



Module 3

JavaScript Functions

JAVA – MODULE 3, DAY 7



Review



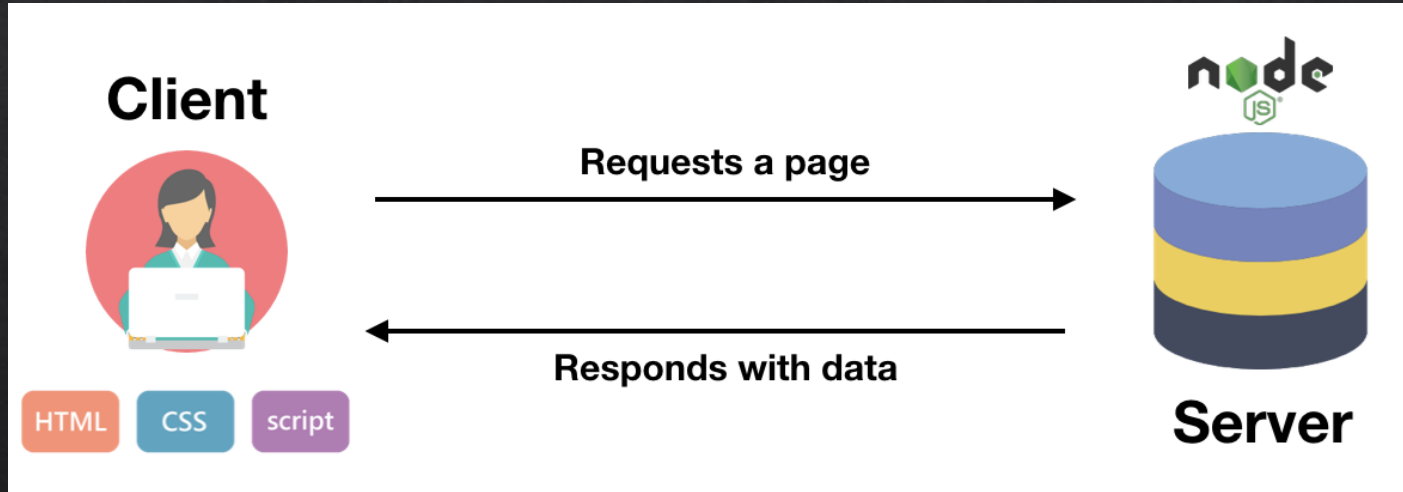
RECAP



Review



WHY JAVASCRIPT?





Review



JAVA vs JAVASCRIPT

- Runs on the Server-Side
- Compiled
- Must be syntactically correct or will not run



- Runs in the Browser
- Interpreted
- Does not have to be syntactically correct to run





DATA TYPES

Some of the more important JavaScript data types to be familiar with are:

- `const numberOfDaysInWeek = 7; //Number`
- `const favoriteDessert = "Bread Pudding"; //String`
- `const person = {name: "Chewbacca", age: 876}; //Object`
- `const isFriendsWithHanSolo = true; //Boolean`
- `const logFunction = console.log; //function`
- `null`
- `Undefined`

In JavaScript, variables aren't associated with any particular data type when you declare them. This makes JavaScript a loosely typed language.

Review





Review



JS OBJECTS

```

1  const myCar = {
2      make: "Tesla",
3      model: "Model 3",
4      year: 2020,
5      color: "Black",
6  };
    
```



Review



JS IS LOOSELY TYPED

```

1 let myVariable = 42;
2
3 myVariable = 'JavaScript is kind of like the multiverse.';
4
5 myVariable = {
6   js: "Even lets you",
7   assign: "objects into things that",
8   stored: "other data types just before"
9 };
10
11 myVariable = ["And", "then", "use", "arrays"];
12
13 myVariable = null;
14
15 myVariable = function someFunction() {
16   console.WriteLine("You can even assign functions to variables");
17 };
18
19 myVariable(); // And call them!
```





JS Hoisting

Review



```
console.log(x);
```

```
var x= 5;
```

message: undefined

Declaration in var is hoisted – take it to the top of memory so that it is known.
But initialization is not hoisted. It stays in the current scope.

```
console.log(x);
```

```
let x= 5;
```

message: Uncaught – ReferenceError: Cannot access 'x' before
initialization

Declaration in let is hoisted.





Review



== VS ===

Equals (==)

- `1 == 1`
- `1 == "1"`

Strictly Equals (===)

- `1 === 1`
- `1 !== "1"`



TRUTHY

Review



Truthy values

'false' (quoted false)
 '0' (quoted zero)
 () (empty functions)
 [] (empty arrays)
 { } (empty objects)
 All other values

Falsy values

false
 0 (zero)
 '' (empty string)
 null
 undefined
 NaN

In JavaScript, a **truthy** value is a value that is considered true when encountered in a Boolean context. All values are truthy unless they are defined as falsy (i.e., except for false, 0, -0, 0n, "", null, undefined, and NaN).

ex... if (25)





TRUTHY

Review



```

1  const result = calculate(); // Might be null or undefined
2
3  if (result) {
4    // Wasn't null / undefined
5  } else {
6    // Was null, undefined, or otherwise falsey
7  }
8
9  // Alternative way of writing this could be:
10 if (result !== null && result !== undefined) {
11   // ...
12 }

```



IF STATEMENTS



Review



```

1 if (user.currentStatus == 'OnFire') {
2   user.Stop();
3   user.Drop();
4   user.Roll();
5 } else {
6   user.conductBusinessAsUsual();
7 }

```



FOR LOOPS

Review



```
let sum = 0; // the sum of all our scores

for(let i = 0; i < testScores.length; i++)
{
    sum = sum + testScores[i]; // add each score to the sum
}

const average = sum / testScores.length;
```



Review



FUNCTIONS

```

1 function addNumbers(x, y) {
2   const sum = x + y;
3   return sum;
4 }
5
6 const result = addNumbers(60, 6);

```



Review



ARRAYS

Declaring and initializing an array

```
const testScores = [];
```

```
const testScores = [ 85, 96, 80, 98, 89, 70, 93, 84, 66, 96 ];
```

Determining the length of an array

```
const size = testScores.length;
```

Accessing elements within an array

```
const testScores = [ 85, 96, 80, 98, 89, 70, 93, 84, 66, 96 ];
```

```
testScores[0] = 82; // update the value at index 0 to 82
```

```
testScores[1] = 72; // update the value at index 1 to 72
```

```
testScores[4] = 80; // update the value at index 4 to 80
```

```
const highScore = testScores[3]; //set highScore to 98
```




Review



ARRAY FUNCTIONS

- `myArray[42]` - Java style array index
- **Push / Pop** (from the end of the array)
- **Unshift / Shift** (from the beginning of the array)
- **Splice** - Remove and/or add elements from an array
- **Slice** - Clone all or part of an array
- **Concat** - Create a new array from other arrays



Lecture



AGENDA

- Functions
- Anonymous Functions
- Array Functions
- Function Documentation



Lecture



FUNCTIONS

To write maintainable code, one of the primary things to avoid is code duplication. Programming languages provide many features and constructs to achieve this, and one of the most common is functions.

In the context of programming, a "function" is a way to package up a block of code, allowing you to reuse that block over and over again. This is especially helpful when you have a piece of logic that's needed in more than one place in a system.

There are two types of functions that you'll learn about in JavaScript: named functions and anonymous functions.



Lecture



NAMED FUNCTIONS

Named functions

Two parts of a named function: the **function signature** and the **function body**.

The components of a **function signature** are:

- The function name
- The function parameters

```
function multiplyBy(multiplicand, multiplier) {
  let result = multiplicand * multiplier;
  return result;
}
```

Diagram illustrating the components of a function signature:

- The function name `multiplyBy` is highlighted with a blue box and labeled "Name" with a blue arrow.
- The function parameters `(multiplicand, multiplier)` are highlighted with a green box and labeled "Parameters" with a green arrow.



Lecture



NAMED FUNCTIONS

Function name

Like variables, functions have names that can be used to reference them. Also, like variable names, careful consideration should be given to choosing names for functions. Function names should be:

- **descriptive** – it should be clear what type of action or calculation the function performs when invoked
- **camelCase** – the first letter of the name is lowercase and the first letter of each subsequent word is uppercase
- **unique** – function names need to be unique across all JavaScript code that's loaded into the page. If a name conflicts with another function, the one that's loaded last overwrites the other one

```
function addTwoNumbers(x, y){
  //Logic
  let result = x + y;
  return result;
}
```



Lecture



FUNCTION PARAMETERS

Function parameters

Parameters are variables that can provide input values to a function. When functions are created, parameter lists indicate what inputs the function can accept.

Optional parameters

Parameters in JavaScript are always optional. So what if a value isn't provided for a parameter when calling a function? if you don't assign a value to a variable, the variable is set to undefined.

Default parameter values

In some cases, there's a reasonable default value that you can use if a parameter value isn't supplied. You can use default parameters to make your functions more robust and useful in varying scenarios.



FUNCTION PARAMETERS

Handling an unknown number of parameters

There may be times when you want to handle an unknown number of parameters. A good example of this is writing a function that concatenates an unknown number of strings together. If you call a function like this:

```
let name = concatAll(firstName, lastName);
```

What if you want/need to call concatAll like the following:

```
let name = concatAll(honorific, firstName, mothersMaidenName,  
  '-', fathersLastName);
```



Lecture





UNKNOWN PARAMETERS



Handling an unknown number of parameters

The issue here is that there's no way to know how many parameters you have for that function. But in JavaScript, you can use a special variable to get at all of the given parameters, whether you expect them or not. This variable is called `arguments`.

With `arguments`, you can treat all parameters that have been passed to the function as an array, even if you don't have any parameters defined in the actual function definition.

```
function concatAll() { // No parameters defined, but we still might get some
  let result = '';
  for(let i = 0; i < arguments.length; i++) {
    result += arguments[i];
  }
  return result;
}
```

Lecture





Lecture



ANONYMOUS FUNCTIONS

Anonymous functions are functions that don't have a name. Functions in JavaScript can be used like any other value, so creating a function without a name is possible. You create an anonymous function with the following syntax:

```

Parameters      Fat Arrow
(multiplicand, multiplier) => {
    let result = multiplicand * multiplier;

    return result;
}
    
```


Functions as first-class citizens in JS

- ◆ Functions are treated as first-class citizens or objects in JavaScript. Functions are treated just like another other objects. Anything the objects can do in JS, functions can do the same.
- ◆ For Example,
- ◆ If you can pass an object as an input parameter to a function, you can pass a function as an input parameter to a function.
- ◆ In math,
 - ◆ $f(x) = x * x$
 - ◆ $a = f(x)$
 - ◆ if $x = 3$, then $a = 9$

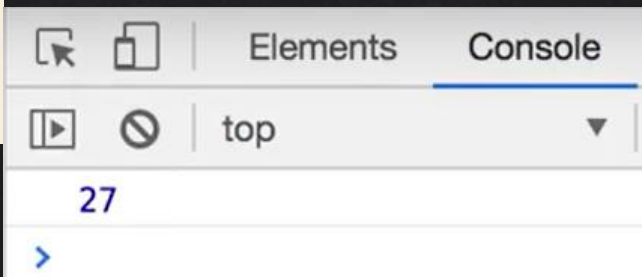
Functions as first-class citizens in JS

```
1  {  
2      let pizza = function pow(x, y){  
3          let total = 1;  
4          for(let i = 0; i < y; i++){  
5              total *= x;  
6          }  
7          return total;  
8      }  
9  
10     console.log(pizza(3,3));  
11 }
```

Functions as first-class citizens in JS

```
function pow(x, y){  
  let total = 1;  
  for(let i = 0; i < y; i++){  
    total *= x;  
  }  
  return total;  
}
```

```
let coolFunctions = [pow];  
console.log(coolFunctions[0](3,3));
```



Lambda/Anonymous FUNCTIONS

- ◆ Lambda functions are intended as a shorthand for defining functions that can come in handy to write concise code without wasting multiple lines defining a function.
- ◆ They are also known as anonymous functions, since they do not have a name unless assigned one.

ANONYMOUS FUNCTIONS

Arrow Syntax

```
1 const multiplyByTwo = (num) => {
2   return num * 2;
3 }
4
5 const answer = multiplyByTwo(21);
```

```
1 const multiplyByTwo = (num) => num * 2;
2
3 const answer = multiplyByTwo(21);
```

```
1 const multiplyByTwo = function multTwo(num) => {
2   return num * 2;
3 }
4
5 const answer = multiplyByTwo(21);
```





Lecture



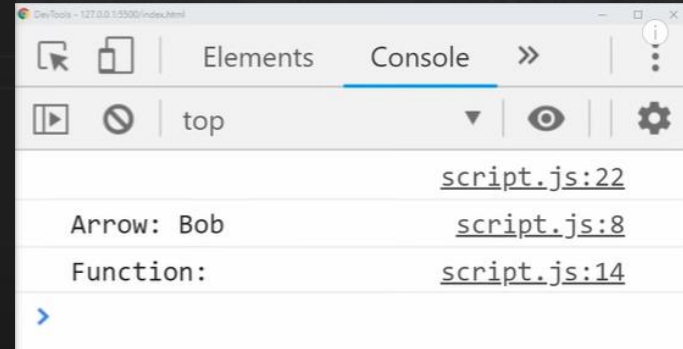
ANONYMOUS FUNCTIONS

Arrow Syntax

```

1 class Person {
2   constructor(name) {
3     this.name = name
4   }
5
6   printNameArrow() {
7     setTimeout(() => {
8       console.log('Arrow: ' + this.name)
9     }, 100)
10  }
11
12  printNameFunction() {
13    setTimeout(function() {
14      console.log('Function: ' + this.name)
15    }, 100)
16  }
17 }
18
19 let person = new Person('Bob')
20 person.printNameArrow()
21 person.printNameFunction()
22 console.log(this.name)


```





ANONYMOUS FUNCTIONS

Arrow Syntax

```
function allDivisibleByThree(numbersToFilter) {  
  // return numbersToFilter.filter( (number)=> {return number % 3 == 0})  
  // return numbersToFilter.filter( number => {return number % 3 == 0} )  
  // return numbersToFilter.filter( number =>  number % 3 == 0 )  
   return numbersToFilter.filter ( x => x % 3 == 0 );  
}
```



Lecture





Lecture



ARRAY FUNCTIONS

Array functions using anonymous functions

Arrays in JavaScript have many useful functions themselves that use anonymous functions.

forEach()

Performs like a for loop, running a passed in anonymous function for every element of an array.

```
let numbers = [1, 2, 3, 4];
```

```
numbers.forEach( (number) => {  
    console.log(`This number is ${number}`);  
});
```




Lecture



ARRAY FUNCTIONS

find()

The `find()` method returns the value of the first element in the provided array that satisfies the provided testing function. If no values satisfy the testing function, `undefined` is returned.

```
1 const array1 = [2, 12, 16, 181, 59];
2
3 const found = array1.find(element => element > 14);
4
5 console.log(found);
6 // expected output: 16
7
```



Lecture



ARRAY FUNCTIONS

findIndex()

The `findIndex()` method returns the index of the first element in the array that satisfies the provided testing function. Otherwise, it returns `-1`, indicating that no element passed the test.

```
1 const array1 = [2, 9, 15, 49, 44];
2
3 const isLargeNumber = (element) => element > 13;
4
5 console.log(array1.findIndex(isLargeNumber));
6 // expected output: 2
7
```



Lecture



ARRAY FUNCTIONS

filter()

The filter() method creates a new array with all elements that pass the test implemented by the provided function.

```
1 const words = ['Neo', 'Morpheus', 'Matrix', 'Trinity', 'Apoc', 'Switch'];
2
3 const result = words.filter(word => word.length > 6);
4
5 console.log(result);
6 // expected output: Array ["Morpheus", "Trinity"]
7
```



Lecture



ARRAY FUNCTIONS

map()

The `map()` method creates a new array populated with the results of calling a provided function on every element in the calling array.

```
1 const array1 = [1, 4, 9, 16];
2
3 // pass a function to map
4 const map1 = array1.map(x => x * 2);
5
6 console.log(map1);
7 // expected output: Array [2, 8, 18, 32]
8
```



Lecture



ARRAY FUNCTIONS

reduce()

The `reduce()` method executes a user-supplied “reducer” callback function on each element of the array, passing in the return value from the calculation on the preceding element. The final result of running the reducer across all elements of the array is a single value.

```

1  const array1 = [1, 2, 3, 4];
2  const reducer = (previousValue, currentValue) => previousValue + currentValue;
3
4  // 1 + 2 + 3 + 4
5  console.log(array1.reduce(reducer));
6  // expected output: 10
7
8  // 5 + 1 + 2 + 3 + 4
9  console.log(array1.reduce(reducer, 5));
10 // expected output: 15
11

```



Lecture



JS DOC

Documentation

One of the core responsibilities of a programmer is writing documentation for the code they create. Documentation is more than comments on the code, and there are a lot of comments that are considered bad practice.

Line Comments



```
// Set number of phones to one
let number = 1;
```



```
// Set number of phones to one
let numberOfOwnedPhones = 3;
```

✓ `let numberOfOwnedPhones = 3;` ← Self-Documenting Code

✓ `// Needed later to build the display table`
`let numberOfOwnedPhones = 3;`



Lecture



JS DOC

Documentation

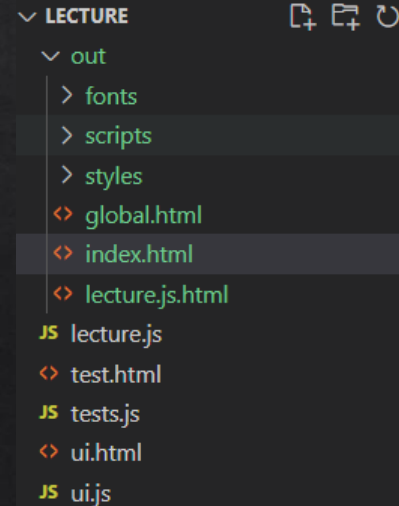
Function Comments - JSDoc

```
>npm install -g jsdoc
```

```
>jsdoc lecture.js
```

Jsdoc will generate these in the out directory:

```
global.html
index.html
lecture.js.html
```





Lecture



JS DOC

Documentation

Function Comments - JSDoc

Comments on functions are typically integrated into the IDE and are used to create documentation of your code for other programmers to use. They follow a standard format called [JSDoc](#).

```
/**
 * Takes two numbers and returns the product of
 * those two numbers.
 *
 * Will return NaN if exactly two numbers are not
 * given.
 *
 * @param {number} multiplicand a number to multiply
 * @param {number} multiplier a number to multiply by
 * @returns {number} product of the two parameters
 */
```


Second pptx

JS Functions

Objectives

- Functions introduction
- Named functions
- Function parameters
 - Optional Parameters
 - Parameter default values
 - Arguments variable
- Anonymous functions
- Array functions
- Function documentation

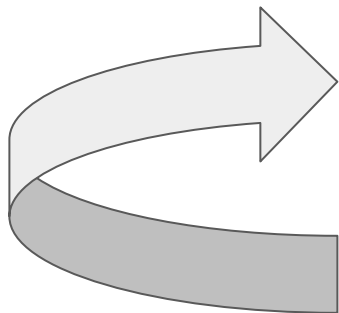


JS Functions

- The function is the primary organizational unit of JS, as opposed to classes in Java.
- Logically, we can also think of javascript functions as being similar to methods within Java classes - they have input parameters, can have outputs you can return etc.

JS Functions Defining and Calling

Here is an extremely simple named function definition, and a call to that function:



```
// A function is defined:  
function doSomething() {  
    console.log('Function is like a method');  
}  
  
// A method is called:  
doSomething();
```

Note that unlike Java, no return data type is defined.

JS Function argument

JavaScript can take in arguments, here is an example:

```
// A function is defined:
function doSomething(a, b) {
  console.log(a+b);
}

function returnSomething(a, b) {
  return a * b;
}

// A function is called:
doSomething(3, 9); // 12
let result = returnSomething(9,2);
console.log(result); // 18
```



JS Function argument, weirdness

Unlike Java, passing in a parameter is optional, regardless of how a function is defined! You can also assign an optional value for when no argument for the parameter is passed in:

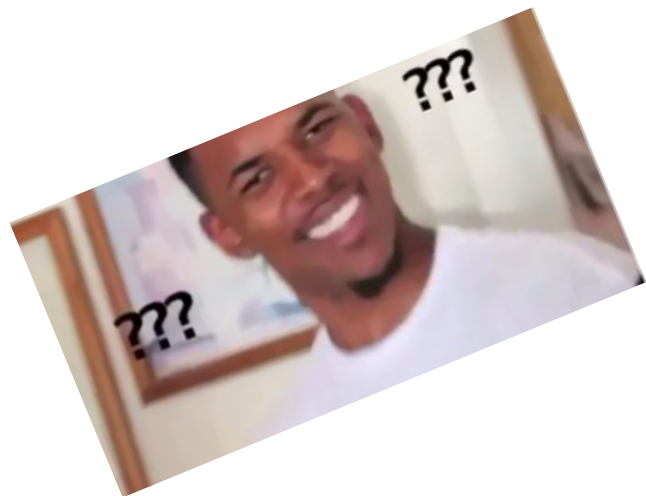
```
function returnSomething(a = 2, b = 5) {  
    return a * b;  
}  
  
let result = returnSomething();  
console.log(result); //10
```



The arguments parameter

```
function longestString() {  
  let longest = '';  
  for (let i = 0; i < arguments.length; i++) {  
    if (arguments[i].length > longest.length) {  
      longest = arguments[i];  
    }  
  }  
  return longest;  
}
```

```
> longestString('hi', 'this', 'supercalifragilisticexpialidocious', 'world');
```



Anonymous Functions

An anonymous function is one that does not have a name. Thus far, all the functions we've seen have a name following the form "function." Consider the following:

```
let listOfNumbers = [2, 3, 7];  
let evensOnly = listOfNumbers.forEach( (num) => (console.log(num * 2)) ); // 4 6 14
```

The anonymous function is highlighted in red, note that this function has an input (num), which represents each element of the array, it will print out that element multiplied by 2.

Anonymous Functions Uses

Anonymous functions are generally used in situations requiring a callback function.

A callback function is a function that is passed as an argument to another function.

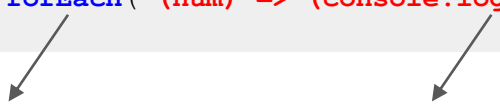
We will be looking at some examples of functions that require callback functions, and how we can use anonymous functions to make the code less verbose.



For Each with Anonymous Function

The For Each function iterates through the elements of the array. We can define an anonymous function to define what action will be undertaken during each iteration:

```
let listOfNumbers = [2, 3, 7];  
let evensOnly = listOfNumbers.forEach( (num) => (console.log(num * 2)) ); // 4 6 14
```



Call the forEach method.

Use an anonymous function to tell the forEach what to do during each iteration of the loop.

For Each without Anonymous Function

The preceding code could have been written without an anonymous function:

Without Anonymous Function:

```
let instructions = function multiplyByTwo(num) {  
    console.log(num * 2);  
}  
  
let listOfNumbers = [2, 3, 7, 76, 91];  
let evensOnly = listOfNumbers.forEach( instructions );
```

Compare the above with what we wrote on the previous screen:

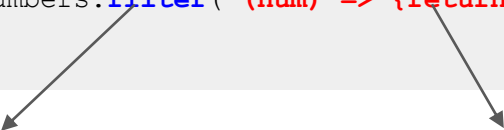
With Anonymous Function:

```
let listOfNumbers = [2, 3, 7];  
let evensOnly = listOfNumbers.forEach( (num) => (console.log(num * 2)) ); // 4 6 14
```

Filter with Anonymous Function

The filter function creates a new array from an existing one, provided that the element meets certain conditions.

```
let listOfNumbers = [2, 3, 7, 76, 91];  
let evensOnly = listOfNumbers.filter( (num) => {return num%2 === 0})  
console.log(evensOnly);
```

Two arrows originate from the explanatory text below. The first arrow points from the text 'Here is the filter method being called.' to the word 'filter' in the code. The second arrow points from the text 'Here we have an anonymous function telling the filter function how to filter' to the anonymous function '{return num%2 === 0}' in the code.

Here is the
filter method
being called.

Here we have an
anonymous function telling
the filter function how to
filter

Filter without Anonymous Function

Again, some comparing and contrasting. You can certainly define a function and not do it anonymously:

```
let instructions = function filterInstructions(num) {  
  
    if (num%2 ==0) {  
        return true;  
    }  
    return false;  
}  
  
let listOfNumbers = [2, 3, 7, 76, 91];  
let evensOnly = listOfNumbers.filter( instructions )  
console.log(evensOnly);
```

Without Anonymous Function:

```
let listOfNumbers = [2, 3, 7, 76, 91];  
let evensOnly = listOfNumbers.filter( (num) => {return num%2 === 0})  
console.log(evensOnly);
```

With Anonymous Function:

Map with Anonymous Function

The map function takes an array and performs some kind of operation on each row. The key point is that you still have the same number of elements.

Consider the example, we just saw, that of multiplying each element in an array by two:

```
let listOfNumbers = [2, 3, 7, 76, 91];  
let doubleNumbers = listOfNumbers.map((n) => n *2);  
console.log(doubleNumbers);  
//[4, 6, 14, 152, 182]
```


Reduce with Anonymous Function

The reduce method takes an array of numbers and makes one number out of the whole process. Simplest example? How about add every element in an array:

```
let listOfNumbers = [2, 3, 7, 76, 91];  
let doubleNumbers = listOfNumbers.reduce( (runningTotal, currentValue) => runningTotal  
+ currentValue);  
console.log(doubleNumbers); //179
```

Let's write some anonymous
functions!

