will be printed on a separat as `{ rows: [number], columns: [number] }` or as `{ width: [number], height: [number` "table-style". It treats the print area to be a table and it divides it to defined amount of r variant allows you to define the `width` and `height` of the piece. It defaults to `false`.
margin	*number* | *object*	Optional. The space on all sides of the paper area to be printed. The `margin` could be a properties for finer grained control of the margin on each side: `{ left: [number], top: [number] }`
marginUnit	*string*	Optional. Size unit for the `margin` option. Supported units are `mm`, `cm`, `in`, `pt`, `pc`. It d
padding	*number* | *object*	Optional. This is applicable only if the option `area` is not set - when the whole graph is p each side of the graph. The `padding` could be also an object with the following properti padding on each side: `{ left: [number], top: [number], right: [number], bottom: [r`

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

```
$('#print-btn').click(function() {

    paper.print({
        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)
        }
    });

});
```

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.

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 data URI representing the content of the paper as the first parameter.

toJPEG(callback [, options])	Executes the callback function with a JPEG data URI string representing the content of the paper as the first parameter.
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 available options are:

width	*number*	Set the width of the image in pixels.
height	*number*	Set the height of the image in pixels.
size	*number*	Size multiplicator for generating raster images in a higher resolution. It expects a string in the form of `[Number] + 'x'` describing how many times the image width and height should be multipicated. It defaults to `'1x'`.
backgroundColor	*string*	The image background color.
padding	*number*	The space between the image border and the image content. It can also be an object with `left`, `top`, `right` and `bottom` properties
type	*string*	The standard MIME type for the image format to return (only applies to `toDataURL()`).
quality	*number*	The quality level of a JPEG image compression in the range of 0.0 to 1.0 (only applies to `toJPEG()`).
area	*g.Rect*	An area, that the resulting raster image should display. The value is an object, which contains `x`, `y`, `width` and `height`, describing the origin and the size of a rectangular area on the paper in the local coordinates. It defaults to the paper content bounding box - `paper.getContentBBox()` transformed into the local coordinates. `// Export selected elements only and add a padding` `var padding = 20;` `paper.toPNG(callback, {` `area:` `graph.getCellsBBox(selection.collection.toArray()).inflate(` `padding)` `});`
useComputedStyles	*boolean*	The same behaviour as useComputedStyles in `paper.prototype.toSVG()`.
stylesheet	*boolean*	The same behaviour as stylesheet in `paper.prototype.toSVG()`.

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 Rappid 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
});
```

format.SVG

This plugins 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.
- `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) {
    sendToServer(svgString);
    offerForDownload(svgString);
});
```

Configuration

The `toSVG(callback[, opt])` method takes an option object `opt` with the following parameters:

preserveDimensions	*boolean | object*	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 `preserveDimensions` to `true`. An object with `width` and `height` properties can be also used here if you need to define the export size explicitely.
area	*g.Rect*	An area, that the resulting SVG should display. The value is an object, which contains `x`, `y`, `width` and `height`, describing the origin and the size of a rectangular area on the paper in the local coordinates. It defaults to the paper content bounding box - `paper.getContentBBox()` transformed into the local coordinates. `// Export selected elements only and add a padding` `var padding = 20;` `paper.toSVG(callback, {` `area:` `graph.getCellsBBox(selection.collection.toArray()).inflate(padding)` `});`
convertImagesToDataUris	*boolean*	Converts all contained images into Data URI format. It defaults to `false`.
useComputedStyles	*boolean*	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. `// External stylesheet.css` `.joint-link .connection { fill: none }` `.joint-element rect { stroke: red }` 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. `// Store SVG Attributes right on the DOM elements` `link.attr('.connection', { fill: 'none' });` `element.attr('rect', { stroke: 'red' });` `paper.toSVG(callback, { useComputedStyles: false });`
stylesheet	*string*	A stylesheet (as string) can be provided and appended to the resulting SVG export.

```
paper.toSVG(callback, {
  stylesheet: [
    '.joint-link .connection { fill: none }',
    '.joint-element rect { stroke: red }'
  ].join('')
});
```

It defaults to `true`.

Rappid - layout

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-layouts 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 tights 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,
    width: 600, height: 400,
    gravityCenter: { x: 300, y: 200 },
    charge: 180,
    linkDistance: 30
});

graphLayout.start();

_.each(_.range(100), function() { graphLayout.step(); });
```

Note: It is recommended to use the requestAnimationFrame for stepping the layout.

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.addCells([
  new joint.shapes.basic.Rect({ size: { width: 80, height: 50 }}),
  new joint.shapes.basic.Rect({ size: { width: 50, height: 50 }}),
  new joint.shapes.basic.Circle({ size: { width: 80, height: 50 }}),
  new joint.shapes.basic.Circle({ size: { width: 50, height: 50 }})
]);

// Layout the entire graph
joint.layout.GridLayout.layout(graph, {
  columns: 2,
  columnWidth: 100,
  rowHeight: 70
});

// Layout the circles with minimal resulting `y` coordinate equals 100.
var circles = graph.getElements().filter(function(el) {
    return el instanceof joint.shapes.basic.Circle;
});
joint.layout.GridLayout.layout(circles, {
    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'`.
- **`dx`** - Shifts the elements horizontally by a given amount. It defaults to `0`.

- `dy` - Shifts the elements vertically by a given amount. It defaults to `0`.
- `centre` - Positions the elements in the centre of a grid cell. It defaults to `true`.
- `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 });
```

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.

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 `direction` defines how an element connects its parent. For instance `element.set('direction', 'R')` will appear on the right side to the parent element. Note that, this option is only used for nodes that do not have a `direction` property explicitelly 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: • `children` is an array of siblings • `parent` is the parent of the `children` or `null`, when `children` are roots • `opt` is the option object passed to `layout()` method when calling the layout. `// 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');` `  });` `}` No element is filtered out by default.

updatePosition(element, position, opt)	A function responsible for setting the resulting position on the elements. It calls `element.set('position', position, opt);` by default. It can be used for positioning the elements in an animated manner. For instance: `element.transition('position', position, { delay: 300, valueFunction: joint.util.interpolate.object, });`
updateVertices(link, vertices, opt)	A function responsible for setting the resulting vertices on the links. It calls `link.set('vertices', vertices, opt);` by default. If the option is defined as `null` no vertices will be set by the layout.
attributeNames	An object that maps element attributes 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 `{ siblingRank: 'index' }` will make the layout look for the `index` attribute instead of `siblingRank` when trying to figure out the order of siblings.

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 `direction` in the TreeLayout constructor function configuration.
siblingRank	The index of the element among its siblings.
offset	The additinal distance from the parent (the real distance is a sum of `parentGap` - defined globally in the TreeLayout constructor function configuration. - and the `offset`).
margin	Extends the area occupied by the element on both sibling sides. `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 }`
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 `siblingRank`.

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 `siblingRank`.
firstChildGap	The distance between the element and the first child. It's applicable only for diagonal layouts (`'TL'`, `'TR'`, `'BL'`, `'BR'`). It overides the global `firstChildGap` 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 `opt`, which will be passed down to all operations on the graph made by the layout call. e.g. `treeLayout.layout({ myFlag: true })` will internally set an element position as `element.position(x, y, { myFlag: true });`.

Rappid - shapes

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 (sub-process handling).
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.) `'.body'` (rect), `'.header'` (rect), `'.lane-body'` (rect), `'.lane-header'` (rect), `'.blackbox-label'` (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 can be
targeted
```

```
                                // 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');
```

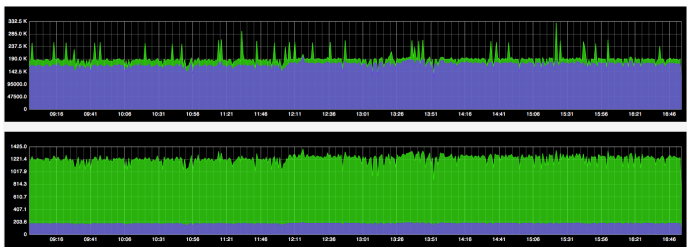
Sub-process handling

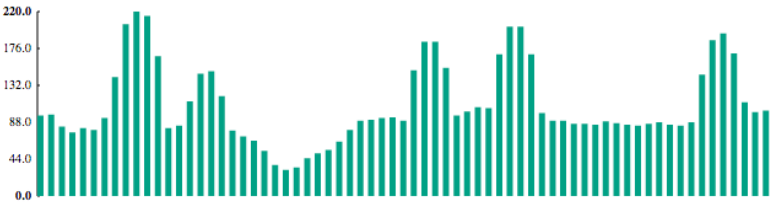
The `subProcess` value is stored as a data attribute on the sub-process icon (SVG path) element. An interaction with the sub-process icon can be therefore caught and handled on the `joint.dia.Paper`. In the example bellow the model's `subProcess` value is shown after an user doubleclicks the icon.

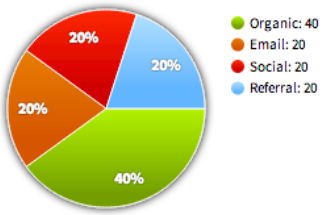
```
conversation.set('subProcess', 'My sub-process ID');

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'
```

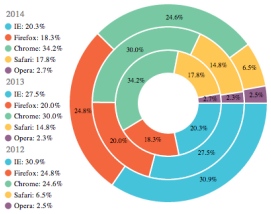
shapes.chart

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/Rappid 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/Rappid, 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

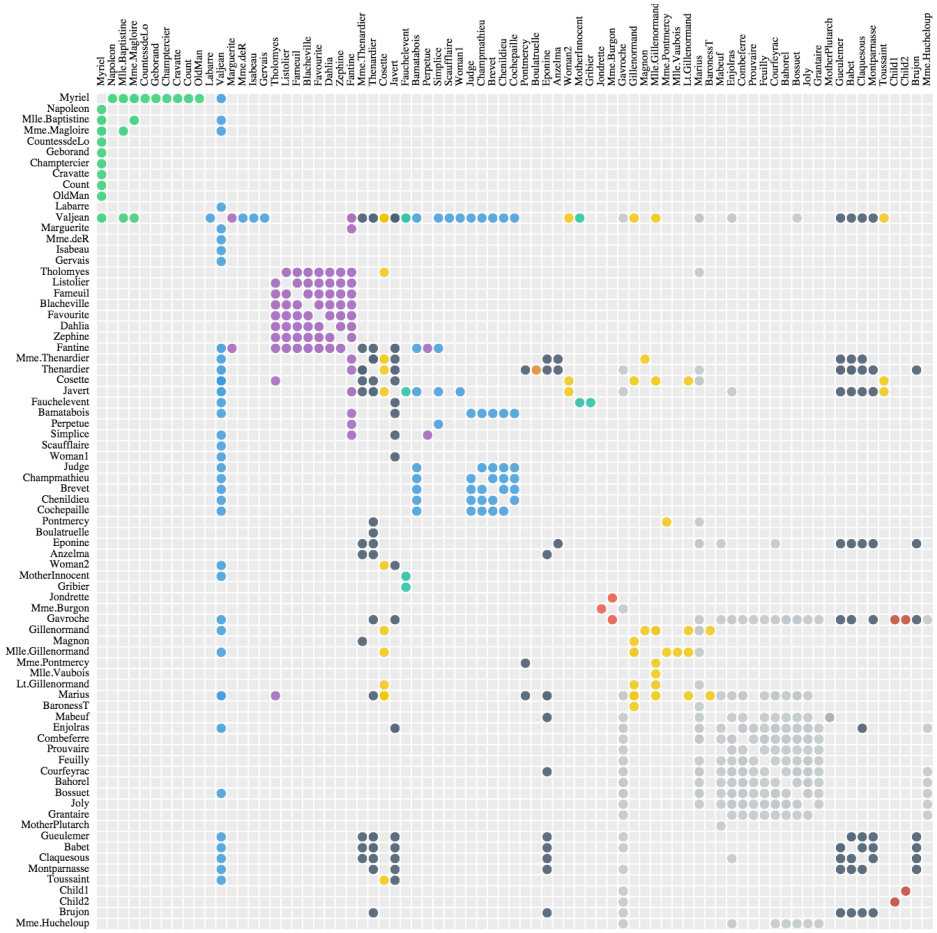
As `chart.Knob` inherits from `chart.Pie` all of the chart.Pie events apply here as well.

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 Miserables 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/Rappid, 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 serie 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 sections 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 lenged 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 serie 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 by 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 degre 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 lable 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 `slice` to the serie identified by `serieIndex`. `opt` can be an arbitrary object that is passed to the `onchange` handlers of the JointJS element.
editSlice(slice, sliceIndex, serieIndex [, opt])	Replace a slice in the pie by a new `slice`. The serie is identified by `serieIndex` in which the slice at index `sliceIndex` will be replaced. `opt` can be an arbitrary object that is passed to the `onchange` 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 `onClickEffect` 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 `onHoverEffect` 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 `slice` and `event` objects.
mouseover	Triggered when the user hovers over a slice. The handler is passed `slice` and `event` objects.
mouseout	Triggered when the user leaves a slice. The handler is passed `slice` and `event` objects.
click	Triggered when the user clicks a slice. The handler is passed `slice` and `event` 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

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

The Channel plugin supports real-time data plotting out-of-the-box. 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 choses 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 twoaxes 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-rx'` - the top x radius of the bars. Together with the `top-ry` value, this allows you to define border radius for the bars.
  - `'top-ry'` - 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 `rx` and `ry` 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. `position` is a string that can be one of `'n'`, `'nw'`, `'w'`, `'sw'`, `'s'`, `'se'`, `'e'`, `'ne'`, `'nne'`, `'nn'`, `'nnw'`, `'nnww'`, `'nww'`, `'ww'`, `'sww'`, `'ssww'`, `'ssw'`, `'ss'`, `'sse'`, `'ssee'`, `'see'`, `'ee'`, `'nee'`, `'nnee'`. See the Legend section for more info. `opt.padding` can be used to set the distance between the legend and the chart edges. Default is `10` px.
addPoint(p, serieName [, opt])	Append point `p` to the serie identified by `serieName`. If `opt.maxLen` is set and the number of points in the serie is higher than `maxLen`, shift the data in the serie.
lastPoint(serieName)	Return the last point in the serie identified by `serieName`.
firstPoint(serieName)	Return the first point in the serie identified by `serieName`.

chart.PlotView

To access the view for a plot chart, use: `paper.findViewByModel(chart)`.

renderPoint(p, serie)	Render a point at the coordinates defined in `p`. `serie` is the object from the `chart.Plot` `series` array and defines the serie for which the point should be rendered. Return the SVG DOM element for the point. See the Tooltips section on why this method is useful.
getSerieColor(serieName)	Return the color used to draw the serie identified by `serieName`.
hideSerie(serieName)	Hide serie identified by `serieName`.
showSerie(serieName)	Show serie identified by `serieName`.
closestPoints(x)	Return an array of the closest points from all the series in the chart for a given `x` 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.
```

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
body	*SVGRectElement*	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:

Custom presentation attributes are wholly 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

Configuration properties are wholly 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

Functions are wholly 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	*SVGRectElement*	Rectangular body of the shape
header	*SVGRectElement*	Rectangular header of the shape, positioned above shape's body
headerLabel	*SVGTextElement*	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');
borderedRecord.attr('itemLabels', { fontSize: 12, fontFamily: 'sans-serif' });
headeredRecord.attr('itemLabels_1/magnet', true);
headeredRecord.addTo(graph);
```

Custom presentation attributes:

Custom presentation attributes are wholly 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

Configuration properties are wholly 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

Functions are wholly inherited from the `joint.standard.Record` shape.

shapes.standard.Record

An object that displays structured/hierarchical data.

Supported `attrs` properties

Selector	Node	Description
root	*SVGGElement*	Container of all nodes
itemBodies	*group of SVGRectElements*	All item bodies across all columns. The node(s) can have a custom attribute itemHighlight.
itemBodies_[group_index] itemBodies_[group_name]	*group of SVGRectElements*	Item bodies in the given group (column). *For example, item bodies in the first column are accessed as* `itemBodies_0`. *Alternatively, item bodies in a group named* `headers` *are accessed as* `itemBodies_headers`. The node(s) can have a custom attribute itemHighlight.
itemBody_[item_id]	*SVGRectElement*	Body of the given item. *For example, item body for an item with id* `my_item` *is accessed as* `itemBody_my_item`. The node can have a custom attribute itemHighlight.
itemLabels	*group of SVGTextElements*	All item labels across all columns The node(s) can have custom attributes itemHighlight and itemText.
itemLabels_[group_index] itemLabels_[group_name]	*group of SVGTextElements*	Item labels in the given group (column). *For example, item labels in the first column are accessed as* `itemLabels_0`. *Alternatively, item labels in a group named* `headers` *are accessed as* `itemLabels_headers`. The node(s) can have custom attributes itemHighlight and itemText.
itemLabel_[item_id]	*SVGTextElement*	Label of the given item. *For example, item label for an item with id* `my_item` *is accessed as* `itemLabel_my_item`. The node can have custom attributes itemHighlight and itemText.
bodiesGroups	*group of SVGGElements*	All groups of item bodies

labelsGroups	*group of SVGGElements*	All groups of item labels
forksGroups	*group of SVGGElements*	All groups of hierarchy fork lines
buttonsGroups	*group of SVGGElements*	All groups of collapse buttons
iconsGroups	*group of SVGGElements*	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:

<table>
<tr><th>Attribute</th><th>Selectors</th><th>Description</th></tr>
<tr><td>itemHighlight</td><td>itemBodies
itemBodies_[group_index]
itemBodies_[group_name]
itemBody_[item_id]
itemLabels
itemLabels_[group_index]
itemLabels_[group_name]
itemLabel_[item_id]</td><td>Object with SVG attributes. Applied on the selected node(s) when the item is highlighted.</td></tr>
<tr><td>itemText</td><td>itemLabels
itemLabels_[group_index]
itemLabels_[group_name]
itemLabel_[item_id]</td><td>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 (<code>true</code> by default)</td></tr></table></td></tr>
</table>

		label	boolean | string	Enable label ellipsis (`true` by default)

```
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	*object[][]*	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:

Property	Type	Description
id	string	(*Required. Must be unique.*) Item's identifier in the DOM and in JointJS. *Use the* `cellView.findAttribute('item-id', id)` *function to find an item by given* `id`.
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[]	Selector 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 `'headers'` as the group name would make the item accessible via the `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?		
padding	object	Padding within the Record object		

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.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`.

Rappid - storage

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 defered 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.

```
var graph = new joint.dia.Graph;
(new joint.shapes.basic.Rect).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 `doc` into the `collection`. Optionally pass a `callback(err, doc)`. If the document `doc` didn't have an `id` property, one is automatically created and will be part of the `doc` object returned in the callback.
find(collection, query, callback)	Find a document in `collection`. `query` can currently be either empty in which case all the documents from the `collection` are returned or it can contain `id` of a document in which case only a document with that `id` is returned. `callback` signature is: `callback(err, docs)`.
remove(collection, query [, callback])	Remove a document from the `collection`. `query` can currently be either empty in which case all the documents from the `collection` are removed or it can contain an `id` of the document to be removed. `callback` signature is: `callback(err)`.

Rappid - ui

ui

Rappid 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 Rappid UI plugins for every theme (you can also open it in a separate window) :

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 });>
```

link	Set of attributes to be set on each copied link on every pasteCells() call. It is defined as an object. e.g. `{ z: 1 }`.
translate	An increment that is added to the pasted elements position on every `pasteCells()` call. It can be either a number or an object with `dx` and `dy` attributes. It defaults to `{ dx: 20, dy: 20 }`.
useLocalStorage	Set to `false` if you don't want to use `window.localStorage` for storing copied cells. It defaults to `true`.

Clipboard API

copyElements(collection, graph, opt)	Store the elements from the `collection` along with all links, which are connected by both source and target to these elements. The options `opt` passed as a parameter can override the global options.
cutElements(collection, graph, opt)	Similar to `copyElements()` but it also removes all copied cells from the `graph`.
pasteCells(graph, opt)	Add copies of stored cells to the `graph`. The method can be called multiple times and it always produces new copies.
clear()	Clear all stored cells.
isEmpty()	Return `true` when there is no cells stored on the Clipboard instance. Returns `false` otherwise.

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.

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:
'
AAAGXRFWHRTb2Z0d2FyZQBBZG9iZSBJbWFnZVJlYWR5ccllPAAABPxJREFUeNrsnc9rU1kUx+/LS2LTBtI4VMiqjYjiQgQ3I
oIUuhvGUmYzUBgV8SdWLS7sooLgxm0XgjKg0P+gMDC7ggjDzGwGOt1psS2WBptioiYVmrbxnviiSc17eaPvvdzz7vcLlyTk5
vVyPtzzzv3mtDWq1arYrVwuNyAfxuUYkaNfQEFoWY4ZOaYymczS7jeNRlASUC9NlOMs4tZRTdNGkcCKX4GyID2V4yjipITm5
Bisw4oAkrIiFk8tNp9AWekOkNSENVVLfaurq1Q4LCImSisbtao7R5mmWTQMYxPx8l6yRohvb2/3tpk2HrVKcFtA6XT6WSwWK
yOk/qlSqfQUCoVTDsBGIk7nJEAKRhRjirXDlP6I024CpGBhUczt3rcFhXtS8HKKeQTh4aHot35Q3vyObW1tpdvNSyQSi8lk8
iWu1yFQtGg59rWbJyuZNVzv+4XUx0QABVCa68+/0mJi8oj4/Y99Xlwuioj6BOn8lSHx4UOs9vr0j2vYUYrJ/G++6zOkRKIih
n9aQepTzV14viB6Lo5lP0N68mhWnDxRACjFIP1w9YYwNjZMryEBlNeQSmVR7e7e9hoSib44rLasMqLRtb6+vlm7D5ZKpf3yc
Nfj4qCY2tnZ2dM2t5tmSY4NbtejexKlO9pJBOndbw/ym4cPtd0ArRyMfD4/ZHeo/uaqz61N4vTDGxWPx9dSqdQ8q+tRdXfhG
hUOtXRnPH44u3nwwDE/HAykPi9KcB/uSQDFEBJAMYEEUEwgAdT3OA4BQgIoBRwHgGLiOAAUE8fBrXzvmaATPR0W3TgEdPhU5
Xo1x+Hy9WbHQR5m5QnZk/XJOa8DAeW2h0A3x8Ht+pD6GJfgAMUcEkDZQSLdmfhbFUgAZQfp9q1/xK+jKyotUXtQTY5DHdLYl
ZeqrVNrUE2Og8KQtAbV5DgoDklbUI2OAwdIJO16Jhp7HOh1eXLizcbPw1tBrw89E24dB2snSUjZTqwPPRP/pwRXPN3pB4o5J
D1AhQBS+EGFBFKoQXFxHLQGxclx0BYUN8dBS1CR9yWRuneflePgVqHpmTCKb83kxWvZyOJSV4PjkJUn0KwK62txgNawZyKfj
4vRc0Pi1UpXpx0H9EzYRzAuRn4hSL1hS3fhAaUJJN6gNILEF5RmkFiCoupON0jsQNE5KXnmQlY3SKxAEaS9V2+K+jlJJ0hsQ
NUhxV4sCB0h1VK+6j0TNcdBprtdjkPgPQ7omQiZ46Bfz4SGJTg/UIDEABQgMQAFSOqD0tVxYAVKZ8eBDSjdHQcWoOA4uFfHe
ia49Tjo2TMRYschPD0TKMEZgAIkBqAAiQEoQFIfFBwHBqDgODAABceBASg4Dj7cQrzumQhrj0O4eibgODDomUAJzuAeBUhMQ
F0aOw5IHECVy3FA8lfe/HvXu5P/1h4V+husANVKAMTXmYAACqnPD9EJ3M3hzm0PgW7XCwyU29/6wPWQ+nCPghQAVa1W4whPs
HKKuS0oeUPsrVQqPQhfMKJYU8ydQC3bvVkoFE4BVjCQKNYOU5ap6puR46bdrlpfXx82TbNoGMYmQupPunPaSZZm6BveAflkE
SFTWtlIJpNZkk+mEQtlNU2M6sXEuBxziIlymrPYfKr6JLGifBgELOUgDVpsvpTnDbCQBhVId42QSIasOr6alcvlBqwtNyJHP
+IWiJatCnzKqhua1BIUpJ4+CjAAVnYzLhKE5pcAAAAASUVORK5CYII=' },
        { 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: **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** — `true` 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 `ui.ColorPalette` widget looks up the selected item from the `options` array (by using the `selected` boolean property). If nothing is selected, the first item in the `options` array is selected by default.
keyboardNavigation	`true` (the default) if you want the `ui.ColorPalette` to provide a keyboard navigation (up/down/left/right/enter/escape keys are supported).
selectBoxOptionsClass	When the color palette is opened, the `ui.ColorPalette` generates an HTML container that contains all the items. This container is then appended to the document body (or `target` if provided). Sometimes, you want to provide custom styling to this color palette container. `selectBoxOptionsClass` 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	`ui.ColorPalette` 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 `target` 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 `selected` index to `-1` and the `placeholder` to any HTML you want to display in the selected item window. This placeholder has the `selected-box-placeholder` 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 `el` property that points to the container HTML element and append it anywhere in the DOM (e.g. `$(document.body).append(colorPalette.el)`).
getSelection()	Get the current selection. The selection points to one of the items from the `options` array.
getSelectionValue()	Get the current selection value (the color).
getSelectionIndex()	Get the index in the `options` array of the item that is currently selected.
select(index, opt)	Select an item. `index` is the index to the `options` array. `opt` is optional and can be an arbitrary object that will be passed to the `option:select` event handler.
selectByValue(color)	Select an item by color.
isOpen()	Returns `true` if the color palette is currently opened. `false` 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 `handler` function that will be called whenever `event` is triggered. The optional `context` is an object that will be the context of the handler function (the `this`).

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 `options` 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 `options` array. If you selected the item with the `select(index, opt)` method, the third argument to the event handler will be your `opt` 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.

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 to 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:

<table>
<tr><td>tools</td><td>An array of tools. Each tool is an object that can have the following properties:<table>
<tr><td>action</td><td>The name of the action. This action is then triggered on the context toolbar object so that you can react on it later.</td></tr>
<tr><td>content</td><td>The content of the button. It can be a text or an arbitrary HTML.</td></tr>
<tr><td>icon</td><td>Optionally, instead of passing a <code>content</code>, you can just pass a path to an image.</td></tr>
<tr><td>attrs</td><td>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>.</td></tr>
</table></td></tr>
<tr><td>target</td><td>The target element (either HTML or SVG).</td></tr>
<tr><td>padding</td><td>The gap between the target element an the context toolbar. Default is <code>20</code>.</td></tr>
<tr><td>autoClose</td><td>Determines if the context toolbar should be closed if the user clicked outside of the area of the context toolbar. Default is <code>true</code>.</td></tr>
<tr><td>vertical</td><td>When set to <code>true</code> the toolbar arrange tools from top to bottom instead of left to right. Default is <code>false</code>.</td></tr>
</table>

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 `autoClose` option to `false`).
on(event, handler [, context])	Register a `handler` function that will be called whenever `event` is triggered. The optional `context` is an object that will be the context of the handler function (the `this`).
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 you `target` 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.

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: `'info'`, `'alert'`, `'warning'`, `'success'` and `'neutral'`.
buttons	Buttons of the dialog. `buttons` is an array of objects of the form `{ action: String, content: String [, position: String] }`.

action	The name of the action the button represents. When the button is clicked, the dialog object triggers an event `action:[name]`, allowing you to react.
content	The label of the button (it can be HTML, as well).
position	Optional. It can be `'left'`, `'right'`, or `'center'`. If `position` is not specified, it is considered to be `'right'`. If multiple buttons are provided with the same `position`, they are placed left-to-right according to the order in which they were provided.

draggable	If `draggable` is `true`, the dialog window will be draggable by the user. It defaults to `false`.
closeButton	If `closeButton` is `false`, 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 `true`.
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 `false`, 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 `true`.
inlined	If `true`, the dialog window will be inlined in a container that you can pass to the `open()` method.

API

open()	Open the dialog window.
close()	Close the dialog window.
on(event, handler [, context])	Register a `handler` function that will be called whenever `event` is triggered. The optional `context` is an object that will be the context of the handler function (the `this`).

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 `[name]`.

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 `'alert'` (default), `'warning'`, `'success'`, `'neutral'`.
content	The content of the flash message. It can be either a text or an arbitrary HTML.
modal	Set to `true` if you want your flash message to be modal.
closeAnimation	Set to `false` if you don't want the flash message to be closed in an animated fashion. It can also be an object of the form `{ delay: Number, duration: Number }` 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). `opt` can contain any option that you can pass to the `joint.ui.FlashMessage` constructor function (see the Configuration section for details).
on(event, handler [, context])	Register a `handler` function that will be called whenever `event` is triggered. The optional `context` is an object that will be the context of the handler function (the `this`).

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.

ui.FreeTransform

FreeTransfrom plugin creates a control panel above an element giving an 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 `true` if you want the resizing to preserve the aspect ratio of the element. Default is `false`.
allowRotation	Set to `false` if you want the rotating handle to be omitted from the handles. Default is `true`.
minWidth	The minimum width allowed for the element when resizing. Default is `0`.
maxWidth	The maximum width allowed for the element when resizing. Default is `Infinity`.
minHeight	The minimum height allowed for the element when resizing. Default is `0`.
maxHeight	The maximum height allowed for the element when resizing. Default is `Infinity`.
rotateAngleGrid	The steps in angle when rotating the element. Default is `15`.
allowOrthogonalResize	Set to `false` if you only want the four corner handles to be displayed. Default is `true`.

clearAll	if set to `true` (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 `clearAll` to `false` (and make sure to call `remove()` once you don't need a freeTransform anymore)
clearOnBlankPointerdown	if set to `true` (the default value), clear the freeTransform when a user clicks the blank area of the paper.

API

The `joint.ui.FreeTransform` objects provide the following methods:

render()	Render the free transform widget. You should call this right after you instantiate the free transform object.
remove()	Remove the free transform widget.
joint.ui.FreeTransform.clear(paper)	Clear all the free transform widgets from a JointJS paper.

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 `defaultLink` option from the `joint.dia.Paper` 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 `paper` and `graph` options but this is now deprecated as those can always be inferred from `cellView.paper` and `cellView.paper.model` properties.
type	the type of the Halo control panel. The default value is `'surrounding'`. Another possible values are `'pie'` in which case the Halo will be displayed as a pie menu and `'toolbar'` 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 (`"top"|"bottom"|"left"|"right"`), default is `"top"`
loopLinkWidth	the self-loop link width in pixels, default is `40`
rotateAngleGrid	the angle increments the rotate action snaps to, default is `15`
boxContent	a string (text or HTML) or a function (of the form `boxContent(cellView, boxDOMElement)`) that returns an HTML string with the content that will be used in the information box below an element. By default, the box displayes the x, y coordinates and width and height dimensions of the element. If `boxContent` is set to `false` (or an empty string), the information box will be hidden

clearAll	if set to `true` (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 `clearAll` to `false` (and make sure to call `remove()` once you don't need a halo anymore)
clearOnBlankPointerdown	if set to `true` (the default value), clear the halo when a user clicks the blank area of the paper.
useModelGeometry	if set to `true`, the model position and dimensions will be used as a basis for the Halo tools position. By default, this is set to `false` 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 `useModelGeometry` to `true`.
clone	a function with signature `function(cell, opt)`. 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: `function(cell, opt) { return cell.clone().unset('z') }`.
pieSliceAngle	*(only valid for the `'pie'` type of Halo)* The angle of one slice in the pie menu. It defaults to `45`.
pieStartAngleOffset	*(only valid for the `'pie'` type of Halo)* The angular offset of the first handle in the pie menu. It defaults to `0`.
pieIconSize	*(only valid for the `'pie'` type of Halo)* The size in pixels of the icon in the pie menu. It defaults to `14`.
pieToggles	*(only valid for the `'pie'` type of Halo)* 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 `[{ name: 'default', position: 'e' }]`. Each item in the array is an object of the form `{ name: [name of you toggle], position: [one of e/w/s/n] }`. The `name` is passed in the options object in the `state:close` and `state:open` events triggered by the Halo when the pie toggle is clicked.
bbox	A bounding box within which the Halo view will be rendered.
magnet	A function accepting an elementView returning an SVGElement used as a magnet for links created via linking or forking handles. `new joint.dia.Halo({ cellView: elementView, magnet: function(elementView, end) { // where `end` is either "source" (linking and forking) or "target" (forking only) // connect the link directly to a rectangle element as opposed to the element group`

```
    return elementView.el.querySelector('rect') ||
elementView.el;
  }
})
```

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 `"basic.Rect"`. We can do this:

```
.halo[data-type="basic.Rect"] .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. `opt.name` 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. `opt.position` is a string that specifies a position of the tool handle. Possible values are `n`, `nw`, `w`, `sw`, `s`, `se`, `e` and `ne`. `opt.icon` 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 `addHandle()` internally for each item of the `handles` array.
removeHandle(name)	Remove a tool handle named `name` from the Halo.
removeHandles()	Remove all handles from the Halo.
changeHandle(name, opt)	Change a tool handle named `name` in the Halo. The `opt` object can contain the same parameters as with the `addHandle()` method except of the `name` property. `opt` 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 (`callback`) for an `event` See the Halo Events section for the list of events the Halo object triggers.
toggleState([toggleName])	Toggle (open/close) the sate of the Halo (applicable for the *pie* type of Halo). `toggleName` is the `name` of the pie toggle as defined in the `pieToggles` option. If `toggleName` is not passed, all pie menus change the state (the most common usage as there is usually only one pie toggle).
isOpen([toggleName])	Return `true` if the Halo is open (applicable for the *pie* type of Halo). `toggleName` is the `name` of the pie toggle as defined in the `pieToggles` option. If `toggleName` is not passed, return `true` is 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 `paper`. Note that this is a static function in the `joint.ui.Halo` 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 `[name]`.
action: [name]:pointermove	Triggered when the user moves with mouse cursor after a tool handle named `[name]` was mousedowned (touched).
action: [name]:pointerup	Triggered when the user releases his mouse cursor after a tool handle named `[name]` was mousedowned (touched).

action: [name]:contextmenu	Triggered when the user invokes contextmenu on a tool handle named `[name]`.
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: `halo.on('action:link:add', function(link) { if (!link.get('source').id || !link.get('target').id) { link.remove(); } });`
state:open	Triggered when the user opens the Halo (applicable for the *pie* type of Halo). The handler is passed the pie toggle `name` as defined in `pieToggles` option.
state:close	Triggered when the user closes the Halo (applicable for the *pie* type of Halo). The handler is passed the pie toggle `name` as defined in `pieToggles` option.

ui.Inspector

Inspector plugin creates an editor and viewer of a 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 on open/closed groups and restore 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 Inspector Methods section.

```
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 how to create an instance of the inspector. Either using `joint.ui.Inspector.create(container, options)` or directly create an instance with `new joint.ui.Inspector(options)`. The inspector can be configured by the `options` object with the following properties:

cellView	*joint.dia.CellView*	an element or a link view [mandatory]
cell	*joint.dia.Cell*	an arbitrary Backbone model (including cells: elements and links) [mutually exclusive with cellView]. Use either `cellView` or `cell`. If you use `cell`, you can take advantage of the Inspector widget for an arbitrary Backbone model.
inputs	*object*	an object that mimics the structure of a cell model. Only instead of the final values, it contains an input type definition. See below for further explanation.
groups	*object*	an object that contains group identifiers as keys and group options as values. These groups define a

		grouping in the inspector panel. See below for further information.
live	*boolean*	tells the inspector whether it should update the cell properties as a reaction on the `"onchange"` event triggered on the form inputs. `true` by default. However, if you don't want the cell properties to update immediately when the user leaves the input fields but rather update the cell in bulk or as a reaction on user pressing an *Update* button somewhere in your application, set `live` to false and call `inspector.updateCell()` manually when you want to.
multiOpenGroups	*boolean*	`true` (the default) to allow the user to have multiple groups opened at the same time in the Inspector. Set to `false` for the classical "accordion" type of the inspector where only one group can be opened at a time.
validateInput(element, path, type, inspector)	*function*	a function that returns `true` if the input is valid. If the input is invalid, the value is not saved to the model. The function is passed three arguments. `element` is the `<input>` element the user interacted with. `path` is the path to the input field within the inspector (`/` separated). `type` is the type defined in the options object for that field. Finally, `inspector` is a reference to the current inspector instance context. The default function checks the `validity` property of `element`: `function(element, path, type, inspector) { return (element.validity ? element.validity.valid : true); }`
renderFieldContent(options, path, value, inspector)	*function*	if defined, this function can return an HTML, DOM element or jQuery object that will be injected into the inspector in place of the field. In other words, this function allows you to define custom fields. The function is passed four arguments. `options` is the object specified in the `inputs` option of the inspector for a property on the `path` (`/` separated). `value` is the value read from the associated cell at the time of rendering of the field (it also takes into account `defaultValue` and `valueRegExp` from the options object for this field). Finally, `inspector` is a reference to the current inspector instance context.
renderLabel(options, path, inspector)	*function*	if defined, this function can return an HTML, DOM element or jQuery object that will be injected into the inspector in place of the label. In other words, this function allows you to define custom labels. The function is passed three arguments. `options` is the

		object specified in the `inputs` option of the inspector for a property on the `path` (`/` separated). Finally, `inspector` is a reference to the current inspector instance context.
getFieldValue(attribute, type, inspector)	*function*	if defined, this function can return an object of the form `{ value: [value read from a custom field] }` or undefined. The `value` is the current value of a custom field and will be then set on the associated cell. This function is especially useful in combination with the `renderFieldContent()` function and tells the inspector how to read a value from a custom field. The function is passed three arguments. `attribute` is the DOM element container of the inspector field (i.e. the value returned from `renderFieldContent()` if used) and `type` is the type defined in the options object for this field. Finally, `inspector` is a reference to the current inspector instance context.
stateKey(model)	*function*	A function that should return some unique identifier for saving/restoring the state of the groups (ie. opened/closed). The function returns the element `id` by default i.e. each element has its own state. e.g `function(model) { return model.get('type'); }` would store the state per element type. i.e. all `basic.Rect` would share the state.
storeGroupsState	*boolean*	Save the group state when set to `true`. It stores the information which groups are opened and which are closed just before the inspector gets closed. Group state can be restored by the `restoreGroupsState` option when the inspector is created. It defaults to `true`. Applicable only when used with the `create` method.
restoreGroupsState	*boolean*	Restore the group state when set to `true` (if there is any state previously stored). It defaults to `true`. Applicable only when used with the `create` method.

Options properties `storeGroupsState` / `restoreGroupsState` is applicable only if they are passed into the static `create` method. Otherwise you can use API methods `storeGroupsState()` and `restoreGroupsState()` for manipulating with the group states.

As it was mentioned, `inputs` is an object that mimics the properties structure of cell models. Only instead of the final values, it contains the input definition that tells the inspector how it should render the form input associated with the cell property. The options object of inputs can contain the following parameters:

<table>
<tr><td>type</td><td>object</td><td>a type of the input [mandatory]. The supported types are:<table>
<tr><td>number</td><td>creates an HTML 5 number input field. Special properties are <code>min</code> and <code>max</code>.</td></tr>
<tr><td>text</td><td>creates a text input field.</td></tr>
<tr><td>textarea</td><td>creates a textarea.</td></tr>
<tr><td>content-editable</td><td>creates a content editable div (resize automatically as user types)</td></tr>
<tr><td>range</td><td>creates an HTML 5 range input field. Special properties are <code>min</code>, <code>max</code>, <code>step</code> and <code>unit</code>.</td></tr>
<tr><td>color</td><td>creates an HTML 5 color input field.</td></tr>
<tr><td>select</td><td>creates a select box. Special properties are <code>options</code> array that contains options for the select box.</td></tr>
<tr><td>toggle</td><td>creates a toggle (checkbox).</td></tr>
<tr><td>list</td><td>creates a widget for adding/removing items from an array.</td></tr>
<tr><td>object</td><td>creates inputs for properties of an object.</td></tr>
</table></td></tr>
<tr><td>label</td><td>string</td><td>Label for the form input.</td></tr>
<tr><td>group</td><td>string</td><td>Group the input belongs to.</td></tr>
<tr><td>index</td><td>number</td><td>Index of the input within its group.</td></tr>
<tr><td>defaultValue</td><td>object</td><td>a value that will be used in the input field in case the associated cell property is undefined.</td></tr>
<tr><td>valueRegExp</td><td>object</td><td>a regular expression used to extract (and set) a property value on a cell. Use with combination with <code>defaultValue</code> to make sure the inspector does not try to extract something from an undefined value.</td></tr>
<tr><td>overwrite</td><td>boolean</td><td>Should the input value overwrite the current contents of the cell? Default is <code>false</code>, which means that the input attributes are merged with the model's current attributes. Example:<pre><code>// overwrite: false (default)
Model { existingProperty: 'value1' } Input { newProperty:</code></pre></td></tr>
</table>

<table>
<tr><td></td><td></td><td>

```
'value2' } => Model { newProperty: 'value2', existingProperty:
'value1' }
// overwrite: true
Model { existingProperty: 'value1' } Input { newProperty:
'value2' } => Model { newProperty: 'value2' }
```

</td></tr>
<tr><td>options</td><td>object</td><td>
an array of options for a several input types (<code>"select"</code>, <code>"select-box"</code>, <code>"select-button-group"</code>, <code>"color-palette"</code>). Can be defined by:
<table>
<tr><td>content</td><td><code>['option1', 'option2']</code></td></tr>
<tr><td>value/content</td><td>

```
// 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 }]
```

</td></tr>
<tr><td>reference</td><td><code>'path'</code> (where <code>cell</code> property defined by <code>'path'</code> is an array of options) i.e.

```
element.prop('path') // => ['option1',
'option2']
```

</td></tr>
</table>
</td></tr>
<tr><td>min</td><td>number</td><td>
<table>
<tr><td>minimum value of a range input. [specific for the <code>"range"</code> input type]</td></tr>
<tr><td>minimum number of items in an array. [specific for the <code>"list"</code> input type]</td></tr>
</table>
</td></tr>
<tr><td>max</td><td>number</td><td>
<table>
<tr><td>maximum value of a range input. [specific for the <code>"range"</code> input type]</td></tr>
<tr><td>maximum number of items in an array. [specific for the <code>"list"</code> input type]</td></tr>
</table>
</td></tr>
<tr><td>item</td><td>object</td><td>a definition of an item of a list. [specific for the <code>"list"</code> type]</td></tr>
</table>

properties	*object*	an object containing definitions of an object properties. [specific for the `"object"` type]
attrs	*object*	an object of the form `[selector] : { [attributeName] : [attributeValue], ... }`. This object allows you to set arbitrary HTML attributes on the generated HTML for this input. Useful if you want to *mark* certain input fields or store some content in them, for example for display in a tooltip.
when	*object*	an object containing conditions that determine if this input should be shown or hidden based on the values of other inputs. For more information see chapter expressions.
previewMode	*object*	`true` to enable "preview mode" on the widget. This option is only supported by the `'select-box'`, `'color-palette'` and `'select-button-group'` types.

In preview mode, when the user hovers over one of the items in the widget (e.g. being it a dropdown item in 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 on 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 allows you to create inputs in the Inspector for an arbitrary nested objects. For example, 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: {}
});
```

Each group options object in the `groups` object can contain the following parameters:

label	A label for the group. This label will be displayed as a header of the group section in the accordion-like inspector.
index	An index of the group within other groups. Use this to put the groups in a certain order.
closed	If set to `true`, the group will be closed by default.

Expressions

The inspector uses expressions to switch the visibility of its inputs (configurable through `when` parameter) based on the values of other inputs.

When an expression is evaluated to `false` (condition is not met) the input using this `when` clause will be hidden. Otherwise, this input will be shown.

Definition of Expressions

An expression when evaluated in a model context returns a boolean (true/false) based on certain model attributes. The expressions are defined recursively as follows.

- **{ <primitive> : { <path> : <value> }}** is an expression
- **{ <unary-operator> : expression }** is an expression
- **{ <multiary-operator> : [expression-1, expression-2, .. , expression-n] }** for **n > 0** is an expression

where:

path	Is a string determining a property of the model (e.g 'attrs/text/text', 'property')
value	Is a number, string or an array (e.g 13, 'jointjs', [1,3,5])
primitive	Can be one of the following:

eq	returns `true` if <value> and the value under <path> equals (`==` operator)
equal	returns `true` if <value> and the value under <path> equals (using `_.isEqual()` internally)
ne	returns `true` if <value> and the value under <path> don't equal
regex	returns `true` if regular expression made of <value> matches the value under <path>
text	returns `true` if <value> is a substring of the value under <path>
lt	returns `true` if the value under <path> is less than <value>

lte	return `true` if the value under <path> is less than or equal to <value>
gt	return `true` if the value under <path> is greater than <value>
gte	returns `true` if the value under <path> is greater than or equal to <value>
in	returns `true` if <value> is an array and contains the value under <path>
nin	returns `true` if <value> is an array and doesn't contain the value under <path>

returns `false` otherwise.

unary_operator

Accepts exactly one expression

not	returns the negation of an expression

multiary_operator

Accepts at least one expression

and	returns a conjunction of all the expressions (i.e. expr1 && exp2)
or	returns a disjunction of all the expressions (i.e. expr1 || exp2)
nor	returns the negation of a disjunction of all the expressions (i.e. !(expr1 || exp2))

Custom Operators in Expressions

As you can see above, the inspector provides a good list of useful primitive operators (`eq`, `lt`, `in`, ...). However, sometimes it is not enough and applications have special requirements on when fields in the inspector should be hidden/displayed based on other fields. To meet this requirements while still taking advantage of the inspector configurability through expressions, `ui.Inspector` provides a way to define your own custom operators. This can be done through the `operators` option on the inspector. The `operators` is an object of the form: `{ [operator]: function(cell, value, *arguments) }`, where the function should return `true` if the operator condition is met, `false` otherwise. The `cell` argument is the cell associated with the inspector, `value` is the value of the referenced field and `*arguments` is anything you pass to the operator in its configuration.

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 a value of a property set on the associated 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) {
            // prop === 'titleThreshold'
            return value.length > cell.prop(prop);
        }
    }
})
```

The example above displays the `description` field only when the `title` is longer than some threshold that we have stored in another property on the cell model, the `titleThreshold` property. Now whenever the user types a text into the title input field in the inspector and that text is longer than `cell.get('titleThreshold')`, the `description` field appears (and vice versa, if the text is shorter than `titleThreshold`, the `description` field gets hidden. This is great but there is a small problem with this. If the `titleThreshold` property changes on the cell model (e.g. due to some other change somewhere else in the application), than this change does not affect the visibility of the `description` field. In order to fix this, we have to tell the inspector that there are dependencies that could affect the resolution of the expression in the `when` clause. Here's a more correct example that does exactly that:

```
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) {
            // prop === 'titleThreshold'
            return value.length > cell.prop(prop);
        }
    }
})
```

Examples on using Expressions

Here is an example of some 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 type with options "email" and "tel". Below this input field, you want to show either a text or a number input field based on the value of the select input. 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's 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 }}}
    }
});
```

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 = $('').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 integrate a third party widget) while still you'd like to take advantage of the two-way data binding and configuration that the inspector provides. For this, inspector provides a facility (through `renderFieldContent(options, path, value)` and `getFieldValue(attribute, type)` functions). 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:

```
var graph = new joint.dia.Graph;
var paper = new joint.dia.Paper({
    el: document.getElementById('paper'),
    width: 500,
    height: 300,
    gridSize: 1,
    model: graph
});

var r1 = new joint.shapes.standard.Rectangle({
    position: { x: 80, y: 80 },
    size: { width: 130, height: 80 },
    attrs: { label: {
        text: 'My Element',
```

```
        textDecoration: 'none'
    }}
});
graph.addCell(r1);

var r2 = new joint.shapes.standard.Rectangle({
    position: { x: 290, y: 80 },
    size: { width: 130, height: 80 },
    attrs: { label: {
        text: 'My Second Element',
        textDecoration: 'none'
    }}
});
graph.addCell(r2);

function createInspector(cellView) {

    joint.ui.Inspector.create('.inspector-container', {
        cellView: cellView,
        inputs: {
            attrs: {
                text: {
                    style: {
                        textDecoration: {
                            type: 'select2',
                            group: 'text-decoration',
                            options: ['none', 'underline', 'overline', 'line-through']
                        }
                    },
                    text: {
                        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':

                    var $buttonSet = $('').css('margin', 20);
                    var $yes = $('').text('Say YES!');
                    var $no = $('').text('Say NO!');
                    $buttonSet.append([$yes, $no]);
                    $buttonSet.data('result', value);
                    // When the user clicks one of the buttons, set the result to our field
attribute
                    // so that we can access it later in `getFieldValue()`.
                    $yes.on('click', function() {
```

```
                        $buttonSet.data('result', 'YES');
                        inspector.updateCell($buttonSet, path, options);
                    });
                    $no.on('click', function() {
                        $buttonSet.data('result', 'NO');
                        inspector.updateCell($buttonSet, path, options);
                    });

                    return $buttonSet;

                case 'select2':

                    var $select = $('').width(170).hide();

                    // select2 requires the element to be in the live DOM.
                    // Therefore, postpone the select2 initialization for after we
                    // add the Inspector container to the live DOM (see below).
                    setTimeout(function() {
                        $select.show().select2({ data: options.options }).val(value ||
'none').trigger('change');
                        $select.data('select2').$container.css('margin', 20);
                        $select.on('change', function() {
                            inspector.updateCell($select, path, options);
                        });
                    }, 0);

                    return $select;
            }
        },

        getFieldValue: function(attribute, type) {

            if (type === 'my-button-set') {
                return { value: $(attribute).data('result') };
            }

            // Note that for our select2 select, we do not need to write
            // a special value extraction code. This is because
            // Inspector will use the .val() method on the  input by default.
        }
    });
}

// start with inspector for `r1`
createInspector(paper.findViewByModel(r1));

// switch inspector at pointerup
paper.on('cell:pointerup', createInspector);
```

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>` with `<label>` inside), which links to an address specified in `myList.url`. Additonally, our templating functionality can be used to define dynamic labels. We

use this for `myList.item` elements; the `'{{index}}'` placeholder is replaced with the element's index inside `myList` when an element is added to the array with the + button.

```
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  element and add  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 `render` 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 under the property path and the input DOM element as arguments.

API

static

create(container, options)	A helper for creating the inspector, where `container` is an `HTMLElement` or a selector (`container` is a DOM placeholder where the Inspector will be rendered into). For more information about `options` please visit ui.Inspector.configuration section. It returns the `joint.ui.Inspector` instance.
close()	A helper for closing the inspector created via `create` 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 (stored in the `el` property) to a live DOM element on the page.

updateCell()	Manually update the associated celll based on the current values in the inspector. This is especially useful if you use the inspector with the `live` mode disabled. See the Configuration section for more information.
remove()	Remove the inspector from the DOM. This should be called once you're finished with using the inspector.
openGroup(name)	Open group by `name`.
closeGroup(name)	Close group by `name`.
toggleGroup(name)	Toggle group by `name`.
openGroups()	Open all groups.
closeGroups()	Close all groups.
storeGroupsState()	Save the current group state - information which groups are opened and which are closed. The key for storing the state is determined by the `stateKey`.
restoreGroupsState()	Apply the stored state - open/close groups according this state. The actual state is looked up by the current `stateKey`.
getGroupsState()	Get the current group state - array of closed groups.

ui.Keyboard

The `ui.Keyboard` is an essential plugin for creating keyboard shortcuts in your Rappid application. It allows you to bind custom handlers (functions) to arbitrary keyboard events (e.g a single keystroke, key combination).

API

on(event, callback, [context])	Bind a callback function to an object. The callback will be invoked whenever the keyboard `event` is fired. Same logic as Backbone.Events.on. More information about `event` could be found in the Event description section.
off(event, callback, [context])	Remove a previously-bound callback function from an object. Same logic as Backbone.Events.off. More information about `event` could be found in the Event description section.
enable()	Start tracking keyboard events (enabled by default).
disable()	Stop tracking keyboard events.

isActive(name, event)	Check if key `name` is currently pressed. `event` is the DOM Event `KeyboardEvent`. **It is available for modifiers only** - `alt`, `ctrl`, `shift`, `command`

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 `ctr+c`, `ctrl+alt+t` etc. If you need to be more specific - in terms of `keydown`, `keyup`, `keypress` events - you can also define 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

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();
});
```

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 `100`.
windowArea	The maximum percentage of the screen covered by lightbox. Default is `0.8`.
closeAnimation	Set to `false` if you don't want the lightbox to be closed in an animated fashion.
closeButton	Set to `false` 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 `true`, a default download button is added to the lightbox (after any `buttons` specified). When clicked, this button triggers the `downloadAction` (`'download'` by default).
fileName	The filename of downloaded images. By default, the downloaded images are named `'Image'`.
downloadAction	If the action `action:[downloadAction]` is triggered on the lightbox, the lightbox `image` is downloaded to the user computer. By default, the download action is `'download'`. Downloaded images will have `fileName` as filename.
buttons	Buttons to show under the lightbox `title`. Explained in `ui.Dialog` configuration. Specifically, a custom download button can be created by providing a button that has `downloadAction` as its `action` (`'download'` by default).

API

open()	Open the lightbox.
close()	Close the lightbox.
on(event, handler [, context])	Register a `handler` function that will be called whenever `event` is triggered. The optional `context` is an object that will be the context of the handler function (the `this`).

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.

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 ui.PaperScroller 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. `zoomOptions.min` and `zoomOptions.max` 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 `joint.dia.Paper`. This is only useful if you use a different paper type for rendering your graphics. In case you inherit from `joint.dia.Paper` to implement some specific rendering, you can just pass your constructor function to this parameter and use the `paperOptions` parameter to pass additional options to your special paper.
paperOptions	Options object that will be passed to the internal paper used for rendering graphics.

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.

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');
```

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. `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 }});`
scrollToContent([opt])	Try to scroll the paper scroller so that the center of paper content is at the center of the paper scroller viewport. `paperScroller.scrollToContent();` `paperScroller.scrollToContent({ animation: { duration: 600 }});`
scrollToElement(element [, opt])	Try to scroll the paper scroller so that the center of `element` (an instance of `joint.dia.Element`) is at the center of the paper scroller viewport. `paperScroller.scrollToElement(element);` `paperScroller.scrollToElement(element, { animation: { duration: 600 }});`

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).

center([opt])	If no coordinate is specified, position the center of paper to the center of the paper scroller viewport. `paperScroller.center();` 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. `paperScroller.center({ padding: { left: 100 }});`
center(x, y [, opt])	Position the point (`x`, `y`) 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. `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)` 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. `paperScroller.center(100, 200, { padding: { left: 100 }});` `paperScroller.center(100, null, { padding: { left: 100 }});` `paperScroller.center(null, 200, { padding: { left: 100 }});`
centerContent([opt])	Position the center of paper content to the center of the paper scroller viewport.

	`` paperScroller.centerContent(); paperScroller.centerContent({ padding: { left: 100 }}); ```
centerElement(element [, opt])	Position the center of `element` (an instance of `joint.dia.Element`) to the center of the paper scroller viewport. ``` paperScroller.centerElement(element); paperScroller.centerContent(element, { padding: { left: 100 }}); ```
positionContent(positionName [, opt])	Align paper content inside the paper scroller viewport as specified by `positionName`. There are 9 possible `positionName` values: `'top'`, `'top-right'`, `'right'`, `'bottom-right'`, `'bottom'`, `'bottom-left'`, `'left'`, `'top-left'`, `'center'`. An appropriate point of the paper content area is positioned to the corresponding point inside the paper scroller viewport. For example, if positionName was `'bottom-left'`, 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). ``` paperScroller.positionContent('bottom-left'); paperScroller.centerContent('bottom-left', { padding: { horizontal: 20, vertical: 10 }}); ```
positionElement(element, positionName [, opt])	Align `element` (an instance of `joint.dia.Element`) inside the paper scroller viewport as specified by `positionName` (detailed above). An appropriate point of the element bbox is positioned to the corresponding point inside the paper scroller viewport, similar to `positionContent` method. ``` paperScroller.positionElement(element, 'right'); paperScroller.centerContent('right', { padding: { right: 50 }}); ```

positionRect(rect, positionName [, opt])	Align a custom `rect` inside the paper scroller viewport as specified by `positionName` (detailed above). An appropriate point of the rectangle is positioned to the corresponding point inside the paper scroller viewport, similar to `positionContent` method. `paperScroller.positionElement(element1.getBBox().union(element2.getBBox()), 'left-top');` `paperScroller.centerContent(path.bbox(), 'right-top', { padding: { vertical: 10, right: 50 }});`
positionPoint(point, x, y [, opt])	Position a custom `point` on the paper (in local coordinates) to the point (`x`, `y`) 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 `x` and `y`; the percentages are relative to the paperScroller area (only the area inside padding, if padding is provided). Negative values/percentages cause the algrithm 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: `paperScroller.positionPoint(new g.Point(0,0), 0, 0);` `paperScroller.positionPoint(new g.Point(0,0), '-100%', '-100%'); // same result` The following example positions `point` 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 visiblity

ui.PaperScroller provides methods for checking whether elements or points are visible and not scrolled outside the viewport.

getVisibleArea()	Returns the size and origin of the current paperScroller viewport. The returned value is a `g.Rect` object, which contains `x`, `y`, `width` and `height` describing the

	visible area of the paper ignoring paper transformations (as would no scale, translate or rotate be applied). `// return elements currenly visible in the paperScroller viewport.` `var visibleElements =` `graph.findModelsInArea(paperScroller.getVisibleArea());`
isElementVisible(element [, opt])	Returns `true` if the `element` is visible in the current paperScroller viewport. It returns `false` otherwise. Use `{ strict: [Boolean] }` option for toggling complete `true` vs. partial `false` visibility. It's non-strict by default. `// Scroll to an element only if it is not completely in the viewport.` `if (!paperScroller.isElementVisible(rect, { strict: true })) {` `  paperScroller.scrollToElement(rect);` `}`
isPointVisible(point)	Returns `true` if the `point` (e.g. `{ x: 100, y: 200 }`) is visible in the current paperScroller viewport. It returns `false` otherwise.

Zooming the paper

The PaperScroller exposes an API to scale the paper inside more easier. Method `zoom()` accepts a value we want the paper to be zoomed by. Optionally we can specify maximum (`{ max: [value] }`) and minimum (`{ min: [value] }`) value, a grid (`{ grid: [value] }`) the resulting value will be rounded to and whether the input value should be considered as a value we want to zoom the PaperScroller to (`{ absolute: true }`) by passing those in the `opt` parameter.

```
paperScroller.zoom(value, [opt]);
```

Setting the certain options can be useful for example when we binding buttons to perform a zooming. When the method is called without arguments, it returns the current zoom level.

```
$('#zoom-in').on('click', function() {
   paperScroller.zoom(0.2, { max: 4 });
});

$('#zoom-out').on('click', function() {
   paperScroller.zoom(-0.2, { min: 0.2 });
});

paperScroller.zoom(); // returns 1
```

Sometimes we want to scale the content of the paper so it fits the window through which the user sees the actual paper. For this purposes the PaperScroller implements `zoomToFit()` method. For the available options please see `paper.scaleContentToFit()` method as it is being used in the body of this method.

```
paperScroller.zoomToFit([opt]);
```

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 paperScroller zoom level.
duration	A string representing the number of seconds or milliseconds a transition animation should take to complete. E.g. `'500ms'`, `'2s'`. It defaults to `'1s'`
delay	A string representing the amount of time to wait before a transition starts. It defaults to `'0s'`
timingFunction	The option to describe how the intermmediate pan positions and zoom values are calculated. For all available values visit CSS `transition-timing-function` property documentation. It defaults to `'ease'`.
onTransitionEnd	A callback function fired with `transitionend` 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 `x` and `y` 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. `{ visibility: 1 }`). For instance 80% coverage could be set with `{ visibility: .8 }`
minScale	A number to make the resulting scale always greater than or equal to `minScale` value.
maxScale	A number to make the resulting scale always less than or equal to `maxScale` value.
scaleGrid	A number to make the resulting scale the largest multiple of `scaleGrid` 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.getCellsBBox([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 behaviour 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.

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 });
```

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	*SVGSVGElement*	Required. The drawing area. Any paths drawn by the user are appended to this `<svg>` element. They are not removed when the PathDrawer is removed.
pathAttributes	*Object*	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 `pathAttributes` object sets the `'class'` attribute (e.g. to `'example-path'`), you can access the path for further styling through the `#example-canvas .example-path` CSS selector.
startPointMarkup	*string*	Optional. The SVG markup of control point overlay elements. Default is `'<circle r="5"/>'`. (CSS styling should use the `.joint-path-drawer .start-point` CSS selector.)
snapRadius	*number*	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.

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:

<table>
<tr><td>mousedown
touchstart</td><td>
<table>
<tr><td>.start-point</td><td>Calls <code>onStartPointPointerDown(event)</code>. Closes the path.</td></tr>
<tr><td></td><td>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 and triggers a bound <code>mousemove</code>/<code>touchmove</code> event, this function starts drawing a <code>curveto</code> control point until a <code>mouseup</code>/<code>touchend</code> event is registered. Otherwise, starts drawing a <code>lineto</code> path segment.</td></tr>
</table>
</td></tr>
<tr><td>mousemove</td><td>Calls <code>onPointerMove(event)</code>. If the user is currently drawing a path segment, the path endpoint moves with user pointer.</td></tr>
<tr><td>dblclick</td><td>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> segement.</td></tr>
<tr><td>contextmenu</td><td>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.</td></tr>
</table>

Events

The `joint.ui.PathDrawer` object triggers various events that you can react on. Events can be handled by using the `on()` method.

<table>
<tr><td>clear</td><td>Triggered when the pathDrawer is cleared.</td></tr>
<tr><td>path:create</td><td>Triggered when a new path is created. The handler is passed a reference to the <code>svgPath</code> SVGPathElement.</td></tr>
<tr><td>path:finish</td><td>Triggered when a path is finished in a valid manner. The handler is passed a reference to the <code>svgPath</code> SVGPathElement. 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> SVGPathElement.</td></tr>
<tr><td>path:stop</td><td>Triggered when a path is finished by being stopped without reconnectiong 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</code> SVGPathElement.</td></tr>
</table>
</td></tr>
<tr><td>path:abort</td><td>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</code> SVGPathElement.</td></tr>
</table>

<table>
<tr>
<td>path:segment:add</td>
<td>Triggered when the user adds a new segment to path segment list. The handler is passed a reference to the <code>svgPath</code> SVGPathElement. 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</code> SVGPathElement.</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</code> SVGPathElement.</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</code> SVGPathElement.</td></tr>
</table>
</td>
</tr>
<tr>
<td>path:edit</td>
<td>Triggered when the user edits the path. The handler is passed a reference to the <code>svgPath</code> SVGPathElement. 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</code> SVGPathElement.</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</code> SVGPathElement.</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</code> SVGPathElement.</td></tr>
</table>
</td>
</tr>
<tr>
<td>path:interact</td>
<td>Triggered when the user interacts with drawer overlay elements. The handler is passed a reference to the <code>svgPath</code> SVGPathElement. The following specific events are triggered alongside this generic event:
<table>
<tr><td>path:control:adjust</td><td>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</code> SVGPathElement.</td></tr>
</table>
</td>
</tr>
</table>

	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 `svgPath` SVGPathElement.

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	*SVGPathElement*	Required. The path element to be edited.
anchorPointMarkup	*string*	Optional. The SVG markup of anchor point overlay elements. Default is `'<circle r="2.5"/>'`. (CSS styling should use the `.joint-path-editor .anchor-point` CSS selector.)
controlPointMarkup	*string*	Optional. The SVG markup of control point overlay elements. Default is `'<circle r="2.5"/>'`. (CSS styling should use the `.joint-path-editor .control-point` 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)	Adjust the position of an anchor point. Identified by `index`, the index of the anchor point within path segment list (starting from 0). The anchor point is translated by (`dx`, `dy`).
adjustControlPoint(index, controlPointIndex, dx, dy)	Adjust the position of a control point. Identified by `index`, the index of the path segment it is attached to within path segment list (starting from 0 for the first `moveto` path), and `controlPointIndex`, the index of the control point within the segment (1 or 2). The control point is translated by (`dx`, `dy`).
getControlPointLockedStates()	Return a list of lists that records, for every control point (identified as `[index][controlPointIndex]`), whether it is currently locked with another point (`true`) or not (`false`). In tandem with `setControlPointLockedStates(lockedStates)`, the function can be used to preserve locking information across multiple path editor sessions for a given element.
setControlPointLockedStates(lockedStates)	Change the `locked` status of the editor's control points, so they correspond to the information in the given `lockedStates` list (e.g. the output of the `getControlPointLockedStates()` function). A `true` value adds the `'locked'` class to the control point element; a `false` value removes the class.

For more fine-tuned programmatic control, you can emulate user interaction with delegated or synthetic mouse events.

Interaction

Users can interact with `joint.ui.PathEditor` objects by clicking/touching editor overlay elements:

mousedown touchstart	**.anchor-point**	Calls `onAnchorPointPointerDown(event)`. The selected point starts moving with user pointer thanks to bound `mousemove`/`touchmove` events, until a `mouseup`/`touchend` event is registered.
	.control-point	Calls `onControlPointPointerDown(event)`. The selected point starts moving with user pointer thanks to bound `mousemove`/`touchmove` events, until a `mouseup`/`touchend` event is registered.
	.segment-path	Calls `onSegmentPathPointerDown(event)`. The selected segment starts moving with user pointer thanks to bound `mousemove`/`touchmove` events, until a `mouseup`/`touchend` event is registered.

Double-clicking editor overlay elements triggers additional actions:

dblclick	**.anchor-point**	Calls `onAnchorPointDoubleClick(event)`. The selected point is removed.
	.control-point	Calls `onControlPointDoubleClick(event)`. The selected point is locked with an adjacent `curveto` segment's control point. The two points then mirror each other's angle around the anchor point. If the event target is a locked control point, this action unlocks it. If the adjacent segment is not a `curveto` segment, the interaction is ignored.
	.segment-path	Calls `onSegmentPathDoubleClick(event)`. An anchor point is created at the location of the user click and inserted into the path segment list.

The `joint.ui.PathEditor` code contains additional interaction methods that are not connected to user actions by default, but may be called upon by delegated events from your application. For example:

```
this.pathEditor.delegate('contextmenu', '.segment-path', function(event) {
    event.stopPropagation();
    event.preventDefault();

    this.convertSegmentPath(event);
}.bind(this.pathEditor));
```

addClosePathSegment(event)	If `event` is targeted on the first or last anchor point, and the path segment list contains no `closepath` segment, append a `closepath` segment to the path segment list.
removeClosePathSegment(event)	If `event` is targeted on a `closepath` segment, remove the segment from the path segment list.
convertSegmentPath(event)	If `event` is targeted on a path segment, convert the segment from linear to curved or vice versa. A `curveto` segment is replaced by a `lineto` segment. A `lineto`/`closepath` segment is replaced by a `curveto` segment whose control points are coincident with anchor points; a replacement `closepath` segment is appended to the path segment list if the converted segment was a `closepath` segment.

Events

The `joint.ui.PathEditor` object triggers various events that you can react on. Events can be handled by using the `on()` method.

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:

path:anchor-point:select	Triggered when the user interacts with an anchor point. The handler is passed the `index` 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 `index` of the segment that this control point is bound to in the path segment list. (Since the first `moveto` segment cannot have control points, the minimal index is 1.) The second value is the `controlPointIndex` of the control point. It can be either 1 or 2.
path:segment:select	Triggered when the user interacts with a path segment. The handler is passed the `index` of the segment in the path segment list. (Since the first `moveto` segment is not clickable, the minimal index is 1.)
path:control-point:lock	Triggered when the user locks a control segment (binds it with a control point in an adjacent `curveto` path). The handler is passed two values. The first value is the `index` of the segment that this control point is bound to in the path segment list. (Since the first `moveto` segment cannot have control points, the minimal index is 1.) The second value is the `controlPointIndex` of the control point. It can be either 1 or 2.
path:control-point:unlock	Triggered when the user unlocks a control segment (stops its binding with a control point in an adjacent `curveto` path). The handler is passed two values. The first value is the `index` of the segment that this control point is bound to in the path segment list. (Since the first `moveto` segment cannot have control points, the minimal index is 1.) The second value is the `controlPointIndex` of the control point. It can be either 1 or 2.

path:edit

Triggered when the user edits the path. The handler is passed a reference to the `svgPath` SVGPathElement. The following specific events are triggered alongside this generic event:

path:anchor-point:adjust	Triggered when the user adjusts the position of an anchor point. The handler is passed a reference to the `svgPath` SVGPathElement.
path:control-point:adjust	Triggered when the user adjusts the position of a control point. The handler is passed a reference to the `svgPath` SVGPathElement.
path:anchor-point:create	Triggered when the user adds a new anchor point to the path. The handler is passed a reference to the `svgPath` SVGPathElement.

	path:anchor-point:remove	Triggered when the user removes an anchor point from the path. The handler is passed a reference to the `svgPath` SVGPathElement.
	path:segment:convert	Triggered when the user converts a lineto path segment into a `curveto` segment or vice versa. The handler is passed a reference to the `svgPath` SVGPathElement.
	path:closepath-segment:add	Triggered when the user adds a `closepath` segment to the path. The handler is passed a reference to the `svgPath` SVGPathElement.
	path:closepath-segment:add	Triggered when the user removes a `closepath` segment from the path. The handler is passed a reference to the `svgPath` SVGPathElement.
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 `svgPath` SVGPathElement.	

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.

```
(new joint.ui.Popup({
    content: 'I am a ui.Popup',
    target: this
})).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:

```
$('#btn-open-image').on('click', function() {
    (new joint.ui.Popup({
        content: '',
        target: this
    })).render();
});

$('#btn-open-diagram').on('click', function() {
    (new joint.ui.Popup({
        content: function(el) {
            var graph = new joint.dia.Graph;
            var paper = new joint.dia.Paper({
                width: 200,
                height: 100,
                gridSize: 1,
                model: graph
            });
            $(el).append(paper.el);
            var r1 = new joint.shapes.basic.Rect({ position: { x: 10, y: 10 }, size: { width:
50, height: 30 }, attrs: { text: { text: 'r1' }, rect: { fill: '#FE854F' } } });
            var r2 = new joint.shapes.basic.Rect({ position: { x: 90, y: 40 }, size: { width:
50, height: 30 }, attrs: { text: { text: 'r2' }, rect: { fill: '#7C68FC' } } });
            var l = new joint.dia.Link({ source: { id: r1.id }, target: { id: r2.id } });
            graph.addCells([r1, r2, l]);
        },
        target: this
    })).render();
});

$('circle').on('click', function() {
    var popup = new joint.ui.Popup({
        events: {
            'click .btn-cancel': 'remove',
            'click .btn-change': function() {
                var strokeWidth = parseInt(this.$('.inp-stroke-width').val(), 10);
                var fill = this.$('.inp-fill').val();
                V(this.options.target).attr({ fill: fill, 'stroke-width': 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: this
    });
});
```

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.
target	The target element (either HTML or SVG).
padding	The gap between the target element an the popup. Default is `20`.
autoClose	Determines if the popup should be closed if the user clicked outside of the area of the popup. Default is `true`.

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 `autoClose` option to `false`).
on(event, handler [, context])	Register a `handler` function that will be called whenever `event` is triggered. The optional `context` is an object that will be the context of the handler function (the `this`).
joint.ui.Popup.close()	This is a static method that closes any poup 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 you `target` element changes its position while you still want to keep the popup opened.

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.SelectoBox` 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: **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 `content` property. **icon** — A path to an image. The icon is prepended to the item and is given the `select-box-option-icon` CSS class. **selected** — `true` 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 `ui.SelectBox` widget looks up the selected item from the `options` array (by using the `selected` boolean property). If nothing is selected, the first item in the `options` array is selected by default.

keyboardNavigation	`true` (the default) if you want the `ui.SelectBox` to provide a keyboard navigation (up/down/left/right/enter/escape keys are supported).
selectBoxOptionsClass	When the dropdown is opened, the `ui.SelectBox` generates an HTML container that contains all the items. This container is then appended to the document body (or `target` if provided). Sometimes, you want to provide custom styling to this dropdown container. `selectBoxOptionsClass` 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	`ui.SelectBox` 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 `target` option is exactly for that. It accepts an HTML element that will be used as a container for the generated dropdown items.
openPolicy	`openPolicy` determines how the dropdown will open when clicked. It can be one of `'auto'`, `'above'`, `'coverAbove'`, `'below'`, `'coverBelow'` and `'selected'`.
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 `selected` index to `-1` and the `placeholder` to any HTML you want to display in the selected item window. This placeholder has the `selected-box-placeholder` 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 `el` property that points to the container HTML element and append it anywhere in the DOM (e.g. `$(document.body).append(selectBox.el)`).
getSelection()	Get the current selection. The selection points to one of the items from the `options` array.
getSelectionValue()	Get the current selection value.
getSelectionIndex()	Get the index in the `options` array of the item that is currently selected.
select(index, opt)	Select an item. `index` is the index to the `options` array. `opt` is optional and can be an arbitrary object that will be passed to the `option:select` event handler.
selectByValue(value)	Select an item by value. The selected item will be chosen based on a strict equality of the `value` passed an a `value` (`content`) of any of the items.
isOpen()	Returns `true` if the dropdown is currently opened. `false` 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 `true` if the dropdown is disabled.
on(event, handler [, context])	Register a `handler` function that will be called whenever `event` is triggered. The optional `context` is an object that will be the context of the handler function (the `this`).

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 `options` 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 `options` array. If you selected the item with the `select(index, opt)` method, the third argument to the event handler will be your `opt` 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.

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:

width	The width of the whole button group in pixels.
options	An array of items (the buttons). Each item is an object with the following properties:

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 `content` property.
icon	A path to an image that represents the button.
iconSelected	A path to an image that represents the selected state of the button. If `iconSelected` is not defined, the selected state is the same as `icon`.
selected	`true` if the item should be selected by default.
buttonWidth	The width of the button in pixels.
buttonHeight	The height of the button in pixels.
iconWidth	The width of the icon (if it is used) in pixels.

iconHeight	The height of the icon (if it is used) in pixels.
selected	The index (or array of indices if `multi` is `true`) of the item which should be selected by default. If this value is undefined (which it is by default), the `ui.SelectButtonGroup` widget looks up the selected item(s) from the `options` array (by using the `selected` boolean property).
multi	Set to `true` to enable multiple selection (multiple buttons can be selected at the same time). `multi` is `false` by default.
buttonWidth	The width of the buttons in pixels. This can be overridden by `buttonWidth` set for an item in the `options` array.
buttonHeight	The height of the buttons in pixels. This can be overridden by `buttonHeight` set for an item in the `options` array.
iconWidth	The width of the icons (if they are used) in pixels. This can be overridden by `iconWidth` set for an item in the `options` array (only for items that have the `icon` option specified).
iconHeight	The height of the icons (if they are used) in pixels. This can be overridden by `iconHeight` set for an item in the `options` array (only for items that have the `icon` option specified).

API

render()	Render the button group. Note that once you render the button group, you can use the `el` property that points to the container HTML element and append it anywhere in the DOM (e.g. `$(document.body).append(selectButtonGroup.el)`).
getSelection()	Get the current selection. The selection points to an item(s) from the `options` array.
getSelectionValue()	Get the current selection value(s).
select(index, opt)	Select an item. `index` is the index to the `options` array. `opt` is optional and can be an arbitrary object that will be passed to the `option:select` event handler.
selectByValue(value)	Select an item by value. The selected item will be chosen based on a strict equality of the `value` passed an a `value` (`content`) of any of the items.
deselect()	Deselect all selected items.
remove()	Remove the button group from the DOM.
on(event, handler [, context])	Register a `handler` function that will be called whenever `event` is triggered. The optional `context` is an object that will be the context of the handler function (the `this`).

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 `options` 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 `options` array. If you selected the item with the `select(index, opt)` method, the third argument to the event handler will be your `opt` object.
mouseout	Triggered when the user leaves the entiner button group with mouse cursor. The handler is passed an event object as the only argument.

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 element with the `.joint-selection` class. The rectangles above the selected elements are also elements having the `.selection-box` class and the final rectangle surrounding all the selected element is a 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	*(required)* The JointJS paper the Selection should operate on.
graph	The JointJS graph the Selection should operate on. If no graph is passed `paper.model` is used.
collection	A Backbone Collection object used to store selected cells. If no collection is passed `new Backbone.Collection` 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 `true` 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: `filter: ['basic.Rect', 'mymodule.HiddenShape']`
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`.
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: `[ { name: 'remove', position: 'nw', events: { pointerdown: 'removeElements' } }, { name: 'rotate', position: 'sw', events: { pointerdown: 'startRotating', pointermove: 'doRotate', pointerup: 'stopBatch' } } ]` 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.

Selection API

addHandle(opt)	Add a custom tool to the Selection. `opt.name` 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. `opt.position` is a string that specifies a position of the tool handle. Possible values are `n`, `nw`, `w`, `sw`, `s`, `se`, `e` and `ne`. `opt.icon` 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 `name` from the Selection.
changeHandle(name, opt)	Change a tool handle named `name` in the Selection. The `opt` object can contain the same parameters as with the `addHandle()` method except of the `name` property. `opt` parameters will be merged with those defined previously.
on(event, callback)	Register a handler (`callback`) for an `event` 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.

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 `[name]`.
action:[name]:pointermove	Triggered when the user moves with mouse cursor after a tool handle named `[name]` was mousedowned (touched).
action:[name]:pointerup	Triggered when the user releases his mouse cursor after a tool handle named `[name]` was mousedowned (touched).
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, `x` and `y` paper local coordinates. Use `elementView.model` if you wish to access the actual element model. 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, `x` and `y` 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, `x` and `y` 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);
});
```

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 and calling `startListening()` method on that object:

```
var snaplines = new joint.ui.Snaplines({ paper: paper })
snaplines.startListening()
```

Demo

Configuration

The `joint.ui.Snaplines` constructor function takes the following parameters:

- `paper` - The JointJS paper (`joint.dia.Paper`) 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: `function(element)` in which case the function should return `true` if the element should not be taken into account for snapping.

Styling Snaplines

Snaplines can be customized in CSS. Snapline is a 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;
}
```

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 `joint.dia.Paper` or `joint.ui.PaperScroller` object [mandatory]
width	the width of the stencil [mandatory]
height	the height of the stencil [mandatory]
search	An object defining what properties of the elements in the stencil will be used for searching. The object has element types as keys (or `*` 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. `search: { '*': ['attrs/text/text'], 'basic.Image': ['description'], 'basic.Path': ['description'] }` 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 `label`, `index` and `closed` properties. `label` property specifies the name of the group that will be displayed in the UI, `index` is a number specifying the position of the group among other groups within the stencil and the optional `closed` property tells stencil whether this group should be closed or opened by default. An example of this object can look like: `groups: { basic: { label: 'Basic Elements', index: 1 }, text: { label: 'Text', index: 2, closed: true }, advanced: { label: 'Advanced Elements', index: 3, closed: true } }`
groupsToggleButtons	If set to `true` buttons for expanding/collapsing groups (expand all/collapse all) are rendered into the stencil. It defaults to `false`.
paperPadding	A number specifying the padding that will be used in the internal papers that hold the elements in the stencil. It defaults to `10`.

dropAnimation	An object defining an animation that will be performed on an element that is dropped outside the target paper area. It defaults to `undefined` meaning that there will be no animation. The `dropAnimation` object can have `duration` and `easing` properties where `duration` is the duration of the animation in milliseconds (defaults to `150`) and `easing` is either `"swing"` (the default) or `"linear"`. Example: `dropAnimation: { duration: 300, easing: 'linear' }`
label	A string or HTML Element rendered at the top of the stencil. It defaults to `Stencil`.
layout	if set to `true` the stencil elements are automatically layed out using the Grid Layout plugin. To adjust the layout behaviour pass an object defining the layout configuration. For all available options see the aforementioned plugin. The option is valid only if the default `layoutGroup` option is used. `layout: { columnWidth: 100, columns: 3, rowHeight: 100, }`
layoutGroup(graph, group)	a function that actually lays the elements out. It accepts a `graph` (1st parameter) and the `group` (2nd) that the graph elements belongs to. The function is called for each group on initialization and after the stencil is filtered. Note that the graph does not contain elements that don't match the search criterium (if any). It defaults to the Grid Layout. `layoutGroup: function(graph, group) { _.each(graph.getElements(), function(el, index) { el.position(0, index * 100); }); });`
dragStartClone(element)	A function that produces an element clone when the user starts dragging. It defaults to `element.clone()`.
dragEndClone(element)	A function that produces an element clone when the user places the dragged element into the paper. It defaults to `element.clone().`
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 `true` dragged clones are automatically scaled based on the current paper transformations. Note, that this option is ignored when `snaplines` option applied (it always scales the elements). It defaults to `false`.
paperOptions	An object with `joint.dia.Paper` options to adjust the look and behaviour of the stencil's papers. e.g. `paperOptions: { elementView: CustomElementView, background: { color: 'red' } }` This options can be also used to adjust a single paper (and takes precedence) when defined inside of the group definition. e.g. `groups: { myGroup: { index: 1, label: 'My Group', paperOptions: { color: 'blue' } } }` To adjust the graph in Stencil papers it's also possible to pass a `model` as well. **Please note, the `paperOptions` has to be a function in such a case:** `paperOptions: function() { return { elementView: CustomElementView, model: new joint.dia.Graph([], { cellNamespace: CustomElementNamespace }), background: { color: 'red' } } }`

Populating Stencil with elements

```
var r = new joint.shapes.basic.Rect({
    position: { x: 10, y: 10 }, size: { width: 50, height: 30 }
});
var c = new joint.shapes.basic.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 `string` or a `html`.
index	position of the group in the stencil.
closed	*[optional]* when `true`, this group will be initially closed.
height	*[optional]* the height of the paper containing the shapes of the group. If not defined, then the height of the stencil will be used. If non of these height are defined, the paper will be automatically scaled to fit its content (recommended).
layout	*[optional]* same as the stencil layout option. It allows you to override the global settings for a particular group.

```
var 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 basic.Rect) or `'*'` to match any type. `ATTRIBUTE_PATH` is a path to element's attribute or property (e.g 'attrs/text/text', 'description').

```
var stencil = new joint.ui.Stencil({
    search: {
        '*': ['attrs/text/text'],
        'basic.Image': ['description'],
        'basic.Path': ['description']
    }
});
```

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.

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.

API

load(groupedElements)	Accept an object with multiple key-value pairs, where *key* is the name of a group and *value* an array of elements to be rendered into that group. For instance: `stencil.load({ group1: [{ type: 'basic.Rect'},{ type: 'basic.Circle' }], group2: [{ type: 'basic.Image' }] });` 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 `loadGroup` below.
loadGroup(elements [, group])	Accept an array of elements and render them into a single stencil group. If no `group` provided, the default group is used.
getGraph(groupName)	Get the graph associated with the group identified by `groupName`. If the stencil does not use groups, just omit the `groupName` parameter to get the only graph present.
getPaper(groupName)	Get the paper associated with the group identified by `groupName`. If the stencil does not use groups, just omit the `groupName` 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 `setPaper(myTab2)` whenever the active tab changes. Note that the `paper`

	argument can be both a `joint.dia.Paper` object or the `joint.ui.PaperScroller` object, the stencil can handle both.
openGroup(groupName)	Open group `groupName`.
closeGroup(groupName)	Close group `groupName`.
toggleGroup(groupName)	Toggle group `groupName`.
isGroupOpen(groupName)	Return `true` if the group with the `groupName` is open. Return `false` 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.

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.

filter	Triggered when the user uses the search input field to filter elements in the stencil. The handler has the following signature: `function(graph, groupName, keyword) {}` where `graph` is a set of elements that matched the filter (wrapped in `joint.dia.Graph`), `groupName` is the name of the group in which elements where filtered and `keyword` is the current search term.
drop:invalid	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 `dropAnimation` option). This event gives you a chance to react when the invalid drop occurs. The handler has the following

	signature: `function(evt, element) {}`, where `evt` is a mouse event object and `element` is the element model that would have been otherwise dropped into the main paper (and graph for that matter).

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 autosizing on the text inside our elements so that they stretch based on the bounding box inside it.

```
// RECOMMENDED

paper.on('cell:pointerdblclick', function(cellView, evt) {
    joint.ui.TextEditor.edit(evt.target, {
        cellView: cellView,
        textProperty: cellView.model.isLink() ? 'labels/0/attrs/text/text' : 'attrs/text/text'
    });
});

function autosize(element) {

    var view = paper.findViewByModel(element);
    var text = view.$('text')[0];
    // Use bounding box without transformations so that our autosizing works
    // even on e.g. rotated element.
    var bbox = V(text).bbox(true);
```

```
    // Give the element some padding on the right/bottom.
    element.resize(bbox.width + 50, bbox.height + 50);
}

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.

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 <text> 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 `'cell:pointerclick'` or `'cell:pointerdblclick'` events) but you can always find a view for any JointJS element or link by using `paper.findViewByModel(cell)`.

opt.textProperty	The path to the text property that causes the element/link to re-render the text inside it. (e.g. `'attrs/text/text'`).
opt.annotationsProperty	The path to the property that holds the text annotations. (e.g. `'attrs/text/annotations'`). *(Rich-text specific.)*
opt.placeholder	Set to `true` to show a placeholder when the text is empty. The placeholder text and visual look can be styled in CSS as follows: `.text-editor .caret.placeholder:before { // The caret.` `  background-color: red;` `}` `.text-editor .caret.placeholder:after { // The placholder text.` `  content: 'Enter text...';` `  font-style: italic;` `  color: blue;` `}`
opt.annotations	Annotations that will be set on the `ui.TextEditor` instance. *(Rich-text specific.)*
opt.annotateUrls	Set to `true` to enable the text editor to automatically detect URL hyperlinks and annotate them using the default `urlAnnotation`.
opt.urlAnnotation	The default annotation for detected URL hyperlinks: `{` `  attrs: {` `  'class': 'url-annotation',` `  fill: 'lightblue',` `  'text-decoration': 'underline'` `  }` `}`
opt.useNativeSelection	ui.TextEditor uses native selections for selecting text. This can be disabled by setting `useNativeSelection` to `false` 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 `useNativeSelection` to `false`. 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: `textareaAttributes: {` `  autocorrect: 'off',` `  autocomplete: 'off',`

```
    autocapitalize: 'off',
    spellcheck: 'false',
    tabindex: '0'
}
```

Some jQuery UI users reported an issue with `autocomplete` attribute being set on the textarea, resulting in the following error thrown in the jQuery UI: *Uncaught Error: cannot call methods on autocomplete prior to initialization; attempted to call method 'off'*. In this case, you might want to set your own `textareaAttributes` to overcome this issue (in this specific case, omitting the `autocomplete` attribute).

The following table lists options that you can pass to the `ui.TextEditor` constructor function (**only for advanced use**):

text	SVG <text> element. It is recommended to use the `joint.ui.TextEditor.getTextElement(el)` method to find the container <text> SVG element wrapping all the lines. You can use this method directly on the `evt.target` element in your event handlers and just check if the returned value is truthy.
placeholder	Set to `true` to show a placeholder when the text is empty. The placeholder text and visual look can be styled in CSS as follows: `.text-editor .caret.placeholder:before { // The caret.` `  background-color: red;` `}` `.text-editor .caret.placeholder:after { // The placholder text.` `  content: 'Enter text...';` `  font-style: italic;` `  color: blue;` `}`

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).

render([root])	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 `root` (HTML DOM element) is specified, the text editor HTML container will be appended to that element instead of to the `document.body`. This is useful if you want the text editor to scroll with some other container.
remove()	Remove the Text Editor. Call this when you're done with inline text editing.
selectAll()	Programmatically select all the text inside the text editor.
select(selectionStart, selectionEnd)	Programmatically select portion of the text inside the text editor starting at `selectionStart` ending at `selectionEnd`. This method automatically swaps `selectionStart` and `selectionEnd` if they are in a wrong order.
deselect()	Programmatically deselect all the selected text inside the text editor.
getSelectionStart()	Return the start character position of the current selection.
getSelectionEnd()	Return the end character position of the current selection.
getSelectionRange()	Return an object of the form `{ start: Number, end: Number }` containing the start and end position of the current selection. Note that the start and end positions are returned *normalized*. 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.
getSelectionLength()	Return the number of characters in the current selection.
getSelection()	Return the selected text.
findAnnotationsUnderCursor(annotations, selectionStart)	Find all the annotations in the `annotations` array that the cursor at `selectionStart` position falls into.

findAnnotationsInSelection(annotations, selectionStart, selectionEnd)	Find all the annotations that fall into the selection range specified by `selectionStart` and `selectionEnd`. This method assumes the selection range is normalized.
setCaret(charNum [, opt])	Programmatically set the caret position. If `opt.silent` is `true`, the text editor will not trigger the `'caret:change'` event.
hideCaret()	Programmatically hide the caret.
getTextContent()	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 `charNum` character position.
getURLBoundary(charNum)	Return the start and end character positions for a URL under `charNum` character position. Return `undefined` if there was no URL recognized at the `charNum` 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 `setCurrentAnnotation({ 'font-weight': 'bold' })` 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. *(Rich-text specific.)*
setAnnotations(annotations)	Set annotations of the text inside the text editor. These annotations will be modified during the course of using the text editor. *(Rich-text specific.)*
getAnnotations()	Return the annotations of the text inside the text editor. *(Rich-text specific.)*

getCombinedAnnotationAttrsAtIndex(index, annotations)	Get the combined (merged) attributes for a character at the position `index` taking into account all the `annotations` that apply. *(Rich-text specific.)*
getSelectionAttrs(range, annotations)	Find a common annotation among all the `annotations` that fall into the `range` (an object with `start` and `end` properties - *normalized*). For characters that don't fall into any of the `annotations`, assume `defaultAnnotation` (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, `undefined` is returned. This is useful e.g. when your toolbar needs to reflect the text attributes of a selection. *(Rich-text specific.)*
joint.ui.TextEditor.getTextElement(el)	Return the containing SVG <text> element closest to the SVG element `el`. This is especially handy in event handlers where you can just pass `evt.target` element to this function and it will return the element that you can directly use as a parameter to the `ui.TextElement` constructor function. If no such element is exists, the function returns `undefined`.
joint.ui.TextEditor.edit(el, opt)	See the Configuration section for details.
joint.ui.TextEditor.close()	Close the currently opened text editor (if there was one). Note that before actually removing the editor (and if the `annotateUrls` option is set to `true`), the `close()` 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.
joint.ui.TextEditor.applyAnnotation(annotations)	Apply `annotations` 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 annotations array from the toolbar changes and apply it using this function on the current selection.

joint.ui.TextEditor.setCurrentAnnotation(attributes)	See the instance `setCurrentAnnotation()` method above.
joint.ui.TextEditor.getAnnotations()	See the instance `getAnnotations()` method above.
joint.ui.TextEditor.setCaret(charNum, opt)	See the instance `setCaret()` method above.
joint.ui.TextEditor.deselect()	See the instance `deselect()` method above.
joint.ui.TextEditor.selectAll()	See the instance `selectAll()` method above.
joint.ui.TextEditor.select(start, end)	See the instance `select()` method above.
joint.ui.TextEditor.getNumberOfChars()	See the instance `getNumberOfChars()` method above.
joint.ui.TextEditor.getCharNumFromEvent(evt)	See the instance `getCharNumFromEvent()` method above.
joint.ui.TextEditor.getWordBoundary()	See the instance `getWordBoundary()` method above.
joint.ui.TextEditor.findAnnotationsUnderCursor()	See the instance `findAnnotationsUnderCursor()` method above except the `annotations` and `index` arguments are automatically added so this method does not accept any arguments.
joint.ui.TextEditor.findAnnotationsInSelection()	See the instance `findAnnotationsInSelection()` method above except the `annotations` and start and end index of the selection are automatically added so this method does not accept any arguments.
joint.ui.TextEditor.getSelectionAttrs(annotations)	See the instance `getSelectionAttrs()` method above except the `range` is automatically added.
joint.ui.TextEditor.getSelectionLength()	See the instance `getSelectionLength()` method above.
joint.ui.TextEditor.getSelectionRange()	See the instance `getSelectionRange()` method above.

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: `function(newText)` 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: `function(selectionStart, selectionEnd)`. For example, you can see the selected text like this: `ed.on('select:change', function(selectionStart, selectionEnd) { var text = ed.getTextContent().substring(selectionStart, selectionEnd); });`
select:changed	Triggered when the user is finished selecting text inside the text editor. The callback has the following signature: `function(selectionStart, selectionEnd)`. For example, you can see the selected text like this: `ed.on('select:changed', function(selectionStart, selectionEnd) { var text = ed.getTextContent().substring(selectionStart, selectionEnd); });`
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 ui.TextEditor is used on a text rendered along a path (see `Vectorizer#text(str, { textPath: '' }))`). In this case, if the user moves his cursor outside the visible text area, `caret:out-of-range` event is triggered so that the programmer has chance to react (if he wants to because these situations are handled seamlessly in ui.TextEditor by hiding the caret).

ui.Toolbar

With the Toolbar plugin you can easily enrich your application with various tools. Either use built-in controls to manage other Rappid 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	*Array<object> | Array<string>*	An array of tools defined as a plain JavasScript object. For more information visit Tools configuration section.
references	*object*	`key => value` storage. Some widgets require additional references for their functionality. Whenever you use such a widget (e.g. `undo/redo` requires `joint.dia.CommandManager`, `zoom` requires `joint.ui.PaperScroller`) you need to pass the required reference into the toolbar as well.
groups	*object*	Split the widgets defined in `tools` option into groups. Groups could be aligned and ordered in the context of the Toolbar.

groups.align	*string*	Can be `left` or `right`.
groups.index	*number*	Set the order of the group.

Tool configuration

type	*string*	Set the type of the widget. Available widgets are listed in the Toolbar widgets section.
name	*string*	`name` is used to differentiate multiple widgets with the same `type` and has to be unique in the context of the toolbar. Note that the toolbar uses the `name` as a prefix for events triggered on widgets. More information can be found in the Events section.
group	*string*	Name of the group where a widget belongs to.
attrs	*object*	JointJS style attribute definition. Additional DOM element attributes can be defined here.
theme	*string*	Allow to override the global `theme`.

Toolbar API

getWidgetByName(name)	*<joint.ui.Widget>*	Return an instance of `joint.ui.Widget` with the matching `name`.
getWidgets()	*Array<joint.ui.Widget>*	Return an array of `joint.ui.Widget` instances included in the toolbar.

Toolbar widgets

<table>
<tr><td>checkbox</td><td></td><td>
<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>
</td></tr>
<tr><td>toggle</td><td></td><td>
<table>
<tr><td>value</td><td>boolean</td></tr>
<tr><td>label</td><td>string</td></tr>
</table>
</td></tr>
<tr><td>separator</td><td></td><td>Vertical line by default.
<table>
<tr><td>width</td><td>number</td></tr>
</table>
</td></tr>
<tr><td>label</td><td></td><td>
<table>
<tr><td>text</td><td>string</td></tr>
</table>
</td></tr>
<tr><td>range</td><td></td><td>
<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>
</td></tr>
<tr><td>selectBox</td><td></td><td>Configuration is described in ui.SelectBox section.</td></tr>
</table>

<table>
<tr><td>button</td><td></td><td><table><tr><td>text</td><td>string</td></tr></table></td></tr>
<tr><td>inputText</td><td></td><td><table><tr><td>label</td><td>string</td></tr><tr><td>value</td><td>string</td></tr></table></td></tr>
<tr><td>inputNumber</td><td></td><td><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></td></tr>
<tr><td>textarea</td><td></td><td><table><tr><td>label</td><td>string</td></tr><tr><td>value</td><td>string</td></tr></table></td></tr>
<tr><td>selectButtonGroup</td><td></td><td>Configuration is described in ui.SelectButtonGroup section.</td></tr>
<tr><td>zoomIn | zoomOut | zoomToFit</td><td></td><td><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></table>
<fieldset><legend>Required references:</legend>paperScroller <code>joint.ui.PaperScroller</code> instance</fieldset></td></tr>
<tr><td>zoomSlider</td><td></td><td><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></table></td></tr>
</table>

value	number	100
unit	number	'%'

Required references:

paperScroller `joint.ui.PaperScroller` instance

undo | **redo**

No additional options.

Required references:

commandManager `joint.dia.CommandManager` instance

fullscreen

Presents an element (*target*

target	HTMLElement|string selector	window.top.document.body

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.

```
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
    }
});
```

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	*string* | *Element* | *jQuery*	A CSS selector or DOM element or jQuery object that is the target for th the user hovers over this element with his mouse cursor, the tooltip app
rootTarget	*string* | *Element* | *jQuery*	A CSS selector or DOM element or jQuery object that is the root target f When using the `rootTarget`, the event handlers that trigger the tooltip delegated (as opposed to directly-bound). This is useful when the `targe` pointing to elements that might get removed from the DOM and then p The tooltip tries to find the selected elements specified by the selector i option when it is initialized (during instantiation) and than uses these ele target for the tooltip. If later on such elements are removed from the DO looses a reference to the target. Therefore, we can prevent this by using selecting the container element and `target` selector for selecting the re tooltip at the time of the `"mouseover"` event.
content	*string* | *Element* | *jQuery* | *function(element)*	The content of the tooltip. It can be an arbitrary string/HTML or a functi
padding	*string* | *Element* | *jQuery* | *function(element)*	The distance between `target` element and the tooltip.
position	*string* | *function(element)*	The position of the element relative to the tooltip. `left` means the elen left of the shown tooltip. Can be either `top`, `left`, `bottom`, `right`
direction	*string* | *function(element)*	The direction of the tooltip arrow. Can be either `top`, `right`, `bottom`, `l` `auto` sets the arrow accordingly to 'position' property. Arrows are disab `direction` is a falsy value or `off`

positionSelector	*string* | *Element* | *jQuery* | *function(element)*	A CSS selector or DOM element or jQuery object or a function used to c position of the edge of the tooltip. It defines bounding box which the to get across.
trigger	*string*	Event which makes the tooltip show up. Can be either `hover`, `focus` or option cannot be set as a data attribute on the DOM element.
minResizedWidth	*number*	A minimal width of the tooltip. The tooltip width can be resized down to `minResizedWidth`. If the available space is smaller than the `minResizedW` direction of the tooltip is changed to its opposite direction (left tooltip is right, top to bottom and vice versa). Set `minResizedWidth` to a falsy valu automatic resizing or direction swapping should take a place. It defaults
hideTrigger	*string*	Event which makes the tooltip disappear. Can be any of the DOM event multiple events with `' '`. `<label data-tooltip="tooltip text" data-tooltip-hide-trigger="mous mouseouver">HIDE ON MOUSEOUT</label>` or `new joint.ui.Tooltip({ rootTarget: '.click-tooltip', target: '[data-tooltip]', trigger: 'click', hideTrigger: 'mouseout' });`
viewport	*object* | *function(element)*	`viewport` defines the boundary for the tooltip. It guarantees that the to will be rendered inside the `viewport` bounding box.

selector	*string* | *jQuery* | *Element*	CSS selector, Elemen object defining the `v` object. The element t tooltip should be ren this element.
padding	*number*	Minimal distance fror `viewport` boundarie
dataAttributePrefix	*string* | *function(element)*	Defines prefix for HT attributes with the to configuration. `new joint.ui.Too target: '[cu def]', dataAttribut 'custom-def' });`

matches with

```
<label data-custom-
content" data-custor
position-selector="
custom-def-
position="bottom">B
```

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. Rappid tooltips come with two build-in sizes for tooltips: small and normal. In order to enable the small version use `'small'` as a `className`.

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();
```

Configuration

The following table lists options that you can pass to the `ui.TreeLayoutView` constructor function:

model	Instance of `joint.layout.TreeLayout`. See layout.TreeLayout plugin for more details.
paper	JointJS paper object.
useModelGeometry	If set to `true`, the cells position and dimensions will be used as a basis for the TreeLayoutView wrapping rectangle position. By default, this is set to `false` which causes the TreeLayoutView 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 TreeLayoutView wrapping rectangle position. In this case, set the `useModelGeometry` to `true`.
clone(element)	A function, that is applied on the `element` to create a clone for dragging. It defaults to `return element.clone();`.
canInteract(element)	A predicate, that returns `true` if the `element`, which the user interacts with, can be actually dragged. It defaults to `return true;`.
previewAttrs	An object with SVG attributes to be applied onto the preview element. The preview consists of 3 components - `child`, `parent` and `link`. e.g. `{ child: { opacity: 0.5 }, link: { opacity: 0.5 }, parent: { rx: 20, ry: 20 } }`

API

startDragging(elementOrElements)	Initiates dragging of a single or multiple elements (`dia.Element` or an array of `dia.Element`). Useful when combinded with the Selection plugin to drag & drop multiple nodes at once.