Before we begin...

- Open up these slides:
  - https://goo.gl/UqMLdT

Functions, Scope & Hoisting

Learning Objectives

- **Identify** the need for functions in JavaScript
- **Use** functions effectively, including providing data as *parameters*/*arguments*, and *returning* information
- **Talk** about scope and hoisting in JavaScript
- **Explain** the benefits of closures, and be able to create and use them effectively
- **Identify** higher-order functions and their role and benefits
- **Create** and use higher-order functions effectively

Agenda

- Functions
  - Parameters/arguments
  - Return values
- Scope
- Hoisting
- *Closures*
- *Higher Order Functions*

A quick review

- Loops
- Composite Data Types
  - Arrays
  - Objects
- Iteration

Functions

What are functions?

- A reusable section of code that has a purpose and a name
- The bread and butter of JS
- They can associate names with subprograms

What are functions?

Creating new words is normally bad practice, though fun.
It is essential in programming!

We give a name to a part of our program, and in doing so, we make it flexible, reusable and more readable

How do they work?

- We **define** a function
- We **call** (or **execute**) it when we want the code within the function to run

What can functions do?

They can perform any code!

- Calculations
- Animations
- Change CSS
- Change, add, or delete elements on the page
- Speak to a server (e.g. an API)
- Anything!

Declaring Functions

```
// A Function Declaration

function sayHello () {
    console.log( "Hello" );
}

// A Function Expression

var sayHi = function () {
    console.log( "Hi" );
};
```

Calling Functions

```
function sayHello () {
    console.log( "Hello!" );
}

sayHello(); // The callsite
```

Calling Functions

```
var sayHello = function () {
    console.log( "Hello!" );
}

sayHello(); // The callsite
```

Exercise

Create a Roll Virtual Dice Function

Optional: Receive a parameter to decide how many sides the Dice actually has (e.g. a 12-side dice)

arguments

```
function seeArguments() {
    console.log(arguments);
}

seeArguments();
seeArguments("hello");
seeArguments("hello", 2);
```

Parameters || Arguments

They aren't dynamic... yet! This brings us to parameters or arguments

Parameters (or **arguments**) allow us to provide a function with extra data or information. This is what makes a function flexible!

Parameters || Arguments

```
function sayHello ( name ) {
    var greeting = "Hello " + name;
    console.log( greeting );
}

sayHello( "Groucho" );

sayHello( "Harpo" );

sayHello(); // ???
```

Parameters || Arguments

```
function multiply (x, y) {
    console.log( x * y );
}

multiply( 5, 4 );

multiply( 10, -2 );

multiply( 100, 0.12 );
```

Passing in Variables

```
var addTwoNumbers = function (x, y) {
  return x + y;
};

var firstNumber = 10;

addTwoNumbers( firstNumber, 4 );
addTwoNumbers( firstNumber, 6 );
```

Parameters vs. Arguments

- A *parameter* is where the data is received
- An *argument* is where the data is passed in (the actual data)

```
function doSomething (parameter) {}

doSomething(argument);
```

Some Pseudocode

```
changeTheme function
    RECEIVE a themeChoice ("light" or "dark")
    IF themeChoice === "light"
        CHANGE the body background to "white"
        CHANGE the text color to "black"
    ELSE
        CHANGE the body background to "black"
        CHANGE the text color to "white"

moveToLeft function
    RECEIVE an element to animate
    STORE the current left position as currentLeft
    STORE the new left position, as desiredLeft, by adding 100px to currentLeft
    UPDATE the left position of the provided element to be desiredLeft
```

Return Values

Sometimes your function calculates something and you want the result!

Return values allow us to do that

We can store the result of calculations with return values. Think of **.toUpperCase();**

Return Values

```
function squareNumber ( x ) {
    var square = x * x;
    return square;
};

var squareOfFour = squareNumber( 4 );
```

Return Values

```
function squareNumber ( x ) {
    var square = x * x;
    return square;
};

var squareOfFour = squareNumber( 4 );

var squareOfTwelve = squareNumber( 12 );

squareNumber(8) + squareNumber(11);

squareOfFour + squareOfTwelve;
```

Return Values

```
function square ( x ) {
    return x * x;
};

function double ( x ) {
    return x * 2;
}

var result = double( square( 5 ) );
```

Return Values

```
var userOne = {
    admin: true
};

var userTwo = {
    admin: false
};

function isAdmin (user) {
    var admin = user.admin;
    return admin;
}

isAdmin(userOne); // true
isAdmin(userTwo); // false
```

Return Values

```
function sayHello () {
  return "No.";
  console.log( "Hi!" );
};

sayHello();
```

- A **return** value means that a function has a result
- It is always the last line that executes

Function Guidelines

Follow the F.I.R.S.T principle:

- **F**ocussed
- **I**ndependent
- **R**eusable
- **S**mall
- **T**estable

*Also, make it error-tolerant. But that isn't in the acronym

Callbacks

```
function runCallback ( cb ) {
    // Wait a second
    cb();
}

function delayedFunction () {
    console.log("I was delayed");
}

runCallback( delayedFunction );
```

Methods

```
var person = {
    firstName: "Jacques",
    lastName: "Cousteau",
    sayHi: function () {
        console.log("Hi, I'm Jacques");
    }
};

person.sayHi();
```

Exercise

Do the exercises found here

Resources

- Function Documentation
- Speaking Javascript: Functions
- Eloquent Javascript: Functions
- Javascript.info's Description

Scope & Hoisting

What is scope?

- Scope defines everything (variables, functions, values, etc.) you have access to at some point in your code
- Scope is like a pyramid. Lower scopes can access those above them, but not below
- The top level is the Global Scope (the complete JavaScript environment)
- Essentially, scoping is name resolution. Where can you access JavaScript identifiers in your code?

Lexical Scoping?

- JavaScript uses Lexical Scoping
- Lexical scoping means that scope is defined by the position in source code

Function Scope

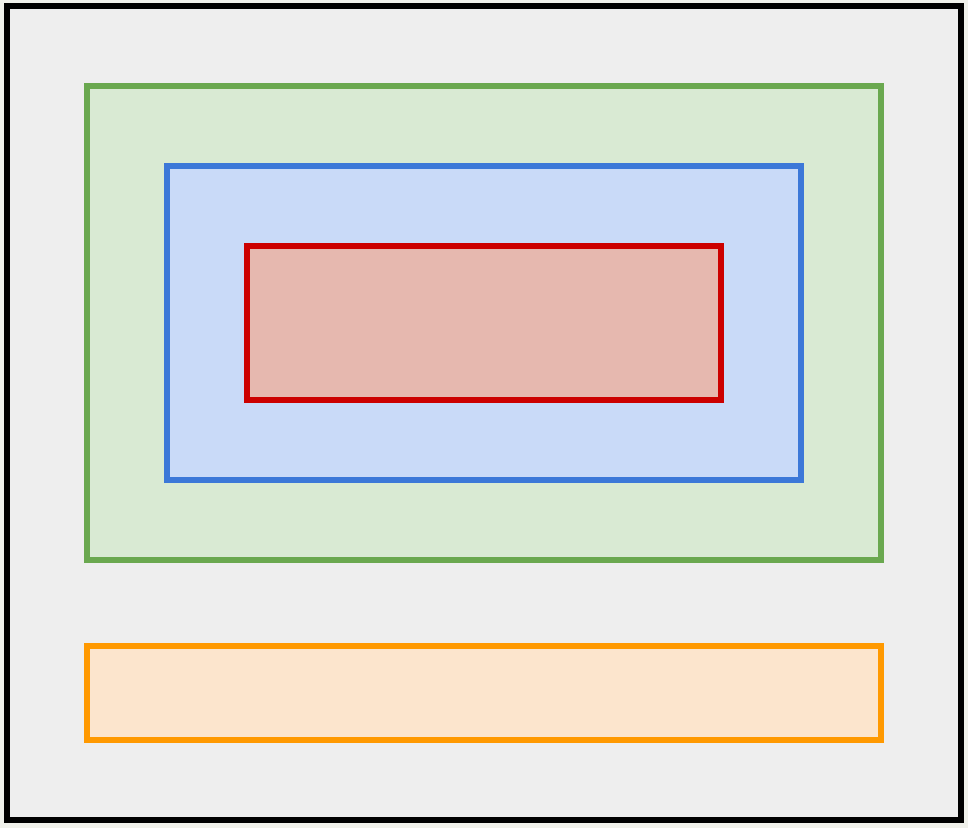
- Function Scoping is the scope that is created when a function is called
- The scope a function creates is called Local Scope

```
var global = "Global Scope";

function someFunction() {
    var local = "Local Scope";
}

console.log(global); // => "Global Scope"
console.log(local);  // => ReferenceError
```

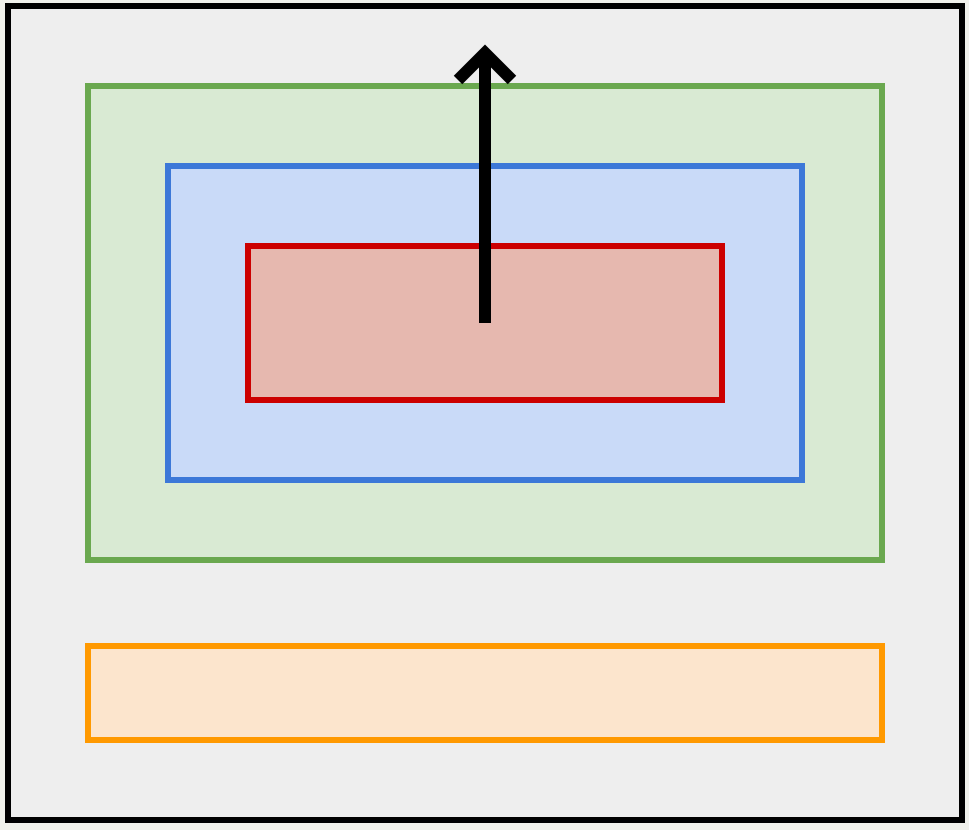
Function Scope

```
var global = "Global Scope";

function someFunction() {
    var innerScope = "Inner Scope";

    function someInnerFunction() {
        var innerInnerScope = "InnerInner Scope";
        // 3. What can we access from here?
    }

    someInnerFunction();
    // 2. What can we access from here?
}

someFunction();
// 1. What can we access from here?
```

Scope



Scope



Each of these is a *function scope*.

You can always look out!

What is hoisting?

Well, it's not a real term

- It is a way to explain the way that *execution contexts* work
- One way to think of it... Variable declarations and function declarations get physically moved to the top of the scope
  - But really, they get put in memory during the *compile phase*

Exercise

Go through the quiz here

Closures

What are closures?

A fancy name for a function that has access to an outer scope's variables etc.

Why would you use them?

- Useful for securing your web applications
- You can create private data and functions
- You can create utility functions easily

What are closures?

```
function createGame() {
    var score = 0;
    return function scoreGoal() {
        score += 10;
        return score;
    }
}

var scoreGoal = createGame();

console.log( scoreGoal() );
console.log( scoreGoal() );
```

What are closures?

```
function createGame() {
    var score = 0;
    return {
        gainPoints: function() {
            return score += 10;
        },
        losePoints: function() {
            return score -= 10;
        },
        getScore: function() {
            console.log( score );
        }
    };
}

var player = createGame();
player.gainPoints();
```

IIFE

Immediately **I**nvoked **F**unction **E**xpressions

Useful for creating a new scope! Essentially, it is a function that runs straight away

```
(function () {
    console.log("This runs");
})();


(function (x) {
    console.log("Parameters work too", x);
})(20);
```

Fix the bug!

```
function printNumbers() {
  for (var i = 0; i < 10; i += 1) {
    setTimeout(function() {
      console.log(i);
    }, i * 100);
  }
}

printNumbers();

// The timer is broken!
// It prints out 10, 10 times
// Make it work properly!
// Hint: You may need to create a new scope
// Try and explain why it happens
```

Higher Order Functions

What are they?

- A higher order function is a function that operates on other functions
  - Either by receiving it as a parameter, or by returning a function

Why would you use them?

- Creating utility functions
- Leads to D.R.Y code (**D**on't **R**epeat **Y**ourself)
- Creates more *declarative programming*
  - You describe patterns
  - The opposite is *imperative programming*, where you describe every single step
- Leads to more *maintainable*, *readable* and *composable* code
- Very common for libraries (like Lodash)

Functions as Input

```
function regularlyCalled() {
    console.log("Named function");
}

setInterval(regularlyCalled, 1000);

setTimeout(function () {
    console.log("Anonymous Function");
}, 1000);
```

Functions as Input

```
function repeatLog(num) {
    for (var i = 0; i < num; i += 1) {
        console.log(i);
    }
}

repeatLog(10);

repeatLog(4);
```

Functions as Input

```
function forEach(arr, callback) {
    for (var i = 0; i < arr.length; i += 1) {
        callback( arr[i], i );
    }
}

function handler(item, index) {
    console.log(item, index);
}

forEach(["one", "two", "three"], handler);

forEach(["one", "two", "three"], function (item, index) {
    console.log(item, index);
});
```

Functions as Output

```
function creator() {
    return function () {
        console.log("Returned function");
    }
}

var created = creator();
created();
```

Functions as Output

```
function createGreeting(start) {
    return function(name) {
        console.log(start + ", " + name);
    }
}

var hi = createGreeting("Hi");
hi("Jane");

var hello = createGreeting("Hello");
hello("Jeff");
```

Functions as Output

```
function makeAdder(x) {
    return function (y) {
        return x + y;
    }
}

var addTen = makeAdder(10);

console.log( addTen(25) );
console.log( addTen(116) );
```

Exercise

- Make a repeat function that runs any arbitrary function
- Create an unless statement
- Create a filter function for arrays
- Create a reduce function for arrays
- Create a map function for arrays
- Create a findIndex function for arrays
- Create anything else you see here

Homework

- Finish all exercises from class
  - Functions
- Upload your homework to GitHub
- Prepare for next lesson

Homework (Extra)

- Go through The Modern JavaScript Tutorial
- Read Eloquent JavaScript
- Read Speaking JavaScript

What's next?

- JavaScript & The Browser!
  - **D**ocument **O**bject **M**odel

Questions?

Feedback

https://ga.co/js05syd

Thanks!