UNIT - 4

HTML5 - CANVAS

CANVAS : Introduction

The canvas element is used to render simple graphics such as l**ine art, graphs, and other custom graphical elements** on the client side. Initially introduced in the summer of **2004 by Apple in its Safari browser.** The canvas element is now supported in many browsers, including like Firefox, Opera , and Chrome , and as such is included in the HTML5 specification. While Internet Explorer does not directly support the tag as of yet, there are JavaScript libraries3 that emulate
<canvas> syntax using Microsoft's Vector Markup Language (VML).
From a markup point of view, there is little that you can do with a **<canvas>** tag. You simply put the element in the page, name it with an **id** attribute, and define its dimensions with **height** and **width** attributes:

```
<canvas  id="canvas"  width="300"  height="300">
   <strong>Canvas Supporting Browser Required</strong>
</canvas>
```

Note the alternative content placed within the element for browsers that don't support the element.

What is HTML Canvas?

The HTML <canvas> element is used to draw graphics, on the fly, via scripting (usually JavaScript).
The <canvas> element is only a container for graphics. You must use a script to actually draw the graphics.
Canvas has several methods for drawing paths, boxes, circles, text, and adding images.

Benefits and Capabilities of HTML5 Canvas

- **Dynamic Graphics:** With Canvas, we can create interactive and visually appealing graphics that respond to user input or change over time.
- **Cross-Browser Compatibility:** Canvas is supported by all modern web browsers, making it a reliable choice for web development.
- **Performance:** Canvas leverages hardware acceleration to ensure smooth and efficient rendering, making it ideal for high-performance applications.
- **Animation:** Canvas allows us to create animated effects and transitions, providing an engaging user experience.
- **Interactivity:** We can incorporate user interactions, such as mouse clicks or touch events, to create interactive elements and games.
- **Versatility:** Canvas can be used for a wide range of applications, including data visualizations, image editing tools, games, and more.

Setting up Canvas :

To use HTML5 Canvas, you need to include a Canvas element in your HTML markup. The Canvas element serves as the drawing surface and can be positioned and sized as required. Here's an example of setting up a Canvas element:

```
<canvas  id="myCanvas"  width="500"  height="300"> </canvas>
```

In this example, we create a Canvas element with an id of "myCanvas" and set its width to 500 pixels and height to 300 pixels. These dimensions determine the size of the drawing area on the web 0

a) Retrieving the Canvas Element

Once the Canvas element is defined in the HTML markup, you can access it in JavaScript using its **id**. This allows you to interact with the Canvas and perform drawing operations. Here's an example of retrieving the Canvas element:

```
const   canvas = document.getElementById('myCanvas');
```

In this code snippet, we use the *getElementById* method to retrieve the Canvas element with the id "*myCanvas*" and assign it to the variable canvas. This variable can then be used to access the drawing context and perform various operations on the Canvas.

b) <u>Drawing Context and API :</u>

The drawing context is the gateway to the drawing capabilities of the Canvas. It provides methods and properties that allow you to draw shapes, apply styles, manipulate images, and more. To access the drawing context, you need to call the **getContext** method on the Canvas element, specifying the type of context you want. In the case of 2D graphics, you use the argument **'2d'**. Here's an example:

```
const context = canvas.getContext('2d');
```

In this code snippet, we retrieve the 2D rendering context of the Canvas element and assign it to the variable context. This context object is used to perform various drawing operations on the Canvas.

Basic Drawing Operations:

HTML5 Canvas provides several methods for drawing basic shapes and lines. These methods allow you to create **lines, rectangles, circles, and paths** on the Canvas. After you place a **<canvas>** tag in a document, your next step is to use **JavaScript** to access and draw on the element.

For example, the following fetches the object by its **id** value and creates a **two-dimensional** drawing context:

```
var canvas = document.getElementById("canvas");        // Find the canvas element
var context = canvas.getContext("2d");                 // Create drawing object
```

Once you have the drawing context, you can deploy various methods to draw on it.

Checking for browser support :

The fallback content is displayed in browsers which do not support <canvas>. Scripts can also check for support programmatically by simply testing for the presence of the getContext() method.

```
<canvas id="can1" width="150" height="150">Fallback content</canvas>
<script type="text/javascript">
var canvas = document.getElementById(can1);
if (canvas.getContext){
var ctx = canvas.getContext('2d');
// drawing code here
} else {
// canvas-unsupported code here}
</script>
```

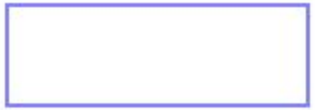
Browser support : Canvas was first introduced by Apple for the Mac OS X Dashboard and later implemented in Safari and Google Chrome. HTML5 Canvas is supported by modern web browsers such as Chrome, Firefox, Safari, Opera, and Internet Explorer 9 and above.

Chrome	Edge	Firefox	Safari	Opera	IE
Yes	Yes	Yes	Yes	Yes	9-11

<u>Drawing Rectangle</u> : strokeRect() , fillRect() , clearRect()

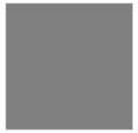
The **strokeRect(x,y,width,height)** method takes x and y coordinates and height and width, all specified as numbers representing pixels. And to set a particular **color** for the stroke, you might set it with the **strokeStyle()** method

```
context.strokeStyle = "blue";
context.strokeRect(10,10,150,50);
```



Similarly, you can use the **fillRect(x,y,width,height)** method to make a rectangle,but this time in a solid fill manner: By **default, the fill color will be black**, but you can define a different fill color by using the **fillColor()** method.

```
context.fillStyle = "grey";
context.fillRect(150,30,75,75);
```

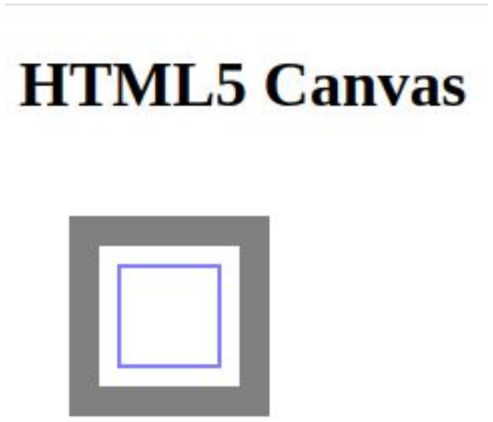


The **clearRect(x, y, width, height)** Clears the specified rectangular area, making it fully transparent.

Each of these **three** functions takes the same parameters.x and y specify the position on the canvas (relative to the origin) of the top-left corner of the rectangle.width and height provide the rectangle's size.
All 3 functions draw a rectangle immediately on the canvas.

A full example using the first **canvas** element with Stroke Rectangle and Filled Rectangle, Clear Rectangle as follows :

```
<!DOCTYPE html>
<html>
<body>
<h1>HTML5 Canvas</h1>
<canvas id="canvas" width="150" height="150">
Browser Doesn't Support
</canvas>
<script type="text/javascript">
 var canvas = document.getElementById('canvas');
 var ctx = canvas.getContext('2d');
 ctx.fillStyle = "grey";
 ctx.fillRect(25,25,100,100);
 ctx.clearRect(40,40,70,70);
 ctx.strokeStyle = "blue";
 ctx.strokeRect(50,50,50,50);
</script>
</body>
</html>
```



The fillRect() function draws a large grey square 100 pixels on each side. The clearRect() function then erases a 70x67 pixel square from the center, And then strokeRect() is called to create a blue rectangular outline 50x50 pixels within the cleared square.

<u>**Drawing Lines**</u> **: moveTo() , lineTo()**

To draw a line on the Canvas, you can use the **moveTo() and lineTo()** methods.

The **moveTo()** method sets the starting point of the line.

The **lineTo()** method specifies the end point. Here's an example:

```
context.moveTo(x1, y1);  // Starting point.
context.lineTo(x2, y2);     // Ending point
context.stroke();              // Draw the line
```

There are a few methods used in drawing and manipulating lines, as well as rectangles and arcs. These methods allow *drawing multiple lines* and shapes in the same canvas, resulting in complex images.

beginPath() :

First thing when we start drawing shapes, we have to call the beginPath(). Internally, paths are stored as a list of sub-paths (lines, arcs, etc) which together form a shape. Every time this method is called, the list is reset and we can start drawing new shapes. I.e :

```
var canvas = document.getElementById('canvas');
var ctx = canvas.getContext('2d');
ctx.beginPath();
// draw shapes methods
```

closePath() VS fill() :

closePath() is optional and tries to close the shape by drawing a straight line from the current point to the start. If the shape has already been closed or there's only one point in the list, this function does nothing.

Alternatively we can use **fill().** This function automatically closes the path and also fills in the shape with color or patterns. When you call fill(), any open shapes are closed automatically, so you don't have to call closePath().

stroke() :

What if we don't want a solid shape? We can use stroke() to draw an empty shape.stroke() draws the shape by stroking its outline. If we don't use fill() nor stroke(), the path or shape will be completely transparent.

moveTo() :

Once we've drawn a line or shape and we want to add another one to the same canvas, we have to use moveTo() to set a different starting point. If we don't use moveTo, the new line or shape will be considered part of the initial shape.

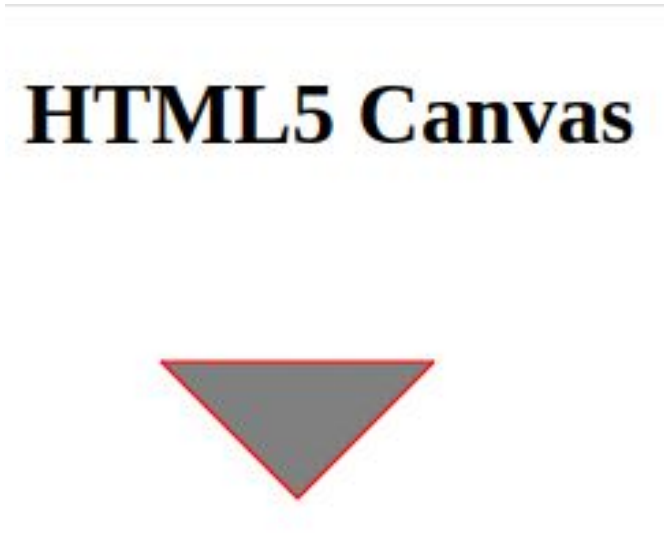
> **moveTo(x, y)** Moves the starting point for a path to the coordinates specified by x and y.

Example : Using beginPath,moveTo,lineTo ..etc

```
<!DOCTYPE html>
<html>
<body><h1>HTML5 Canvas</h1>
<canvas id="canvas" width="150" height="150">Browser Doesn't Support</canvas>
<script type="text/javascript">
var canvas = document.getElementById('canvas');
var context = canvas.getContext('2d');
context.fillStyle = "grey";
context.strokeStyle = "red";

context.beginPath();
context.moveTo(50, 50);
context.lineTo(100, 100);
context.lineTo(150, 50);
context.closePath();

context.fill();
context.stroke();
</script>
</body>
</html>
```



To style the drawing, you can specify the **fillStyle** and **strokeStyle** and maybe even define the width of the line using **context.lineWidth = 10;**

Drawing and Styling Lines and Shapes :

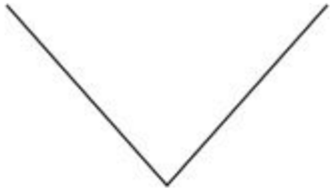
HTML5 defines a complete API for drawing on a canvas element, which is composed of many individual sub-APIs for common tasks. For example, to do some more complex shapes, the **path API** must be used.

The path API stores a collection of subpaths formed by various shape functions and connects the subpaths via a **fill()** or **stroke()** call. Below is the steps to draw complex shapes :

- To begin a path, **context.beginPath()** is called to reset the path collection
- Then, any variety of shape calls can occur to add a subpath to the collection.
- Once all subpaths are properly added, **context.closePath()** can optionally be called to close the loop.
- Then **fill() or stroke()** will also display the path as a newly created shape.

This simple example draws a V shape using lineTo():

```
context.beginPath();
context.lineTo(20,100);
context.lineTo(120,300);
context.lineTo(220,100);
context.stroke();
```



If you were to add context.closePath()before context.stroke(), the V shape would turn into a triangle, because **closePath()** would connect the last point and the first point.

by calling **fill() i**nstead of stroke(), the triangle will be filled in with whatever the fill color is, or black if none is specified.

Drawing Arcs and Curves : Drawing on canvas isn't limited to simple lines; it is also possible to create curved lines using arc(), arcTo(), quadraticCurveTo(), and bezierCurveTo().

Drawing Circles : arc()

To draw a circle on the Canvas, you can use the `arc()` method. The `arc()` method takes the center coordinates, radius, start angle, end angle, and direction (optional) as parameters. Here's an example:

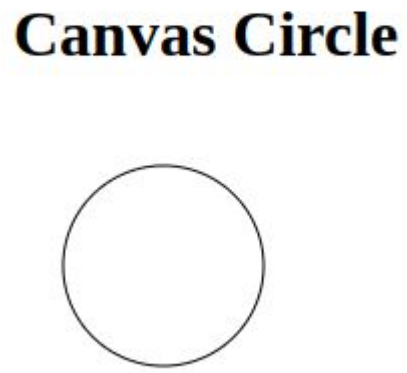
```
context.arc(x, y, radius, startAngle, endAngle, anticlockwise);
context.fill(); // Fill the circle
context.stroke(); // Draw the circle outline
```

- **x and y** are the coordinates of the center of the circle on which the arc should be drawn.
- **radius** is self-explanatory, circle's radius
- The **startAngle** and **endAngle** parameters define the start and end points of the arc in radians, along the curve of the circle.These are measured from the **x** axis
- The anticlockwise parameter is a Boolean value which, when **true**, draws the arc anticlockwise; otherwise, the arc is drawn clockwise.

Angles in the **arc** function are measured in radians, not degrees. To convert degrees to radians you can use the following JavaScript expression: radians = (Math.PI/180)*degrees.

Example of Circle Using arc :

```
<!DOCTYPE html>
<html>
<body><h1>Canvas Circle</h1>
<canvas id="canvas" width="150" height="150">Browser Doesn't Support</canvas>
<script type="text/javascript">
var canvas = document.getElementById('canvas');
var ctx = canvas.getContext('2d');
ctx.beginPath();
ctx.arc(75,75,50,0,Math.PI*2,true);
ctx.stroke();  // if we want a filled circle use ctx.fill()
</script>
</body>
</html>
```


Canvas Circle

arcTo() Method:

The **arcTo()** method is used to draw a curved line segment between two lines. It takes the following parameters:

The syntax for the arcTo() method is as follows:

```
context.arcTo(x1, y1, x2, y2, radius);
```

- **X1 , y1** : The x and y coordinate of the starting point of the first line segment.
- **X2, y2** : The x and y coordinate of the endpoint of the first line segment and the starting point of the curved segment.
- **radius**: The radius of the arc used to create the curved segment.

Note: The **arcTo()** method requires a pen position before calling it. You can set the pen position using the **moveTo()** method.

In summary, the **arc()** method is used to draw circular or elliptical arcs on the Canvas,
while the **arcTo()** method is used to draw curved line segments between two lines.

note that **arcTo()** is not currently supported properly in all browsers, so it may render oddly

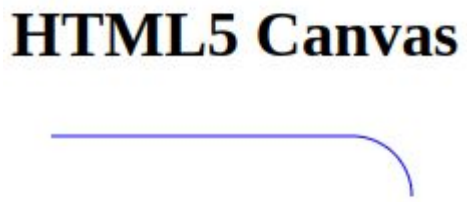
Example of arcTo :

```
<!DOCTYPE html>
<html>
<body><h1>HTML5 Canvas</h1>
<canvas id="myCanvas" width="300" height="300" ></canvas>

<script>
const canvas = document.getElementById("myCanvas");
const context = canvas.getContext("2d");

context.strokeStyle = "blue";
context.beginPath();
context.moveTo(20,10);
context.arcTo(200,10,200,200,30);
context.stroke();

</script>
</body>
</html>
```



<u>Bezier curves</u> : **quadraticCurveTo() , bezierCurveTo()**

In HTML Canvas, the quadraticCurveTo() and bezierCurveTo() methods are used to draw curved paths. Both methods allow you to create smooth curves by defining control points that influence the shape of the curve.

a.) quadraticCurveTo() :

Syntax : **quadraticCurveTo(cpx, cpy, x, y)**

Draws a quadratic Bézier curve from the current position to the end point specified by **x and y,** using the control point specified by **cpx and cpy.**

b.) bezierCurveTo() :

Syntax : **bezierCurveTo(cpx1, cpy1, cpx2, cpy2, x, y)**

Draws a cubic Bézier curve from the current position to the end point specified by **x and y,** using the control points specified by **(cpx1, cpy1) and (cpx2, cpy2).**

The difference between a quadratic Bézier curve and a cubic Bézier curve is that :

- The quadratic curve has just **one control point.**
- And the cubic Bézier curve uses **two control points.**

The **x and y** parameters in both of these methods are the coordinates of the end point.

Both methods (quadraticCurveTo() and bezierCurveTo()) can be used within a path drawing sequence.

- You typically start by calling **beginPath()**,
- move to a starting point using **moveTo(),**
- and then use either **quadraticCurveTo()** or **bezierCurveTo()** to draw the curves.
- Finally, you can use **stroke() or fill()** to render the path on the canvas.

That is :

```
context.beginPath();
context.moveTo(startX, startY);
context.quadraticCurveTo(cpx, cpy, endX, endY);
context.bezierCurveTo(cpx1, cpy1, cpx2, cpy2, x, y);
context.stroke();
```

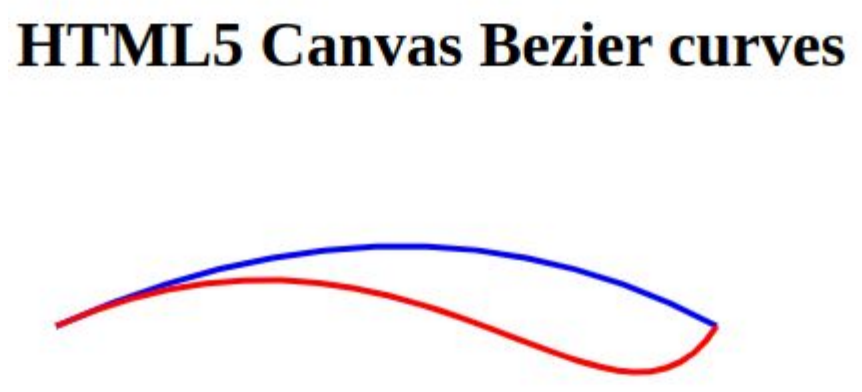
Example of Bezier curves :

```
<!DOCTYPE html>
<html><body><h1>HTML5 Canvas Bezier curves  </h1>
<canvas id="myCanvas" width="360" height="360" ></canvas>
<script>
const canvas = document.getElementById("myCanvas");
const context = canvas.getContext("2d");
context.lineWidth = 3;

  context.beginPath();
  context.moveTo(20, 100);
  context.quadraticCurveTo(200, 20, 350, 100);
  context.strokeStyle = 'blue';
  context.stroke();

  context.beginPath();
  context.moveTo(20, 100);
  context.bezierCurveTo(200, 20, 300, 180, 350, 100);
  context.strokeStyle = 'red';
  context.stroke();

</script>
</body>
</html>
```


HTML5 Canvas Bezier curves

Canvas Color and Style :

As we know, we can change color by setting the **fillStyle** or **strokeStyle** property. Which Can be

color names ("red", "blue", etc.),

hexadecimal values ("#FF0000", "#0000FF", etc.),

RGB values ("rgb(255, 0, 0)", "rgba(0, 0, 255, 0.5)", etc.), or

HSL values ("hsl(0, 100%, 50%)", "hsla(240, 100%, 50%, 0.5)", etc.).

In addition to the CSS color values, you can also set the fillColor to a gradient object. A gradient object can be created by using **createLinearGradient() or createRadialGradient().**

a.) Linear Gradients:

Creates a transition of colors in a straight line between two points. You can create a linear gradient using the **createLinearGradient()** method. After creating the gradient, you can add color stops using the **addColorStop()** method. A color stop defines a *position* and *color* along the gradient line. The **position** is a value between **0 and 1**, representing the relative distance along the gradient line.

Syntax :

```
const gradient = context.createLinearGradient(x1, y1, x2, y2);
gradient.addColorStop(position, color);
```

(x1, y1) is coordinates of the starting point , (x2, y2) is coordinates of the ending point

b.) Radial Gradients:

Radial gradients create a transition of colors radiating from a central point. You can create a radial gradient using the **createRadialGradient()** method. Similar to linear gradients, you can add color stops to the radial gradient using the **addColorStop()** method.

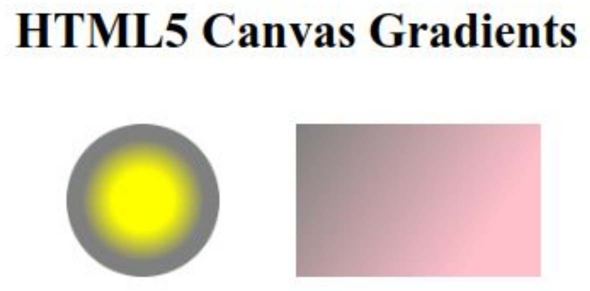
Syntax :

```
const gradient = createRadialGradient(x1, y1, r1, x2, y2, r2)
gradient.addColorStop(position, color);
```

- The parameters represent two circles, one with its center at (x1, y1) and a radius of r1,
- And the other with its center at (x2, y2)with a radius of r2.
- **gradient.addColorStop(position, color)** Creates a new color stop on the gradient object.
- The position is a number between 0.0 and 1.0. You can add as many color stops as you want

Example of Linear and Radial Gradients:

```
<!DOCTYPE html>
<html>
<body><h1> &nbsp; HTML5 Canvas Gradients</h1>
<canvas id="myCanvas" width="400" height="150"></canvas>
<script>
const canvas = document.getElementById('myCanvas');
const context = canvas.getContext('2d');
const Rgrad = context.createRadialGradient(100, 70, 20, 100, 70, 40);
Rgrad.addColorStop(0, 'yellow');
Rgrad.addColorStop(1, 'grey');
context.fillStyle = Rgrad;
context.beginPath();
context.arc(100, 70, 50, 0, Math.PI * 2);
context.fill();
const Lgrad = context.createLinearGradient(200, 20, 320, 100);
Lgrad.addColorStop(0, 'grey');
Lgrad.addColorStop(1, 'pink');
context.fillStyle = Lgrad;
context.fillRect(200, 20, 160, 100);
</script>
</body>
</html>
```


HTML5 Canvas Gradients

Filling and Stroking Paths with Different Styles:

HTML5 Canvas offers different options to style and customize the appearance of filled and stroked paths. These options include:

fillStyle: Specifies the color or style used to fill the path. It can be set to a solid color, a gradient, or a pattern.

strokeStyle: Specifies the color or style used to stroke the path (draw its outline). It follows the same options as fillStyle.

lineWidth: Sets the thickness of the path's stroke.

lineCap: Determines the shape of the ends of lines. Options include "**butt**", "**round**", and "**square**".

lineJoin: Specifies how line segments are joined. Options include "**round**", "**bevel**", and "**miter**".

miterLimit: Applies only to mitered line joins. It sets the limit at which mitered joints are cut off.

Scaling, Rotating,Translation and Transformations :

Scaling Elements : The **scale(x, y)** function, which can be used to scale objects. The x parameter shows how much to scale in the horizontal direction and the y parameter indicates how much to scale vertically.

```
context.scale(.5, 1.5);     // scale tall and thin
context.scale(1.75, .2);   // move short and wide
```

Rotating Elements : The **rotate(angle)** method, which can be used to rotate a drawing in a clockwise direction by an angle defined in radians. Positive values rotate clockwise, and negative values rotate counterclockwise.

```
context.rotate(Math.PI/8);          //  rotate to the right
context.rotate(-Math.PI/8);         // rotate to the left
```

Translating (Moving) Elements : The **translate(x, y)** method is used to move the coordinate system, effectively translating (shifting) all subsequent drawings.

```
context.translate(50, 50);    // Move the coordinate system by (50, 50)
```

Transform Elements : The Transform can be done in 2 Ways :
First, it can be directly modified by calling s**etTransform(m11,m12,m21,m22,dx,dy)**, which resets the matrix to the identity matrix and then calls **transform()** with the given parameters. **Second,** we can directly use the **transform(m11, m12, m21, m22, dx, dy)** method which applies a general transformation matrix to subsequent drawings.

```
context.transform(1, 0.5, 0.5, 1, 0, 0);  // scale and skew the drawing horizontally and vertically
```

On the bright side, with the method, we can do just about anything we want like scale, skew, translate the object with single function.

Working with Images : Using Bitmaps in Drawings

A very interesting feature of canvas is the ability to insert images into the drawing There are several ways to do this :

a) drawImage(img, x, y), which takes an image object and the coordinates for where the image should be placed. The image will be its natural size when called in this manner.

b) We can use **drawImage(img, x, y, w, h)** if you need to modify the image size and set the **width** and **height**.

c) drawImage(img, sx, sy, sw, sh, dx, dy, dw, dh) may be called allows a part of the image to be cut out and drawn to the canvas. The **(sx,sy)** coordinates are the location on the image, and **sw and sh** are the width and height, respectively. The rest of the parameters prefixed with **d** are the same as in the previous form of the method.

The actual image (img) passed in to the **drawImage()** method can come from a few places.

- An image already loaded on the page
- Dynamically created through the DOM
- Another canvas object
- An image created by setting its src to a data: URL

The important thing to remember is that for **drawImage()**, the image must be loaded by the time canvas is ready to access it. This may require use of the **onload function** for the image:

Loading and Displaying Images on the Canvas:

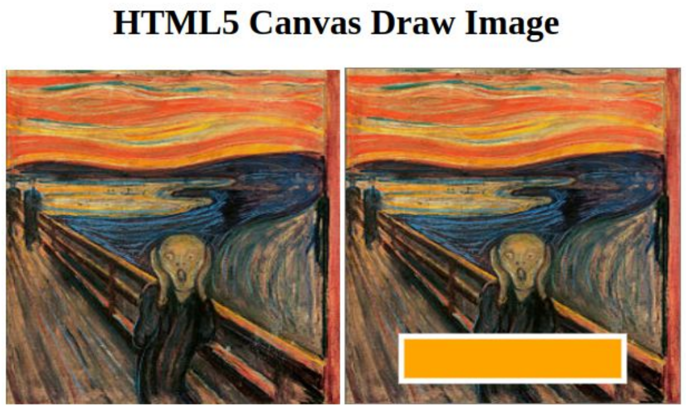
To load an image, you can use the **Image** object and set its **src** property to the image URL. Once the image is loaded, you can draw it on the Canvas using the **drawImage()** method. Here's an example:

```
const image = new Image();
image.src = 'image.png';
image.onload = function() {
  context.drawImage(image, x, y);
};
```

In this code snippet, we create a new **Image** object,
set its **src** property to the image URL,
define a callback function for the **onload** event.
Inside the function, we use the **drawImage()** method to draw the image on the Canvas at the specified coordinates (x, y).

Once an image is on the canvas, it is then possible to draw on it. The following example loads an image and draws a region in preparation for eventually adding a caption:

```
<!DOCTYPE html>
<html><body>
<h1>HTML5 Canvas Draw Image</h1>
<img id="scream" width="300" height="300" src="scream.jpg" alt="The Scream">
<canvas id="myCanvas" width="300" height="300" style="border:1px solid grey;"></canvas>
<script>
  const canvas = document.getElementById("myCanvas");
  const context = canvas.getContext("2d");
  const img = document.getElementById("scream");
  context.lineWidth = 5;
  context.drawImage(img,0,0,300,300);
  context.beginPath();
  context.lineWidth = 5;
  context.fillStyle = "orange";
  context.strokeStyle = "white";
  context.rect(50,240,200,40);
  context.fill();
  context.stroke();
</script>
</body>
</html>
```



Text Support for canvas

In browsers that supported early forms of the canvas element, text was not well supported in a drawing, if at all. Per HTML5, text functions should now be supported by the canvas API, and several browsers already do support it.
Adding Text to the Canvas using **fillText() and strokeText()**:

- The **fillText(text, x, y [,maxWidth])** method is used to fill the specified text at the specified coordinates
- The **strokeText(text, x, y [,maxWidth])** method only draws the text outline at the specified coordinates

Both functions take an optional last parameter, **maxWidth**, that will cut the text off if the text width is longer than maxWidth. Which can be achieved as follows :

```
const canvas = document.getElementById("myCanvas");
const context = canvas.getContext("2d");
context.fillStyle = "blue";
context.strokeStyle = "black";
context.fillText("Canvas is great!", 10, 40);
context.strokeText("Canvas is great!", 10, 80);
```

<u>Customizing Text</u> :

To get more-customized text, you can use the **font property**, which you set identically to a CSS font property.

You can use **textAlign** and **textBaseline** to set the horizontal and vertical alignment of the text string.

<u>textAlign property</u> has the possible values of start, end, left, right, and center.

<u>textBaseline property</u> can be set to top, middle, bottom, hanging, alphabetic, ideographic etc.

```
context.font = "bold   30px   sans-serif";
context.textAlign = "center";
context.textBaseline = "middle";
```

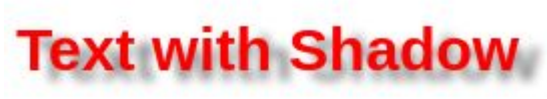
<u>Text Shadow</u> :

We can add shadows to shapes simply by setting the shadow properties : **shadowOffsetX**, **shadowOffsetY**, **shadowBlur**, and **shadowColor**.

- The **shadowoffsets** simply set how far the shadow should be offset from the image. A positive number would make the shadow go to the right and down. A negative number would make it go to the left and up.
- The **shadowBlur** property indicates how blurred the shadow will be
- The **shadowColor** property indicates the color.

This code fragment demonstrates setting a **shadow**.

```
context.shadowOffsetX = 10;
context.shadowOffsetY = 5;
context.shadowColor = "grey";
context.shadowBlur = 5;
context.fillStyle = "red";
context.fillText ("Text with Shadow", 50, 80);
```



CANVAS Animation :

These are the steps you need to take to draw a frame :

1. Clear the canvas : Unless the shapes you'll be drawing fill the complete canvas (for instance a backdrop image), you need to clear any shapes that have been drawn previously. The easiest way to do this is using the **clearRect() method.**
The **CanvasRenderingContext2D.clearRect()** method of the Canvas 2D API erases the pixels in a rectangular area by setting them to transparent black.

Syntax : **clearRect(x, y, width, height)**

x : The x-axis coordinate of the rectangle's starting point.
y : The y-axis coordinate of the rectangle's starting point.
width : The rectangle's width. Positive values are to the right, and negative to the left.
height : The rectangle's height. Positive values are down, and negative are up.

The **clearRect()** method sets the pixels in a rectangular area to **transparent black** . The rectangle's top-left corner is at (x, y), and its size is specified by width and height.

2. Save the canvas state : If you're changing any setting (such as styles, transformations, etc.) which affect the canvas state and you want to make sure the original state is used each time a frame is drawn, you need to save that original state. The **CanvasRenderingContext2D.save()** method of the Canvas 2D API saves the entire state of the canvas by pushing the current state onto a stack.

The drawing state that gets **saved** onto a stack consists of:

- The current transformation matrix.
- The current clipping region.
- The current dash list.
- The current values of the following

  attributes: strokeStyle, fillStyle, globalAlpha, lineWidth, lineCap, lineJoin, miterLimit, lineDashOffset, shadowOffsetX, shadowOffsetY, shadowBlur, shadowColor, globalCompositeOperation, font, textAlign, textBaseline, direction, imageSmoothingEnabled.

3. Draw animated shapes The step where you do the actual frame rendering.

4. Restore the canvas state : If you've saved the state, restore it before drawing a new frame. **CanvasRenderingContext2D.restore() method** of the Canvas 2D API restores the most recently saved canvas state by popping the top entry in the drawing state stack. If there is no saved state, this method does nothing.

Controlling an animation :

Shapes are drawn to the canvas by using the canvas methods directly or by calling custom functions. In normal circumstances, we only see these results appear on the canvas when the script finishes executing. For instance, it isn't possible to do an animation from within a for loop.
That means we need a way to execute our drawing functions over a period of time. We can Schedule these drawing functions with below methods.

1. setInterval() : Starts repeatedly executing the function specified by function **every delay milliseconds**.
The setInterval() method, offered on the Window and Worker interfaces, repeatedly calls a function or executes a code snippet, with a fixed time delay between each call. This method returns an interval ID which uniquely identifies the interval, so you can remove it later by calling clearInterval().

Syntax: **setInterval(function, delay, param1, param2, ...);**

Parameters:

`function`: The function to be executed at each interval. This can be either a function reference or an anonymous function.
`delay`: The time interval between each function execution, in milliseconds. Defaultsto 0 if not specified.
`param1, param2, ...`: Optional parameters that can be passed to the function being executed.

Return value : The returned intervalID is a numeric, non-zero value which identifies the timer created by the call to setInterval(); this value can be passed to clearInterval() to cancel the interval.

Example :

```
setInterval ( function() {
        console.log("This message will be shown every 2 seconds.");
        }, 2000);
```

2. setTimeout() : The `setTimeout()` method is used to execute a function once after a specified delay (in milliseconds). It triggers the function **only once** . The global setTimeout() method sets a timer which executes a function or specified piece of code once the timer expires.

Syntax : **setTimeout(function, delay, param1, param2, ...);**

Parameters:

- **`function`**: The function to be executed after the delay. It can be a function reference or an anonymous function.
- **`delay`**: The time delay in milliseconds before the function is executed. Defaultsto 0 if not specified.
- **`param1, param2, ...`**: Optional parameters that can be passed to the function being executed.

Return value : The returned timeoutID is a positive integer value which identifies the timer created by the call to setTimeout(). This value can be passed to clearTimeout() to cancel the timeout.

<u>Example :</u>

```
setTimeout(function() {
        console.log("This message will be shown after 3 seconds.");
        }, 3000);
```

<u>3. requestAnimationFrame(callback) :</u> Tells the browser that you wish to perform an animation and requests that the browser call a specified function to update an animation before the next repaint.

The **window.requestAnimationFrame()** method tells the browser that you wish to perform an animation and requests that the browser calls a specified function to update an animation right before the next repaint. The method takes a callback as an argument to be invoked before the repaint.

Syntax : **requestAnimationFrame(callback);**

<u>Parameter:</u>

- **`callback`:** The function to be called before the repaint. It should perform the necessary animation updates and rendering.

Return value : A long integer value, the request id, that uniquely identifies the entry in the callback list. This is a non-zero value, but you may not make any other assumptions about its value. You can pass this value to window.cancelAnimationFrame() to cancel the refresh callback request.

Example :

```
function animate() {
    // Animation logic and rendering
    requestAnimationFrame(animate);
}
animate();   // Start the animation loop
```

If you don't want any user interaction you can use the setInterval() function, which repeatedly executes the supplied code. If **we wanted to make a game**, we could **use keyboard or mouse events** to control the animation and use setTimeout(). By setting listeners **using addEventListener()**, we catch any user interaction and execute our animation functions.

HTML5 Canvas - Composition

HTML5 canvas provides compositing attribute globalCompositeOperation which affect all the drawing operations.

We can draw new shapes behind existing shapes and mask off certain areas, clear sections from the canvas using globalCompositeOperation attribute as shown below in the example.

There are following values which can be set for **globalCompositeOperation** :

source-over	This is the default setting and draws new shapes on top of the existing canvas content.
source-in	The new shape is drawn only where both the new shape and the destination canvas overlap. Everything else is made transparent.
source-out	The new shape is drawn where it doesn't overlap the existing canvas content.
source-atop	The new shape is only drawn where it overlaps the existing canvas content.
destination-over	New shapes are drawn behind the existing canvas content.
destination-in	The existing canvas content is kept where both the new shape and existing canvas content overlap. Everything else is made transparent.
destination-out	The existing content is kept where it doesn't overlap the new shape.

destination-atop	The existing canvas is only kept where it overlaps the new shape. The new shape is drawn behind the canvas content.
lighter	Where both shapes overlap the color is determined by adding color values.
xor	Shapes are made transparent where both overlap and drawn normal everywhere else.
darker	Where both shapes overlap the color is determined by subtracting color values.