EVENTS – LAB

EVENTS IN JAVASCRIPT

Javascript is often described as 'event driven'. That is the script is called by a range of events that can be either user driven or the result of other 'events' in the page.

ONLOAD AND THE DOM

When a web page loads the browser loads the DOM into memory so that it can be manipulated by Javascript. In previous labs we have added our Javascript code to the bottom of the page before the closing `</body>` tag.

Javascript code can also be added to the top of a web page and is commonly found in the `<head>` of a HTML file. This is done in the same way as when code is placed at the bottom of the file using the `<script>` tag. However, there is one important difference in that when Javascript is placed at the top of the file, the Javascript loads but the DOM has not yet loaded.

As the Javascript will no doubt reference elements in the DOM (through the document object) this can lead to errors as the DOM has not yet loaded.

Therefore, the Javascript must be stalled until the document is loaded. A common way to do this is to use an `onload` event, attached as an inline event to the `<body>` tag.

1. Open the *index.html* file
2. Notice that there is a `<script>` tag in the `<head>`.
3. Notice that there is also a `onload` event as an attribute of the `<body>` tag.

The `onload` event can call a function when the body has loaded - in this example the function is called `init` which is a common convention.

With this technique all the other Javascript code is placed inside this `init` function.

The event here is known as an inline event. Generally, these are discouraged for other events as they can lead to unmanageable code – however the `onload` event on the body tag is still a commonly seen technique.

As we did in previous labs it is recommended that you place your code before the closing `</body>`. In that way, the DOM has already loaded and thus there is no need to worry about an `onload` event.

As such remove the `onload` event and the `<script>` tag that is in the `<head>` as we'll be placing our code elsewhere. Your body tag should now appear simply as:

```
<body>
```

PLACING CODE IN EXTERNAL FILES

As your Javascript files become more complex it can become unwieldly to have all the code in the HTML file. Another option is to place the code in an external file saved with a JS file extension.

In the file structure for the lab there is *scripts* folder with a basic JS file inside called *main.js*.

Before the closing `</body>` attach the *main.js* file with the following:

```
<script src="scripts/main.js"></script>
```

Notice that this uses a standard relative path to locate the file to load.

To test this is worked you should see the message in the console. All Javascript code from now on we'll place in the *main.js* file.

LISTENING FOR EVENTS

The recommended way to work with events is to use the `addEventListener()` method. This is attached to DOM elements that you can identify with the likes of `getElementById()` and `querySelector()`.

We are going to build a simple game where players collect items against the clock.

First we'll need a start button so we'll use the HTML button of id *startGameBtn* and attach an event to it as follows:

```
document.getElementById('startGameBtn').addEventListener('click',
startGame);
```

The 'click' event is now registered with the button such that when it is clicked it will call the function *startGame*. We therefore need to create *startGame* function. To test it create a simple function with a console output ie:

```
function startGame(){
      console.info('Button Clicked');
}
```

Functions called by an event listener are known as event handlers.

DOM ELEMENT VARIABLES

As we'll likely reference the start button again it will be easier to create a variable to store it. Amend your code as follows:

```
var startGameBtn = document.getElementById('startGameBtn');
startGameBtn.addEventListener('click', startGame);
function startGame(){
      console.info('Button Clicked');
}
```

Our game is going to use other elements in the DOM, so it will be useful to create DOM variables for them to. Add the following declarations:

```
var playGround = document.querySelector('.playGround');
var myTimer = document.querySelector('time');
var myFound = document.getElementById('found');
```

Our game is going to place a series of dots in the grid for the player to collect. As such we'll also declare a couple of variables related to the number of dots and their size:

```
var numDots = 10;
var dotSize = 20;
```

To score the game we'll be timing the user and counting how many dots they've collected. This requires two more variables:

```
var timePlayed = 0;
var numFound = 0;
```

CREATING THE DOTS

The dots are going to be created in the grid/playground by using HTML. The dots will just be HTML `<div>` elements appropriately styled.

In the *mobileFirs.css* stylesheet there is a rule as follows:

```
.dot{
	position:absolute;
	width:20px;
	height:20px;
	background-color:#6FC5DF;
	border-radius:8px;
}
```

This class styles the dots. Notice that the CSS uses `position:absolute`. The dots will be created as child nodes of the `div.playGround` which is styled with `position:relative`. This means the dots will be positioned relative to the parent element - the playground.

A test add the following HTML inside of the `div.playGround`.

```
<div class="playGround">
	<div class="dot"></div>
</div>
```

Save and test you file to view the dot.

- Remove the dot added manually above because we are going to add dots via a Javascript loop.

USING A FOR LOOP TO GENERATE THE DOTS

Inside the *startGame* event handler function add the following:

```
for(var i=0; i<numDots; i++){
	console.info('Create Dot');
}
```

Save and test the page to view the console outputs when you press the start button.

Remove the console and replace it with the following to create the dots:

```
var newDot = document.createElement('div');
newDot.setAttribute('class', 'dot');
playGround.appendChild(newDot);
```

This code will generate 20 HTML `<div>`, assign them a class of dot and then append them as children of the `div.playGround`.

Save and test this file. It only looks as if one dot has been created but if you use the Chrome inspector you'll see there are 20 dots but all sat on top of each other.

RANDOMIZE DOT PLACEMENT

To randomize dot placement we'll use `Math.random()`. This generates a random number between 0 and 1.

We want two values, one for the CSS property `left` and one for `top`. These will need to stay within the `div.playGround` so we'll multiple the width and height of the `div.playGround` by our random number. The DOM elements have properties of `offsetWidth` and `offsetHeight` that we can use here.

Add the following to the `for` loop:

```
var dotTop = Math.random()*(playGround.offsetHeight - dotSize);
var dotLeft = Math.random()*(playGround.offsetWidth - dotSize);
newDot.style.top =  dotTop+'px';
newDot.style.left =  dotLeft+'px';
```

Save and test your file. You should have twenty randomly placed dots.

ADD EVENTS TO THE DOTS

The simple game idea is for the user to click on the dots to collect them. That means we need to attach an event to each dot that was dynamically generated. We can do this in the `for` loop as follows:

```
newDot.addEventListener('click', dotClick);
```

We then need to create the event handler *dotClick* function. However, we need to know which dot was clicked. This can be done by using the event object that is sent to the event hander as a parameter. In the following we create the event object as `ev`.

```
function dotClick(ev){
      console.info(ev.target);
}
```

In your console you should see information about the target of the event ie the dot you clicked on.

As we know the target of the event and we know the parent node (playground) we can now use the `removeChild()` method to remove the dot as follows:

```
playGround.removeChild(ev.target);
```

UPDATING THE SCORE

Inside of the *dotClick* event handler function we'll also update the user's score with:

```
numFound++;
myFound.innerHTML = numFound;
```

TIMING THE GAME

To time the player we'll use `setInterval()` which calls a function at a set interval. As we'll want to stop `setInterval()`, as well as start it, we'll create it as a variable that can be used to call the method `clearInterval()` when we want it stopping.

Alongside the previously declared variables add:

```
var clockInterval;
```

Inside the *startGame* event handler add:

```
clockInterval = setInterval(clockCount, 1000);
```

Now create the *clockCount* method as follows:

```
function clockCount(){
      timePlayed++;
      var minutes = parseInt(timePlayed / 60);
      var seconds = timePlayed % 60;
      seconds = seconds < 10 ? "0" + seconds : seconds;
      var displayTime = minutes + ":" + seconds;
      myTimer.innerHTML = displayTime;
}
```

This updates the time played and displays it in the *myTimer* DOM element. The minutes taken is calculated by dividing seconds by 60. The seconds remaining is calculated with the modulus % operator to get the remainder when divided by 60. A ternary operator is used to see if a leading zero is required. A ternary operator is the equivalent of a one line 'if' statement.

GAME END

To end the game add the following to the *dotClick* event handler.

```
if(numFound == numDots){
            clearInterval(clockInterval);
            myTimer.setAttribute('class', 'finished');
            myFound.setAttribute('class', 'finished');
}
```

Here the `clearInterval()` stops the clock. The two `setAttribute()` methods add a class of finished to the score and timer.

TIDY IT UP

To tidy the game up can you:

- Apply an attribute of `disabled` to the start button when the game is playing and remove it on game end.
- Reset the score and time variables when the game is played a second time.
- Adjust the dot sizes so they don't overlap with the edge of the play ground.

MOBILE VERSION

This game will work on your mobile phone. To test it save it to your *f://public_html* folder so that you can view it on the *homepages.shu.ac.uk* server.

For example if you saved the file to:

```
F:/public_html/js-game/
```

The URL would be:

```
http://homepages.shu.ac.uk/~YOURSTUDENTNUMBER/js-game/
```

We could make the game more responsive on a mobile by adding touch events.

First check for touch events with:

```
var isTouch = false;
(function(){
  if('ontouchstart' in window){
        alert('touch');
        isTouch = true;
  }else{
        alert('no touch');
        isTouch = false;
  };
})()
```

Remove the alerts once you have tested it and then amend the *startGame* event handler using:

```
if(isTouch){
      newDot.addEventListener('touchstart', dotClick);
}else{
      newDot.addEventListener('click', dotClick);
}
```

The game should now be (slightly) more responsive on mobile.