BEFORE FUNCTIONS AND SCOPE, QUICK REVIEW FROM LAST WEEK

Let's look at a for loop:

```
let a = [ 1, 2, 3, 4, 5 ];
for ( let i = 0; i < a.length; i++ ) {
        console.log( a[i] );
}
```

We can cache the **array's** length, to save some time:

```
let a = [ 1, 2, 3, 4, 5 ];
let arrayLength = a.length;
for ( let i = 0; i < arrayLength; i++ ) {
        console.log( a[i] );
}
```

The same technique but with **strings**:

```
let departments = ['Fine Art', 'Illustration', 'Cartooning'];

for ( let i = 0; i < departments.length; i++ ) {
 let department = departments[i];
 console.log( department );
}
```

But what if we actually want to perform some sort of operation on our data? Rather than just seeing it?

We can use a **function** in conjunction with a **method** native to **JavaScript**.

JavaScript **arrays** have several **iterator methods**. Many of the methods require a **function** to be passed in as an **argument**

Each **element** in the **array** has the **statement** in the **function body** applied to it individually.

For example, the forEach() method is a cleaner approach to the previous code:

```
let departments = ['Fine Art', 'Illustration', 'Cartooning'];

departments.forEach( function( department ) {
 console.log( department );
});
```

And here it is again, but actually applying some sort of logic to the data:

```
let pets = [ "dog", "cat", "turle", "bunny" ];

pets.forEach(function( currentValue, index) {
        console.log( "I want a ", currentValue );
        console.log( index );
 }
);
```

In the previous examples, **'department'** or **'currentValue'** were just **elements**; it was arbitrary. We decided that term.

And the **function** is called a callback.

In brief, a **callback** is a **function** to execute for each **element**.

The **callback** also takes three **arguments**, the element value, the element index, the array being traversed

So, this:

```
departments.forEach( function(department) {
  console.log(department);
});
```

And this:

```
function useThisLater(element, index, array) {
  console.log("element: " + element);
  console.log("index: " + index);
  console.log(" ");
}

departments.forEach( useThisLater );
```

Function the same way.

Similar to how we discussed the difference in **variables** in **ES5** versus **ES6** (**var** versus **let** & **const**), there are are options to how we might write a **function** or **callback** in **ES6**.

Here we're moving slightly into **functions**, but we're going to repetitively cover this.

There are **function declartions**:

```
function myFuncName(param) {
        return param;
}
```

Function expressions:

```
let myFuncName = function(param) {
        return param;
}
```

And now, with ES6, arrow functions:

```
(param1, param2, …, paramN) => { statements }
(param1, param2, …, paramN) => expression
// equivalent to: => { return expression; }

// Parentheses are optional when there's only one parameter na
(singleParam) => { statements }
singleParam => { statements }

// The parameter list for a function with no parameters
// should be written with a pair of parentheses.
() => { statements }
```

In a practical context, this would manifest like this:

```
let materials = ['Hydrogen', 'Helium', 'Lithium', 'Beryllium']

console.log(materials.map(material => material.length));
// expected output: Array [8, 6, 7, 9]
```

Taken from MDN Arrow Functions using the Map method.

A direct comparison:

```
let materials = ['Hydrogen', 'Helium', 'Lithium', 'Beryllium']

materials.map(material => material.length);
```

```
let materials = ['Hydrogen', 'Helium', 'Lithium', 'Beryllium']

materials.map(function(material){
        return material.length;
})
```

So, we just covered a lot of ground and remembering all of the particular **syntax** and names of these **methods** is exceedingly difficult to memorize--which is totally fine and normal.

Because of this, we constantly need to reference **documentation**.

If you recall, many documentation websites are on the syllabus.

In any case, let's take roughly 10 or so minutes to skim over some documention on the Mozilla developer site.

Go here: https://developer.mozilla.org/en-US/docs/Web/JavaScript and track down the documention for:

- .every()
- .some()
- .filter()
- .map()

After you've looked over the **documentation**, open repl.it and create these **arrays**:

```
let evens = [];
evens.push( 2, 4, 6, 8, 10 );

let odds = [];
odds.push( 1, 3, 5, 7, 9 );
```

The **every() method** tests whether **ALL** elements in an array pass the test implemented by the provided **function**

```
let evenResult = evens.every(num => num % 2 === 0);
let allDivisibleByFour = evens.every(num => num % 4 === 0);

console.log("evenResult", evenResult);
console.log("allDivisibleByFour", allDivisibleByFour);
```

The **some() method** tests whether **AN** element in the array passes the test implemented by the provided **function**

```
let someDivisibleByFour = evens.some(num => num % 4 === 0);

console.log("someDivisibleByFour", someDivisibleByFour);
```

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

Note, this **method** does not mutate the original **array**

```
let bigNums = evens.filter(num => num > 5);
let smallNums = odds.filter(num => num < 5);

console.log("bigNums", bigNums);
console.log("smallNums", smallNums);
```

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

```
let timesFive = evens.map(num => num * 5);
let timesTen = odds.map(num => num * 10);

console.log("timesFive", timesFive);
console.log("timesTen", timesTen);
```

FUNCTIONS AND SCOPE

What's a **function**?

We've used these before, in a limited manner, but what's actually going on?

A **function** is a reusable statement, a group of reusable statements, that can be called later or anywhere in a program. Note! As long as you have access to it...

What's the point? It helps us avoid the need to re-write the same statement over and over and over again.

Functions help us tame our code. We can divide large unwieldy pieces of code into smaller, more manageable, pieces.

This is related to the principle of **DRY** programming--
Don't Repeat Yourself

We want to write as few lines of code as possible. Work smart, not hard.

Here it gets a bit complicated, but I promise we'll go over it.

In JavaScript, every function:

- is an instance of the **object** data type
- can have **properties**
- has a link to its **constructor method**
- can be stored in a **variable**
- can be **returned** from another **function**
- can be passed into another **function** as an **argument**

Before we **call** or **invoke** a **function**, we have to define it.

There are lots of ways to go about this, but the most common are **functions declarations** and **function expressions**, as noted before.

They both, obviously, use the **function** keyword.

Function declaration:

```
function message( words ) {
        console.log( words );
}
// Note: no semicolon
```

Function expression:

```
let message = function( words ) {
        console.log( words );
}
```

Both are similar, but only **function declarations** allow us to call the **function** *before* it's defined.

In practice:

```
message( 'Hello World!' );

function message( words ) {
        console.log( words );
}
// This won't give us an error
```

Why?

A **function declaration** causes its **identifier** to be **bound** before anything in its code-block is **executed**.

The **function expression** is evaluated in a more typical top-down manner.

```
message( 'Hello World!' );

let message = function ( words ) {
        console.log( words );
}
// This will throw an error, try it in repl.it
```

Function declarations have:

- a **name** for the **function** after the **function** keyword
- statements inside the **function** body, which get **executed** every time the **function** is called, are inside curly brackets {}
- an *optional* list of parameters inside parantheses () with multiple **parameters** separated by a comma

Calling, or **invoking**, a **function executes** the code defined inside the **function**

Defining and **calling** a **function** are two different things.

A **function** *is not* called when it's defined.

We can **call** a **function** by using parantheses after its name:

```
function hello() {
        console.log( "Hello World!" );
}
hello();
// note the semicolon
```

JavaScript functions are often defined as **methods** on **objects**. To call these **methods**:

```
const person = {
        speak : function() {
                console.log( "Hello World!" );
        }
}
person.speak();
```

Parameters and Arguments

If a **function** did the same thing every time it was called, that's rather limiting.

We'd have to write a **function** for every new feature or circumstance in order to add new features to our application.

We would have a problem like this:

```
function helloJustin() {
        console.log( "Hello Justin" );
}

function helloRonald() {
        console.log( "Hello Ronald" );
}
```

With **parameters**, we can make our code more useful:

```
function sayHi( name ) {
        console.log( "Hello " + name );
}

sayHi("Justin");
sayHi("Ronald");
```

Parameters refer to the variables defined in the **function's declaration**. **Arguments** refer to the actual values passed into the **function** when it's called.

```
function fnName( parameter ) {

}

fnName( argument );
```

Parameters from one **function** will never affect **parameters** in another **function** so long as they're not nested. **Parameters** are **local** to each **function**

We can use a comma-separated list to write a **function** with more than one **parameter**. The **parameters** and **arguments** should be ordered the same way.

```
function sum( x, y, z ) {
        console.log( "Sum: " x + y + z );
}

sum( 1, 2, 3 );
```

JavaScript functions don't perform **type checking**, like we described in previous weeks. Also, we can't specify the **type** of a **parameter** when defining the **function**.

So we have to be careful to prevent erros. We'll almost always use the same **type** for the same **parameter** every time we call the **function**.

But, the **parameters** in the **function definition** can be of different types.

Last week, we used a **return statement**.

If we want to update a **variable** using values computed in a **function** or pass it to another **function**, we use a **return statement**.

Using the **return statement** ends the **function's execution** and passes the value we're **returning**.

Some of you have noticed this, by default all functions in **JavaScript return *undefined***.

Even if we don't have the **return** keyword in our **function body**, it will return **undefined**.

We can store the **returned** value in a **variable**

```
function sum( x, y ) {
        return x + y;
}

let z = sum( 3, 4 );
console.log( z );
```

Passing a function into a function:

```
let num = sum( 3, 4 );

function double( x ) {
        return x * 2;
}
// this:
let numDouble = double( num );
// roughly same as:
let numDouble = double( sum( 3, 4) );
```

Try that in repl.it

And just a reminder, the **return** statement will stop the **function's execution**.

```
function speak( words ) {
        return;

        console.log( words );
}
// what will happen?
```

Alright, let's talk about **Scope**

Scope is a concept in programming languages that refers to the current context of **execution**, with context being which values can be referenced.

If a **variable** is *not* in **scope**, then we can't use it because we don't have access to it.

It's as if whatever piece of code we're **executing** doesn't even know it exists.

If we try to use a **variable** we don't have access to, we get an error:

```
function speak( words ) {
        console.log( words );
}
// versus this:
console.log( words );
// try that in repl.it, what happens?
```

Global scope: by default, we're in **global scope**. Anytime a **variable** is declared outside of a **function**, it is part of the **global scope**.

If that's the case, we'd call it a **global variable**.

Global variables are technically bad practice, because it's easier to overwrite the value of a **globally scoped variable**. Any **function** or **expression** on the page can reference a **global variable**.

As mentioned in the first week, when defining **let** and **const**, deal with scope.

let

const

const:

Constants are **block-scoped**, much like **variables** defined using the **let** statement. The value of a constant can't be changed through reassignment, and it can't be redeclared.

let:

let allows you to **declare variables** that are limited to a **scope** of a **block statement**, or **expression** on which it is used, unlike the **var** keyword, which defines a variable **globally**, or **locally** to an entire **function** regardless of **block scope**.

let versus var:

Variables declared by **let** have their **scope** in the **block** for which they are defined, as well as in any contained **sub-blocks**. In this way, **let** works very much like **var**.

The main difference is that the **scope** of a **var** variable is the entire **enclosing function**.

```
function varTest() {
  var x = 1;
  {
    var x = 2;  // same variable!
    console.log(x);  // 2
  }
  console.log(x);  // 2
}

function letTest() {
  let x = 1;
  {
    let x = 2;  // different variable
    console.log(x);  // 2
  }
```

The environment for **global variables** is accessible via the **global object**.

In the browser, this would be the **window object**.

All **global variables** are attached to the **global object**.

```
let message = "Hello!"
console.log("message");

// Using the window object:
console.log( window.message );
```

There's also **namespace**: a **namespace** is a container for a set of **variables** and **objects**, e.g. **functions**.

In terms of best practice, we don't want to pollute the **namespace**.

Later we'll look at how to create **namespaces** to organize our code. It's a way of preventing collision with other **objects** or **variables**.

Local scope: we can create a new **scope** whenever we declare a **function**. Inside the **function** body, we have access to **variables** declared inside that function and in the **outer scope**. Any **variables** declared inside that **function** are local to it.

A **function** inside of a **function** has access to the **outer function's variables**,

```
const globalNumber = 1;
function fn() {
        let localNumber = 2;
        console.log( globalNumber );
        console.log( localNumber );
}
fn();
// what happens if you add this, try it in repl.it:
console.log( localNumber );
```

Local scope example:

```
const a = "This a variable in the global scope.";
function myFunction() {
        let b = "This is a variable in the scope of myFunction
        return b;
}
console.log( myFunction() );
console.log( b );
```

Try that in repl.it

A **function** can access **variables** of the **parent scope**. So a **function** defined in the **global scope** can access all **variables** defined in the **global scope**.

```
// Global scope
const prefix = "Hello";

// sayHello is defined in global scope
function sayHello( name ) {
        // prefix was defined in global scope
        console.log( prefix + " " + name);
}

sayHello("JavaScript");
```

If it's a **function declaration**, we can also call it anywhere that has access to **global scope**

Nested function scope example:

```
let a = 1;
function getScore() {
        let b = 2;
        let c = 3;
        function add() {
                return a + b + c;
        }
        return add();
}
getScore();
```

When a **function** is defined inside another **function**, it's possible to access **variables** defined in the **outer function** from the **inner function**.

Alright, let's try and make a little dice roll application with some of these new techniques.