ESAP Lab: Javascript and Canvas

Lab Brief

The purpose of this lab is to learn the basics of Javascript and Canvas. Javascript is a programming language used to make your website interactive in response to clicking on buttons, filling out forms , animation, games, etc. Canvas is a means to make 2d graphics using Javascript and the `canvas` HTML element.

We will create a basic drawing application by first laying it out in HTML and CSS, then getting into the details of making it interactive with Javascript and Canvas. Our app will allow users to draw on a canvas using their mouse, change the color and thickness of the stroke, switch between a pencil and an eraser, and clear the canvas.

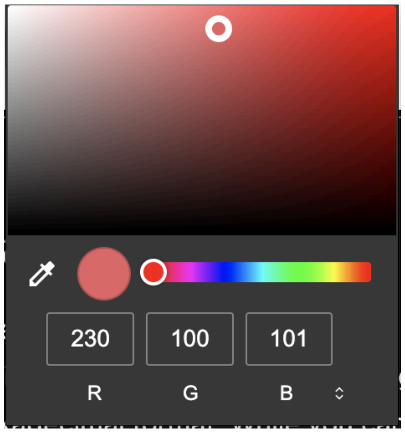
Setting up HTML

1. Open your code editor, create a new file called `paint.html`.
2. Create the `html` element and within it, create `head` and `body` elements.

In the following steps we'll build out `paint.html` with (for now, non-functional) HTML elements that users will be able to interact with in the drawing application.

Create a color picker

Let's add an element that allows the user to choose the color that they want to draw with, such as this color picker:



1. In the body element of paint.html simply create an input element of type color, with an id and value specifying the default color. The ID can be “colorPicker”, and the default color can be black (“#000000”).

Create a line thickness slider

Next we'll allow the user to choose the **line thickness** of their drawing tool using a range slider.

1. In body create an input element of type range, with an id, min, max and a default value. The ID can be “colorPicker”, the “min” can be 1 (i.e, the default line width will be 1 pixel), the “max” can be 10, and the initial “value” can be 3.

Create a "clear all" button

We'll add a button element that allows the user to clear the canvas.

1. This will be a button element. In body, create a button for "Clear All". Give that button an ID like “clearButton”.

Create a canvas

Now we'll create the actual canvas area that the user can draw in.

1. Create a canvas element that defines an area within which you can draw.
2. Specify an id and width and height attributes for how large you want the canvas to be. Give the canvas an ID of “canvasElement”, a width of 800 and a height of 600.

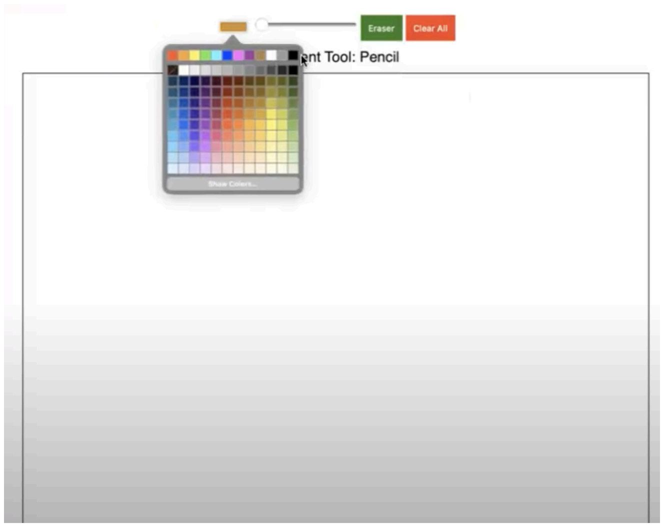
Now open the paint.html endpoint in your browser and make sure that all the elements appear on the browser page.

CSS Styling

Now let's make the elements look nicer in the browser with some CSS.

1. Create a `style.css` file where we will add styling to the elements defined in `paint.html`.
2. In the `head` element in `paint.html`, add a link to the `style.css` file you just created: `<link rel="stylesheet" type="text/css" href="style.css">`

We will leave the actual styling in CSS as a further exercise for you. Your drawing application may look something like this:



Javascript

Let's implement the Javascript code that allows the app to be interactive.Create a new file called `paint.js` in the same directory of the `paint.html` and `styles.css` files.

1. In `body` of `paint.html` we need to link to the javascript that will execute using a script element. The script element is used to embed executable code. Specify the `src` attribute to link to `paint.js`:
   `<script src="paint.js"></script>`

Introduction to event listeners

We'll make use of **event listeners** to make the elements respond to **events** that occur in the browser window—like a button getting clicked or a key getting pressed. An event listener is a Javascript function that waits for an event like this and then reacts to it.

Let's set up an event listener that handles the event that the full HTML document has loaded. We do this by using the target's `addEventListener` method, which sets up the function to be called when a specified event is delivered to the target. The basic syntax looks like this: `addEventListener(eventType, listener)` where parameter `eventType` is a string representing the event type to listen for, and `listener` is a Javascript function that runs when the event occurs.

1. Attach an event listener to the Document target, listening for the event DOMContentLoaded:
   ```
   document.addEventListener('DOMContentLoaded', function (event) {

   })
   ```
   Breaking this down, once the `DOMContentLoaded` event is fired indicating that the HTML document is loaded, then the Javascript function with the parameter `event` runs. The parameter `event` is the event object with properties that you can reference within the function.
   All of the javascript that you write for this lab will be placed inside this function.

Referencing the Canvas HTML element

Within this listener function, we will nest the rest of the event listeners that will respond to users interacting with the HTML elements of our drawing application. First let's define the variable that references the HTML canvas element.

1. Reference the HTML `canvas` element using document.getElementById(). This is a method on Document that returns the element object whose `id` matches the provided string. Your code may look like:
   ```
   const canvas = document.getElementById("canvasElement");
   ```
2. Next use the getContext() method on `canvas` to get the drawing context for the drawing surface of the `canvas` HTML element. The context type "2d" indicates that the drawing will be in two dimensions:
   ```
   const ctx = canvas.getContext("2d");
   ```

Javascript Variables

Now we'll declare some status variables about the state of the canvas.

   a. Add a boolean for whether the user is drawing.
   ```
   let drawing = false;
   ```
   b. Add a variable referencing the stroke color.
   ```
   let strokeColor =
   document.getElementById('colorPicker').value;
   ```
   Since `getElementById` returns the element object that `'colorPicker'` references, to access the actual value we add the `.value`.

   c. Add a variable referencing the width of the line.
   ```
   let lineWidth =
   document.getElementById('widthSlider').value;
   ```

Context methods and properties

Recall the context object ctx we defined in step 2b. The context object provides methods for drawing paths, boxes, circles, characters, and adding images. It also allows for styling (like setting colors and line widths) and manipulating the graphics (like erasing or overlaying colors). We will use the following methods on the context object:

1. beginPath(): Starts a new path by emptying the list of sub-paths. We call this method when we want to create a new drawing.
2. moveTo(x, y): Moves the pen to the coordinates specified by x and y without drawing anything.
3. lineTo(x, y): Draws a line from the current drawing position to the position specified by x and y.
4. stroke(): Actually draws the path previously defined with methods like `moveTo()` and `lineTo()`.
5. clearRect(x, y, width, height):Clears the specified rectangular area and makes it fully transparent. This is used to clear the canvas.

And the following properties on the context object:

6. strokeStyle: This property of the context is used to set the color, gradient, or pattern used for strokes (outlines). In this code, it changes based on whether we are drawing or erasing.
7. lineWidth: This property sets the width of lines drawn in the future. It's used here to adjust the thickness of the drawing line based on a slider input.

With these variables declared and methods identified, we can use them in the following event listeners, still nested in the outer event listener.

Drawing lines

1. Add an event listener for the event that the mouse moves.
2. Within this event listener function, add an if statement.
    a. If drawing is `true` during mousemove, `ctx.lineTo()` adds a line to the path from the last position to the current mouse position.
    b. `ctx.stroke()` actually renders the path on the canvas, so call it at the end.

At the end of these steps your code may look something like this:

```
canvas.addEventListener('mousemove', function(event) {
      if (drawing) {
            ctx.lineTo(event.offsetX, event.offsetY);
            ctx.stroke();
      }
})
```

Start drawing

1. Add an event listener for the event that the mouse is pressed down, indicating that the user is starting to draw. Hint: look up the `mousedown` event.
2. Within this event listener function, set the boolean for whether the user is drawing to `true`. Then use the method `beginPath()` on the context object to start a new path. The method `moveTo(x, y)` specifies the coordinates to draw the line to. Your event listener function may now look like this:

```
canvas.addEventListener('mousedown', function(event) {
     drawing = true;
     ctx.beginPath();
     ctx.moveTo(event.offsetX, event.offsetY);
});
```

Stop drawing

1. Add an event listener for the event `mouseup` indicating that the user has stopped using the pencil or eraser.
   a. In this event listener function set drawing to `false`.

Try out your simple drawing app in our browser! You will notice that the color picker and linewidth do not yet work but at least you can make some lines on the canvas.

Changing line width

Add functionality such that the pencil width reflects the width chosen in the slider.

1. Add a const variable referencing the line width slider's value:

```
let lineWidthValue =
document.getElementById('widthSlider').value;
```

2. Within the event listener that handles the `mousedown` event indicating that a user is starting to draw, set the context object property `lineWidth` to this variable.
   Your code may look something like this:

```
const lineWidthValue =
document.getElementById("widthSlider").value;
ctx.lineWidth = lineWidthValue;
```

Clearing the canvas

Add functionality such that when the "Clear Canvas" button is clicked, erase everything on the canvas.

1. Add a const variable referencing the clear canvas button:

```
const clearButton = document.getElementById("clear-button");
```

2. Create an event listener that handles the `click` event.
3. Create a listener function that uses `ctx.clearRect` to clear the canvas.

Your code may look something like this:
```
const clearButton = document.getElementById("clear-button");
clearButton.addEventListener("click", function(event) {
     ctx.clearRect(0, 0, canvas.width, canvas.height);
});
```

Now changing the line width and clearing the canvas works.

As a further exercise:

1. Implement changing the color of the pencil.
2. Add a button and functionality to toggle on/off an "erase" mode.
3. Add CSS styling to the drawing application.

Further exercise: CSS Styling

This section provides some guidance if you want to add CSS styling to the drawing application. In the next steps in style.css we will use CSS selectors to target HTML elements and define their style.

Center the body and add margin on top

1. In style.css select body and use the text-align property to center the body elements.
2. Then use the margin-top property to add some margin on top of the body elements.

Style the canvas

You'll notice that the canvas isn't visible because it's transparent. We'll add some styling to delineate the canvas area.

1. Select canvas and use the border property to add a border.
2. To make the canvas element display below the other elements instead of inline, use the display property set to block.
3. Add some margin to the canvas.

Style the eraser and "clear canvas" tools

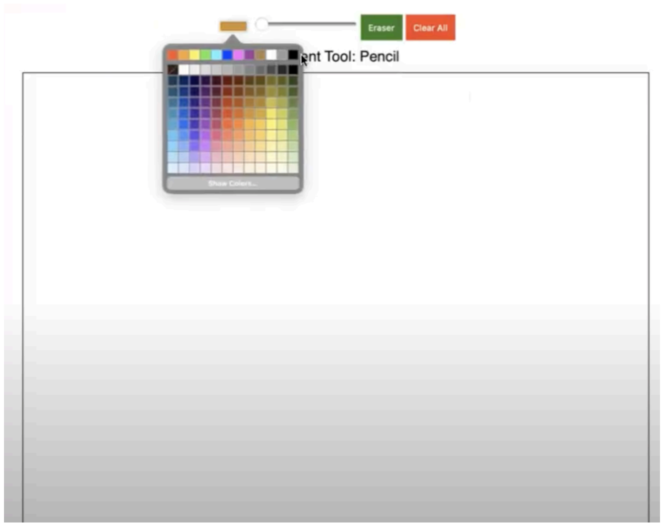
1. Select the eraser element by id, and add some properties to style it:
    a. background-color
    b. color
    c. border

    d. padding
    e. cursor
2. You can do something similar to style the "Clear All" button by specifying the above properties for the element.

Style the tool status

1. Select the tool status element and specify
    a. font-family
    b. font-size
2. Create some spacing between the tool status display and the rest of the buttons with margin-top.

We're done with the CSS styling! Your drawing app may now look something like this:


Eraser
Clear All
nt Tool: Pencil