JavaScript III

Common JavaScript Objects

- JavaScript has many built-in objects that take the place of libraries in other languages
  - Math
  - Date
  - RegExp

Math

- The Math object has properties that function as constants
    - Math.E
    - Math.PI
- The methods of the math object provide many standard calculations
    - Math.round(x)
    - Math.random()
    - Math.cos(x)
    - Math.pow(x,y)

```
In [ ]: %%script node
        console.log("The area of a circle with radius 3 is " +
                    Math.PI * Math.pow(3,2))
        console.log("")
        console.log("The cosine is (and all trig) expects radians " +
                    Math.cos(Math.PI/2) + " " + Math.cos(0))
        console.log("")
        console.log("Enjoy this random number " + Math.random())
```

Date

- The JavaScript Date object is used to both get the date, and interact with dates
- To get a Date object, you must use the new operator

  ```
  javscript
  var my_date = new Date();
  ```

  - Calling `Date()` with out new returns a string
- If you don't want a full blown object, and just need the current Unix time, call `Date.now()`

In [ ]:

```
%%script node
var now = new Date()
console.log("Today is " + now)
console.log("Today is " + Date())
console.log("The current UNIX time is " + Date.now())
```

Common Date Methods

- The various fields of a date can be accessed and updated using getter and setter methods
  - `getYear(),setYear()`
  - `getMonth(),getDay()` - Both of these start at 0!
- Converting the Date to a string
  - `toDateString()` returns only the date part
  - `toTimeString()` returns only the time part
  - There is no easy way to specify a custom format

In [ ]:
```
%%script node
var now = new Date()
console.log("The date is " + now)
now.setFullYear(1900)
console.log("The date is " + now)
console.log(now.toDateString())
console.log(now.toTimeString())
```

Regular Expressions

- Regular Expressions in JavaScript are almost PCRE
- They can be created using
  - A perl style literal, e.g. /abc/i;
  - By using the RegExp object constructor

In [ ]:
```
%%script node
var my_reg = /\d\d\d-\d\d\d-\d\d\d\d/i;
var another_re = new RegExp("\\d\\d\\d-\\d\\d\\d-\\d\\d\\d\\d",
                            "i");
```

RegExp Methods

- regex.test returns a boolean if at least one match was found
- regex.exec returns an array with various information about the first match found
    - If the regex was created with the `g` flag, successive calls to regex.exec will find additional matches
    - Any capture buffers are also included in this array

In [ ]:

```
%%script node
var my_reg = /(\d\d\d)-(\d\d\d)-\d\d\d\d/g;
var text = "410-555-1234 301-555-1234 443-555-1234";
var result
while((result = my_reg.exec(text)) !== null)
{
    console.log(result)
    text = "443-555-1234 301-555-1234"
}
```

In [ ]:

```
%%script node
var my_reg = new RegExp("(\\w+) .+? \\1 (\\w+)",'g');
var text = "Doe a deer a female deer ray a drop of golden sun me a name I call mys
elf";
var result;
while((result = my_reg.exec(text)) !== null)
{
    console.log(result)
}
```

String

- The string object has two methods that accept regular expressions (in either format)
  - split(separator)
  - replace(old,new)

In [ ]:
```
%%script node
var text = "Doe a deer a female deer ray a drop of golden sun me a name I call mys
elf";
console.log(text.split(/\s/));
console.log(text.split(RegExp("\\w\\w+")))
```

```
In [ ]: %%script node
        var text = "Doe a deer a female deer ray a drop of golden sun me a name I call mys
        elf";
        console.log(text.replace(/(\w)\1/,"**"))
        console.log(text.replace(/(\w)\1/g,"**"))
        console.log(text.replace("ee","**"))
```

Events

- So far the programming we have been doing has focused on a set of instructions to be executed roughly in sequence
- Instead of providing the order to execution when programming, we can define handlers that will be executed in response to a specific event
- Events are sent from a dispatcher
    - In the case of web programming, the dispatcher is the web browser
    - The dispatcher could be hardware based too, like a sensor (see NodeBots)
- The alternative to using events would be to program an instruction to check the status of something through out your code

Event Basics

- In the JavaScript event system, we need three pieces of information to handle an event
    - The object or objects on the page to get events from
        - The entire document, all paragraphs, a specific element, etc.
    - The event we want to handle
        - A mouse click, a key press, the copying of text, etc.
    - The function to call when the event happens, often called the *handler*

Event Propagation

- For this discussion consider the following HTML:

```
<div>
  <p>
      <input type="text"/>
  </p>
</div>
```

Event Propagation

- One event we can listen to is **input** that is fired when the input changes
- This event can be captured from not only a listener on the *input* tag, but on the *p* and *div* tags as well
    - The event moving up the HTML tree is known as event propagation.
    - It is also sometimes referred to as bubbling.
    - The order the handlers will be called in is always the element itself followed by its closest parents
- We can prevent this behavior if we wish

Issues with Event Programming

- A webpage can generate hundreds of events a second
  - It is impossible to process them all as they come in
  - Events are stored in a queue for processing
- Should all events be processed?
  - If you processed every change in the scrollbar position, your page would lock up
- How are events refered to
  - In JS they are strings, so a mistyped event won't cause a syntax or error

The Event Object

- When the handler is executed, it recieves at least one object, which contains properties about the event
- The properties of this object depend on the event that produced it
- The MDN page on events is a good resource for this
- Some common properties are:
  - **screenX** and **screenY** for events involving the mouse
  - **key** or **keyCode** for events from the keyboard
  - **clipboardData** for copy, cut, and paste events

addEventListener

- The **addEventListener** function can be called on any element
    - The `this` inside of a handler refers to the object that triggered the event
- It takes two parameters
    - The event type as a string
    - The handler function

```
element.addEventListener('eventString',
function(event){
doSomething();
})
```

Click

- One of the most commons events is responding to a mouse click
- The event string for this is 'click'

```
In [ ]: %%html
<html>
    <head>
        <script>
             function paragraphHandle(){
                    this.style.background = "green"
                }
            document.querySelector('#one').addEventListener('click',paragraphHandl
e)
            document.querySelector('#two').addEventListener('click',paragraphHandl
e)
            document.querySelector('#three').addEventListener('click',paragraphHan
dle)

            document.querySelector('.contain').addEventListener('click',
                function(){
                    this.style.background = "purple"
                })
        </script>
    </head>
    <body>
        <div class="contain">
            <p id="one"> A Paragraph</p>
            <p id="two"> A Second Paragraph</p>
            <p id="three"> A Third Paragraph</p>
        </div>
```

querySelectorAll Revisited

- Having to put an id on every element that we want to add an event listener to is cumbersome
- **document.querySelectorAll** will return a **NodeList** of the matching elements
- **NodeList** is not an **Array**
    - It does not have **forEach** (reliably across all browsers anyways)
    - We need to loop over it using old fashion C-style loops

In [ ]:

```
%%html
<html>
    <head>
        <script>
             function paragraphHandle(event){
                    this.style.background = "green"
                    event.stopPropagation()

                }
            var paragraphs = document.querySelectorAll('p')
            for(var i = 0; i < paragraphs.length; i++)
            {
                paragraphs[i].addEventListener('click',paragraphHandle)
            }

            document.querySelector('.contain2').addEventListener('click',
                function(){
                    this.style.background = "purple"
                })
        </script>
    </head>
    <body>
        <div class="contain2">
            <p> A Paragraph</p>
            <p> A Second Paragraph</p>
            <p> A Third Paragraph</p>
        </div>
    </body>
</html>
```

Input

- The **input** event is fired when ever the value of on input changes

In [ ]:
```
%%html
<html>
    <head>

        <script>
        document.querySelector("#theText").addEventListener('input',
            function(event)
            {
                var lastChar = this.value[this.value.length -1]
                var out = document.querySelector("#output")
                if(lastChar == 'a' || lastChar == 'e' ||
                   lastChar == 'i' || lastChar == "o" ||
                   lastChar == 'u')
                {
                    out.innerHTML = "You typed a vowel"
                }
                else{
                    out.innerHTML = "You typed a consonant"

                }
            })
        </script>
    </head>
    <body>
        <input id="theText" type="text" />
        <p id="output"></p>
    </body>
</html>
```

Keyboard Events

- To respond to whats typed on they keyboard at any time, not just when the user is in an input field, keyboard events are used
- There are many keyboard events, including **keyup**, **keypress**, **keydown**.
- The implementation of these events is one of the less standardized parts of JavaScript to this day

In [ ]:

```
%%html
<html>
    <head>
        <script>
        document.addEventListener('keydown',
            function(event)
            {
                var output = document.querySelector("#output2")
                if( event.keyCode == 77 && event.ctrlKey == true)
                {
                    output.style.border = "3px solid black"
                }
                else if(event.keyCode == 77 && event.altKey == true)
                {
                    output.style.border = "3px dashed black"
                }
                else{
                      output.style.border = "0px solid black"
                }
            })
        </script>
    </head>
    <body>
        <p id="output2">Watch This Space</p>
    </body>
</html>
```

Blur and Focus

- Certain HTML elements are meant for the user to interact with: buttons, input fields, radios, etc.
- Theoretically all HTML elements can recieve focus, but this is browser dependent
- There are several visual cues in the default styles of browsers to show what event is in focus
  - The soft blue glow around a text box is one
  - See http://medialize.github.io/ally.js/tests/focus-outline-styles/ for examples

In [ ]:
```
%%html
<html>
    <head>
    <script>
        function getInputType(event)
        {
            document.querySelector("#inputType").innerHTML = this.type
        }

        document.querySelector("#in").addEventListener('focus',getInputType)

        document.querySelector("#aSelect").addEventListener('focus',getInputType)

        document.querySelector("input[type='email']").addEventListener('focus',get
InputType)
    </script>
    </head>
    <body>
        <h3>A Simple Form</h3>
        <p id="inputType">The Input Type Will Go Here</p>
        <p>
            <label>Name:</label> <input type="text" id="in"/>
        </p>
        <p>
            <label>Class Standing:</label>
        <select id="aSelect">
            <option>Freshman</option>
            <option>Sophmore</option>
            <option>Junior</option>
            <option>Senior</option>
        </select>
        </p>
        <p>
        <label>Email: </label> <input type="email"/>
        </p>
```

Mouse Movement

- It is possible to fire an event when a mouse enter or leaves an element, or just moves over it in general

```
In [ ]: %%html
<html>
    <head>
        <style>
            #trackHere{width:100px;height:100px; border:1px solid blue;margin:10px
 auto}
        </style>
        <script>
            document.querySelector('#trackHere').addEventListener('mousemove',func
tion(event){
                    document.querySelector("#point").innerHTML= event.screenX +
 "," + event.screenY
                })
        </script>
    </head>
    <body>
        <h1>A Random Heading</h1>
        <div id="trackHere"></div>
        <p id="point"></p>
    </body>
</html>
```

Live Coding Event Example

- From https://eloquentjavascript.net/15_event.html
  - Inflate a balloon 10% when hitting the up arrow
  - Deflate a balloon 10% when hitting the down arrow
  - If the balloon is too big it should pop (💥)
  - Don't scroll!

In [2]:
```
%%html
<p id="bal">🎈</p>

<script>
  // Your code here
</script>
```

🎈

The Event Object Practice

- Write JavaScript to let the user know while they are dragging selected text using the **drag** event and the HTML below

In [ ]:
```
%%html
<html>
    <head>
        <script>

        </script>
    </head>
    <body>
        <div>
            <span>Message:</span> <span id="selected"></span>
        </div>
        <p id="toWatch">The giant anteater (Myrmecophaga tridactyla), also known a
s the ant bear, is a large insectivorous mammal native to Central and South Americ
a. It is one of four living species of anteaters and is classified with sloths in
 the order Pilosa. This species is mostly terrestrial, in contrast to other living
 anteaters and sloths, which are arboreal or semiarboreal. The giant anteater is t
he largest of its family, 182–217 cm (5.97–7.12 ft) in length, with weights of 33–
41 kg (73–90 lb) for males and 27–39 kg (60–86 lb) for females. It is recognizable
 by its elongated snout, bushy tail, long fore claws, and distinctively colored pe
lage.</p>
        <p><small>From the Wikipedia entry on the Giant Anteater</small></p>al and
 South America. It is one of four living species of anteaters and is classified wi
th sloths in the order Pilosa. This species is mostly terrestrial, in c
    </body>
</html>
```

Time Outs

- Calling a handler everytime an event fires for events that fire in rapid suggestion (mousemove, etc.) can freeze the browser
- There is no way to tell the browser not to call the events so often, but we can make sure the computationally expensive code is run only at certain intervals
- We use a function in JavaScript called **setTimeout** to achieve this

In [ ]:

```
%%html
<!DOCTYPE html>
<!-- From Eloquent Javascript-->
<html>
<head>

        <style>
            #trackHereAgain{width:100px;height:100px; border:1px solid blue}
        </style>
<script>
  function displayCoords(event) {
    document.querySelector("#point2").textContent =
      "Mouse at " + event.pageX + ", " + event.pageY;
  }

  var scheduled = false, lastEvent;
  document.querySelector("#trackHereAgain").addEventListener("mousemove", function
(event) {
    lastEvent = event;
    if (!scheduled) {
      scheduled = true;
      setTimeout(function() {
        displayCoords(lastEvent);
        scheduled = false;
      }, 2000);
    }
  });
</script>
</head>
<body>
      <div id="trackHereAgain"></div>
      <p id="point2"></p>
</body>
</html>
```

SetInterval

- Rather than setting a time out in a function, we can request that a function be run every so many milliseconds
- Usually called on the `window` object

In [ ]:
```
%%html
<!DOCTYPE html>
<html>
    <head>
        <script>
            window.setInterval(function(){
                document.getElementById('timeContainer').innerHTML = "The time is
 now " + Date()
            },2000);
        </script>
    </head>
    <body>
        <h1 id="timeContainer"></h1>
    </body>
</html>
```

Canvas

- One of the most exciting parts of HTML is the `<canvas>` element
- This provides a blank space to create programmatic
    - drawings
    - animations
    - interactive games
- It must be scripted somehow, the HTML element just provides a place holder

The `<canvas>` tag

- The `<canvas>` tag allows the width and height to be set in HTML
- Anything placed between the opening and closing `<canvas>` tag will only be displayed on browsers that do not support canvas
  - This provides a good fall back mechanism
- Make sure to give it an id so you can interact with it through JavaScript

Basic Drawings

- The canvas object only supports two types of drawings
  - Rectangles
  - Paths
- All drawing is done through a context object

```
var my_canvas = document.getElementById("canvas1");
var context = my_canvas.getContext('2d');
```

```
In [ ]: %%html
        <!DOCTYPE html>
        <html>
            <script>

                var my_canvas = document.getElementById("canvas1");
                var context = my_canvas.getContext('2d');
                context.fillRect(0,0,100,100);

            </script>
            <body>
                <canvas id="canvas1">Canvas is unsupported </canvas>
            </body>
        </html>
```

Rectangles

- Rectangles are drawn by calling either
  - fillRect(x,y,width,height)
  - strokeRect(x,y,width,height)
- A rectangular portion of the canvas can be erased using
  - clearRect(x,y,width,height)
- The x and y refer to the position of the upper left corner of the rectangle

In [ ]:

```
%%html
<!DOCTYPE html>
<html>
    <script>

        var my_canvas = document.getElementById("canvasa");
        var context = my_canvas.getContext('2d');
        context.strokeRect(0,0,100,100);
        context.fillRect(90,90,50,50);
        context.clearRect(100,100,25,25);
        context.fillRect(126,126,10000,10000);

    </script>
    <body>
        <canvas id="canvasa">Canvas is unsupported </canvas>
    </body>
</html>
```

Paths

- Paths are more complex, first you must start the path using
  - `beginPath`
- Then a combination of the following calls actually draw the lines
  - `moveTo(x,y)`
  - `lineTo(x,y)`
  - `arc(x,y,startAng,endAng,direction)`
- Finally, call `fill` or `stroke`
  - `fill` automatically closes the path, adding a line from the current spot to the first spot
  - `stroke` just draws the outline, so the path isn't automatically closed

In [ ]:

```
%%html
<!DOCTYPE html>
<html>
    <script>

        var my_canvas = document.getElementById("canvas2");
        var context = my_canvas.getContext('2d');
        context.beginPath();
        context.lineTo(100,100);
        context.lineTo(0,0);
        context.stroke();

    </script>
    <body>
        <canvas id="canvas2">Canvas is unsupported </canvas>
    </body>
</html>
```

In [ ]:

```
%%html
<!DOCTYPE html>
<html>
    <script>

        var my_canvas = document.getElementById("canvas3");
        var context = my_canvas.getContext('2d');
        context.beginPath();
        context.moveTo(50,50);
        context.lineTo(100,100);
        context.lineTo(100,50);
        context.fill();

    </script>
    <body>
        <canvas id="canvas3">Canvas is unsupported </canvas>
    </body>
</html>
```

In [ ]:
```
%%html
<!DOCTYPE html>
<html>
    <script>

        var my_canvas = document.getElementById("canvas4");
        var context = my_canvas.getContext('2d');
        context.beginPath();
        context.moveTo(50,50);
        context.lineTo(100,100);
        context.lineTo(100,50);
        context.stroke();
        context.beginPath();
        context.moveTo(150,150);
        context.lineTo(175,150);
        context.stroke()
    </script>
    <body>
        <canvas id="canvas4">Canvas is unsupported </canvas>
    </body>
</html>
```

```
In [ ]: %%html
        <!DOCTYPE html>
        <html>
            <script>

                var my_canvas = document.getElementById("canvas5");
                var context = my_canvas.getContext('2d');
                context.beginPath();
                context.moveTo(50,50);
                context.lineTo(100,100);
                context.lineTo(100,50);
                context.closePath();
                context.stroke();

            </script>
            <body>
                <canvas id="canvas5">Canvas is unsupported </canvas>
            </body>
        </html>
```

Style

- Style is set on the entire context at once
  - Each drawing after this style is set will have the same style
  - The general pattern is change the style, draw somethings, repeat
- The two main style properties are
  - fillStyle
  - strokeStyle

```
In [ ]: %%html
        <!DOCTYPE html>
        <html>
            <script>

                var my_canvas = document.getElementById("canvas6");
                var context = my_canvas.getContext('2d');
                context.fillStyle="rgba(255,0,0,0.25)"
                context.fillRect(10,10,100,100);
                context.fillStyle="rgba(0,0,255,0.5)"
                context.fillRect(50,50,100,100);

            </script>
            <body>
                <canvas id="canvas6">Canvas is unsupported </canvas>
            </body>
        </html>
```

Basic Animations

- Basic animations can be created by clearing and drawing new things over and over
- We could use `setInterval` for this
    - A better method is `requestAnimationFrame`, which knowns about graphics and wont abuse the system
    - `requestAnimationFrame` takes a callback function

In [ ]:

```
%%html
<!DOCTYPE html>
<html>
    <script>
        var global_x = 0;
        var global_alpha = 1;
        function draw(){
        var my_canvas = document.getElementById("canvas7");
        var context = my_canvas.getContext('2d');
        context.clearRect(0, 0, 300, 150); // clear canvas
        global_alpha = global_alpha - 0.01;
        if (global_alpha < 0){
            global_alpha = 1
        }
        context.fillStyle="rgba(0,0,0," + global_alpha + ")";
        context.fillRect(global_x,10,50,50);
        if(global_x > 250){
            global_x = 0
        }
            else{
        global_x = global_x + 0.5;
            }
        window.requestAnimationFrame(draw);
        }
        window.requestAnimationFrame(draw);

    </script>
    <body>
        <canvas id="canvas7">Canvas is unsupported </canvas>
    </body>
</html>
```