Raphaël—JavaScript Library

Main Function

Raphael

Creates a canvas object on which to draw. You must do this first, as all future calls to drawing methods from this instance will be bound to this canvas.

Parameters

1. container *HTMLElement* or *string*
2. width *number*
3. height *number*

or

1. x *number*
2. y *number*
3. width *number*
4. height *number*

or

1. all *array* (first 3 or 4 elements in the array are equal to [containerID, width, height] or [x, y, width, height]. The rest are element descriptions in format {type: type, <attributes>})

Usage

```
// Each of the following examples create a canvas that is 320px wide by 200px high
// Canvas is created at the viewport’s 10,50 coordinate
var paper = Raphael(10, 50, 320, 200);
// Canvas is created at the top left corner of the #notepad element
// (or its top right corner in dir="rtl" elements)
var paper = Raphael(document.getElementById("notepad"), 320, 200);
// Same as above
var paper = Raphael("notepad", 320, 200);
// Image dump
var set = Raphael(["notepad", 320, 200, {
    type: "rect",
    x: 10,
    y: 10,
    width: 25,
    height: 25,
    stroke: "#f00"
}, {
    type: "text",
    x: 30,
    y: 40,
    text: "Dump"
}]);
```

circle

Draws a circle.

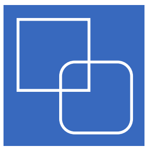
Parameters

1. x *number* X coordinate of the centre
2. y *number* Y coordinate of the centre
3. r *number* radius

Usage

```
var c = paper.circle(50, 50, 40);
```

rect



Draws a rectangle.

Parameters

1. x *number* X coordinate of top left corner
2. y *number* Y coordinate of top left corner
3. width *number*
4. height *number*
5. r *number* [optional] radius for rounded corners, default is 0

Usage

```
// regular rectangle
var c = paper.rect(10, 10, 50, 50);
// rectangle with rounded corners
var c = paper.rect(40, 40, 50, 50, 10);
```

ellipse

Draws an ellipse.

Parameters

1. x *number* X coordinate of the centre
2. y *number* X coordinate of the centre
3. rx *number* Horisontal radius
4. ry *number* Vertical radius

Usage

```
var c = paper.ellipse(50, 50, 40, 20);
```

image

Embeds an image into the SVG/VML canvas.

Parameters

1. src *string* URI of the source image
2. x *number* X coordinate position
3. y *number* Y coordinate position
4. width *number* Width of the image
5. height *number* Height of the image

Usage

```
var c = paper.image("apple.png", 10, 10, 80, 80);
```

set

Creates array-like object to keep and operate couple of elements at once. Warning: it doesn't create any elements for itself in the page.

Usage

```
var st = paper.set();
st.push(
    paper.circle(10, 10, 5),
    paper.circle(30, 10, 5)
);
st.attr({fill: "red"});
```

text

Raphaël
kicks
butt!

Draws a text string. If you need line breaks, put "\n" in the string.

Parameters

1. x *number* X coordinate position.
2. y *number* Y coordinate position.
3. text *string* The text string to draw.

Usage

```
var t = paper.text(50, 50, "Raphaël\nkicks\nbutt!");
```

path

Creates a path element by given path data string.

Parameters

1. pathString *string* [optional] Path data in SVG path string format (http://www.w3.org/TR/SVG/paths.html#PathData).

Usage

```
var c = paper.path("M10 10L90 90");
// draw a diagonal line:
// move to 10,10, line to 90,90
```

clear

Clears the canvas, i.e. removes all the elements.

Usage

```
var c = paper.path("M10 10L90 90");
paper.clear();
```

Element's generic methods

Each object created on the canvas shares these same generic methods:

node

Gives you a reference to the DOM object, so you can assign event handlers or just mess around.

Usage

```
// draw a circle at coordinate 10,10 with radius of 10
var c = paper.circle(10, 10, 10);
c.node.onclick = function () {
    c.attr("fill", "red");
};
```

paper

Internal reference to “paper” where object drawn. Mainly for use in plugins and element extensions.

Usage

```
Raphael.el.cross = function () {
    this.attr({fill: "red"});
    this.paper.path("M10,10L50,50M50,10L10,50")
        .attr({stroke: "red"});
}
```

remove

Removes element from the DOM. You can’t use it after this method call.

Usage

```
var c = paper.circle(10, 10, 10);
c.remove();
```

hide

Makes element invisible.

Usage

```
var c = paper.circle(10, 10, 10);
c.hide();
```

show

Makes element visible.

Usage

```
var c = paper.circle(10, 10, 10);
c.show();
```

rotate

Rotates the element by the given degree from its center point.

Parameters

1. degree *number* Degree of rotation (0 – 360°)
2. isAbsolute *boolean* [optional] Specifies is degree is relative to previous position (`false`) or is it absolute angle (`true`)

or

1. degree *number* Degree of rotation (0 – 360°)
2. cx *number* [optional] X coordinate of the origin of rotation
3. cY *number* [optional] Y coordinate of the origin of rotation

Usage

```
var c = paper.circle(10, 10, 10);
c.rotate(45);        // rotation is relative
c.rotate(45, true);  // rotation is absolute
```

translate

Moves the element around the canvas by the given distances.

Parameters

1. dx *number* Pixels of translation by X axis
2. dy *number* Pixels of translation by Y axis

Usage

```
var c = paper.circle(10, 10, 10);
// moves the circle 10 px to the right and down
c.translate(10, 10);
```

scale

Resizes the element by the given multiplier.

Parameters

1. Xtimes *number* Amount to scale horizontally
2. Ytimes *number* Amount to scale vertically
3. centerX *number* [optional] X of the center of scaling, default is the center of the element
4. centerY *number* [optional] Y of the center of scaling, default is the center of the element

Usage

```
var c = paper.circle(10, 10, 10);
// makes the circle 1.5 times larger
c.scale(1.5, 1.5);
// makes the circle half as wide, and 75% as high
c.scale(.5, .75);
```

attr

Sets the attributes of elements directly.

Parameters

1. attributeName *string*
2. value *string*

or

1. params *object*

or

1. attributeName *string* in this case method returns current value for given attribute name

or

1. attributeNames *array* in this case method returns array of current values for given attribute names

or
no parameters, in this case object containing all attributes will be returned

Possible parameters

Please refer to the SVG specification (http://www.w3.org/TR/SVG/) for an explanation of these parameters.

- clip-rect *string* comma or space separated values: x, y, width and height
- cursor *string* CSS type of the cursor
- cx *number*
- cy *number*
- fill *colour* or *gradient*
    - linear gradient: "‹angle›-‹colour›[-‹colour›[:‹offset›]]*-‹colour›", example: "`90-#fff-#000`" – 90° gradient from white to black or "`0-#fff-#f00:20-#000`" – 0° gradient from white via red (at 20%) to black
    - radial gradient: "r[(‹fx›, ‹fy›)]‹colour›[-‹colour›[:‹offset›]]*-‹colour›", example: "`r#fff-#000`" – gradient from white to black or "`r(0.25, 0.75)#fff-#000`" – gradient from white to black with focus point at 0.25, 0.75
    - Focus point coordinates are in 0..1 range
    - Radial gradients can only be applied to circles and ellipses
- fill-opacity *number*
- font *string*
- font-family *string*
- font-size *number*
- font-weight *string*
- height *number*
- href *string* URL, if specified element behaves as hyperlink
- opacity *number*
- path *pathString* SVG path string format (http://www.w3.org/TR/SVG/paths.html#PathData)
- r *number*
- rotation *number*
- rx *number*
- ry *number*
- scale *string* comma or space separated values: xtimes, ytimes, cx, cy. See: scale (#scale)
- src *string* (URL)
- stroke *colour*
- stroke-dasharray *string* ["", "`-`", "`.`", "`-.`", "`-..`", "`. `", "`- `", "`--`", "`- .`", "`--.`", "`--..`"]
- stroke-linecap *string* ["`butt`", "`square`", "`round`"]
- stroke-linejoin *string* ["`bevel`", "`round`", "`miter`"]
- stroke-miterlimit *number*
- stroke-opacity *number*
- stroke-width *number*
- target *string* used with href (#attr-href)
- text-anchor *string* ["`start`", "`middle`", "`end`"], default is "middle"
- title *string* will create tooltip with a given text
- translation *string* comma or space separated values: x and y
- width *number*
- x *number*
- y *number*

Usage

```
var c = paper.circle(10, 10, 10);
// using strings
c.attr("fill", "black");
// using params object
c.attr({fill: "#000", stroke: "#f00", opacity: 0.5});
c.attr({
    fill: "90-#fff-#000",
    "stroke-dasharray": "-..",
    "clip-rect": "10, 10, 100, 100"
});
```

animate

Changes an attribute from its current value to its specified value in the given amount of milliseconds.

Parameters

1. newAttrs *object* A parameters object of the animation results. (Not all attributes can be animated.)
2. ms *number* The duration of the animation, given in milliseconds.
3. callback *function* [optional]

or

1. newAttrs *object* A parameters object of the animation results. (Not all attributes can be animated.)
2. ms *number* The duration of the animation, given in milliseconds.
3. easing *string* [“>”, “<”, “<>”, “`backIn`”, “`backOut`”, “`bounce`”, “`elastic`”, “`cubic-bezier(p1, p2, p3, p4)`”] or *function* [optional], see explanation re cubic-bezier syntax (http://www.w3.org/TR/css3-transitions/#transition-timing-function_tag)
4. callback *function* [optional]

or

1. keyFrames *object* Key-value map, where key represents keyframe timing: [“from”, “20%”, “to”, “35%”, etc] and value is the same as `newAttrs` from above, except it could also have `easing` and `callback` properties
2. ms *number* The duration of the animation, given in milliseconds.

Look at the example of keyframes usage (bounce.html)

Attributes that can be animated

The `newAttrs` parameter accepts an object whose properties are the attributes to animate. However, not all attributes listed in the `attr` method reference can be animated. The following is a list of those properties that can be animated:

- clip-rect *string*
- cx *number*
- cy *number*
- fill *colour*
- fill-opacity *number*
- font-size *number*
- height *number*
- opacity *number*
- path *pathString*
- r *number*
- rotation *string*
- rx *number*
- ry *number*
- scale *string*
- stroke *colour*
- stroke-opacity *number*
- stroke-width *number*
- translation *string*
- width *number*
- x *number*
- y *number*

Easing

For easing use built in functions or add your own by adding new functions to `Raphael.easing_formulas` object. Look at the example of easing usage (easing.html).

Usage

```
var c = paper.circle(10, 10, 10);
c.animate({cx: 20, r: 20}, 2000);
c.animate({cx: 20, r: 20}, 2000, "bounce");
c.animate({
    "20%": {cx: 20, r: 20, easing: ">"},
    "50%": {cx: 70, r: 120, callback: function () {…}},
    "100%": {cx: 10, r: 10}
}, 2000);
```

Stop

Stops current animation.

Usage

```
var c = paper.circle(10, 10, 10);
c.animate({cx: 20, r: 20}, 2000);
// stop animation half way
setTimeout(function () { c.stop(); }, 1000);
```

animateWith

The same as `animate` (#animate) method, but synchronise animation with another element.

Parameters

The same as for `animate` (#animate) method, but first argument is an element.

Usage

```
var c = paper.circle(10, 10, 10),
    r = paper.rect(10, 10, 10, 10);
c.animate({cx: 20, r: 20}, 2000);
r.animateWith(c, {x: 20}, 2000);
```

animateAlong

Animates element along the given path. As an option it could rotate element along the path.

Parameters

1. path *object* or *string* path element or path string along which the element will be animated
2. ms *number* The duration of the animation, given in milliseconds.
3. rotate *boolean* [optional] if true, element will be rotated along the path
4. callback *function* [optional]

Usage

```
var p = r.path("M100,100c0,50 100-50 100,0c0,50 -100-50 -100,0z").attr({stroke: "#ddd"}),
    e = r.ellipse(104, 100, 4, 4).attr({stroke: "none", fill: "#f00"}),
    b = r.rect(0, 0, 620, 400).attr({stroke: "none", fill: "#000", opacity: 0}).click(function () {
        e.attr({rx: 5, ry: 3}).animateAlong(p, 4000, true, function () {
            e.attr({rx: 4, ry: 4});
        });
    });
```

animateAlongBack

Similar to `animateAlong` (animateAlong). Animates element along the given path, starting from the end of it.

onAnimation

Sets or resets the function that will be called on each stage of the animation.

Parameters

1. f *function* function that will be called on each stage of animation

Usage

```
var p = r.path("M10,50c0,50,80-50,80,0c0,50-80-50-80,0z"),
    p2 = r.path(),
    e = r.ellipse(10, 50, 4, 4).attr({stroke: "none", fill: "#f00"}).onAnimation(function () {
        p2.attr({path: "M50,10L" + e.attr("cx") + "," + e.attr("cy")});
    }),
    b = r.rect(0, 0, 620, 400).attr({stroke: "none", fill: "#000", opacity: 0}).click(function () {
        e.attr({rx: 5, ry: 3}).animateAlong(p, 4000, true, function () {
            e.attr({rx: 4, ry: 4});
        });
    });
```

getBBox

Returns the dimensions of an element.

Usage

```
var c = paper.circle(10, 10, 10);
var width = c.getBBox().width;
```

toFront

Moves the element so it is the closest to the viewer's eyes, on top of other elements.

Usage

```
var c = paper.circle(10, 10, 10);
c.toFront();
```

toBack

Moves the element so it is the furthest from the viewer's eyes, behind other elements.

Usage

```
var c = paper.circle(10, 10, 10);
c.toBack();
```

insertBefore

Inserts current object before the given one.

Usage

```
var r = paper.rect(10, 10, 10, 10);
var c = paper.circle(10, 10, 10);
c.insertBefore(r);
```

insertAfter

Inserts current object after the given one.

Usage

```
var r = paper.rect(10, 10, 10, 10);
var c = paper.circle(10, 10, 10);
r.insertAfter(c);
```

clone

Returns a clone of the current element.

Usage

```
var r = paper.rect(10, 10, 10, 10);
var c = r.clone();
```

Graphic Primitives

Methods of “paper” object, created with `Raphael` function call.

raphael

Internal reference to `Raphael` object. In case it is not available.

Usage

```
Raphael.el.red = function () {
    var hsb = this.paper.raphael.rgb2hsb(this.attr("fill"));
    hsb.h = 1;
    this.attr({fill: this.paper.raphael.hsb2rgb(hsb).hex});
}
```

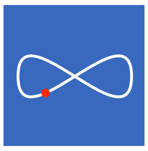
getTotalLength

Path specific method. Returns length of the path in pixels.

Usage

```
var p = r.path("M100,100c0,50 100-50 100,0c0,50 -100-50 -100,0z");
    alert(p.getTotalLength());
```

getPointAtLength



Path specific method. Returns point description at given length.

Parameters

1. length *number* length in pixels from the begining of the path to the point

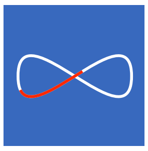
Usage

```
var p = r.path("M10,50c0,50,80-50,80,0c0,50-80-50-80,0z");
    var point = p.getPointAtLength(30);
    r.circle(point.x, point.y, 3);
```

Returned object format:

- x – x coordinate of the point
- y – y coordinate of the point
- alpha – angle of the path at the point

getSubpath



Path specific method. Returns the subpath string of a given path.

Parameters

1. from *number* length in pixels from the beginning of the path to the beginning of the subpath
2. to *number* length in pixels from the beginning of the path to the end of the subpath

Usage

```
var p = r.path("M10,50c0,50,80-50,80,0c0,50-80-50-80,0z");
    var path = p.getSubpath(10, 60);
    r.path(path).attr({stroke: "#f00"});
```

setSize

If you need to change dimensions of the canvas call this method

Parameters

1. width *number* new width of the canvas
2. height *number* new height of the canvas

setWindow

Should be called before main Raphael method. Sets which window should be used for drawing. Default is the current one. You need to use it if you want to draw inside `iframe`

Parameters

1. window *object*

getRGB

Parses passes string and returns an colour object. Especially useful for plug-in developers.

Parameters

1. color *string* Colour in form acceptable by library

Usage

```
var stroke = Raphael.getRGB(circle.attr("stroke")).hex;
```

angle

Gives you an angle of the line formed by two points or angle between two lines formed by three points.

Parameters

1. x1 *number* X of the first point
2. y1 *number* Y of the first point
3. x2 *number* X of the second point
4. y2 *number* Y of the second point
5. x3 *number* X of the third point [optional]
6. y3 *number* Y of the third point [optional]

Usage

```
var angle = Raphael.angle(10, 10, 50, 50);
```

rad

Converts angle to radians.

Parameters

1. deg *number* value of an angle in degrees

Usage

```
var Sin = Math.sin(Raphael.rad(45));
```

deg

Converts angle to degree.

Parameters

1. rad *number* value of an angle in radians

snapTo

Returns a number adjusted to one of various values, if it’s close enough.

Parameters

1. values *number or array* If values is a number, the value will be snapped to any multiple. If an array, the value will be snapped to the first element that’s within tolerance.
2. value *number* The value to adjust
3. tolerance *number* The value must be within tolerance of one of the snap values, or it will be returned unchanged [optional, default 10]

Usage

```
// adjust -10..10 to 0, 40..60 to 50, 90-110 to 100, etc
x = Raphael.snapTo(50, x);
// adjust 5..35 to 20, 45..75 to 60, otherwise no change
x = Raphael.snapTo([20, 60], x, 15);
```

getColor

Returns a colour object for the next colour in spectrum

Parameters

1. value *number* brightness [optional]

Usage

```
var c = paper.path("M10,10L100,100").attr({stroke: Raphael.getColor()});
```

getColor.reset

Resets getColor function, so it will start from the beginning

registerFont

Adds given font to the registered set of fonts for Raphaël. Should be used as an internal call from within Cufón's font file. More about Cufón and how to convert your font form TTF, OTF, etc to JavaScript file (http://wiki.github.com/sorccu/cufon/about). Returns original parameter, so it could be used with chaining.

Parameters

1. font *object* the font to register

Usage

```
Cufon.registerFont(Raphael.registerFont({…}))
```

getFont

Finds font object in the registered fonts by given parameters. You could specify only one word from the font name, like "Myriad" for "Myriad Pro".

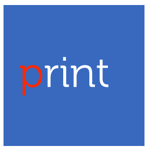
Parameters

1. family *string* font family name or any word from it
2. weight *string* weight of the font [optional]
3. style *string* style of the font [optional]
4. stretch *string* stretch of the font [optional]

Usage

```
paper.print(100, 100, "Test string", paper.getFont("Times", 800), 30);
```

print



Creates set of shapes to represent given font at given position with given size. Result of the method is set object (see set (#set)) which contains each letter as separate path object.

Parameters

1. x *number* x position of the text
2. y *number* y position of the text
3. text *string* text to print
4. font *object* font object (see getFont (#getFont))
5. font_size *number* size of the font

Usage

```
var txt = r.print(10, 50, "print", r.getFont("Museo"), 30).attr({fill: "#fff"});
// following line will paint first letter in red
txt[0].attr({fill: "#f00"});
```

Adding your own methods to canvas

You can add your own method to the canvas. For example if you want to draw pie chart, you can create your own pie chart function and ship it as a Raphaël plugin. To do this you need to extend Raphael.fn object. Please note that you can create your own namespaces inside fn object. Methods will be run in context of canvas anyway. You should alter fn object before Raphaël instance was created, otherwise it will take no effect.

Usage

```
Raphael.fn.arrow = function (x1, y1, x2, y2, size) {
    return this.path( ... );
};
// or create namespace
Raphael.fn.mystuff = {
    arrow: function () {…},
    star: function () {…},
    // etc…
};
var paper = Raphael(10, 10, 630, 480);
// then use it
paper.arrow(10, 10, 30, 30, 5).attr({fill: "#f00"});
paper.mystuff.arrow();
paper.mystuff.star();
```

Adding your own methods to elements

You can add your own method to elements. This is usefull when you want to hack default functionality or want to wrap some common transformation or attributes in one method. In difference to canvas mathods, you can redefine element method at any time.

Usage

```
Raphael.el.red = function () {
    this.attr({fill: "#f00"});
};
// then use it
paper.circle(100, 100, 20).red();
```

Custom Attributes

If you have a set of attributes that you would like to represent as a function of some number you can do it easily with custom attributes:

Usage

```
paper.customAttributes.hue = function (num) {
    num = num % 1;
    return {fill: "hsb(" + num + ", .75, 1)"};
};
// Custom attribute “hue” will change fill
// to be given hue with fixed saturation and brightness.
// Now you can use it like this:
var c = paper.circle(10, 10, 10).attr({hue: .45});
// or even like this:
c.animate({hue: 1}, 1e3);

// You could also create custom attribute
// with multiple parameters:
paper.customAttributes.hsb = function (h, s, b) {
    return {fill: "hsb(" + [h, s, b].join(",") + ")"};
};
c.attr({hsb: ".5 .8 1"});
c.animate({hsb: "1 0 .5"}, 1e3);
```

Supported colour formats

You could specify colour in this formats:

- Colour name (“`red`”, “`green`”, “`cornflowerblue`”, etc)
- #••• — shortened HTML colour: (“`#000`”, “`#fc0`”, etc)
- #•••••• — full length HTML colour: (“`#000000`”, “`#bd2300`”)
- rgb(•••, •••, •••) — red, green and blue channels’ values: (“`rgb(200, 100, 0)`”)
- rgb(•••%, •••%, •••%) — same as above, but in %: (“`rgb(100%, 175%, 0%)`”)
- rgba(•••, •••, •••, •••) — red, green and blue channels’ values: (“`rgba(200, 100, 0, .5)`”)
- rgba(•••%, •••%, •••%, •••%) — same as above, but in %: (“`rgba(100%, 175%, 0%, 50%)`”)
- hsb(•••, •••, •••) — hue, saturation and brightness values: (“`hsb(0.5, 0.25, 1)`”)
- hsb(•••%, •••%, •••%) — same as above, but in %
- hsba(•••, •••, •••, •••) — same as above, but with opacity
- hsl(•••, •••, •••) — almost the same as hsb, see Wikipedia page (http://en.wikipedia.org/wiki/HSL_and_HSV)
- hsl(•••%, •••%, •••%) — same as above, but in %
- hsla(•••, •••, •••) — same as above, but with opacity
- Optionally for hsb and hsl you could specify hue as a degree: “`hsl(240deg, 1, .5)`” or, if you want to go fancy, “`hsl(240°, 1, .5)`”

Usage

```
paper.circle(100, 100, 20).attr({
    fill: "hsb(0.6, 1, 0.75)",
    stroke: "red"
});
```

safari

There is an inconvenient rendering bug is Safari (WebKit): sometimes the rendering should be forced. This method should help with dealing with this bug.

Usage

```
paper.safari();
```

“Ninja Mode”

If you want to leave no trace of Raphaël (Well, Raphaël creates only one global variable `Raphael`, but anyway.) You can use `ninja` method. Beware, that in this case plugins could stop working, because they are depending on global variable existance.

Usage

```
(function (local_raphael) {
    var paper = local_raphael(10, 10, 320, 200);
    …
})(Raphael.ninja());
```

Events

You can attach events to elements by using element.node and your favourite library (`$(circle.node).click(…)`) or you can use built-in methods:

Usage

```
element.click(function (event) {
    this.attr({fill: "red"});
});
element.dblclick(function (event) {
    this.attr({fill: "red"});
});
element.mousedown(function (event) {
    this.attr({fill: "red"});
});
element.mousemove(function (event) {
    this.attr({fill: "red"});
});
element.mouseout(function (event) {
    this.attr({fill: "red"});
});
element.mouseover(function (event) {
    this.attr({fill: "red"});
});
element.mouseup(function (event) {
    this.attr({fill: "red"});
});
element.hover(function (event) {
    this.attr({fill: "red"});
}, function (event) {
    this.attr({fill: "black"});
}, overScope, outScope);
```

Second parameter is optional scope. By default handlers are run in the scope of the element. To unbind events use the same method names with "un" prefix, i.e. `element.unclick(f);`

Drag 'n' Drop

To make element "draggable" you need to call method `drag` on element.

Parameters

1. onmove *function* event handler for moving
2. onstart *function* event handler for start
3. onend *function* event handler for end of the drag

Usage

```
var c = R.circle(100, 100, 50).attr({
    fill: "hsb(.8, 1, 1)",
    stroke: "none",
    opacity: .5
});
var start = function () {
    // storing original coordinates
    this.ox = this.attr("cx");
    this.oy = this.attr("cy");
    this.attr({opacity: 1});
},
move = function (dx, dy) {
    // move will be called with dx and dy
    this.attr({cx: this.ox + dx, cy: this.oy + dy});
},
up = function () {
    // restoring state
    this.attr({opacity: .5});
};
c.drag(move, start, up);
```

To unbind drag use the `undrag` method.