Web Front-End Development
Week 5: Browser APIs

Application programming interfaces (APIs) provide a defined way to communicate with an application for use by other programs.
There are many types of APIs:

- Programming languages have an API so you can write programs in the language
- Mobile devices have APIs so you can access device data such as location, orientation, or other sensor data
- Operating systems have APIs so other programs can access files, memory, and interact with the screen

In web development we can use JavaScript with two different types of APIs

- Client-side APIs are built into the browser to expose data and functionality from the browser and surrounding computer environment and do useful complex things with it.
    - We've been using the DOM API to interact with the web document
    - Many others https://developer.mozilla.org/en-US/docs/Web/API
    - Today we're going to look at the Canvas API https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API
    - Just like with libraries, it's only through documentation and examples provided that you can figure out how to interact with an API and what you can do with it
- Server-side APIs are provided by companies as a way to interact with their web site/application. We'll be looking at those next week.

Canvas API
https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API
The canvas API was introduced in HTML5 to provide a canvas for 2D drawing
Using JavaScript you can use the Canvas API to dynamically generate and manipulate 2D graphics in the canvas element <canvas>

- Supported in all the latest major browsers
- Default canvas size is 300px wide 150px height
- You can use CSS to position the canvas grid in your page
- A canvas element has no content or border so you can't see it
- Can have more than one canvas element on a page
- Must have an id so you can use document.getElementById() to access it
- Access its drawing context using getContext("2d")
- The context holds all the information about your canvas and lets you draw on it
- The drawing context is where all the drawing methods and properties are defined.
- There is no 3d drawing context yet, but is a possibility in the future
- The WebGL API supports both 2D and 3D and also uses the <canvas> element

Drawing Shapes https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Drawing_shapes

Grid
(slide)

A canvas is a grid where each square represents a single pixel on the screen

- When you draw on the canvas you need to specify the (x,y) coordinates where you want to draw
- (0,0) is the top left corner of the canvas
- Every time you draw on a canvas you create a new layer

https://repl.it/@aileenjp/Canvas-basics
Setup the canvas and context for use throughout the JavaScript file.

```
let thecanvas = document.getElementById("myCanvas"); //canvas
let context = thecanvas.getContext('2d'); //context
```

Rectangles
fillRect(x, y, width, height) Draws a filled rectangle
strokeRect(x, y, width, height) Draws a rectangular outline
You can also use **clearRect(x, y, width, height)** to clears the specified rectangular area

- x and y are the coordinates to start the rectangle
- Width and height indicates the size of the rectangle

Fill refers to the filled in area of a shape
Stroke refers to the outline

```
function rectangle() {
  context.strokeRect(5, 5, 50, 25);
  context.fillRect(100, 100, 25, 50);
}
window.addEventListener("load", rectangle);
```

The load event is fired when the whole page has loaded, including all dependent resources such as stylesheets images.

Styles and Colors https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Applying_styles_and_colors

The **fillStyle** property determines the fill color, pattern, or gradient

- Default is black

The **strokeStyle** property determines the stroke color, pattern, or gradient

- Default is black

The **globalAlpha** property determines the alpha or transparency

- Values are 0 to 1

Now let's add some color. Notice you must set any styles before you call stroke or fill to draw.

```
  context.strokeStyle="#FF7F00";
  context.strokeRect(5, 5, 50, 25);
  context.fillStyle="#0000FF";
```

```
  context.fillRect(100, 100, 25, 50);
```

Gradients

You can create a linear or radial gradient

- **createLinearGradient(x0, y0, x1, y1)** paints along a line from (x0, y0) to (x1, y1)
    - Linear gradient goes between two points
- **createRadialGradient(x0, y0, r0, x1, y1, r1)** paints along a cone between two circles (0 is the start circle, 1 is the end circle)
    - Radial Gradient paints along a cone between two circles

We can then assign this object to the `fillStyle` or `strokeStyle` properties to use for a rectangle or line.

Color stops add color to your gradient

- **grad.addColorStop(position, color)** adds color to your gradient
- position should be a value between 0 and 1

```
function gradient() {
  let grad;
  let grad2;
  let grad3;
  //Because the y values are both 0, this gradient will shade evenly
across the x axis
  grad = context.createLinearGradient(0, 0, 150, 0);
  grad.addColorStop(0, "purple");
  grad.addColorStop(1, "white");
  context.fillStyle = grad;
  context.fillRect(0, 0, 150, 125);
  //Because the x values are both 0, this gradient will shade evenly
across the y axis
  grad2 = context.createLinearGradient(0, 150, 0, 275);
  grad2.addColorStop(0, "blue");
  grad2.addColorStop(1, "white");
  context.fillStyle = grad2;
  context.fillRect(0, 150, 150, 275);
//radial
  grad3 = context.createRadialGradient(75, 400, 10, 75, 400, 75);
  grad3.addColorStop(0, "orange");
  grad3.addColorStop(1, "white");
  context.fillStyle = grad3;
  context.fillRect(0, 300, 150, 425);
}
```

(change the event listener to call gradient)

Paths

A path is a list of points, connected by segments of lines that can be of different shapes, width, and color. You can make shapes using paths.

- First you define your path, like planning it in pencil
- **beginPath()** creates a new path
- Drawing commands are directed into the path and used to build the path up
- **moveTo(x, y)** moves the pencil to the specified starting point.
- **lineTo(x, y)** draws a line to the specified ending point.
- Use these to build your path, but you won't see anything yet
- **closePath()** closes the path
    - When you call fill(), any open shapes are closed automatically, so you don't have to call closePath(). This is **not** the case when you call stroke().
- **stroke()** draws the shape you built in your path by stroking its outline
- **fill()** draws a solid shape by filling the path's content area.

```
function line() {
  // Filled triangle
  context.beginPath();
  context.moveTo(25, 25);
  context.lineTo(105, 25);
  context.lineTo(25, 105);
  context.closePath();
  context.fillStyle = "#0000FF";
  context.fill();

  // Stroked triangle
  context.beginPath();
  context.moveTo(125, 125);
  context.lineTo(125, 45);
  context.lineTo(45, 125);
  context.closePath();
  context.strokeStyle = "#FF7F00";
  context.stroke();
}
```

When the current path is empty, such as immediately after calling beginPath(), or on a newly created canvas, the first path construction command is always treated as a moveTo(), regardless of what it actually is. For that reason, you will almost always want to specifically set your starting position after resetting a path.

Images

https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Using_images

Using images in a canvas is basically a two step process:

1. Get a reference to the image source (can be a URL)
2. Draw the image on the canvas

- **drawImage(image, dx, dy)** takes an image and draws it on the canvas
- **drawImage(image, dx, dy, dw, dh)** takes an image, scales it and draws it
- **drawImage(image, sx, sy, sw, sh, dx, dy, dw, dh)** takes an image, clips it, scales it, and draws it
    - The coordinates (dx, dy) will be the upper-left corner of the image
    - scales it to dimensions (dw, dh)
    - clips it to the rectangle (sx, sy, sw, sh)
- If you try to call drawImage() or createPattern() before the image has finished loading, it won't do anything. You need to be sure to use the load event so you don't try this before the image has loaded

```
let atlas = new Image();
atlas.src = "images/ATLS.png";

function image() {
  context.drawImage(atlas, 0, 0);
}
```

You can create a pattern by setting the style to repeat an image **createPattern(image, "repeat")**

```
function image2() {
  context.fillStyle = context.createPattern(atlas, "repeat");
  context.fillRect(0, 0, 800, 800);
}
```

Arcs
https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Drawing_shapes#arcs
You can draw arcs, circles, and more complex shapes using **arc(x, y, radius, startAngle, endAngle, anticlockwise)**

- x and y are the coordinates of the center of the circle on which the arc should be drawn
- radius is the radius of the arc
- startAngle and endAngle define the start and end points of the arc in radians (not degrees)
- anticlockwise is a Boolean value
    - true draws the arc anticlockwise
    - false draws the arc clockwise
- Like moveTo() and lineTo, the arc() method is a “pencil” method. To actually draw the circle, we need to set the strokeStyle and call stroke() to trace it in “ink.”
- pi is the ratio of a circle’s circumference to its diameter. C=PI*d=2PI*r)
- You can use the Math module that’s built into JavaScript to calculate radians.
    - 180 degrees=Math.PI
    - radians=(Math.PI/180)*degrees

```
function smile() {
  context.beginPath();
  context.arc(75, 75, 50, 0, Math.PI * 2, true); // Outer circle
  context.moveTo(110, 75);
  context.arc(75, 75, 35, 0, Math.PI, false);  // Mouth (clockwise)
  context.moveTo(65, 65);
  context.closePath();
  context.lineWidth = 2;
  context.strokeStyle = "red";
  context.stroke();
  context.beginPath();
  context.arc(60, 65, 5, 0, Math.PI * 2, true);  // Left eye
  context.moveTo(95, 65);
  context.arc(90, 65, 5, 0, Math.PI * 2, true);  // Right eye
  context.closePath();
  context.fillStyle = "red";
  context.fill();
}
```

Text
https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Drawing_text
You can also draw text on your canvas using the **fillText(text, x, y)** or **strokeText(text, x, y)** methods to draw the text

- font can be anything you would put in a CSS font rule
    - font style, font variant, font weight, font size, line height, and font family.
    - You can use % and em for font
- textAlign controls text alignment
    - textAlign is similar (but not identical) to a CSS text-align rule.
    - Possible values are start, end, left, right, and center.
- textBaseline controls where the text is drawn relative to the starting point
    - possible values are top, hanging, middle, alphabetic, ideographic, or bottom.

```
  context.font="bold 1em sans-serif";
  context.fillText("Smile!", 50, 150);
```

Animations
https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Basic_animations
Simple animation is done with the following steps:

1. clear the current canvas
2. move and draw the shape

The **setInterval(function, milliseconds)** function calls a function at a set interval of time (in milliseconds)

The **clearInterval()** function clears the timer

Let's create a bouncing ball by drawing a circle and then continuously move it using a timer.

```
let x = 100; //x position
let y = 200; //y position
let dx = 5; //x position change
let dy = 5; //y position change

function ball() {
  setInterval(draw, 10); // call draw every 10 milliseconds
}

function draw() {
  context.beginPath();
  context.fillStyle = "#0000ff";
  // Creates a circle of radius 20 at the coordinates 100,100 on the
canvas
  context.arc(x, y, 20, 0, Math.PI * 2, true);
  context.closePath();
  context.fill();
  x = dx + x;
  y = dy + y;
}
```

Note that the variables are outside of the function and therefore global.
What do you think the problem is?
We need to clear the canvas before drawing a new circle.
Add at the beginning of draw()

```
context.clearRect(0,0, 600,600); //clears previous circles
```

What do we need to do to create a bouncing action?
Check if x and y are outside of the dimensions and reverse them.
Add before incrementing x and y.

```
  if (x < 20 || x > 580) {
    dx = -dx;
  }
  if (y < 20 || y > 580) {
    dy = -dy;
  }
```

<u>More Canvas</u>
You can do a lot of other things in canvas

- Transformation methods include rotation, scaling, transformation and translation.

- Image capture from images, videos, other canvas elements.
- Access individual pixels and manipulate them.
- Save your canvas to a file.