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, or 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.

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**, **function expressions**, and **arrow function expressions**, as noted before.

Also, we've primarily used **function declarations** so far.

We'll go over **arrow function expressions** after we cover the difference of **function declarations** and **function expressions**.

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

Function declaration:

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

Function expression:

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

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
```

An **arrow function expression** is a compact alternative to a traditional **function expression**, but is limited and can't be used in all situations.

There's several differences, linked below, but one example would be that **arrow function expression** don't have binding to this or super and shouldn't be used as a method.

Link: MDN Arrow Functions

Traditional function:

```
function ( parameter ) {
        return paramter + 100;
}
```

Arrow function:

```
parameter => parameter + 100;
```

Let's break that down though.

```
// Traditional Function
function ( parameter ) {
  return parameter + 100;
}

// Arrow Function Break Down
// 1. Remove the word "function" and place arrow
//    between the argument and opening body bracket
( parameter ) => {
  return parameter + 100;
}
```

```
// 2. Remove the body brackets and word "return" --
//    the return is implied.
( parameter ) => parameter + 100;

// 3. Remove the argument parentheses
parameter => parameter + 100;
```

Another example:

```
// Traditional Function
function (a, b){
  return a + b + 100;
}

// Arrow Function
(a, b) => a + b + 100;
```

```
// Traditional Function (no arguments)
let a = 4;
let b = 2;
function (){
        return a + b + 100;
}

// Arrow Function (no arguments)
let a = 4;
let b = 2;
() => a + b + 100;
```

If the **function** requires multiple lines of code...

```
// Traditional Function
function (a, b){
  let chuck = 42;
  return a + b + chuck;
}

// Arrow Function
(a, b) => {
  let chuck = 42;
  return a + b + chuck;
}
```

Named functions...

```
// Traditional Function
function justinsCoolFunction (param){
  return param + 100;
}

// Arrow Function
let justinsCoolFunction = param => param + 100;
```

There's much more complexity and nuance to these differences, which you can explore, but we'll stop at this for now.

These examples were taken from MDN documentation, you might want to take some time to read through it, so here's the link again:

MDN Arrow Functions

Functions as Methods

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

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

"Person" is the object; "Speak" is the method on the object

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 errors. 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 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**.

You should definitely take time to read documentation on **functions**, they're hard and complicated, it'll take time to get used to.

In class we'll try and apply all of these concepts in demos in the coming weeks. We'll use these ideas repeatedly.

As usual, send any questions my way.