jsPlumb

jsPlumb Community edition provides a means for a developer to visually connect elements on their web pages, using SVG.

jsPlumb has no external dependencies.

The 1.7.x line of releases will be the last ones to support IE8. We are trickling the occasional bugfix and feature into 1.7.x but the next major event in the Community edition will be release 2.0.0, which will work only in modern browsers that support SVG.

Imports and Setup

It is a good idea to read this entire page. There are a few things you need to know about integrating jsPlumb with your UI.

Browser Compatibility

jsPlumb runs on everything from IE6 up. There are some caveats, though, because of various browser/library bugs:

- jQuery 1.6.x and 1.7.x have a bug in their SVG implementation for IE9 that causes hover events to not get fired.
- jQuery 2.0 does not support IE 6,7 or 8.
- Safari 5.1 has an SVG bug that prevents mouse events from being passed through the transparent area of an SVG element (Safari 6.x does not seem to have the same problem)

Setup

Doctype

text/html

If you're serving your pages with content type `text/html`, as most people are, then you should use this doctype:

```
<!doctype html>
```

This gives you Standards mode in all browsers and access to HTML5.

Required Imports

No required imports.

```
<script src="PATH_TO/jsPlumb-x.x.x-min.js "></script>
```

Initializing jsPlumb

You should not start making calls to jsPlumb until the DOM has been initialized - perhaps no surprises there. To handle this, you should bind to the `ready` event on jsPlumb (or the instance of jsPlumb you are working with):

```
jsPlumb.bind("ready", function() {
  ...
  // your jsPlumb related init code goes here
  ...
});
```

There's a helper method that can save you a few precious characters:

```
jsPlumb.ready(function() {
    ...
    // your jsPlumb related init code goes here
    ...
});
```

If you bind to the `ready` event after jsPlumb has already been initialized, your callback will be executed immediately.

Multiple jsPlumb instances

jsPlumb is registered on the browser's window by default, providing one static instance for the whole page to use. You can also instantiate independent instances of jsPlumb, using the `getInstance` method, for example:

```
var firstInstance = jsPlumb.getInstance();
```

The variable `firstInstance` can now be treated exactly as you would treat the `jsPlumb` variable - you can set defaults, call the `connect` method, whatever:

```
firstInstance.importDefaults({
  Connector : [ "Bezier", { curviness: 150 } ],
  Anchors : [ "TopCenter", "BottomCenter" ]
});

firstInstance.connect({
  source:"element1",
  target:"element2",
  scope:"someScope"
});
```

`getInstance` optionally takes an object that provides the defaults:

```
var secondInstance = jsPlumb.getInstance({
  PaintStyle:{
    strokeWidth:6,
    stroke:"#567567",
    outlineStroke:"black",
    outlineWidth:1
  },
  Connector:[ "Bezier", { curviness: 30 } ],
  Endpoint:[ "Dot", { radius:5 } ],
  EndpointStyle : { fill: "#567567"  },
  Anchor : [ 0.5, 0.5, 1, 1 ]
});

secondInstance.connect({
  source:"element4",
  target:"element3",
  scope:"someScope"
});
```

It is recommended to use separate instances of jsPlumb wherever possible.

Element Ids

jsPlumb uses the `id` attribute of any element with which it interacts. If `id` is not set, jsPlumb will create an id for the element. It is recommended that you set appropriate ids for the elements in your UI yourself.

Changing Element Id

Because of the fact that jsPlumb uses element ids, you need to tell jsPlumb if an element id changes. There are two methods to help you do this:

- `jsPlumb.setId(el, newId)` Use this if you want jsPlumb to take care of changing the id in the DOM. It will do so, and then update its references accordingly.
- `jsPlumb.setIdChanged(oldId, newId)` Use this if you have already changed the element's ID, and you just want jsPlumb to update its references.

Method Arguments

Almost every method in jsPlumb that operates on elements supports multiple formats for specifying the element(s) on which to operate.

Selectors/NodeLists

In jQuery there is the concept of a "selector" - a list of elements conforming to some CSS path spec, for instance `$(".myClass")`. Both the jQuery flavour, and also vanilla, jsPlumb support these selectors as arguments for elements (vanilla jsPlumb can do this because jQuery's selector object is *list-like*, ie. it has a `length` property).

When using vanilla jsPlumb, you can use a `NodeList` - it is, effectively, the native equivalent of the selectors we just discussed. There are several ways of getting a `NodeList` from the DOM; perhaps the most useful (supported in IE8+) is `document.querySelectorAll("some selector")`.

You will see **selector** referred to in many places in the jsPlumb documentation. Just think of it as a list of elements that match some CSS spec.

Element Ids

Passing a single string as argument will cause jsPlumb to treat that string as an element id.

Elements

You can pass DOM elements as arguments. This will probably not surprise you.

Arrays

You can also pass an array of any or all of the types we just listed. The contents of the array can be mixed - they do not have to be all one type.

Z-index Considerations

You need to pay attention to the z-indices of the various parts of your UI when using jsPlumb, in particular to ensure that the elements that jsPlumb adds to the DOM do not overlay other parts of the interface.

jsPlumb adds an element to the DOM for each Endpoint, Connector and Overlay. So for a Connection having visible Endpoints at each end and a label in the middle, jsPlumb adds four elements to the DOM.

To help you organise z-indices correctly, jsPlumb adds a CSS class to each type of element it adds. They are as follows:

Component	Class
Endpoint	jsplumb-endpoint
Connector	jsplumb-connector
Overlay	jsplumb-overlay

In addition, whenever the mouse is hovering over an Endpoint or Connection, that component is assigned the class `jsplumb-hover`. For more information about styling jsPlumb with CSS, see this page.

Where does jsPlumb add elements?

It's important to understand where in the DOM jsPlumb will add any elements it creates. If you want a TL;DR version, then it boils down to this:

- It is strongly recommended that you set a Container before you begin plumbing.
- A Container is some element that jsPlumb will use as the parent for all of the artefacts it adds to the UI. Typically, you would make this the parent element of the nodes you are connecting.
- If you do not specify a Container, jsPlumb will infer that the Container should be the `offsetParent` of the first element on which you call `addEndpoint`, `makeSource` or `makeTarget`, or the `offsetParent` of the source element in the first `connect` call - whichever happens first.

Let's discuss this in more detail.

Early versions of jsPlumb added everything to the **body** element. This has the advantage of being the most flexible arrangement in terms of supporting which elements can be connected, but in certain use cases produced unexpected results.

Consider the arrangement where you have some connected elements in a tab: you would expect jsPlumb to add elements inside the tab, so that when the user switches tabs and the current one is hidden, all the jsPlumb stuff is hidden too. But when the elements are on the body, this does not happen!

It is also quite common for jsPlumb to be used in a page in which the diagram is contained within some element that should show scrollbars when its content overflows. Appending elements to the document body prevents this from occurring automatically.

Container

You can - and **should** - instruct jsPlumb to use some element as the parent of everything jsPlumb adds to the UI through usage of the `setContainer` method in jsPlumb, or by providing the Container in the parameters to a `jsPlumb.getInstance` call.

Important This is a change from versions of jsPlumb prior to 1.6.2. In versions prior to 1.6.2 you could assign the Container directly via the `jsPlumb.Defaults.Container` property. You can still do this - we're in Javascript here of course, it's a free-for-all - but it will be ignored.

Also Important If you happen to be using jsPlumb's `draggable` method to make other parts of your UI draggable (ie. not just things you're plumbing together), be careful not to call `draggable` on the element that is acting as the `Container` for the current instance, or you will see some odd situations occur when dragging. It is suggested that the best thing to do if you wish to use jsPlumb to enable dragging on non-plumbed elements is to create a new instance:

```
var nonPlumbing = jsPlumb.getInstance();
nonPlumbing.draggable("some element");
```

Some examples

- Set a Container to use as the default Container, using a jQuery selector, and then add an Endpoint. The related UI artefact will be a child of the document body:

```
jsPlumb.setContainer($("body"));
...
jsPlumb.addEndpoint(someDiv, { endpoint options });
```

- Set a container to use as the default container, using Plain Old DOM:

```
jsPlumb.setContainer(document.getElementById("foo"));
...
jsPlumb.addEndpoint(someDiv, { endpoint options });
```

- Set a container to use as the default container, using an element id, and then connect two elements. The elements created in this example will be children of the element with id "containerId":

```
jsPlumb.setContainer("containerId");
...
jsPlumb.connect({ source:someDiv, target:someOtherDiv });
```

- Get a new instance of jsPlumb and specify the container to use in the constructor params:

```
var j = jsPlumb.getInstance({
  Container:"foo"
});
...
jsPlumb.addEndpoint(someDiv, { endpoint options });
```

Container CSS

The container you choose should have `position:relative` set on it, because it is the origin of that element that jsPlumb will use to compute the placement of the artefacts it adds to the DOM, and jsPlumb uses absolute positioning.

Element Dragging

A common feature of interfaces using jsPlumb is that the elements are draggable. You should use `jsPlumbInstance.draggable` to configure this:

```
myInstanceOfJsPlumb.draggable("elementId");
```

...because if you don't, jsPlumb won't know that an element has been dragged, and it won't repaint the element.

Dragging is covered in detail on this page .

Performance

The speed at which jsPlumb executes, and the practical limit to the number of manageable connections, is greatly affected by the browser upon which it is being run. At the time of writing, it will probably not surprise you to learn that jsPlumb runs fastest in Chrome, followed by Safari/Firefox, and then IE browsers in descending version number order.

Suspending Drawing

Every `connect` or `addEndpoint` call in jsPlumb ordinarily causes a repaint of the associated element, which for many cases is what you want. But if you are performing some kind of "bulk" operation - like loading data on page load perhaps - it is recommended that you suspend drawing before doing so:

```
jsPlumb.setSuspendDrawing(true);
...
- load up all your data here -
...
jsPlumb.setSuspendDrawing(false, true);
```

Notice the second argument in the last call to `setSuspendDrawing`: it instructs jsPlumb to immediately perform a full repaint (by calling, internally, `repaintEverything`).

I said above it is recommended that you do this. It really is. It makes an enormous difference to the load time when you're dealing with a lot of Connections on slower browsers.

batch

This function abstracts out the pattern of suspending drawing, doing something, and then re-enabling drawing:

```
jsPlumb.batch(fn, [doNotRepaintAfterwards]);
```

Example usage:

```
jsPlumb.batch(function() {
  // import here
  for (var i = 0, j = connections.length; i < j; i++) {
      jsPlumb.connect(connections[i]);
  }
});
```

Here, we've assumed that `connections` is an array of objects that would be suitable to pass to the `connect` method, for example:

```
{ source:"someElement", target:"someOtherElement" }
```

By default, this will run a repaint at the end. But you can suppress that, should you want to:

```
jsPlumb.batch(function() {
  // import here
}, true);
```

Note this method used to be called `doWhileSuspended` and was renamed in version 1.7.3.

Anchor Types

Continuous anchors are the type that require the most maths, because they have to calculate their position every time a paint cycle occurs.

Dynamic and Perimeter anchors are the next slowest (with Perimeter being slower than most Dynamic Anchors as they are actually Dynamic Anchors that have, by default, 60 locations to choose from).

Static anchors like `Top`, `Bottom` etc are the fastest.

Zooming

A fairly common requirement with the sorts of applications that use jsPlumb is the ability to zoom in and out. As of release 1.5.0 there is a way to do this for browsers that support CSS3 (meaning, essentially, everything except IE < 9).

Changing zoom requires that you do two things:

1. Set a `transform` property on an appropriate container
2. Tell jsPlumb what the zoom level is.

Container

You need to identify some element that is the parent of all of your nodes and the jsPlumb artefacts. This is probably fairly obvious. What you might not know about, though, is the `Container` concept in jsPlumb. If you don't, I'd encourage you to go and read this page just quickly, because the best thing to do is to correctly configure a `Container` and then manipulate the `transform` property of that element.

Let's say we have some `div` whose id is `drawing`, and we're going to use that as the `Container`:

```
jsPlumb.setContainer("drawing");
```

CSS `transform` property

Now to set the zoom to 0.75, say, we change the `transform` property accordingly. Remember that `transform` is one of those properties that have several vendor prefix versions, so there are several ways to do what I've got here, and, given that you're probably a computer programmer, you've most likely got a favourite. But anyway, here's something.

```
$("#drawing").css({
  "-webkit-transform":"scale(0.75)",
  "-moz-transform":"scale(0.75)",
  "-ms-transform":"scale(0.75)",
  "-o-transform":"scale(0.75)",
  "transform":"scale(0.75)"
});
```

jsPlumb.setZoom

You now need to tell jsPlumb about the new zoom level:

```
jsPlumb.setZoom(0.75);
```

A Helper Function

Maybe you'd like to just grab this:

```
window.setZoom = function(zoom, instance, transformOrigin, el) {
  transformOrigin = transformOrigin || [ 0.5, 0.5 ];
  instance = instance || jsPlumb;
  el = el || instance.getContainer();
  var p = [ "webkit", "moz", "ms", "o" ],
      s = "scale(" + zoom + ")",
      oString = (transformOrigin[0] * 100) + "% " + (transformOrigin[1] * 100) + "%";

  for (var i = 0; i < p.length; i++) {
    el.style[p[i] + "Transform"] = s;
    el.style[p[i] + "TransformOrigin"] = oString;
  }

  el.style["transform"] = s;
  el.style["transformOrigin"] = oString;

  instance.setZoom(zoom);
};
```

Notes

- This is not a jQuery function. It does not, amazingly, even know jQuery exists.
- `el` must be a plain DOM element. If you've got a jQuery selector, pass in `el[0]`. You don't have to pass in `el`; if you do not, it uses the Container from the jsPlumb instance. Note that jsPlumb automatically converts the Container into a plain DOM element, so it doesn't matter what you pass to jsPlumb as the value of Container.
- `transformOrigin` is optional; it defaults to [0.5, 0.5] - the middle of the element (this is the browser default too)
- `instance` is an instance of jsPlumb - either `jsPlumb`, the static instance, or some instance you got through `jsPlumb.newInstance(...)`. The function will default to using the static instance of jsPlumb if you do not provide one.
- `zoom` is a decimal where 1 means 100%.

Configuring Defaults

The easiest way to set a look and feel for your plumbing is to override the defaults that jsPlumb uses. If you do not do this you are forced to provide your overridden values on every call. Every argument to the `connect` and `addEndpoint` methods has an associated default value in jsPlumb.

The defaults that ship with jsPlumb are stored in `jsPlumb.Defaults`, which is a Javascript object. Valid entries, and their initial values, are:

```
Anchor : "BottomCenter",
Anchors : [ null, null ],
ConnectionsDetachable   : true,
ConnectionOverlays  : [],
Connector : "Bezier",
Container : null,
DoNotThrowErrors  : false,
DragOptions : { },
DropOptions : { },
Endpoint : "Dot",
Endpoints : [ null, null ],
EndpointOverlays : [ ],
EndpointStyle : { fill : "#456" },
EndpointStyles : [ null, null ],
EndpointHoverStyle : null,
EndpointHoverStyles : [ null, null ],
HoverPaintStyle : null,
LabelStyle : { color : "black" },
LogEnabled : false,
Overlays : [ ],
MaxConnections : 1,
PaintStyle : { strokeWidth : 8, stroke : "#456" },
ReattachConnections : false,
RenderMode : "svg",
Scope : "jsPlumb_DefaultScope"
```

Note that in `EndpointHoverStyle`, the default `fill` is **null**. This instructs jsPlumb to use the `stroke` from the attached Connector's hover style to fill the Endpoint.

Note also that you can specify either or both (or neither) of `EndpointStyle` and `EndpointStyles`. This allows you to specify a different style for each Endpoint in a Connection. `Endpoint` and `Endpoints` use the same concept. jsPlumb will look first in the individual `Endpoint/EndpointStyle` arrays, and then fall back to the single default version.

You can override these defaults by using the `importDefaults` method:

```
jsPlumb.importDefaults({
  PaintStyle : {
    strokeWidth:13,
    stroke: 'rgba(200,0,0,0.5)'
  },
  DragOptions : { cursor: "crosshair" },
  Endpoints : [ [ "Dot", { radius:7 } ], [ "Dot", { radius:11 } ] ],
  EndpointStyles : [{ fill:"#225588" }, { fill:"#558822" }]
});
```

...after the jsPlumb script has been loaded of course! Here we have specified the following default behaviour:

- connectors are 13 pixels wide and painted with a semi-transparent red line
- when dragging an element the crosshair cursor is used
- the source endpoint is a dot of radius 7; the target endpoint is a dot of radius 11
- the source endpoint is blue; the target endpoint is green

Explanation of each Default setting

- **Anchor** - this will be used as the Anchor for any Endpoint for which no Anchor is declared - this applies to both the source and/or target of any Connection.
- **Anchors** - default source and target Anchors for Connections.
- **Connector** - The default Connector to use.
- **ConnectionsDetachable** - Whether or not Connections are detachable by default using the mouse.
- **Container** - The element to use as the parent for all artefacts added by jsPlumb. You cannot set this using `importDefaults` - it will only be honoured when you provide it in the defaults you pass to a `getInstance` call. To change the container after instantiating an instance of jsPlumb, use `setContainer`. See here for a discussion of the container concept.
- **DoNotThrowErrors** - Whether or not jsPlumb will actually throw an exception if an Anchor, Endpoint or Connector that does not exist is requested.
- **ConnectionOverlays** - Default Overlays to attach to every Connection
- **DragOptions** - Default options with which to configure any element made draggable with `jsPlumb.draggable`
- **DropOptions** - Default options with which to configure the droppable behaviour of any target Endpoint.
- **Endpoint** - The default Endpoint definition. Will be used whenever an Endpoint is added or otherwise created and jsPlumb has not been given any explicit Endpoint definition.
- **Endpoints** - Default source and target Endpoint definitions for use with `jsPlumb.connect`.
- **EndpointOverlays** - Default Overlays to attach to every Endpoint.
- **EndpointStyle** - Default appearance of an Endpoint.

- **EndpointStyles** - Default appearance of the source and target Endpoints in a Connection
- **EndpointHoverStyle** - Default appearance of an Endpoint in hover state.
- **EndpointHoverStyles** - Default appearance of the source and target Endpoints in a Connection in hover state.
- **HoverPaintStyle** - Default appearance of a Connection in hover state.
- **LabelStyle** - Default style for a Label. **Deprecated**: use CSS for this instead.
- **LogEnabled** - Whether or not jsPlumb's internal logging is switched on.
- **Overlays** - Default Overlays to add to both Connections and Endpoints
- **MaxConnections** - The default maximum number of Connections any given Endpoint supports.
- **PaintStyle** - The default appearance of a Connector
- **ReattachConnections** - Whether or not to reattach Connections that were detached using the mouse and then neither reconnected to their original Endpoint nor connected to some other Endpoint.
- **RenderMode** - The default render mode.
- **Scope** - The default scope of Endpoints and Connections. Scope provides a rudimentary control over which Endpoints can be connected to which other Endpoints.

Providing defaults to jsPlumb.getInstance

When you create a new instance of jsPlumb via `jsPlumb.getInstance` you can provide defaults for that instance as a constructor parameter - here's how we'd create a new instance using the same default values as the example above:

```
jsPlumb.getInstance({
  PaintStyle : {
    strokeWidth:13,
    stroke: 'rgba(200,0,0,100)'
  },
  DragOptions : { cursor: "crosshair" },
  Endpoints : [ [ "Dot", { radius:7 } ], [ "Dot", { radius:11 } ] ],
  EndpointStyles : [
    { fill:"#225588" },
    { fill:"#558822" }
  ]
});
```

Basic Concepts

Introduction

jsPlumb is all about connecting things together, so the core abstraction in jsPlumb is the `Connection` object, which is itself broken down into these five concepts:

- **Anchor** - a location, relative to an element's origin, at which an Endpoint can exist. You do not create these yourself; you supply hints to the various jsPlumb functions, which create them as needed. They have no visual representation; they are a logical position only. Anchors can be referenced by name, for the Anchors that jsPlumb ships with, or with an array containing various parameters, for greater control. See the Anchors page for more detail.

- **Endpoint** - the visual representation of one end of a Connection. You can create and attach these to elements yourself, which you are required to do to support drag and drop, or have jsPlumb create them when creating a Connection programmatically using `jsPlumb.connect(...)`. You can also join two Endpoints programmatically, by passing them as arguments to `jsPlumb.connect(...)`. See the Endpoints page for more detail.

- **Connector** - the visual representation of the line connecting two elements in the page. jsPlumb has four types of these available as defaults - a Bezier curve, a straight line, 'flowchart' connectors and 'state machine' connectors. You do not interact with Connectors; you just specify definitions of them when you need to. See the Connectors page for more detail.

- **Overlay** - a UI component that is used to decorate a Connector, such as a Label, Arrow, etc.

- **Group** - a group of elements contained within some other element, which can be collapsed, causing connections to all of the Group members to be pooled on the collapsed Group container. For more information, see the Groups page.

One Connection is made up of two Endpoints, a Connector, and zero or more Overlays working together to join two elements. Each Endpoint has an associated Anchor.

jsPlumb's public API exposes only Connection and Endpoint, handling the creation and configuration of everything else internally. But you still need to be across the concepts encapsulated by Anchor, Connector and Overlay.

Connector, Endpoint, Anchor & Overlay Definitions

Whenever you need to define a Connector, Endpoint, Anchor or Overlay, you must use a "definition" of it, rather than constructing one directly. This definition can be either a string that nominates the artifact you want to create - see the `endpoint` parameter here:

```
jsPlumb.connect({
    source:"someDiv",
    target:"someOtherDiv",
    endpoint:"Rectangle"
});
```

...or an array consisting of both the artifact's name and the arguments you want to pass to its constructor:

```
jsPlumb.connect({
    source:"someDiv",
    target:"someOtherDiv",
    endpoint:[ "Rectangle", {
      cssClass:"myEndpoint",
      width:30,
      height:10
  }]
});
```

There is also a three-argument method that allows you to specify two sets of parameters, which jsPlumb will merge together for you. The idea behind this is that you will often want to define common characteristics somewhere and reuse them across a bunch of different calls:

```
var common = {
    cssClass        :        "myCssClass",
    hoverClass        :        "myHoverClass"
};
jsPlumb.connect({
    source:"someDiv",
    target:"someOtherDiv",
    endpoint:[ "Rectangle", { width:30, height:10 }, common ]
});
```

This syntax is supported for all Endpoint, Connector, Anchor and Overlay definitions. Here's an example using definitions for all four:

```
var common = {
    cssClass:"myCssClass"
};
jsPlumb.connect({
  source:"someDiv",
  target:"someOtherDiv",
  anchor:[ "Continuous", { faces:["top","bottom"] }],
  endpoint:[ "Dot", { radius:5, hoverClass:"myEndpointHover" }, common ],
  connector:[ "Bezier", { curviness:100 }, common ],
  overlays: [
        [ "Arrow", { foldback:0.2 }, common ],
        [ "Label", { cssClass:"labelClass" } ]
    ]
});
```

The allowed constructor parameters are different for each artifact you create, but every artifact takes a single JS object as argument, with the parameters as (key,value) pairs in that object.

Anchors

Introduction

An `Anchor` models the notion of where on an element a `Connector` should connect - it defines the location of an `Endpoint`. There are four main types of Anchors:

- **Static** - these are fixed to some point on an element and do not move. They can be specified using a string to identify one of the defaults that jsPlumb ships with, or an array describing the location (see below)
- **Dynamic** - these are lists of Static anchors from which jsPlumb picks the most appropriate one each time a Connection is painted. The algorithm used to determine the most appropriate anchor picks the one that is closest to the center of the other element in the Connection. A future version of jsPlumb might support a pluggable algorithm to make this decision.
- **Perimeter** anchors - these are anchors that follow the perimeter of some given shape. They are, in essence, Dynamic Anchors whose locations are chosen from the perimeter of the underlying shape.
- **Continuous** anchors - These anchors are not fixed to any specific location; they are assigned to one of the four faces of an element depending on that element's orientation to the other element in the associated Connection. Continuous anchors are slightly more computationally intensive than Static or Dynamic anchors because jsPlumb is required to calculate the position of every Connection during a paint cycle, rather than just Connections belonging to the element in motion.

Static Anchors

jsPlumb has nine default anchor locations you can use to specify where the Connectors connect to elements: these are the four corners of an element, the center of the element, and the midpoint of each edge of the element:

- `Top` (also aliased as `TopCenter`)
- `TopRight`
- `Right` (also aliased as `RightMiddle`)
- `BottomRight`
- `Bottom` (also aliased as `BottomCenter`)
- `BottomLeft`
- `Left` (also aliased as `LeftMiddle`)
- `TopLeft`
- `Center`

Each of these string representations is just a wrapper around the underlying array-based syntax `[x, y, dx, dy]`, where `x` and `y`are coordinates in the interval `[0,1]` specifying the position of the anchor, and `dx` and `dy`,which specify the orientation of the curve incident to the anchor, can have a value of 0, 1 or -1. For example, `[0, 0.5, -1, 0]` defines a `Left` anchor with a connector curve that emanates leftward from the anchor. Similarly, `[0.5, 0, 0, -1]` defines a `Top` anchor with a connector curve emanating upwards.

```
jsPlumb.connect({...., anchor:"Bottom", ... });
```

is identical to:

```
jsPlumb.connect({...., anchor:[ 0.5, 1, 0, 1 ], ... });
```

Anchor Offsets

In addition to supplying the location and orientation of an anchor, you can optionally supply two more parameters that define an offset in pixels from the given location. Here's the anchor specified above, but with a 50 pixel offset below the element in the y axis:

```
jsPlumb.connect({...., anchor:[ 0.5, 1, 0, 1, 0, 50 ], ... });
```

Dynamic Anchors

These are Anchors that can be positioned in one of a number of locations, choosing the one that is most appropriate each time something moves or is painted in the UI.

There is no special syntax for creating a Dynamic Anchor; you just provide an array of individual Static Anchor specifications, eg:

```
var dynamicAnchors = [ [ 0.2, 0, 0, -1 ],  [ 1, 0.2, 1, 0 ],
                       [ 0.8, 1, 0, 1 ], [ 0, 0.8, -1, 0 ] ];

jsPlumb.connect({...., anchor:dynamicAnchors, ... });
```

Note that you can mix the types of these individual Static Anchor specifications:

```
var dynamicAnchors = [ [ 0.2, 0, 0, -1 ],  [ 1, 0.2, 1, 0 ],
                            "Top", "Bottom" ];

jsPlumb.connect({...., anchor:dynamicAnchors, ... });
```

Default Dynamic Anchor

jsPlumb provides a dynamic anchor called `AutoDefault` that chooses from `Top`, `Right`, `Bottom` and `Left`:

```
jsPlumb.connect({...., anchor:"AutoDefault", ... });
```

Location Selection

The algorithm that decides which location to choose just calculates which location is closest to the center of the other element in the Connection. It is possible that future versions of jsPlumb could support more sophisticated choice algorithms, if the need arose.

Draggable Connections

Dynamic Anchors and Draggable Connections can interoperate: jsPlumb locks the position of a dynamic anchor when you start to drag a connection from it, and unlocks it once the connection is either established or abandoned. At that point you may see the position of the dynamic anchor change, as jsPlumb optimises the connection.

You can see this behaviour in the draggable connections demonstration, when you drag a connection from the blue endpoint on window 1 to the blue endpoint on window 3 - the connection is established and then window 1's blue endpoint jumps down to a location that is closer to window 3.

Perimeter Anchors

These are a form of Dynamic Anchor in which the anchor locations are chosen from the perimeter of some given shape. jsPlumb supports six shapes:

- `Circle`
- `Ellipse`
- `Triangle`
- `Diamond`
- `Rectangle`
- `Square`

`Rectangle` and `Square`, are not, strictly speaking, necessary, since rectangular shapes are the norm in a web page. But they are included for completeness.

```
jsPlumb.addEndpoint("someElement", {
  endpoint:"Dot",
  anchor:[ "Perimeter", { shape:"Circle" } ]
});
```

In this example our anchor will travel around the path inscribed by a circle whose diameter is the width and height of the underlying element.

Note that the `Circle` shape is therefore identical to `Ellipse`, since it is assumed the underlying element will have equal width and height, and if it does not, you will get an ellipse. `Rectangle` and `Square` have the same relationship.

By default, jsPlumb approximates the perimeter with 60 anchor locations. You can change this, though:

```
jsPlumb.addEndpoint("someDiv", {
        endpoint:"Dot",
        anchor:[ "Perimeter", { shape:"Square", anchorCount:150 }]
});
```

Obviously, the more points the smoother the operation. But also the more work your browser has to do.

Here's a triangle and diamond example, just for completeness:

```
jsPlumb.connect({
        source:"someDiv",
        target:"someOtherDiv",
        endpoint:"Dot",
        anchors:[
                [ "Perimeter", { shape:"Triangle" } ],
                [ "Perimeter", { shape:"Diamond" } ]
        ]
});
```

Perimeter Anchor Rotation

You can supply a `rotation` value to a Perimeter anchor - an example can be seen in this demo. Here's how you would use it:

```
jsPlumb.connect({
        source:"someDiv",
        target:"someOtherDiv",
        endpoint:"Dot",
        anchors:[
                [ "Perimeter", { shape:"Triangle", rotation:25  } ],
                [ "Perimeter", { shape:"Triangle", rotation:-335  } ]
        ]
});
```

Note that the value must be supplied **in degrees**, not radians, and the number may be either positive or negative. In the example above, both triangles are of course rotated by the same amount.

Continuous Anchors

As discussed above, these are anchors whose positions are calculated by jsPlumb according to the orientation between elements in a Connection, and also how many other Continuous anchors happen to be sharing the element. You specify that you want to use Continuous anchors using the string syntax you would use to specify one of the default Static Anchors, for example:

```
jsPlumb.connect({
        source:someDiv,
        target:someOtherDiv,
        anchor:"Continuous"
});
```

Note in this example I specified only "anchor", rather than "anchors" - jsPlumb will use the same spec for both anchors. But I could have said this:

```
jsPlumb.connect({
  source:someDiv,
  target:someOtherDiv,
  anchors:["Bottom", "Continuous"]
});
```

...which would have resulted in the source element having a Static Anchor at `BottomCenter`. In practise, though, it seems the Continuous Anchors work best if both elements in a Connection are using them.

Note also that Continuous Anchors can be specified on `addEndpoint` calls:

```
jsPlumb.addEndpoint(someDiv, {
  anchor:"Continuous",
  paintStyle:{ fill:"red" }
});
```

...and in `makeSource` / `makeTarget`:

```
jsPlumb.makeSource(someDiv, {
  anchor:"Continuous",
  paintStyle:{ fill:"red" }
});

jsPlumb.makeTarget(someDiv, {
  anchor:"Continuous",
  paintStyle:{ fill:"red" }
});
```

... and in the jsPlumb defaults:

```
jsPlumb.Defaults.Anchor = "Continuous";
```

Continuous Anchor Faces

By default, a Continuous anchor will choose points from all four faces of the element on which it resides. You can control this behaviour, though, with the `faces` parameter on the anchor spec:

```
jsPlumb.makeSource(someDiv, {
    anchor:["Continuous", { faces:[ "top", "left" ] } ]
});
```

Allowed values are: - `top` - `left` - `right` - `bottom`

If you provide an empty array for the `faces` parameter, jsPlumb will default to using all four faces.

Associating CSS Classes with Anchors

The array syntax discussed above supports an optional 7th value, which is a string that represents a CSS class. This CSS class is then associated with the Anchor, and applied to the Anchor's Endpoint and Element whenever the Anchor is selected.

A Static Anchor is of course always "selected", but a Dynamic Anchor cycles through a number of different locations, and each of these may have a different CSS class associated with it.

The CSS class that gets written to the Endpoint and Element is prefixed with the associated jsPlumb instance's `endpointAnchorClass` prefix, which defaults to:

```
jsplumb-endpoint-anchor-
```

So if you had the following, for example:

```
var ep = jsPlumb.addEndpoint("someDiv", {
  anchor:[0.5, 0, 0, -1, 0, 0, "top" ]
};
```

Then the Endpoint created by jsPlumb and also the element `someDiv` would have this class assigned to them:

```
jsplumb-endpoint-anchor-top
```

An example using Dynamic Anchors:

```
var ep = jsPlumb.addEndpoint("someDiv", {
  anchor:[
    [ 0.5, 0, 0, -1, 0, 0, "top" ],
    [ 1, 0.5, 1, 0, 0, 0, "right" ]
    [ 0.5, 1, 0, 1, 0, 0, "bottom" ]
    [ 0, 0.5, -1, 0, 0, 0, "left" ]
  ]
});
```

Here, the class assigned to Endpoint and Element would cycle through these values as the anchor location changes:

```
jsplumb-endpoint-anchor-top
jsplumb-endpoint-anchor-right
jsplumb-endpoint-anchor-left
jsplumb-endpoint-anchor-bottom
```

Note that if you supply a class name that consists of more than one term, jsPlumb will not prepend the prefix to each term in the class:

```
var ep = jsPlumb.addEndpoint("someDiv", {
  anchor:[ 0.5, 0, 0, -1, 0, 0, "foo bar" ]
});
```

would result in **2** classes being added to the Endpoint and Element:

```
jsplumb-endpoint-anchor-foo
```

and

```
bar
```

Changing the anchor class prefix

The prefix used with anchor classes is stored as the jsPlumb member `endpointAnchorClass`. You can change this to whatever you like on some instance of jsPlumb:

```
jsPlumb.endpointAnchorClass = "anchor_";
```

or maybe

```
var jp = jsPlumb.getInstance();
jp.endpointAnchorClass = "anchor_";
```

Connectors

Connectors are the lines that actually join elements of the UI. jsPlumb has four connector implementations - a straight line, a Bezier curve, "flowchart", and "state machine". The default connector is the Bezier curve.

You optionally specify a Connector by setting the `connector` property on a call to `jsPlumb.connect`, `jsPlumb.addEndpoint(s)`, `jsPlumb.makeSource` or `jsPlumb.makeTarget`. If you do not supply a value for `connector`, the default will be used.

You specify Connectors using the syntax described in [Connector, Endpoint, Anchor & Overlay Definitions]. Allowed constructor values for each Connector type are described below.

- **Bezier** Provides a cubic Bezier path between the two Endpoints. It supports a single constructor argument:

  - `curviness` - Optional; defaults to 150. This defines the distance in pixels that the Bezier's control points are situated from the anchor points. This does not mean that your Connector will pass through a point at this distance from your curve. It is a hint to how you want the curve to travel. Rather than discuss Bezier curves at length here, we refer you to [Wikipedia].

- **Straight** Draws a straight line between the two endpoints. Two constructor arguments are supported:

  - `stub` - Optional, defaults to 0. Any positive value for this parameter will result in a stub of that length emanating from the Endpoint before the straight segment connecting to the other end of the connection.
  - `gap` - Optional, defaults to 0. A gap between the endpoint and either the start of the stub or the segment connecting to the other endpoint.

- **Flowchart** Draws a connection that consists of a series of vertical or horizontal segments - the classic flowchart look. Four constructor arguments are supported:

  - `stub` - this is the minimum length, in pixels, of the initial stub that emanates from an Endpoint. This is an optional parameter, and can be either an integer, which specifies the stub for each end of the Connector, or an array of two integers, specifying the stub for the [source, target] endpoints in the Connection. Defaults to an integer with value 30 pixels.
  - `alwaysRespectStubs` - optional, defaults to false. This parameter instructs jsPlumb to always paint a line of the specified stub length out of each Endpoint, instead of reducing the stubs automatically if the two elements are closer than the sum of the two stubs.
  - `gap` - optional, defaults to 0 pixels. Lets you specify a gap between the end of the Connector and the elements to which it is attached.
  - `midpoint` - optional, defaults to 0.5. This is the distance between the two elements that the longest section of the flowchart connector will be drawn at. This parameter is useful for those cases where you have programmatic control of the drawing and perhaps want to avoid some other element on the page.
  - `cornerRadius` Defaults to 0. A positive value for this parameter will result in curved corner segments.

This Connector supports Connections that start and end on the same element ("loopback" connections).

- **State Machine** draws slightly curved lines (they are actually quadratic Bezier curves), similar to the State Machine connectors you may have seen in software like GraphViz. Connections in which some element is both the source and the target ("loopback") are supported by these Connectors (as they are with Flowchart Connectors); in this case you get a circle. Supported constructor parameters are:

  - `margin` - Optional; defaults to 5. Defines the distance from the element that the Connector begins/ends.
  - `curviness` - Optional, defaults to 10. This has a similar effect to the curviness parameter on Bezier curves.
  - `proximityLimit` - Optional, defaults to 80. The minimum distance between the two ends of the Connector before it paints itself as a straight line rather than a quadratic Bezier curve.

Endpoints

An `Endpoint` models the appearance and behaviour of one end of a `Connection`; it delegates its location to an underlying `Anchor`.

jsPlumb comes with four Endpoint implementations - `Dot`, `Rectangle`, `Blank` and `Image`. You optionally specify Endpoint properties using the `endpoint` parameter in a call to **jsPlumb.connect**, **jsPlumb.addEndpoint**, **jsPlumb.makeSource** or **jsPlumb.makeTarget**.

As with Connectors and Overlays, you specify Endpoints using the syntax described the page on basic concepts.

Creating an Endpoint

Endpoints are created in a number of different ways:

- when you call `jsPlumb.connect(..)` and pass an element id or DOM element as source/target, a new Endpoint is created and assigned.
- when you call `jsPlumb.addEndpoint(...)` a new Endpoint is created (and returned from the call)
- when you have configured an element using `jsPlumb.makeSource(...)` and subsequently drag a connection from that element, a new Endpoint is created and assigned.

In each of these different cases, the parameters you can use to specify the Endpoint you wish to have created are exactly the same.

Specifying Endpoint parameters

Endpoint types

- **Dot** This Endpoint draws a dot on the screen. It supports three constructor parameters:
  - `radius` - Optional; defaults to 10 pixels. Defines the radius of the dot.
  - `cssClass` - Optional. A CSS class to attach to the element the Endpoint creates.
  - `hoverClass` - Optional. A CSS class to attach to the element the Endpoint creates whenever the mouse is hovering over the element or an attached Connection.
- **Rectangle** Draws a rectangle. Supported constructor parameters are:
  - `width` Optional; defaults to 20 pixels. Defines the width of the rectangle.
  - `height` Optional; defaults to 20 pixels. Defines the height of the rectangle.
  - `cssClass` Optional. A CSS class to attach to the element the Endpoint creates.
  - `hoverClass` Optional. A CSS class to attach to the element the Endpoint creates whenever the mouse is hovering over the element or an attached Connection.
- **Image** Draws an image from a given URL. This Endpoint supports three constructor parameters:
  - `src` Required. Specifies the URL of the image to use
  - `cssClass` Optional. A CSS class to attach to the element the Endpoint creates.
  - `hoverClass` Optional. A CSS class to attach to the element the Endpoint creates whenever the mouse is hovering over the element or an attached Connection.
- **Blank** Does not draw anything visible to the user. This Endpoint is probably not what you want if you need your users to be able to drag existing Connections - for that, use a Rectangle or Dot Endpoint and assign to it a CSS class that causes it to be transparent.

Overlays

Overlays are UI elements that are painted onto Connections, such as Labels or Arrows.

Introduction

jsPlumb comes with five types of Overlays:

- **Arrow** - a configurable arrow that is painted at some point along the connector. You can control the length and width of the Arrow, the 'foldback' point - a point the tail points fold back into, and the direction (allowed values are 1 and -1; 1 is the default and means point in the direction of the connection)
- **Label** - a configurable label that is painted at some point along the connector.
- **PlainArrow** - an Arrow shaped as a triangle, with no foldback.
- **Diamond** - as the name suggests...a diamond.
- **Custom** - allows you to create the Overlay yourself - your Overlay may be any DOM element you like.

`PlainArrow` and `Diamond` are actually just configured instances of the generic `Arrow` overlay (see examples).

Overlay Location

A key concept with Overlays is that of their **location**.

For a Connector, the location of an Overlay refers to some point along the path inscribed by the Connector. It can be specified in one of three ways:

- as a decimal in the range [0..1], which indicates some proportional amount of travel along the path inscribed by the Connector. The default value of 0.5 is in this form, and it means the default location of an Overlay on a Connector is a point halfway along the path.
- as an integer greater than 1, which indicates some absolute number of pixels to travel along the Connector from the start point
- as an integer less than zero, which indicates some absolute number of pixels to travel backwards along the Connector from the end point.

For an Endpoint, the same principles apply, but location is specified as an [x,y] array. For instance, this would specify an Overlay that was positioned in the center of an Endpoint:

`location:[ 0.5, 0.5 ]`

Whereas this would specify an Overlay that was positioned 5 pixels along the x axis from the top left corner:

`location: [ 5, 0 ]`

And this would specify an Overlay that was positioned 5 pixels along the x axis from the bottom right corner:

`location: [ -5, 0 ]`

All Overlays support these two methods for getting/setting their location:

- **getLocation** - returns the current location
- **setLocation** - sets the current location. For Endpoints, location is expressed in terms of an [x, y] array whose values are either proportional to the width/height of the Endpoint (decimals in the range 0-1 inclusive), or absolute values (decimals greater than 0).

Adding Overlays

You can specify one or more overlays when making a call to `jsPlumb.connect`, `jsPlumb.addEndpoint` or `jsPlumb.makeSource` (but not `jsPlumb.makeTarget`: overlays are always derived from what the source of a Connection defines) The three cases are discussed below:

Specifying one or more overlays on a jsPlumb.connect call

In this example we'll create an Arrow with the default options for an Arrow, and a label with the text "foo":

```
jsPlumb.connect({
  ...
  overlays:[
    "Arrow",
      [ "Label", { label:"foo", location:0.25, id:"myLabel" } ]
    ],
  ...
});
```

This connection will have an arrow located halfway along it, and the label "foo" one quarter of the way along. Notice the **id** parameter; it can be used later if you wish to remove the Overlay or change its visibility (see below).

Another example, this time with an absolute location of 50 pixels from the source:

```
jsPlumb.connect({
  ...
  overlays:[
    "Arrow",
      [ "Label", { label:"foo", location:50, id:"myLabel" } ]
    ],
    ...
});
```

Specifying one or more overlays on a jsPlumb.addEndpoint call.

Note in this example that we use the parameter `connectorOverlays` and not `overlays` as in the last example. This is because `overlays` would refer to Endpoint Overlays:

```
jsPlumb.addEndpoint("someDiv", {
  ...
  overlays:[
    [ "Label", { label:"foo", id:"label", location:[-0.5, -0.5] } ]
  ],
  connectorOverlays:[
    [ "Arrow", { width:10, length:30, location:1, id:"arrow" } ],
    [ "Label", { label:"foo", id:"label" } ]
  ],
  ...
});
```

This connection will have a 10x30 Arrow located right at the head of the connection, and the label "foo" located at the halfway point. The Endpoint itself also has an overlay, located at [-0.5 * width, -0.5 * height] relative to the Endpoint's top,left corner.

Specifying one or more overlays on a jsPlumb.makeSource call.

Note in this example that we again use the parameter `connectorOverlays` and not `overlays`. The `endpoint` parameter to `jsPlumb.makeSource` supports everything you might pass to the second argument of a `jsPlumb.addEndpoint` call:

```
jsPlumb.makeSource("someDiv", {
  ...
  endpoint:{
    connectorOverlays:[
      [ "Arrow", { width:10, length:30, location:1, id:"arrow" } ],
      [ "Label", { label:"foo", id:"label" } ]
    ]
  }
  ...
});
```

This connection will have a 10x30 Arrow located right at the head of the connection, and the label "foo" located at the halfway point.

Using the `addOverlay` method on an Endpoint or Connection

Endpoints and Connections both have an `addOverlay` method that takes as an argument a single Overlay definition. An example:

```
var e = jsPlumb.addEndpoint("someElement");
e.addOverlay([ "Arrow", { width:10, height:10, id:"arrow" }]);
```

Overlay Types

Arrow

Draws an arrow, using four points: the head and two tail points, and a `foldback` point, which permits the tail of the arrow to be indented. Available constructor arguments for this Overlay are:

- **width** - width of the tail of the arrow
- **length** - distance from the tail of the arrow to the head
- **location** - where, either as a proportional value from 0 to 1 inclusive, or as an absolute value (negative values mean distance from target; positive values greater than 1 mean distance from source) the Arrow should appear on the Connector
- **direction** - which way to point. Allowed values are 1 (the default, meaning forwards) and -1, meaning backwards
- **foldback** - how far along the axis of the arrow the tail points foldback in to. Default is 0.623.
- **paintStyle** - a style object in the form used for paintStyle values for Endpoints and Connectors.

PlainArrow

This is just a specialized instance of `Arrow` in which jsPlumb hardcodes `foldback` to 1, meaning the tail of the Arrow is a flat edge. All of the constructor parameters from Arrow apply for PlainArrow.

Diamond

This is a specialized instance of `Arrow` in which jsPlumb hardcodes 'foldback' to 2, meaning the Arrow turns into a Diamond. All of the constructor parameters from Arrow apply for Diamond.

Label

Provides a text label to decorate Connectors with. The available constructor arguments are:

- **label** - The text to display. You can provide a function here instead of plain text: it is passed the Connection as an argument, and it should return a String.
- **cssClass** - Optional css class to use for the Label. This is now preferred over using the `labelStyle` parameter.
- **labelStyle** - Optional arguments for the label's appearance. Valid entries in this JS object are:
    - **font** - a font string in a format suitable for the Canvas element
    - **fill** - the color to fill the label's background with. Optional.
    - **color** - the color of the label's text. Optional.
    - **padding** - optional padding for the label. This is expressed as a proportion of the width of the label, not in pixels or ems.
    - **borderWidth** - optional width in pixels for the label's border. Defaults to 0.
    - **borderStyle** - optional. The color to paint the border, if there is one.
- **location** - As for Arrow Overlay. Where, either proportionally from 0 to 1 inclusive, or as an absolute offset from either source or target, the label should appear.

The Label overlay offers two methods - `getLabel` and `setLabel` - for accessing/manipulating its content dynamically:

```
var c = jsPlumb.connect({
  source:"d1",
  target:"d2",
  overlays:[
    [ "Label", {label:"FOO", id:"label"}]
  ]
});

...

var label = c.getOverlay("label");
console.log("Label is currently", label.getLabel());
label.setLabel("BAR");
console.log("Label is now", label.getLabel());
```

In this example you can see that the Label Overlay is assigned an id of "label" in the connect call, and then retrieved using that id in the call to Connection's getOverlay method.

Both Connections and Endpoints support Label Overlays, and because changing labels is quite a common operation, **setLabel** and **getLabel** methods have been added to these objects:

```
var conn = jsPlumb.connect({
  source:"d1",
  target:"d2",
  label:"FOO"
});

...

console.log("Label is currently", conn.getLabel());
conn.setLabel("BAR");
console.log("Label is now", conn.getLabel());
```

These methods support passing in a Function instead of a String, and jsPlumb will create a label overlay for you if one does not yet exist when you call setLabel:

```
var conn = jsPlumb.connect({
  source:"d1",
  target:"d2"
});

...

conn.setLabel(function(c) {
  var s = new Date();
  return s.getTime() + "milliseconds have elapsed since 01/01/1970";
});
console.log("Label is now", conn.getLabel());
```

Custom

The Custom Overlay allows you to create your own Overlays, which jsPlumb will position for you. You need to implement one method - `create(component)` - which is passed the component on which the Overlay is located as an argument, and which returns either a DOM element or a valid selector from the underlying library:

```
var conn = jsPlumb.connect({
  source:"d1",
  target:"d2",
  paintStyle:{
    stroke:"red",
    strokeWidth:3
  },
  overlays:[
    ["Custom", {
      create:function(component) {
        return $("<select id='myDropDown'><option value='foo'>foo</option><option value='bar'>bar</option>
</select>");
      },
      location:0.7,
      id:"customOverlay"
    }]
  ]
});
```

Here we have created a select box with a couple of values, assigned to it the id of 'customOverlay' and placed it at location 0.7. Note that the 'id' we assigned is distinct from the element's id. You can use the id you provided to later retrieve this Overlay using the `getOverlay(id)` method on a Connection or an Endpoint.

Hiding/Showing Overlays

You can control the visibility of Overlays using the `setVisible` method of Overlays themselves, or with `showOverlay(id)` or `hideOverlay(id)` on a Connection.

Remember the **id** parameter that we specified in the examples above? This can be used to retrieve the Overlay from a Connection:

~~~
var connection = jsPlumb.connect({
  ...
  overlays:[
    "Arrow",
    [ "Label", { label:"foo", location:0.25, id:"myLabel" } ]
  ],
  ...
});

// time passes

var overlay = connection.getOverlay("myLabel");
// now you can hide this Overlay:
overlay.setVisible(false);
// there are also hide/show methods:
overlay.show();
overlay.hide();
```

However, Connection and Endpoint also have two convenience methods you could use instead:

```javascript
var connection = jsPlumb.connect({
  ...
  overlays:[
    "Arrow",
    [ "Label", { label:"foo", location:-30 }, id:"myLabel" ]
  ],
  ...
});

// time passes

connection.hideOverlay("myLabel");

// more time passes

connection.showOverlay("myLabel");
~~~

Removing Overlays

Connection and Endpoint also have a `removeOverlay` method, that does what you might expect:

```
var connection = jsPlumb.connect({
  ...
  overlays:[
    "Arrow",
    [ "Label", { label:"foo", location:0.25 }, id:"myLabel" ]
  ],
  ...
});

// time passes

connection.removeOverlay("myLabel");
```

Groups

jsPlumb supports the concept of 'Groups', which are elements that act as the parent for a set of child elements. When a Group element is dragged, its child elements (which are, in the DOM, child nodes of the Group element) are dragged along with it. Groups may be collapsed, which causes all of their child elements to be hidden, and for any connections from a child element to outside of the Group to be "proxied" onto the Group's collapsed container.

Adding a Group

To add a Group you first need to have added it as an element in the DOM. Let's assume you have some element called `foo`. It can be any valid HTML element but since Groups are by their nature containers for other elements it's probably best to just stick to trusty `div`. So let's assume you have a variable `foo` that references a DIV element.

A simple example, using the defaults:

```
jsPlumb.addGroup({
  container:foo,
  id:"aGroup"
});
```

Here we've created a Group with ID `aGroup`. Its container element - `foo` - will be made draggable, and it will also be configured to accept elements being dropped onto it. By default, child elements will be draggable outside of the Group container, but if they are not dropped onto another Group they will revert to their position inside the Group container before they were dragged.

Several aspects of a Group's behaviour can be configured; broadly speaking these fall into two categories: the behaviour of the Group container, and the behaviour of its child elements.

Group container parameters

- **draggable** Set to true by default. If false, the Group container will not be made into a draggable element.
- **dragOptions** Options for the Group container's drag behaviour. One parameter you will likely want to consider in the `dragOptions` is `filter`, which provides a selector, or selectors, identifying elements that should not cause a drag to begin. For a Group container you will probably want to identify the child elements of the Group, so that they can be dragged without kicking off a drag of the Group container.
- **droppable** Set to true by default. If false, the Group container will not allow elements to be dropped onto it in order to add them to the Group.
- **dropOptions** Options for the Group container's drop behaviour.
- **proxied** True by default. Indicates that connections to child elements inside the Group (which emanate from outside of the Group) should be proxied, when the Group is collapsed, by connections attached to the Group's container.

Child element behaviour parameters

- **revert** By default this is true, meaning that child elements dropped outside of the Group (and not onto another Group that is accepting droppables) will revert to their last position inside the group on mouseup. If you set `revert:false` you get a Group that allows child elements to exist outside of the bounds of the Group container, but which will still drag when the Group is dragged and will be made invisible when the Group is collapsed.

- **prune** Set to false by default. If true, a child element dropped in whitespace outside of the Group container will be removed from the Group and from the instance of jsPlumb, and any connections attached to the element will also be cleaned up.

- **orphan** Set to false by default. If true, a child element dropped in whitespace outside of the Group container will be removed from the Group, but not from the instance of jsPlumb.

- **constrain** Set to false by default. If true, child elements are constrained to be dragged inside of the Group container only.

- **ghost** Set to false by default. If true, a child element that is dragged outside of the Group container will have its original element left in place, and a 'ghost' element - a clone of the original - substituted, which tracks with the mouse.

- **dropOverride** False by default. If true, child elements may be dragged outside of the Group container (assuming no other flag prevents this), but may not be dropped onto other Groups.

Groups can be removed with the `removeGroup` method:

```
jsPlumb.removeGroup("aGroup");
```

This will remove the Group with ID `aGroup`. If you also wish to remove all of the Group's child elements you can provide a second argument:

```
jsPlumb.removeGroup("aGroup", true);
```

Proxy Endpoints

You can control the location, appearance and behaviour of the Endpoints that appear when a Group is collapsed with the `anchor` and `endpoint` parameters. These take the same values as in the other parts of the API in which they appear. For instance, perhaps you want to show a smallish dot that tracks the perimeter of a Group when it is collapsed:

```
jsPlumb.addGroup({
    container:someElement,
    id:"aGroup",
    anchor:"Continuous",
    endpoint:[ "Dot", { radius:3 } ]
});
```

Perhaps you want to show a large rectangle in the top left corner:

```
jsPlumb.addGroup({
    container:someElement,
    id:"aGroup",
    anchor:"TopLeft",
    endpoint:[ "Rectangle", { width:10, height:10 } ]
});
```

etc. Any valid `anchor` or `endpoint` may be used.

Adding an element to a Group

You can add an element to a Group programmatically with the `addToGroup(group, el)` method:

```
jsPlumb.addToGroup("aGroup", someElement);
```

If you try to add an element that already belongs to some other Group an exception is thrown.

Removing an element from a Group

You can remove an element from a Group programmatically with the `removeGromGroup(el)` method:

```
jsPlumb.removeFromGroup(someElement);
```

Note that you do not need to nominate the Group from which you wish to remove the element. Since an element can belong to only one Group at a time, it will simply be removed from the Group. If you call this method with an element that does not belong to a Group, nothing happens.

Dragging and dropping child elements

By default, Groups are configured to accept elements being dropped onto them - any element that is currently being managed by the jsPlumb instance, not just existing members of other groups. There are a few ways to prevent this, both already discussed above:

- Set **droppable:false** when you create a Group:

```
jsPlumb.addGroup({
    container:someElement,
    droppable:false
});
```

This prevents the Group from accepting dropped elements.

- Set **constrain:true** when you create a Group:

```
jsPlumb.addGroup({
  container:someElement,
  constrain:true
});
```

This will prevent elements from being dragged outside of the Group.

- Set **dropOverride:true** when you create a Group:

```
jsPlumb.addGroup({
  container:someElement,
  dropOverride:true
});
```

This will prevent elements from being dropped onto some other Group.

Collapsing and expanding Groups

Use `collapseGroup` and `expandGroup`:

```
jsPlumb.collapseGroup("aGroup");

jsPlumb.expandGroup("aGroup");

```

CSS

Three CSS classes are used:

Class	Purpose
jsplumb-group-expanded	added to non-collapsed group elements (added when group initialised)
jsplumb-group-collapsed	added to collapsed group elements
jsplumb-ghost-proxy	added to the proxy element used when `ghost` is set to true

Events

A number of events are fired by the various methods for working with Groups.

Event	Description	Parameters
group:add	Fired when a new Group is added	**{group:Group}**
group:remove	Fired when a Group is removed	**{group:Group}**
group:addMember	Fired when an element is added to a Group	**{group:Group, el:Element}**
group:removeMember	Fired when an element is removed from a Group	**{group:Group, el:Element}**
group:collapse	Fired when a Group is collapsed	**{group:Group}**
group:expand	Fired when a Group is expanded	**{group:Group}**

Miscellaneous Methods

Retrieving a Group

- **getGroup(ID)** Return the Group with the given ID. May be null.

Finding out the Group to which an element belongs

- **getGroupFor(elementOrID)** - Gets the Group to which the given element (specified as a DOM element or element ID) belongs to. Returns null if the element either does not exist or does not belong to a Group.

Getting the members of a Group

- **getMembers()** - Note that this is a method on the Group object, which you either got back from a call to `addGroup` on a jsPlumb instance, or you retrieved from a jsPlumb instance using `getGroup(groupId)`.

Element Dragging

A common feature of interfaces using jsPlumb is that the elements are draggable. You should use the `draggable` method on a `jsPlumbInstance` to configure this:

```
myInstanceOfJsPlumb.draggable("elementId");
```

...because if you don't, jsPlumb won't know that an element has been dragged, and it won't repaint the element.

Allowed argument types

`draggable` supports several types as argument:

- a String representing an element ID
- an Element
- any "list-like" object whose contents are Strings or Elements.

A "list-like" element is one that exposes a `length` property and which can be indexed with brackets (`[ ]`). Some examples are:

- arrays

```
jsPlumbInstance.draggable(["elementOne", "elementTwo"]);
```

- jQuery selectors

```
jsPlumbInstance.draggable($(".someClass"));
```

- NodeLists

```
var els = document.querySelectorAll(".someClass");
jsPlumbInstance.draggable(els);
```

Options

If you absolutely cannot use `jsPlumb.draggable`, you will have to arrange to repaint the drag element manually, via `jsPlumb.repaint`.

Note `jsPlumb` is an instance of the `jsPlumbInstance` class. If you are working with your own instances of jsPlumb, be sure to call the draggable method on those instances, not the global instance.

Vanilla jsPlumb

In vanilla jsPlumb, drag support is provided by a bundled library called Katavorio. Katavorio supports drag containment and constraining elements to a grid, as well as drag/drop of multiple elements.

Required CSS

You *must* set `position:absolute` on elements that you intend to be draggable. The reasoning for this is that all libraries implement dragging by manipulating the left and top properties of an element's style attribute.

When you position an element absolute, these `left/top` values are taken to mean with respect to the origin of this element's `offsetParent`, and the `offsetParent` is the first ancestor of the element that has `position:relative` set on it, or the body if no such ancestor is found.

When you position an element relative, the `left/top` values are taken to mean *move the element from its normal position by these amounts*, where "normal position" is dependent on document flow. You might have a test case or two in which relative positioning appears to work; for each of these you could create several others where it does not.

You cannot trust relative positioning with dragging, and, despite popular opinion, jQuery's `draggable` method does not "work" with it at all, it just might seem to at times.

Drag Containment

A common request is for the ability to contain the area within which an element may be dragged. Vanilla jsPlumb offers support for containing a dragged element within the bounds of its parent:

```
jsPlumb.draggable("someElement", {
   containment:true
});
```

Dragging on a Grid

To constrain elements to a grid, supply the size of the grid in the `draggable` call:

```
jsPlumb.draggable("someElement", {
    grid:[50,50]
});
```

Dragging nested elements

jsPlumb takes nesting into account when handling draggable elements. For example, say you have this markup:

```
<div id="container">
  <div class="childType1"></div>
  <div class="childType2"></div>
</div>
```

...and then you connect one of those child divs to something, and make container draggable:

```
jsPlumb.connect({
  source:$("#container .childType1"),
  target:"somewhere else"
});

jsPlumb.draggable("container");
```

Now when you drag `container`, jsPlumb will have noticed that there are nested elements that have Connections, and they will be updated. Note that the order of operations is not important here: if container was already draggable when you connected one of its children to something, you would get the same result.

Nested Element offsets

For performance reasons, jsPlumb caches the offset of each nested element relative to its draggable ancestor. If you make changes to the draggable ancestor that will have resulted in the offset of one or more nested elements changing, *you need to tell jsPlumb about it* , using the `revalidate` function.

Consider the example from before, but with a change to the markup after initializing everything:

```
<div id="container">
  <div class="header" style="height:20px;background-color:blue;">header</div>
  <div class="childType1"></div>
  <div class="childType2"></div>
<div>
```

Connect a child div to some element and make it draggable:

```
jsPlumb.connect({
  source:$("#container .childType1"),
  target:"somewhere else"
});

jsPlumb.draggable($(".childType1"));
```

Now if you manipulate the DOM in such a way that the internal layout of the container node is changed, you need to tell jsPlumb:

```
$("#container .header").hide();    // hide the header bar. this will alter the offset of the other child ele
ments...
jsPlumb.revalidate("container");   // tell jsPlumb that the internal dimensions have changed.
// you can also use a selector, eg $("#container")
```

Note in this example, one of the child divs was made draggable. jsPlumb automatically recalculates internal offsets after dragging stops in that case - you do not need to call it yourself.

Dragging Multiple Elements

Vanilla jsPlumb, because it uses Katavorio as its drag library, supports dragging multiple elements at once. There are two ways to do this.

The drag selection

Every jsPlumb instance has an associated `dragSelection` - a set of elements that are considered to be "selected" (and which have a CSS class assigned to them to indicate this fact). Any drag event occurring in the instance will cause the currently selected elements to be dragged too.

There are three methods provided to work with the drag selection:

- **addToDragSelection: function (elements)** Add the specified elements to the drag selection.
- **removeFromDragSelection: function (elements)** Remove the specified elements from the drag selection
- **clearDragSelection** Clear the drag selection.

As previously mentioned, these methods are all provided by Katavorio.

Here, `elements` can be a single element Id or element, or an array of either of these (or, in fact, a jQuery selector, since jsPlumb knows that selectors are "list-like").

Posses

Since jsPlumb 2.0.0 there is another way of configuring multiple element drag - the concept of a `posse`. This is a group of elements that should all move whenever one of them is dragged. This mechanism is intended to be used for more "permanent" multiple drag arrangements - you may have a set of nodes that should always move together and they do not need to be considered to be "selected".

You can define multiple posses in an instance but each element can belong to one posse only. Three methods are provided for working wth Posses:

- **addToPosse(elements, posse...)**

Here, `elements` can be a single element Id or element, or an array of either of these (or, in fact, a jQuery selector, since jsPlumb knows that selectors are "list-like"). `posse` is required, and is a varargs parameter: you can provide more than one. Each `posse` parameter can either be a String, indicating a Posse to which the element should be added as an active drag participant, or an object containing the Posse's ID and also whether or not the element should be an active drag participant.

For example, in the Flowchart demo, we could make these couple of calls:

```
jsPlumb.addToPosse(["flowchartWindow1", "flowchartWindow2"], "posse");
jsPlumb.addToPosse("flowchartWindow3", {id:"posse",active:false})
```

And the result would be that whenever window 1 or 2 was dragged, all of windows 1, 2 and 3 would be dragged. But if window 3 was dragged, it would be dragged alone; windows 1 and 2 would not move.

- **removeFromPosse(elements, posseId)**

Remove the given element(s) from Posse(s). As with `addToPosse`, the second parameter is a varargs parameter: you can supply a number of posse IDs at once:

```
jsPlumb.removeFromPosse("flowchartWindow1", "posse1", "posse2");
```

- **removeFromAllPosses

Remove the given element(s) from all of the Posses to which it/they belong.

```
jsPlumb.removeFromAllPosses("flowchartWindow1");
jsPlumb.removeFromAllPosses(["flowchartWindow2", "flowchartWindow3"]);
```

What's with the name?

The name **posse** was chosen over **group** because `group` is something that will be supported in the future, having one key difference from a `posse`: a `group` of elements will have a common parent as their container.

Text Selection while dragging

The default browser behaviour on mouse drag is to select elements in the DOM. jQuery suppresses this behaviour, but vanilla jsPlumb does not. To assist with handling this, however, this class is attached to the body at drag start:

`jsplumb-drag-select`

The class is removed at drag end.

A suitable value for this class (this is from the jsPlumb demo pages) is:

```
.jsplumb-drag-select * {
    -webkit-touch-callout: none;
    -webkit-user-select: none;
    -khtml-user-select: none;
    -moz-user-select: none;
    -ms-user-select: none;
    user-select: none;
}
```

Connections

Programmatic Connections

The most simple connection you can make with jsPlumb looks like this:

```
jsPlumb.connect({source:"element1", target:"element2"});
```

In this example we have created a Connection from 'element1' to 'element2'. Remember that a Connection in jsPlumb consists of two Endpoints, a Connector, and zero or more Overlays. But this call to 'connect' supplied none of those things, so jsPlumb uses the default values wherever it needs to. In this case, default values have been used for the following:

- The type and appearance of each Endpoint in the Connection. jsPlumb's default for this is the "Dot" endpoint, of radius 10, with fill color "#456".
- The Anchors that define where the Connection's Endpoints appear on each element. The jsPlumb default is `Bottom`.
- Whether or not each Endpoint can be a source or target for new Connections. The default is false.
- The type and appearance of the Connection's Connector. The default is a "Bezier" connector of line width 8, and color "#456".

So this call will result in an 8px Bezier, colored "#456", from the bottom center of 'element1' to the bottom center of 'element2', and each Endpoint will be a 10px radius Dot Endpoint, colored "#456".

Let's beef up this call a little and tell jsPlumb what sort of Endpoints we want, and where we want them:

```
jsPlumb.connect({
  source:"element1",
  target:"element2",
  anchors:["Right", "Left" ],
  endpoint:"Rectangle",
  endpointStyle:{ fill: "yellow" }
});
```

This is what we have told jsPlumb we want this time:

- **anchors** - this array tells jsPlumb where the source and target Endpoints should be located on their parent elements. In this case, we use the shorthand syntax to name one of jsPlumb's default anchors; you can also specify custom locations (see Anchors. Instead of `anchors` you can use `anchor`, if you want the source and target Endpoints to be located at the same place on their parent elements.
- **endpoint** - this tells jsPlumb to use the `Rectangle` Endpoint for both the source and target of the Connection. As with anchors, `endpoint` has a plural version that allows you to specify a different Endpoint for each end of the Connection.
- **endpointStyle** - this is the definition of the appearance of the Endpoint you specified above. Again, there is a plural equivalent of this that allows you to specify a different style for each end of the Connection. For more information about allowed values for this value, see Connector, Endpoint, Anchor & Overlay definitions.

Reusing common settings between jsPlumb.connect calls

A fairly common situation you will find yourself in is wanting to create a bunch of Connections that have only minor differences between them. To support that, `jsPlumb.connect` takes an optional second argument. For example:

```
var common = {
  anchors:[ "BottomCenter", "TopCenter" ],
  endpoints:["Dot", "Blank" ]
};

jsPlumb.connect({ source:"someElement", target:"someOtherElement" }, common);

jsPlumb.connect({ source:"aThirdElement", target:"yetAnotherElement" }, common);
```

Endpoints created by jsPlumb.connect

If you supply an element id or selector for either the source or target, jsPlumb.connect will automatically create an Endpoint on the given element. These automatically created Endpoints are not marked as drag source or targets, and cannot be interacted with. For some situations this behaviour is perfectly fine, but for more interactive UIs you should set things up using one of the drag and drop methods discussed below.

Note: given that `jsPlumb.connect` creates its own Endpoints in some circumstances, in order to avoid leaving orphaned Endpoints around the place, if the Connection is subsequently deleted, these automatically created Endpoints are deleted too. Should you want to, you can override this behaviour by setting `deleteEndpointsOnDetach` to false in the connect call:

```
jsPlumb.connect({
  source:"aThirdElement",
  target:"yetAnotherElement",
  deleteEndpointsOnDetach:false
});
```

Detaching Connections

By default, connections made with `jsPlumb.connect` will be detachable via the mouse. You can prevent this by either setting an appropriate default value:

```
jsPlumb.importDefaults({
  ...
  ConnectionsDetachable:false
  ...
});
```

...or by specifying it on the connect call like this:

```
jsPlumb.connect({
  source:"aThirdElement",
  target:"yetAnotherElement",
  detachable:false
});
```

Drag and Drop Connections

To support drag and drop connections, you first need to set a few things up. Every drag and drop connection needs at least a source Endpoint that the user can drag a connection from. Here's a simple example of how to create an Endpoint:

```
var endpointOptions = { isSource:true };
var endpoint = jsPlumb.addEndpoint('elementId', endpointOptions);
```

This Endpoint will act as a source for new Connections, and will use the jsPlumb defaults for its own appearance and that of any Connections that are drawn from it.

Tip: use the three-argument addEndpoint method for common data

One thing that happens quite often is that you have an Endpoint whose appearance and behaviour is largely the same between usages on different elements, with just a few differences.

```
var exampleGreyEndpointOptions = {
  endpoint:"Rectangle",
  paintStyle:{ width:25, height:21, fill:'#666' },
  isSource:true,
  connectorStyle : { stroke:"#666" },
  isTarget:true
};
```

Notice there is no `anchor` set. Here we apply it to two elements, at a different location in each:

```
jsPlumb.addEndpoint("element1", {
  anchor:"Bottom"
}, exampleGreyEndpointOptions));

jsPlumb.addEndpoint("element2", {
  anchor:"Top"
}, exampleGreyEndpointOptions));
```

Now that you have a source Endpoint, you need to either create a target Endpoint on some element, or notify jsPlumb that you wish to make an entire element a drop target. Let's look at how to attach a target Endpoint first:

```
var endpointOptions = { isTarget:true, endpoint:"Rectangle", paintStyle:{ fill:"gray" } };
var endpoint = jsPlumb.addEndpoint("otherElementId", endpointOptions);
```

This Endpoint, a gray rectangle, has declared that it can act as a drop target for Connections.

Elements as sources & targets

jsPlumb also supports turning entire elements into Connection sources and targets, using the methods `makeSource` and `makeTarget`. With these methods you mark an element as a source or target, and provide an Endpoint specification for jsPlumb to use when a Connection is established. `makeSource` also gives you the ability to mark some child element as the place from which you wish to drag Connections, but still have the Connection on the main element after it has been established.

These methods honour the jsPlumb defaults - if, for example you set up the default Anchors to be this:

```
jsPlumb.importDefaults({
  Anchors : [ "Left", "BottomRight" ]
});
```

... and then used `makeSource` and `makeTarget` without specifying Anchor locations, jsPlumb would use `Left` for the `makeSource` element and `BottomRight` for the `makeTarget` element.

A further thing to note about `makeSource` and `makeTarget` is that any prior calls to one of these methods is honoured by subsequent calls to `jsPlumb.connect`. This helps when you're building a UI that uses this functionality at runtime but which loads some initial data, and you want the statically loaded data to have the same appearance and behaviour as dynamically created Connections (obviously quite a common use case).

For example:

```
jsPlumb.makeSource("el1", {
  anchor:"Continuous",
  endpoint:["Rectangle", { width:40, height:20 }],
  maxConnections:3
});

...

jsPlumb.connect({source:"el1", target:"el2"});
```

In this example, the source of the connection will be a Rectangle of size 40x20, having a Continuous Anchor. The source is also configured to allow a maximum of three Connections.

You can override this behaviour by setting a `newConnection` parameter on the connect call:

```
jsPlumb.makeSource("el1", {
  anchor:"Continuous",
  endpoint:["Rectangle", { width:40, height:20 }],
  maxConnections:1,
  onMaxConnections:function(params, originalEvent) {
      // params contains:
      // {
      //    endpoint:..
      //    connection:...
      //    maxConnections:N
      // }
      //
  }
});

...

jsPlumb.connect({
  source:"el1",
  target:"el2",
  newConnection:true
});
```

Note the `onMaxConnections` parameter to this call - it allows you to supply a function to call if the user tries to drag a new Connection when the source has already reached capacity.

Reference Parameters

As with `jsPlumb.connect`, `makeSource` can take an optional third argument consisting of parameters that may be common across several different calls:

```
var common = {
  anchor:"Continuous",
  endpoint:["Rectangle", { width:40, height:20 }],
};

jsPlumb.makeSource("el1", {
  maxConnections:1,
  onMaxConnections:function(params, originalEvent) {
    console.log("element is ", params.endpoint.element, "maxConnections is", params.maxConnections);
  }
}, common);
```

jsPlumb.makeTarget

This method takes two arguments, the first of which specifies some element (or list of elements); the second specifies the Endpoint you wish to create on that element whenever a Connection is established on it. In this example we will use the exact same target Endpoint we used before - the gray rectangle - but we will tell jsPlumb that the element `aTargetDiv` will be the drop target:

```
var endpointOptions = {
  isTarget:true,
  maxConnections:5,
  endpoint:"Rectangle",
  paintStyle:{ fill:"gray" }
};
jsPlumb.makeTarget("aTargetDiv", endpointOptions);
```

The allowed values in 'endpointOptions' are identical for both the `jsPlumb.addEndpoint` and `jsPlumb.makeTarget` methods, but `makeTarget` supports an extended Anchor syntax that allows you more control over the location of the target endpoint.
This is discussed below.

Notice in the `endpointOptions` object above there is a parameted called `isTarget` - this may seem incongruous, since you know you're going to make some element a target. Remember that the `endpointOptions` object is the information jsPlumb will use to create an Endpoint on the given target element each time a Connection is established to it. It takes the exact same format as you would pass to `addEndpoint`; `makeTarget` is essentially a deferred `addEndpoint` call followed by a `connect` call. So in this case, we're telling jsPlumb that any Endpoints it happens to create on some element that was configured by the `makeTarget` call are themselves Connection targets. You may or may not want this behaviour in your application - just control it by setting the approriate value for that parameter (it defaults to false).

`makeTarget` also supports the `maxConnections` and `onMaxConnections` parameters, as `makeSource` does, but note that `onMaxConnections` is passed one extra parameter than its corresponding callback from `makeSource` - the Connection the user tried to drop:

```
jsPlumb.makeTarget("aTargetDiv", {
  isTarget:true,
  maxConnections:5,
  endpoint:"Rectangle",
  paintStyle:{ fill:"gray" },
  maxConnections:3,
  onMaxConnections:function(params, originalEvent) {
    console.log("user tried to drop connection", params.connection, "on element", params.endpoint.element,
"with max connections", params.maxConnections);
  }
};
```

Reference Parameters

As with `jsPlumb.connect` and `jsPlumb.makeSource`, `jsPlumb.makeTarget` can take an optional third argument consisting of parameters that may be common across several different calls:

```
var common = {
  anchor:"Continuous",
  endpoint:["Rectangle", { width:40, height:20 }],
};

jsPlumb.makeTarget("el1", {
  maxConnections:1,
  onMaxConnections:function(params, originalEvent) {
    console.log("element is ", params.endpoint.element, "maxConnections is", params.maxConnections);
  }
}, common);
```

Preventing Loopback Connections

In vanilla jsPlumb only, you can instruct jsPlumb to prevent loopback connections without having to resort to a beforeDrop interceptor. You do this by setting `allowLoopback:false` on the parameters passed to the `makeTarget` method:

```
jsPlumb.makeTarget("foo", {
  allowLoopback:false
});
```

Unique Endpoint per Target

jsPlumb will create a new Endpoint using the supplied information every time a new Connection is established on the target element, by default, but you can override this behaviour and tell jsPlumb that it should create at most one Endpoint, which it should attempt to use for subsequent Connections:

```
var endpointOptions = {
  isTarget:true,
  uniqueEndpoint:true,
  endpoint:"Rectangle",
  paintStyle:{ fill:"gray" }
};
jsPlumb.makeTarget("aTargetDiv", endpointOptions);
```

Here, the `uniqueEndpoint` parameter tells jsPlumb that there should be at most one Endpoint on this element. Notice that `maxConnections` is not set: the default is 1, so in this setup we have told jsPlumb that `aTargetDiv` can receive one Connection and no more.

Deleting Endpoints on detach

By default, any Endpoints created using makeTarget have `deleteEndpointsOnDetach` set to true, which means that once all Connections to that Endpoint are removed, the Endpoint is deleted. You can override this by setting the flag to true on the `makeTarget` call:

```
var endpointOptions = {
  isTarget:true,
  maxConnections:5,
  uniqueEndpoint:true,
  deleteEndpointsOnDetach:false,
  endpoint:"Rectangle",
  paintStyle:{ fill:"gray" }
};
jsPlumb.makeTarget("aTargetDiv", endpointOptions);
```

In this setup we have told jsPlumb to create an Endpoint the first time `aTargetElement` receives a Connection, and to not delete it even if there are no longer any Connections to it. The created Endpoint will be reused for subsequent Connections, and can support a maximum of 5.

Detaching connections made with the mouse

As with `jsPlumb.connect`, Connections made with the mouse after setting up Endpoints with one of the functions we've just covered will be, by default, detachable. You can prevent this in the jsPlumb defaults, as previously mentioned:

```
jsPlumb.importDefaults({
  ...
  ConnectionsDetachable:false,
  ...
});
```

And you can also set this on a per-endpoint (or source/target) level, like in these examples:

```
jsPlumb.addEndpoint("someElementId", {
  connectionsDetachable:false
});

jsPlumb.makeSource("someOtherElement", {
  ...
  connectionsDetachable:false,
  ...
});

jsPlumb.makeTarget("yetAnotherElement", {
  ...
  connectionsDetachable:false,
  ...
});
```

Note that in the jsPlumb defaults, by convention each word in a parameter is capitalised ("ConnectionsDetachable"), whereas for a call to one of these methods, we use camel case ("connectionsDetachable").

Target Anchors positions with makeTarget

When using the `makeTarget` method, jsPlumb allows you to provide a callback function to be used to determine the appropriate location of a target Anchor for every new Connection dropped on the given target. It may be the case that you want to take some special action rather than just relying on one of the standard Anchor mechanisms.

This is achieved through an extended Anchor syntax (note that this syntax is **not supported** in the `jsPlumb.addEndpoint` method) that supplies a "positionFinder" to the anchor specification. jsPlumb provides two of these by default; you can register your own on jsPlumb and refer to them by name, or just supply a function. Here's a few examples:

- Instruct jsPlumb to place the target anchor at the exact location at which the mouse button was released on the target element. Note that you tell jsPlumb the anchor is of type "Assign", and you then provide a "position" parameter, which can be the name of some position finder, or a position finder function itself:

```
jsPlumb.makeTarget("someElement", {
  anchor:[ "Assign", {
    position:"Fixed"
  }]
});
```

`Fixed` is one of the two default position finders provided by jsPlumb. The other is `Grid`:

```
jsPlumb.makeTarget("someElement", {
  anchor:[ "Assign", {
    position:"Grid",
    grid:[3,3]
  }]
});
```

The Grid position finder takes a `grid` parameter that defines the size of the grid required. [3,3] means 3 rows and 3 columns.

Supplying your own position finder

To supply your own position finder to jsPlumb you first need to create the callback function. First let's take a look at what the source code for the `Grid` position finder looks like:

```
function(eventOffset, elementOffset, elementSize, constructorParams) {
  var dx = eventOffset.left - elementOffset.left, dy = eventOffset.top - elementOffset.top,
      gx = elementSize[0] / (constructorParams.grid[0]),
      gy = elementSize[1] / (constructorParams.grid[1]),
      mx = Math.floor(dx / gx), my = Math.floor(dy / gy);

  return [ ((mx * gx) + (gx / 2)) / elementSize[0], ((my * gy) + (gy / 2)) / elementSize[1] ];
}
```

The four arguments are: - **eventOffset** - Page left/top where the mouse button was released (a JS object containing left/top members like you get from a jQuery offset call) - **elementOffset** - JS offset object containing offsets for the element on which the Connection is to be created - **elementSize** - [width, height] array of the dimensions of the element on which the Connection is to be created - **constructorParams** - the parameters that were passed to the Anchor's constructor. In the example given above, those parameters are 'position' and 'grid'; you can pass arbitrary parameters.

The return value of this function is an array of [x, y] - proportional values between 0 and 1 inclusive, such as you can pass to a static Anchor.

To make your own position finder you need to create a function that takes those four arguments and returns an [x, y] position for the anchor, for example:

```
jsPlumb.AnchorPositionFinders.MyFinder = function(dp, ep, es, params) {
  ... do some maths ...
  console.log("my custom parameter is ", params.myCustomParameter);
  return [ x, y ];
};
```

Then refer to it in a makeTarget call:

```
jsPlumb.makeTarget("someElement", {
  anchor:[ "Assign", {
    position:"MyFinder",
    myCustomParameter:"foo",
    anInteger:5
  }]
});
```

jsPlumb.makeSource

There are two use cases supported by this method. The first is the case that you want to drag a Connection from the element itself and have an Endpoint attached to the element when a Connection is established. The second is a more specialised case: you want to drag a Connection from the element, but once the Connection is established you want jsPlumb to move it so that its source is on some other element.

Here's an example code snippet for the basic use case of makeSource:

```
jsPlumb.makeSource(someDiv, {
  paintStyle:{ fill:"yellow" },
  endpoint:"Blank",
  anchor:"BottomCenter"
});
```

Notice again that the second argument is the same as the second argument to an `addEndpoint` call. `makeSource` is, essentially, a type of `addEndpoint` call. In this example we have told jsPlumb that we will support dragging Connections directly from `someDiv`. Whenever a Connection is established between `someDiv` and some other element, jsPlumb assigns an Endpoint at BottomCenter of `someDiv`, fills it yellow, and sets that Endpoint as the newly created Connection's source.

Specifying drag source area

Configuring an element to be an entire Connection source using `makeSource` means that the element cannot itself be draggable. There would be no way for jsPlumb to distinguish between the user attempting to drag the element and attempting to drag a Connection from the element. To handle this there is the `filter` parameter.

Supplying a `filter` parameter

You can supply a `filter` parameter to the makeSource call, which can be either a function or a selector. Consider this markup:

```
<div id="foo">
  <span>FOO</span>
  <button>click me</button>
<div>
```

Let's suppose we do not want to interfere with the operation of the "click me" button.

Selector Filter

We can supply a selector filter to the `makeSource` call to do so:

```
jsPlumb.makeSource("foo", {
  filter:":not(button)"
});
```

Valid values for a filter selector are as per the spec.

Negating a Selector Filter

One of the limitations of CSS3 selectors is that the contents of a `:not` selector are restricted to what are called **simple selectors**. What this means is that only one item may appear inside a `:not`, eg. `:not(.someClass)`, `:not(button)` etc. You cannot, however, do something like this: `:not(.myClass button)`.

To allow for these sorts of negation selectors, you can set the `filterExclude:true` flag and re-write your selector:

```
jsPlumb.makeSource("foo", {
  filter:"span",
  filterExclude:true
});
```

Filter Function

Alternatively, you can supply a function as the filter:

```
jsPlumb.makeSource("foo", {
  filter:function(event, element) {
    return event.target.tagName !== "BUTTON";
  }
});
```

In this example, if the filter returns anything other than a boolean false, the drag will begin. It's important to note that only boolean false will prevent a drag. False-y values will not.

Endpoint options with makeSource

There are many things you can set in an Endpoint options object; for a thorough list see the API documentation for Endpoint.

Here's an example of specifying that you want an Arrow overlay halfway along any Connection dragged from this Endpoint:

```
var exampleGreyEndpointOptions = {
  endpoint:"Rectangle",
  paintStyle:{ width:25, height:21, fill:'#666' },
  isSource:true,
  connectorStyle : { stroke:"#666" },
  isTarget:true,
  connectorOverlays: [ [ "Arrow", { location:0.5 } ] ]
};
```

This is an Endpoint that moves around the element it is attached to dependent on the location of other elements in the connections it is attached to (a 'dynamic' anchor):

```
var exampleGreyEndpointOptions = {
  endpoint:"Rectangle",
  paintStyle:{ width:25, height:21, fill:'#666' },
  isSource:true,
  connectorStyle : { stroke:"#666" },
  isTarget:true,
  connectorOverlays: [ [ "Arrow", { location:0.5 } ] ]
  anchor:[ "TopCenter","RightMiddle","BottomCenter","LeftMiddle" ]
};
```

Testing if an element is a target or source

You can test if some element has been made a connection source or target using these methods:

- `isTarget(some id, selector or DOM element)`
- `isSource(some id, selector or DOM element)`

The return value is a boolean.

Toggling an element as connection target or source

You can toggle the state of some source or target using these methods:

- `setTargetEnabled(some id, selector or DOM element)`
- `setSourceEnabled(some id, selector or DOM element)`

The return value from these methods is the current jsPlumb instance, allowing you to chain them:

`jsPlumb.setTargetEnabled("aDivId").setSourceEnabled($(".aSelector"));`

Note that if you call either of these methods on an element that was not originally configured as a target or source, nothing will happen.

You can check the enabled state of some target or source using these methods:

- `isTargetEnabled(some id, selector or DOM element)`
- `isSourceEnabled(some id, selector or DOM element)`

Canceling previous `makeTarget` and/or `makeSource` calls

jsPlumb offers four methods to let you cancel previous `makeTarget` or `makeSource` calls. Each of these methods returns the current jsPlumb instance, and so can be chained:

- `unmakeTarget(some id, selector or DOM element)`
- `unmakeSource(some id, selector or DOM element)`
- `unmakeEveryTarget`
- `unmakeEverySource`

These last two are analogous to the `removeEveryConnection` and `deleteEveryEndpoint` methods that have been in jsPlumb for a while now.

`unmakeTarget` and `unmakeSource` both take as argument the same sorts of values that `makeTarget` and `makeSource` accept - a string id, or a selector, or an array of either of these:

```
jsPlumb.unmakeTarget("aDivId").unmakeSource($(".aSelector"));
```

Drag Options

These are options that will be passed through to the supporting library's drag API. jsPlumb passes everything you supply here through, inserting wrapping functions if necessary for the various lifecycle events that jsPlumb needs to know about. So if, for example, you pass a function to be called when dragging starts, jsPlumb will wrap that function with a function that does what jsPlumb needs to do, then call yours.

At the time of writing, jsPlumb supports jQuery, as well as having its own built-in support (actually supplied by a bundled library called [Katavorio](). If you're using jQuery you can supply jQuery drag option as values on the dragOptions; the best drag library to use, though, is Katavorio, which is bundled in vanilla jsPlumb from 1.6.0 onwards: it supports everything jQuery drag supports, as well as multiple scopes and multiple element dragging.

Given that the options here are library-specific, and they are all well-documented, we're going to discuss just the three drag options that behave the same way in all (see below for hoverClass):

- **opacity** - the opacity of an element that is being dragged. Should be a fraction between 0 and 1 inclusive.
- **zIndex** - the zIndex of an element that is being dragged.
- **scope** - the scope of the draggable. can only be dropped on a droppable with the same scope. this is discussed below.

For more information about drag options, take a look at the jQuery or Katavorio docs.

Drop Options

Drop options are treated by jsPlumb in the same way as drag options - they are passed through to the underlying library.

Here are three common jQuery/Katavorio droppable options that you might want to consider using:

- **hoverClass** - the CSS class to attach to the droppable when a draggable is hovering over it.
- **activeClass** - the CSS class to attach to the droppable when a draggable is, um, being dragged.
- **scope** - the scope of the draggable. The draggable can only be dropped on a droppable with the same scope. This is discussed below.

For more information about drop options when using jQuery, see here

Drag and Drop Scope

jsPlumb borrowed the concept of 'scope' from jQuery's drag/drop implementation: the notion of which draggables can be dropped on which droppables. In jsPlumb you can provide a 'scope' entry when creating an Endpoint. Here's the example grey Endpoint example with 'scope' added:

```
var exampleGreyEndpointOptions = {
  endpoint:"Rectangle",
  paintStyle:{ width:25, height:21, fill:"#666" },
  isSource:true,
  connectionStyle : { stroke:"#666" },
  isTarget:true,
  scope:"exampleGreyConnection"
};
```

If you do not provide a 'scope' entry, jsPlumb uses a default scope. Its value is accessible through this method:

`jsPlumb.getDefaultScope();`

If you want to change it for some reason you can do so with this method:

`jsPlumb.setDefaultScope("mySpecialDefaultScope");`

You can also, should you want to, provide the scope value separately on the drag/drop options, like this:

```
var exampleGreyEndpointOptions = {
  endpoint:"Rectangle",
  paintStyle:{ width:25, height:21, fill:'#666' },
  isSource:true,
  connectionStyle : { stroke:"#666" },
  isTarget:true,
  dragOptions:{ scope:"dragScope" },
  dropOptions:{ scope:"dropScope" }
};
```

Multiple Scopes

If you're using vanilla jsPlumb you can assign multiple scopes to some Endpoint, or element configured via `makeSource` or `makeTarget`. To do so, provide the scopes as a space-separated list, as you would with CSS classes:

```
var exampleGreyEndpointOptions = {
  endpoint:"Rectangle",
  paintStyle:{ width:25, height:21, fill:"#666" },
  isSource:true,
  connectionStyle : { stroke:"#666" },
  isTarget:true,
  scope:"foo bar baz"
};
```

Changing the scope(s) of a source or target element

jsPlumb offers a few methods for changing the scope(s) of some element configured via `makeSource` or `makeTarget`:

- `setScope(el, scopeString)` Sets both the source *and* target scope(s) for the given element.
- `setSourceScope(el, scopeString)` Sets the source scope(s) for the given element.
- `setTargetScope(el, scopeString)` Sets both the target scope(s) for the given element.

Disconnecting and Reconnecting

By default, jsPlumb will allow you to detach connections from either Endpoint by dragging (assuming the Endpoint allows it; Blank Endpoints, for example, have nothing you can grab with the mouse). If you then drop a Connection you have dragged off an Endpoint, the Connection will be detached. This behaviour can be controlled using the `detachable` and `reattach`parameters, or their equivalents in the jsPlumb Defaults.

Some examples should help explain:

- Create a Connection that is detachable using the mouse, from either Endpoint, and which does not reattach:

```
jsPlumb.connect({ source:"someElement", target:"someOtherElement" });
```

- Create a Connection that is not detachable using the mouse:

```
jsPlumb.connect({ source:"someElement", target:"someOtherElement", detachable:false });
```

- Create a Connection that is detachable using the mouse and which reattaches on drop:

```
jsPlumb.connect({ source:"someElement", target:"someOtherElement", reattach:true });
```

This behaviour can also be controlled using jsPlumb defaults:

```
jsPlumb.importDefaults({
  ConnectionsDetachable:true,
  ReattachConnections:true
});
```

The default value of ConnectionsDetachable is **true**, and the default value of ReattachConnections is **false**, so in actual fact those defaults are kind of pointless. But you probably get the idea.

Setting detachable/reattach on Endpoints

Endpoints support the `detachable` and `reattach` parameters too. If you create an Endpoint and mark `detachable:false`, then all Connections made from that Endpoint will not be detachable. However, since there are two Endpoints involved in any Connection, jsPlumb takes into account the `detachable` and `reattach` parameters from both Endpoints when establishing a Connection. If either Endpoint declares either of these values as true, jsPlumb assumes the value to be true. It is possible that in a future version of jsPlumb the concepts of detachable and reattach could be made more granular, through the introduction of parameters like `sourceDetachable/targetReattach` etc.

Dropping a dragged Connection on another Endpoint

If you drag a Connection from its target Endpoint you can then drop it on another suitable target Endpoint - suitable meaning that it is of the correct scope and it is not full. If you try to drop a Connection on another target that is full, the drop will be aborted and then the same rules will apply as if you had dropped the Connection in whitespace: if `reattach` is set, the Connection will reattach, otherwise it will be removed.

You can drag a Connection from its source Endpoint, but you can only drop it back on its source Endpoint - if you try to drop it on any other source or target Endpoint jsPlumb will treat the drop as if it happened in whitespace. Note that there is an issue with the `hoverClass` parameter when dragging a source Endpoint: target Endpoints are assigned the hover class, as if you could drop there. But you cannot; this is caused by how jsPlumb uses the underlying library's drag and drop, and is something that will be addressed in a future release.

Removing Nodes

If you have configured a DOM element with jsPlumb in any way you should use jsPlumb to remove the element from the DOM (as opposed to using something like jQuery's `remove` function, for example).

To help you with this, jsPlumb offers two methods:

jsPlumb.remove

This removes an element from the DOM and all Connections and Endpoints associate with that element:

```
var conn = jsPlumb.connect({source:"element1", target:"element2"});
...
jsPlumb.remove("element1");
```

`conn` is now detached and `element1` is gone from the DOM.

You can also pass a selector or DOM element to the `remove` method.

jsPlumb.empty

This removes all the child elements from some element, and all of the Connections and Endpoints associated with those child elements. Perhaps you have this markup:

```
<ul id="list">
  <li id="one">One</li>
</ul>
```

```
var conn = jsPlumb.connect({source:"one", target:"someOtherElement"});
...
jsPlumb.empty("list");
```

`conn` is now detached and the UL is empty.

You can also pass a selector or DOM element to the `empty` method.

Removing Connections/Endpoints

There are a number of different functions you can use to remove Connections and/or Endpoints.

Connections

Detaching a single connection

To remove a single Connection, use `jsPlumb.detach`:

```
var conn = jsPlumb.connect({ some params});
...
jsPlumb.detach(conn);
```

When you call `jsPlumb.detach` to remove a Connection, the Endpoints associated with that Connection may or may not also be deleted - it depends on the way the Connection was established. The Endpoints *will* be deleted under the following circumstances:

- you created the Connection using `jsPlumb.connect` and you did not set `deleteEndpointsOnDetach:false`.
- the Connection was created via the mouse by a user on an element configured via `makeSource` which did not have `deleteEndpointsOnDetach:false` set.

The Endpoints *will not* be deleted under the following circumstances:

- you created the Connection using `jsPlumb.connect` and you set `deleteEndpointsOnDetach:false`
- the Connection was created via the mouse by a user from an Endpoint registered with `addEndpoint`.
- the Connection was created via the mouse by a user on an element configured via `makeSource` which had `deleteEndpointsOnDetach:false` set.

Detaching all Connections from a single element

To detach all the Connections from some given element:

```
jsPlumb.detachAllConnections(el, [params])
```

el may be:

- a String representing an element id
- a DOM element
- a selector representing a single element

params is optional and may contain:

- **fireEvent** - whether or not to fire a disconnection event. The default is true.

Detaching all Connections from every element

To detach every Connection in jsPlumb:

```
jsPlumb.detachEveryConnection();
```

This leaves all Endpoints in place according to the deletion rules outlined in the description of `jsPlumb.detach` above.

Endpoints

Deleting a single Endpoint

To delete a single Endpoint:

```
var ep = jsPlumb.addEndpoint(someElement, { ... });
...
jsPlumb.deleteEndpoint(ep);
```

ep may be either:

- a String, representing an Endpoint's UUID (when you add an Endpoint you can provide a `uuid` member for that Endpoint)
- an actual Endpoint object (as in the example above)

Deleting every Endpoint

To delete every Endpoint in jsPlumb:

```
jsPlumb.deleteEveryEndpoint();
```

This has the effect of removing every Endpoint and also every Connection.

Note this method is quite similar to `jsPlumb.reset`, except that this method does not remove any event handlers that have been registered.

Connection & Endpoint Parameters

jsPlumb has a mechanism that allows you to set/get parameters on a per-connection basis. These are not parameters that affect the appearance of operation of the object on which they are set; they are a means for you to associate information with jsPlumb objects. This can be achieved in a few different ways:

- Providing parameters to a `jsPlumb.connect` call
- Providing parameters to a `jsPlumb.addEndpoint` call to/from which a connection is subsequently established using the mouse
- using the `setParameter` or `setParameters` method on a Connection

Getting parameters is achieved through either the `getParameter(key)` or `getParameters` method on a Connection.

jsPlumb.connect

Parameters can be passed in via an object literal to a jsPlumb.connect call:

```
var myConnection = jsPlumb.connect({
    source:"foo",
    target:"bar",
    parameters:{
        "p1":34,
        "p2":new Date(),
        "p3":function() { console.log("i am p3"); }
    }
});
```

Note that they can be any valid Javascript objects - it's just an object literal. You then access them like this:

```
myConnection.getParameter("p3")();     // prints 'i am p3' to the console.
```

jsPlumb.addEndpoint

The information in this section also applies to the `makeSource` and `makeTarget` functions.

Using `jsPlumb.addEndpoint`, you can set parameters that will be copied in to any Connections that are established from or to the given Endpoint using the mouse. (If you set parameters on both a source and target Endpoints and then connect them, the parameters set on the target Endpoint are copied in first, followed by those on the source. So the source Endpoint's parameters take precedence if they happen to have one or more with the same keys as those in the target).

Consider this example:

```
var e1 = jsPlumb.addEndpoint("d1", {
    isSource:true,
    parameters:{
        "p1":34,
        "p2":new Date(),
        "p3":function() { console.log("i am p3"); }
    }
});

var e2 = jsPlumb.addEndpoint("d2", {
    isTarget:true,
    parameters:{
        "p5":343,
        "p3":function() { console.log("FOO FOO FOO"); }
    }
});

var conn = jsPlumb.connect({source:e1, target:e2});
```

'conn' will have four parameters set on it, with the value for "p3" coming from the source Endpoint:

```
var params = conn.getParameters();
console.log(params.p1);     // 34
console.log(params.p2);     // Mon May 14 2012 12:57:12 GMT+1000 (EST) (or however your console prints out a Date)
console.log((params.p3)()); // "i am p3"  (note: we executed the function after retrieving it)
console.log(params.p5);     // 343
```

Connection and Endpoint Types

Introduction

A Type is a collection of attributes such as paint style, hover paint style, overlays etc - it is a subset, including most but not all, of the parameters you can set in an Endpoint or Connection definition. It also covers behavioural attributes such as `isSource` or `maxConnections` on Endpoints.

An Endpoint or Connection can have zero or more Types assigned; they are merged as granularly as possible, in the order in which they were assigned. There is a supporting API that works in the same way as the class stuff does in jQuery:

- **hasType**
- **addType**
- **removeType**
- **toggleType**
- **setType**
- **clearTypes**

and each of these methods (except `hasType`) takes a space-separated string so you can add several at once. Support for these methods has been added to the `jsPlumb.select` and `jsPlumb.selectEndpoint` methods, and you can also now specify a `type` parameter to an Endpoint or Connection at create time.

Types are a useful tool when you are building a UI that has connections whose appearance change under certain circumstances, or a UI that has various types of connections etc.

Connection Type

Probably the easiest way to explain Types is with some code. In this snippet, we'll register a Connection Type on jsPlumb, create a Connection, and then assign the Type to it:

```
jsPlumb.registerConnectionType("example", {
  paintStyle:{ stroke:"blue", strokeWidth:5  },
  hoverPaintStyle:{ stroke:"red", strokeWidth:7 }
});

var c = jsPlumb.connect({ source:"someDiv", target:"someOtherDiv" });
c.bind("click", function() {
  c.setType("example");
});
```

Another example - a better one, in fact. Say you have a UI in which you can click to select or deselect Connections, and you want a different appearance for each state. Connection Types to the rescue!

```
jsPlumb.registerConnectionTypes({
  "basic": {
    paintStyle:{ stroke:"blue", strokeWidth:5  },
    hoverPaintStyle:{ stroke:"red", strokeWidth:7 },
    cssClass:"connector-normal"
  },
  "selected":{
    paintStyle:{ stroke:"red", strokeWidth:5 },
    hoverPaintStyle:{ strokeWidth: 7 },
    cssClass:"connector-selected"
  }
});

var c = jsPlumb.connect({ source:"someDiv", target:"someOtherDiv", type:"basic" });

c.bind("click", function() {
  c.toggleType("selected");
});
```

Notice here how we used a different method - `registerConnectionTypes` - to register a few Types at once.

Notice also the `hoverPaintStyle` for the `selected` Type: it declares only a `strokeWidth`. As mentioned above, Types are merged with as much granularity as possible, so that means that in this case the `strokeWidth` value from `selected` will be merged into the `hoverPaintStyle` from `basic`, and voila, red, 7 pixels.

These examples, of course, use the `jsPlumb.connect` method, but in many UIs Connections are created via drag and drop.
How would you assign that `basic` Type to a Connection created with drag and drop? You provide it as the Endpoint's `connectionType` parameter, like so:

```
var e1 = jsPlumb.addEndpoint("someDiv", {
  connectionType:"basic",
  isSource:true
});

var e2 = jsPlumb.addEndpoint("someOtherDiv", {
  isTarget:true
});

//... user then perhaps drags a connection...or we do it programmatically:

var c = jsPlumb.connect({ source:e1, target:e2 });

// now c has type 'basic'
console.log(c.hasType("basic));   // -> true
```

Note that the second Endpoint we created did not have a `connectionType` parameter - we didn't need it, as the source Endpoint in the Connection had one. But we could have supplied one, and jsPlumb will use it, but only if the source Endpoint has not declared `connectionType`. This is the same way jsPlumb treats other Connector parameters such as `paintStyle` etc - the source Endpoint wins.

Supported Parameters in Connection Type objects

Not every parameter from a Connection's constructor is supported in a Connection Type - as mentioned above, Types act pretty much like CSS classes, so the things that are supported are related to behaviour or appearance (including the ability to set CSS classes on the UI artefacts). For instance, `source` is not supported: it indicates the source element for some particular Connection and therefore does not make sense inside a type specification: you cannot make a Connection Type that is fixed to a specific source element. Here's the full list of supported properties in Connection Type objects:

- **anchor** Anchor specification to use for both ends of the Connection.
- **anchors** Anchor specifications to use for each end of the Connection.
- **detachable** - whether or not the Connection is detachable using the mouse
- **paintStyle**
- **hoverPaintStyle**
- **scope** - remember, Connections support a single scope. So if you have multiple Types applied, you will get the scope from the last Type that defines one.
- **cssClass** a class to set on the element used to render the Connection's connector. Unlike with `scope`, when multiple types assign a CSS class, the UI artefact gets all of them written to it.
- **parameters** - when you add/set a Type that has parameters, any existing parameters with the same keys will be overwritten. When you remove a Type that has parameters, its parameters are NOT removed from the Connection.
- **overlays** - when you have multiple types applied to a Connection, you get the union of all the Overlays defined across the various Types. **Note** when you create a Connection using jsPlumb.connect and you provide a 'type', that is equivalent to calling 'addType': you will get the Overlays defined by the Type(s) you set as well as any others you have provided to the constructor.
- **endpoint** Only works with a type applied to a new Connection. But very useful for that particular use case.

Parameterized Connection Types

Connection Types support parameterized values - values that are derived at runtime by some object you supply. Here's the first example from above, with a parameterized value for `stroke`:

```
jsPlumb.registerConnectionType("example", {
  paintStyle:{ stroke:"${color}", strokeWidth:5  },
  hoverPaintStyle:{ stroke:"red", strokeWidth:7 }
});

var c = jsPlumb.connect({ source:"someDiv", target:"someOtherDiv" });
  c.bind("click", function() {
    c.setType("example", { color:"blue" });
});
```

`setType`, `addType` and `toggleType` all now support this optional second argument.

You can also use a parameterized Type in a `jsPlumb.connect` call, by supplying a `data` value:

```
jsPlumb.registerConnectionType("example", {
  paintStyle:{ stroke:"${color}", strokeWidth:5  },
  hoverPaintStyle:{ stroke:"red", strokeWidth:7 }
});

var c = jsPlumb.connect({
  source:"someDiv",
  target:"someOtherDiv",
  type:"example",
  data:{ color: "blue" }
});
```

Here are a few examples showing you the full Type API:

```
jsPlumb.registerConnectionTypes({
  "foo":{ paintStyle:{ stroke:"yellow", strokeWidth:5, cssClass:"foo" } },
  "bar":{ paintStyle:{ stroke:"blue", strokeWidth:10 } },
  "baz":{ paintStyle:{ stroke:"green", strokeWidth:1, cssClass:"${clazz}" } },
  "boz":{ paintStyle: { stroke:"${color}", strokeWidth:"${width}" } }
});

var c = jsPlumb.connect({
  source:"someDiv",
  target:"someOtherDiv",
  type:"foo"
});

// see what types the connection has.
console.log(c.hasType("foo"));  // -> true
console.log(c.hasType("bar"));  // -> false

// add type 'bar'
c.addType("bar");

// toggle both types (they will be removed in this case)
c.toggleType("foo bar");

// toggle them back
c.toggleType("foo bar");

// getType returns a list of current types.
console.log(c.getType()); // -> [ "foo", "bar" ]

// set type to be 'baz' only
c.setType("baz");

// add foo and bar back in
c.addType("foo bar");

// remove baz and bar
c.removeType("baz bar");

// what are we left with? good old foo.
console.log(c.getType()); // -> [ "foo" ]

// now let's add 'boz', a parameterized type
c.addType("boz", {
  color:"#456",
  width:35
});

console.log(c.getType()); // -> [ "foo", "boz" ]

// now clear all types
c.clearTypes();

console.log(c.getType()); // -> [  ]
```

Things to note here are that every method **except hasType** can take a space-delimited list of Types to work with. So types work like CSS classes, basically. I think I might have mentioned that already though.

Endpoint Type

Endpoints can also be assigned one or more Types, both at creation and programmatically using the API discussed above.

The only real differences between Endpoint and Connection Types are the allowed parameters. Here's the list for Endpoints:

- **paintStyle**
- **endpointStyle** - If this and `paintStyle` are provided, this takes precedence
- **hoverPaintStyle**
- **endpointHoverStyle** - If this and `hoverPaintStyle` are provided, this takes precedence
- **maxConnections**
- **connectorStyle** - paint style for any Connections that use this Endpoint.
- **connectorHoverStyle** - hover paint style for Connections from this Endpoint.
- **connector** - a Connector definition, like `StateMachine`, or `[ "Flowchart", { stub:50 } ]`
- **connectionType** - This allows you to specify the Connection Type for Connections made from this Endpoint.
- **scope** - remember, Endpoints support a single scope. So if you have multiple Types applied, you will get the scope from the last Type that defines one.
- **cssClass** - This works the same as CSS class for Connections: any class assigned by any active type will be written to the UI artefact.
- **parameters** - when you add/set a Type that has parameters, any existing parameters with the same keys will be overwritten. When you remove a Type that has parameters, its parameters are NOT removed from the Connection.
- **overlays** - when you have multiple Types applied to an Endpoint, you get the union of all the Overlays defined across the various types.

Note There are two sets of parameters you can use to set paint styles for Endpoints - `endpointStyle/endpointHoverStyle` and `paintStyle/hoverPaintStyle`. The idea behind this is that when you use the `endpoint..` versions, you can use a single object to define a Type that is shared between Endpoints and Connectors.

One thing to be aware of is that the parameters here that are passed to Connections are only passed from a source Endpoint, not targets. Here's an example of using Endpoint Types:

```
jsPlumb.registerEndpointTypes({
  "basic":{
    paintStyle:{fill:"blue"}
  },
  "selected":{
    paintStyle:{fill:"red"}
  }
});

var e = jsPlumb.addEndpoint("someElement", {
  anchor:"TopMiddle",
  type:"basic"
});

e.bind("click", function() {
  e.toggleType("selected");
});
```

So it works the same way as Connection Types. There are several parameters allowed by an Endpoint Type that affect Connections coming from that Endpoint. Note that this does not affect existing Connections. It affects only Connections that are created after you set the new Type(s) on an Endpoint.

Parameterized Endpoint Types

You can use parameterized Types for Endpoints just as you can Connections:

```
jsPlumb.registerEndpointType("example", {
  paintStyle:{ fill:"${color}"}
});

var e = jsPlumb.addEndpoint("someDiv", {
  type:"example",
  data:{ color: "blue" }
});
```

Reapplying Types

If you have one or more parameterized Types set on some object and you wish for them to change to reflect a change in their underlying data, you can use the `reapplyTypes` function:

```
jsPlumb.registerConnectionType("boz",{
  paintStyle: {
    stroke:"${color}",
    strokeWidth:"${width}"
  }
});

var c = jsPlumb.connect({ source:"s", target:"t" });
c.addType("boz",{ color:"green", width:23 });

    .. things happen ..

c.reapplyTypes({ color:"red", width:0 });
```

`reapplyTypes` applies the new parameters to the merged result of all Types currently set on an object.

Fluid Interface

As mentioned previously, all of the Type operations are supported by the `select` and `selectEndpoints` methods.

So you can do things like this:

```
jsPlumb.selectEndpoints({
  scope:"terminal"
}).toggleType("active");

jsPlumb.select({
  source:"someElement"
}).addType("highlighted available");
```

Obviously, in these examples, `available` and `highlighted` would have previously been registered on jsPlumb using the appropriate register methods.

Events

jsPlumb supports binding to several different events on Connections, Endpoints and Overlays, and also on the jsPlumb object itself.

jsPlumb Events

To bind an to event on jsPlumb itself (or a jsPlumb instance), use `jsPlumb.bind(event, callback)`:

```
jsPlumb.bind("connection", function(info) {
   .. update your model in here, maybe.
});
```

These are the events you can bind to on the jsPlumb class:

connection(info, originalEvent)

Notification a Connection was established.

`info` is an object with the following properties:

- **connection** - the new Connection. you can register listeners on this etc.
- **sourceId** - id of the source element in the Connection
- **targetId** - id of the target element in the Connection
- **source** - the source element in the Connection
- **target** - the target element in the Connection
- **sourceEndpoint** - the source Endpoint in the Connection
- **targetEndpoint** - the targetEndpoint in the Connection

Note: `jsPlumb.connect` causes this event to be fired, but there is of course no original event when a connection is established programmatically. So you can test to see if `originalEvent` is undefined to determine whether a connection was estblished using the mouse or not.

All of the source/target properties are actually available inside the Connection object, but - for one of those rubbish historical reasons - are provided separately because of a vagary of the `connectionDetached` callback, which is discussed below.

connectionDetached(info, originalEvent)

Notification a Connection was detached.

As with `connection`, the first argument to the callback is an object with the following properties:

- **connection** - the Connection that was detached.
- **sourceId** - id of the source element in the Connection *before* it was detached
- **targetId** - id of the target element in the Connection before it was detached
- **source** - the source element in the Connection before it was detached
- **target** - the target element in the Connection before it was detached
- **sourceEndpoint** - the source Endpoint in the Connection before it was detached
- **targetEndpoint** - the targetEndpoint in the Connection before it was detached

This event is not fired when a newly dragged Connection is abandoned before being connected to something. To catch that, use `connectionAborted`.

The `source/target` properties are provided separately from the Connection, because this event is fired whenever a Connection is either detached and abandoned, or detached from some Endpoint and attached to another. In the latter case, the Connection that is passed to this callback is in an indeterminate state (that is, the Endpoints are still in the state they are in when dragging, and do not reflect static reality), and so the `source/target` properties give you the real story.

The second argument is the original mouse event that caused the disconnection, if any.

connectionMoved(info, originalEvent)

Notification that an existing connection's source or target endpoint was dragged to some new location. `info` contains the following properties:

- **index** - 0 for source endpoint, 1 for target endpoint
- **originalSourceId** - id of connection source element before move
- **newSourceId** - id of connection source element after move
- **originalTargetId** id of connection target before move
- **newTargetId** - id of connection target after move
- **originalSourceEndpoint** - source endpoint before move
- **newSourceEndpoint** - source endpoint after move
- **originalTargetEndpoint** - target endpoint before move
- **newTargetEndpoint** - target endpoint after move

connectionAborted(connection, originalEvent)

Fired when a new Connection is dragged but abandoned before being connected to an Endpoint or a target element.

connectionDrag(connection)

Notification an existing Connection is being dragged. Note that when this event fires for a brand new Connection, the target of the Connection is a transient element that jsPlumb is using for dragging, and will be removed from the DOM when the Connection is subsequently either established or aborted.

connectionDragStop(connection)

Notification a Connection drag has stopped. This is only fired for existing Connections.

click(connection, originalEvent)

Notification a Connection was clicked.

dblclick(connection, originalEvent)

Notification a Connection was double-clicked.

endpointClick(endpoint, originalEvent)

Notification an Endpoint was clicked.

endpointDblClick(endpoint, originalEvent)

Notification an Endpoint was double-clicked.

contextmenu(component, originalEvent)

A right-click on some given component. jsPlumb will report right clicks on both Connections and Endpoints.

beforeDrop(info)

This event is fired when a new or existing connection has been dropped. `info` contains the following properties:

- **sourceId** - the id of the source element in the connection
- **targetId** - the id of the target element in the connection
- **scope** - the scope of the connection
- **connection** - the actual Connection object. You can access the 'endpoints' array in a Connection to get the Endpoints involved in the Connection, but be aware that when a Connection is being dragged, one of these Endpoints will always be a transient Endpoint that exists only for the life of the drag. To get the Endpoint on which the Connection is being dropped, use the `dropEndpoint` member.
- **dropEndpoint** - this is the actual Endpoint on which the Connection is being dropped. This **may be null**, because it will not be set if the Connection is being dropped on an element on which makeTarget has been called.

If you return false (or nothing) from this callback, the new Connection is aborted and removed from the UI.

beforeDetach(connection)

This event is fired when a Connection is about to be detached, for whatever reason. Your callback function is passed the Connection that the user has just detached. Returning false from this interceptor aborts the Connection detach.

zoom(value)

Notification the current zoom was changed.

Connection Events

To bind to an event on a Connection, you also use the `bind` method:

```
var connection = jsPlumb.connect({source:"d1", target:"d2"});
connection.bind("click", function(conn) {
    console.log("you clicked on ", conn);
});
```

These are the Connection events to which you can bind a listener:

- `click(connection, originalEvent)` - notification a Connection was clicked.
- `dblclick(connection, originalEvent)` - notification a Connection was double-clicked.
- `contextmenu(connection, originalEvent)` - a right-click on the Connection.
- `mouseover(connection, originalEvent)` - notification the mouse is over the Connection's path.
- `mouseout(connection, originalEvent)` - notification the mouse has exited the Connection's path.
- `mousedown(connection, originalEvent)` - notification the mouse button was pressed on the Connection's path.
- `mouseup(connection, originalEvent)` - notification the mouse button was released on the Connection's path.

Endpoint Events

To bind to an event on a Endpoint, you again use the `bind` method:

```
var endpoint = jsPlumb.addEndpoint("d1", { someOptions } );
endpoint.bind("click", function(endpoint) {
    console.log("you clicked on ", endpoint);
});
```

These are the Endpoint events to which you can bind a listener:

- `click(endpoint, originalEvent)` - notification an Endpoint was clicked.

- `dblclick(endpoint, originalEvent)` - notification an Endpoint was double-clicked.
- `contextmenu(endpoint, originalEvent)` - a right-click on the Endpoint.
- `mouseover(endpoint, originalEvent)` - notification the mouse is over the Endpoint.
- `mouseout(endpoint, originalEvent)` - notification the mouse has exited the Endpoint.
- `mousedown(endpoint, originalEvent)` - notification the mouse button was pressed on the Endpoint.
- `mouseup(endpoint, originalEvent)` - notification the mouse button was released on the Endpoint.
- `maxConnections(info, originalEvent)` - notification the user tried to drop a Connection on an Endpoint that already has the maximum number of Connections. `info` is an object literal containing these values:
    - **endpoint** : Endpoint on which the Connection was dropped
    - **connection** : The Connection the user tried to drop
    - **maxConnections** : The value of `maxConnections` for the Endpoint

Overlay Events

Registering event listeners on an Overlay is a slightly different procedure - you provide them as arguments to the Overlay's constructor. This is because you never actually act on an Overlay object.

Here's how to register a click listener on an Overlay:

```
jsPlumb.connect({
    source:"el1",
    target:"el2",
    overlays:[
      [ "Label", {
        events:{
          click:function(labelOverlay, originalEvent) {
            console.log("click on label overlay for :" + labelOverlay.component);
          }
        }
      }],
      [ "Diamond", {
        events:{
          dblclick:function(diamondOverlay, originalEvent) {
            console.log("double click on diamond overlay for : " + diamondOverlay.component);
          }
        }
      }]
    ]
  });
```

The related component for an Overlay is available to you as the `component` member of the Overlay.

Unbinding Events

On the jsPlumb object and on Connections and Endpoints, you can use the `unbind` method to remove a listener. This method either takes the name of the event to unbind:

```
jsPlumb.unbind("click");
```

...or no argument, meaning unbind all events:

```
var e = jsPlumb.addEndpoint("someDiv");
e.bind("click", function() { ... });
e.bind("dblclick", function() { ... });

...

e.unbind("click");
```

Interceptors

Interceptors are basically event handlers from which you can return a value that tells jsPlumb to abort what it is that it was doing. There are four interceptors supported - `beforeDrop`, which is called when the user has dropped a Connection onto some target, `beforeDetach`, which is called when the user is attempting to detach a Connection (by dragging it off one of its Endpoints and dropping in whitespace), `beforeDrag`, which is called when the user begins to drag a Connection, and `beforeStartDetach`, which is when the user has just begun to drag an existing Connection off of one of its Endpoints (compared with `beforeDetach`, in which the user is allowed to drag the Connection off).

Interceptors can be registered via the `bind` method on an instance of jsPlumb just like any other event listeners, and they can also be passed in to the `addEndpoint`, `makeSource` and `makeTarget` methods.

Note that binding `beforeDrop` (as an example) on a jsPlumb instance itself is like a catch-all: it will be called every time a Connection is dropped on *any* Endpoint, unless that Endpoint has its own `beforeDrop` interceptor. But passing a beforeDrop callback into an Endpoint constrains that callback to just the Endpoint in question.

beforeDrop

This event is fired when a new or existing connection has been dropped. Your callback is passed a JS object with these fields:

- **sourceId** - the id of the source element in the connection
- **targetId** - the id of the target element in the connection
- **scope** - the scope of the connection
- **connection** - the actual Connection object. You can access the 'endpoints' array in a Connection to get the Endpoints involved in the Connection, but be aware that when a Connection is being dragged, one of these Endpoints will always be a transient Endpoint that exists only for the life of the drag. To get the Endpoint on which the Connection is being dropped, use the 'dropEndpoint' member.
- **dropEndpoint** - this is the actual Endpoint on which the Connection is being dropped. This **may be null**, because it will not be set if the Connection is being dropped on an element on which makeTarget has been called.

If you return false (or nothing) from this callback, the new Connection is aborted and removed from the UI.

beforeDetach

This is called when the user has detached a Connection, which can happen for a number of reasons: by default, jsPlumb allows users to drag Connections off of target Endpoints, but this can also result from a programmatic 'detach' call. Every case is treated the same by jsPlumb, so in fact it is possible for you to write code that attempts to detach a Connection but then denies itself! You might want to be careful with that.

Note that this interceptor is passed the actual Connection object; this is different from the beforeDrop interceptor discussed above: in this case, we've already got a Connection, but with beforeDrop we are yet to confirm that a Connection should be created.

Returning false - or nothing - from this callback will cause the detach to be abandoned, and the Connection will be reinstated or left on its current target.

beforeDrag

This is called when the user starts to drag a new Connection. These parameters are passed to the function you provide:

- **endpoint** the Endpoint from which the user is dragging a Connection
- **source** the DOM element the Endpoint belongs to
- **sourceId** the ID of the DOM element the Endpoint belongs to

`beforeDrag` operates slightly differently to the other interceptors: it is still the case that returning false from your interceptor function will abort the current activity - in this case cancelling the drag - but you can also return an object from your interceptor, and this object will be passed in as the `data` parameter in the constructor of the new Connection:

```
jsPlumbInstance.bind("beforeDrag", function(params) {
  return {
   foo:"bar"
  }
});
```

This is particularly useful if you have defined any [parameterized connection types](). With this mechanism you can arrange for a newly dragged Connection to be populated with data of your choice.

Note all versions of jsPlumb prior to 1.7.6 would fire `beforeDetach` for both new Connection drags and also dragging of existing Connections. From 1.7.6 onwards this latter behaviour has been moved to the `beforeStartDetach` interceptor.

beforeStartDetach

This is called when the user starts to drag an existing Connection. These parameters are passed to the function you provide:

- **endpoint** the Endpoint from which the user is dragging a Connection
- **source** the DOM element the Endpoint belongs to
- **sourceId** the ID of the DOM element the Endpoint belongs to
- **connection** The Connection that is about to be dragged.

Returning false from `beforeStartDetach` prevents the Connection from being dragged.

Paint Styles

Defining the appearance of Connectors and Endpoints is achieved through a `paintStyle` (or a quite similar name) object passed as a parameter to one of `jsPlumb.connect`, `jsPlumb.addEndpoint`, `jsPlumb.makeSource` or `jsPlumb.makeTarget`. Depending on the method you are calling, the parameter names vary.

Connector Paint Styles

These are specified in a `paintStyle` parameter on a call to `jsPlumb.connect`:

```
jsPlumb.connect({
    source:"el1",
    target:"el2",
    paintStyle:{ stroke:"blue", strokeWidth:10 }
});
```

or in the `connectorPaintStyle` parameter on a call to `jsPlumb.addEndpoint` or `jsPlumb.makeSource`:

```
jsPlumb.addEndpoint("el1", {
    paintStyle:{ fill:"blue", outlineStroke:"black", outlineWidth:1 },
    connectorPaintStyle:{ stroke:"blue", strokeWidth:10 }
});
```

```
jsPlumb.makeSource("el1", {
    paintStyle:{ fill:"blue", outlineStroke:"black", outlineWidth:1 },
    connectorPaintStyle:{ stroke:"blue", strokeWidth:10 }
});
```

Notice the `paintStyle` parameter in those examples: it is the paint style for the Endpoint, which we'll discuss below.

Endpoint Paint Styles

These are specified in a `paintStyle` parameter on a call to `addEndpoint`. This is the example from just above:

```
jsPlumb.addEndpoint("el1", {
    paintStyle:{ fill:"blue", outlineStroke:"black", outlineWidth:1 },
    connectorPaintStyle:{ stroke:"blue", strokeWidth:10 }
});
```

...or as the `endpointStyle` parameter to a `connect` call:

```
jsPlumb.connect({
    source:"el1",
    target:"el2",
    endpointStyle:{ fill:"blue", outlineStroke:"black", outlineWidth:1 },
    paintStyle:{ stroke:"blue", strokeWidth:10 }
});
```

... or as an entry in the `endpointStyles` array passed to a `jsPlumb.connect` call:

```
jsPlumb.connect({
    source:"el1",
    target:"el2",
    endpointStyles:[
        { fill:"blue", outlineStroke:"black", outlineWidth:1 },
        { fill:"green" }
    ],
    paintStyle:{ stroke:"blue", strokeWidth:10 }
});
```

or as the `paintStyle` parameter passed to a `makeTarget` or `makeSource` call:

```
jsPlumb.makeTarget("el1", {
    ...
    paintStyle:{ fill:"blue", outlineStroke:"black", outlineWidth:1 },
    ...
});
```

```
jsPlumb.makeSource("el1", {
    paintStyle:{ fill:"blue", outlineStroke:"black", outlineWidth:1 }
    parent:"someOtherDivIJustPutThisHereToRemindYouYouCanDoThis"
});
```

In the first example we made `el1` into a drop target, and defined a paint style for the Endpoint jsPlumb will create when a Connection is established. In the second we made el1 a connection source and assigned the same values for the Endpoint jsPlumb will create when a Connection is dragged from that element.

Overlay Paint Styles

The preferred way to set paint styles for Overlays is to use the `cssClass` parameter in the constructor arguments of an Overlay definition.

Paint Style Parameters

This is the full list of parameters you can set in a paintStyle object, but note that `fill` is ignored by Connectors, and `stroke` is ignored by Endpoints. Also, if you create a Connection using jsPlumb.connect and do not specify any Endpoint styles, the Endpoints will derive their fill from the Connector's stroke.

`fill`, `stroke` and `outlineStroke` can be specified using any valid CSS3 syntax.

- **fill** - color for an Endpoint, eg. rgba(100,100,100,50), "blue", "#456", "#993355", rgb(34, 56, 78).
- **stroke** - color for a Connector. see fill examples.
- **strokeWidth** - width of a Connector's line. An integer.
- **outlineWidth** - width of the outline for an Endpoint or Connector. An integer.
- **outlineStroke** - color of the outline for an Endpoint or Connector. see fill examples.
- **dashstyle** - This comes from VML, and allows you to create dashed or dotted lines. It has a better syntax than the equivalent attribute in SVG (`stroke-dasharray`, discussed below), so even though VML is no longer a supported renderer we've decided to keep this attribute. The `dashstyle` attribute is specified as an array of strokes and spaces, where each value is some multiple of **the width of the Connector** , and that's where it's better than SVG, which just uses absolute pixel values.

The VML spec is a good place to find valid values for dashstyle. Note that jsPlumb does not support the string values for this attribute ("solid", "dashdot", etc).

jsPlumb uses the `strokeWidth` parameter in conjunction with the values in a `dashstyle` attribute to create an appropriate value for `stroke-dasharray`.

- **stroke-dasharray** - This is the SVG equivalent of `dashstyle`. The SVG spec discusses valid values for this parameter.
- **stroke-dashoffset** - This is used in SVG to specify how far into the dash pattern to start painting. For more information, see the SVG spec
- **stroke-linejoin** - This attribute specifies how you want individual segments of connectors to be joined.

Hover Paint Styles

Connectors and Endpoints both support the concept of a "hover" paint style - a paint style to use when the mouse is hovering over the component. These are specified in the exact same format as paint styles discussed above, but hover paint styles also inherit any values from the main paint style. This is because you will typically want to just change the color, or perhaps outline color, of a Connector or Endpoint when the mouse is hovering, but leave everything else the same. So having hover paint styles inherit their values precludes you from having to define things in more than one place.

The naming convention adopted for hover paint styles is pretty much to insert the word 'hover' into the corresponding main paint style parameters. Here are a couple of examples:

```
jsPlumb.connect({
    source:"el1",
    target:"el2",
    paintStyle:{ stroke:"blue", strokeWidth:10 },
    hoverPaintStyle:{ stroke:"red" },
    endpointStyle:{ fill:"blue", outlineStroke:"black", outlineWidth:1 },
    endpointHoverStyle:{ fill:"red" }
});
```

In this example we specified a hover style for both the Connector, and each of its Endpoints. Here's the same thing, but using the plural version, to specify a different hover style for each Endpoint:

```
jsPlumb.connect({
    source:"el1",
    target:"el2",
    paintStyle:{ stroke:"blue", strokeWidth:10 },
    hoverPaintStyle:{ stroke:"red" },
    endpointStyle:{ fill:"blue", outlineStroke:"black", outlineWidth:1 },
    endpointHoverStyles:[
        { fill:"red" },
        { fill:"yellow" }
    ]
});
```

Calls to `addEndpoint`, `makeSource` and `makeTarget` can also specify various hover paint styles:

```
jsPlumb.addEndpoint("el1", {
    paintStyle:{ fill:"blue", outlineStroke:"black", outlineWidth:1 },
    hoverPaintStyle:{ fill:"red" },
    connectorPaintStyle:{ stroke:"blue", strokeWidth:10 },
    connectorHoverPaintStyle:{ stroke:"red", outlineStroke:"yellow", outlineWidth:1 }
});

jsPlumb.makeSource("el2", {
    paintStyle:{
        fill:"transparent",
        outlineStroke:"yellow",
        outlineWidth:1
    },
    hoverPaintStyle:{ fill:"red" },
    connectorPaintStyle:{
        stroke:"green",
        strokeWidth:3
    },
    connectorHoverPaintStyle:{
        stroke:"#678",
        outlineStroke:"yellow",
        outlineWidth:1
    }
});

jsPlumb.makeTarget("el3", {
    paintStyle:{
        fill:"transparent",
        outlineStroke:"yellow",
        outlineWidth:1
    },
    hoverPaintStyle:{ fill:"red" }
});
```

In these examples we specified a hover paint style for both the Endpoint we are adding, and any Connections to/from the Endpoint.

Note that `makeTarget` does not support Connector parameters. It is for creating targets only; Connector parameters will be set by the source Endpoint in any Connections that are made to the element that you turned into a target by using this method.

Gradients

jsPlumb uses its own syntax to define gradients; this was initially to abstract out the differences between the syntax required by canvas and that required by SVG, but in fact since jsPlumb does not support the canvas or VML renderers any more, it is possible that a future release will switch to using the SVG syntax for gradients.

There are two types of gradients available - a `linear` gradient, which consists of colored lines all going in one direction, and a `radial` gradient, which consists of colored circles emanating from one circle to another. Because of their basic shape, jsPlumb supports only linear gradients for Connectors. But for Endpoints, jsPlumb supports both linear and radial gradients.

Connector gradients

To specify a linear gradient to use in a Connector, you must add a `gradient` object to your Connector's `paintStyle`, for instance:

```
jsPlumb.connect({
  source : "window2",
  target : "window3",
  paintStyle:{
    gradient:{
      stops:[[0,"green"], [1,"red"]]
    },
    strokeWidth:15
  }
});
```

Here we have connected window2 to window3 with a 15 pixel wide connector that has a gradient from green to red.

Notice the `gradient` object and the `stops` list inside it - the gradient consists of an arbitrary number of these "color stops". Each color stop is comprised of two values - [position, color]. Position must be a decimal value between 0 and 1 (inclusive), and indicates where the color stop is situated as a fraction of the length of the entire gradient. Valid values for the colors in the stops list are the same as those that are valid for stroke when describing a color.

As mentioned, the stops list can hold an arbitrary number of entries. Here's an example of a gradient that goes from red to blue to green, and back again through blue to red:

```
jsPlumb.connect({
  source : 'window2',
  target : 'window3',
  paintStyle : {
    gradient:{
      stops:[[0,'red'], [0.33,'blue'], [0.66,'green'], [0.33,'blue'], [1,'red']]
    },
    strokeWidth : 15
  }
});
```

Endpoint gradients

Endpoint gradients are specified using the same syntax as Connector gradients. You put the gradient specifier either in the `endpoint` member, or if you are specifying different Endpoints for each end of the Connector, in one or both of the values in the `endpoints` array. Also, this information applies to the case that you are creating standalone Endpoints that you will be configuring for drag and drop creation of new Connections.

This is an example of an Endpoint gradient that is different for each Endpoint in the Connector. This comes from the main demo; it is the Connector joining Window 2 to Window 3:

```
var w23Stroke = 'rgb(189,11,11)';
jsPlumb.connect({
  source : 'window2',
  target : 'window3',
  paintStyle:{
    strokeWidth:8,
    stroke:w23Stroke
  },
  anchors:[ [0.3,1,0,1], "TopCenter" ],
  endpoint:"Rectangle",
  endpointStyles:[{
    gradient : {
      stops:[[0, w23Stroke], [1, '#558822']]
    }
  },{
    gradient : {
      stops:[[0, w23Stroke], [1, '#882255']]
    }
  }]
});
```

The first entry in the gradient will be the one that is on the Connector end of the Endpoint. You can of course have as many color stops as you want in this gradient, just like with Connector gradients.

Applying the gradient in Endpoints

Only the Dot and Rectangle endpoints honour the presence of a gradient. The Image endpoint of course ignores a gradient as it does no painting of its own.

The type of gradient you will see depends on the Endpoint type:

- **Dot** - renders a radial endpoint, with color stop 0 on the outside, progressing inwards as we move through color stops.

Radial gradients actually require more data than linear gradients - in a linear gradient we just move from one point to another, whereas in a radial gradient we move from one *circle* to another. By default, jsPlumb will render a radial gradient using a source circle of the same radius as the Endpoint itself, and a target circle of 1/3 of the radius of the Endpoint (both circles share the same center as the Endpoint itself). This circle will be offset by radius/2 in each direction.

You can supply your own values for these inside the gradient descriptor:

```
var w34Stroke = 'rgba(50, 50, 200, 1)';
var w34HlStroke = 'rgba(180, 180, 200, 1)';
jsPlumb.connect({
  source : 'window3',
  target : 'window4',
  paintStyle:{
    strokeWidth:10,
    stroke:w34Stroke
  },
  anchors:[ "RightMiddle", "LeftMiddle" ],
  endpointStyle:{
    gradient : {
      stops:[ [0, w34Stroke], [1, w34HlStroke] ],
      offset:37.5,
      innerRadius:40
    },
    radius:55
  }
});
```

Here we have instructed jsPlumb to make the gradient's inner radius 10px instead of the default 25/3 = 8 ish pixels, and the offset in each direction will be 5px, instead of the default radius / 2 = 12.5 pixels.

It is also possible to specify the offset and inner radius as percentages - enter the values as strings with a '%' symbol on the end:

```
var w34Stroke = 'rgba(50, 50, 200, 1)';
var w34HlStroke = 'rgba(180, 180, 200, 1)';
jsPlumb.connect({
  source : 'window3',
  target : 'window4',
  paintStyle:{
    strokeWidth:10,
    stroke:w34Stroke
  },
  anchors:[ "RightMiddle", "LeftMiddle" ],
  endpointStyle:{
    gradient : {
      stops:[ [0, w34Stroke], [1, w34HlStroke] ],
      offset:'68%',
      innerRadius:'73%'
    },
    radius:25
  }
});
```

This will give roughly the same output as the example above (the percentages are not entirely exact).

- **Rectangle** - renders a linear endpoint, with color stop 0 closest to the end of the Connector

Styling via CSS

Using CSS to style the artefacts that jsPlumb creates is a lot more flexible than using `paintStyle` or `hoverPaintStyle`.

On this page we'll first run through the default CSS classes that jsPlumb attaches, followed by a quick explanation for how to attach your own, and then we'll discuss how to style SVG.

Z Index

One thing you can - and should - use CSS for, regardless of the renderer, is z-index. Every Connection, Endpoint and Overlay in jsPlumb adds some element to the UI, and you should take care to establish appropriate z-indices for each of these, in conjunction with the nodes in your application.

Basic CSS classes

By default, jsPlumb adds a specific class to each of the three types of elements it creates (These class names are exposed on the jsPlumb object and can be overridden if you need to do so - see the third column in the table)

Component	CSS Class	jsPlumb Member
Connector	jsplumb-connector	connectorClass
Connector Outline	jsplumb-connector-outline	connectorOutlineClass (SVG only)
Endpoint	jsplumb-endpoint	endpointClass
Endpoint when full	jsplumb-endpoint-full	endpointFullClass
Overlay	jsplumb-overlay	overlayClass

In a simple UI, you can set appropriate z-index values for these classes. The jsPlumb demo pages, for instance, typically use a class of `.window` for the nodes in each demo page, and the z-index of the UI is controlled with CSS rules like this:

```
.window { z-index:20; }
.jsplumb-connector { z-index:4; }
.jsplumb-endpoint { z-index:5; }
.jsplumb-overlay { z-index:6; }
```

Interactive CSS Classes

jsPlumb assigns these classes on both Connectors and Endpoints when specific user interactivity occurs:

Activity	CSS Class	jsPlumb Member	Description
Mouse Hover	jsplumb-hover	hoverClass	Assigned to both Connectors and Endpoints when the mouse is hovering over them
Connection Drag	jsplumb-dragging	draggingClass	Assigned to a Connection when it is being dragged (either a new Connection or an existing Connection)
Element Dragging	jsplumb-element-dragging	elementDraggingClass	Assigned to all Connections whose source or target element is currently being dragged, and to their Endpoints.
Source Element Dragging	jsplumb-source-element-dragging	sourceElementDraggingClass	Assigned to all Connections whose source element is being dragged, and to their Endpoints
Target Element Dragging	jsplumb-target-element-dragging	targetElementDraggingClass	Assigned to all Connections whose target element is being dragged, and to their Endpoints
Anchor Class	***jsplumb-endpoint-anchor-***	endpointAnchorClassPrefix	Assigned to Endpoints, and their associated elements, that have either a static Anchor with an associated class, or a Dynamic Anchor whose individual locations have an associated CSS class. The `***` suffix in the class name above is the associated class. Note that this class is added to both the artefact that jsPlumb creates and also the element on which the Endpoint resides, so you will normally have to build a selector with more criteria than just this class in order to target things properly. See the documentation regarding Anchors for a discussion of this.
Drop Allowed on Endpoint	jsplumb-endpoint-drop-allowed	endpointDropAllowedClass	Assigned to an Endpoint when another Endpoint is hovering over it and a drop would be allowed
Drop Forbidden on Endpoint	jsplumb-endpoint-drop-forbidden	endpointDropForbiddenClass	Assigned to an Endpoint when another Endpoint is hovering over it and a drop would not be allowed

Connection Hover	jsplumb-source-hover	hoverSourceClass	Assigned to the source element in a Connection when the mouse is hovering over the Connection
Connection Hover	jsplumb-target-hover	hoverTargetClass	Assigned to the target element in a Connection when the mouse is hovering over the Connection
Drag	jsplumb-drag-select	dragSelectClass	Assigned to the document body whenever a drag is in progress. It allows you to ensure document selection is disabled - see [here](home#dragSelection)

Note the last two classes work in conjunction with the `checkDropAllowed` interceptor that you can register on jsPlumb. For more information about interceptors, see here, but in a nutshell you just provide a function that takes the two Endpoints and a Connection as argument, and returns whether not a drop would be allowed:

```
jsPlumb.bind("checkDropAllowed", function(params) {

  // Here you have access to:
  // params.sourceEndpoint
  // params.targetEndpoint
  // params.connection

  return true; // or false.  in this case we say drop is allowed.
});
```

Preventing selection while dragging

jsPlumb puts a class on the body that you can use to disable the default browser behaviour of selecting DOM elements when dragging:

```
jsplumb-drag-select
```

A suitable value for this (from the jsPlumb demos) is:

```
.jsplumb-drag-select {
  -webkit-touch-callout: none;
  -webkit-user-select: none;
  -khtml-user-select: none;
  -moz-user-select: none;
  -ms-user-select: none;
  user-select: none;
}
```

Custom CSS Classes

In addition to the default CSS classes, each of the main methods you use to configure Endpoints or make Connections in jsPlumb support the following two parameters:

- **cssClass** - class(es) to set on the display elements
- **hoverClass** - class(es) to set on the display elements when in hover mode

In addition, `addEndpoint` and `makeSource` allow you to specify what these classes will be for any Connections that are dragged from them:

- **connectorClass** - class(es) to set on the display elements of Connections
- **connectorHoverClass** - class(es) to set on the display elements of Connections when in hover mode

These parameters should be supplied as a String; they will be appended as-is to the class member, so feel free to include multiple classes. jsPlumb won't even know.

CSS for SVG elements

Connections

SVG connections in jsPlumb consist of a parent `svg` element, inside of which there are one or more `path` elements, depending on whether or not you have specified an `outlineStyle`. To target the path element for some connection, you need a rule like this:

```
svg path {
  stroke:red;
}
```

Hooking this up to the custom class mechanism, you might do something like this:

```
jsPlumb.connect({
  source:"someElement",
  target:"someOtherElement",
  cssClass:"redLine"
});
```

CSS:

```
svg.redLine path {
  stroke:red;
  stroke-width:3;
}
```

You might be thinking to yourself, why have the `svg` and `path` elements in this? In fact they are perhaps not required: they're just there to call out the fact that this is a style on an SVG connector.

Endpoints

SVG Endpoints created by jsPlumb consist of a `div`, inside of which is an `svg` parent element, inside of which there is some shape, the tag name of which depends on the type of Endpoint:

Endpoint Type	SVG Shape
Rectangle	rect
Dot	circle

So you can choose, when writing CSS rules for these, whether or not you specify the shape exactly, or just leave it up to a wildcard:

```
jsPlumb.addEndpoint("someElement", {
  cssClass:"aRedEndpoint",
  endpoint:"Dot"
});
```

CSS:

```
.aRedEndpoint svg circle {
  fill:red;
  stroke:yellow;
}
```

...or:

```
.aRedEndpoint svg * {
  fill:red;
  stroke:yellow;
}
```

For a full discussion of the properties you can configure on an SVG element via CSS, we refer you to the SVG spec.

Querying jsPlumb

Connections

Selecting and operating on a list of Connections

The `jsPlumb.select` function provides a fluid interface for working with lists of Connections. The syntax used to specify which Connections you want is identical to that which you use for `getConnections`, but the return value is an object that supports most operations that you can perform on a Connection, and which is also chainable, for setter methods. Certain getter methods are also supported, but these are not chainable; they return an array consisting of all the Connections in the selection along with the return value for that Connection.

Setter Operations

This is the full list of setter operations supported by jsPlumb.select:

- addClass
- removeClass
- addOverlay
- removeOverlay
- removeOverlays
- showOverlay
- hideOverlay
- showOverlays
- hideOverlays
- removeAllOverlays
- setLabel
- setPaintStyle
- setHoverPaintStyle
- setDetachable
- setReattach
- setConnector
- setParameter
- setParameters
- detach
- repaint
- setType
- addType
- removeType
- toggleType
- setVisible

Each of these operations returns a selector that can be chained.

Getter Operations

This is the full list of getter operations supported by `jsPlumb.select`:

- getLabel
- getOverlay
- isHover
- isVisible
- isDetachable
- isReattach
- getParameter
- getParameters
- getType
- hasType

Each of these operations returns an array whose entries are [value, Connection] arrays, where `value` is the return value from the given Connection. Remember that the return values from a getter are not chainable, but a getter may be called at the end of a chain of setters.

Examples

- Select all Connections and set their hover state to be false:

```
jsPlumb.select().setHover(false);
```

- Select all Connections from "d1" and remove all Overlays:

```
jsPlumb.select({source:"d1"}).removeAllOverlays();
```

- Select all connections in scope "foo" and set their paint style to be a thick blue line:

```
jsPlumb.select({scope:"foo"}).setPaintStyle({
        stroke:"blue",
        strokeWidth:5
});
```

- Select all Connections from "d1" and detach them:

```
jsPlumb.select({source:"d1"}).detach();
```

- Select all Connections and add the classes "foo" and "bar" to them :

```
jsPlumb.select().addClass("foo bar");
```

- Select all Connections and remove the class "foo" from them :

```
jsPlumb.select().removeClass("foo");
```

Iterating through results

The return value of `jsPlumb.select` has a `.each` function that allows you to iterate through the list, performing an operation on each one:

```
jsPlumb.select({scope:"foo"}).each(function(connection) {
                // do something
});
```

`.each` is itself chainable:

```
jsPlumb.select({scope:"foo"}).each(function(connection) {
                // do something
}).setHover(true);
```

Other properties/functions

- **length** - this member reports the number of Connections in the selection
- **get(idx)** - this function allows you to retrieve a Connection from the selection

Retrieving static Connection lists

As well as the `select` method, jsPlumb has another method that can be used to retrieve static lists of Connections - `getConnections`. The return value of this method is not chainable and does not offer operations on the Connections in the way that the return value of `select` does.

Retrieving connections for a single scope

To do this, you call `getConnections` with either no arguments, in which case jsPlumb uses the default scope, or with a string specifying one scope

```
var connectionList = jsPlumb.getConnections(); // you get a list of Connection objects that are in the defau
lt scope.
```

Compare this with:

```
var connectionList = jsPlumb.getConnections("myScope");     // you get a list of Connection objects that are
in "myScope".
```

Filtering by source, target and/or scope

`getConnections` optionally takes a JS object specifying filter parameters, of which there are three:

- **scope** - the scope(s) of the connection type(s) you wish to retrieve
- **source** - limits the returned connections to those that have this source id
- **target** - limits the returned connections to those that have this target id

Each of these three parameters may be supplied as a string, which for source and target is an element id and for scope is the name of the scope, or a list of strings. Also, you can pass "*" in as the value for any of these - a wildcard, meaning any value. See the examples below.

Important: The return value of a call to `getConnection` using a JS object as parameter varies on how many scopes you defined. If you defined only a single scope then jsPlumb returns you a list of Connections in that scope. Otherwise the return value is a dictionary whose keys are scope names, and whose values are lists of Connections. For example, the following call:

```
jsPlumb.getConnections({
  scope:["someScope", "someCustomScope"]
});
```

would result in this output:

```
{
  "someScope" : [ 1..n Connections ],
  "someCustomScope": [ 1..m Connections ]
}
```

This has tripped up many a developer who has been reluctant to take the time to read the documentation.

There is an optional second parameter that tells getConnections to flatten its output and just return you an array. The previous example with this parameter would look like this:

```
jsPlumb.getConnections({
  scope:["someScope", "someCustomScope"]
}, true);
```

...and would result in this output:

`[ 1..n Connections ]`

The following examples show the various ways you can get connection information:

- Get all connections:

  ```
  jsPlumb.getAllConnections();
  ```

- Get all connections for the default scope only(return value is a list):

  ```
  jsPlumb.getConnections();
  ```

- Get all connections for the given scope (return value is a list):

  ```
  jsPlumb.getConnections({scope:"myTestScope"});</pre>
  ```

- Get all connections for the given scopes (return value is a map of scope names to connection lists):

  ```
  jsPlumb.getConnections({
      scope:["myTestScope", "yourTestScope"]
  });
  ```

- Get all connections for the given source (return value is a map of scope names to connection lists):

  ```
  jsPlumb.getConnections({
      source:"mySourceElement"
  });
  ```

- Get all connections for the given sources (return value is a map of scope names to connection lists):

  ```
  jsPlumb.getConnections({
      source:["mySourceElement", "yourSourceElement"]
  });
  ```

- Get all connections for the given target (return value is a map of scope names to connection lists):

  ```
  jsPlumb.getConnections({
      target:"myTargetElement"
  });
  ```

- Get all connections for the given source and targets (return value is a map of scope names to connection lists):

  ```
  jsPlumb.getConnections({
          source:"mySourceElement",
          target:["target1", "target2"]
  });
  ```

- Get all connections for the given scope, with the given source and target (return value is a list of connections):

```
jsPlumb.getConnections({
        scope:'myScope',
        source:"mySourceElement",
        target:"myTargetElement"
});
```

Selecting and operating on a list of Endpoints

`jsPlumb.selectEndpoints` provides a fluid interface for working with lists of Endpoints.

The syntax used to specify which Endpoints you want is identical to that which you use for `jsPlumb.select`, and the return value is an object that supports most operations that you can perform on an Endpoint (and which is also chainable, for setter methods). Certain getter methods are also supported, but these are not chainable; they return an array consisting of all the Endpoints in the selection along with the return value for that Endpoint.

Four parameters are supported by selectEndpoints - each of these except `scope` can be provided as either a string, a selector, a DOM element, or an array of a mixture of these types. `scope` can be provided as either a string or an array of strings:

- **element** - element(s) to get both source and target endpoints from
- **source** - element(s) to get source endpoints from
- **target** - element(s) to get target endpoints from
- **scope** - scope(s) for endpoints to retrieve

Setter Operations

This is the full list of setter operations supported by `jsPlumb.selectEndpoints`:

- addClass
- removeClass
- addOverlay
- removeOverlay
- removeOverlays
- showOverlay
- hideOverlay
- showOverlays
- hideOverlays
- removeAllOverlays
- setLabel
- setPaintStyle
- setHoverPaintStyle
- setConnector
- setParameter
- setParameters
- repaint
- setType
- addType
- removeType
- toggleType
- setVisible
- setAnchor

Each of these operations returns a selector that can be chained.

Getter Operations

This is the full list of getter operations supported by `jsPlumb.selectEndpoints`:

- getLabel
- getOverlay
- isHover
- isVisible
- getParameter
- getParameters
- getType
- hasType
- getAnchor

Each of these operations returns an array whose entries are `[ value, Endpoint ]` arrays, where 'value' is the return value from the given Endpoint. Remember that the return values from a getter are not chainable, but a getter may be called at the end of a chain of setters.

Other methods/properties (not chainable):

- **delete** - deletes the Endpoints in the selection
- **detachAll** - detaches all Connections from the Endpoints in the selection
- **length** - this member reports the number of Endpoints in the selection
- **get(idx)** - this function allows you to retrieve an Endpoint from the selection

Examples

- Select all Endpoints and set their hover state to be false:

```
jsPlumb.selectEndpoints().setHover(false);
```

- Select all source Endpoints on "d1" and remove all Overlays:

```
jsPlumb.selectEndpoints({source:"d1"}).removeAllOverlays();
```

- Select all source Endpoints on "d1" and add the classes "foo" and "bar" to them :

```
jsPlumb.selectEndpoints({source:"d1"}).addClass("foo bar");
```

- Select all source Endpoints on "d1" and remove the class "foo" from them :

```
jsPlumb.selectEndpoints({source:"d1"}).removeClass("foo");
```

- Select all Endpoints in scope "foo" and set their fill style to be blue:

```
jsPlumb.selectEndpoints({ scope:"foo" }).setPaintStyle({ fill:"blue" });
```

- Select all Endpoints from "d1" and detach their Connections:

```
jsPlumb.selectEndpoints({source:"d1"}).detachAll();
```

`.each` iterator

The return value of jsPlumb.selectEndpoints also has a `.each` function that allows you to iterate through the list, performing an operation on each one:

```
jsPlumb.selectEndpoints({scope:"foo"}).each(function(endpoint) {
                // do something
});
```

`.each` is itself chainable:

```
jsPlumb.selectEndpoints({scope:"foo"}).each(function(endpoint) {
                // do something

}).setHover(true);
```

Animation

jsPlumb offers an `animate` function, which wraps the underlying animation engine for whichever library you happen to be using, and inserts a callback for jsPlumb to repaint whatever it needs to at each step. You could of course do this yourself; it's a convenience method really.

The method signature is:

`jsPlumb.animate : function(el, properties, options)`

The arguments are as follows: - **el** - element id, or element object from the library you're using. - **properties** - properties for the animation, such as duration etc. - **options** - options for the animation, such as callbacks etc.

jQuery and the 'revert' option

jQuery offers a `revert` option that you can use to instruct it to revert a drag under some condition. This is rendered in the UI as an animation that returns the drag object from its current location to wherever it started out. It's a nice feature.

Unfortunately, the animation that runs the revert does not offer any lifecycle callback methods - no 'step', no 'complete', etc - so its not possible for jsPlumb to know that the revert animation is taking place.

Vanilla jsPlumb supported properties

The adapter used by vanilla jsPlumb (from 1.6.0 onwards) supports only `left` and `top` properties in the animate method.

Utility Functions

This page contains a summary of some common scenarios you may find when using jsPlumb, and what functions jsPlumb offers to help you.

Repainting an element or elements

Assuming you use `jsPlumb.draggable` to initialise your draggable elements, you do not typically need to instruct jsPlumb to repaint. However, there are times when you need to, for instance:

- you have resized an element and you need jsPlumb to recompute the location of Endpoints on that element
- you have moved an element programmatically
- you didn't actually use `jsPlumb.draggable` to initialise some draggable element.

To force a repaint, use:

```
jsPlumb.repaint(el, [ui])
```

The first argument - `el` - can be a number of different datatypes:

- a string, representing the id of some element
- a list of strings, representing the ids of some elements
- a DOM element
- a list of DOM elements
- a selector from your underlying library

Performance: provide the element offset

Notice the `ui` argument? This is an offset in the form:

```
{ left:X, top:Y }
```

...which, if provided, means that jsPlumb does not have to go and compute the offset of the element.

Note this is of course relevant only if you are repainting a single element. If you provide this when you're painting multiple elements, they will all be assumed to be at the given offset!

Repainting everything

To repaint everything:

```
jsPlumb.repaintEverything();
```

Element Ids

jsPlumb uses element ids as keys for Connections and Endpoints, so if an element id changes, jsPlumb needs to know about it. You can either have jsPlumb do it for you:

```
jsPlumb.setId(el, newId);
```

or tell jsPlumb about it afterwards:

```
jsPlumb.setIdChanged(oldId, newId);
```

jsPlumb.connect Examples

This section provides various examples of how to use the programmatic API to establish Connections.

The basic syntax of a call is that you execute 'connect', providing a source and a target, and optionally a paintStyle and preferences for where you want the plumbing to be anchored on each element, as well as the type of connector to use.

- Connect window1 to window2 with the default settings:

  ```
  jsPlumb.connect({source:"window1", target:"window2"});
  ```

- Connect window1 to window2 with a 15 pixel wide yellow Connector, and a slightly brighter endpoint (remember the default Endpoint is a Dot):

  ```
  jsPlumb.connect({
    source:'window1',
    target:'window2',
    paintStyle:{strokeWidth:15,stroke:'rgb(243,230,18)'},
    endpointStyle:{fill:'rgb(243,229,0)'}
  });
  ```

- Connect window1 to window2 with a 15 pixel wide yellow Connector, and a slightly brighter endpoint:

  ```
  jsPlumb.connect({
    source:'window1',
    target:'window2',
    paintStyle:{strokeWidth:15,stroke:'rgb(243,230,18)'},
    endpointStyle:{fill:'rgb(243,229,0)'}
  });
  ```

- Connect window3 to 'window4' with a 10 pixel wide, semi opaque blue Connector, anchored to the left middle of window3, and the right middle of window4, with a Rectangle endpoint of width 10 and height 8:

  ```
  jsPlumb.connect({
    source:'window3',
    target:'window4',
    paintStyle:{ strokeWidth:10, stroke:'rgba(0, 0, 200, 0.5)' },
    anchors:["Right", "Left"],
    endpoint:[ "Rectangle", { width:10, height:8 } ]
  });
  ```

- Connect window2 to window3 with a default Connector from the top center of window2 to the bottom center of window3, and rectangular endpoints:

  ```
  jsPlumb.connect({
    source:'window2',
    target:'window3',
    paintStyle:{strokeWidth:8, stroke:'rgb(189,11,11    )'},
    anchors:["Bottom", "Top"],
    endpoint:"Rectangle"
  });
  ```

- Connect window1 to window2 with a 15 px wide yellow Bezier. Endpoints are a slightly lighter shade of yellow.

  ```
  jsPlumb.connect({
    source:'window1',
    target:'window2',
    anchors:["Bottom", [0.75,0,0,-1]],
    paintStyle:{strokeWidth:15,stroke:'rgb(243,230,18)'},
    endpointStyle:{fill:'rgb(243,229,0)'}
  });
  ```

- Connect window3 to window4 with a 10px wide blue-ish half transparent Bezier. Put Endpoints underneath the element they attach to. The Endpoints have a radial gradient. Both ways of specifying gradient positioning are shown here.

  ```
  var w34Stroke = 'rgba(50, 50, 200, 1)';
  var w34HlStroke = 'rgba(180, 180, 200, 1)';
  jsPlumb.connect( {
    source:'window3',
    target:'window4',
    paintStyle:{strokeWidth:10, stroke:w34Stroke},
    anchors:["RightMiddle", "LeftMiddle"],
    endpointStyle:{ gradient : {stops:[[ 0, w34Stroke ], [ 1, w34HlStroke ]], offset:17.5, innerRadius:15 }, radius:35},
    //endpointStyle:{ gradient : {stops:[[0, w34Stroke], [1, w34HlStroke]], offset:'78%', innerRadius:'73%'}, radius:35 }
  });
  ```

- Connect window2 to window3 with an 8px red Bezier and default rectangular endpoints. See also how the first Anchor is specified here - this is how you create Anchors in locations jsPlumb does not offer shortcuts for. The Endpoints in this example have linear gradients applied.

  ```
  var w23Stroke = 'rgb(189,11,11)';
  jsPlumb.connect({
      source:'window2',
      target:'window3',
      paintStyle:{strokeWidth:8,stroke:w23Stroke},
      anchors:[[0.3,1,0,1], "Top"],
      endpoint:"Rectangle",
      endpointStyles:[{ gradient : {stops:[[0, w23Stroke], [1, '#558822']] }},
              { gradient : {stops:[[0, w23Stroke], [1, '#882255']] }}]
  });
  ```

- Connect window5 to window6 from center to center, 5px wide line that is green and half transparent. the Endpoints are 125px in radius and spill out from underneath their elements.

  ```
  jsPlumb.connect({
    source:'window5',
    target:'window6',
    anchors:["Center", "Center"],
    paintStyle:{strokeWidth:5,stroke:'rgba(0,255,0,0.5)'},
    endpointStyle:{radius:125}
  });
  ```

- Connect window4 to window5 from bottom right to top left, with a 7px straight line purple Connector, and an image as the endpoint, placed on top of the element it is connected to.

```
jsPlumb.connect({
source:"window4",
target:"window5",
anchors:[ "BottomRight","TopLeft" ],
paintStyle:{ strokeWidth:7, stroke:"rgb(131,8,135)" },
endpoint:[ "Image", { src:"http://morrisonpitt.com/jsPlumb/img/endpointTest1.png" } ],
connector:"Straight"
});
```

- Connect window5 to window6 between their center points with a semi-opaque connector, and 125px endpoints:

```
jsPlumb.connect({
  source:"window5",
  target:"window6",
  anchors:[ "Center", "Center" ],
  paintStyle:{ strokeWidth:5, stroke:"rgba(0,255,0,0.5)" },
  endpointStyle:{ radius:125 }
});
```

- Connect window7 to window8 with a 10 pixel wide blue Connector, anchored on the top left of window7 and the bottom right of window8:

```
jsPlumb.connect({
  source:"window7",
  target:"window8",
  paintStyle:{ strokeWidth:10, stroke:"blue" },
  anchors:[ "TopLeft", "BottomRight" ]
});
```

- Connect the bottom right corner of window4 to the top left corner of window5, with rectangular Endpoints of size 40x40 and a hover color of light blue:

```
jsPlumb.connect({
  source:"window4",
  target:"window5",
  anchors:["BottomRight","TopLeft"],
  paintStyle:{strokeWidth:7,stroke:'rgb(131,8,135)'},
  hoverPaintStyle:{ stroke:"rgb(0, 0, 135)" },
  endpointStyle:{ width:40, height:40 },
  endpoint:"Rectangle",
  connector:"Straight"
});
```

- Connect window1 to window2 with the default paint settings but provide some drag options (which are passed through to the underlying library's draggable call):

```
jsPlumb.connect({
  source:'window1',
  target:'window2',
  dragOptions:{
    cursor:'crosshair'
  }
});
```

Draggable Connections Examples

A note on `dragOptions` and `dropOptions`

dragOptions

There are two methods in jsPlumb that allow you to configure an element from which Connections can be dragged - **addEndpoint** and **makeSource**. Each of these methods supports a `dragOptions` object is one of the parameters in its options. The allowed values in this object vary depending on the drag library in use: if you are using vanilla jsPlumb then the drag library is https://github.com/jsplumb/katavorio; otherwise with the jQuery flavour it is jQuery UI. `dragOptions` is passed directly through to the underlying library; check the docs for each of these to find out what options are available to you.

dropOptions

There are two methods that allow you to configure an element as a Connection drop target - **addEndpoint** and **makeTarget**. Where the source methods support `dragOptions`, these methods (and yes, one of these methods is the same as a method you'd use to configure a Connection source), these methods support a `dropOptions` parameter. Again, check the appropriate documentation for details on supported values in this object.

Examples

This is a list of examples of how to use jsPlumb to create Connections using drag and drop. The basic procedure is:

1. Create Endpoints and register them on elements in your UI, or create a source Endpoint and then make some element a drop target
2. Drag and Drop

There are plenty of options you can set when doing this...it will be easier to show you some examples:

- Define an Endpoint with default appearance, that is both a source and target of new Connections:

```
var endpointOptions = { isSource:true, isTarget:true };
```

- Register that Endpoint on `window3`, specifying that it should be located in the top center of the element:

```
var window3Endpoint = jsPlumb.addEndpoint('window3', { anchor:"Top" }, endpointOptions );
```

Notice here the usage of the three-argument `addEndpoint` - we can reuse `endpointOptions` with a different Anchor for another element. This is a useful practice to get into.

- Register that Endpoint on window4, specifying that it should be located in the bottom center of the element:

```
var window4Endpoint = jsPlumb.addEndpoint('window4', { anchor:"BottomCenter" }, endpointOptions );
```

Now we have two Endpoints, both of which support drag and drop of new Connections. We can use these to make a programmatic Connection, too, though:

- Connect window3 to window4 with a 25px wide yellow Bezier that has a 'curviness' of 175:

```
jsPlumb.connect({
  source:window3Endpoint,
  target:window4Endpoint,
  connector: [ "Bezier", { curviness:175 } ],
  paintStyle:{ strokeWidth:25, stroke:'yellow' }
});
```

- Define an Endpoint that creates Connections that are 20px wide straight lines, that is both a source and target of new Connections, and that has a `scope` of `blueline`. Also, this Endpoint mandates that once it is full, Connections can no longer be dragged from it (even if `reattach` is specified on a Connection):

```
var endpointOptions = {
          isSource:true,
          isTarget:true,
          connector : "Straight",
          connectorStyle: { strokeWidth:20, stroke:'blue' },
          scope:"blueline",
          dragAllowedWhenFull:false
};
```

- Define an Endpoint that will be anchored to `TopCenter`. It creates Connections that are 20px wide straight lines, that are both a source and target of new Connections, and that have a `scope` of `blueline`. Also, this Endpoint mandates that once it is full, Connections can no longer be dragged from it (even if `reattach` is specified on a Connection):

```
var endpointOptions = {
          anchor:"Top",
          isSource:true,
          isTarget:true,
          connector : "Straight",
          connectorStyle: { strokeWidth:20, stroke:'blue' },
          scope:"blueline",
          dragAllowedWhenFull:false
};
```

- Define an Endpoint that will create a dynamic anchor which can be positioned at `Top` or `Bottom`. It creates Connections that are 20px wide straight lines, it is both a source and target of new Connections, and it has a 'scope' of 'blueline'. Also, this Endpoint mandates that once it is full, Connections can no longer be dragged from it (even if `reattach` is specified on a Connection):

```
var endpointOptions = {
          anchor:[ "TopCenter", "BottomCenter" ],
          isSource:true,
          isTarget:true,
          connector : "Straight",
          connectorStyle: { strokeWidth:20, stroke:'blue' },
          scope:"blueline",
          dragAllowedWhenFull:false
};
```

- Exactly the same as before, but shows how you can use `anchors` instead of `anchor`, if that makes you feel happier:

```
var endpointOptions = {
  anchors:[ "TopCenter", "BottomCenter" ],
  isSource:true,
  isTarget:true,
  connector : "Straight",
  connectorStyle: { strokeWidth:20, stroke:'blue' },
  scope:"blueline",
  dragAllowedWhenFull:false
};
```

- Define an Endpoint that is a 30px blue dot, creates Connections that are 20px wide straight lines, is both a source and target of new Connections, has a `scope` of `blueline`, and has an event handler that pops up an alert on drop:

```
var endpointOptions = {
  isSource:true,
  isTarget:true,
  endpoint: [ "Dot", { radius:30 } ],
  style:{fill:'blue'},
  connector : "Straight",
  connectorStyle: { strokeWidth:20, stroke:'blue' },
  scope:"blueline",
  dropOptions:{
    drop:function(e, ui) {
      alert('drop!');
    }
  }
};
```

- Same example again, but `maxConnections` being set to -1 means that the Endpoint has no maximum limit of Connections:

```
var endpointOptions = {
          isSource:true,
          isTarget:true,
          endpoint: [ "Dot", {radius:30} ],
          style:{ fill:'blue' },
          maxConnections:-1,
          connector : "Straight",
          connectorStyle: { strokeWidth:20, stroke:'blue' },
          scope:"blueline",
          dropOptions:{
          drop:function(e, ui) {
            alert('drop!');
          }
      }
};
```

- Assign a UUID to the Endpoint options created above, and add as Endpoints to "window1" and "window2":

```
jsPlumb.addEndpoint("window1", { uuid:"abcdefg" }, endpointOptions );
jsPlumb.addEndpoint("window2", { uuid:"hijklmn" }, endpointOptions );
```

- Connect the two Endpoints we just registered on "window1" and "window2":

```
jsPlumb.connect({uuids:["abcdefg", "hijklmn"]});
```

- Create a source Endpoint, register it on some element, then make some other element a Connection target:

```
var sourceEndpoint = { isSource:true, endpoint:[ "Dot", { radius:50 } ] };
var targetEndpoint = { endpoint:[ "Rectangle", { width:10, height:10 } ] };
jsPlumb.addEndpoint( "window1", sourceEndpoint );
jsPlumb.makeTarget( "window2", targetEndpoint );
```

Notice that the endpoint definition we use on the target window does not include the `isTarget:true` directive. jsPlumb ignores that flag when creating a

Connection using an element as the target; but if you then tried to drag another connection to the Endpoint that was created, you would not be able to. To permit that, you would set `isTarget:true` on the targetEndpoint options defined above.

Utility Functions

- Detach window5 from all connections

  ```
  jsPlumb.detachAll("window5");
  ```

- Hide all window5's connections

  ```
  jsPlumb.hide("window5");
  ```

- Hide all window5's connections endpoints

  ```
  jsPlumb.hide("window5", true);
  ```

- Show all window5's connections

  ```
  jsPlumb.show("window5");
  ```

- Show all window5's connections and endpoints. Note that in the case that you call jsPlumb.show with two arguments, jsPlumb will also not make a connection visible if it determines that the other endpoint in the connection is not visible.

  ```
  jsPlumb.show("window5", hide);
  ```

- Toggle the visibility of window5's connections

  ```
  jsPlumb.toggleVisible("window5");
  ```

- Force repaint of all of window5's connections

  ```
  jsPlumb.repaint("window5");
  ```

- Force repaint of all of window5, window6 and window11's connections

  ```
  jsPlumb.repaint( [ "window5", "window6", "window11" ] );
  ```

- Force repaint of every connection

  ```
  jsPlumb.repaintEverything();
  ```

- Detach every connection that the given instance of jsPlumb is managing

  ```
  jsPlumb.detachEveryConnection();
  ```

- Detach all connections from "window1"

  ```
  jsPlumb.detachAllConnections("window1");
  ```

- Remove all Endpoints for the element 'window1', deleting their Connections.

  ```
  jsPlumb.removeAllEndpoints("window1");
  ```

- Deletes every Endpoint managed by this instance of jsPlumb, deleting all Connections. This is the same as jsPlumb.reset(), effectively, but it does not clear out the event listeners list.

  ```
  jsPlumb.deleteEveryEndpoint();
  ```

- Deletes the given Endpoint and all its Connections.

  ```
  jsPlumb.deleteEndpoint(endpoint);
  ```

- Removes every endpoint, detaches every connection, and clears the event listeners list. Returns jsPlumb instance to its initial state.

  ```
  jsPlumb.reset();
  ```

- Set window1 to be not draggable, no matter what some jsPlumb command may request.

  ```
  jsPlumb.setDraggable("window1", false);
  ```

- Set window1 and window2 to be not draggable, no matter what some jsPlumb command may request.

  ```
  jsPlumb.setDraggable(["window1","window2"], false);
  ```

- Initialises window1 as a draggable element (all libraries). Passes in an on drag callback

  ```
  jsPlumb.draggable("window1");
  ```

- Initialises window1 and window2 as draggable elements (all libraries)

  ```
  jsPlumb.draggable(["window1","window2"]);
  ```

- Initialises window1 as a draggable element (all libraries)

  ```
  jsPlumb.draggable("window1");
  ```

- Initialises all elements with class 'window' as draggable elements (jQuery)

  ```
  jsPlumb.draggable($(".window"));
  ```

- Initialises window1 as a draggable element (jQuery)

  ```
  jsPlumb.draggable($("#window1"));
  ```

jsPlumb Development

Building jsPlumb

Instructions for running a build can be found here.

Pluggable Library Support

Out of the box, jsPlumb can be run standalone, or on top of jQuery. This is achieved by delegating several core methods - tasks such as finding an element by id, finding an element's position or dimensions, initialising a draggable, etc - to the library in question.

It is possible to develop your own library adapter, should you need to, although since the introduction of "vanilla" jsPlumb there is less reason to do so. If you do decide to do so, the existing implementations may be documented well enough for you to create your own, but contact us if you need assistance.

ExtJS

To date, we've been contacted by a few different groups who have been working on an ExtJS adapter, but nothing seems to have yet come to fruition.

Building

jsPlumb consists of several scripts in development, which are concatenated together when it comes time to release. You can run this yourself on the source - you need Grunt and Jekyll.

Building jsPlumb

In the main project directory, execute the following command:

```
grunt build
```

The output is written into `./dist`. Subsequent builds will overwrite the contents of the `dist` directory.

Building Custom Versions

1.5.0 introduced the ability to build custom versions of jsPlumb, omitting connectors you do not need. This does not, admittedly, save you a huge amount; future releases will take this ability a step further and allow you, for instance, to leave out the whole `makeTarget/makeSource` module, should you wish to.

If you need only the Flowchart connectors, you'd do this:

```
grunt build --connectors=flowchart
```

If you need only the State Machine connectors:

```
grunt build --connectors=statemachine
```

Valid values for `connectors` are connector names, all in lower case.

Note It is important you do not leave a space between values for the `connectors` or `renderers` parameters. Grunt will get confused if you do.

Documentation

jsPlumb's documentation is maintained in the associated wiki on GitHub, which is at git@github.com:sporritt/jsPlumb.wiki.git. A branch that matches the current dev branch is used; at release time this branch is merged into master and then you see it on GitHub.

Documentation is edited locally using Gollum (or by hand) as per the discussion on this page. During development, all of the `Documentation` links in the demo pages point to `http://localhost:4567`, as this is the default port on which Gollum runs.

The jsPlumb build script converts all the markdown files into html and inserts them into a final template, copying everything into `dist/docs`.

API Documentation

API documentation, from release 1.6.0, is being generated with YUIDoc, as part of the Grunt build.

The build places the API docs in `dist/apidocs`.

2.2.0

- Overhaul of keys used in paintStyle and hoverPaintStyle objects:

strokeStyle -> style fillStyle -> fill lineWidth -> strokeWidth outlineColor -> outlineStroke outlineWidth -> outlineWidth (yes, unchanged)

2.1.5

- issue 533 - Dragging multiple nodes causes incorrect connectors position
- `reset` method sets hover suspended flag to false now.

2.1.4

- issue 530 - Further fix related to issue 530, in which elements that had connections prior to being added to a group were sometimes getting an offset applied when dragging. The fix for this removed some code that was put in for issue 231, but it turns out the fix for issue 231 had broken somewhere along the line and this change set that right too.

2.1.3

- issue 530 - Element with existing connections being added to Groups.
- issue 526 - bower version incorrect

2.1.2

- issue 523 - Endpoint click registration problems
- issue 522 - Groups documentation

2.1.1

- bugfix for groups: element exposed now via getEl method, not directly as el.

2.1.0

- 'elementDraggable' event now fired whenever an element is made draggable via the `draggable` function
- add support for 'groups' - elements that can contain other elements, and which are collapsible.

- upgrade to Mottle 0.7.2. a few fixes for event delegation.

- upgrade to katavorio 0.15.0

- upgrade to mottle 0.7.2

- upgrade to jsBezier 0.8

- upgrade to Biltong 0.3

ISSUES

- 483 - srcElement undefined in Firefox
- 484 - changed a couple of variables refs so that they are not reserved words

2.0.6

- add `connectionAborted` event, fired whenever a new connection is abandoned before being dropped on an endpoint or target. also fired if `beforeDrop` interceptor returns false.

- fixed docs for `connectionDetached` to call out the fact that it is not fired when a connection is abandoned.

ISSUES

- 472 - Pending connections do not fire the documented events
- 469 - Scopes not applied to new drag & drop connections
- 452 - Why "connection.scope" property cannot get scope value ?

2.0.5

- Refactor Bezier and StateMachine connectors to extend common AbstractBezierConnector parent. This means Bezier connectors now support loopback connections.

- add support for loopback connections to Flowchart connector (issue 457).

ISSUES

- 458 connectionDetached is fired the first time only

- 457 'Flowchart' connector: loopback connections broken

- 451 cannot bind events to arrow overlays

- 446 addClass and removeClass on Endpoint and Connection now also add/remove class from their overlays, by default. This can be overridden by providing 'true' as the second argument to the addClass/removeClass methods.

- 434 wrong arrow drawing (offset) when creating a connection on IE9

2.0.4

- upgrade to Katavorio 0.13.0
- upgrade to Mottle 0.7.1

- add `droppable` method to jsPlumbInstance: the ability to make *elements* droppable. Not connections or endpoints - DOM elements.
- fixes for offset calculation when dragging a nested element.

2.0.3

Issues

- 444 - maxConnections not honoured in makeSource

Backwards Compatibility

- `removeFromPosse` now requires the ID of the Posse, since the new Katavorio version supports multiple Posses per element.

New Functionality

- Upgrade to Katavorio 0.12.0, with support for multiple Posses and active/passive elements in a Posse.
- `removeFromAllPosses(element)` method added.

Miscellaneous

- Fixed an issue in which overlays on Endpoint types were not being converted to 'full' syntax upon registration. This was an internal issue that could manifest in user code occasionally.

- We now ensure drag scope is set on an element when source scope changes, even though the code can derive source scope when the user begins to drag. The Toolkit edition makes use of this new update.

2.0.2

Fix issues with CSS class documentation.

2.0.1

Bugfix release: connectionDetached event was no longer firing.

2.0.0

Backwards Compatibility

- Removal of the VML renderer. IE8 no longer supported.
- All class names such as `_jsPlumb_connector` renamed to, for example, `jsplumb-connector`.
- makeSource and makeTarget require explicit anchor/endpoint parameters: they do not source these things from the jsPlumb Defaults.

New Functionality

- makeSource now supports multiple registrations per element, keyed by the `connectionType` parameter. You can configure elements to be connection sources for different connection types, and also when you call `connect` with a `type` parameter that matches a `makeSource` registration, that type will be used.
- new connection drag: if the type of connection is known, that type's target endpoint is now used.
- addition of support for `dragProxy` to endpoint/makeSource: an endpoint spec defining what the drag endpoint should look like when dragging a new connection. The existence of a `dragProxy` will override any other behaviour (such as the behaviour discussed in the point above)
- addition of "posses" - groups of elements that should always be dragged together.
- when dragging a new connection, jsPlumb now uses as the source endpoint a true representation of what the endpoint will be if a connection is established. Previous versions just used a static, in-place, endpoint.

1.7.10

Changes between 1.7.9 and 1.7.10

- Small update to getOffset to make it return the correct value if the input element was the container itself.
- Small update to animation to fix incorrect falsey check.
- Documented the `on` method of a `jsPlumbInstance` in the API docs.
- `on` and `off` event registration methods now return the current jsPlumb instance

1.7.9

Changes between 1.7.8 and 1.7.9

- No more jQuery flavour. Vanilla jsPlumb is the only jsPlumb, and as such, has been renamed to simply `jsPlumb-1.7.9.js`.
- First version of jsPlumb to be published to npm.
- Addition of getManagedElements method. Returns a map of all the elements the instance of jsPlumb is currently managing.

Issues

- **421** svg gradient elements not cleaned up properly

1.7.8

Changes between 1.7.7 and 1.7.8

Issues

- **381** - instance.detach(connection) will detach source endpoint as well
- **419** - endpoints not cleaned up properly when connection converted to looback to endpoints not cleaned up properly when connection converted to loopback
- **420** - Image endpoint not cleaned up correctly

1.7.7

Changes between 1.7.6 and 1.7.7

Issues

- **408** - setIdChanged doesn't correctly handle element sources/targets
- **410** - setConnector (whether applied via type or directly) removes custom css classes of other types
- **412** - Endpoint style cannot be transparent
- **415** - Unnecessary endpoint may be created at when drag and drop endpoint from one node to another.

1.7.6

Changes between 1.7.5 and 1.7.6

A minor bugfix release, with a few new options for controlling connection detachment (and one small backwards compatibility issue to be aware of)

Backwards Compatibility

- All versions of jsPlumb prior to 1.7.6 would fire `beforeDetach` for both new Connection drags and also dragging of existing Connections. As of 1.7.6 this latter behaviour has been moved to the `beforeStartDetach` interceptor.

New Functionality

- `revalidate` now supports the same arguments as repaint - an ID, an Element, or a list-like object (such as the results of $(..) or document.querySelectorAll)

- added `beforeStartDetach` interceptor: a function that is called before an existing connection is dragged off of one of its endpoints, and which can return false to cancel the drag.

- The `unbind` method on various objects (jsPlumbInstance, Connection, Endpoint to name a few) now supports passing a Function to be unbound, rather than just some event name.

- Connectors now have a `getLength` function, which returns their length in pixels. To access from a Connection, you need to first get the connector: `someConnection.getConnector().getLength()`

Issues

- **350** - recalculateOffsets not working
- **353** - multiple select disabled
- **367** - rendering and drag/drop errors when parent element scrolled
- **369** - unbinding events
- **383** - jsPlumb.setDraggable fails for getElementsByClassName return value
- **392** - onMaxConnections jpc isn't defined
- **402** - offset update cache
- **404** - statemachine demo makes ghost endpoints

1.7.5

Changes between 1.7.4 and 1.7.5

A minor-ish release; no changes to the API. Some refactoring of JS and of CSS. But one notable thing is that touch events on Windows touch laptops are working now (in Chrome and IE; FF seems to still have issues)

Backwards Compatibility

- The jQuery flavour was removed from the `main` section in `bower.json`.

Issues

- **295** - draggable not working in chrome
- **340** - Draggable stop event doesn't get called on all elements when dragging multiple elements
- **341** - Add possibility to change z-order of the "inPlaceCopy" endpoint.
- **344** - add getUuids method to Connection
- **345** - Error when two linked objects are with exactly same position

1.7.4

Changes between 1.7.3 and 1.7.4

Issues

- **237** - scroll is ignored in offset calculations
- **314** - jsPlumbUtil is not defined (webpack)
- **329** - Scroll issue
- **332** - endpoint label not working in newest version
- **333** - ReattachConnections not working when a connection is detached (jquery & vanilla 1.7.3)
- **336** - cannot drop a connection back on the endpoint to which it was previously attached

1.7.3

Changes between 1.7.2 and 1.7.3

Predominantly a minor bugfix release, this version brings a degree of uniformity to the behaviour of elements configured with `makeSource` and `makeTarget`, and is a recommended upgrade if you are currently using any other 1.7.x version.

New Functionality

- There is a new interceptor in this release: `beforeDrag`. You can use it to abort dragging a connection as soon as it starts, and also to supply the initial data for a Connection that uses a parameterized type.
- Added `jsPlumb.empty` function: remove child content from a node, including endpoints and connections, but not the element itself.

Backwards Compatibility

- The `doWhileSuspended` method has been aliased as `batch`, and `doWhileSuspended` is now deprecated, to be removed in version 2.0.0.

Issues

- **187** - jsPlumb.draggable() doesn't work with forms
- **281** - beforeDetach not triggered by `jsPlumb.detachAllConnections`
- **287** - Cannot drop source of connection on makeTarget element after 1.6.4
- **289** - Cannot prevent drop of source edge using beforeDrop on nested makeTarget elements
- **297** - Distinguish drag\click for Vanilla jsPlumb
- **298** - Fix for using library inside shadowDom (e.g. Polymer etc.)
- **307** - Setting Container multiple times fires events multiple times
- **311** - addType resets overlays
- **313** - setContainer does not work when container has overflow: scroll;
- **315** - setConnector removes existing overlays
- **317** - Docs incorrectly refer to "mouseenter"
- **326** - Connections not updating position - (detach, delete, readd, reconnect)

1.7.2

Changes between 1.7.1 and 1.7.2

- Reverted a minor bugfix introduced by the fix for issue 276
- Updated continuous anchors to allow for several Continuous anchors to be in use on the one element.

1.7.1

Changes between 1.7.0 and 1.7.1

Issues

- **276** - TypeError on dragging empty target endpoint

1.7.0

Changes between 1.6.4 and 1.7.0

Backwards Compatibility

- Perhaps the biggest change between 1.6.4 and 1.7.0 is that YUI and MooTools are no longer supported. It is recommended you use vanilla jsPlumb now. jQuery is still supported but it is neither as fast nor does it have as many features as vanilla jsPlumb.

- The `parent` argument to the `makeSource` function is no longer supported. It was being kept because neither YUI nor MooTools have the ability to support a drag filter, but now that those libraries are not supported this feature has been removed. The `filter` approach is much more powerful.

New Functionality

Perhaps not strictly new functionality, but shiny enough to warrant being associated with the word "new", is the fact that jsPlumb 1.7.0 is considerably faster than any previous version. A rough comparison: the default settings for the load test in jsPlumb generate 360 connections in total between 10 elements. in 1.6.4 this test averages about 1600ms in Chrome on a Mac. In 1.7.0 that number is about 600ms on the same computer.

Issues

- **178** - Detachable endpoints: different behaviour between connect() and mouse-based connections
- **214** - Endpoint stays visible when should be terminated (right mouse button)
- **242** - Distinguish drag\click for Vanilla jsPlumb
- **245** - reinstate isConnectedTo method on Endpoint
- **246** - outlineColor ignored when gradient defined in paintStyle
- **248** - dynamic anchor create fail
- **257** - allow for the scope of a makeSource element to be changed
- **258** - Typo in documentation: s/container/Container
- **260** - isSource and isTarget usage with makeSource and makeTarget causes broken connections
- **261** - Two target endpoints close to each other: "TypeError: Cannot read property '0' of null"
- **262** - hoverPaintStyle only works for the first connection (maxConnections > 1)
- **263** - TypeError: conn.endpoints is null
- **267** - continuous anchors with faces set do not paint on selected faces when not connected to anything
- **268** - Endpoint "Blank" generates endpoint with class "undefined"
- **269** - Source endpoint does not/cannot respect uniqueEndpoint setting
- **270** - Support `endpointStyle` in args to addEndpoint and makeSource/makeTarget

1.6.4

Changes between 1.6.3 and 1.6.4

Backwards Compatibility

- No issues

New Functionality

- Connection types support 'anchor' and 'anchors' parameters now.

Miscellaneous

- YUI adapter now sets a 'base' url and retrieves everything via https.

1.6.3

Changes between 1.6.2 and 1.6.3

Backwards Compatibility

- No issues

New Functionality

- Added optional `allowLoopback` boolean parameter to vanilla jsPlumb's `makeTarget` method.
- When using parameterized types, unmatched values are now replaced with blank strings, rather than being left in place. For instance, if you had `label="${foo}"`, and you passed a blank 'foo' value, you used to see `"${foo}"`. Now you see `""`.
- You can set `visible:false` on an overlay spec, to have it initially invisible.
- Added `setHoverEnabled` method to jsPlumb.
- Added `clearTypes` method to Connection and Endpoint
- Connection and Endpoint types now support `cssClass` property. These are merged into an array if multiple types declare a cssClass.

Issues

- **222** - Endpoints incorrectly calculated when the anchor faces of source/target are set to left/right
- **223** - beforeDetach not fired by jsPlumb
- **224** - endpointStyle of the jsPlumb.connect method does not work
- **227** - MaxConnections=1 console log error
- **230** - Endpoints not cleaned up after connector move
- **236** - makeTarget/makeSource drag issues
- **241** - Dropping existing connection creates an orphaned endpoint when beforeDrop returns false
- **243** - setConnector not correctly re-assigning event handler on overlays

1.6.2

Changes between 1.6.1 and 1.6.2

Backwards Compatibility

- 1.6.2 has improved behaviour for determining what element to use as the Container. Previous 1.6.x versions defaulted to the document body, with the docs strongly recommending you set a Container. From 1.6.2, if there is no Container set when the user makes a first call to either addEndpoint, makeSource, makeTarget or connect, the Container is set to be the offsetParent of either the element being configure (in the case of `addEndpoint`, `makeSource` and `makeTarget`), or the source element, for the `connect` method.

- a consequence of this is that you can no longer manipulate `Defaults.Container` manually. Your changes will be ignored; `Defaults.Container` is referenced only in the constructor or in the `importDefaults` method. If you need access to the current Container, use the `getContainer` method.

- the order of parameters to the function `jsPlumbInstance.on` has changed, in the case that you are passing 4 parameters and using it for event delegation. Previously, the order was `(element, filter, eventId, callback)` and now the order is `(element, eventId, filter, callback)`. This brings it into line with the order of parameters in jQuery's `on` function. It is not very likely this will affect you: `jsPlumbInstance.on` is used internally, mostly (although it can be used to register events independently of jsPlumb if you want to use it).

New Functionality

- The Container inferencing discussed above is both a backwards compatibility issue and also new functionality!
- added `setContainer`, to allow you to move an entire jsPlumb UI to some new parent
- added `getContainer`, to allow you to retrieve the current Container.

Issues

- **207** - problem with absolute overlays
- **211** - setVisible(true) on hidden overlay whose connection has moved causes the overlay to repaint in the wrong place

1.6.1

This is a minor release in which a few issues related to zooming have been fixed.

Changes between 1.6.0 and 1.6.1

Backwards Compatibility

No issues

Issues

- **206** Fix documentation error about jsPlumb.Defaults.Endpoints

New Functionality

Better handling of zooming in vanilla jsPlumb.

1.6.0

Version 1.6.0 is a major release of jsPlumb. With this version ships a "vanilla" version - it relies on no external libraries, and also has a few features that the other library adapters do not (see below).

Changes between 1.5.5 and 1.6.0

Backwards Compatibility

- There is no support for the canvas renderer in jsPlumb 1.6.0.
- The way in which library adapters inject their functionality into jsPlumb has changed. This will affect very few people; contact jsPlumb if you need help with this.
- All elements added by jsPlumb are appended to the current "Container", which defaults to the document body. This differs from previous versions, in which if there was no Container set then jsPlumb would append elements to the parent of a connection's source endpoint. For this reason it is now more than ever recommended that you set a Container.
- The `container` parameter on `addEndpoint` or `connect` calls is no longer supported.

Issues

- **91** - Old ID is being used on events after setId
- **143** - SVG gradient fails when page url already contains a hash
- **153** - jsPlumb.animate no longer supports jQuery selectors
- **157** - connectionMoved event not fired (when using makeTarget)
- **162** - Connector 'Flowchart' occurs an error.
- **164** - makeSource fails when used in conjunction with uniqueEndpoint
- **173** - jsPlumb.setDraggable([element_id],false); fails
- **177** - Flowchart straight Line
- **202** - Spurious mouse events in connector with outline
- **203** - hoverClass on endpoints doesn't work

New Functionality

DOM Adapter

It isn't actually true to say that this adapter has no external dependencies; it actually relies on a couple of new projects written specifically for this (Mottle for events, and Katavorio for drag/drop support. However, these dependencies are wrapped into the concatenated jsPlumb 1.6.0 JS file.

Multiple element dragging

The DOM adapter supports dragging (and dropping!) multiple elements at once.

Multiple drag/drop scopes

Also supported are multiple scopes for each draggable/droppable element.

Using Vanilla jsPlumb with jQuery

Even if you have jQuery in the page you can use vanilla jsPlumb; it will accept jQuery selectors as arguments. Keep in mind that you won't get jQuery selectors out of it, though - any methods that return an Element will return plain DOM Elements and you'll need to turn them in jQuery selectors yourself.

Miscellaneous

- Events now have `this` set correctly
- Added qUnit tests for Vanilla, YUI and MooTools adapters
- Various YUI and MooTools methods were upgraded to support passing in an element list (`setId` for one)
- Added setSource/setTarget methods, allowing you to retarget a Connection programmatically.
- Reduced the amount of functionality that is delegated to a support library
- Rewrote the way support libraries are integrated

1.5.5

Changes between 1.5.4 and 1.5.5

Issues

- **138** - allow for connection type to be derived from connection params AND endpoint params.

1.5.4

Changes between 1.5.3 and 1.5.4

Issues

- **105** - Blank endpoint cleanup fails
- **116** - Assign anchors wont connect
- **117** - Assign anchors fail on source
- **127** - Docs on making elements draggable should note required CSS
- **128** - expose original event on `connectionDragStop` callback
- **129** - connection event fired twice by makeTarget with parent option.

New Functionality

- `"Assign"` anchors now work with the `makeSource` method.
- The `connectionDragStop` event now supplies the original event as the second argument to the callback function.

Miscellaneous

- fixed an issue causing SVG gradients to fail when a BASE tag is present in the document.

1.5.3

Changes between 1.5.2 and 1.5.3

Backwards Compatibility

- The fix for issue 112 involved making a change to the circumstances under which a `connectionDetached` event is fired. When you drag the source or target of an existing connection to some other endpoint, `connectionDetached` is no longer fired. Instead, a `connectionMoved` event is fired, containing the connection that was moved, the index of the endpoint that changed (0 for source, 1 for target), and the original and new source and target endpoints.

Issues

- **77** - Endpoint types should support Anchor parameter
- **88** - reinstate labelStyle parameter on Label overlay.
- **90** - overlay setVisible not working (SVG/VML)
- **95** - Nested element positions not updated
- **100** - add setParent function
- **101** - JS error when detaching connection during connection callback
- **103** - IE8: connector hide does not hide overlays or background lines
- **107** - remove the necessity to set isSource/isTarget in order to make an endpoint draggable
- **108** - strange anchor orientation behaviour
- **109** - Dropping new connections on overlapping elements leads to crash after connection is deleted
- **111** - Absolute positioned arrow in wrong location
- **112** - Deleting a connection after changing its source endpoint causes failure.
- **113** - IE8 - state machine - loops are not displayed

New Functionality

- A setParent function was added. jsPlumb changes the parent of some element and updates its internal references accordingly (issue 100).
- Endpoint types now support the anchor parameter (issue 77)
- The `labelStyle` parameter on Label overlays has made a comeback (issue 88). The argument went along the lines of it being useful if you wanted to programmatically generate a label style.
- jsPlumb now automatically updates the internal offsets of some element that has draggable children (obviating the need for you to call `recalculateOffsets` yourself).
- When making a programmatic connection to an endpoint that was not marked `isSource:true` or `isTarget:true`, if the connection is detachable then the endpoint is made draggable, in order to allow users to drag the connection to detach it. Connections dragged off of source or target endpoints in this way can be dropped back onto their original endpoint or onto other endpoints with the same scope, but you cannot subsequently drag a new connection from an endpoint that has been made draggable by this method.
- `connectionMoved` event added. This is fired whenever the source or target of an existing connection is dragged to some other Endpoint.

Miscellaneous

- An issue was fixed that was preventing the ability to supply a dynamic anchor with parameters, eg

  ```
  [ [ [ 1,0,0,1], [1,1,1,1] ], { selector:function() { ... } } ]
  ```

1.5.2

Changes between 1.5.1 and 1.5.2

Backwards Compatibility

- Issue 86, fixed in 1.5.2, changes the priority in which parameters are applied to a connection. The documentation has always stated that source takes priority, but in fact the code was the other way round, with target taking priority. Now source does take priority.

Issues

- **84** - jsPlumb 1.5.1 Arrow Disappears on IE8 when connector is straight
- **85** - dragging target endpoints created by makeTarget not working
- **86** - Connection parameters override order

Miscellaneous

- An issue that caused the SVG renderer to paint overlays before the connector was ready when the types API was used was also fixed.

1.5.1

Changes between 1.5.0 and 1.5.1

Issues

- **81** - Uncaught TypeError: Cannot read property 'uuid' of null
- **82** - Blank endpoint doesn't cleanup properly
- **83** - for connections made with makeTarget originalEvent is not set

1.5.0

Changes between 1.4.1 and 1.5.0

Release 1.5.0 contains several bugfixes and one or two minor enhancements, but the biggest change since 1.4.1 is the way jsPlumb handles inheritance internally - it has switched from a 'module pattern' architecture to a prototypal-based setup. The module pattern is good for information hiding, but it makes objects bigger, and its far easier to leak memory with that sort of arrangement than it is with a prototypal inheritance scheme.

The build has been switched from the original Ant build to Grunt with release 1.5.0, and with this has come the ability to build versions of jsPlumb that omit functionality you do not need (see here).

Backwards Compatibility

- `jsPlumb.addClass`, `jsPlumb.removeClass` and removed `jsPlumb.hasClass` removed. You don't need these. You can use the methods from the underlying library.
- `makeTargets` method removed from jsPlumb. You can pass an array or selector to `makeTarget`.
- `makeSources` method removed from jsPlumb. You can pass an array or selector to `makeSource`.
- `jsPlumb.detach` no longer supports passing in two elements as arguments. Use instead either

```
jsPlumb.detach({source:someDiv, target:someOtherDiv});
```

or

```
jsPlumb.select({source:someDiv, target:someOtherDiv}).detach();
```

- `jsPlumbConnectionDetached` event, which was deprecated, has been removed. Use `connectionDetached`.
- `jsPlumbConnection` event, which was deprecated, has been removed. Use `connection`.
- `Endpoint.isConnectedTo` method removed. it didnt work properly as it only checked for connections where the Endpoint was the source.
- Many places in jsPlumb that used to use library-specific selectors for elements now use pure DOM elements. It is best to re-select any elements you are getting from a jsPlumb object, even if you supplied them as a selector, as jsPlumb will have unwrapped your selector into a DOM element.

New Functionality

- `jsPlumb.setSuspendDrawing` returns the value of `suspendDrawing` *before* the call was made.
- `Endpoint.setElement` works properly now.

Issues Fixed

- **27** - investigate why a new connection is created after drag
- **37** - .addClass() not working - IE8
- **39** - problem about connectionDrag event
- **49** - Calling detachEveryConnection winds up calling repaintEverything once for each endpoint
- **51** - arrow overlay orientation at location 1 on flowchart connectors
- **54** - Memory Leak Issue
- **57** - DOMException while dragging endpoints
- **60** - flowchart connector start position wrong
- **63** - Flowchart midpoint=0 is ignored
- **65** - Uncaught exception in IE 8
- **69** - jsPlumb.detach(connection) is really slow with larger graphs
- **72** - Drag and drop connections fail to work correctly when using makeTarget
- **75** - changing continuous anchor is ignored
- **76** - jsPlumb doesn't work in XHTML documents

Miscellaneous

Nothing to report.