Painting on an HTML5 Canvas

The title explains it all. Let's create a paint canvas with a color palette and line thickness options!

This exercise is designed as an on-site interview question, which means you don't have hours to complete it. Therefore, we will not use heavy templates, Babel, or any tooling.

Exercise:

Create a webpage, where you can paint on a canvas. The user should be able to select the color and the thickness (pixel) of the drawn line. You may use any HTML5 elements.

Solution:

Different browsers render the same elements differently. To combat this problem, the demonstration of using CSS resets or normalizers is beneficial. Resets remove all element styles, while normalizers make the default styles consistent for as many browsers as possible. I chose to use `normalize.css`. You can get it using

```
npm install normalize.css
```

We can reference this normalizer in our HTML file. Let's create our `index.html` file:

```
<!doctype html>
<html>
    <head>
        <title>Paint - zsoltnagy.eu</title>
        <link rel="stylesheet"
              href="node_modules/normalize.css/normalize.css">
        <link rel="stylesheet" href="styles/styles.css">
    </head>
    <body>
```

```
        <input type="color"  class="js-color-picker  color-picker">
        <canvas class="js-paint  paint-canvas"
                width="600"

                height="300"></canvas>

    </body>
</html>
```

Save the file as `index.html`.

The markup contains a color picker, and the canvas element.

Notice that we created a reference to `styles/styles.css`. Let's create it, and put some border settings around the canvas for clarity:

```
.paint-canvas {
  border: 1px black solid;
  display: block;
  margin: 1rem;
}

.color-picker {
  margin: 1rem 1rem 0 1rem;
}
```

The canvas is now clearly visible, and we can also select the color.

In the JavaScript file, we will first reference our canvas element by selecting the `.js-canvas` element. Notice I used the `.js-` prefix in the class in order to make it clear, this class is used for functionality, not for styling. This is *separation of concerns* in action.

> Using a `.js-` prefixed class in the CSS is discouraged. Imagine a web styler using your `.js-` class to hack some styles in your application. Then one day, the implementer of a feature decides the `.js-` class is not needed anymore. Relying on the naming, he deletes the `.js-` class, breaking the styles. We don't want these things to happen. We also want to give both parties the flexibility of owning their own class names. Styling and functionality are two independent aspects. They should be separated properly.

In the canvas, we can choose between a 2D and a 3D graphical context. We will retrieve the two-dimensional context for drawing on the canvas.

```
const paintCanvas = document.querySelector( '.js-paint' );
const context = paintCanvas.getContext( '2d' );
```

The color picker reference is also needed. Let's use this reference to add a *change event listener* that console logs the chosen color:

```
const colorPicker = document.querySelector( '.js-color-picker');

colorPicker.addEventListener( 'change', event => {
    console.log( event.target.value );
} );
```

As we know the chosen color, we can set the *stroke style* of the graphical context to this color:

```
colorPicker.addEventListener( 'change', event => {
    context.strokeStyle = event.target.value;
} );
```

Let's get to business and start drawing. We need to listen to three events of the canvas:

- `mousedown` indicates that we have to start drawing
- `mousemove` indicates that we have to draw a line from the *last position* where the cursor was to the current position.
- `mouseup` indicates that we have to stop drawing

Why do we need to draw a line from the last position to the current position? Because by just drawing a dot at the current position, our image would depend on the frame rate of the browser. This frame rate is not constant. The garbage collector may start running; your browser may start slowing down; a notification may appear, and so on. We don't want our image to depend on external conditions.

To make the drawing happen, we need to determine the state space of the application. We need to keep track of the `x` and `y` coordinates, and the state of the mouse.

```
let x = 0, y = 0;
let isMouseDown = false;
```

> Thinking about the state space is always an important step when it comes to animation. In more complex examples, you might want to store the position, velocity, and acceleration of objects that may collide. This is where your high school physics studies come in handy.

In a canvas, the top-left point has the coordinates `(x, y) = (0, 0)`, and the bottom-right point has the coordinates `(x, y) = (canvas.width, canvas.height)`. You can get the current mouse coordinates from the `mousemove` event.

Let's implement the three canvas event listeners:

```
paintCanvas.addEventListener( 'mousedown', () => {
    isMouseDown = true;
} );
paintCanvas.addEventListener( 'mousemove', event => {
    if ( isMouseDown ) {
        console.log( event );
    }
} );
paintCanvas.addEventListener( 'mouseup', () => {
    isMouseDown = false;
} );
```

For now, the `mousemove` event only contains a conditional console log. When we execute the code and start logging some values, we may get confused seeing all the different values. Let me give you an example:

```
event
    .clientX: 425
    .clientY: 109
    .layerX: 405
    .layerY: 35
    .offsetX: 409
    .offsetY: 109
    .pageX: 425
    .pageY: 109
    .screenX: 426
    .screenY: 560
    .x: 425
    .y: 109
```

If you randomly select a pair of values that make sense to you, you increase

your chances of failing an interview. Not knowing which value stands for what is entirely acceptable. Just start searching for the answer and move on.

Don't guess. Don't experiment either, because these values are tricky. The fact that `clientX`, `pageX`, and `x` have the same value in this one object, *does not imply* that they are always equal. They are only equal if their definitions say so. Therefore, it's time to look up these definitions.

We can conclude that `x` is indeed equal to `clientX`, because in the documentation of MouseEvent.x, we can read that *"The `MouseEvent.x` property is an alias for the `MouseEvent.clientX` property."* However, after reading a bit more about these properties, it turns out that they are not the ones we need.

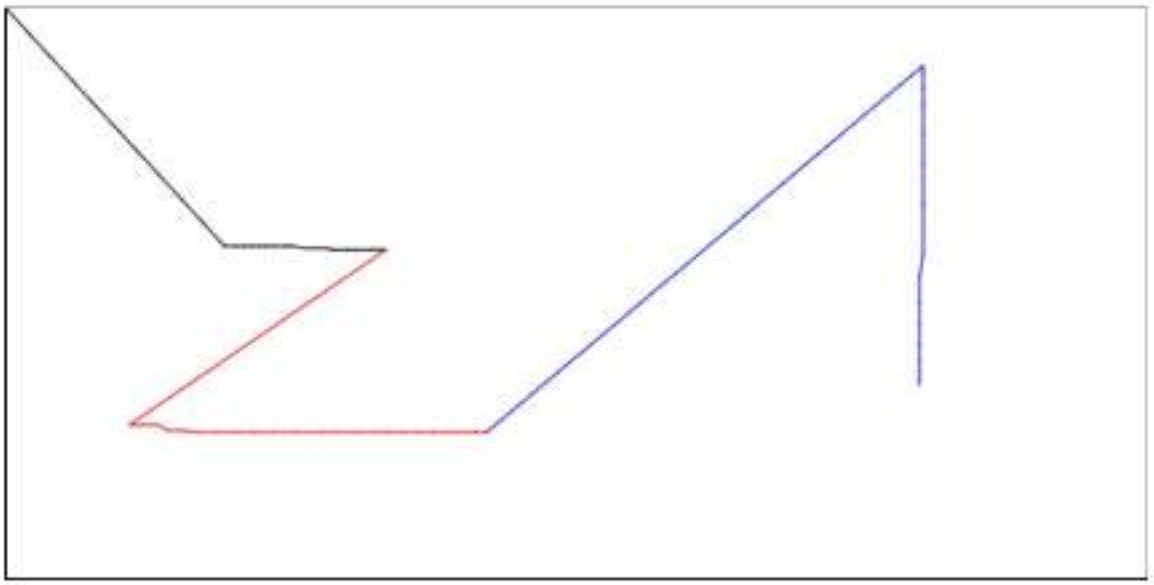
Once we read the documentation of MouseEvent.offsetX, depending on temperament, we could imitate the Backstreet Boys singing *"You're the one I need."* Oh well, they are right about the mouse event, just don't listen to their songs for dating advice.

Now that we know that the current coordinates are `event.offsetX` and `event.offsetY`, and the initial coordinates are saved in our state space as `x` and `y`, we know everything to draw our line:

```
paintCanvas.addEventListener( 'mousemove', event => {
    if ( isMouseDown ) {
        const newX = event.offsetX;
        const newY = event.offsetY;
        context.beginPath();
        context.moveTo( x, y );
        context.lineTo( newX, newY );
        context.stroke();
        [x, y] = [newX, newY];
    }
} );
```

As we start drawing, we can notice a couple of problems.

First of all, when we first start drawing, a line connects the point we first moved to with `(0,0)`. Second, after releasing the mouse, once we start drawing again, a line connects our last drawing with the current one.



Erroneous drawing: the last dot of the previous sequence is connected to the first dot of the new sequence.

Both problems are due to not setting `x` and `y` in the state space to an initial value once we pressed the mouse.

We can solve this problem in multiple ways. One fix involves setting `x` and `y` to `null` whenever we stop drawing. Their initial values should also be `null`. Then, once we start drawing, in the condition, we have to check if `x` and `y` are numbers. If `x` and `y` are null, we ignore drawing.

This solution looks all right on screen, but it is not pixel perfect, because it ignores the very first line of our path. Even if no-one pointed it out, notice that you would have to change the code in at least three different places. Let alone any future modifications in case we wanted to define more events.

We will therefore look for an easier fix. `event.offsetX` and `event.offsetY` are also available inside the `mousedown` event. Therefore, we can initialize the value of `x` and `y` there.

```
paintCanvas.addEventListener( 'mousedown', event => {
    isMouseDown = true;
    [x, y] = [event.offsetX, event.offsetY]
} );
```

Wow! Drawing is now working like a charm. There is just one task left: the

ability to change the line thickness.

For this purpose, we will use a HTML5 `<slider>` element. We will initialize its value to `1`, and allow our range of thickness between `1Px` and `72Px`.

To read the value of the slider, we will also add a label, displaying the thickness of the line:

```
<!doctype html>
<html>
    <head>
        <title>Paint - zsoltnagy.eu</title>
        <link rel="stylesheet"
              href="node_modules/normalize.css/normalize.css">
        <link rel="stylesheet" href="styles/styles.css">
    </head>
    <body>
        <input type="color" class="js-color-picker">
        <input type="range"
               class="js-line-range"
               min="1"
               max="72"
               value="1">
        <label class="js-range-value">1</label>Px
        <canvas class="js-paint  paint-canvas"
                width="600"
                height="300"></canvas>
    </body>
</html>
```

In the JavaScript code, after getting the reference of both objects, we will listen to an event of the slider. If we used the change event, we would get an unwanted surprise: the change event only fires once we release the slider. After looking up the documentation, you can find the `input` event, which fires upon changing the value of the slider.

```
const lineWidthRange = document.querySelector( '.js-line-range' );
const lineWidthLabel = document.querySelector( '.js-range-value' );

lineWidthRange.addEventListener( 'input', event => {
    const width = event.target.value;
    lineWidthLabel.innerHTML = width;
    context.lineWidth = width;
} );
```

As we start drawing a thicker line, another unwanted phenomenon occurs: whenever we make curves, our lines do not form a curvy path. We can see some irregularities instead.

This is because we have not set up the `lineCap` property of the graphical context to `round`:

```
context.lineCap = 'round';
```

This wraps up the canvas painter exercise. We can draw a line of any color and any thickness ranging from `1px` to `72px`.

Play around with it.

If you are thorough, you might have done the following scenario: press the mouse button on the canvas and start drawing. Without releasing the mouse button, exit the canvas. Release the mouse button outside the canvas. Scroll back to the canvas with your mouse button in the released state. Surprise: drawing continues.

This is because we modeled the mouse button in our internal state, but we never took care of the mouse release if it happened outside the scope of the canvas.

How can we fix this?

We set the `isMouseDown` method to `false` in the `mouseup` event of `paintCanvas`.

```
paintCanvas.addEventListener( 'mouseup', () => {
    isMouseDown = false;
} );
```

A dirty trick might inspire you to change the `paintCanvas` to `document`. If you do this and test your code shallowly, you might even succeed to a certain extent.

However, dirty tricks often get caught. In a Windows machine, for instance, you can start drawing, then press Alt+tab to change the currently active task, and click your current window. After the click, your mouse button is in the released state, and we keep drawing.

A proper fix entails listening to the `mouseout` event of `paintCanvas`. For the sake of maintainability, we can also refactor the handler function to indicate

that `mouseup` and `mouseout` handlers take care of the same piece of

functionality. This way, other people maintaining your code will not forget adding their fix to one of the event handlers:

```
const stopDrawing = () => { isMouseDown = false; }
paintCanvas.addEventListener( 'mouseup', stopDrawing );
paintCanvas.addEventListener( 'mouseout', stopDrawing );
```

To make our code more semantic, we can do this refactoring for all our drawing event handlers.

```
const startDrawing = event => {
    isMouseDown = true;
    [x, y] = [event.offsetX, event.offsetY];
}
const stopDrawing = () => { isMouseDown = false; }
const drawLine = event => {
    if ( isMouseDown ) {
        const newX = event.offsetX;
        const newY = event.offsetY;
        context.beginPath();
        context.moveTo( x, y );
        context.lineTo( newX, newY );
        context.stroke();
        [x, y] = [newX, newY];
    }
}

paintCanvas.addEventListener( 'mousedown', startDrawing );
paintCanvas.addEventListener( 'mousemove', drawLine );
paintCanvas.addEventListener( 'mouseup', stopDrawing );
paintCanvas.addEventListener( 'mouseout', stopDrawing );
```

Refactoring is always great. For instance, suppose an angry customer comes to you with a complaint. He says, our canvas is a piece of crap, because if we click without moving the mouse, nothing is drawn on the screen. As you examine the code, you may already be grateful for the refactoring step above, because the request can be handled by adding just one line to `startDrawing`:

```
const startDrawing = event => {
    isMouseDown = true;
    [x, y] = [event.offsetX, event.offsetY];
    drawLine( event );
}
```

Problem solved!

Check out the final result in this CodePen.