**Functions**

**Intro to Functions**

As the code gets longer, inevitably there will be code that repeats the same number of steps. Instead of repeating the same steps multiple times, we can package up those processes into reusable chunks of code called functions.

Functions are lines of code, all packaged together that we can use and reuse, when we need them.

In most cases, a function is given some data (argument), performs operations on the data, and then returns the altered data.

**Declaring Function**

Functions allow us to package up lines of code that we often use in our programs.

We’re going to use the example of Pizza mode in Microwave.

**Parameters**

As the parameter we chose the number of slices:

function reheatPizza(numSlices) {

// code that figures out reheat settings!

}

The parameter comes after the function name, inside the parentheses. Also, it is possible to have multiple parameters - separated by commas inside the parentheses.

function doubleGreeting(name, otherName) {

// code to greet two people!

}

However, it is also possible to have functions with no parameters. These functions don't require an input, they just package up some code and perform some task.

// accepts no parameters! parentheses are empty

function sayHello() {

var message = "Hello!"

console.log(message);

}

**Return Statement**

Every function must have a return statement to return the altered input data.

// declares the sayHello function

function sayHello() {

var message = "Hello!"

return message; // returns value instead of printing it

}

**Running a Function**

We need to **invoke** (**call**) the function using the function name, followed by parentheses with any arguments that are passed into it.

// declares the sayHello function

function sayHello() {

var message = "Hello!"

return message; // returns value instead of printing it

}

// function returns "Hello!" and console.log prints the return value

console.log(sayHello());

**Parameters vs Arguments**

At first, it can be a bit tricky to know when something is either a parameter or an argument:

* Parameter: is always going to be a variable name and appears in the function declaration.
* Argument: is always going to be a value (number, string, boolean, etc.) and will appear in the code when the function is called or invoked.

**Returning vs Logging**

return and print are not the same thing. Printing a value to the JavaScript console only displays a value (that we can view for debug purposes), but the value it displays can’t really be used for anything else. For this reason, we should remember to only use console.log to test our code in the JavaScript console.

If we don’t explicitly define a return value, the function will return undefined by default.

As soon as the function reaches the first return statement, it will exit the function.

**Using Return Values**

A function’s return value can be stored in a variable or reused throughout the program as a function argument.

function addTen(x) {

return x + 10;

}

function divideByThree(y) {

return y / 3;

}

var result = addTen(2);

console.log(divideByThree(result));

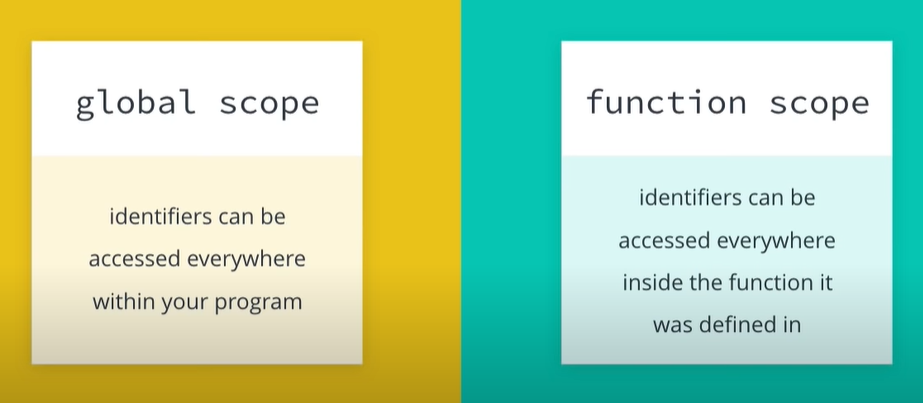
**Scope**

Apart from regular syntax errors, scope will be at the heart of many coding bugs, pretty much in any programming language.

Scope is about visibility/accessibility of a variable or a function.

In JavaScript, there are two different kinds of scopes: global and function.

* **Global:** if an identifier is defined outside of all functions, it is considered part of global scope, meaning it can be accessible everywhere within the program. All functions in our program can access variables defined in global scope.
* **Functions**: if an identifier is defined inside of a function, it is visible everywhere inside that function (even inside other functions inside that function).



**Shadowing**

When having the same variable name used in two different scopes, one as global and other as function, we can get a scope overriding or shadowing.

The global variable will be reassigned with the value from the function variable.

To prevent this from happening, we need to declare the function variable as a new variable, by using the keyword var.

**Global Variables**

*Why wouldn't we always use global variables?*

Global variables might seem like a convenient idea at first, especially when writing small scripts and programs, but there are many reasons why we shouldn’t use them unless we have to.

For instance, global variables can conflict with other global variables with the same name. Once our program starts getting larger and larger, it’ll get harder and harder to keep track and prevent this from happening.

Let’s minimize the use of global variables as much as possible.

**Scope Recap**

* If an identifier is declared in global scope, it's available everywhere.
* If an identifier is declared in function scope, it's available in the function it was declared in (even in functions declared inside the function).
* When trying to access an identifier, the JavaScript Engine will first look in the current function. If it doesn't find anything, it will continue to the next outer function to see if it can find the identifier there. It will keep doing this until it reaches the global scope.
* Global identifiers are a bad idea. They can lead to bad variable names, conflicting variable names, and messy code.

**Hoisting**

In most programming languages, we have to declare a function before we call it.

However, there is a feature in JavaScript called hoisting, which hoists all functions declared to the top of their current scope.

Hoist also happens with variable declarations, hoisting variables to the top of functions, however, variable assignments are not hoisted.

To avoid hoiting bugs, it is a better practice to declare functions in the top of the script, as well as variables at the top of functions.

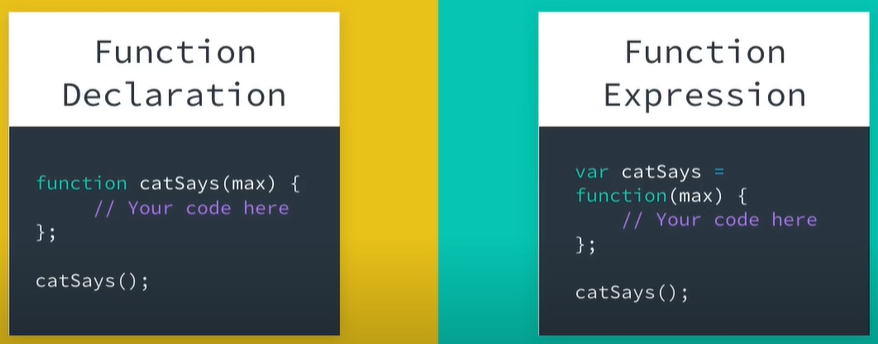
**Function Expressions**

Once we know how to declare functions, a whole new set of possibilities will open up to us.

In JavaScript we can also store functions in variables. When functions are stored inside variables, it’s called **function expressions**.

In this case, functions no longer will have names, and will become an anonymous function:

Now, we have two ways to define a function in JavaScript: **function declaration** and **function expression**.



var catSays = function(max) {

var catMessage = "";

for (var i = 0; i < max; i++) {

catMessage += "meow ";

}

return catMessage;

};

x = catSays(5)

console.log(x);

Function expressions are not hoisted, since they involve variable assignment, and only variable declarations are hoisted. The function expressions will not be loaded until the interpreter reaches it in the script.

**Function as Parameters**

Being able to store a function in a variable, makes it really simple to pass the function into another function. A function that is passed into another function is called **callback**.

// function expression catSays

var catSays = function(max) {

var catMessage = "";

for (var i = 0; i < max; i++) {

catMessage += "meow ";

}

return catMessage;

};

// function declaration helloCat accepting a callback

function helloCat(callbackFunc) {

return "Hello " + callbackFunc(5);

}

// pass in catSays as a callback function

console.log(helloCat(catSays));

**Inline Function Expression**

A function expression is when a function is assigned to a variable. And, in JavaScript, this can also happen when we pass a function **inline** as an argument to another function:

// Function expression that assigns the function displayFavorite

// to the variable favoriteMovie

var favoriteMovie = function displayFavorite(movieName) {

console.log("My favorite movie is " + movieName);

};

// Function declaration that has two parameters: a function for displaying

// a message, along with a name of a movie

function movies(messageFunction, name) {

messageFunction(name);

}

// Call the movies function, pass in the favoriteMovie function and name of movie

movies(favoriteMovie, "Finding Nemo");

//or//

// Function declaration that takes in two arguments: a function for displaying

// a message, along with a name of a movie

function movies(messageFunction, name) {

messageFunction(name);

}

// Call the movies function, pass in the function and name of movie

movies(function displayFavorite(movieName) {

console.log("My favorite movie is " + movieName);

}, "Finding Nemo");

**Why Use Anonymous Inline Functions???**

Why define a function that can only be used once and we can’t even call it by name?

Anonymous inline function expressions are often used with function callbacks that are probably not going to be reused elsewhere.

When we know that the function is not going to be reused, it could save us many lines of code to just define it inline.

**Function Expressions Recap**

Function expression: when a function is assigned to a variable. The function can be named, or anonymous. We use the variable name to call a function defined in a function expression:

*// anonymous function expression*

**var** doSomething = **function**(y) {

**return** y + 1;  
};

*// named function expression*

**var** doSomething = **function** **addOne**(y) {

**return** y + 1;  
};

*// for either of the definitions above, call the function like this:*  
doSomething(5);

***Returns****: 6*

We can even pass a function into another function inline. The pattern is commonly used in JavaScript and can be helpful streamlining our code:

*// function declaration that takes in two arguments: a function for displaying*

*// a message, along with a name of a movie*

**function** **movies**(messageFunction, name) {

messageFunction(name);

}

*// call the movies function, pass in the function and name of movie*

movies(**function** **displayFavorite**(movieName) {

console.log("My favorite movie is " + movieName);  
}, "Finding Nemo");