# LC 101

## Unit 3 - JavaScript

March 6, 2017

# Higher-Order Functions

- Functions in JavaScript are *first-class citizens*
  - This means that functions can be assigned to variables, passed as arguments to functions, and returned from functions, just like any other value
- A *higher-order function* is a function that either
  - Takes one or more functions as arguments and/or
  - Returns and function

```
function greaterThan(n) {
  return function(m) {
    // Because of closure, we have access to n in here.
    return m > n;
  };
}
var greaterThan10 = greaterThan(10);  // greaterThan10 is a function
console.log(greaterThan10(11));  // outputs true
```

# Anonymous Functions

- An anonymous function is a function that is not given a name
  - Often used when creating a function to pass as an argument or as a return value

```
function greaterThan(n) {
  return function(m) {  // This is an anonymous function.
    // Because of closure, we have access to n in here.
    return m > n;
  };
}
```

# Array.forEach

- We can apply some function to every element of an array using the **`forEach`** method
  - Note that this does not exist on IE8 and earlier
  - Potentially slower than looping (but still useful)
  - No way to break out of the “loop” early (other than throwing an exception)

```
var someNumbers = [ 1, 2, 3];
someNumbers.forEach(function(n) {
  console.log(n);
});
```

# Array.filter

- The `filter` method will return a new array containing only those values that pass a test
  - The supplied function should return a boolean value
  - The supplied function can take three arguments
    - element – the current element being processed
    - index – the index of the current element
    - array – the array itself

```
var someNumbers = [ 1, 2, 3];
var evenNumbers = someNumbers.filter(function(n) {
  return n % 2 == 0;
});
console.log(evenNumbers);  // outputs [2]
```

# Array.map

- The **`map`** method creates a new array by applying some function to each element of the original array
  - Like **`filter`**, the supplied function can take three arguments: element, index, and array

```
var someNumbers = [ 1, 2, 3];
var doubled = someNumbers.map(function(n) {
  return n * 2;
});
console.log(doubled);  // outputs [2, 4, 6]
```

# Array.reduce

- The **`reduce`** method combines the values of an array by applying a two-argument function to an accumulator value and each element of the array
  - The arguments to the reduce method are the function to apply and the starting value for the accumulator
    - The accumulator is optional. If not provided then the first element of the array is used
  - The supplied function can take four argument: accumulator, element, index, and array

```
var nums = [ 1, 2, 3];
var sum = nums.reduce(function(s, n) {
  return s + n;
}, 0);
console.log(sum);  // outputs 6
```

```
var nums = [ 1, 2, 3];
var sum = nums.reduce(function(s, n) {
  return s + n;
});
console.log(sum);  // outputs 6
```

# Array.every and Array.some

- The **`every`** method will return true if every element of the array passes the provided test
- The **`some`** method will return true if at least one element of the array passes

```
var someNumbers = [ 1, 2, 3];
var allEven =
someNumbers.every(function(n) {
  return n % 2 == 0;
});
console.log(allEven);  // outputs false
```

```
var someNumbers = [ 1, 2, 3];
var someEven =
someNumbers.some(function(n) {
  return n % 2 == 0;
});
console.log(someEven);  // outputs true
```

# JSON

- The JavaScript Object Notation (JSON) is a text-based format for storing and exchanging data
  - Originally based on the JavaScript format for objects and arrays
  - Now used as a general data format across languages and systems
- JSON values are strings whose syntax is similar to JavaScripts object and array syntax
  - Keys must be in double-quotes
  - Values can only be objects, arrays, strings, numbers, true, false, or null
    - No functions or variables
    - No comments
  - Newlines and whitespace (outside of quoted strings) is unimportant

# Converting to JSON

- Can convert a JavaScript object to JSON using the **`JSON.stringify`** method

```
var people = [{ "name":"Alice", "age":31, "city":"New York" },
              { "name":"Bob", "age":42, "city":"Saint Louis" }];
var json = JSON.stringify(people);
// json is now the string '[{ "name":"Alice", "age":31, "city":"New York" },
//                          { "name":"Bob", "age":42, "city":"Saint Louis" }]';
```

# Converting from JSON

- We can convert JSON to a JavaScript object using the **`JSON.parse`** method
  - **DO NOT** use **`eval`** to parse JSON – it will execute any embedded code

```
var json = '[{ "name":"Alice", "age":31, "city":"New York" },' +
           ' { "name":"Bob", "age":42, "city":"Saint Louis" }]';
var people = JSON.parse(json);
```

# Handling Events

- We can attach *event handlers* to DOM elements to execute code when events occur
  - An event handler is just a function that should be called when an event occurs

```
var button = document.getElementById("myButton");
button.addEventListener("click", function() {
  console.log("My button was clicked!");
});
```

- Any DOM element, including the entire windows, can have event handlers attached

```
// Called when anywhere in the windows is clicked.
addEventListener("click", function() { … });
```

# The onclick Attribute

- Many HTML elements have attributes such as **`onclick`** which can be used to attach a single handler to that element
  - Limited to the available attributes
  - Can only attach one handler of each type to an element

```
<button onclick="alert('hello');">Hello</button>
```

# Adding and Removing Event Handlers

- The **`addEventListener`** method can be used to add any number of handlers to an element
- The **`removeEventListener`** method can be used to remove event handlers
  - These methods are not available in IE8 and earlier (can use **`attachEvent`** instead)

```
var button = document.getElementById("myButton");
function doOnce() {
  alert("First!");
  button.removeEventListener("click", doOnce);
}
button.addEventListener("click", doOnce);
```

# Event Types

- Many different types of events can be generated
  - onclick, onmousedown, onmouseup, onkeypress, onload, onbeforeunload, onfocus, onblur, etc.
  - Event handlers can be attached to any of these
- See https://www.w3schools.com/jsref/dom_obj_event.asp

# The Event Object

- The event handlers added via **`addEventListener`** are passed an Event object when called
  - Not available in IE8 and earlier. Can use **`window.event`** instead
- Contains a lot of useful properties
  - type – the type of the event (e.g., "click")
  - target – reference to the element that generated the event
  - And many more
- Some properties are type-specific
  - which – on mouse and keyboard events – which button was pressed/released
- Again see https://www.w3schools.com/jsref/dom_obj_event.asp

# Event Propagation

- Some events on child elements will be *propagated* up to parent/ancestor nodes
  - These events are said to *bubble up* the tree
- We can stop this propagation with the **`stopPropagation`** method
  - Often, if we have handled an event at one node, we do not want the event propagated
  - The **`stopPropagation`** method is not available in IE8 and earlier. Can use the **`cancelBubble`** property instead. (I'm going to stop saying this on every slide.)

```
button.addEventListener("mousedown", function(event) {
  …
  event.stopPropagation();
});
```

# Default Actions

- Many events have a default action
  - E.g., clicking a link will take you to that link's target
- For most events, any JavaScript handler we've attached will get executed first
  - This gives us a chance to suppress the default action
- The **`preventDefault`** method will suppress the default action

```
element.addEventListener("click", function(event) {
  …
  event.preventDefault();
});
```

# The onload Event

- Often, we have code we want to execute when our page loads, but we want to be sure the page has been loaded entirely first
  - The **`onload`** event lets us do this

```
window.addEventListener("onload", onLoad);
```

# Threads of Execution

- Some languages allow you to execute multiple pieces of code “simultaneously”
  - These are generally called *threads* of execution
  - Might actually be doing *time slicing* – that is, rapidly switching between the threads – but it looks simultaneous to us
  - Can think of it as multiple copies of your program all executing in the same memory space but executing different parts of your code
- JavaScript is *single-threaded*
  - This means that we **cannot** spend too much time in an event handler (or doing anything else) or the page will become unresponsive

# Web Workers

- *Web workers* are a way to execute long-running pieces of code outside of the main JavaScript thread of execution
  - Only available to very recent browsers (e.g., IE11)
- Communication between the main thread and a worker can only be done via posting and receiving *messages*
- A web worker runs in its own global scope
  - So it does **not** have access to the window or document
- The code for a worker must be in a separate file

# Web Workers

In `code/squareworker.js`:

```
addEventListener("message", function(event) {
  postMessage(event.data * event.data);
});
```

In our main code:

```
var squareWorker = new Worker("code/squareworker.js");
squareWorker.addEventListener("message", function(event) {
  console.log("The worker responded:", event.data);
});
squareWorker.postMessage(10);
squareWorker.postMessage(24);
```

# setTimeout

- The `setTimeout` function can be used to schedule code for later execution
  - And `clearTimeout` can be used to cancel an unrun scheduled execution

```
var bombTimer = setTimeout(function() {
  console.log("BOOM!");
}, 500);

if (Math.random() < 0.5) { // 50% chance
  console.log("Defused.");
  clearTimeout(bombTimer);
}
```

# setInterval

- The **`setInterval`** method can be used to schedule a function to repeat every *x* milliseconds
  - And clearInterval can be used to cancel future runs

```
var ticks = 0;
var clock = setInterval(function() {
  console.log("tick", ticks++);
  if (ticks == 10) {
    clearInterval(clock);
    console.log("stop.");
  }
}, 200);
```

# requestAnimationFrame

- The **`requestAnimationFrame`** function is similar to setTimeout but instead of scheduling a function to run after a fixed time interval, it will run when the browser next repaints the screen
  - As the name suggests, it is generally used to update an animation
  - There is also a **`cancelAnimationFrame`** to unschedule an unrun execution