ADVANCED JAVASCRIPT

GOALS FOR THIS UNIT

1. Variable Scope (review)
2. Hoisting
3. Closure
4. Immediately Invoked Function Expressions (IIFEs)
5. Object-oriented JavaScript

VARIABLE SCOPE

There are two reasons for this outcome:

- Function level scoping
- Hoisting

FUNCTION LEVEL SCOPE

We've already talked about how variables are scoped at the function level. But let's revisit it.

In JavaScript variables are scoped to their functions. That means any variable in a function is available regardless of where it was declared as long as it was in that function.

FUNCTION LEVEL SCOPE EXAMPLE

Does this make sense?

```
function fnScope() {
  i = 3;
  console.log("locally" + i);
  var i;
}

fnScope();

console.log("globall" + i);

// > 3
// > Reference Error: i is not defined
```

DEMO

JavaScript WHY!?

HOISTING

HOISTING

Which of the following is correct?

Call the function first?

```
hello("Abraham");

function hello(name) {
  console.log("Hi " + name + "!");
}
```

Or declare it first?

```
function hello(name) {
  console.log("Hi " + name + "!");
}

hello("Abraham");
```

HOISTING

Both!

Welcome to JavaScript, my friend!

HOISTING

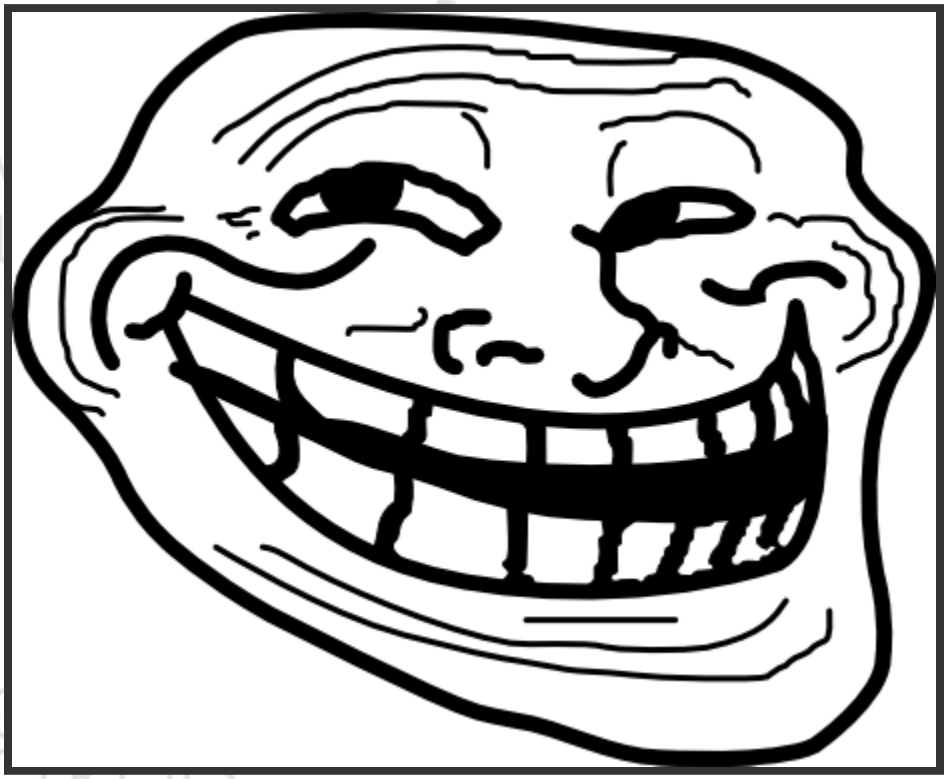
When a file is loaded, JavaScript collects all local variable declarations and brings them to the top of whatever function they are in.

HOISTING

Remember this example?

```
var meaningOfLife = 0;
function doStuff() {

  console.log(meaningOfLife);
  if(true) {
    var meaningOfLife = 42;
  }
}
// > undefined (not a reference error and not 0?)
```

When a function hoists the variables inside it. It does NOT bring the assignment with it. Only the name.



CLOSURES

VARIABLE SCOPE

Now consider this example. It's similar but there's an important difference. There's no variable declared *inside* the function.

```
var meaningOfLife = 42;
function doStuff() {
  console.log(meaningOfLife);
}

// > 42
```

Why is this not a reference error?

CLOSURES

A nifty side effect of function-level scoping is that because variables are scoped to functions, and locally-scoped functions have access to their parent scope, we can create closures as a form of encapsulation in JS.

CLOSURES

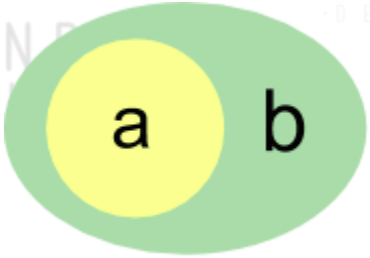
```
function yellow() {

  var a = 1;

  function green() {
    var b = 2;
    console.log("a: " + a); // > a: 1
    console.log("b: " + b); // > b: 2
  }

  green();
}

yellow();
```

DEMO

ENCAPSULATION USING CLOSURE

The set of variables visible to yellow:

The set of variables visible to green:



CLOSURE TAKE-AWAY #1

Variables inside a closure are only accessible *within* that closure.

In JavaScript, closure is basically the same concept as function scope.

IMMEDIATELY INVOKED FUNCTION EXPRESSIONS (IIFE)

IIFE

As we've discussed, one of the biggest problems in JavaScript is the global scope and keeping it as clean as we can. As such, a number of the important techniques and design patterns in JS were developed to address this problem.

One such technique is the Immediately Invoked Function Expression (IIFE). It is used to keep the global namespace from being cluttered with all of the variables and functions in a javascript file. When you use an IIFE, all of its variables are private because they are not visible out of the IIFEs scope. Compare these two:

Global vs. IIFE

GLOBAL

```
var makeA = function() {
  var a = 0;

  return function() {
    console.log(++a);
  };
};

var a = makeA();
var b = makeA();

a();
a();
b();
```

DEMO

IIFE

```
(function () {
  var makeA = function() {
    var a = 0;

    return function() {
      console.log(++a);
    };
  };

  var a = makeA();
  var b = makeA();

  a();
  a();
  b();
})();
```

DEMO

IIFE

Basically we're just putting a closure around our whole file.

To make a closure, we have to make a funciton.

IIFE

How do we make that function run?

We can make a function and call it later.

```
var fn = function(){ /* code */ };
fn();
```

Can we make them and call them all at once?

```
function(){ /* code */ }()
```

NOT QUITE!

IIFE

There's a bit of a catch!

Whenever the interpreter sees the `function` keyword it assumes it's looking at a *function declaration* which requires a name. The fix is relatively simple. If you wrap the function expression in parenthesese it will work.

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

DEMO

FUNCTION DECLARATION VS EXPRESSION

Let's review what function declarations as opposed to function expressions.

FUNCTION DECLARATIONS

A Function Declaration defines a named function variable without requiring variable assignment. Function Declarations occur as standalone constructs and cannot be nested within non-function blocks. It's helpful to think of them as siblings of Variable Declarations. Just as Variable Declarations must start with "var", Function Declarations must begin with "function".

i.e.

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

FUNCTION EXPRESSION

A Function Expression defines a function as a part of a larger expression syntax (typically a variable assignment). Functions defined via Functions Expressions can be named or anonymous. Function Expressions must not start with “function” (hence the parentheses around the self invoking example below)

FUNCTION EXPRESSION EXAMPLES

```
//anonymous function expression
var a = function() {
    return 3;
}

//named function expression
var a = function bar() {
    return 3;
}

//self invoking function expression
(function sayHello() {
    alert("hello!");
})();
```

FUNCTION DECLARATION VS EXPRESSION

DECLARATION

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

EXPRESSION

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

IIFE

Let's look through the syntax one more time.

FUNCTION EXPRESSION

We've established that IIFEs are an anonymous *function expression* (-FE).

```
function () {
  //remember how this didn't work before?
}();
```

IMMEDIATELY INVOKED

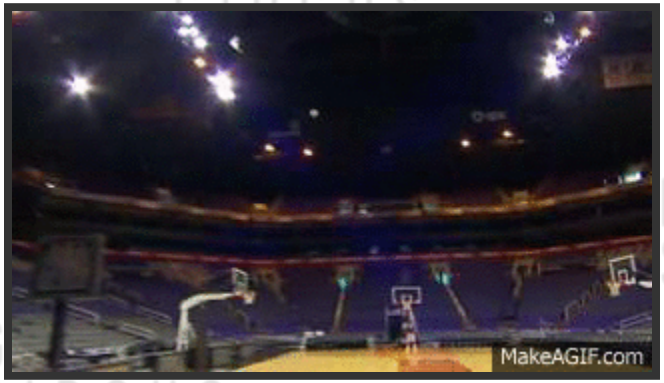
Remember to get the interpreter to differentiate between function declarations and function expression the function can't start with the `function` keyword. Hence, the parentheses. Once we have wrapped it in a set of parentheses, it will be *invoked immediately* (II-).

```
(function(){
  //adding the first set of parenthesis
})();
```

IIFE KEY TAKE-AWAY

An IIFE is an anonmyous function that is invoked as soon as it's created. It creates a *closure* that keeps all the variables inside private and contained.

QUESTIONS?



CLOSURES II

Consider this:

```
function makeCounter() {
  var i = 0;

  return function () {
    console.log(++i);
  }
}

var counter1 = makeCounter();
var counter2 = makeCounter();

counter1(); // > 1
counter1(); // > 2
counter1(); // > 3
counter2(); // > 1
console.log(i); // > Reference Error.
```

Because of this, each counter variable gets it's own scoped variable `i`.

DEMO

CLOSURE TAKE-AWAY #2

A closure (i.e. function) "remembers" the context in which it was created.

Even if that function is removed from that context and used elsewhere, it still has access to all the variables of its original context.

CLOSURE EXERCISE

1. Create a function named `makeCounter`.
2. This function should build and return a new counter function.
3. `makeCounter` takes one parameter, which is the number at which to start the new counter.
4. Test your `makeCounter` function as follows.

```
var counterA = makeCounter(0);
var counterB = makeCounter(5);
console.log( counterA() ); // 0
console.log( counterA() ); // 1
console.log( counterB() ); // 5
console.log( counterB() ); // 6
console.log( counterA() ); // 2
```

OBJECT-ORIENTED JAVASCRIPT

OOP

Object-oriented programming (OOP) is a paradigm that is pervasive in modern programming languages. Almost any other major language you are likely to encounter is an object-oriented (OO) language. JavaScript is not *exactly* object-oriented in the same way, though it does support many of the features that OOP programmers have come to expect. We will take a very brief look at the way some of these features work.

One of the most awesome and powerful things about JavaScript is that it supports programming in a wide variety of styles and paradigms. One of the worst things about JavaScript is that it supports programming in a wide variety of styles and paradigms.

YOU MAKE ME WANNA (SH)OOP

Many frameworks and companies do not use OOP, but many do.

OOP

OOP is based on modeling things in the real world (objects) in a programming language. Typically we create blueprints for these models (classes) and then create instances of those classes (objects) that do the actual work.

An object consists of two parts: state and behavior, or data and methods. That is, data about the object and functions (methods) that can act on that data.

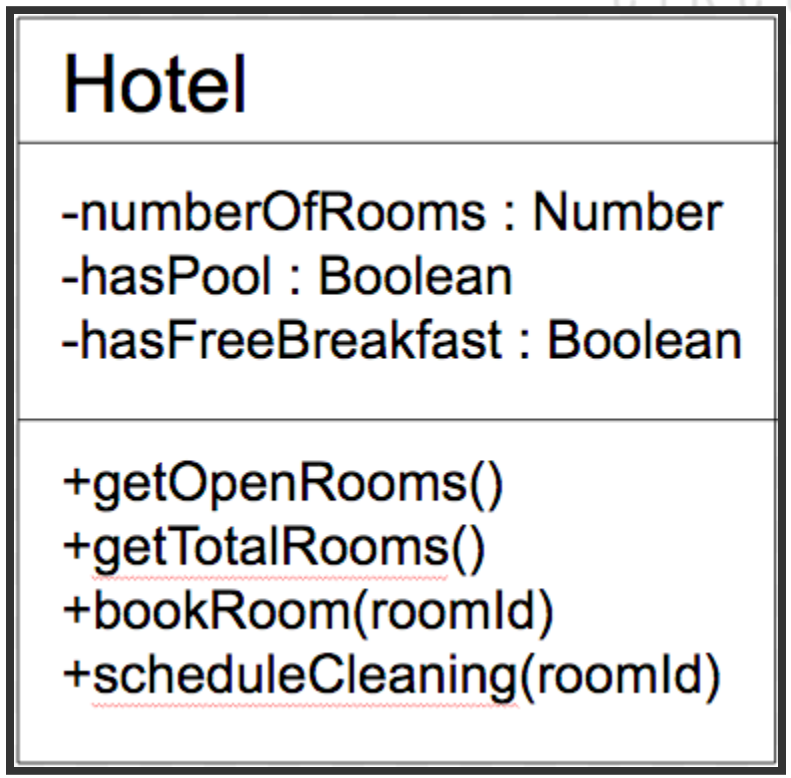
OOP

Imagine we want to write a program to manage a chain of hotels. We might *model* the chain's hotels by creating a `Hotel` class.

- A `Hotel` has data (total number of rooms, number of booked rooms for a given day, whether there's a pool, free continental breakfast, etc.).
- A `Hotel` can also have methods that act on that data (get the total number of rooms, get the number of booked rooms, book a room)

OOP

Sometimes classes are represented using what is called UML (Unified Modeling Language). It shows a class by breaking it down to it's state and behavior.



ARRAY

`Array` is a class that we've already come across.

What are some properties and methods that Array has?

KEY TAKE-AWAY: OBJECTS VS. CLASSES

Classes are like stamps or cookie cutters for creating *objects*. All Hotels (class) have some things in common. But each each individual Hotel (object) has some unique attributes.

Array is a class. All arrays have push, pop, shift, unshift, etc. But each individual array is an object and can have different elements and different length.

ES6 WARNING

ES6 is still pretty new. It adds some features to JavaScript that don't work everywhere.

OOP IN JS

JavaScript doesn't have classes like most OO languages. Even though ES6 - or ES2016, the newest version of JavaScript, adds a `class` keyword, it's still not *quite* the same.

```
class Hotel {
  constructor(numberOfRooms){
    this.totalRooms = numberOfRooms;
    this.bookedRooms= 0;
    this.hasPool = true;
  }
}

var laQuinta = new Hotel(109);
```

DEMO

OOP JS

Classes in JavaScript can also have functions that act on its data and give it behavior.

OOP IN JS

```
class Hotel {
  constructor(numberOfRooms){
    this.totalRooms = numberOfRooms;
    this.bookedRooms= [];
  }
  bookRoom(roomId){
    this.bookedRooms.push(roomId);
  }
  getOpenRooms(){
    return this.totalRooms - this.bookedRooms.length;
  }
  getTotalRooms(){
    return this.totalRooms;
  }
}

var laQuinta = new Hotel(109);
laQuinta.getOpenRooms(); // 109
laQuinta.bookRoom(101);
laQuinta.getOpenRooms(); // 108
```

DEMO

PROTOTYPES IN JS

If we inspect these objects after we create one, we will not see the methods on the obects.

```
laQuinta.hasOwnProperty('totalRooms'); // true
laQuinta.hasOwnProperty('bookRoom');   // false
```

What's up with that?!

The methods are stored on the object's prototype.

KEY TAKE-AWAY: PROTOTYPE

The *prototype* holds all the common stuff in a class (push, pop, etc. for an array). Every individual Array object has all it's own unique properties *plus* all the properties from the prototype.

KEY TAKE-AWAY: HASOWNPROPERTY

hasOwnProperty is a method you can use to determine whether a property is part of the individual object or part of its common class.

```
var array = new Array();

console.log(array.length);
console.log(array.pop);
console.log(array.hasOwnProperty("length")); // true (individual object)
console.log(array.hasOwnProperty("pop")); // false (common class)
```

DEMO

OO JS

We can even set it up so that we can create *subclasses*. Again, this works a little differently in JavaScript because it is not a pure OO language. But imagine we'd like to be able to create a `Luxury Hotel` object that inherits from `Hotel` and has behaviors specific to the subclass but are not part of the original `Hotel`.

OO JS

Functions or properties added to the prototypes are not shared with the parent class. This code sample is too big for this window so let's look at a demo.

DEMO

OVERLOADING VS. OVERRIDING

Like with almost everything else these terms mean slightly different things in JS than they do in other traditional object-oriented languages. Here's what they mean *in JavaScript*.

- **Overloading** - Calling a method with the same name and using it in a different way.
- **Overriding** - When a subclass has a method with the same name as its parent class but uses it in a different way.

EXERCISE

OBJECT ORIENTED ANIMALS

CREATE SOME OBJECT ORIENTED SHAPE CLASSES

1. Create a `class` called `Animal` that has a constructor that takes a `species` argument and assigns it to the species property of the object.
2. Define a method called `eat()` that console logs "Nom Nom Nom!".
3. Define a method called `speak()` that console logs `<Animal.species> makes a sound`.
4. Design a subclass that extends `Animal` called `Dog`. In the dog, constructor accept a `name` argument and sets it to a name property on the object. The constructor should also set the species to "Dog".
5. Override the `speak()` for the `Dog` class so that it logs `<Dog.name> says hello!`

OOJS EXERCISE

Test your implementation with this code

```
var koala = new Animal('koala');
var barkley = new Dog("Barkley");

koala.speak(); // "Koala makes a sound."
koala.eat();   // "Nom Nom Nom"
koala.species; // "Koala"

barkley.speak(); // "Barkley says hello!"
barkley.species; // "Dog"
barkley.eat();   // "Nom Nom Nom"
```

BONUS CHALLENGE

1. Create a separate kind of subclass for Dog broken into Working breeds and Companion breeds. Give each subclass a specific property `job` and set it to "herding" or 'guarding' for the working breeds and 'being a pal' for the Companion breeds.
2. Further subclass into specific breeds (i.e. Chihuahua and St. Bernard) with a `weight` property and logic in the constructors to limit the weight properties for an adult of the breed.

RESOURCES

- The Principles of Object-Oriented JavaScript
- JavaScript Constructor Functions Vs Factory Functions - for a dissenting opinion on the matter

RECAP

You should understand and be able to use:

- Hoisting
- Closures
- IIFE's
- OOJS