Lesson 6:
Scope and Objects

Learning Objectives

- Understand variable scope, and global vs. local scope (the last part of lesson 4)
- Create, access and modify objects
- Create objects using constructor functions
- Add methods to objects through their constructors
- Understand the basic purpose of object-oriented programming

Up till now:

- We learned most fundamental data types
- We learned to write and use functions
- We learned to use control flow structures to manage the execution of code in our programs

Today…

- Variable scope
    - Scope is a super-important concept with a couple of quirks in JavaScript that we need to be aware of
- Objects! The last of our major datatypes

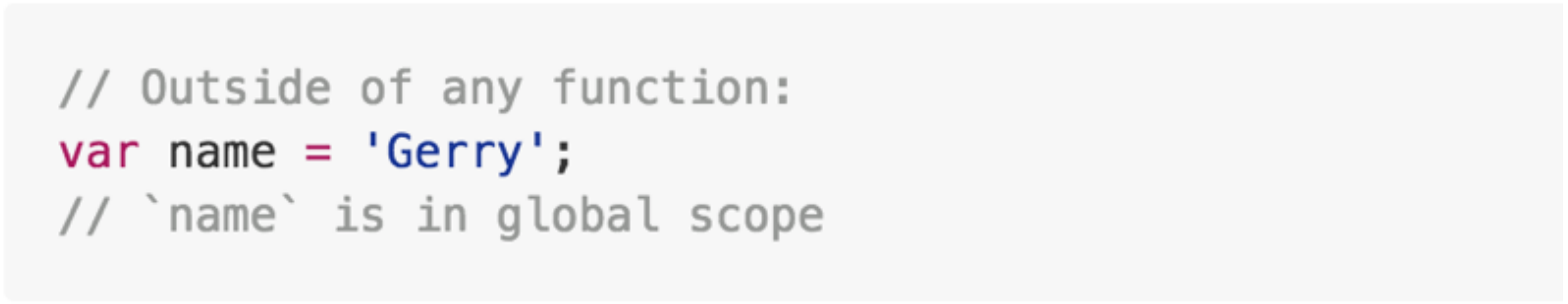
Demo: scope

Intro to Scope

- In any program, when we're executing one part of the code, other parts of the code may or may not be visible.
- By "visible", we mean that our part of the code has knowledge of, and access to, the variables that are defined in other parts of the code.
- When we declare variables inside functions, these things are NOT visible to the rest of the code, outside of the function.
- We say that those variables are in the function's "scope".

Global scope

- Before you write a line of JavaScript, you're in what we call the "global scope".
- Any variable you declare will be outside of any function, and is public, which we call "global".
- This means any other part of your code can reference this variable, meaning know that it exists, know what its value is, or change it.

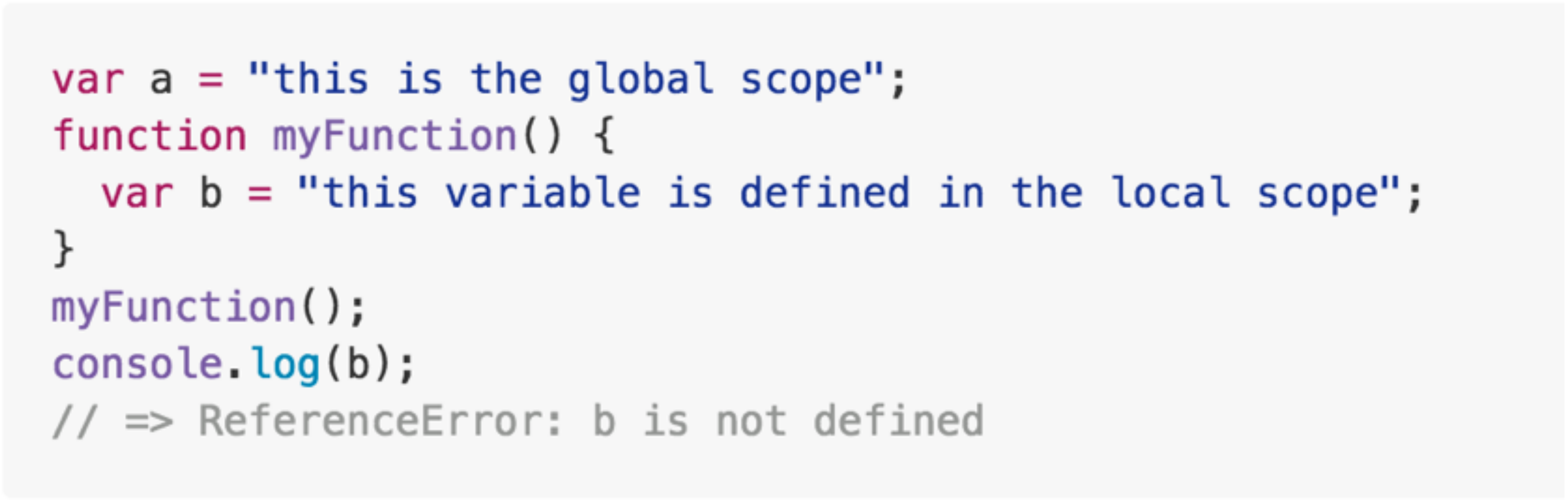


```
// Outside of any function:
var name = 'Gerry';
// `name` is in global scope
```

Local scope

- If a variable is declared inside a function, it is referred to as "local", and has a "local scope".
- A variable with local scope cannot be referenced outside of that function.
- A new copy of the variable is created every time you run the function
- The variable is thrown away when the function finishes executing

Local scope

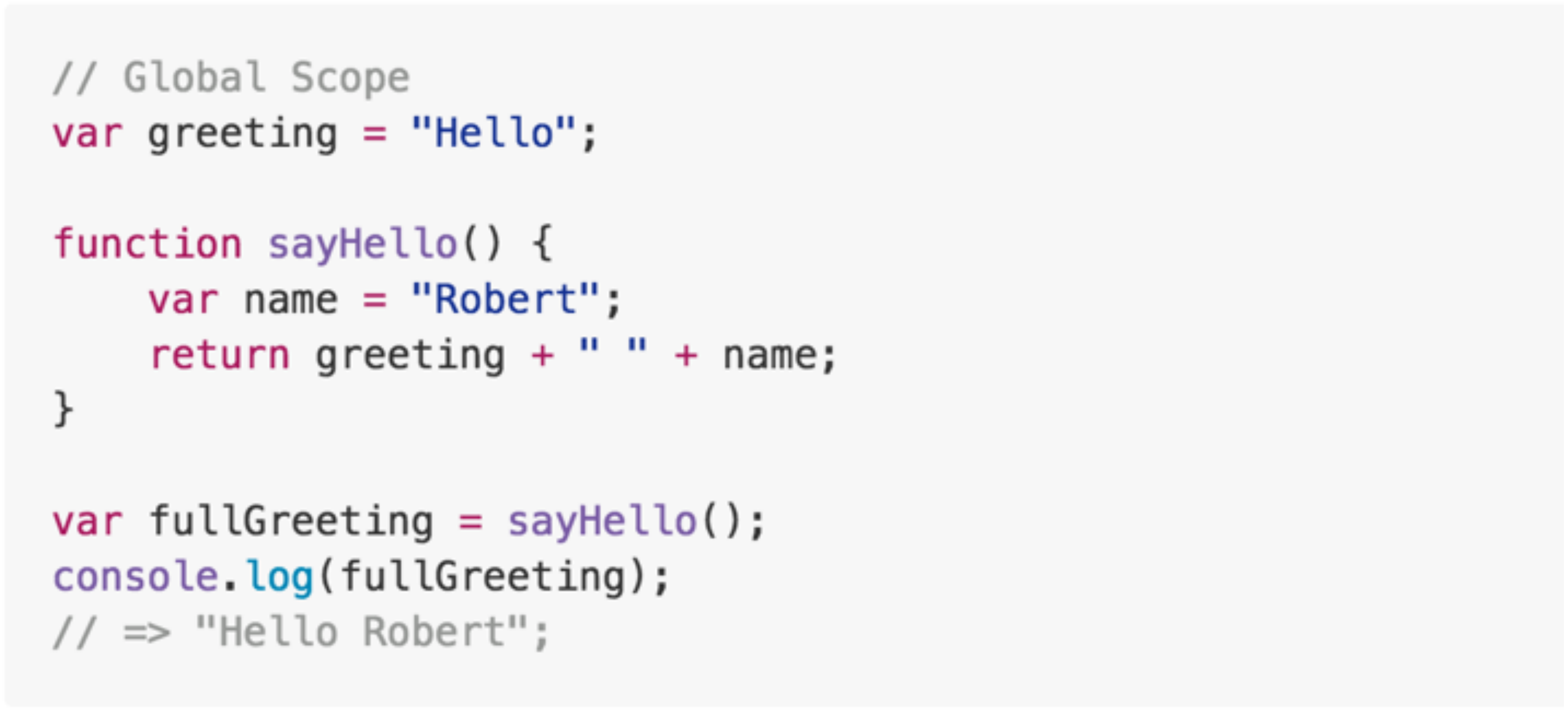


```javascript
var a = "this is the global scope";
function myFunction() {
  var b = "this variable is defined in the local scope";
}
myFunction();
console.log(b);
// => ReferenceError: b is not defined
```

- In this case, we get a reference error because the variable **b** is not accessible outside the scope of the function in which it is defined.

Access to the parent scope

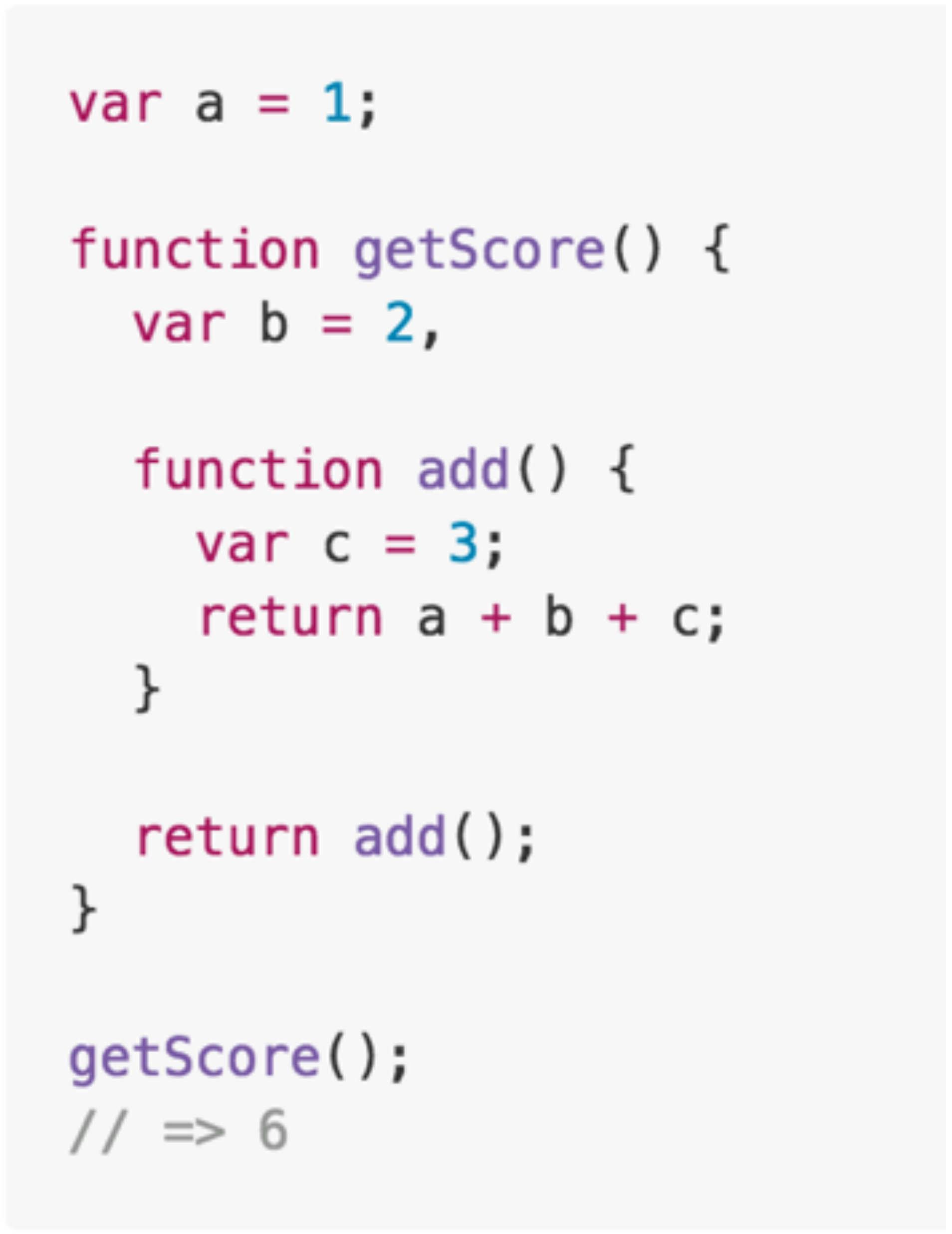
- A function can access variables of the parent scope.
- In the example below, in the function, you have access to all the variables defined in the global scope.



```
// Global Scope
var greeting = "Hello";

function sayHello() {
    var name = "Robert";
    return greeting + " " + name;
}

var fullGreeting = sayHello();
console.log(fullGreeting);
// => "Hello Robert";
```

Nested scope

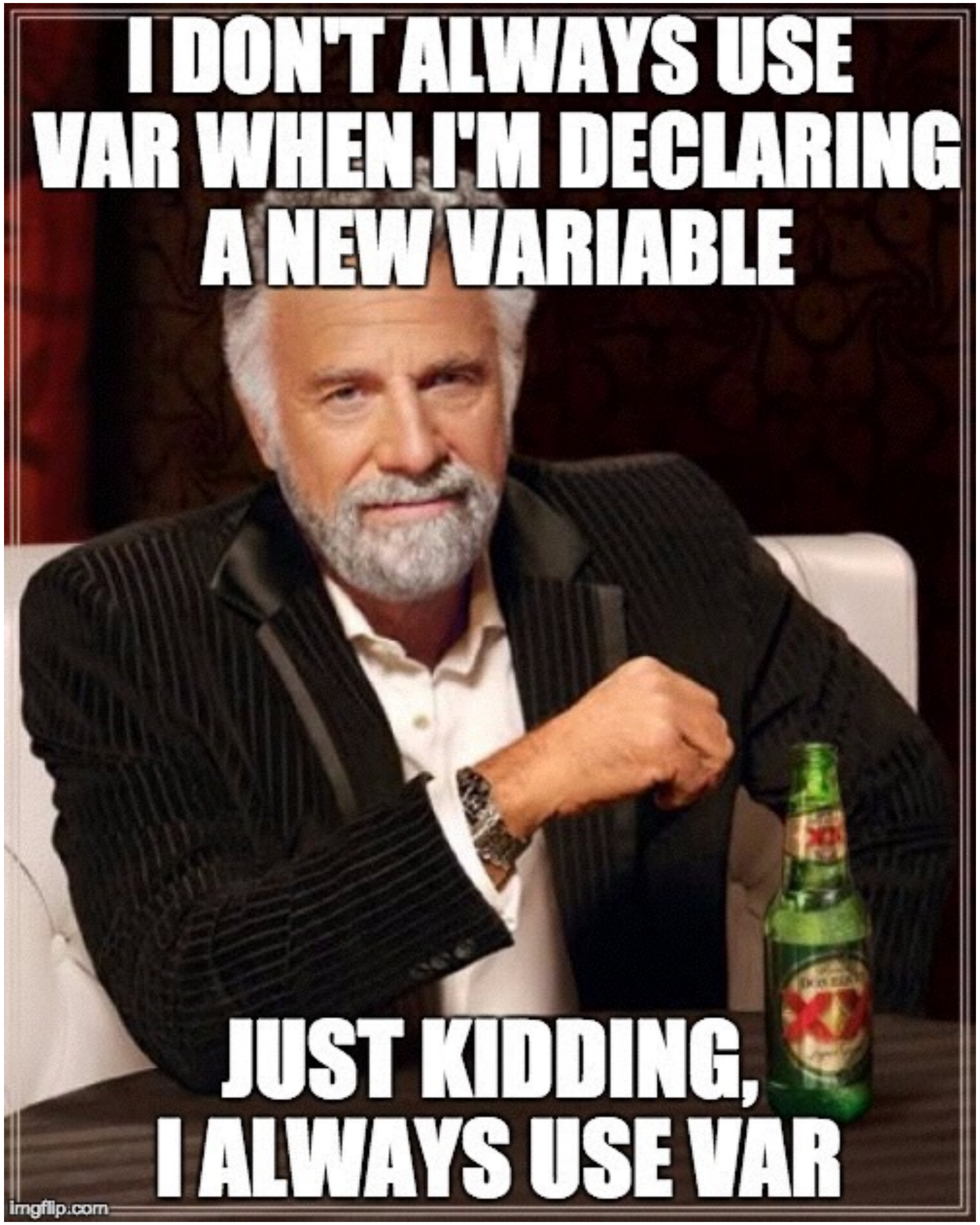
- When a function is defined inside another function, it is possible to access variables defined in the parent from the child:



```
var a = 1;

function getScore() {
  var b = 2,

  function add() {
    var c = 3;
    return a + b + c;
  }

  return add();
}

getScore();
// => 6
```

Codealong: scope

- cd ~/GA-JS
- mkdir lesson6
- cd lesson6
- touch scope.js
- subl scope.js
- Go back and forth between Sublime Text and the terminal, editing the code in Sublime and typing `node scope.js` in the terminal


I DON'T ALWAYS USE VAR WHEN I'M DECLARING A NEW VARIABLE
JUST KIDDING, I ALWAYS USE VAR
imgflip.com

Practice on your own

- Make a global variable called **counter**. Make two functions called **countUp** and **countDown** which each take an argument and change the value of counter by that much, and a third function **printCounter** (which console.logs the value of **counter**).

- Example:

```
printCounter();
// => 5

countUp(9);

printCounter();
// => 14
```

Demo: objects

Objects! (rationale)

- So far the only way we can store collections of related values is using arrays (ordered lists)
- As our code gets more complex, it will need more structure than that
- Objects are collections of “key-value pairs”
- For example, **age: 46**, or **name: “Richard”**

What objects do

- Objects in JavaScript serve two main purposes:
    - as a simple structured data store of key-value pairs, which we'll go over first
    - as full-blown Objects as in Object-Oriented Programming, a powerful paradigm and way of structuring code which we'll go over in the second half of the class.

Creating objects

- 2 ways to create empty objects:

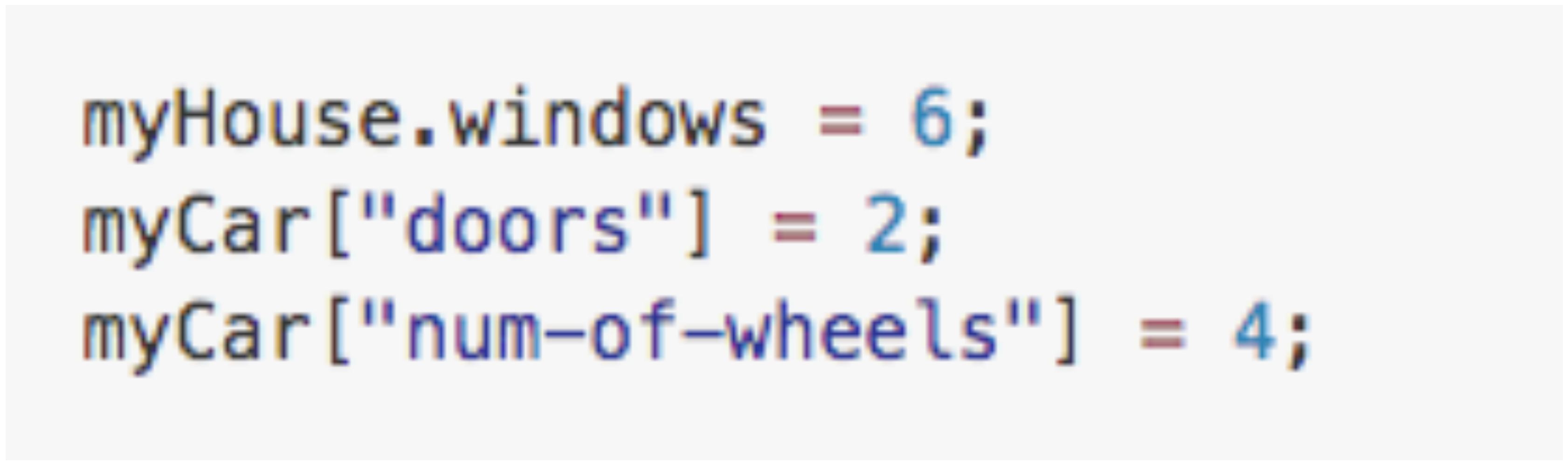
```
var myHouse = {};
var myCar = new Object();
```

- Create new objects with data:

```
var myMotorcycle = {
  wheels: 2,
  color: "blue",
  maxSpeed: 300
}
```

Setting properties

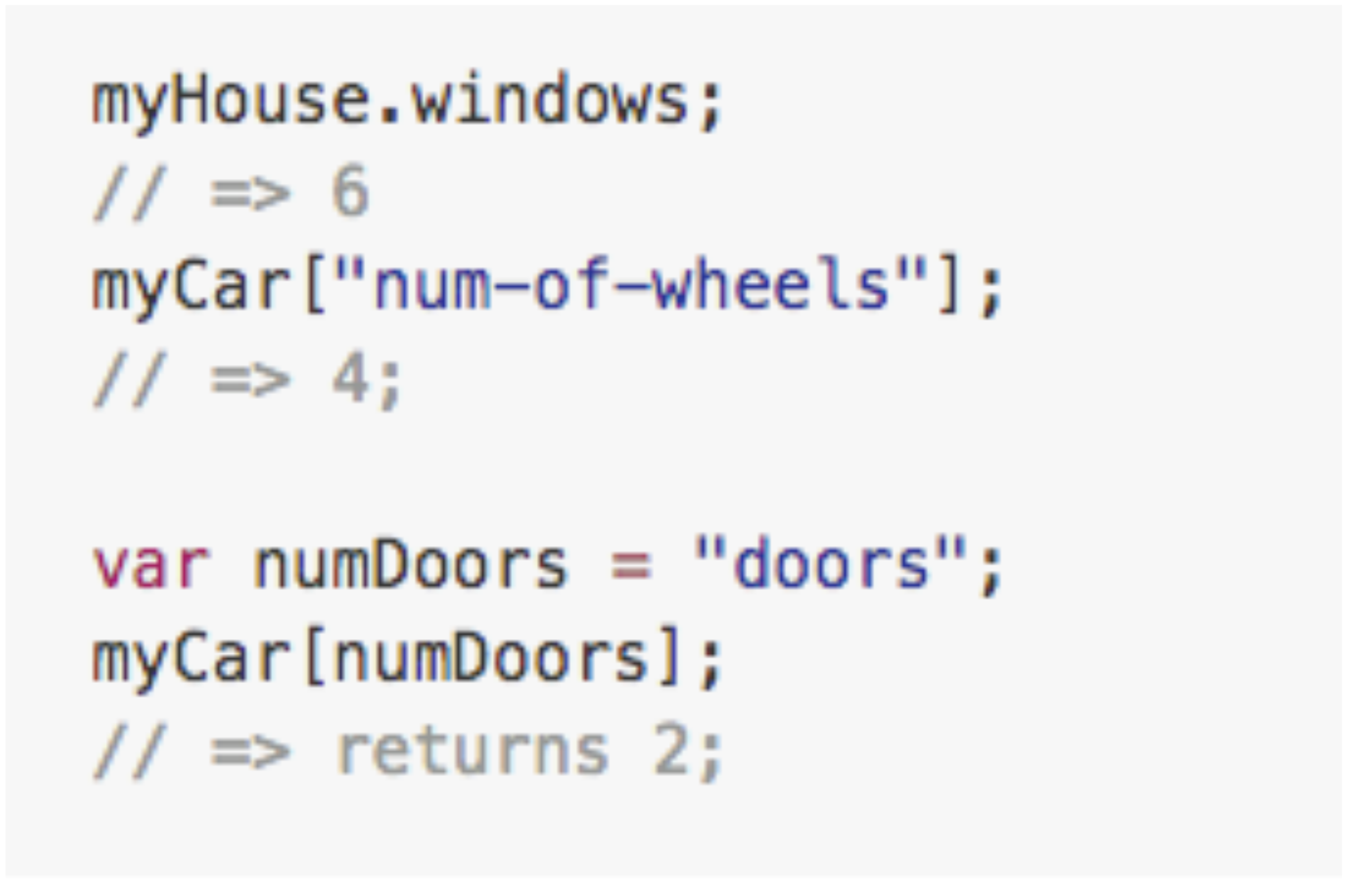
- 2 ways to add properties: dot notation and bracket notation.
  - (need bracket notation if your property name has weird characters in it)



```
myHouse.windows = 6;
myCar["doors"] = 2;
myCar["num-of-wheels"] = 4;
```

Getting properties

- Can also use variables when using bracket notation:



```
myHouse.windows;
// => 6
myCar["num-of-wheels"];
// => 4;

var numDoors = "doors";
myCar[numDoors];
// => returns 2;
```

Codealong: objects

- touch objects.js
- subl objects.js
- Go back and forth between Sublime Text and the terminal, editing the code in Sublime and typing `node objects.js` in the terminal

Practice on your own

- Make a person object with a **"name"** and an **"eyeColor"**.
- Now assign it another property, **"height"**, and give that property a value. Print out all three of these values, using both **richard.name** syntax and **richard["name"]** syntax.
- **Bonus:** assign another property to your person object, **"address"**, that is itself an object with four properties: **“streetNumber”**, **“streetName”**, **“city”**, **“zip”**

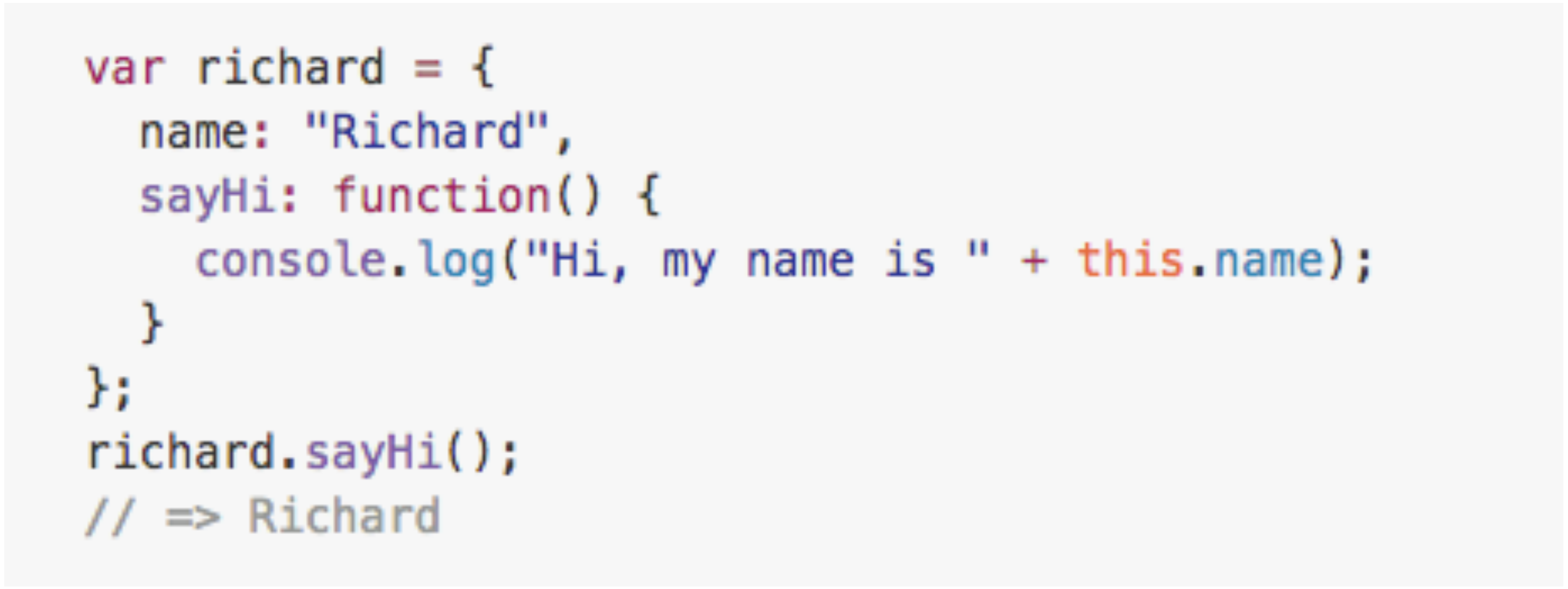
Demo: methods

Intro to Methods

- When a property of an object is a function, we call that a “method”
- We use the methods to get information about, and manipulate, the properties of the object.
- We can call our object methods the same way we call our object properties through the dot notation, with the main difference being that we add () at the end of our statement.

Intro to 'this'

- We often use the **this** keyword inside a method,
- **this** is a very special word in JavaScript, to refer to the object itself:



```
var richard = {
  name: "Richard",
  sayHi: function() {
    console.log("Hi, my name is " + this.name);
  }
};
richard.sayHi();
// => Richard
```

Object Oriented Programming: the 30-second version

- OOP models data in our code to match how we think of the real world: as nouns, descriptive information about those nouns, and verbs

- Ex: I am a **person**, I have a **name**, I can **say** my name

- OOP is often extremely powerful and useful,

- May seem convoluted and hard to understand when things get more abstract

- No perfect system has been developed which both computers and humans can easily understand. They're working on it!

Codealong: methods

- subl objects.js
- Go back and forth between Sublime Text and the terminal, editing the code in Sublime and typing `node objects.js` in the terminal

Practice on your own: methods

- Make a person object that has a **name** property and an **age** property and a **sayMyAge** method that says"Hi, I'm Richard, I'm 46".

- **Bonus:** Make another method, **setAge** that changes the **age** property of the object, unless the number that is passed in is a negative number, in which case it DOESN'T change the age, and it console.logs "failed to set age".

Demo: Constructors

Constructors and Prototype methods

- Use a **constructor function** to create a bunch of objects that all have the same structure (that is, the same property names) and also the same methods.
- Called like a regular JavaScript function but with the **new** keyword.

Creating an empty object using a constructor

- Make a constructor function that doesn’t do anything:

```
function Person() {};
```

- Use the **new** keyword when calling your constructor function

```
var clark = new Person();
var bruce = new Person();
clark;
// => {}
bruce;
// => {}
```

Setting properties with the constructor

- So far we have been setting property values by hand every time we create an object.

- Constructors help us create a blueprint for our data, so every object created with a constructor has the same property names and the same methods

- We use parameters in our constructor function to set the new property values.

Setting properties with the constructor (part 2)

```
function Superhero(newFirstName, newSuperheroName) {
  this.firstName = newFirstName;
  this.superheroName = newSuperheroName;
};

var superman = new Superhero('Clark', 'Superman');
console.log(superman.firstNAme + ' is ' + superman.superheroName);
```

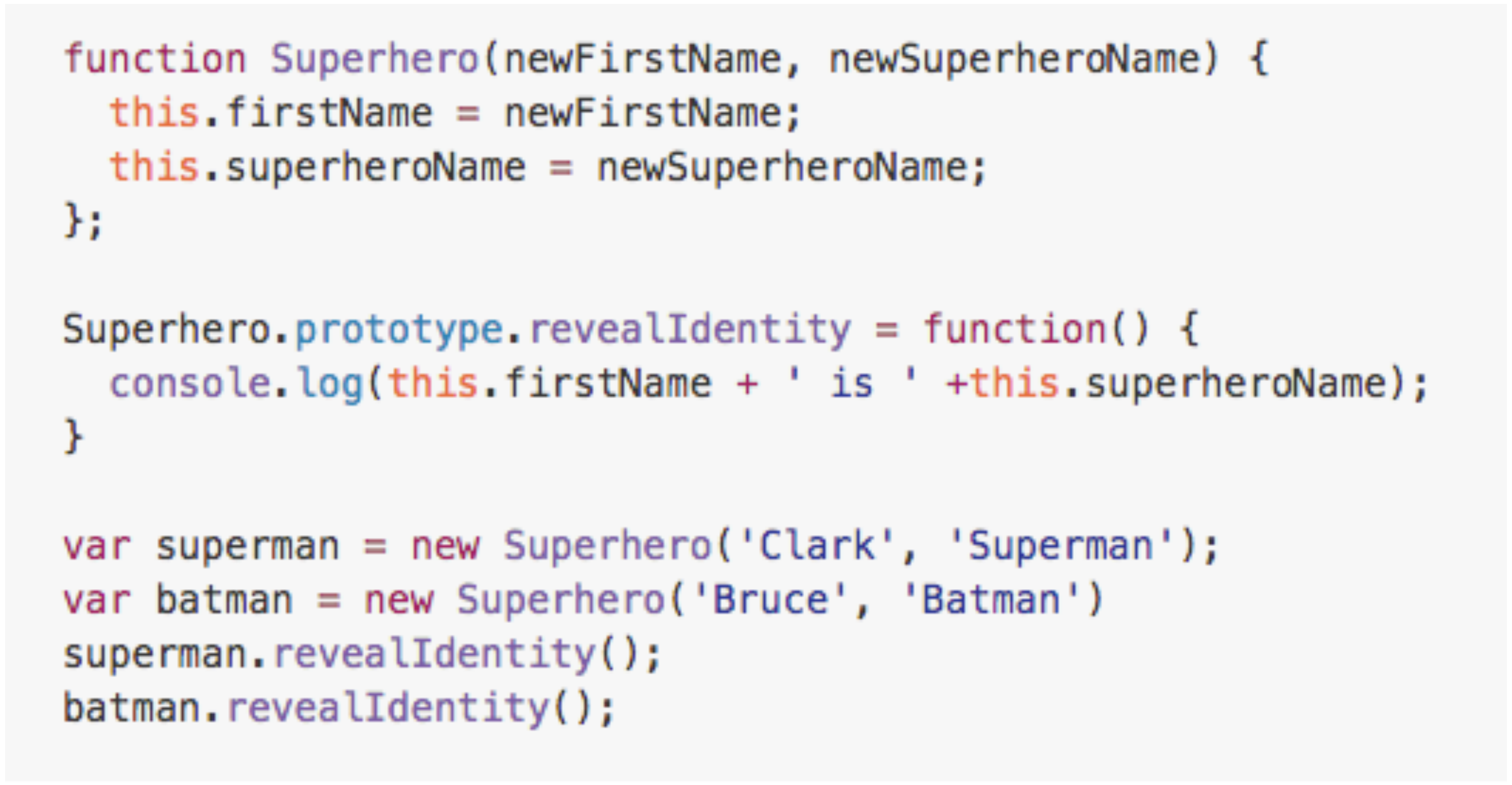
What does a constructor function actually do?

1. Creates a new object

2. Makes the special **this** variable point to the new object, while the constructor is executing.

3. Returns the new object (note the lack of a return statement — it kind of does it automagically)

Adding common methods using a constructor



```
function Superhero(newFirstName, newSuperheroName) {
  this.firstName = newFirstName;
  this.superheroName = newSuperheroName;
};

Superhero.prototype.revealIdentity = function() {
  console.log(this.firstName + ' is ' +this.superheroName);
}

var superman = new Superhero('Clark', 'Superman');
var batman = new Superhero('Bruce', 'Batman')
superman.revealIdentity();
batman.revealIdentity();
```

- Whoa, what’s “prototype”? Don’t worry for now!

What's going on in the previous slide?

- We're making our **Superhero** constructor function, as before

- We're adding a method to **Superhero**'s prototype, which causes all objects created with the **Superhero** constructor to have access to that method

- Most crucially: *when the **revealIdentity** method is called, the **this** variable in the method will refer to whatever object it's been called on*.

Monkeys!

- Create a new file called “monkeys.js”
- Work with a partner to create a monkey object, which has the following properties:

name
species
foodsEaten

- And the following methods:
  - **eatSomething**: takes a thing to eat (as a string) and adds it to foodsEaten.
  - **introduce**: introduces itself (using console.log), including its name, species, and what it's eaten.
- Create 3 monkeys total. Use a constructor function, with arguments passed in, to create your monkeys. Give the monkeys access to the methods above using the constructor’s “prototype” syntax.
- Exercise your monkeys by retrieving their properties and using their methods. Practice using both syntaxes for retrieving properties (dot notation and square bracket notation)