# Drawing with HTML5

In this chapter, we will learn about two things:

* Lesson 1: Drawing by using the <canvas> element
* Lesson 2: Using scalable vector graphics

## Lesson 1: Drawing by using the <canvas> element

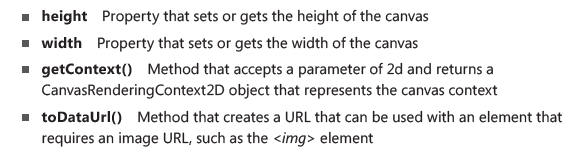
Below is an example of using the <canvas> element. Note that the content within the <canvas> tag is only shown if the browser does not support the <canvas> element.

e.g.  


**JsFiddle**: https://jsfiddle.net/qevzyoLL/

### The <canvas> element reference

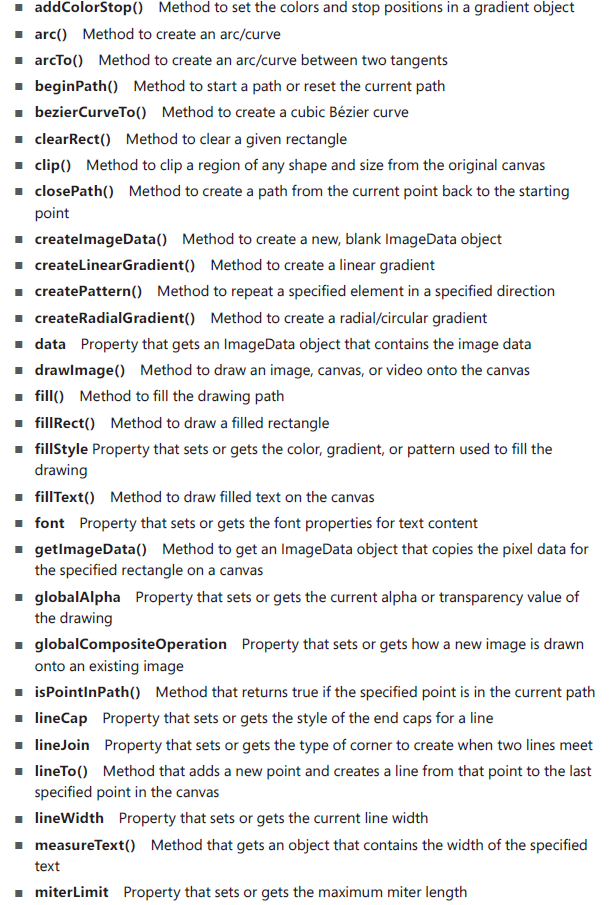
The <canvas> element has the following attributes:

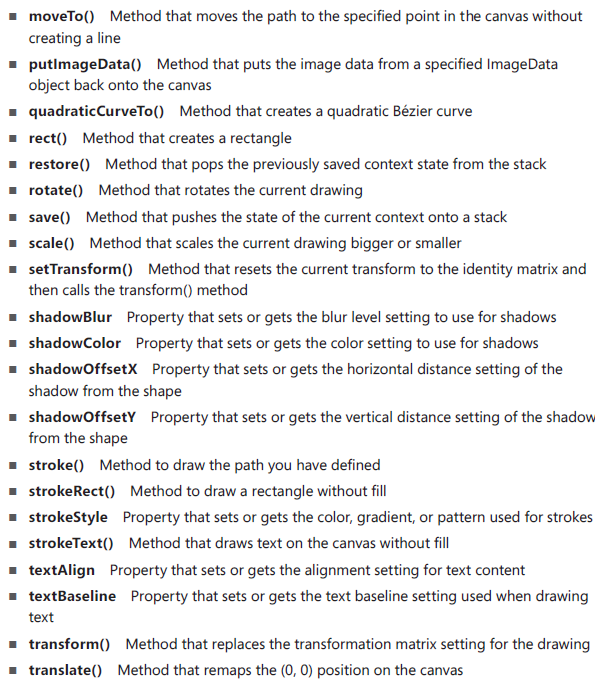


### CanvasRenderingContext2D context object reference

The <canvas> element is simply a graphics container; the **context object** that is returned from the *getContext* method is **what we** **actually use to draw on the canvas**.

The following is a list of the context object's members:

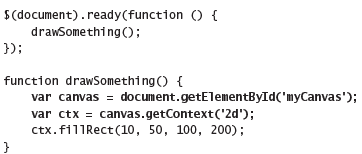




### Implementing the canvas

To work with the canvas object, call the getContext method and use the canvas context returned.

The getContext method **takes one parameter, which is '2d' in all browsers**, and **returns a CanvasRenderingContext2D** object for making **2-dimensional drawings**. (However in Firefox and Chrome, the getContext method accepts an ' experimental-webgl' parameter, which supports three-dimensional drawings)

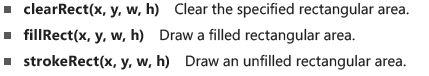
e.g.   


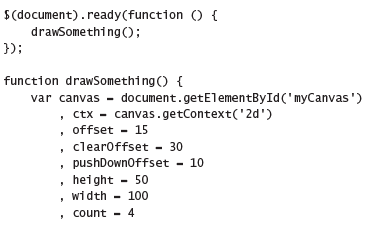
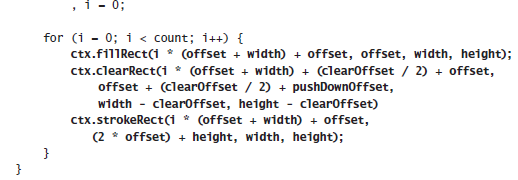
**JsFiddle**: https://jsfiddle.net/qevzyoLL/1/

### Drawing rectangles

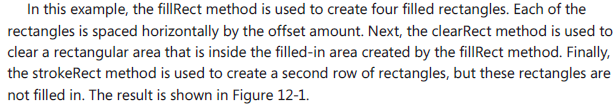
The method for creating rectangles accepts four parameters

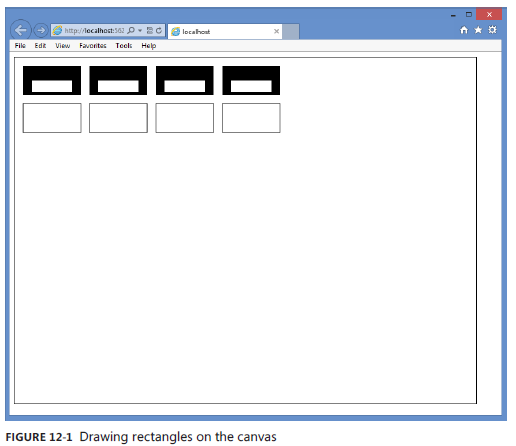
* First two parameters are x and y locations of the upper-left corner of the rectangle
* The last two parameters are the width and height of the rectangle

We can create rectangles using any of the following methods:  


e.g.   
  


**Code explanation:**



**Code Result:  
**

**JsFiddle:** https://jsfiddle.net/qevzyoLL/2/

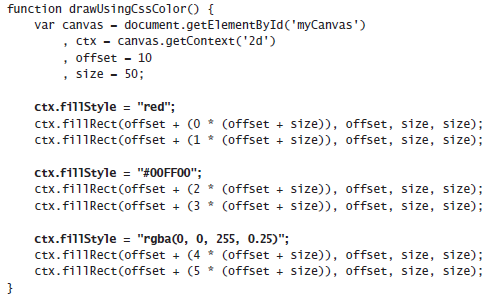
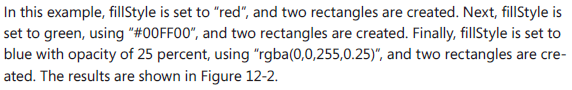
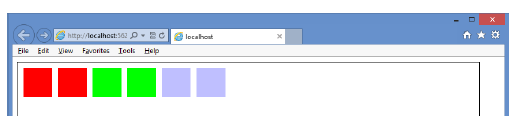
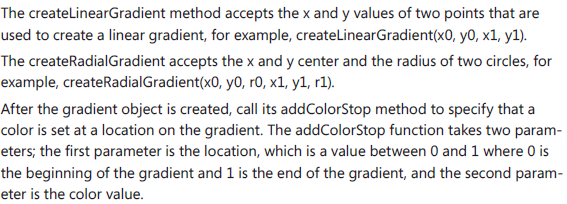
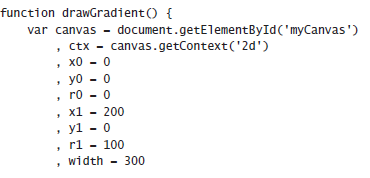
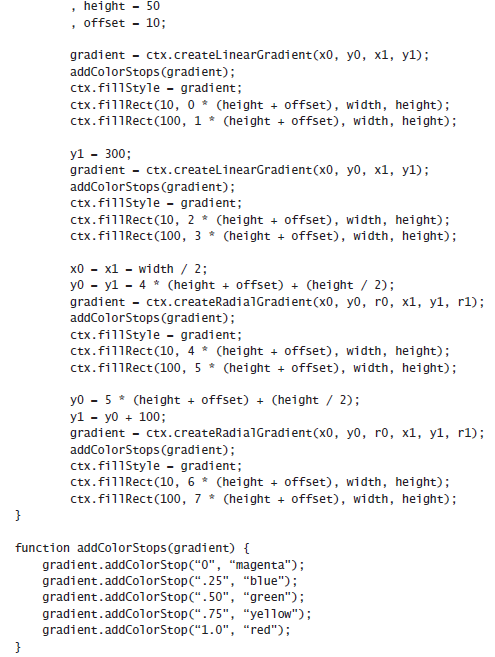
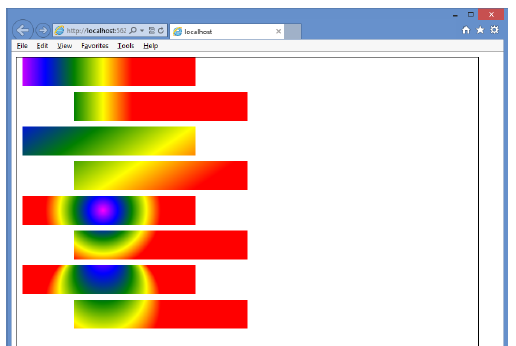
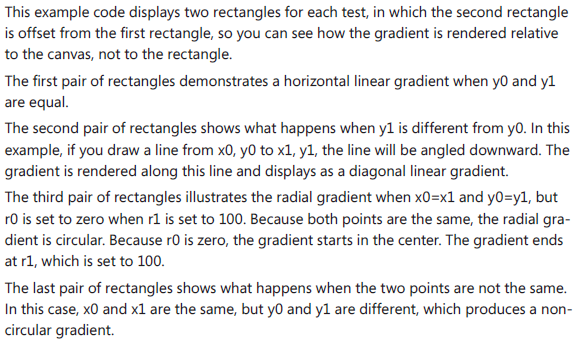
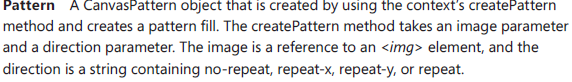
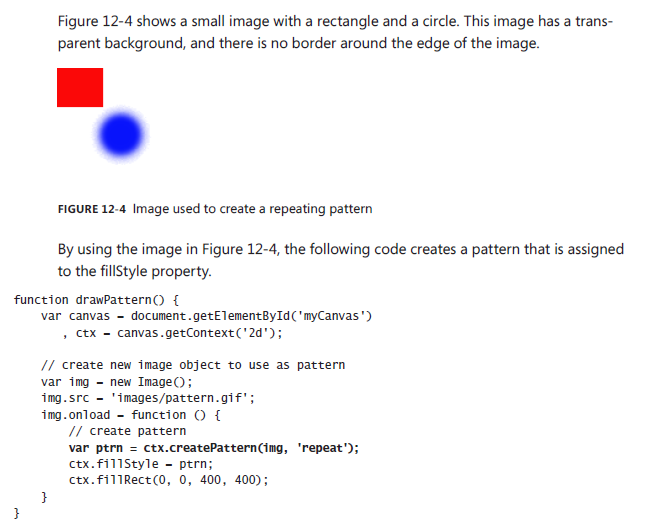
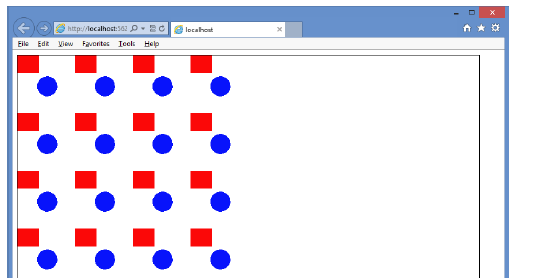
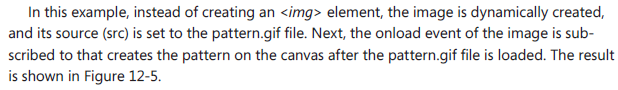
### Configuring the drawing state

The canvas context has properties that you can set before you call any of the drawing methods. After you change a property, the new value is used for subsequent drawing statements.

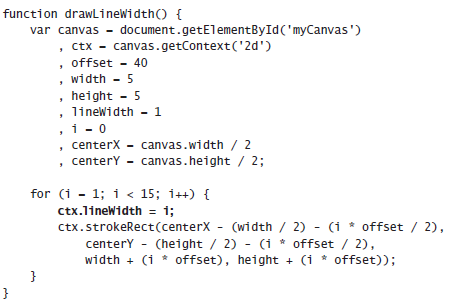
#### Setting fillStyle

Use the fillStyle property to specify how you want to fill shapes.

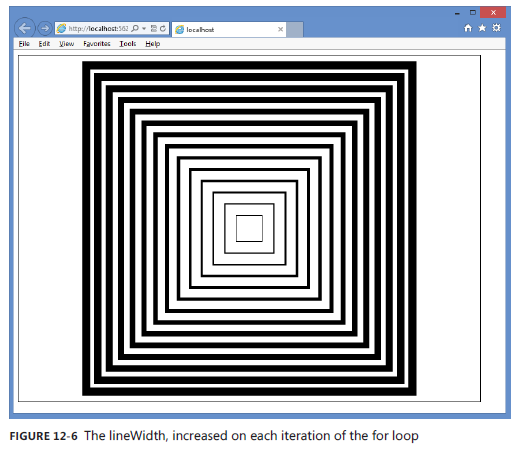
We can set the fillStyle to any of the following values:

* fillStyle **CSS color** property**.**   
  e.g.  
    
  **Code explanation:** **Code Result:**  
   **JsFiddle:** https://jsfiddle.net/qevzyoLL/3/
* fillStyle **Gradient** property**.**A gradient is a CanvasGradient object that is created by the context’s createLinearGradient or createRadialGradient method to create a gradient fill.  
  ****  
  e.g.  
  ****  
  ****  
    
  **Code Result**:   
  **  
    
  Code Explanation:  
  **
* fillStyle **Pattern** property  
  ****  
  e.g.  
  ****  
  **Code Result:  
    
    
  Code explanation:  
  **

### Setting lineWidth

The lineWidth property specifies the thickness of any line you draw.  
e.g.  
****

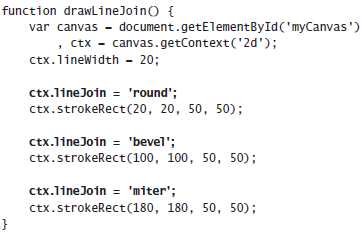
**Code Explanation:  
  
  
Code Result:**

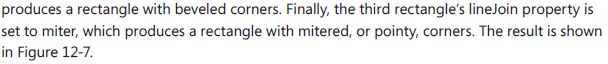
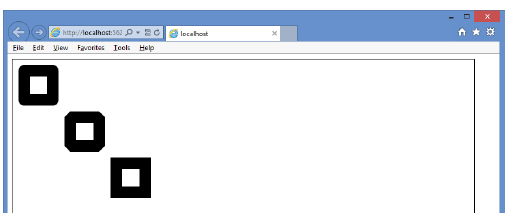
**  
JsFiddle:** https://jsfiddle.net/qevzyoLL/4/

### Setting lineJoin

The lineJoin property specifies the way lines that join each other are drawn.

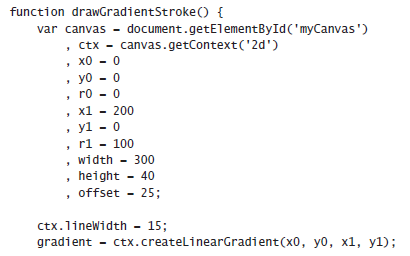
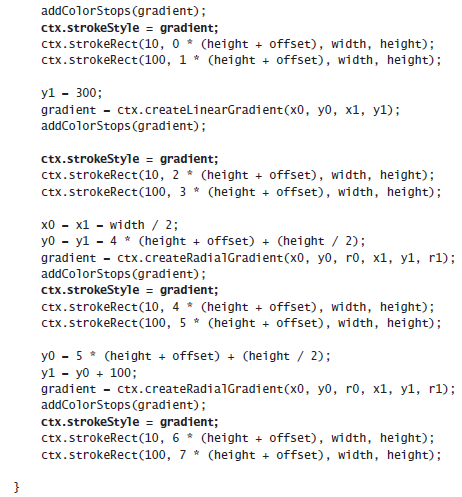
You can set the lineJoin property to round, bevel, or miter. The default value is miter

e.g.  


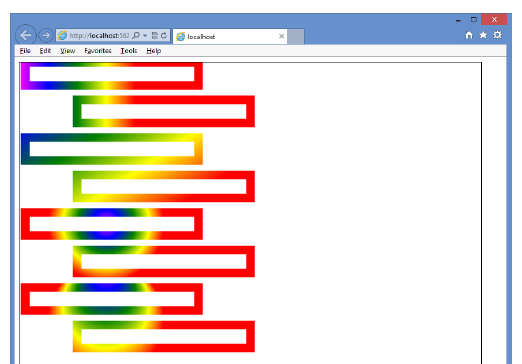
**Code explanation:  
  
  
  
Code result:  
**

### Setting strokeStyle

The strokeStyle property specifie the way you want to draw lines.

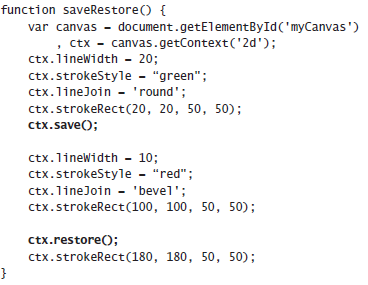
e.g.  
  


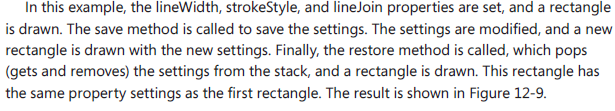
**Code Explanation**: Reference the fillStyle drawGradient function for an explanation (exactly the same concept)

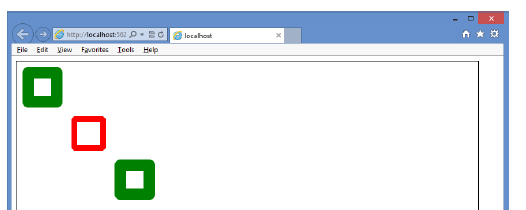
**Code Result:**  


### Saving and restoring the drawing state

It is possible to save all the context object properties to a stack, which is a last-in, first-out (LIFO) collection. This does not save the actual canvas; just the settings are saved. The save method saves the current settings, and the restore method restores the settings.

e.g.  


**Code Explanation:**  


**Code Result:**

**JsFiddle:** https://jsfiddle.net/h33fu3u6/

### Drawing by using paths

A path is a set of lines that are used to draw shapes.

Steps for creating a shape using paths:

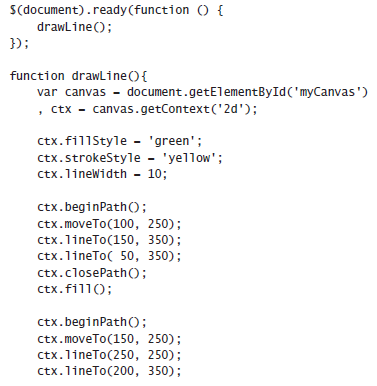
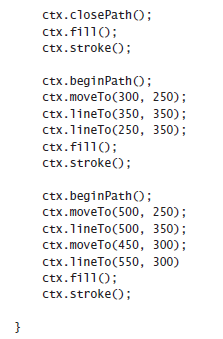
* Start a path by calling the beginPath method.
* Go to the starting position by calling the moveTo method.
* Draw sub-paths calling methods such as lineTo and rect.
* End the path by optionally calling the closePath method.
* Render the filled shape or outlined shape by calling fill or stroke methods, respectively.

Using the HTML & CSS code below, we will be drawing a bunch of shapes

HTML:  


CSS:  


#### Drawing Lines

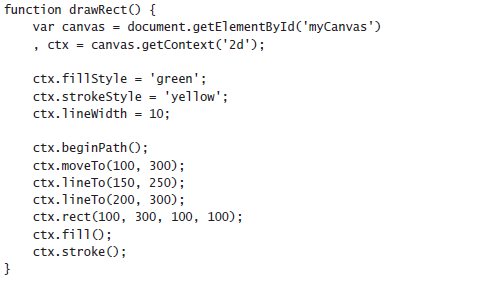
Javascript:  
  


**Code Result:**   


#### Drawing rectangles

**While drawing complex shapes**, if you need to draw a rectangle on the path, use the *rect()* method instead of using multiple *lineTo* method calls to define a rectangle (use fillRect or strokeRect to draw lone rectangles).

e.g.

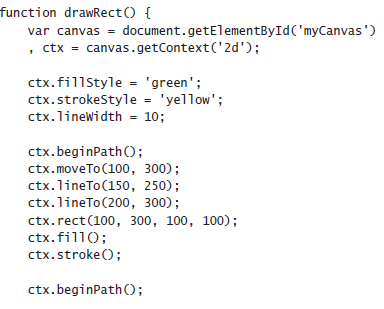
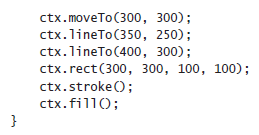
  
**JSFiddle**: https://jsfiddle.net/h33fu3u6/1/

#### Ordering the fill and the stroke method calls

There is a difference in calling the fill method before stroke & vice versa.

When stroke is called, the outline is created using the lineWidth property. Half of lineWidth is outside the shape, and half is inside the shape.

**Key**: When the stroke method is called after the fill method, the part of the outline that is inside the shape overwrites the fill. When stroke is called before fill however, fill overwrites the stroke that is inside the shape.

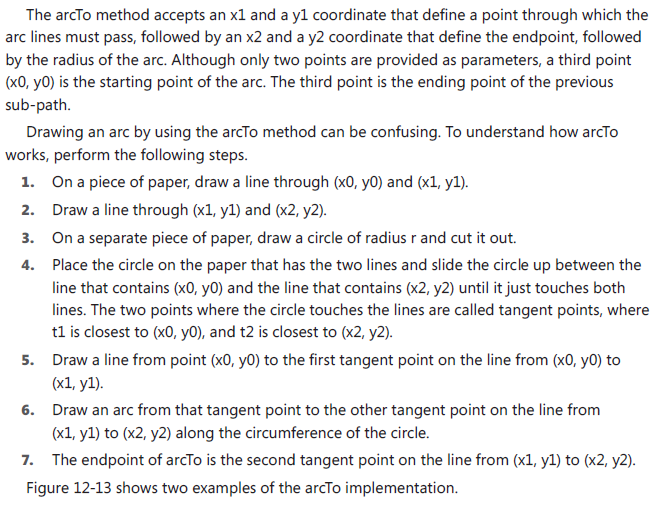
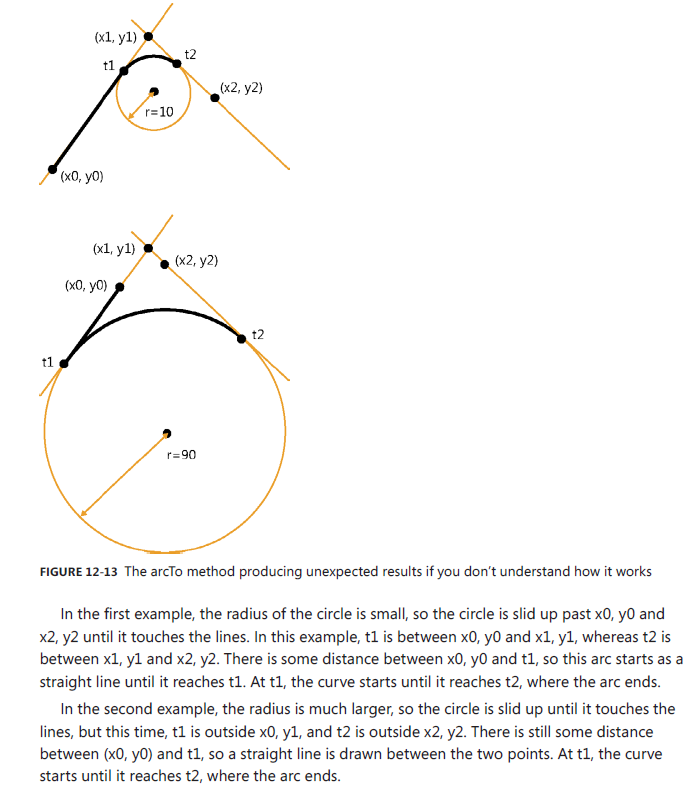
e.g.  
  


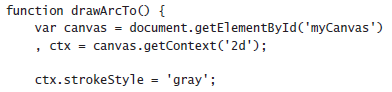
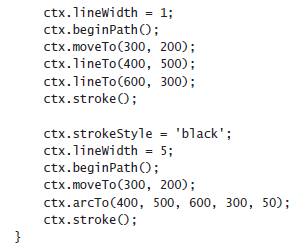
**JsFiddle**: https://jsfiddle.net/h33fu3u6/2/

#### Drawing arcs by using the arcTo method

You can draw curved lines by using the arc and the arcTo methods on the context object

**Understanding the arcTo method:**

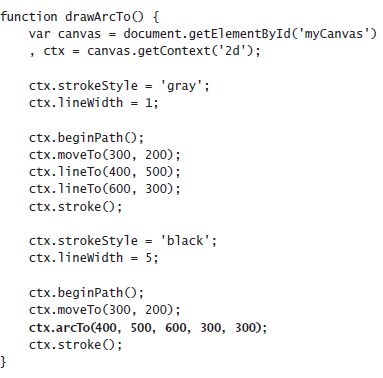
  


**Example 1**:  
  


**JsFiddle**: https://jsfiddle.net/h33fu3u6/3/

**NOTE**: Because the starting point is always the ending point of the previous sub-path (or, in this case, the moveTo location), the stroke continues from that point until it reaches the first tangent, and then the curve starts until it reaches the second tangent, and then the arc is finished.

**Example 2**: In this code example, the points stay the same, but the radius is changed from 50 to 300



**JsFiddle**: https://jsfiddle.net/h33fu3u6/4/

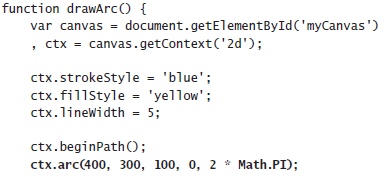
**Code Explanation**: The radius of the circle has a profound impact on the rendered arc. In addition, because the circle is slid into the angle that’s formed by the line x0, y0 is on and the line x2, y2 is on, you can’t possibly create an arc that is greater than 180 degrees because the circle will always touch these lines to create tangents t1 and t2 before the arc reaches 180 degrees.

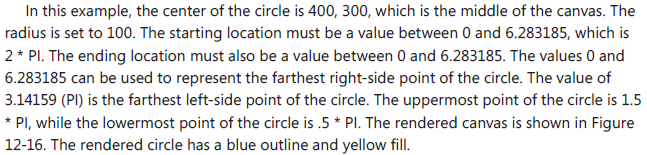
#### Drawing arcs by using the arc method

The arc method is much simpler to use than the arcTo method, as it can be used to draw a circle or any part of a circle.

The arc method accepts x and y coordinates as the center of the circle used to draw the arc, followed by the radius of the circle that the arc will use, followed by the starting angle and the ending angle. You can add a direction parameter that indicates the direction of the arc.

e.g.


**Explanation**:  
   
 **JsFiddle**: https://jsfiddle.net/h33fu3u6/5/

**To draw an arc that represents part of the circle**

We need a valid start and end location, and we need to consider the optional direction parameter.

By default the stroke is rendered clockwise (direction is set to false).

Example 1:



**JsFiddle**: https://jsfiddle.net/h33fu3u6/6/

**Scenario - What if we want to render only the circle portion missing from 'Example 1'**:

We have 2 choices:

1. Start at 1.5 \* Math.PI and end at 0

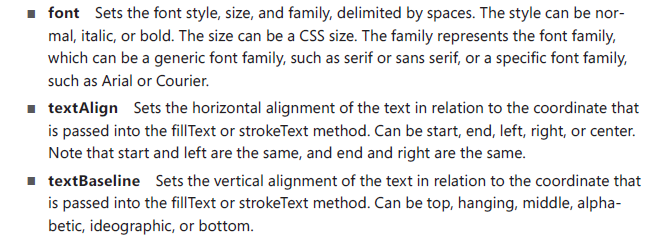
2. Leave the start at 0 and the end at 1.5 \* Math.PI, but add the optional direction parameter, passing in the value of true to change the rendering direction to counter-clockwise.

Drawing text  
We can also draw text by using the fillText or strokeText methods.

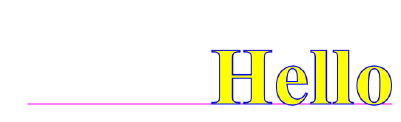
Both methods require the following parameters:

* Param 1: the text to be drawn
* Param 2: x and y coordinates that specify where the text is drawn (The exact meaning of the x & y coordinates depend on the value of the textAlign and textBaseline properties. The default is that the coordinate is at the lower-left corner of the text that's drawn
* Optional param 3: a parameter to indicate the maximum width of the text to be drawn

Additionally, the following properties can be set to control the look of the rendered text:



**Example 1**

  
**JsFiddle**: https://jsfiddle.net/h33fu3u6/7/

**Alternative Example**

If you change the textAlign property to center and change the textBaseline to middle, the text will be centered horizontally and vertically within the canvas, as shown below



### Drawing with images

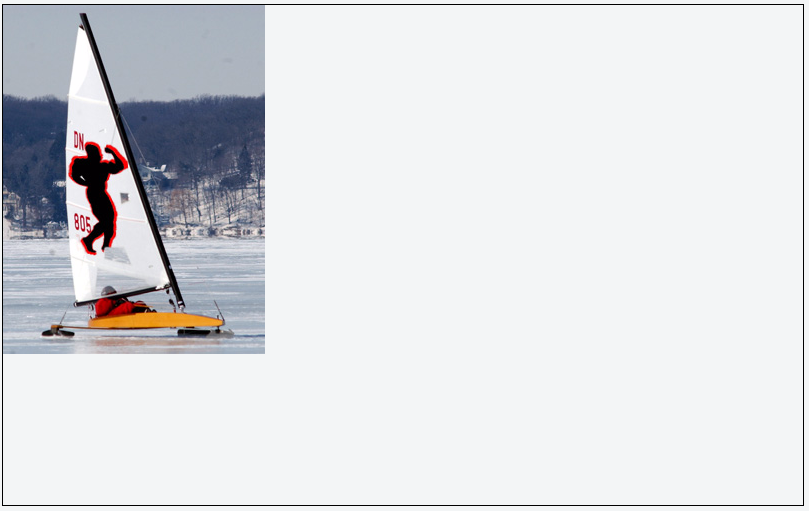
We might want to place images on the canvas sometimes; we do this by drawing the image on the canvas using the **drawImage** method

The drawImage methods accepts either 3 or 5 parameters.

The three-parameter drawImage method takes in the following parameters:

* The source of the image (an <img> or <video> element, or another <canvas> element). Note: When the <video> element is used, a snapshot of the video frame that is currently displayed is used as the image
* The second and third arguments are the x & y coordinates of the upper-corner of the image
* **Additional Note:** You can also create the <img> element in your javascript (in a case where the image is conditional)

**Example of using the three-parameter drawImage method:**

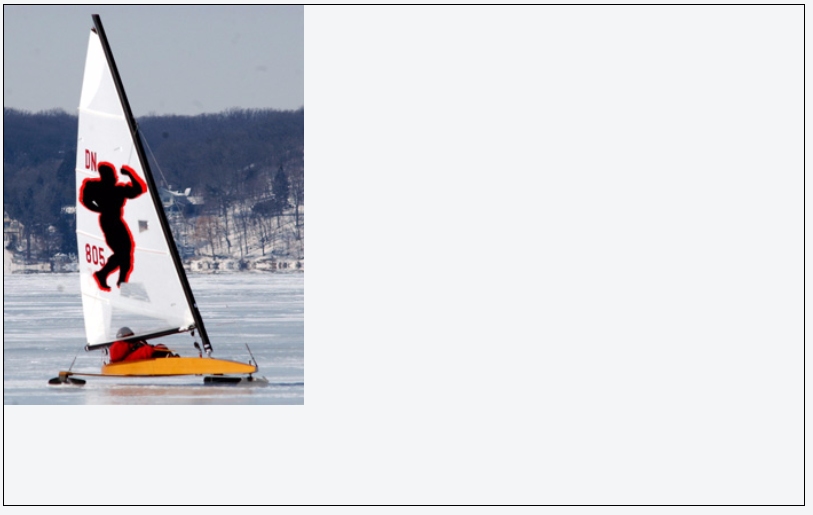


**JsFiddle**: https://jsfiddle.net/h33fu3u6/8/

**Five-parameter drawImage method:**

The five-parameter drawImage method takes in the same parameters as the 3-parameter drawImage method, with an additional width & height parameter.

**Example of using the five-parameter drawImage method:**



**JsFiddle**: https://jsfiddle.net/h33fu3u6/9/

**We can draw an image and overlay other shapes as necessary, e.g.**



**JsFiddle**: https://jsfiddle.net/h33fu3u6/10/

## Lesson 2: Using the scalable vector graphics

Scalable vector graphics (SVG), unlike the canvas element, stores all commands executed on it and hence the commands can be re-executed when the scale changes, to produce a clean, crisp image regardless of the scale.

**Performance Note**: Even though SVG renders much better than the canvas, it takes time to scale and re-execute the commands, so performance is not as good. The canvas is preferable when performance is more important.

The <*svg*> element, plus its content, is part of the document object model (DOM), so **all of the <*svg*> element and its children are accessible from JavaScript**. Events can also be attached to any of the elements.

**Recommendation**: you don’t want to write all the XML that’s required to create a complex SVG drawing. It’s typically best to use an SVG editor to create the drawing and then embed the drawing into your webpage

### Using the <svg> element

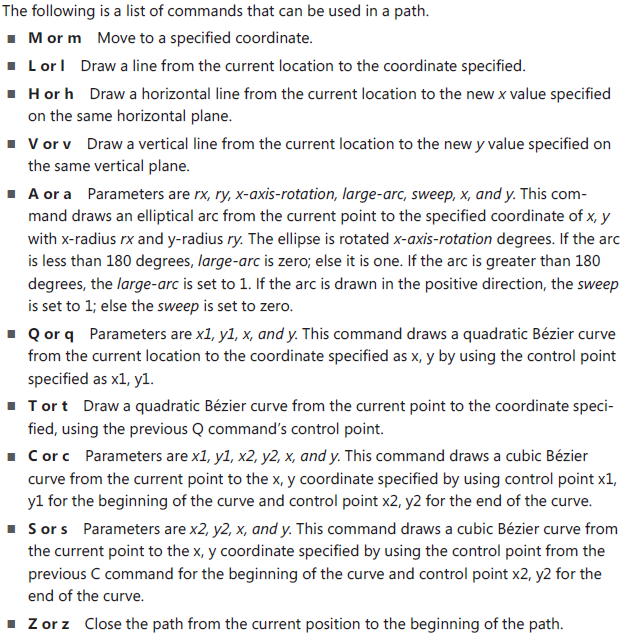
The <*svg*> element is a container for the XML-based commands.

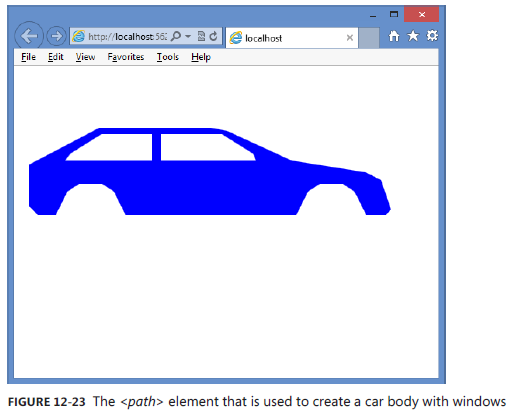
e.g. <svg width="500" height="300" xmlns="http://www.w3.org/2000/svg"> </svg>

#### Creating a path

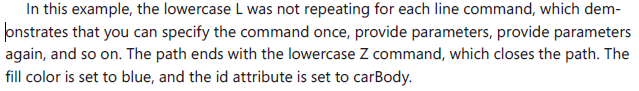
A path is a sequence of commands that create a complex shape.

Use the <*path*> element, which has id, fill, and d attributes, to create a path.   
The fill attribute is passed the, while the d attribute is for the <*path*> element’s data.



**Example 1**. The following example draws a car body by using a path that moves to 267, 76 and draws lines by using the l (lowercase L) command  


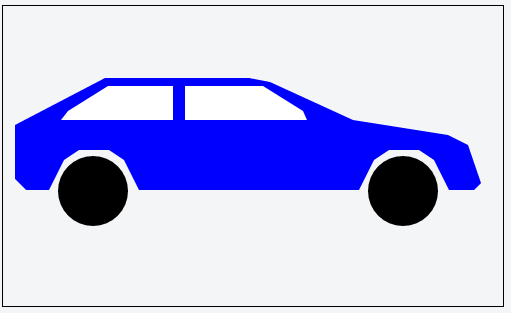
**Explanation**:

  
**JsFiddle**: https://jsfiddle.net/h33fu3u6/11/

#### Drawing circles

You can add circles to an SVG drawing by using the <*circle*> element, which has r, cx, cy, fill, and id attributes.

The r attribute sets the circle radius. The cx and cy attributes set the circle center coordinate. The fill attribute sets the color of the circle.

**Example 1**: Let us add two wheels to the vehicle body created in the previous example  


**JsFiddle**: https://jsfiddle.net/h33fu3u6/12/

### Displaying SVG files using the <img> element

We have seen how we can create SVG images by writing the XML yourself, but you can use any of the SVG editors, such as svg-edit, which is a browser-based application, available at *http://code.google.com/p/svg-edit*

When using an SVG editor, we typically want to externalize the SVG into its own file to make it easier to work on the drawing rather than embedding the drawing into our HTML page

We can display the SVG on our using the <img> tag

**Example**:  


**Note**: If you try to resize the page, the image won’t resize because the settings need to be changed

#### Making the SVG scalable

To make the SVG scalable, first change the height and width setting to 100 percent, then add the viewBox attribute to the <svg> element.

**About the ViewBox property**:

The viewBox attribute describes the part of the canvas you want the viewer to see. Even though the drawing covers the entire computer screen, the figure on your drawing might only exist in a small part of the drawing. The viewBox attribute enables you to tell the parser to zoom in on that part to eliminate the extra white space. Set viewBox to get the proper zoom capabilities when you resize your HTML page.

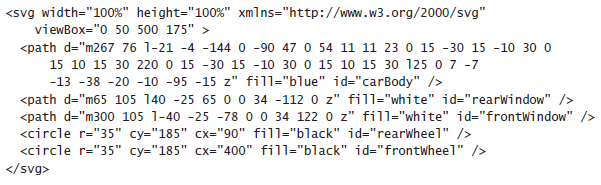
The viewBox has four parameters

* The minimum x coordinate
* The minimum y coordinate
* The width, and
* The height

The viewBox parameters enables us define the rectangular area to be displayed,

e.g. viewBox="0 50 500 175".  
This setting crops 50 pixels from the top and limits the viewing height to 175 pixels

After applying the changes mentioned above, the .svg file looks like the following:



In the svg.css file, the following style rule has been added to resize the <*img*> element automatically when the HTML page is resized.



**SVG file change result**:

