

IT614

HTML5 CANVAS

HTML5 <canvas> element lets us draw graphics using JavaScript.

The <canvas> element produces a rectangular drawing surface area on which graphics are drawn. Reference: http://www.w3.org/html/wg/drafts/2dcontext/html5_canvas_CR/

SYNTAX:

```
<canvas width="200" height="150"></canvas>
```

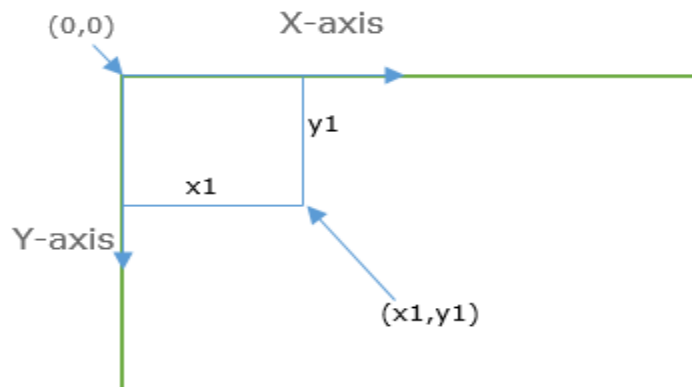
This creates a canvas of specified width and height.

Note:

- If you don't include width and height attributes, the canvas will have default width of 300 and height 150.
- In drawing graphics on a canvas, we need to access it. So we always include **id** attribute in <canvas> element.

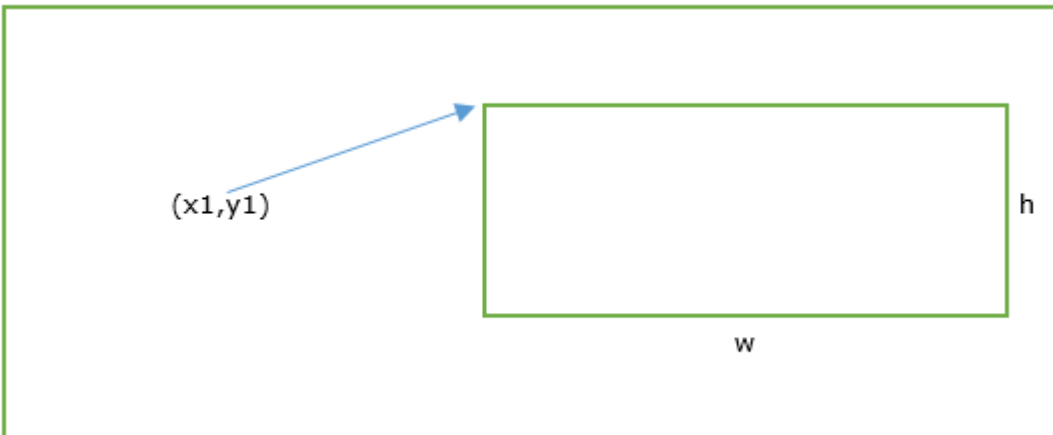
THE COORDINATE SYSTEM

Once you have a canvas, it determines a coordinate system. The top left corner of the canvas is the origin and its coordinates are (0,0). The horizontal line starting from (0,0) going to the right is the positive X axis and the vertical line starting from (0,0) going down is the positive Y axis. Any point on the canvas is represented by two numbers representing the x coordinate and the y coordinate of the point. These two numbers are in pixels giving the distances of the point from X and Y axes.



Note:

A rectangle on a canvas is typically specified using $(x1,y1,w,h)$. This specifies the rectangle with top left corner at $(x1,y1)$ and width w , and height h :



1. The rendering context

We use JavaScript to render graphics on canvas. The first step in doing this is to access "rendering context" for the canvas element. As mentioned earlier, we always include an **id** attribute `<canvas>` element and use the id value to get context using the following two lines of code:

```
var canvas = document.getElementById("idValue");
var context2D = canvas.getContext('2d');
```

The variable `context2D` is a 2-d context object and we use it to draw 2-d graphics on the canvas.

Before we explore methods for various shapes, let us do one example.

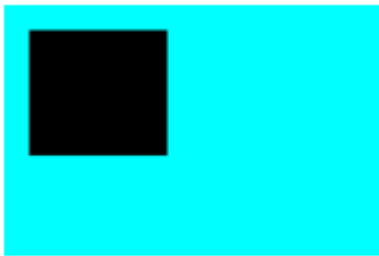
EXAMPLE:

In this example we use a canvas method, `context2D.fillRect(x1,y1,w,h)`, which paints the given rectangle onto the canvas, using the default color of black. We will see later how to change the default color. We will discuss `fillRect()` method in detail later.

```
<!DOCTYPE HTML>
<html> <head>
<style>
canvas{
```

```
background-color:cyan;
}
</style>
</head>
<body>
<canvas id="myCanvas" width="150" height="100"></canvas>
<script>
    var cv = document.getElementById("myCanvas");
    var context2D = cv.getContext('2d');
    context2D.fillRect (10, 10, 55, 50);
</script>
</body>
</html>
```

RESULT:



Note:

In all examples, we need these statements:

```
var cv = document.getElementById("myCanvas");
var context2D = cv.getContext('2d');
```

In practice, most of the canvas methods we will see are used with the object context2D.

There are several canvas methods available to draw various kinds of shapes, rotating shapes, moving and so on. Let us start exploring these methods.

2. Canvas paths

A HTML5 Canvas path is a segment consisting of several pieces shapes such as lines, arcs, and so on. Each sub segment is called a subpath. A closed subpath is one where the last point of the subpath is connected to the first point of the subpath by a straight line.

➤ The **beginPath()** method

The **beginPath()** starts a new path. Once you start a new path, you set properties to the path. Even if you don't start a new path, the context always has a current default path. You need to use **beginPath()** when you want a new path different properties. A new path started using **beginPath()** will not have any subpaths.

Here is a path property you can set:

➤ The **strokeStyle** property

The **strokeStyle** property specifies the color to use for the lines around the shapes.

Syntax:

```
context.strokeStyle = "colorValue"
```

Note: The **strokeStyle** property can be set to values other than color, which we will see later.

3. Building paths

We will now explore methods related to drawing lines and curves.

➤ The **moveTo(x,y)** method

The **moveTo(x,y)** method creates a new subpath starting from (x,y). In other words, the drawing head moves to (x,y). There is no default value for the starting point, you must use this to create a new subpath.

➤ The **lineTo(x,y)** method

The **lineTo(x,y)** method draws a line from the current position to (x,y). The line drawn is not actually visible on the screen until the **stroke()** method is called (see below).

➤ The **stroke()** method

This is a very important method. This method actually renders the path on the screen. The lines and curves you have drawn are not visible on the screen until the **stroke()** method is called.

Example:

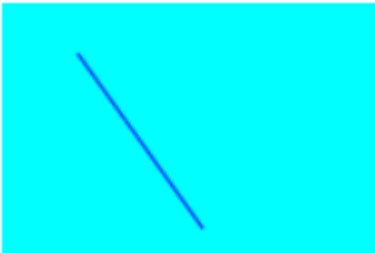
```
<!DOCTYPE HTML>
<html> <head>
  <style>
    canvas{
      background-color:cyan;
```

```

}
</style>
</head>
<body>
<canvas id="myCanvas" width="150" height="100"></canvas>
<script>
    var cv = document.getElementById("myCanvas");
    var context2D = cv.getContext('2d');
    context2D.strokeStyle="blue";
    context2D.moveTo(30,20)
    context2D.lineTo(80,90);
    context2D.stroke();
</script>
</body>
</html>

```

Output:



Note:

It is important to understand the position of the "drawing head" at any given time. The statement, `context2D.moveTo(30,20)`, moves the drawing point to the coordinates (30,20). The statement `context2D.lineTo(80,90)` draws the line (which you don't see until `stroke()` method is applied) and moves the point to the location (80,90). Thus the new location for the drawing head is (80, 90).

➤ The `closePath()` method

The `closePath()` method draws a line from the current location to the start of the subpath. Again, this line is not visible until `stroke()` method is called.

EXAMPLE:

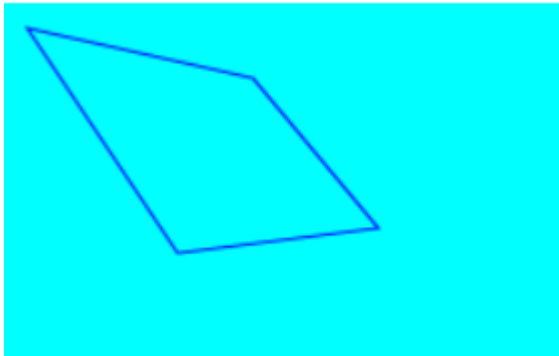
```

<body>
<canvas id="myCanvas" width="250" height="200"></canvas>

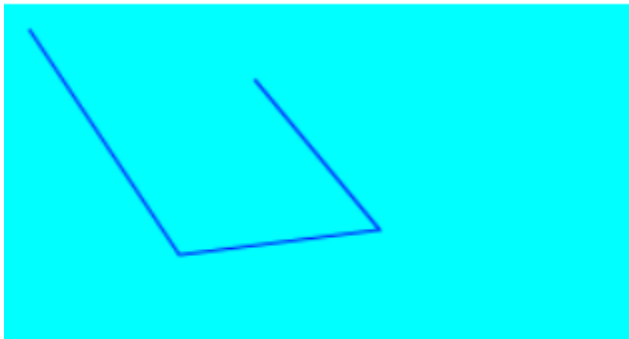
```

```
<script>
  var cv = document.getElementById("myCanvas");
  var context2D = cv.getContext('2d');
  context2D.beginPath();
  context2D.strokeStyle="blue";
  context2D.moveTo(100,30);
  context2D.lineTo(150,90);
  context2D.lineTo(70,100);
  context2D.lineTo(10,10);
  context2D.closePath();
  context2D.stroke();
</script>
```

Output:



Notice the statement `context2D.closePath()`. This statement is drawing the line between the last point and the first point. If you remove this statement, the output will be,



➤ The `arc()` method

The **`arc()`** method draws an arc of a circle.

SYNTAX:

```
context2D.arc(x,y,r,startAngle, endAngle,direction)
```

This draws an arc of a circle. Argument details:

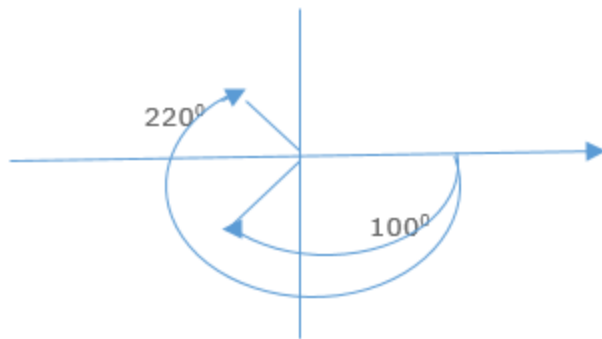
(x,y)	Coordinates of the center of the circle
r	radius of the circle whose arc is being drawn
startAngle and endAngle	Starting and ending angles of the arc. See below for more details
direction	Arc will be drawn from the starting angle to ending angle depending on the value of direction: clockwise or anticlockwise. See below for details.

More on startAngle and endAngle:

These are in radians (not in degrees). To convert degrees into radians, just multiple by $(\text{Math.PI}/180)$. For example, 45 degrees in radians is $(\text{Math.PI}/180)*45$. 75 degrees in radians is $(\text{Math.PI}/180)*75$.

These angles are measured in clockwise direction starting from positive X-axis.

EXAMPLES:



More on direction

The argument *direction* specifies clockwise or counterclockwise.

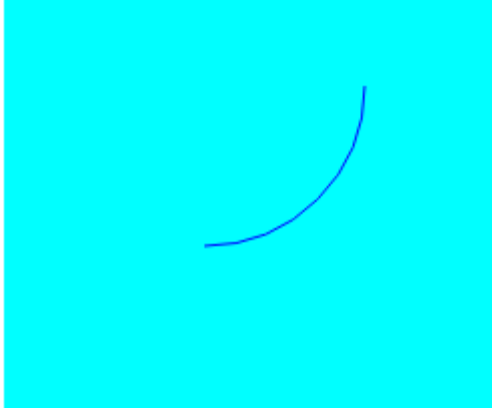
Default is clockwise. That is, if you don't include this argument, the arc is drawn clockwise from starting angle to ending angle.

If the argument is "counterclockwise" or "anticlockwise" or true, the arc is drawn counterclockwise from starting angle to ending angle.

EXAMPLE:

```
context2D.arc(100,70,90,(Math.PI/180)*0,(Math.PI/180)*90);
context2D.stroke();
```

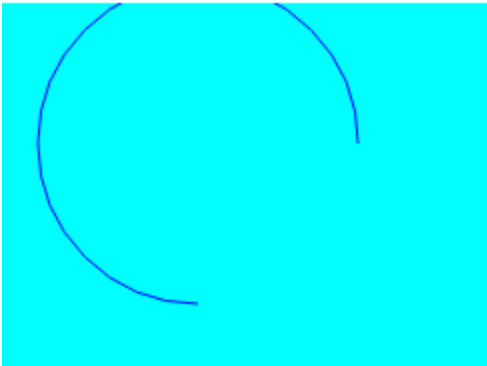
Here the center of the circle is (100,70), radius 90 and start angle is 0 degrees and end angle is 90 degrees. Resulting arc (clockwise 0° to 90°):



EXAMPLE:

```
context2D.arc(100,70,80,(Math.PI/180)*0,(Math.PI/180)*90, "counterclockwise");  
context2D.stroke();
```

Same as before except the arc is drawn counterclockwise.

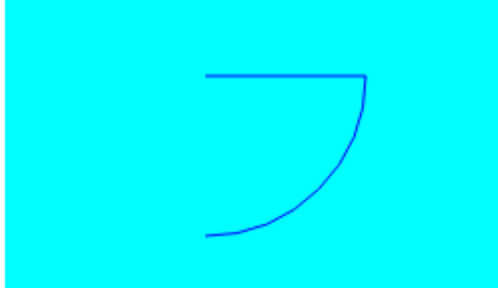


Using closePath() with arcs

The following code moves the drawing point to the center and draws the arc:

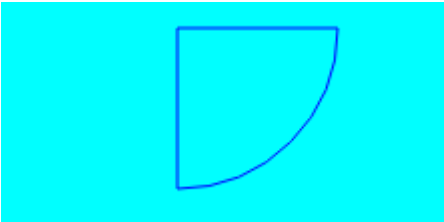
```
context2D.moveTo(100,70);  
context2D.arc(100,70,80,(Math.PI/180)*0,(Math.PI/180)*90);  
context2D.stroke();
```

RESULT:



Let us now include, `closePath()`:

```
context2D.moveTo(100,70);
context2D.arc(100,70,80,(Math.PI/180)*0,(Math.PI/180)*90);
context2D.closePath();
context2D.stroke();
```



4. Setting line styles

The following attributes can be used to set line properties.

➤ The **lineWidth** attribute

The **lineWidth** attribute is used to set the width of lines.

Syntax:

```
context2D.lineWidth = "n";
```

Where n is a number specifying the line width in pixels.

EXAMPLE:

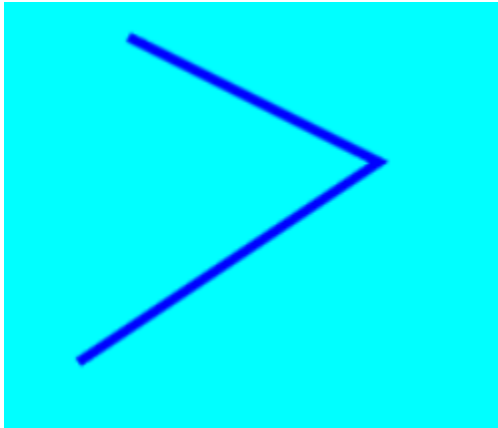
```
<body>
<canvas id="myCanvas" width="250" height="350"></canvas>
<script>
  var cv = document.getElementById("myCanvas");
  var context2D = cv.getContext('2d');
  context2D.beginPath();
  context2D.strokeStyle="blue";
  context2D.moveTo(100,70);
  context2D.lineWidth="4";
```

```

    context2D.lineTo(200,120);
    context2D.lineTo(80,200);
    context2D.stroke();
</script>
</body>

```

OUTPUT:



➤ **The lineCap attribute**

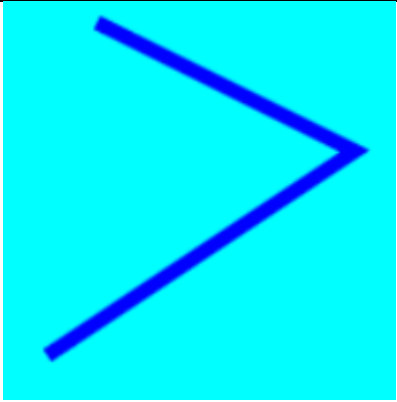
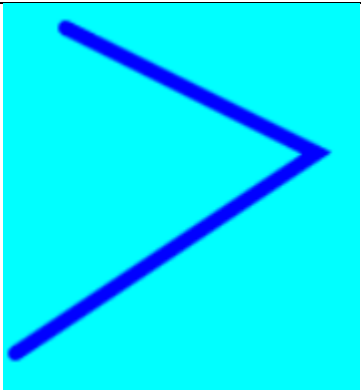
The **lineCap** attribute specifies how the ends of lines should be displayed: with a cap or not.

Three possible values for **lineCap** are:

butt	Flat edge is added to ends of lines. This is default.
square	A square cap is attached to the end of lines.
round	A round cap is attached to the end of lines.

EXAMPLES:

<pre> context2D.lineWidth="6"; context2D.lineCap="butt"; context2D.lineTo(200,120); context2D.lineTo(80,200); context2D.stroke(); </pre>	
--	--

<pre>context2D.moveTo(100,70); context2D.lineWidth="6"; context2D.lineCap="square"; context2D.lineTo(200,120); context2D.lineTo(80,200); context2D.stroke();</pre>	
<pre>context2D.moveTo(100,70); context2D.lineWidth="6"; context2D.lineCap="round"; context2D.lineTo(200,120); context2D.lineTo(80,200); context2D.stroke();</pre>	

Carefully notice the ends of lines.

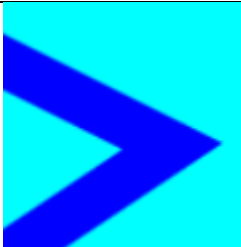
➤ The **lineJoin** attribute


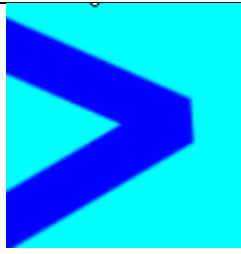
The lineJoin attribute specifies the style of corners where two lines meet. Three possible values for **lineJoin** are:

miter	The corners will be pointed
round	The corners will be round
bevel	The corners will be beveled

EXAMPLES:

In the following, I have deliberately made the line width large to see the results clearly.

<pre>context2D.lineJoin="miter";</pre>	
--	--

<code>context2D.lineJoin="round";</code>	
<code>context2D.lineJoin="bevel";</code>	

Notice the corners carefully.

5. Drawing text to the canvas

We will now explore attributes and methods to write text to the canvas.

Before we draw text to the canvas, we have to specify (1) text font and (2) text style. Yes, we have canvas properties for these two.

➤ The font property

SYNTAX:

`context2D.font="font properties list";`

The "font properties list" consists of the following property values, separated by spaces (these are same as in CSS): font-style font-variant font-weight font-size/line-height font-family. The font value in canvas does not support all CSS font properties.

The default is 10px sans-serif. The list need not contain all of the properties.

Following are possible values for these properties:

Property	Possible values	Remarks
font-style	normal italic oblique	Default is normal. Italic forms are generally cursive in nature while oblique faces are typically sloped versions of the regular face.
font-variant	normal small-caps	Small-caps display uppercase letters but are reduced to the

		size of lowercase letters. This works well in Chrome, but not in Firefox.
font-weight	normal bold bolder lighter 100 200 300 400 500 600 700 800 900	The 'font-weight' property specifies the weight of glyphs in the font, their degree of blackness or stroke thickness. Higher the number the darker is the font.
font-size	xx-small x-small small medium large x-large xx-large larger smaller specify length using px, pt, em or %	The names indicate the size of the font. You can specify absolute length using the usual units.
line-height	Just a number (like 2, 3...) specifying line spacing.	
font-family	Font-family such as Helvetica, Verdana, sans-serif "Lucida" Grande, sans-serif '21st Century', fantasy	This is same as in CSS.

Note:

You need not specify all the properties. Default is assumed for properties not indicated.

EXAMPLES:

In the following examples, text rendered is "New York". We will soon see how to render texts.

Font specification	Result
context2D.font = "normal bold 20pt serif";	New York
context2D.font = "italic bold 20pt sans-serif";	<i>New York</i>
context2D.font = "normal 500 20pt Verdana";	New York
context2D.font = "normal 500 20pt '21st Century', fantasy";	New York
context2D.font = "normal 500 xx-small serif";	New York

<code>context2D.font = "normal 500 xx-large serif";</code>	New York
<code>context2D.font = "normal small-caps bold 20px serif";</code>	NEW YORK

Note:

Everything you ever wanted to know about fonts is here: <http://www.w3.org/TR/css3-fonts/#propdef-font-style>.

➤ The **fillStyle** property

The value of **fillStyle** property can be a simple color value or a much nicer gradient value. The text is rendered using the specified color or the gradient.

To assign a color value, just use the syntax:

```
context2D.fillStyle = "colorValue"
```

Assigning gradients to **fillStyle**

This is more involved and has nicer effect than just one color. This is similar to linear gradients in CSS.

There are two steps in defining a linear gradient: (1) Create linear gradient object and (2) specify stops defining how the colors are distributed along the gradient.

Syntax for creating a linear gradient object:

```
gradientObject = context2D. createLinearGradient(x0, y0, x1, y1)
```

`gradientObject` represents a linear gradient object that paints along the line starting from (x0,y0) through (x1,y1).

Once a gradient object has been created, stops are placed along it to define how the colors are distributed along the gradient, using the syntax:

```
gradientObject.addColorStop(offset, color);
```

Adds a color stop with the given color to the gradient at the given offset. 0.0 is the offset at beginning of the gradient, 1.0 is the offset at the end.

After defining the gradient object completely, assign it to **fillStyle** property.

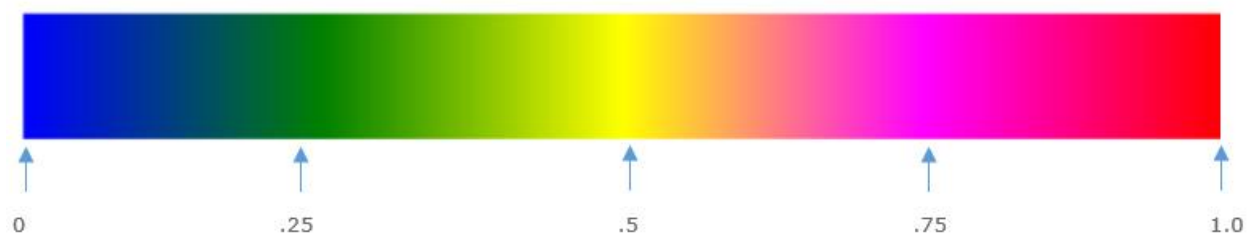
EXAMPLE:

```

gradient = context2D.createLinearGradient(10,10,310,10);
gradient.addColorStop("0", "blue");
gradient.addColorStop(".25", "green");
gradient.addColorStop(".50", "yellow");
gradient.addColorStop(".75", "magenta");
gradient.addColorStop("1.0", "red");
context2D.fillStyle = gradient;

```

This example creates a gradient object covering (10,10) to (310,10). It then specifies color stops of blue, green, yellow, magenta and red at 0, .25, .5, .75 and 1.0. To understand how these color stops work, consider a small strip from (10,10) to (310,10):



The total width is 300 pixels. At each $\frac{1}{4}$ of the distance the color changes (gradually). Blue starts at 0, gradually changes over to green until .25 (distance 75 pixels), then gradually changes over to yellow until .5 (distance 150 pixels), then gradually changes over to magenta until .75 (distance 225 pixels) and finally gradually changes over to red until 1.0 (distance 300 pixels). It will be the same color vertically. Any rendering left of this area will be blue and right will be red.

Now rendering text on canvas:

➤ The **fillText()** method

Basic syntax:

```
context2D.fillText("text",x,y);
```

The **fillText**("text",x,y) method fills the given "text" at position (x,y). In rendering the text, it uses the values specified in **font** and **fillStyle** properties.

EXAMPLE:

```

<html>
<head><title>Fill Text Example</title></head>
<body>
<canvas id="myCanvas" width="500" height="350"></canvas>

```

```

<script type="text/javascript">
    var canvas = document.getElementById("myCanvas");
    var context2D = canvas.getContext("2d");
    context2D.font = "normal small-caps bold 40px serif";
    context2D.fillStyle = "blue";
    context2D.fillText("New York",30,100);
</script>
</body>
</html>

```

OUTPUT:

NEW YORK

EXAMPLE:

```

<html>
<head><title>Fill Text Example</title></head>
<body>
<canvas id="myCanvas" width="500" height="300"></canvas>
<script type="text/javascript">
    var canvas = document.getElementById("myCanvas");
    var context2D = canvas.getContext("2d");
    gradient = context2D.createLinearGradient(30,130,300,130);
    gradient.addColorStop("0", "blue");
    gradient.addColorStop(".25", "green");
    gradient.addColorStop(".50", "yellow");
    gradient.addColorStop(".75", "magenta");
    gradient.addColorStop("1.0", "red");
    context2D.font = "normal small-caps bold 40px serif";
    context2D.fillStyle = gradient;
    context2D.fillText("New York", 30,100);
</script>
</body>
</html>

```

OUTPUT

NEW YORK

NOTE:

- Another variation of **fillText()** method: `context2D.fillText("text",x,y,maxWidth)`; The last argument `maxWidth` is a number specifying maximum width. If this argument is provided, the text will be scaled to fit that width if necessary.
- Yet another variation of `fillText()` is the method `strokeText()`. The `strokeText()` works exactly like `fillText()`, except that the text is rendered as outline, without filling it, as shown below:



6. Drawing rectangles to the canvas

There are three methods that immediately draw rectangles to the canvas. They each take four arguments; the first two give the x and y coordinates of the top left of the rectangle, and the second two give the width w and height h of the rectangle, respectively.

➤ **The `fillRect(x,y,w,h)` method**

Paints the specified rectangle onto the canvas, using the current fill style.

➤ **The `strokeRect(x,y,w,h)` method**

Paints the box that outlines the given rectangle onto the canvas, using the current stroke style.

➤ **The `clearRect(x,y,w,h)` method**

Clears all pixels on the canvas in the given rectangle to background color.

Example:

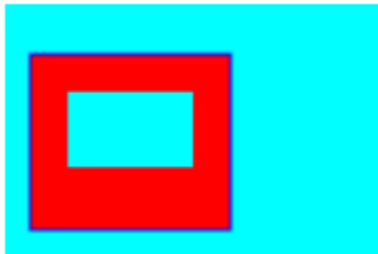
```
<!DOCTYPE HTML>
<html>
<head>
<style>
canvas{
background-color:cyan;
}
</style>
```

```

</head>
<body>
<canvas id="myCanvas" width="150" height="100"></canvas>
<script>
    var cv = document.getElementById("myCanvas");
    var context2D = cv.getContext('2d');
    context2D.fillStyle = "red";
    context2D.strokeStyle="blue";
    context2D.fillRect(10,20,80,70);
    context2D.strokeRect(10,20,80,70);
    context2D.clearRect(25,35,50,30);
</script>
</body>
</html>

```

Output:



➤ Rectangle fillStyle value linear gradient

This example uses linear gradient for fillStyle.

```

<!DOCTYPE HTML>
<html>
<head><title>Fill Text Example</title></head>
<body>
<canvas id="myCanvas" width="500" height="300"></canvas>
<script type="text/javascript">
    var canvas = document.getElementById("myCanvas");
    var context2D = canvas.getContext("2d");
    gradient = context2D.createLinearGradient(10,10,300,10);
    gradient.addColorStop("0", "blue");
    gradient.addColorStop(".25", "green");
    gradient.addColorStop(".50", "yellow");
    gradient.addColorStop(".75", "magenta");

```

```

    gradient.addColorStop("1.0", "red");
    context2D.fillStyle = gradient;
    context2D.fillRect(10,10,250,200);
</script>
</body>
</html>

```

Result:



➤ **Rectangle fillStyle value radial gradient**

Radial gradients are similar to linear gradients, except in radial gradients colors spread is circular, not linear. The method used to create radial gradient is `createRadialGradient()`,

SYNTAX:

```
context2D.createRadialGradient(x0,y0,r0,x1,y1,r1);
```

Here,

(x0,y0) and r0 are the center and radius of the "start circle."

And

(x1,y1) and r1 are the center and radius of the "end circle."

After this object is created, as in the case of linear gradient, you add "color stops". After defining the gradient object completely, assign it to **fillStyle** property.

EXAMPLE:

```

<!DOCTYPE HTML>
<html>
<head><title>Fill Text Example</title></head>
<body>
<canvas id="myCanvas" width="500" height="300"></canvas>
<script type="text/javascript">
    var canvas = document.getElementById("myCanvas");
    var context2D = canvas.getContext("2d");

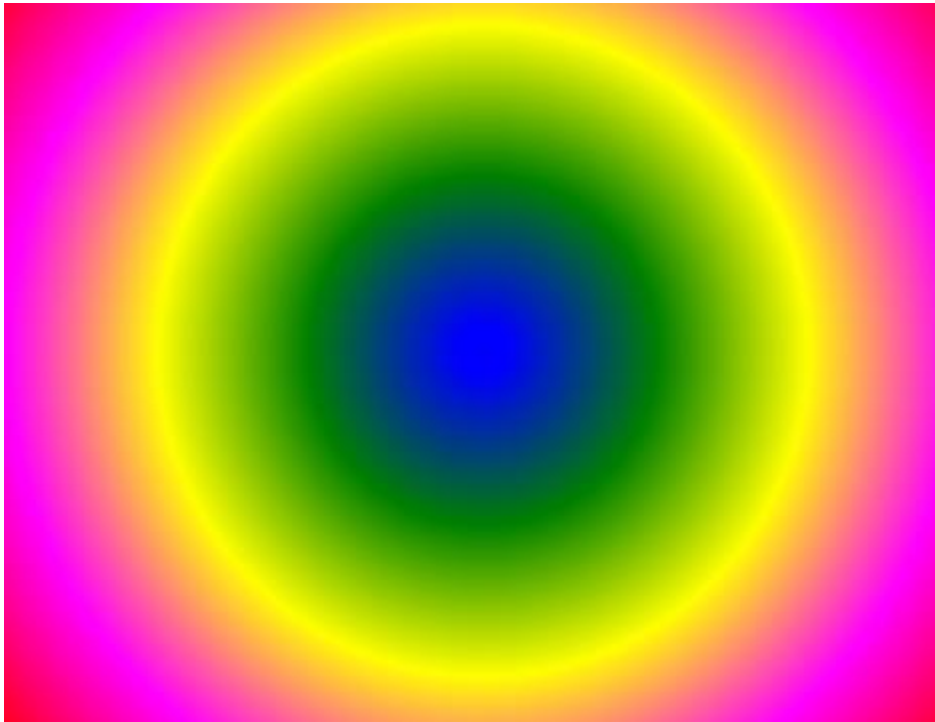
```

```

gradient = context2D.createRadialGradient(250,150,10,250,150,250);
gradient.addColorStop("0", "blue");
gradient.addColorStop(".25", "green");
gradient.addColorStop(".50", "yellow");
gradient.addColorStop(".75", "magenta");
context2D.fillStyle = gradient;
context2D.fillRect(10,10,450,300);
</script>
</body>
</html>

```

OUTPUT:



➤ **Rectangle fillStyle value pattern**

To paint a rectangle with an image, you need to take these steps:

1. First create an image object:

```
var imageObject = new Image();
```

2. Assign an image to the object:

```
imageobject.src = "imageFile";
```

3. Create a pattern:

```
var myPattern = context2D.createPattern(imageObject,"repeatFactor");
```

Where *repeatFactor* is one of these:

repeatFactor	Meaning
repeat	Repeats the image in both X and Y directions and fills the rectangle.
repeat-x	Repeats the image only in X direction
repeat-y	Repeats the image only in Y direction
no-repeat	Image does not repeat – appears only once.

Example:

```
<!DOCTYPE HTML>
<html>
<head><title>pattern Example</title></head>
<body>
<canvas id="myCanvas" width="500" height="300"></canvas>
<script type="text/javascript">
    var canvas = document.getElementById("myCanvas");
    var context2D = canvas.getContext("2d");
    var imageObject = new Image();
    imageObject.src="lotus.jpg";
    var pattern = context2D.createPattern(imageObject,"repeat");
    context2D.fillStyle = pattern;
    context2D.fillRect(10,10,450,300);
</script>
</body></html>
```

Output:



Note:

Works well in Firefox (wait for the image to load). Does not seem to work in Chrome.

7. Saving and restoring graphics state

You can save the current state of the context by using the context method **save()**. Once you save a context state, you can retrieve it by using the **restore()** method. The **save()** and **restore()** methods work like a "stack." This means "last in first out." Each **restore()** method retrieves the last saved context which has not been restored.

EXAMPLE:

In this example, we fill a rectangle with a color and save it. Then we fill the same rectangle with a different color and save it again. We repeat this several times. This sequence of savings creates a stack of context states. Every time we apply **restore()**, we retrieve the last state we saved. We use a simple JavaScript to apply restore.

```
<!DOCTYPE HTML>
<html>
<head><title>canvas Example</title>
<script>
function next()
{
    context2D.restore();
    context2D.fillRect(10,10,250,200);
}
</script>
</head>
<body>
<canvas id="myCanvas" width="250" height="200"></canvas>
<script type="text/javascript">
    var canvas = document.getElementById("myCanvas");
    var context2D = canvas.getContext("2d");
    context2D.fillStyle = "blue";
    context2D.fillRect(10,10,250,200);
    context2D.save();
    context2D.fillStyle = "orange";
    context2D.fillRect(10,10,250,200);
    context2D.save();
    context2D.fillStyle = "green";
    context2D.fillRect(10,10,250,200);
    context2D.save();
```

```

context2D.fillStyle = "magenta";
context2D.fillRect(10,10,250,200);
context2D.save();
context2D.fillStyle = "yellow";
context2D.fillRect(10,10,250,200);
context2D.save();
</script>
<a href="javascript:next()">Next</a>
</body>
</html>

```

Try It: <http://vulcan.seidenberg.pace.edu/~nmurthy/IT614/Chapter10/canvasSaveRestore.html>

8. Creating shadows

We can create shadows in all drawings and text on a canvas. The context provides four attributes to create shadow effects:

➤ **`context.shadowColor = "colorValue";`**

This sets the shadow color to the specified color.

➤ **`context.shadowoffsetX = "value";`**

This sets the horizontal distance of the shadow. The value is just a number. The default value is zero. You can use positive or negative values to control the position of a shadow.

➤ **`context.shadowoffsetY = "value";`**

This sets the vertical distance of the shadow. The value is just a number. The default value is zero. You can use positive or negative values to control the position of a shadow.

➤ **`context.blur = "value";`**

This sets level of blurring effect on shadows. The value is just a number. The default value is 0.

Example:

```

<!DOCTYPE HTML>
<html>
<head>
<title>Shadow example</title>
<script type="text/javascript">
    function draw() {
        var canvas = document.getElementById("myCanvas");

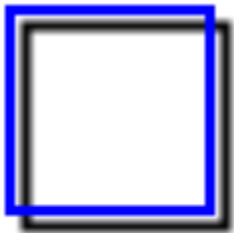
```

```

    if (canvas.getContext) {
    var ctx = canvas.getContext("2d");
    ctx.lineWidth = "5";
    ctx.strokeStyle = "blue";
    ctx.shadowColor = "black";
    ctx.shadowOffsetX = "8";
    ctx.shadowOffsetY = "8";
    ctx.shadowBlur = "4";
    ctx.strokeRect(10, 10, 100, 100);
    }
}
</script>
</head>
<body>
    <canvas id="myCanvas" width="150" height="150"></canvas>
    <a href="javascript:draw()">Draw</a>
</body>
</html>

```

When you click the JavaScript link, you will see rectangle with shadow. Experiment by changing the attribute vales and see the output.



[Draw](#)

9. Transformations

HTML5 canvas provides methods to transform graphics on canvas.

➤ The **translate(x,y)** method

The **translate(x,y)** moves the canvas origin to the specified location (x,y).

EXAMPLE:

```
<!DOCTYPE HTML>
```

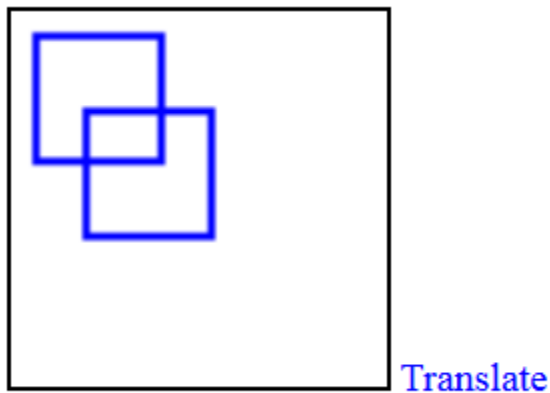


```

<html>
<head>
  <title>Canvas Translate example</title>
  <script>
    function translate() {
      ctx.translate(20, 30);
      ctx.strokeRect(10, 10, 50, 50);
      ctx.stroke();
    }
  </script>
</head>
<body>
  <canvas id="myCanvas" width="150" height="150"></canvas>
  <a href="javascript:translate()">Translate</a>
  <script>
    var canvas = document.getElementById("myCanvas");
    var ctx = canvas.getContext('2d');
    canvas.style.border = "2px solid"; // Put a border around the canvas.
    ctx.beginPath();
    ctx.lineWidth = "3";
    ctx.strokeStyle = "blue";
    ctx.strokeRect(10, 10, 50, 50);
  </script>
</body>
</html>

```

This example first draws a rectangle. When you click the Translate JavaScript link, the rectangle is translated by (20,30). That is, 20 pixels to the right and 30 pixels down.



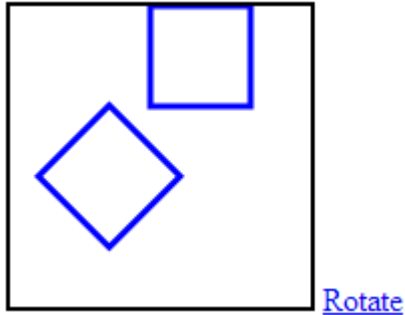
➤ The rotate(*angle*) method

The canvas is rotated by the specified angle. The argument angle is radians. To convert degrees into radians use the formula: $(\text{Math.PI}/180) * (\text{degrees})$.

EXAMPLE:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Canvas Rotate example</title>
  <script>
    function rotate() {
      ctx.rotate((Math.PI/180)*45);
      ctx.strokeRect(70, 0, 50, 50);
      ctx.stroke();
    }
  </script>
</head>
<body>
  <canvas id="myCanvas" width="150" height="150"></canvas>
  <a href="javascript:rotate()">Rotate</a>
  <script>
    var canvas = document.getElementById("myCanvas");
    var ctx = canvas.getContext('2d');
    canvas.style.border = "2px solid";
    ctx.beginPath();
    ctx.lineWidth = "3";
    ctx.strokeStyle = "blue";
    ctx.strokeRect(70, 0, 50, 50);
  </script>
</body>
</html>
```

Output:



Note

It is not the rectangle which is rotated, the canvas is rotated and the rectangle is redrawn on the new canvas location.

➤ The **scale(x,y)** method

Scales the current context by the specified horizontal (x) and vertical (y) factors.

Scaling results in shrink or enlarge (depending on the arguments). For example, `context.scale(1,.5)` halves the vertical (or y-axis) values that are used in context and leaves the horizontal (or x-axis) values the same. Similarly, `context.scale(2,2)` doubles the size of the graphics.

EXAMPLE:

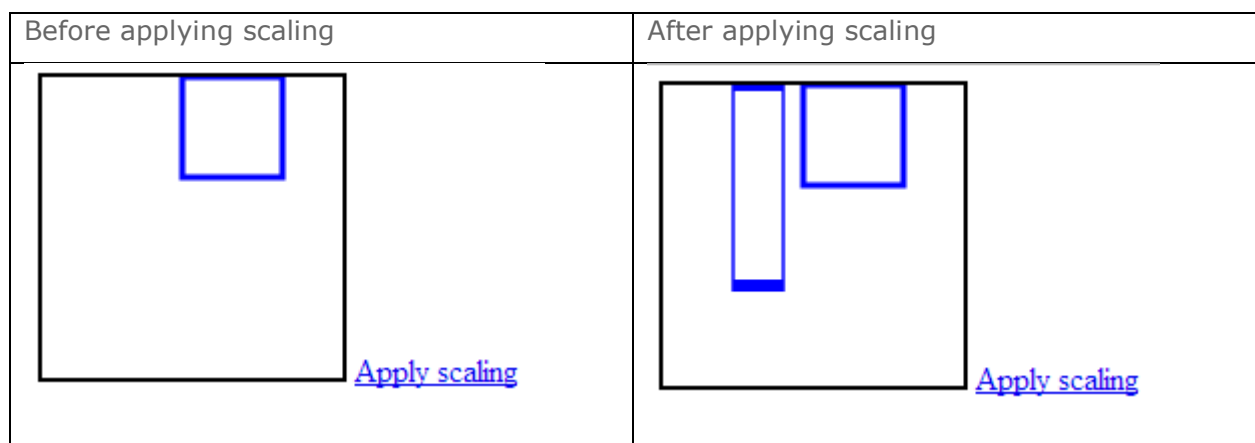
```
<!DOCTYPE HTML>
<html>
<head>
  <title>Canvas scale example</title>
  <script>
    function scale() {
      ctx.scale(.5,2);
      ctx.strokeRect(70, 0, 50, 50);
      ctx.stroke();
    }
  </script>
</head>
<body>
  <canvas id="myCanvas" width="150" height="150"></canvas>
  <a href="javascript:scale()">Apply scaling</a>
  <script>
    var canvas = document.getElementById("myCanvas");
    var ctx = canvas.getContext('2d');
```

```

        canvas.style.border = "2px solid";
        ctx.beginPath();
        ctx.lineWidth = "3";
        ctx.strokeStyle = "blue";
        ctx.strokeRect(70, 0, 50, 50);
    </script>
</body>
</html>

```

Output:



➤ The **setTransform(*a,b,c,d,e,f*)** method

To understand `setTransform()` completely involves matrix multiplication (from linear algebra). We will not discuss matrix multiplication and see how this works. Instead, we will see what each of the arguments do.

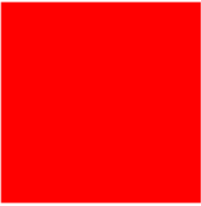
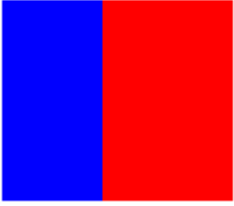
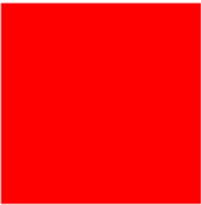

The **setTransform()** method takes 6 parameters. Here is an explanation of what they do:

- The first parameter (*a*)

This scales the drawing horizontally.

EXAMPLE:


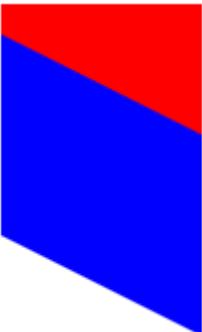

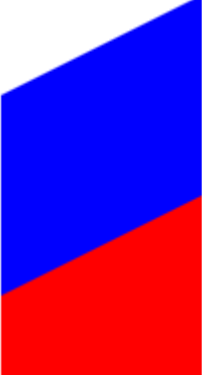
Before applying scaling	Apply horizontal scaling	Result

	<code>setTransform(.5,0,0,1,0,0);</code>	
	<code>context.setTransform(1.5,0,0,1,0,0);</code>	

- The second parameter (b)

This skews the drawing horizontally.


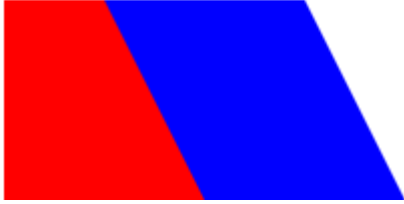


EXAMPLE:

Before applying skewing	Apply horizontal skewing	Result
	<code>setTransform(1,0.5,0,1,0,0);</code>	
	<code>setTransform(1,-0.5,0,1,0,0);</code>	

- The third parameter (c)

This skews the drawing vertically.



EXAMPLE:


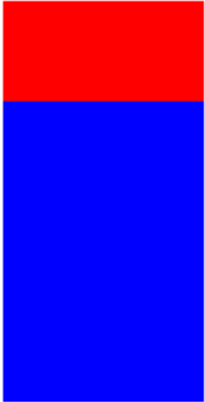
Before applying skewing	Apply horizontal skewing	Result
	<code>setTransform(1,0,0.5,1,0,0);</code>	
	<code>setTransform(1,0,-0.5,1,0,0);</code>	

- The fourth parameter (d)

This scales the drawing vertically

EXAMPLE:




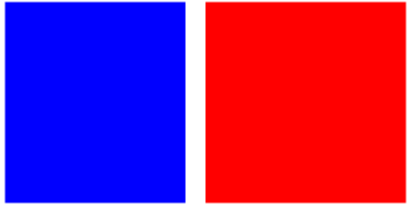
Before applying scaling	Apply vertical scaling	Result
	<code>setTransform(1,0,0.5,1,0,0);</code>	

	<code>setTransform(1,0,0,1.5,0,0);</code>	
---	---	---

- The fifth parameter (e)

This shifts the drawing horizontally

EXAMPLE:







Before applying transformation	Apply horizontal shift	Result
	<code>setTransform(1,0,0,1,110,0);</code>	
	<code>setTransform(1,0,0,1,-110,0);</code>	

- The sixth parameter (f)

This shifts the drawing vertically.

EXAMPLE:

Before applying transformation	Apply vertical shift	Result
--------------------------------	----------------------	--------

	<code>setTransform(1,0,0,1,0,110);</code>	 
	<code>setTransform(1,0,0,1,0,-110);</code>	 

10. Drawing images to the canvas

Before you can draw an image to the canvas, you should create an image object of your image file. This how you do it:

```
var img = new Image() ; // Create new Image object
img.src = 'Empire.jpg' ; // assign source to image
```

Now, you can add the image object to the canvas using the `context.drawImage()` method.

Syntax

The `drawImage()` method has three versions: 3 arguments, 5 arguments and 9 arguments:

- **context.drawImage(image,x,y);**

The first argument is the image object to be drawn. (x,y) is the point on the canvas where the top-left corner of the image will be placed.

- **context.drawImage(image,x,y,w,h);**

The arguments image, x and y are same as before. w and h specify the width and the height to be used to display the image.

- **context.drawImage(image,sx,sy,dx,dy,x,y,w,h);**

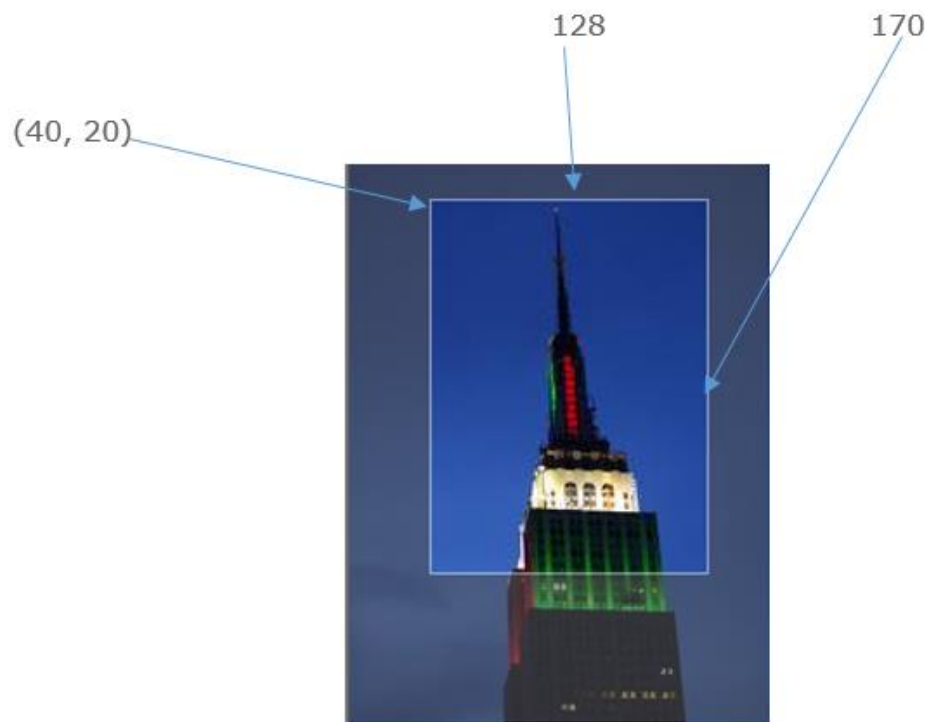
The arguments image, x,y,w, and h are same as before. Let us now focus on the parameters, sx,sy,dx,dy. Consider the rectangle at (sx,sy) width dx and height dy on the original image. This **drawImage()** method clips that part of the rectangle from the image and draws it on the canvas.

EXAMPLE:

Here is an image (width=196, height=260):



Now consider a portion of the image to be cropped (crop width=128, height=170)



11.Animation on canvas

First let us discuss a basic way of animation using the usual JavaScript API..

➤ The **setInterval()** method

The windows `setInterval(function, delay)` method takes two arguments, function and delay. When the `setInterval()` method is executed, JavaScript starts repeatedly executing the function specified in the first argument every delay milliseconds.

Here is a simple example.

EXAMPLE:

This example keeps showing an alert box with "Hello" every 5000 milliseconds:

```
<!DOCTYPE html>
<html>
  <head> <title>setInterval example</title>
  <script>
function sayHello(){
  alert("Hello");
}
</script>
</head>
<body>
<h2>Say Hello every 5000 milliseconds</h2>
  <script type="text/javascript">
    setInterval(sayHello,5000);
  </script>
</body>
</html>
```

Note:

The call to the function (the first argument) in `setInterval()` does not require the usual `()`. It is NOT, `setInterval(sayHello(), 5000)`; but it is `setInterval(sayHello, 5000)`;

Note:

setInterval() returns a value, which can be used to stop execution of the function *functionName()*.

➤ The **setTimeout()** method

The windows **setTimeout(function, delay)** method is similar to **setInterval()** method, except the **setTimeout()** method runs only once. That is, when the **setTimeout()** method is executed, JavaScript executes the function specified in the first argument once after milliseconds.

➤ The **clearInterval()** method

As described, **setInterval()** method makes a function execute every specified number of milliseconds. To cancel this the **clearInterval()** method is used.

The **setInterval()** method returns a code. This code is need to apply **clearInterval()** method.

First assign the return code a variable:

```
timerId = setInterval(functionName,n);
```

Now, to cancel this, the syntax is:

```
clearInterval(timerId)
```

Where *timerId* is the value returned by the **setInterval()** call.

EXAMPLE:

In this example, we first draw an image on a canvas. Our JavaScript code replaces that image by another image after 1000 milliseconds.

```
<html>
<head> <title>Two images example</title>
<script>
function replace(){
  ctx.drawImage(image2,0,0,200,200);
}
</script>
</head>
<body>
<canvas id="myCanvas" width="400" height="400"></canvas>
<script>
var ctx = document.getElementById('myCanvas').getContext('2d');
```

```

var image1 = new Image();
image1.src = "MtFuji.gif";
var image2 = new Image();
image2.src = "lotus.jpg";
ctx.drawImage(image1,0,0,200,200);
setTimeout(replace,1000);
</script>
</body>
</html>

```

Important note:

Whenever you are loading an image to a page the following problem occurs:

The **drawImage()** method may start working before the image is loaded. In such a case, the **drawImage()** does not display any image. In fact, the above code does not show the first image in Chrome (works ok in Firefox). So a solution to the problem is to force JavaScript to wait for the image to completely load before drawing it. It is done using the following code:

```

imageObject.onload = function() {
    ctx.drawImage();
}

```

To make the code above to work well, replace `ctx.drawImage(image1,0,0,200,200);` by the code,

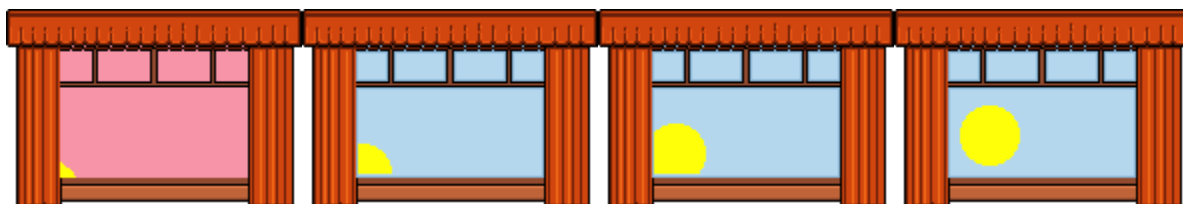
```

image1.onload = function() {
    ctx.drawImage(image1,0,0,200,200);
}

```

Example:

This example uses 15 images:sunrise1.gif,...,sunrise15.gif. Each is an image of sunrise with slight variations. Here are the first four images:



This program displays these fifteen images one after the other with a slight delay to produce animation of sunrise. We use the method **setInterval()**.

```
<!DOCTYPE HTML>
<html>
<head><title>Sunrise </title>
<script>
function init(){
    sun = new Array();
    k = 1;
    for(i=1;i<16;i++){
        sun[i] = new Image();
        sun[i].src = "sunrise/sunrise"+i+".gif";
    }
    ctx.drawImage(sun[k],0,0,300,300);
    setInterval(next,500);
}
function next() {
    k++;
    if(k>15){k=1;}
    ctx.drawImage(sun[k],0,0,300,300);
}
</script>
</head>
<body>
<h2>Sunrise....</h2>
<canvas id="myCanvas" width="300" height="300"></canvas>
<a href="javascript:init()">Start</a>
<script>
var ctx = document.getElementById('myCanvas').getContext('2d');
</script>
</body>
</html>
```

Try It: <http://vulcan.seidenberg.pace.edu/~nmurthy/IT614/Chapter10/sunrise.html>

Another example of a simple animation

The principle of animation: Select a sequence of images any two of which differ in a minor way. Display the sequence of images one after the other with a small delay of time between them. The following example first loads a sequence of images into an array and displays one of the other with a small delay of time between them.

We use **setTimeout()** method. We provide two buttons, faster and slower to decrease and increase the delay, which effects the speed. The STOP button cancels the action.

```
<!DOCTYPE HTML>
<html><head><title>Animation - Cat running </title>
<script>
function slower(){
    speed = speed+50;
}
function faster(){
    speed = speed-50;
}
function stop(){
    clearTimeout(animation);
}
function init(){
    cats = new Array();
    k = 1;
    for(i=1;i<13;i++){
        cats[i] = new Image();
        cats[i].src = "cats/cat"+i+".gif";
    }
    ctx.drawImage(cats[k],0,0,300,300);
    drawNextCat();
}
function drawNextCat() {
    k++;
    if(k>12){k=1;}
    ctx.drawImage(cats[k],0,0,300,300);
    animation = setTimeout(drawNextCat,speed);
}
```

```
</script></head>
<body>
<canvas id="myCanvas" width="400" height="300"></canvas>
<script>
var ctx = document.getElementById('myCanvas').getContext('2d');
speed=250;
</script>
<form>
<input type="button" value="START" onclick="init()">
<input type="button" value="slower" onclick="slower()">
<input type="button" value="faster" onclick="faster()">
<input type="button" value="STOP" onclick="stop()">
</form>
</body></html>
```

Try It: <http://vulcan.seidenberg.pace.edu/~nmurthy/IT614/Chapter10/canvasAnimationCats.html>

Note:

See this Mozilla Web site for three very nice examples of animation:

https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Canvas_tutorial/Basic_animations

New APIs for animation

W3C has recently introduced new APIs for animation, which work more efficiently. The main method in the new API is `requestAnimationFrame`. We will not discuss it. If you like to learn more on this, see W3C Web site: <http://www.w3.org/TR/animation-timing/>. Also see <https://developer.mozilla.org/en-US/docs/Web/API/window.requestAnimationFrame>.
