Table of Contents

Introduction	1.1
Section 2: Execution Contexts and Lexical Environments	1.2
Section 3: Types and Operators	1.3
Section 4: Objects and Functions	1.4
Section 5: Object-Oriented Javascript and Prototypal Inheritance	1.5
Section 6: Building Objects	1.6
Section 7: Odds and Ends	1.7
Section 8: Examining Famous Frameworks and Libraries	1.8
Section 9: Let's Build a Framework / Library!	1.9
Section 10: BONUS Lectures	1.10
Section 11: BONUS: Getting Ready for ECMAScript 6	1.11

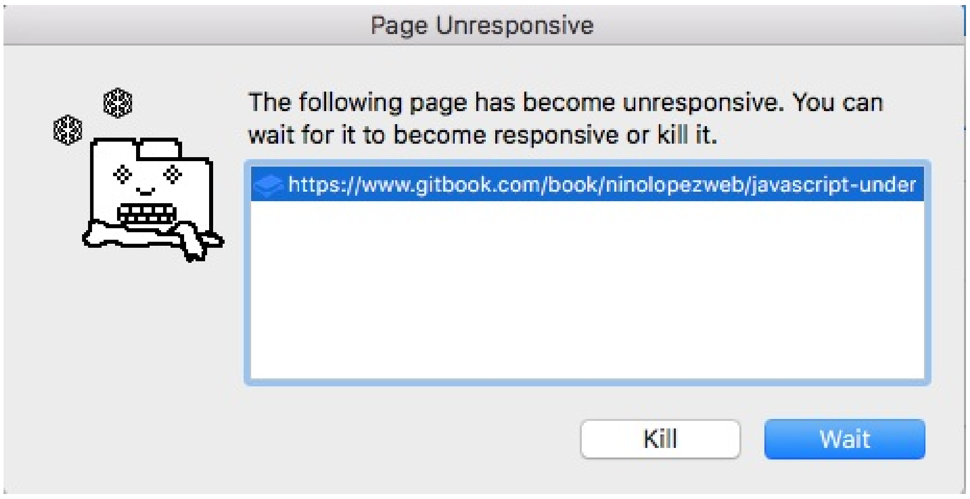
JavaScript: Understanding the Weird Parts

Course link: https://www.udemy.com/understand-javascript/

Contact me: http://ninolopezweb.com/contact/

NOTE:

3.13.17. I really liked Gitbook, however, I couldn't continue with the crashing. I'm going to complete these notes somewhere else.


Page Unresponsive

The following page has become unresponsive. You can wait for it to become responsive or kill it.

https://www.gitbook.com/book/ninolopezweb/javascript-under

Section 2: Execution Contexts and Lexical Environments

Syntax Parsers: A program that reads your code and determines what it does and if its grammer is valid. In other words, it's when programs, interpreters and compilers actually read your code, determines if the code is valid, and implements that syntax in a way that the computer can understand.

Lexical Environments: Where something sits physically in the code you write and what's around it. In other words, in programming languages, *where* you decide to write your code is very important.

Execution Contexts: A wrapper to help manage the code that is running. The wrapper can contain things beyond what you've written in your code.

Global: means not inside a function.

Creation Phase a.k.a. Global Execution Context when running code at the base level, the JS engine will automattically do the following:

1. Creates a **global object**. Your code will sit inside of this global object.
2. Creates a special variable called, **"this"**. At the global level, "this" would equal the Global (window) Object.
3. A reference or link to the **outer environment** if there is one.
4. **Hoisting**, which setups up functions and creates a list of variables (but sets them all to "undefined"). Example code is below.
5. Then JS will run your code line by line (which includes adding values to variables, thus replacing the "undefined".

Name/Value Pair: A name which maps to a unique value. And a value can be more name/value pairs. Those are called objects.

Object: A collection of name/value pairs.

Code Example 10-01

Consider the following code:

```
var a = 'Hello World';
function b() {
  console.log('Called b!');
}

b();
console.log(a);
```

console.log would report:

```
Called b!
Hello World
```

Now what if we did this?

```
b();
console.log(a);

var a = 'Hello World';
function b() {
  console.log('Called b!');
}
```

console.log would report:

```
Called b!
undefined
```

What happened? JS engine will sweep through your code in this order:

1. Stores every function to memory, in their entirety.
2. Stores every variable names to memory, but not the value! Instead JS will set a placeholder called *undefined*.
3. JS engine will begin executing every line of code, which includes assigning values to variables.

This process is called **hoisting**.

---

undefined is more like a placeholder. JS is saying, "I'm going to set some memory aside for when you decide to place a value here."

Single Threaded: means one command is being executed one line at a time. JS behaves in a single threaded manner.

Synchronous: means one at a time. JS executes in a certain order, one line at a time. In JS, one thing is happening at a time.

Function Invocation and the Execution Stack

Invocation means running (or calling) function. As in, "Hey, run the function." We do this by using parenthesis, which will, *invoke* the function.

What happens when you invoke a function? Consider the following code:

```
function a() {
  b();
  var c;
}

function b() {
  var d;
}

a();
var d;
```

Every time a new function is called, a new execution context is created. After Global Execution Context is created, the code above will run in this order, line by line:

1. `a();`
2. Which runs `function a()`
3. Within `function a()`, it takes a detour to function `b();`
4. `function b()` is executed, then returns to finish up `function a()`.
5. `function a()` is complete, then finally we get to `var d;`

Function, Context, and Variable Environments

Variable Environment: where the variable lives. Consider the following code:

```
function b() {
  var myVar;
  console.log(myVar);
}

function a() {
  var myVar = 2;
  console.log(myVar)
  b();
}

var myVar = 1;
console.log(myVar)
a();
console.log(myVar)
```

Here's the path:

1. `var myVar = 1;` is recorded first.
2. Execute `function a()` which record `myVar = 2;` .
3. Still inside `function a()` , we bounce to `function b();` .
4. Inside `function b();` , `myVar` is `undefined` .
5. We're back outside of the function scope where `var MyVar` still equals `1` .

In the console.log, you should see:

```
1
2
undefined
1
```

The Scope Chain

What console.log do you expect to see with the following code:

```
function b() {
  console.log(myVar);
}

function a() {
  var myVar = 2;
  b();
}

var myVar = 1;
a();
```

You will get: `1`. But why? Each function has an outer reference based on the lexical location where the function is written. If a variable is not defined within a function, myVar will look for it's value in the outer reference, and keep digging out until it reaches the Global Execution Context. This act of searching is called the **Scope Chain**.

Consider this code. What will console.log report?

```
function a() {
  function b() {
    console.log(myVar);
  }
  var myVar = 2;
  b();
}

var myVar = 1;
a();
```

Console log would return: `2`

`function b()` sits lexically, or physically, inside of `function a()`. `function b()`'s outer reference is `function b()` and that us where `myvar` found it's value. So it stopped there.

Scope, ES6, And let

Scope: is where your variable is available in your code. And if it's truly the same variable, or a new copy.

What About Asynchronous Callbacks

Asynchronous: more than one at a time.

When a web page is loaded, simultaneous things are happening like JS creating the global stack, user events and HTTP requests. Events are saved in an Event Queue. After the stack is finished (or empty), the event queue (if any are relevant) are processed. That means any functions relying on a user event will be saved until the end.

Section 3: Types and Operators

Conceptual Aside: Types and Javascript

Dynamic Typing: with JS, you don't have to tell JS what type of data a variable holds, JS will figure that out while your code is running.

Primitive Types

Primitive type: a type of data that represents a single value. That is, not an object. There are 6 of the them:

1. undefined: there's a bucket, but nothing in it.
2. null: there's not even a bucket.
3. boolean: true or false
4. number: this includes floating numbers.
5. string: a sequence of character enclosed in single or double quotes.
6. symbol: new for ES6.

Conceptual Aside: Operators

Operator: Generally, operators take two parameters and return one result. Using the operators invokes a special function; operators are functions.

`+` Addition

`-` Subtraction

`*` Multiplication

`/` Division

`%` Modulus

`++` Increment

`--` Decrement

Operator Precedence and Associativity

Operator precedence: determines which operator function gets called first. Functions are called in order of precedence.

Associativity: if we have two or more operators of equal precedence, then the equation may move left or right. Here's handy table. For example:

```
var a = 10 / 2 * 100;
```

Is `a` 500 or .05? Because associativity for multiplication and division has us moving from left-to-right, `a` is 500.

Conceptual Aside: Coercion

Coercion: Converting a value from one type to another (sometimes when you don't want it to). For example: `var a = 1 + '2';` would return a as `12` .

To avoid coercion, use "==" equality, or "===" strict equality. Use strict by default, to keep JS from coercing anything.

Here's a comparison table using double or triple equal signs.

Comparison Operators

Comparison operators move from left-to-right. The following code would equal `true` .

```
console.log(1 < 2 < 3);
```

But what about this code?

```
console.log(3 < 2 < 1);
```

It also returns `true` . Huh? Let's walk through the code. Since we're going from left-to-right, JS runs this first: `3 < 2` which is false. Still moving from left-to-right, JS now sees the code as:

```
console.log(false < 1);
```

In programming, false is converted to 0 through coercion, so 0 is the less than 1!

If you try to pass `undefined` as a number, you'll get `NaN` .

`null` will return `0` .

Existence and Booleans

Coercion also occurs with booleans. You can use to check on values. That's pretty much it.

Default Values

Regarding the or `||` operation, if you pass two value to coerced to true and false, it will always return the true value. Example:

```
function greet(name) {
  name = name || '$Your name here$';
  console.log('Hello ' + name);
}

greet();
```

Since name was never defined in the parameter, the '$Your name here$' is returned instead. Console.log would read, `Hello $Your name here$` .

Framework Aside: Default Values

What happens when libraries or plugins contain the same variables? The last variable loaded in the global execution context will take precedence. To check if a variable name is already in use, you can use a code like this:

`window.libraryName = window.libraryName || "Lib 2";`

This will provide a stop gap. *Wouldn't it make more sense to use an && inside of ||?*

Section 4: Objects and Functions

Objects and the Dot

What values (or properties) can an object have?

1. Primitive type
2. Another object
3. A function (or a "method" when sitting in the object)

JS will build a set of references (or addresses) to link a core object and it's properties and keep it memory.

You can use the Computed Member Access operator, which are `[]` brackets. We will use the dot operator.

Let's make a new object with some properties:

```
var person = new Object();

person.firstname = "Nino";
person.lastname = "Lopez";

//new object within a object!
person.address = new Object();
person.address.street = "123 Pacific Drive";
person.address.city = "San Diego";
person.address.state = "CA";

console.log(person)
console.log(person.address.street);
console.log(person.address.city);
console.log(person.firstname);
```

Console.log will return:

```
Object
    address: Object
    firstname: "Nino"
    lastname: "Lopez"
    __proto__: Object
123 Pacific Drive
San Diego
Nino
```

Objects and Object Literals

Object literal: is a comma-separated list of name-value pairs wrapped in curly braces. This is considered a faster approach to creating objects.

You can also add object values when you call the function.

```
var favMovie1 = {
    title: 'Goodfellas',
    year: '1990',
    actors: {
        lead: 'Ray Liotta',
        supporting1: 'Robert DeNiro',
        supporting2: 'Joe Pesci'
    }
};

var favMovie2 = {
    title: 'Magnolia',
    year: '1999',
    actors: {
        lead: 'John C. Reilly',
        supporting1: 'Tom Cruise',
        supporting2: 'Julianne Moore'
    }
};

//parameter in this case means, "I'll choose an object to pass later"
function movie(favorite) {
    console.log(favorite.title + ' was released in ' + favorite.year + ' and it stars
' + favorite.actors.lead + ', with ' + favorite.actors.supporting1 + ' and ' + favorit
e.actors.supporting2 + '.');
}

movie(favMovie1);
movie(favMovie2);

movie({
    title: 'Inside Out',
    year: '2015',
    actors: {
        lead: 'Amy Poehler',
        supporting1: 'Bill Hader',
        supporting2: 'Lewis Black'
    }
});
```

Framework Aside: Faking Namespaces

Namespace: is a container for variables and functions. It keeps variable names from being overwritten. JS does not specifically use namespaces like other languages do, but we can do something like it using object notations.

```
var english = new Object();
var spanish = new Object();

english.greet = 'Hello!'
spanish.greet = 'Hola!'

console.log(spanish);
```

JSON and Object Literals

How do you change an JS object literal (notation) into a JSON format string? JSON is technically a subset of the object literal syntax (it has quotes around the property names). So JS and can read JSON syntax, but JSON may not read JS syntax.

You can use `JSON.stringify` will covert an object to JSON string. Example:

```
var objectLiteral = {
    firstname: 'Mary',
    isAProgrammer: true
}
console.log(JSON.stringify(objectLiteral));
```

Output would be:

```
{"firstname":"Mary","isAProgrammer":true}
```

You can also turn a JSON string into JS object. Example:

```
var jsonValue = JSON.parse('{ "firstname": "John", "isADesigner":true}');
console.log(jsonValue);
```

The output would be:

```
Object {firstname: "John", isADesigner: true}
```

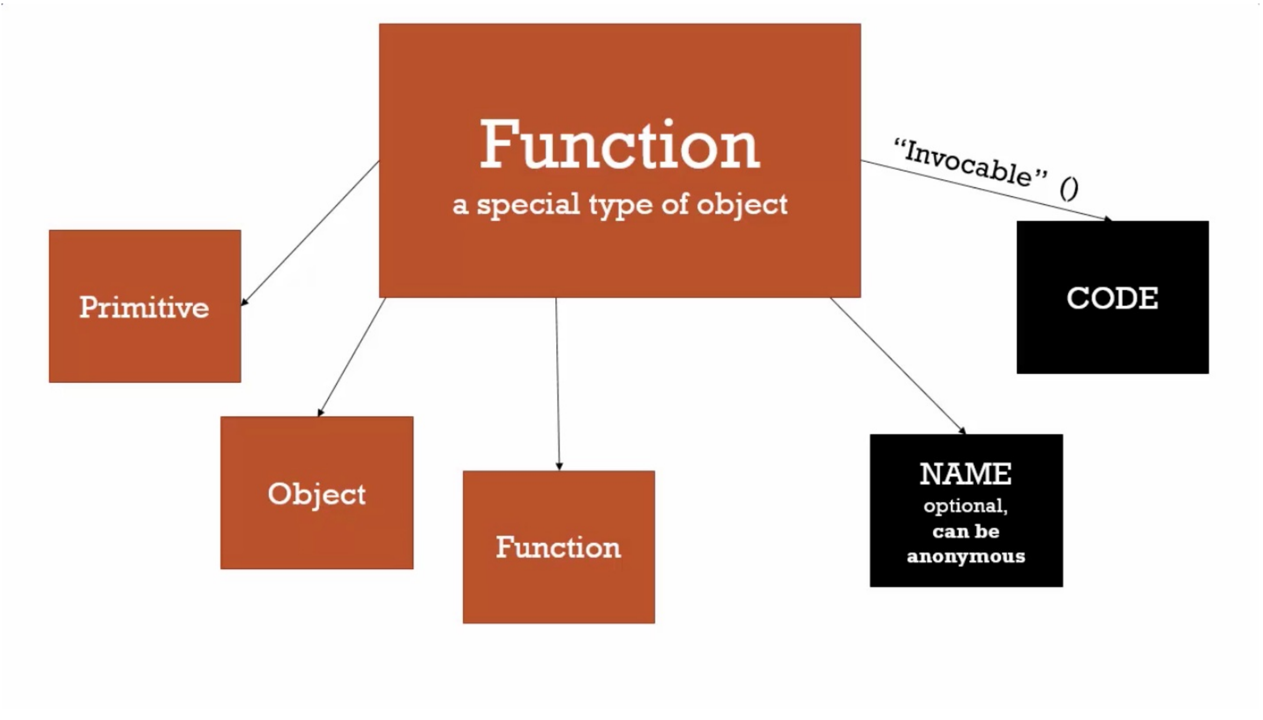
Functions are Objects

In JS, functions **are** objects.

First Class Functions: Everything you can do with other types you can do with functions.

- You can assign variables that has variables that is a function.
- Pass functions as parameters to other functions.
- You can create functions on the fly.
- You can also attach properties and methods to a function!

Now that you know a function is an object, you can attach a primitive, object, and another function to a function. Two special properties functions have are a name (which is optional) and the actual code (which becomes a property of the object). And then you say, "run that code that is in that property." It is invocable().



Function Statements and Function Expressions

Expression: is a unit of code that results in a value. It doesn't have to save to a variable. Functions do the work, expressions returns a value.

What is the difference between a **function statement** and **function expression**?

This is a **function statement**:

```
function greet() {
    console.log('hi');
}
```

This function statement does not result in a value. JS runs through and says, "ok, I'll commit this to memory, but I won't do anything with it until you tell me."

//

This is a **function expression**:

```
var anonymousGreet = function() {
    console.log('hi');
}
```

Notice that this function does not have a name. It is called an **anonymous function**. How do we invoke this function?

```
var anonymousGreet = function() {
    console.log('hi');
}

anonymousGreet();
```

The parenthesis invoke the function.

//

Pop quiz! What would this block return?

```
function log(a) {
    console.log(a);
}

log(function() {
    console.log('hi');
});
```

It would return:

```
function () {
    console.log('hi');
}
```

Ok, but how do you get the function to run? Like this:

```
function log(cat) {
    cat();
}

log(function() {
    console.log('hi');
});
```

I not too sure about this one, but this is what I think is happening:

1. "Cat" is now representing the function. As if it were the name of the function.
2. Use parenthesis to call the "cat" function, and there you are.

Conceptual Aside By Value vs By Reference

By Value and by Reference has to do with variables.

by value means when you assign an primitive value to a variable, e.g., `a = 3`, the variable `a` sets its own memory address. Then say you want to set `b = a`. Even though `b` will also be set to `3`, `b` will get its own address. You're making a copy of the value.

by reference is when you're making copies with objects. If `b = a`, instead of two memory address being created, both variables will point to the same address, or object. Think of it when someone calls you by your real name and your nickname. Both names refer to you.

What should console.log return in this **by value** example?

```
var a = 3;
var b;

b = a;
a = 2;

console.log(a);
console.log(b);
```

The result would be:

```
2
3
```

What should console.log return in this **by reference** example?

```
var c = { greeting: 'hi' };
var d;

d = c;
c.greeting = 'hello'; //mutate

console.log(c);
console.log(d);
```

The result would be:

```
Object {greeting: "hello"}
Object {greeting: "hello"}
```

Did you know you can change reference using parameters? What would console.log return in this example?

```
var c = { greeting: 'hi' };
var d;

d = c;
c.greeting = 'hello'; //mutate

function changeGreeting(robot) {
    robot.greeting = 'Hola'; //mutate
}

changeGreeting(d);
//"d" will take the place of robot in my function
console.log(c);
console.log(d);
```

The result would be:

```
Object {greeting: "Hola"}
Object {greeting: "Hola"}
```

Here's where it might get confusing... When using references, the equals operator will create a new address for an existing variable during the creation phase. What would this code result in?

```
var c = { greeting: 'hi' };
var d;

d = c;
c.greeting = 'hello'; //mutate

function changeGreeting(robot) {
    robot.greeting = 'Hola'; //mutate
}

changeGreeting(d);
//"d" will take the place of robot in my function
console.log(c);
console.log(d);

c = { greeting: 'howdy' };
console.log(c)
console.log(d)
```

The result would be:

```
Object {greeting: "Hola"}
Object {greeting: "Hola"}
Object {greeting: "howdy"}
Object {greeting: "Hola"}
```

Wait. What does "mutate" mean? **Mutate** means to change something. That's all.

Objects, Functions, and 'this'

Execution Context is created with a function is invoked. The process in which the context is being created is called the **creation phase**. The context determines *how* the code is run, or executed.

So what happens during the creation phase?

1. Variable environment is created. *Tell me what the variables are.*
2. Outer enviroment is referenced; it's lexical placement. *Where am I?*
3. The 'this' variable is created. *Pay attention.* This will be pointing at a different object, a different thing, depending on how the function is invoked.

As is, 'this' refers to the global object. For example:

```
function a() {
    console.log(this);
    this.newvariable = 'hello';
}

var b = function() {
    console.log(this);
}

a();
console.log(newvariable);
b();
```

The result would be:

```
Window {...}
hello
Window {...}
```

What if you set your `this` inside of an object? In cases where a function is actually a method attached to a function, 'this' keyword becomes the object 'this' is sitting inside of. *I'm going to point your 'this' in the object that contains you.* Now you can grab other properties in that object.

What would this code return?

```
var c = {
    name: 'The c object',
    log: function() {
        console.log(this);
    }
}

c.log(); //run the method inside the object.
```

The result would be:

```
Object {name: "The c object"}
```

What do you think following code will return?

```
var c = {
    name: 'The c object',
    log: function() {
        this.name = 'Updated c object';
        console.log(this);

        var setname = function(newname) {
            this.name = newname;
        }
        setname('Updated again! The c object');
        console.log(this);
    }
}

c.log();
```

The result is:

```
Object {name: "Updated c object"}
Object {name: "Updated c object"}
```

Why does `setname('Updated again! The c object');` point back to the global object? Most people think it's a bug.

When a method is within the method, it's `this` will actually target the global object. **Not** its containing object.

You can fix this using the `self` pattern. How does JS know what `self` means when it's burried within a method? It will go up the lexical chain. Example:

```
var c = {
    name: 'The c object',
    log: function() {
        var self = this;

        self.name = 'Updated c object';
        console.log(self);

        var setname = function(newname) {
            self.name = newname;
        }
        setname('Updated again! The c object');
        console.log(self);
    }
}

c.log();
```

Conceptual Aside: Arrays - Collections of Anything

An **Array** is a collection that hold many things inside of it. You can place numbers, booleans, objects, expressions and strings inside of an array.

There are two ways to create an array:
`var arr = new Array();`
or
`var arr = [];`
Most coders use the second type.

How do you grab a value in an array? Use brackets:

```
var arr = [1, 2, 3];

arr[0];
```

What we have in `var arr;` is a number, boolean, object, a function, and a string. What would the code return?

```
var arr = [
    1,
    false,
    {
        name: 'Tony',
        address: '111 Main St.'
    },
    function(name) {
        var greeting = 'Hello ';
        console.log(greeting + name);
    },
    "hello"
];

console.log(arr);
//I want to run the function using "Tony" as the name
arr[3](arr[2].name);
```

The result would be:

```
[1, false, Object, function, "hello"]
Hello Tony
```

Let's breakdown that last line of the code.

1. I grab the function and run it by adding parenthesis: `arr[3]();`

2. I need a parameter for the function (otherwise the function will come back as `undefined`. Let's use the `name`, "Tony," which can be found in the 2nd value of the object. `arr[2].name`.

'arguments' and spread
Let's review. When the execution context is created, JS will generate the variable environment, 'this', and the outer environment. It will also create a keyword called 'arguments.'

arguments creates a list of all the values of all the parameters you pass to whatever function you're calling.

In the following code, JS hosting has set up memory space for `firstname`, `lastname`, and `language` and has set them to `undefined`.

If you start to pass arguments, parameters will be processed from left to right. What would the following code return?

```
function greet(firstname, lastname, language) {
    console.log(firstname);
    console.log(lastname);
    console.log(language);
    console.log('--------');
};

greet();
greet('John', 'Doe');
```

The result would be:

```
undefined
undefined
undefined
--------
John
Doe
undefined
--------
```

For ES6, you will be able to see a default parameter like so:

```
function greet(firstname, lastname, language = 'en') {}
```

For older browsers, you can set up a default this way:

```
language = language || 'en';
```

Which means (I think), you either have a defined parameter or you get an 'en'.

The full code is:

```
function greet(firstname, lastname, language = 'en') {

    language = language || 'en';

    console.log(firstname);
    console.log(lastname);
    console.log(language);
    console.log('--------');
};

greet();
greet('John', 'Doe');
```

Which results in:

```
undefined
undefined
en
--------
John
Doe
en
--------
```

Earlier we said that **arguments** keeps track of parameters. We can use that info to our advantage. Say we forgot to add parameters. We can write something to notify ourselves.

What would the following code result in:

```
function greet(firstname, lastname, language = 'en') {

    language = language || 'en';

    if (arguments.length === 0) {
        console.log('Missing parameters!');
        console.log('++++');
        return;
    }

    console.log(firstname);
    console.log(lastname);
    console.log(language);
    console.log('--------');
};

greet();
greet('John', 'Doe');
```

The result would be:

```
Missing parameters!
++++
John
Doe
en
--------
```

Remember: the `return` keyword stops and takes us out of the `greet();` function. It quits the function.

spread is an ES6 feature that handles extra parameters that are not defined by the function. It's written like this:

```
function greet(firstname, lastname, language, ...other) {}
```

Framework Aside: Function Overloading
Consider the following code:

```
function greet(firstname, lastname, language) {

    language = language || 'en';

    if (language === 'en') {
        console.log('Hello ' + firstname + ' ' + lastname);
    }

    if (language === 'es') {
        console.log('Hola ' + firstname + ' ' + lastname);
    }
};

greet('John', 'Doe', 'en');
greet('John', 'Doe', 'es');
```

The result would be:

```
Hello John Doe
Hola John Doe
```

There's a better, cleaner way to write this with default parameters. Check this out:

```
function greetEnglish(firstname, lastname) {
    greet(firstname, lastname, 'en');
}

function greetSpanish(firstname, lastname) {
    greet(firstname, lastname, 'es');
}

greetEnglish('John', 'Doe', 'en');
greetSpanish('John', 'Doe', 'es');
```

The result would be:

```
Hello John Doe
Hola John Doe
```

I think the point of this lesson is that you can have as many functions as you want in JS.

Conceptual Aside: Syntax Parsers
The instructor wanted to make it a point that JS may change your code while JS executes it.

Dangerous Aside: Automatic Semicolon Insertion
Always place your own semicolons. Letting JS insert semicolons will lead to problems. For example, if JS sees a carriage return after the keyword, `return`, it will automatically place a semicolon for you.

Consider this code:

```
function getPerson() {
    return
    {
        firstname: 'Tony'
    }
}

console.log(getPerson());
```

The result would be:

```
undefined
```

This is because JS saw the carriage return after `return` and added a semicolon, and told the function to quit running.

Always place return on the same line as the curly brace:

```
function getPerson() {
    return {
        firstname: 'Tony'
    }
}

console.log(getPerson());
```

The result would be:

```
Object {firstname: "Tony"}
```

Framework Aside: Whitespace

Whitespace are invisible characters that create literal "space" in your written code, such as, carriage returns, tabs, and spaces. Use it to your advantage. Add tons of comments.

Immediately Invoked Functions Expressions (IIFEs)

IIFE is when you invoke a function immediately after creating it. You can do this by adding parenthesis at the end of your function. Example:

```
var greetFunc = function(name) {
    console.log('Hello ' + name);
}('Nino');
console.log(greetFunc);
```

The result would be:

```
Hello Nino
```

You can also IIFE the function in the console.log:

```
var greetFunc = function(name) {
    return 'Hello ' + name;
};
console.log(greetFunc('Lopez'));
```

The result would be:

```
Hello Lopez
```

So how do you function statement as an IFFE? You can make it a function expression by wrapping the function in parenthesis. Those parenthesis make it an expression.

```
(function(name) {
    return 'Hello ' + name;
})("Bob");
```

Will result in:

```
"Hello Bob"
```

Or like this:

```
var firstname = "Nino";

(function(name) {
    var greeting = "Inside IIFE: Hello";
    console.log(greeting + " " + name);
}(firstname));
```

Result would be:

```
Inside IIFE: Hello Nino
```

Framework Aside: IIFEs and Safe Code

Regarding the Global Execution Context, the anonymous function above would be stored in it's own Execution Context. That means when the function is stored, the variable and its value is stored with it. For example:

```
var greeting = "Hola"; //being set in the Global Execution Context

(function(name) {
    var greeting = "Hello"; //being set in it's own Execution Context (via a function)
 so not to collide with identically named variables in other scripts.
    console.log(greeting + " " + name);
}("John"));

console.log(greeting);
```

The result would be:

```
Hello John
Hola
```

You will see most scripts wrapping their entire scripts in a function for this reason.

Understanding Closures
You can invoke a function within a function by doing this:

```
function greet(whattosay) {

    return function(name) {
        console.log(whattosay + ' ' + name);
    }

}

greet('Hi')('Tony');
```

The result:

```
Hi Tony
```

Let's walk through this:

1. `greet('Hi');` is invoking a function that `return` another function.
2. We are going to invoke the returned function by adding parenthesis and parameters right after `greet('Hi');` so that it looks like this: `greet('Hi')('Tony');`

Now, you can save a function as a variable:

```
function greet(whattosay) {

    return function(name) {
        console.log(whattosay + ' ' + name);
    }

}

var sayHi = greet('Hi');
sayHi('Tony');
```

The result would be:

```
Hi Tony
```

So how did `Tony` reach `function(name);` ?

1. When `greet('Hi');` is invoked, `Hi` is passed to the `whattosay` parameter.
2. When function(name); is returned and invoked, there is no parameter for the `whattosay` variable.
3. The JS engine will go up the scope chain to the outer lexical environment. In other words, to the area outside of where the function was created.

Closures are functions that refer to independent (free) variables (variables that are used locally, but defined in an enclosing scope). In other words, these functions 'remember' the environment in which they were created.

Understanding Closures - Part 2

```
function buildFunctions() {

    var arr = [];
    for (var i = 0; i < 3; i++) {

        arr.push(
            function() {
                console.log(i);
            }
        )
    }

    return arr;
}

var fs = buildFunctions();

fs[0]();
fs[1]();
fs[2]();
```

Here what I think is happening:

1. `buildFunctions()` is created in the Global Execution Context.
2. `var arr` is created.
3. Three of the same functions are pushed in `arr` .
4. The functions console.logs `i` at the time its passed.
5. `fs[0]();` invokes the `buildFunctions();`
6. Brackets are used to return one particular placement within the array.

You would think each function call would return a different number. But they don't, they all return 3. Why?

This is the way I understand it:

1. `buildFunctions();` places three of the same functions in each of the array items.
2. `i` ends at 3, and shuts down the for loop.
3. Console.log is not executing or defining `i` in the arrays' functions, so the code goes looking for the value for the `i` variable.

4. The code goes up one level and sees that `i` ended at 3 and gives that value to you. `i` is 3 when you call each of the array functions.

So how would you fix this code to return the function's placement in the array? Here's how:

```
function buildFunctions2() {

    var arr = [];
    for (var i = 0; i < 3; i++) {

        arr.push(
            (function(j) {
                return function() {
                    console.log(j);
                }
            }(i))
        )
    }

    return arr;
}

var fs2 = buildFunctions2();
fs2[0]();
fs2[1]();
fs2[2]();
```

The function inside the array is invoked anonymously with `i` as a parameter. The `i` value is passed to `j`.

Framework Aside: Function Factories

Function factory is when a function returns an object.

What would this code return?

```
function makeGreeting(language) {

    return function(firstname, lastname) {

        if (language === 'en') {
            console.log('Hello ' + firstname + ' ' + lastname);
        }
        if (language === 'es') {
            console.log('Hola ' + firstname + ' ' + lastname);
        }

    }

}

var greetEnglish = makeGreeting('en');
var greetSpanish = makeGreeting('es');

greetEnglish('John', 'Johnson');
greetSpanish('Juan', 'Juanson');
```

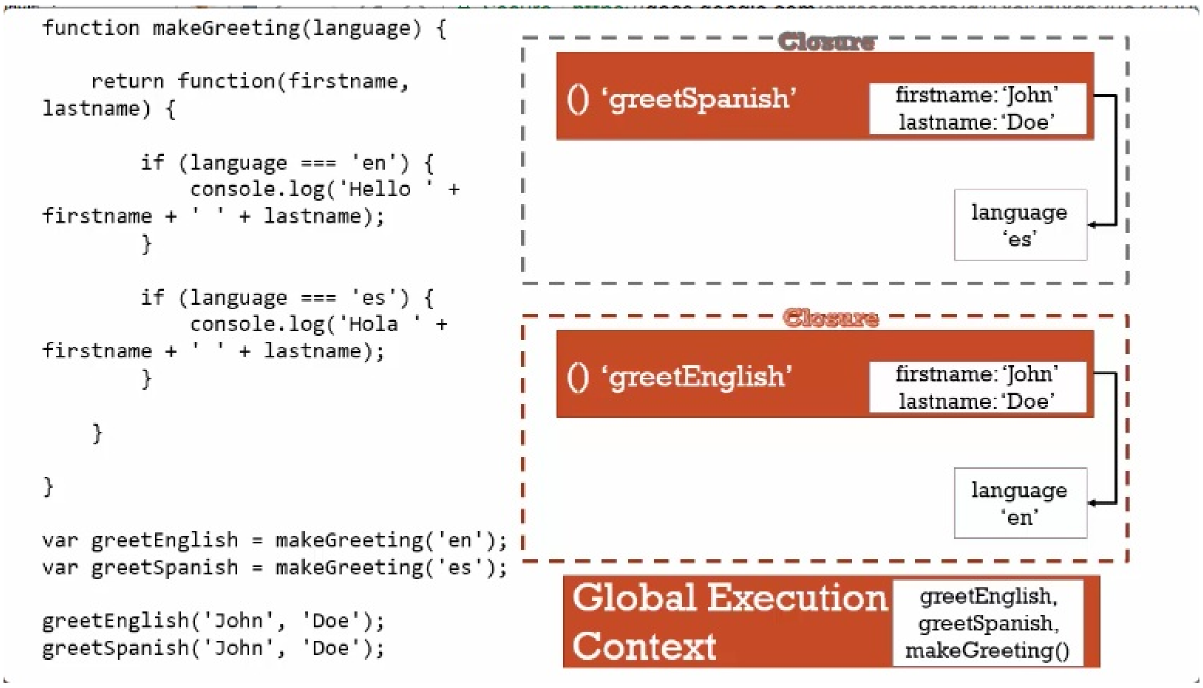
Result would be:

```
Hello John Johnson
Hola Juan Juanson
```

When invoked, `makeGreeting('en');` and `makeGreeting('es');` are creating their own execution contexts. That's why `language` is able to search its own execution context, its closure, for its value.

The instructor explains it this way:

1. During the code's first pass, the global execution context is created and `greetEnglish`, `greetSpanish`, and `makeGreeting()` are committed to memory.
2. During the second pass, `makeGreeting()` is invoked twice for each parameter. During invocation, new execution contexts are created, first for language `'en'` and second for language `'es'`.
3. On the third pass, `greetEnglish()` is invoked and creates a new execution context. The outer environment requires a value for `'language'`. JS knows that `'en'` was created first so it uses `'en'` as its outer reference.



```
function makeGreeting(language) {

    return function(firstname,
lastname) {

        if (language === 'en') {
            console.log('Hello ' +
firstname + ' ' + lastname);
        }

        if (language === 'es') {
            console.log('Hola ' +
firstname + ' ' + lastname);
        }

    }

}

var greetEnglish = makeGreeting('en');
var greetSpanish = makeGreeting('es');

greetEnglish('John', 'Doe');
greetSpanish('John', 'Doe');
```

Confused about how `greetEnglish` and `greetSpanish` are able to invoke all of this? Me too. I feel like `greetEnglish` is basically saying this:

```
makeGreeting('en')('John', 'Johnson');
```

Using this premise, I tried to add third level by adding a function inside of the `if` statement. It didn't work. The function came back as undefined. Do closures only work on two levels? I found the answer online:

> To use a closure, simply define a function inside another function and expose it. To expose a function, return it or pass it to another function.
>
> The inner function will have access to the variables in the outer function scope, even after the outer function has returned.

Closures and Callbacks

Callback Function is a function you give to another function, to be run hen the other function is finished. So the function you call (i.e. invoke), 'calls back' by calling the function you gave it when it finishes.

This is an example of a `setTimeout` callback function:

```
function sayHiLater() {

    var greeting = "Hi!";

    setTimeout(function() {
        console.log(greeting);
    }, 3000);

}

sayHiLater();
```

Console.log would return `Hi` after three seconds. You're saying, invoke this function and when you're done, invoke this other function.

Also,take a closer look at the code's formating. You are passing a function as a parameter.

Here's another example of a callback:

```
function tellMeWhenDone(callback) {

    var a = 1000; //some work
    var b = 2000; //some work

    callback(); //the 'callback' will run the function I give via the parameter!

}

tellMeWhenDone(function() {
    console.log("I am done!");
});
tellMeWhenDone(function() {
    console.log("No, I really am done!");
});
```

The result would be:

```
I am done!
No, I really am done!
```

---

call(), apply(), and bind()

What would the following code result in?

```
var person = {
    firstname: "John",
    lastname: "Doe",
    getFullName: function(){
        var fullname = this.firstname + " " + this.lastname;
        return fullname;
    }
}

var logName = function(lang1, lang2) {
    console.log('Logged: ' + this.getFullName());
}

logName();
```

The result would be:

```
Uncaught TypeError: this.getFullName is not a function
```

Why is this happening? Because the `this` is pointing the global object, not the `person` object.

.bind() You can fix this by using the `bind()` method, which tells the function where it should point `this` to. The last two lines of code have been modified:

```
var person = {
    firstname: "John",
    lastname: "Doe",
    getFullName: function(){
        var fullname = this.firstname + " " + this.lastname;
        return fullname;
    }
}

var logName = function(lang1, lang2) {
    console.log('Logged: ' + this.getFullName());
}

var logPersonName = logName.bind(person);

logPersonName();
```

The result is now:

```
Logged: John Doe
```

You can also use `bind()` by adding it to the end of the function:

```
var person = {
    firstname: "John",
    lastname: "Doe",
    getFullName: function(){
        var fullname = this.firstname + " " + this.lastname;
        return fullname;
    }
}

var logName = function(lang1, lang2) {
    console.log('Logged: ' + this.getFullName());
}.bind(person)

logName();
```

The result would be the same:

```
Logged: John Doe
```

.call() Using `call()` is another way of invoking your function. So why would you use it? Because you define `this` inside the parenthesis. And you can still pass it parameters.

```
var person = {
    firstname: "John",
    lastname: "Doe",
    getFullName: function(){
        var fullname = this.firstname + " " + this.lastname;
        return fullname;
    }
}

var logName = function(lang1, lang2) {
    console.log('Logged: ' + this.getFullName());
    console.log('Arguments: ' + lang1 + ' ' + lang2);
    console.log('----------');
}

logName.call(person, 'en', 'es');
```

Result would be:

```
Logged: John Doe
Arguments: en es
----------
```

.apply() `apply()` does the same thing as `call()`, except it takes an array of parameters. Like this:

```
logName.apply(person, ['en', 'es']);
```

You can make your code even shorter by using a function expression. **I'm not sure why this approach does not work for me**:

```
var person = {
    firstname: 'John',
    lastname: 'Doe',
    getFullName: function(){
        var fullname = this.firstname + " " + this.lastname;
        return fullname;
    }
}

(function(lang1, lang2) {
    console.log('Logged: ' + this.getFullName());
    console.log('Arguments: ' + lang1 + ' ' + lang2);
    console.log('----------');
}).call(person, 'en', 'de');
```

It comes back with an error on line 10. But this approach works just fine!

```
var person = {
    firstname: 'John',
    lastname: 'Doe',
    getFullName: function(){
        var fullname = this.firstname + " " + this.lastname;
        return fullname;
    }
}

var run = function(lang1, lang2) {
    console.log('Logged: ' + this.getFullName());
    console.log('Arguments: ' + lang1 + ' ' + lang2);
    console.log('----------');
};

run.call(person, 'en', 'de');
```

function borrowing Here's an example:

```
var person = {
    firstname: 'John',
    lastname: 'Doe',
    getFullName: function(){
        var fullname = this.firstname + " " + this.lastname;
        return fullname;
    }
}

var run = function(lang1, lang2) {
    console.log('Logged: ' + this.getFullName());
    console.log('Arguments: ' + lang1 + ' ' + lang2);
    console.log('----------');
};

run.call(person, 'en', 'de');

var person2 = {
    firstname: 'Jane',
    lastname: 'Doe'
}

console.log(person.getFullName.apply(person2));
```

The result is:

```
Logged: John Doe
Arguments: en de
----------
Jane Doe
```

Let me walk you through that last line in the code.

1. I need a function and I already wrote it in the `person` object, so I'm going to borrow it.
2. I write the object, `person`.
3. I write the name of the method, `getFullName`.
4. I use `apply()` to invoke and change the `this` to `person2`.

function currying means creating a copy of a function but with some preset parameters. Example:

```
function multiply(a, b) {
    return a*b;
}

var multipleByTwo = multiply.bind(this, 10);
console.log(multipleByTwo(4));
```

This is confusing, but what you're saying is:

1. I want to use `function multiply(a, b)`, but I want the a parameter to always be 10.
2. So you're literally saying, I want `this`, `a` to always be 10.
3. Then I invoke the function by setting 4 as the parameter. Console.log would return 40.

You can set both parameters by writing something like this:

```
var multipleByTwo = multiply.bind(this, 10, 12);
```

Any parameters added at invocation would be extra parameters that may not be used in the function. Example:

```
function multiply(a, b) {
    return a*b;
}

var multipleByTwo = multiply.bind(this, 3, 5);
console.log(multipleByTwo(8));
```

Console.log would return `15`.

Functional Programming Here's an example:

```
function mapForEach(arr, fn) {
    var newArr = [];
    for (var i = 0; i < arr.length; i++) {
         newArr.push(
             fn(arr[i])
         )
         console.log(newArr + " is new.")
    };

    return newArr;
}

var arr1 = [3, 9, 12];
console.log(arr1);

var arr2 = mapForEach(arr1, function(item) {
    return item * 2;
});

console.log(arr2);
```

Here's what I think is happening:

1.

Functional Programming - Part 2

```
```

Section 5: Object-Oriented Javascript and Prototypal Inheritance

Section 6: Building Objects

Section 7: Odds and Ends

Section 8: Examining Famous Frameworks and Libraries

Section 9: Let's Build a Framework / Library!

Section 10: BONUS Lectures

Section 11: BONUS: Getting Ready for ECMAScript 6