Practise Exercise

Sketch

We are going to learn about echa-a-sketch that you can control by moving the arrow keys around. Topics that we are going to include covers DOM elements and Keyboard events. We will be learning about canvas in JavaScript and switch stateents. Lets see an example first lets see the HTML code as below.

```
<body>
<div class="canvasWrap">
<canvas width="1600" height="1000" id="etch-a-sketch"></canvas>
<div class="buttons">
 <button class="shake">Shake!</button>
</div>
</div>
 <script src="./etch-a-sketch"></script>
 <style>
body {
min-height: 100vh;
display: grid;
align-items: center;
justify-items: center;
background: white;
background:
url(https://s3.amazonaws.com/media.locally.net/original/HABA_ALT_
2017-08-02-13-22-45.jpg);
background-size: cover;
}
canvas {
border: 30px solid #e80000;
```

```
border-radius: 10px;
/* Set the width and height to half the actual size so it doesn't look pixelated */
width: 800px;
height: 500px;
background: white;
}
canvas.shake {
animation: shake 0.5s linear 1;
}
@keyframes shake {
10%, 90% {
transform: translate3d(-1px, 0, 0);
}
20%, 80% {
transform: translate3d(2px, 0, 0);
}
30%, 50%, 70% {
transform: translate3d(-4px, 0, 0);
}
40%, 60% {
transform: translate3d(4px, 0, 0);
}
}
</style>
</body>
```

We are going to start by writing pseudo code for what we need to do within the JavaScript file. If you never seen canvas before think something of Microsoft paints. Rectangles, drawing, circles, lines etc., all

of those things are available to us in canvas they are used for when you want to programmatically draw something in browser. If you inspect canvas elements in dev tools will see regular old canvas element. Grab the canvas as we have given the id of canvas as etch-a-sketch.

const canvas = document.querySelector(“#etch-a-sketch”);

Next, we need grab the context. A canvas is an element and the context is the place we are going to draw the canvas is called context. There are 2 types of context 2D and 3D at present we are learning 2D context.

const ctx = canvas.getContext(‘2D’);

Next, we need shake button. The shake button as the class of one we need to grab it.

const shakeButton = document.querySelector(“.shake”);

We have all the elements we want, we need to setup our canvas for drawing. We will set a couple of defaults on the canvas.

ctx.lineJoin = ‘round’;

ctx.lineCap = ‘round’;

That 2 things ensure us that we will be having smooth drawing. By default, we will be getting a square off edge which is not that pleasant. We can set the width of the line as wider as we need but by default, its one pixel. We want it to be thicker so set the lineWidth to be 10 (you don't have to specify the pixel value).

ctx linewidth = 10;

Refresh the page and open the console and type ctx so we can have a look at ctx. We will get CanvasRenderingContext2D this is our canvas object. We want it to be thicker so set the lineWidth to be 10 (you don't have to specify the pixel value). Inside are all these different properties that you can set or get from them, including what color the fill will be, the stroke will be, etc. Next you have to put the drawer somewhere, meaning you have to put the dot somewhere because that will be the starting point of hte etch-a sketch. When ever you load the page you will see the dot in randomly place.

First, we will try to place the dot, then we will randomize it. Take the ctx and run something called beginPath() which will start the drawing. The method beginPath() is something the starting point if you think that you are drawing with some marker there will be some starting where you need to place the marker and start drawing that point to start is beginPath(). Next move the context ctx.moveTo(200, 200); That will place the dot 200 pixel from the left and 200 pixel top. Run the following

ctx.lineTo(200, 200);

ctx.stroke(); will draw a line between where you started and where you drew your line to. That will sort to make an invisible line on the canvas that will stroke it. Now if you refresh the page, you will see a dot 200 pixels over and 200 pixels down. What is cool about an Etch-a-Sketch is that it can start in a random spot.

How can you pick a random spot in this Etch-a-Sketch?

You need to take the width and height of the canvas and generate a random number between 0. How to take the width and height of the canvas.

const {width, height} = canvas this is called destructing. It means you are taking width and height property of canvas and placing it in the width and height variables.

Why are you grabbing the width and height variables?

It is easy to use when doing math. It's much easier than writing canvas.width every time. Now we need to create a random X and Y starting points on canvas.

So how do you create random values in JavaScript?

You may have seen that we have **Math.random()**. If you type that into the console, you will see that it gives us a random number that seems to always be underneath one.

Can we just pass a random number?

Could we do something like Math.random(100)? No, that doesn't do anything. Math.random returns a floating-point (that means it contains decimals), pseudo-random number in the range of 0 to 1, inclusive of 0 but not 1. That means there is a possibility that you will get zero, but you will never get one. It doesn't take any arguments or anything like that. What you can do is take the Math.random value that it gives you and multiply it by 100, you are going to get a random number between zero

and 99.99999. Then we can do is Math.floor(Math.random() * 100); which will give you the number between 0 and 99. So modify the code as.

const {width, height} = canvas;

let x = Math.floor(Math.random() * width);

let y = Math.floor(Math.random() * height);

Replace the x and y values with ctx moveTo and lineTo.

ctx.begin();

ctx.moveTo(x, y);

ctx.lineTo(x, y);

ctx.stroke();

For now, whenever you refresh the page you get a random x and y value that moves the dot around the page. If you change ctx.lineCap = 'round' to ctx.lineCap = 'square' the canvas will change to square. But for now we will be using round so let it be round. Now we need write the handler for working with the arrow keys. You need to know whether to move the line up or down or left or right. Make a function called handleKey listen for the arrow keys using keydown event listener as below.

Function handleKey(){

Console.log('Handle Key');

}

windows.addEentListener("keydown", handleKey);

Refresh the index.html page and open the console. Click on the HTML page and try pressing the arrow keys (up, down, left, right) and you should see the console logging "HANDLING KEY" every time you press a key. You might notice that when you hit the arrow keys, the page is also scrolling. You might notice that when you hit the arrow keys, the page is also scrolling.

That is because it's a default. You probably don't want to turn off the scroll default unless you were accounting for the page being loaded on a very small window but if you did want to, you could just pass the event to the handleKey function like so handleKey(e) and call e.preventDefault(); on it. If you refresh the HTML page, and then hit COMMAND + R on your keyboard, that should refresh the page. However, when you try it, you will notice that it does not refresh the page.

That is because you called preventDefault() on every keydown event. Comment out the e.preventDefault() line of code within the handleKey() function and then manually refresh the HTML page.

```
function handleKey(e) {
// e.preventDefault();
console.log(e.key);
console.log("HANDLING KEY");
}
```

Now check whether the key contains the word arrow. To check the key lets write and if statement as below.

```
if(e.key.includes(“arrow”)){
e.preventDefaulf();
console.log(e.key);
console.log(‘Handling Keys’);
}
```

Now, if you refresh the HTML page, if you use any of the arrow keys, you will see it logged in the console, but if you press any other key, it won't. You should now be able to press command + R to refresh your html page using the keyboard shortcut. That was just the handler. Now you need to hand off the key to the draw function. Instead of having a key passed as the first argument, it will take in an options object. That option object contains everything you wish to pass to the draw function as shown below.

```
function draw(options) {
console.log(options.key);
}
```

You will be passing in an object that has a key property contains e.key like so.

```
function handleKey(e){
If(e.key.includes(“Arrow”)){
e.preventDefault();
draw({key: e.key});
```

```
console.log(e.key);
console.log('Handling Keys');
}
}
```

At this is point we have following

1. Event Listener, which listens for the keydown and runs handleKey.

```
Function handleKey(){
Console.log('Handling Keys');
}
Windows.addEventListener('keydown', handleKey);
```

2. handleKey checks wheather the key is Arrow if it is, passes that along the draw function.

```
function handleKey(e){
If(e.key.includes("Arrow")){
e.preventDefault();
draw({key: e.key});
console.log(e.key);
console.log('Handling Keys');
}
}
```

3. draw has an object and inside of that there is one property key which tell us the key is arrow up.

similar to how we used destructing in an earlier example, is using destructuring in function declarations. We could destructure the options object directly into a key variable like so.

```
function draw({key}){
console.log(key);
}
```

Those curly braces are called as object destructuring where you can take properties and rename them into actually variables and it allows you to have shorter variable names. If you refresh the HTML page and press the arrow keys, you will see they are being logged. Now, when someone goes ahead and uses their keys, you can start to draw directly onto the canvas. Just like you have created ctx.beginPath earlier in the JavaScript file, call call beginPath on the context within the draw function.

```
function draw({key}){

console.log(key);

ctx.beginPath();

}
```

Next you need to move the x and y depending on what the user did.

Add the following:

```
function draw({key}){

console.log(key);

ctx.beginPath();

ctx.moveTo(x, y);

x -= 10;

y -=10;

ctx.lineTo(x, y);

ctx.stroke();

}
```

What happening is any time you press an arrow key we are moving 10 pixels from x and 10 pixels from y, and then we paint. That's obviously not what we want but this is just to demonstrate that it is responding to our key pressed. Add the following towards the top of the file `const MOVE_AMOUNT = 10;` When it is a true constant, meaning that value will never change, we tend to make it uppercase and use underscores. Now, go through our etch-a-sketch.js file and everywhere that you have

used 10, and replace it with MOVE_AMOUNT. Remove the following lines from the draw function.

```
x -= MOVE_AMOUNT;

y -= MOVE_AMOUNT;
```

Instead run a switch statement. However a switch statement is a lot more readable. Add a case for 'ArrowUp'. If key equals 'ArrowUp', we will move the y 10 pixels like so y = y - MOVE_AMOUNT;.

```
switch (key) {

case "ArrowUp": y -= MOVE_AMOUNT;

 break;

default:

 break;

}
```

You have only implemented the case for ArrowUp. If you refresh the page and hit the arrow up key, you should see a line being drawn!

1. For ArrowRight, you will increment the x value.
2. For ArrowDown the y value will be incremented.
3. For ArrowLeft, the x value will be decremented.

```
switch (key) {

case "ArrowUp":

y -= MOVE_AMOUNT;

break;

case "ArrowRight":

x += MOVE_AMOUNT;

break;

case "ArrowDown":
```

```
y += MOVE_AMOUNT;

break;

case "ArrowLeft":

x -= MOVE_AMOUNT;

break;

default: break;

}
```

Now what you want to do is modify the code so the colour of the line changes as you go along. That is actually very simple to do, you just have to change the HSL values. HSL is actually changing the colour in the browser. It is similar to Hex Codes and RGB, but HSL is a cool way to do it. Above the draw() function, make a new variable called hue and set it to 0. Next update the stroke color to start at a bright green like so

```
const hue = 0;

ctx.strokeStyle = `hsl(100, 100%, 50%)`;
```

Now instead of hard coding the first argument to hsl, interpolate it to pass in the hue variable like so

```
ctx.strokeStyle = `hsl(${hue}, 100%, 50%)`;
```

The line will always be red, but what you will do is every time that you use the draw function, increment hue by one and update the stroke style.

```
/ write a draw function

function draw({ key }) {

// increment the hue

hue += 1;

ctx.strokeStyle = `hsl(${hue}, 100%, 50%)`;

}
```

You need call strokeStyle again, even though you set it on page load to be a variable. That is done once the page loads, it's not like a live variable that will update itself. You can also try to play around with it and do something like this

ctx.strokeStyle = `hsl(${Math.random() * 360}, 100%, 50%);

That will give you cool effects.

Shaking and Clearing the Canvas

The idea for this effect is to be similar to what you do to clear a real etch-a-sketch, you shake it! We will have a shake button that is responsible for triggering this.

Add the following code

function clearCanvas(){

canvas.classAdd('shake');

}

Now, when you refresh the HTML page and run clearCanvas() in the console, you should notice the etch-a-sketch doing a little shake animation. That is because the class of shake is applied to it. Has we written some CSS to create that effect in this code.

canvas.shake {

animation: shake 0.5s linear 1;

}

What we are saying there is when the canvas has a class of shake, run the shake over 0.5 seconds, run it linearly and only run it once. You could run the animation 10 times if you wanted by pass in the 10 instead of 1. If you were to run clearCanvas() again in the console, you will notice nothing happens because the canvas already has a class of shake. Just like we can listen to a click, we can listen to this event called animationend. Within the clearCanvas() function, add an event listener that listens for the animationend event and when that event happens, run a function that will remove the class of shake for the canvas.

```
// clear or shake function

function clearCanvas() {

canvas.classList.add("shake");
canvas.addEventListener("animationend",

function() {

console.log("done the shake!");

canvas.classList.remove("shake");

});

}
```

Now if you refresh the page and run clearCanvas(), you should see the console log and the canvas should shake again. However if you run it again and again from the console, you will notice that "done the shake" is being called many times. The number of times you ran clearCanvas() in the console was the number of event listeners that were being added for the same thing. That is obviously not what you want, there is another argument to addEventListener. The first two arguments specify what event the listener should listen on, and what to do when the event occurs. There is a third arguments object for addEventLIstener, and we have only so far used the capture option.

Event Listener “once” Option

Now you are going to use the option once, which you will set to true.

This causes addEventListener to unbind itself. It will call removeEventListener for you, without having to write any code.

```
// clear or shake function

function clearCanvas() {

canvas.classList.add("shake");
canvas.addEventListener("animationend",

function() { console.log("done the shake!");
canvas.classList.remove("shake");
```

```
}, { once: true }

);

}
```

If you didn't have that option, you would have to call canvas.removeEventListener('animationend' and reference the function within which you are calling removeEventListener, which is a bit of a pain. Thankfully you don't have to do that and can use the options argument to set once to true, which will automatically remove the event listener when the animation is done. If you refresh the page and use the arrows to draw something on the etch-a-sketch and click shake, you might notice the animation happens but the content is not removed from the etch-a-sketch. That is because we forgot to do an entire part, which is clearing the canvas (which is actually very simple).

Modify the clearCanvas() function to add some more code. Right after you add the shake class, and before tiy add the animationend event listener, call clearRect on the canvas context.

What that does is clear part of or all of the canvas.

Tell it to start at 0,0 which is the top left hand corner of the canvas and go for 500 & 500 pixels. Add the following code

```
ctx.clearRect(0, 0, 500, 500);

// clear or shake function

function clearCanvas() {

canvas.classList.add("shake");

ctx.clearRect(0, 0, 500, 500);

canvas.addEventListener("animationend",

function() { console.log("done the shake!");
canvas.classList.remove("shake");

}, { once: true }

);
```

}

You should notice only a rectangle that is 500 by 500 pixels from the top left corner is cleared. Instead of hard coding the 500x500 pixel value, we will replace those with the variables that we created at the very top of the file, width and height.

ctx.clearRect(0, 0, width, height);

Now if you refresh, it should all be cleared. That's it all!

Click Outside Modal

In this topic we will learn about how to check you have clicked outside of an element. Let's look at the HTML we will be working with.

```
<body>

<div class="cards">

 <div class="card" data-description="Wes is cool">

 <img src="https://picsum.photos/200?random=1" alt="Wes Bos">
<h2>Wes Bos</h2>

 <button>Learn more →</button>

 </div>

 <div class="card" data-description="Scott is neat!">

 <img src="https://picsum.photos/200?random=2" alt="Wes Bos">
<h2>Scott Tolinski</h2>

 <button>Learn more →</button>

</div>

<div class="card" data-description="Kait is beautiful!">

 <img src="https://picsum.photos/200?random=3" alt="Wes Bos">
<h2>Kait Bos</h2>

<button>Learn more →</button>

 </div>
```

```
<div class="card" data-description="Snickers is a dog!">

<img src="https://picsum.photos/200?random=4" alt="Wes Bos">
<h2>Snickers the dog</h2>

 <button>Learn more →</button>

</div>

 </div>

<div class="modal-outer">

<div class="modal-inner">

 </div>

</div>

<script src="./click-outside.js"></script>

</body>
```

As you can see, each card has a random image, an h2 and a button that says, "learn more". Let's add the following CSS.

```
<style>

 .cards {

display: grid;

grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));

grid-gap: 20px;

padding: 2rem;

}

.card {

background: white;

padding: 1rem;
```

```
border-radius: 2px;

}

.card img {

width: 100%;

}

 .card h2 {

color: black;

}</style>
```

Moving onto the modal, add some dummy content inside of it so when we style it we can visualize what it will look like.

```
<div class="modal-outer">

 <div class="modal-inner">

 <p>Testing 123</p>

</div>

</div>
```

As you can see, there are two nested divs each with a class ofmodal-inner or modal-outer. Style the modal in the on position and then we will turn it off with CSS.

```
 .modal-outer {

display: grid;

background: hsla(50, 100%, 50%, 0.7);

position: fixed;

height: 100vh;

width: 100vw;

top: 0;
```

```
left: 0;

justify-content: center;

align-items: center;

}

.modal-inner {

max-width: 600px;

min-width: 400px;

padding: 2rem;

border-radius: 5px;

min-height: 200px;

background: white;

}
```

Now you have this modal which will pop up and you want to be able to close it. We need to hide the modal by default. There are different way to do that such as display:none and visibility:hidden.

Closest Method

Closest() is querrySelectorAll(), expect it is opposite where it will climb up the nested tree of DOM elements instead of looking down the DOM tree. Lets see an example.

```
function handleCardButtonClick() {

const button = event.currentTarget;

const card = button.closest('.card');

 // grab the image src

const imgSrc = card.querySelector('img').src;

console.log(imgSrc);
```

}

If you refresh the page and click on the buttons, you should see the image source logged to the console.

Scroll Event and Intersection Observer

The scroll event is when someone goes ahead and scroll on the page or inside element. One thing you are likely to encounter in your career as a developer is building a terms and conditions scroll to accept. That is where the user is forced to scroll all the way to the bottom of the text before the accept button will work. If you want to listen for window scroll you need to use windows.addEventListener();. Select that element and listen for a scroll on it by selecting the terms-and-conditions class.

```
const terms = document.querySelector('terms-and-conditions');
```

Add an event listener to terms on the scroll event and just log the event with the handler.

```
terms.addEventListener('scroll', function(e) {

console.log(e);});
```

If you refresh the page, you will see the following error in your console

scroll-to-accept.js:3 Uncaught TypeError: Cannot read property 'addEventListener' of null at scroll-to-accept.js:3

This is a problem you will run into often. We will creates a function like scrollToAccept and puts all code inside of that function.

Then within that function, after he grabs the selector, he will check if that element exists using a bang, and if it doesn't, he will return so the function exits.

```
function scrollToAccept() {

const terms = document.querySelector('.terms-and-conditions');
if(!terms) {

return;

//quit this there isn't that item on tha page }
terms.addEventListner('scroll', function(e) {
```

```
console.log(e);

})}

scrollToAccept();
```

What that will do is check whether something is found by the querySelector, and if it is, the rest of the code will run as expected and if not, you return from the function which will stop it from running and then it will never run. How do you know if you are scrolled to the bottom? How would you know that 1,828 pixels is the bottom for example? You need to also grab the scrollHeight to figure that out. The scroll height will tell you how high the scrolling thing is.

```
terms.addEventListener('scroll', function(e) {
console.log(e.currentTarget.scrollTop);
console.log(e.currentTarget.scrollHeight);

});
```

Now when you log that, you will see how far from the top you are scrolled and the second number is how high the actual scroll-able div is. When you reach the very end, you should see the values are close. They are not perfectly close and that is because the elements have different CSS styles, one of them has margins and padding. That becomes a pain to work with because you have to work with offset heights and that is a thing of the past. You do not need to do it that way anymore.

Intersection Observer

Rather than figuring out how far the page is scrolled, you can use intersection observer to figure out if something is viewable on the page. Inside of the terms HTML, between one of the paragraphs, we will add a strong tag with a class of watch. We want to know when that strong tag is visible on the page. Grab the watch element at the top of the file.

```
const watch = document.querySelector('.watch');
```

We need to create intersectionObserver().

```
const watch = document.querySelector('.watch');
```

The intersection observer is going to take a **callback**, which is a function that gets called at a certain point.

```
function obCallback(payload) {

console.log(payload);

}

const ob = new IntersectionObserver(obCallback);
```

If you refresh the page nothing happens because we haven't done anything inside the function. Take the observer and call the observe() method on it, and then you pass it something to watch for, such as the strong tag.

```
function obCallback(payload) {

console.log(payload);}

const ob = new IntersectionObserver(obCallback);

ob.observe(watch);
```

Now, every time we you ahead and scroll, you will notice that you get this IntersectionObserver entity logged.

There is also the boolean isIntersecting which will tell you if it is on the page or off as shown below.

```
function obCallback(payload) {

console.log(payload[0].isIntersecting);

}
```