JAVA FULL STACK PROGRAM

Session: JavaScript

OUTLINE

- JavaScript Version
  - Overview
  - Version
- JavaScript Basics
  - Syntax
  - Data Type
  - Comparison
  - Scope & Closure
  - this

- JavaScript Advance
  - IIFE & Anonymous Function
  - Asynchronous
  - Constructor
  - Prototype
  - Inheritance
  - ES6 Feature

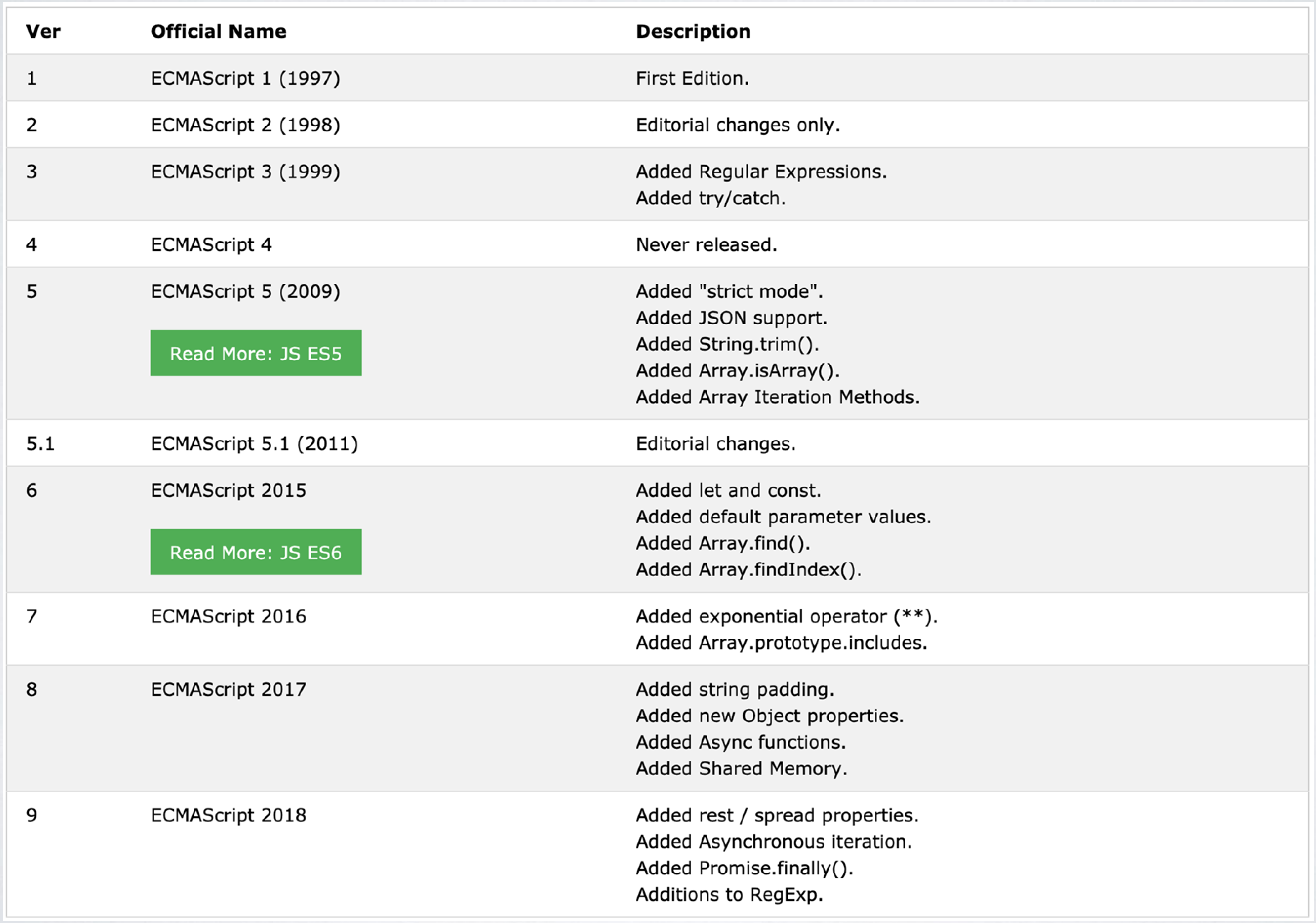
WHAT IS JAVASCRIPT

- JavaScript is a dynamic computer programming language. It is lightweight and most commonly used as a part of web pages, whose implementations allow client-side script to interact with the user and make dynamic pages.
- It is an interpreted programming language with object-oriented capabilities.
- The ECMA-262 Specification defined a standard version of the core JavaScript language.

Introduction

- JavaScript has two forms
    - Inline: written inside html file
    - External: written in a separate JavaScript file and imported into html
        - Usually put in <head> tag
- Suggested IDE: Visual Studio Code (VS Code)

JAVASCRIPT VERSION



Ver	Official Name	Description
1	ECMAScript 1 (1997)	First Edition.
2	ECMAScript 2 (1998)	Editorial changes only.
3	ECMAScript 3 (1999)	Added Regular Expressions. Added try/catch.
4	ECMAScript 4	Never released.
5	ECMAScript 5 (2009) Read More: JS ES5	Added "strict mode". Added JSON support. Added String.trim(). Added Array.isArray(). Added Array Iteration Methods.
5.1	ECMAScript 5.1 (2011)	Editorial changes.
6	ECMAScript 2015 Read More: JS ES6	Added let and const. Added default parameter values. Added Array.find(). Added Array.findIndex().
7	ECMAScript 2016	Added exponential operator (**). Added Array.prototype.includes.
8	ECMAScript 2017	Added string padding. Added new Object properties. Added Async functions. Added Shared Memory.
9	ECMAScript 2018	Added rest / spread properties. Added Asynchronous iteration. Added Promise.finally(). Additions to RegExp.

JAVASCRIPT VERSION

- ECMAScript (or ES) is a scripting-language specification standardized by Ecma International
  - It was created to standardize JavaScript to help foster multiple independent implementations
- Now most of the company will use ES6 as their JS version, but you should be aware of what is added in ES5 and ES6.
  - https://www.w3schools.com/js/js_es5.asp
  - https://www.w3schools.com/js/js_es6.asp

ES5

- There will be a lot of changes in ES5, most of them are easy to understand.
  - Arrays methods: https://www.w3schools.com/jsref/jsref_obj_array.asp
  - JSON: *JSON.parse(), JSON.stringify()*
  - Date: *Date.now()*
- There is only one which might be a little bit confusing
  - use strict: it enables the strict mode in JS

HOISTING

- Hoisting can be thought up as a general way of thinking about how execution contexts (specifically the creation and execution phases) work in JavaScript.
- Conceptually, for example, a strict definition of hoisting suggests that variable and function declarations are physically moved to the top of your code, but this is not in fact what happens. Instead, **the variable and function declarations are put into memory during the *compile* phase but stay exactly where you typed them in your code.**

```
catName("Chloe");

function catName(name) {
  console.log("My cat's name is " + name);
}
```

```
console.log(num);
var num;
num = 6;
```

```
function catName(name) {
  console.log("My cat's name is " + name);
}

catName("Tigger");
```

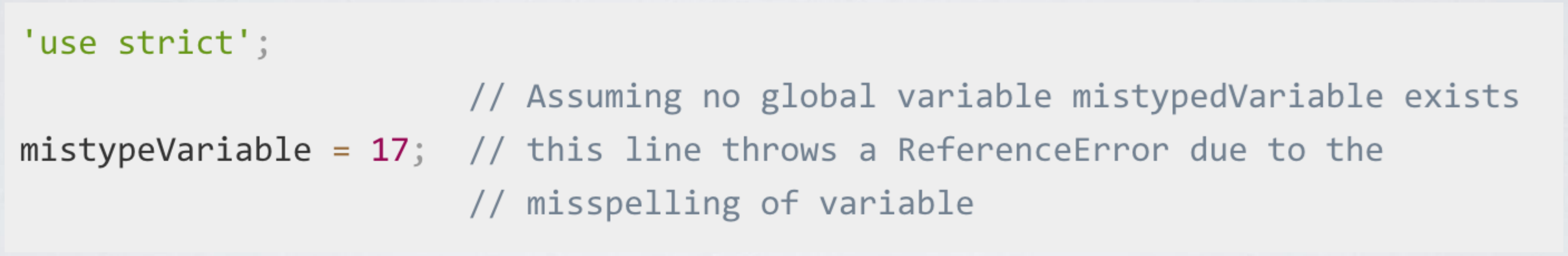
```
num = 6;
console.log(num);
var num;
```

- JavaScript only hoists declarations, not initializations

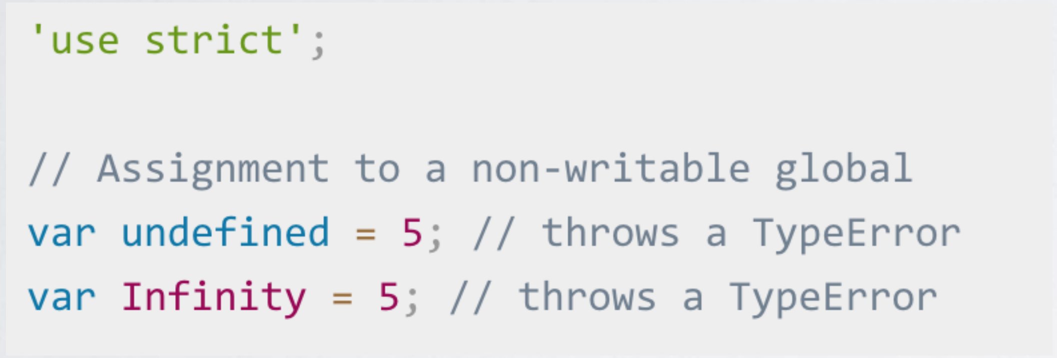
USE STRICT

- The purpose of "use strict" is to indicate that the code should be executed in "strict mode”
  - With strict mode, you can not, for example, use undeclared variables.
- Strict mode makes several changes to normal JavaScript semantics:
  - Eliminates some JavaScript silent errors by changing them to throw errors.
  - Fixes mistakes that make it difficult for JavaScript engines to perform optimizations: strict mode code can sometimes be made to run faster than identical code that's not strict mode.
  - Prohibits some syntax likely to be defined in future versions of ECMAScript.

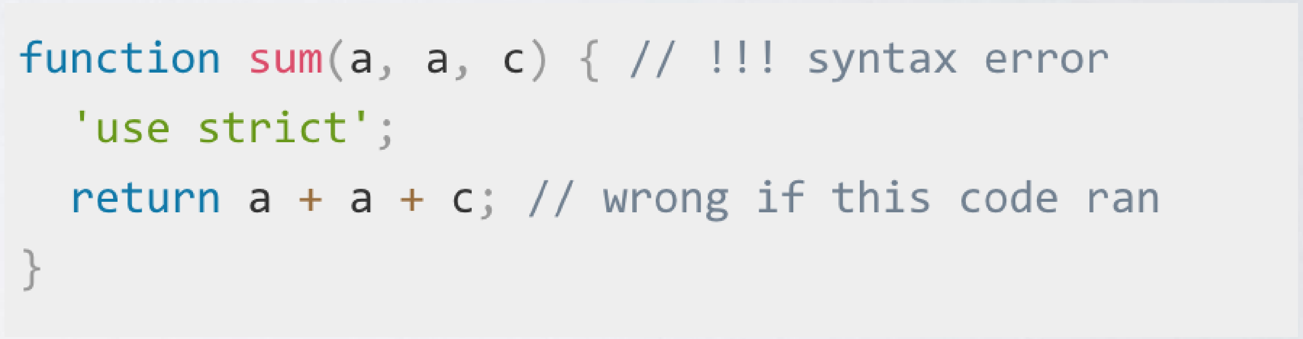
USE STRICT



```
'use strict';
                          // Assuming no global variable mistypedVariable exists
mistypeVariable = 17;   // this line throws a ReferenceError due to the
                          // misspelling of variable
```



```
'use strict';

// Assignment to a non-writable global
var undefined = 5; // throws a TypeError
var Infinity = 5; // throws a TypeError
```



```
function sum(a, a, c) { // !!! syntax error
  'use strict';
  return a + a + c; // wrong if this code ran
}
```

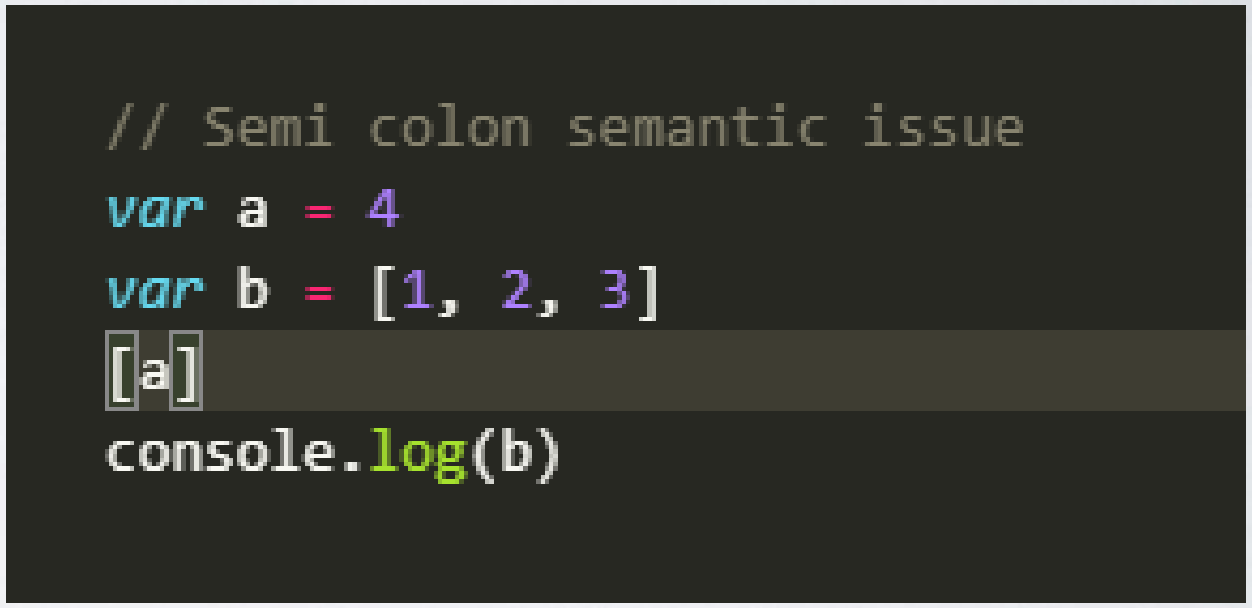
- The *this* keyword in functions behaves differently in strict mode.
  - The *this* keyword refers to the object that called the function.
  - If the object is not specified, functions in strict mode will return *undefined* and functions in normal mode will return the global object (window)
- Keywords reserved for future JavaScript versions can NOT be used as variable names in strict mode.
  - implements, interface, let, package, private, protected, public, static

JAVASCRIPT BASIC

- Just like Java, JavaScript also has flow control statements (if, for, while), operators (+, -, *, /, %, **), String, Date and Arrays.
  - For String, it has several built-in functions: https://www.w3schools.com/jsref/jsref_obj_string.asp
  - For Array, it has several built-in functions: https://www.w3schools.com/jsref/jsref_obj_array.asp
- I will go through something basic concepts which are different from Java and confusing.

SYNTAX

- JS syntax is similar to Java
  - Each statement ends with " ; "
  - Use { } to construct block
  - Unlike Java, JS does not require user to add " ; " at the end of each statement, JS engine will add that for user. (Automatic Semicolon Insertion)
  - But it is still recommended to add " ; " manually. As some times the it will change the semantic of the statement.



```
// Semi colon semantic issue
var a = 4
var b = [1, 2, 3]
[a]
console.log(b)
```

DATA TYPE

- In JavaScript there are two different kinds of data: **primitives**, and **objects**.
  - A primitive is simply a data type that is not an object, and has no methods
- In JS, there are seven primitive data types:
  - Boolean — Only has two values: true, false
  - Number — Numbers can be written with or without a decimal point. In JavaScript, Number is a numeric data type in the double-precision 64-bit floating point format (IEEE 754),A number can also be NaN (not a number)
  - bigint --- BigInt is a built-in object whose constructor returns a bigint primitive — also called a BigInt value, or sometimes just a BigInt — to represent whole numbers larger than 2^53 - 1 (Number.MAX_SAFE_INTEGER)
  - String — Strings must be inside of either double or single quotes. In JS, Strings are immutable.
  - Undefined — Only one value: undefined
  - Symbol — A Symbol is an immutable primitive value that is unique.
  - Null — Only one value: null, Structural Root Primitive,typeof instance === "object". Special primitive type having additional usage for its value: if object is not inherited, then null is shown;

DATA TYPE

- Objects are **not** a primitive data Type.
  - typeof instance === "object". Special non-data but Structural type for any constructed object instance also used as data structures: new Object, new Array, new Map, new Set, new WeakMap, new WeakSet, new Date and almost everything made with new keyword;
    - new Object:
    - An object is a collection of properties.
    - These properties are stored in key/value pairs.
    - Properties can reference any type of data, including objects and/or primitive values.

- Function : a non-data structure, though it also answers for typeof operator: typeof instance === "function". This is merely a special shorthand for Functions, though every Function constructor is derived from Object constructor.

- JavaScript is a **loosely typed** language.

  - This means you don’t have to declare a variable’s type. JavaScript automatically determines it for you

    - var car = ‘Honda’

    - car = 2015

DATA CONVERSION

- Implicit Data Conversion
  - When doing operation, data type will be converted automatically
  - We can use + to combine integer and string and JS will not throw exception

https://dev.to/promhize/what-you-need-to-know-about-javascripts-implicit-coercion-e23#:~:text=Javascript's%20implicit%20coercion%20simply%20refers,feature%20that%20is%20best%20avoided.

COMPARISON

- equality operator(==) and identity operator(===)
  - The == operator will compare for equality after doing any necessary type conversions
  - The === operator will not do the conversion, so if two values are not the same type === will simply return false
- What are outputs for following statement
  - ‘0’ == 0
  - ‘0’ === 0
  - 5 != ‘5’
  - false == undefined
  - [1, 2] == ‘1, 2’

SCOPE

- There are two scopes in JavaScript:
  - Local Scope — Variables declared within a JavaScript function, become **LOCAL** to the function.
    - Local variables have **Function scope**: They can only be accessed from within the function.
  - Global Scope — A variable declared outside a function, becomes **GLOBAL**.
    - A global variable has **global scope**: All scripts and functions on a web page can access it.

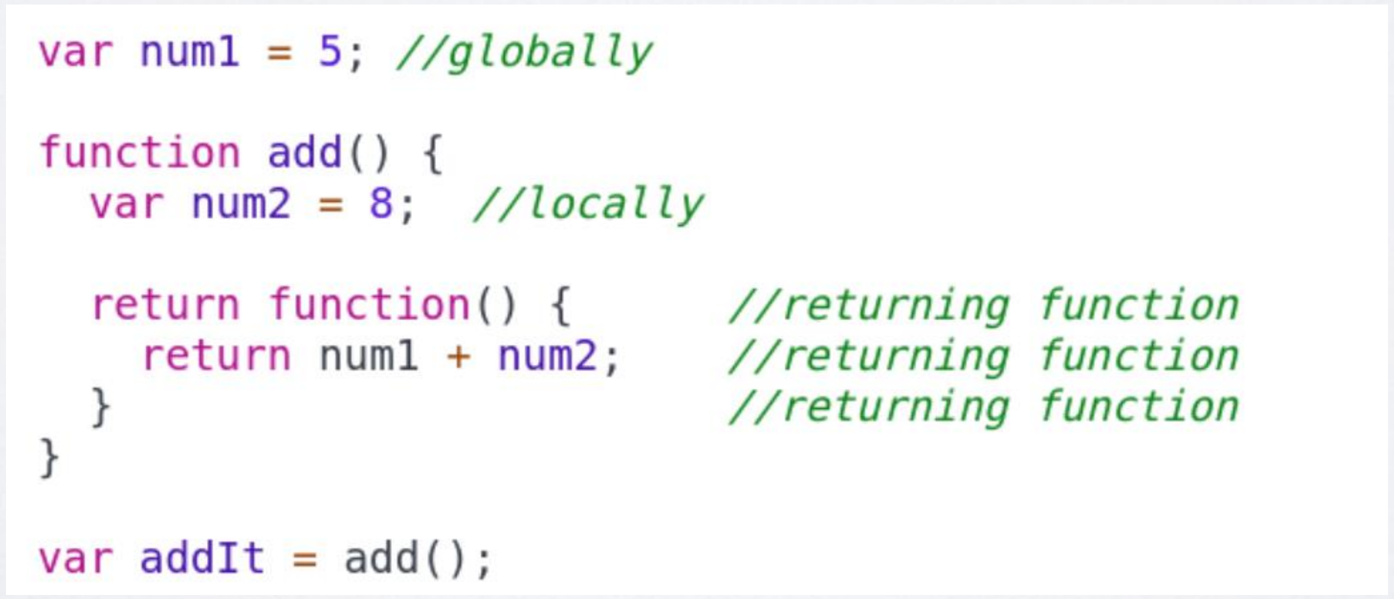
Global Scope – when we open a JS document, we are at global scope. Anything outside a function is under global scope

Local Scope – local scope is bind with function, we can have multiple local scope

Block will not create new scope, variables defined in block will be stored in upper level scope

CLOSURE

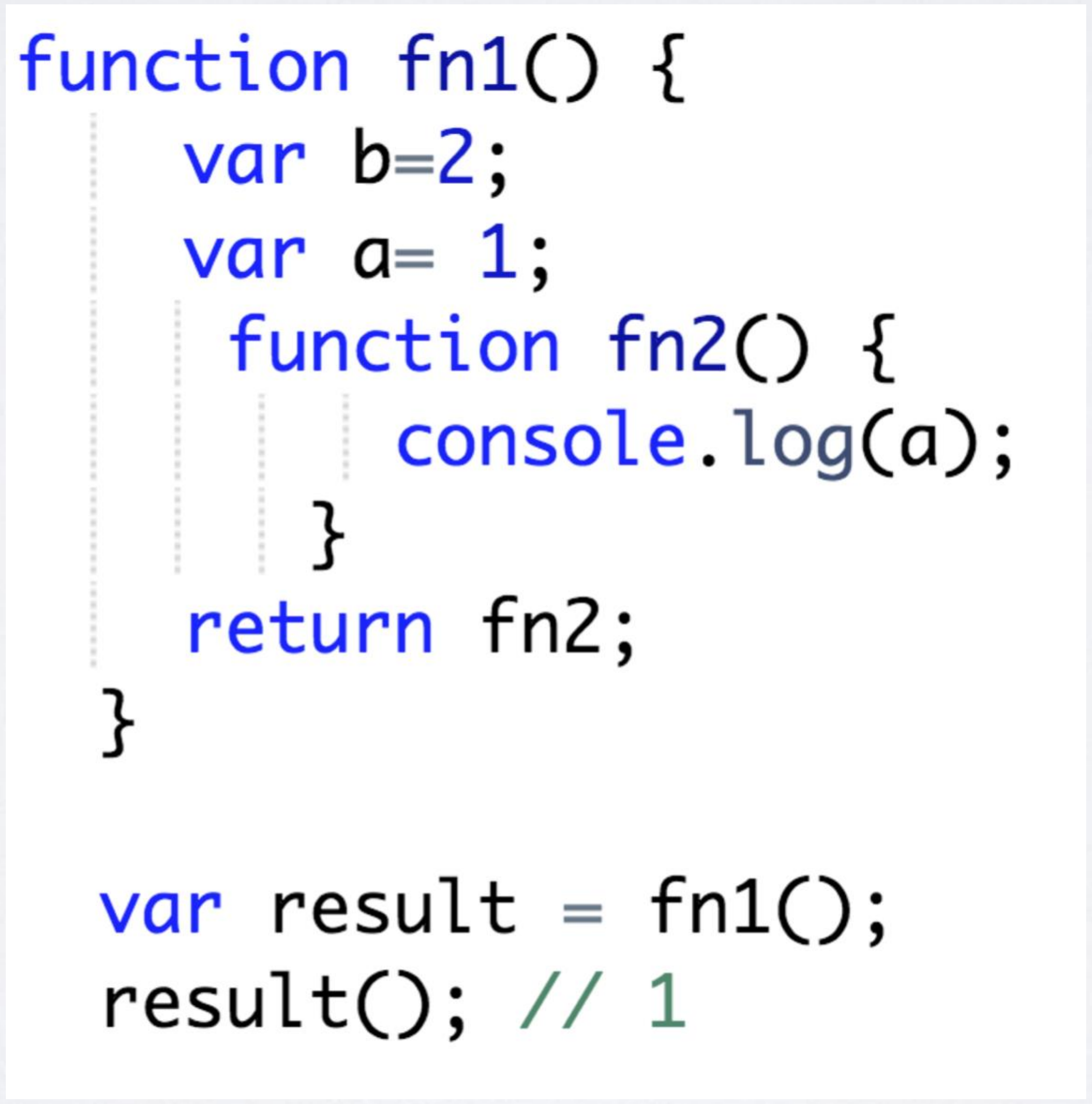
- Lexical Scope (Static Scope) — Lexical Scope is the same as the nested local scopes but there be little bit difference between them
    - If the current scope doesn’t find the value then it moves outside to that scope and tries to find the value there
    - This process continues until it finds the value



```
var num1 = 5; //globally

function add() {
  var num2 = 8;  //locally

  return function() {       //returning function
    return num1 + num2;     //returning function
  }                         //returning function
}

var addIt = add();
```

CLOSURE

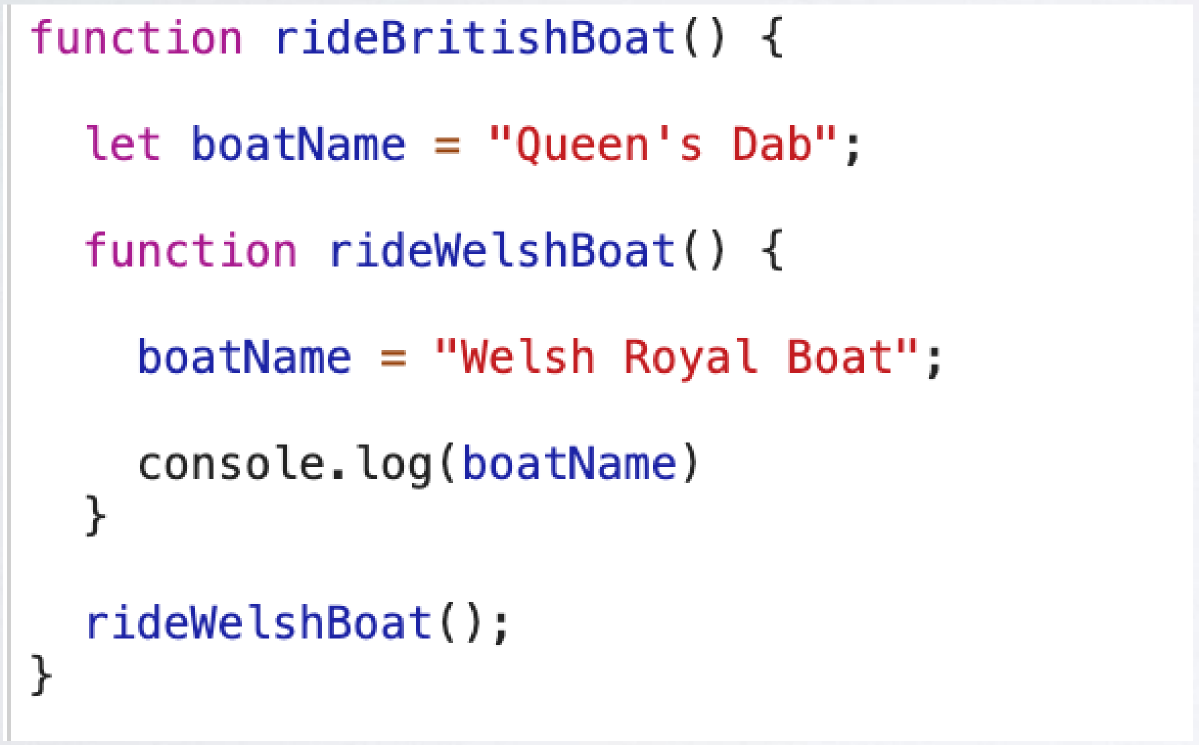
- Closure — *Closures* are simply functions that access data outside their own scope



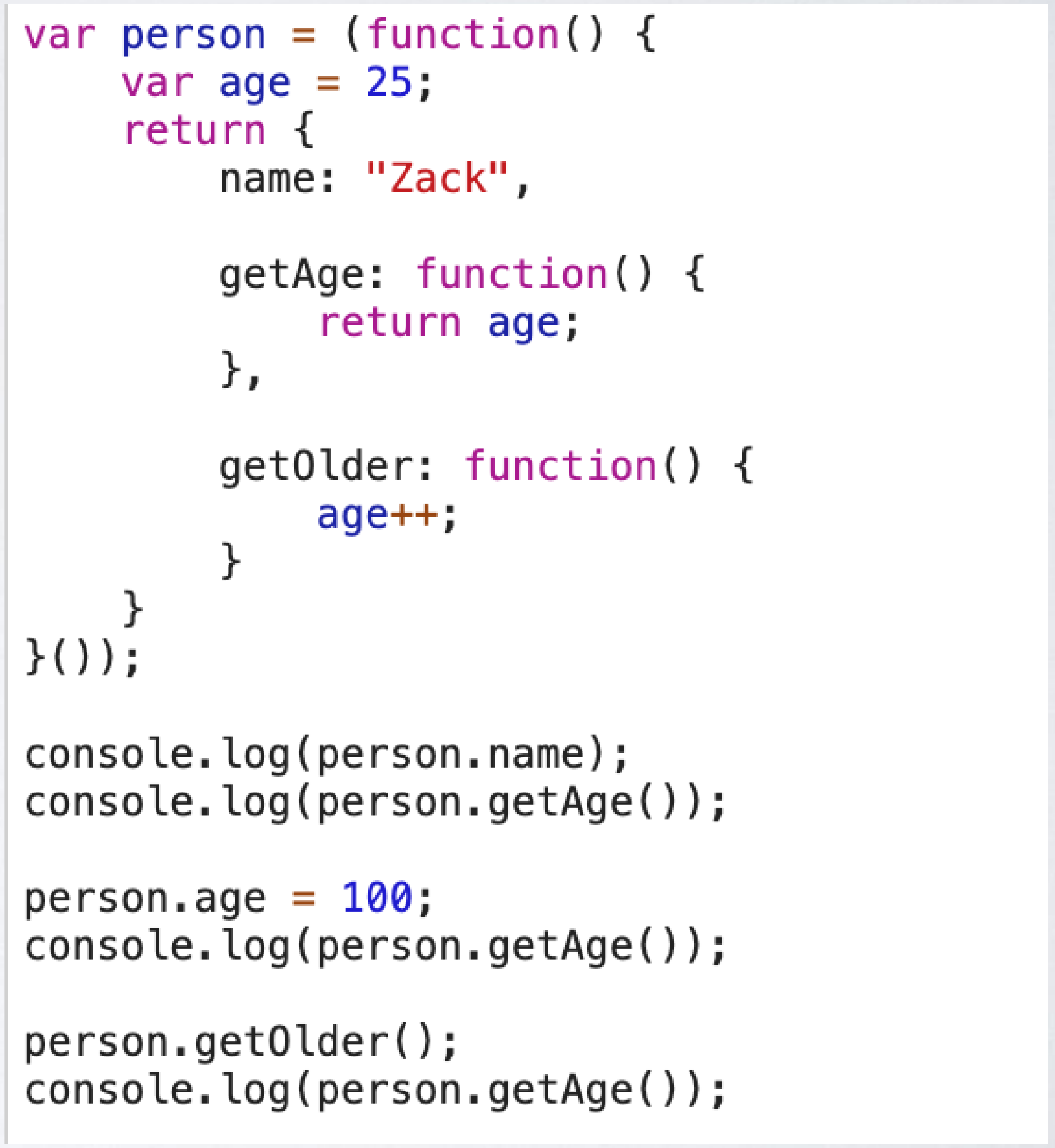
```
function fn1() {
    var b=2;
    var a= 1;
     function fn2() {
         console.log(a);
       }
    return fn2;
  }

  var result = fn1();
  result(); // 1
```

CLOSURE

- What is the output for following code?



```
function rideBritishBoat() {

  let boatName = "Queen's Dab";

  function rideWelshBoat() {

    boatName = "Welsh Royal Boat";

    console.log(boatName)
  }

  rideWelshBoat();
}
```



```
var person = (function() {
    var age = 25;
    return {
        name: "Zack",

        getAge: function() {
            return age;
        },

        getOlder: function() {
            age++;
        }
    }
}());

console.log(person.name);
console.log(person.getAge());

person.age = 100;
console.log(person.getAge());

person.getOlder();
console.log(person.getAge());
```

THIS

- In JavaScript, this keyword has different meaning in different context
  - Method
  - Alone
  - Function
  - Event — In an event, this refers to the element that received the event.

THIS

- In a method, this refers to the owner object.

```
var o = {
  prop: 37,
  f: function() {
    return this.prop;
  }
};

console.log(o.f()); // 37
```

```
var o = {prop: 37};

function independent() {
  return this.prop;
}

o.f = independent;

console.log(o.f()); // 37
```

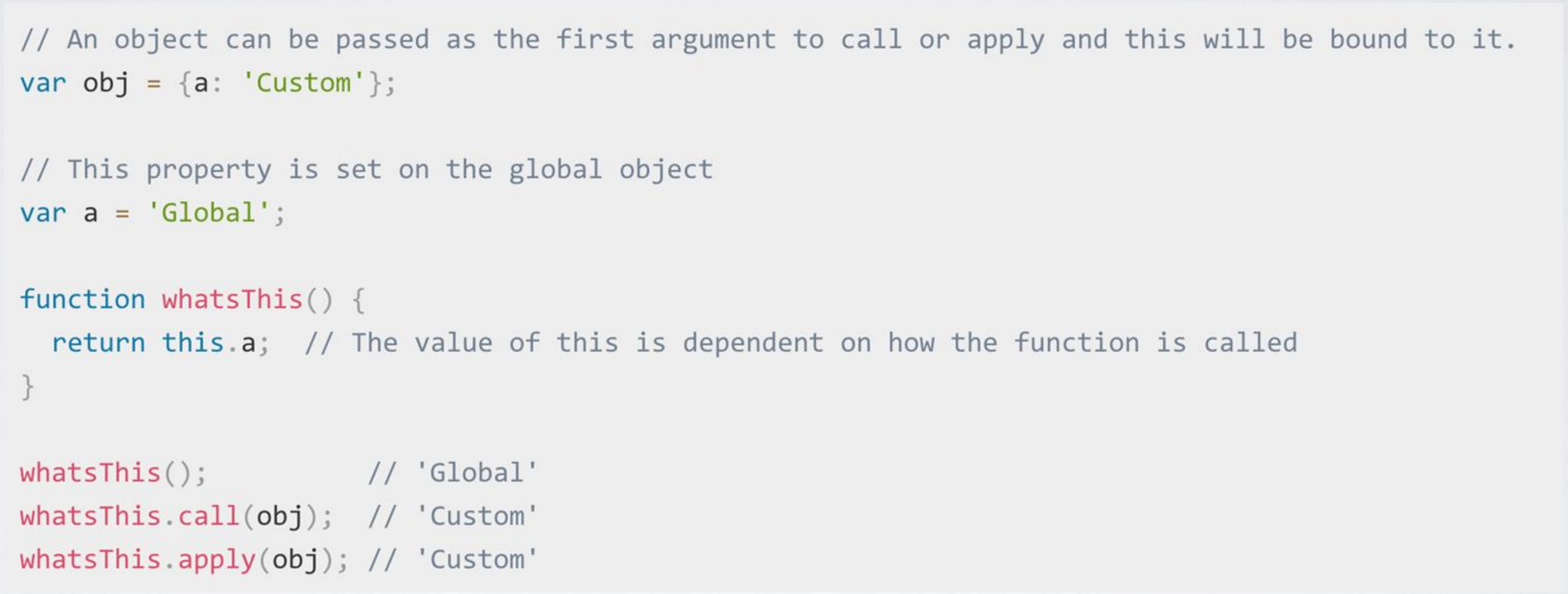
THIS

- Alone, this refers to the global object.

```
// In web browsers, the window object is also the global object:
console.log(this === window); // true

a = 37;
console.log(window.a); // 37

this.b = "MDN";
console.log(window.b)  // "MDN"
console.log(b)         // "MDN"
```

THIS

- this keyword in function has two different behavior
  - In normal mode, this refers to the global object
  - In strict mode, this is undefined

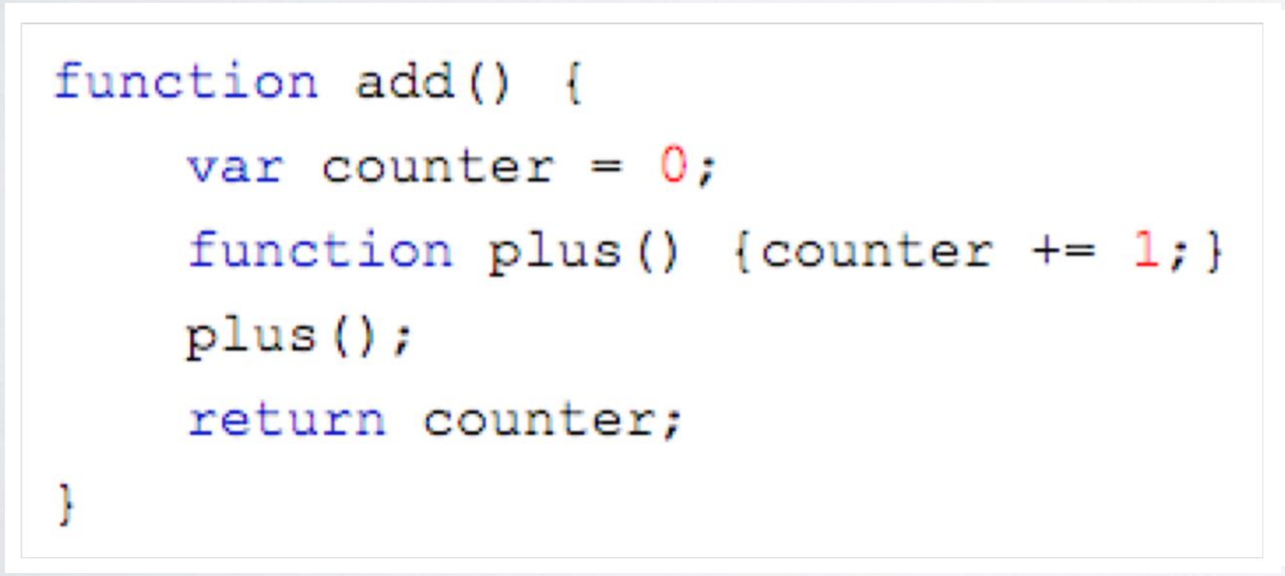


```
// An object can be passed as the first argument to call or apply and this will be bound to it.
var obj = {a: 'Custom'};

// This property is set on the global object
var a = 'Global';

function whatsThis() {
  return this.a;  // The value of this is dependent on how the function is called
}

whatsThis();          // 'Global'
whatsThis.call(obj);  // 'Custom'
whatsThis.apply(obj); // 'Custom'
```

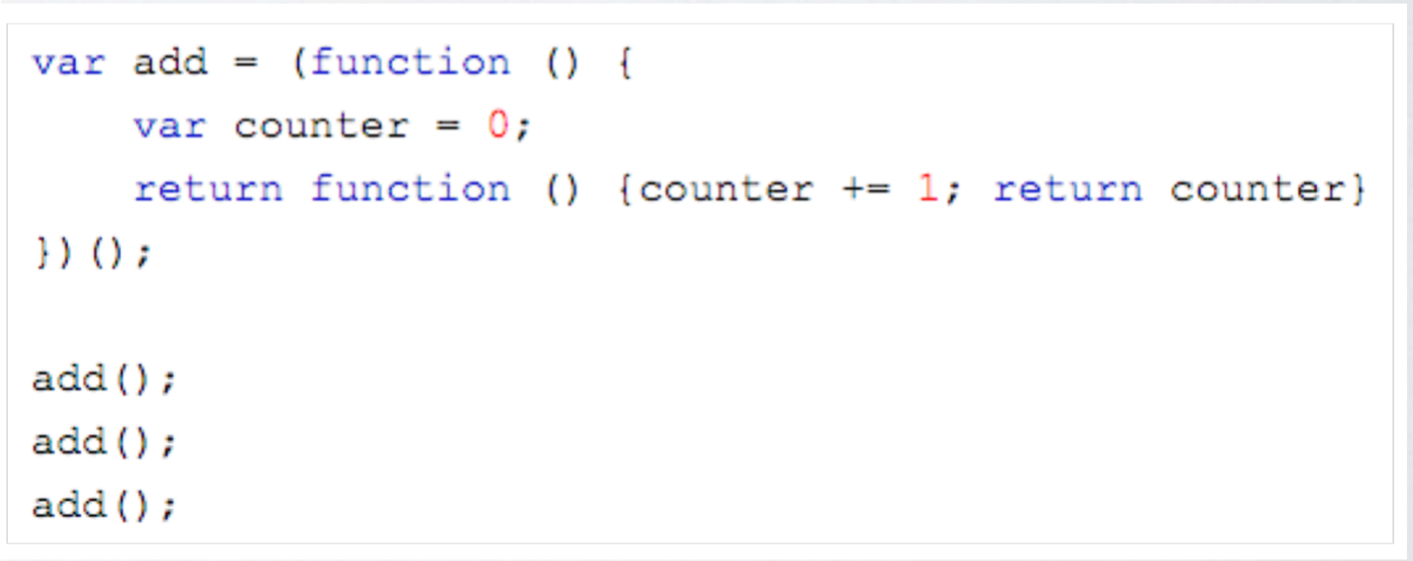
- call() — allows for a function/method belonging to one object to be assigned and called for a different object.
- apply() — apply is very similar to call(), except for the type of arguments it supports. You use an arguments array instead of a list of arguments (parameters).

IIFE

- IIFE stands for Immediately Invoked Function Expression
- An IIFE is a function expression that is defined and then called immediately to produce a result.
- That function expression can contain any number of local variables that aren't accessible from outside that function.

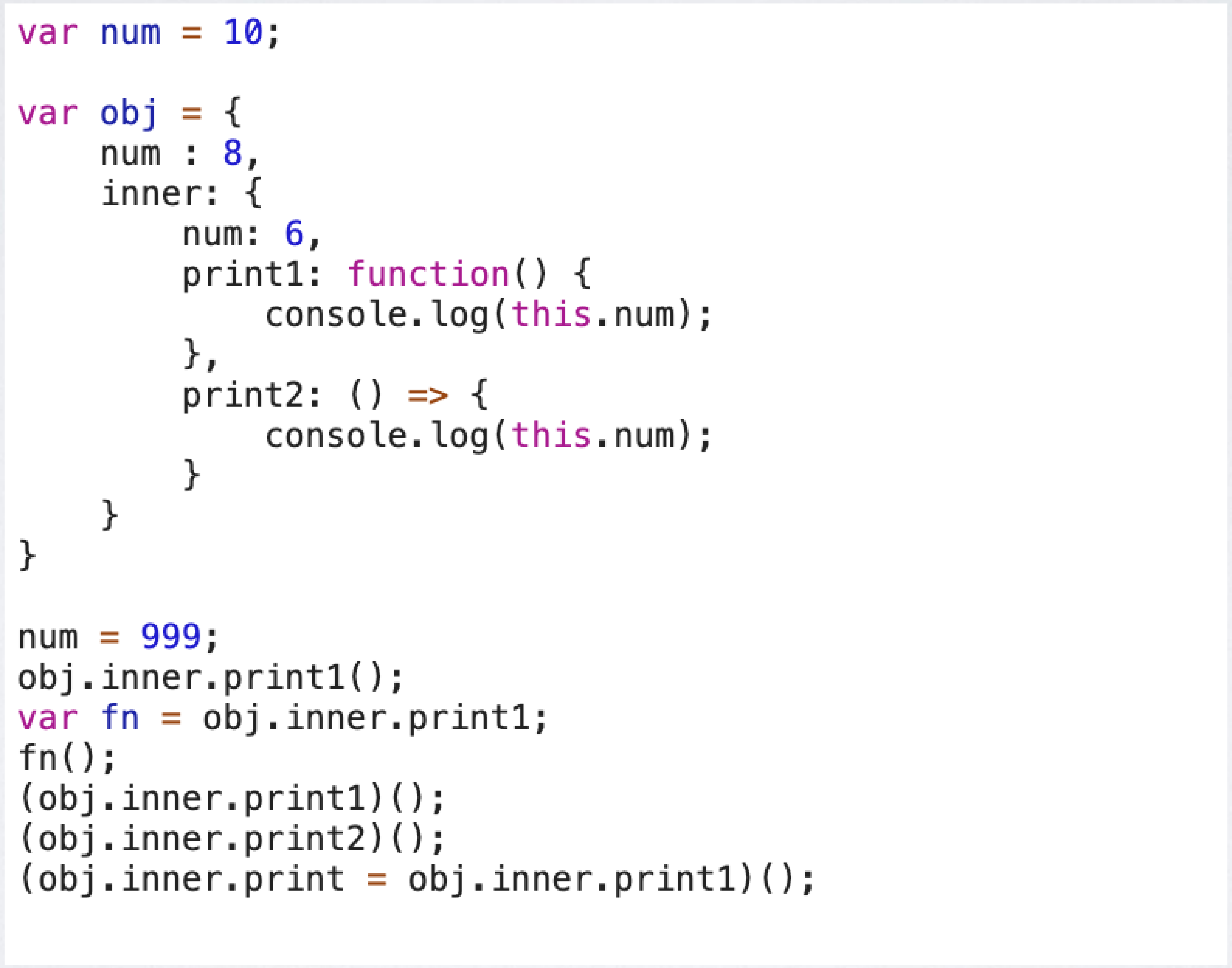


```
function add() {
    var counter = 0;
    function plus() {counter += 1;}
    plus();
    return counter;
}
```



```
var add = (function () {
    var counter = 0;
    return function () {counter += 1; return counter}
})();

add();
add();
add();
```

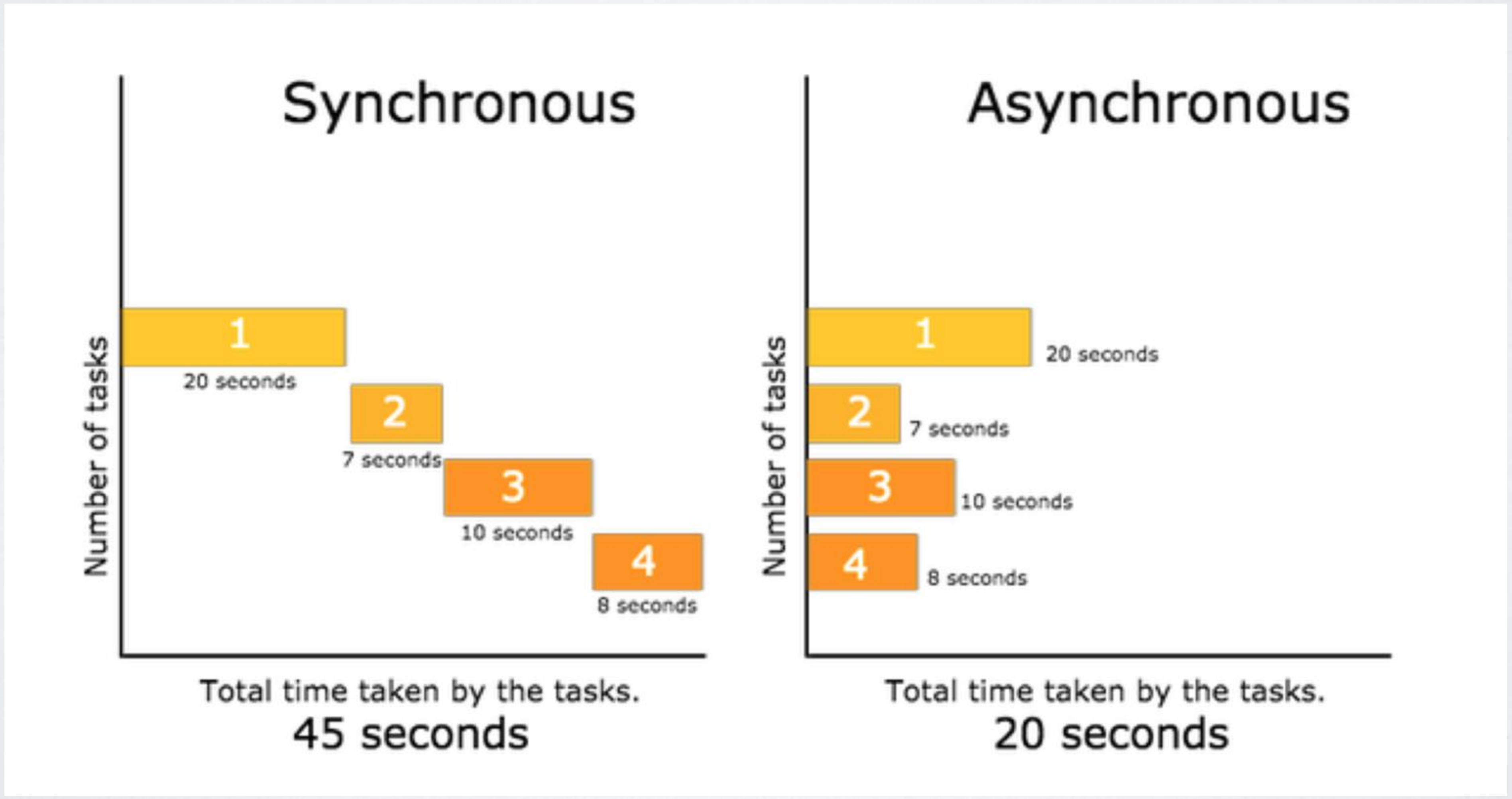
IIFE



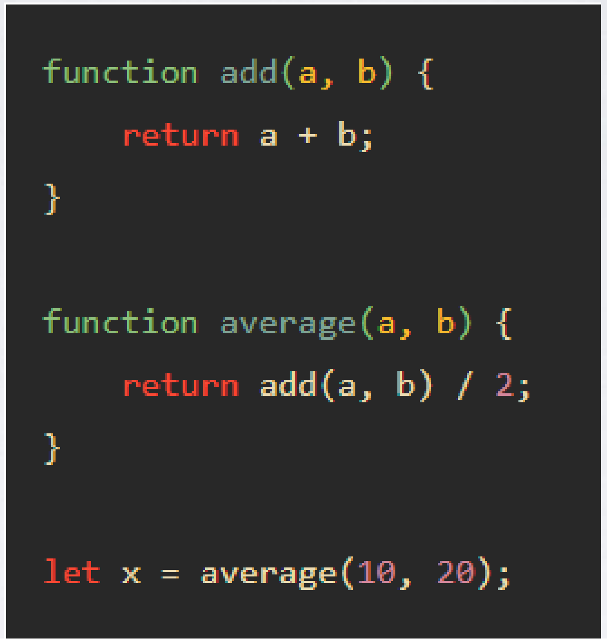
```javascript
var num = 10;

var obj = {
    num : 8,
    inner: {
        num: 6,
        print1: function() {
            console.log(this.num);
        },
        print2: () => {
            console.log(this.num);
        }
    }
}

num = 999;
obj.inner.print1();
var fn = obj.inner.print1;
fn();
(obj.inner.print1)();
(obj.inner.print2)();
(obj.inner.print = obj.inner.print1)();
```

ASYNCHRONOUS

- In *asynchronous* programs, you can have two lines of code (L1 followed by L2), where L1 schedules some task to be run in the future, but L2 runs before that task completes.

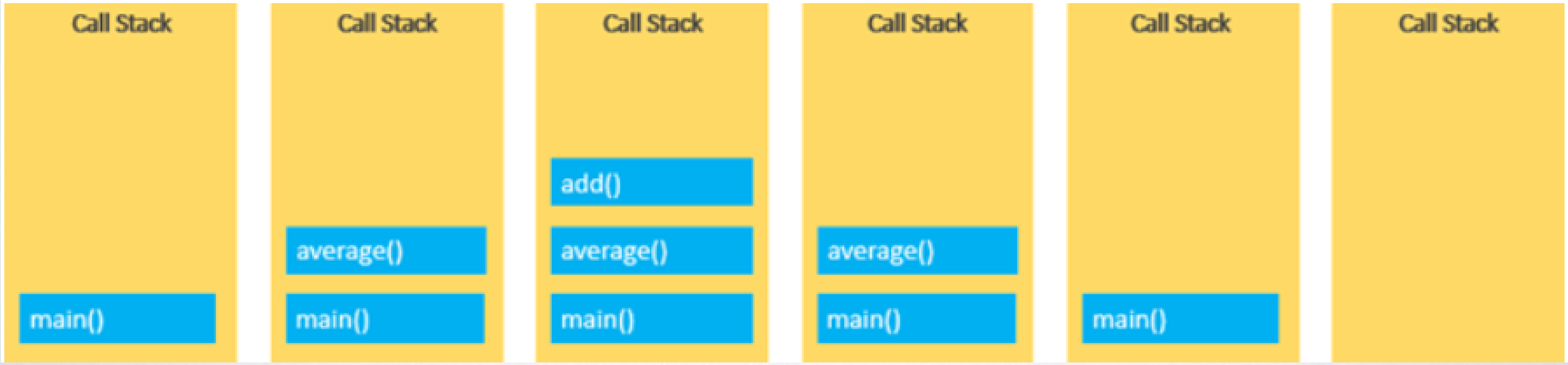


JavaScript Call Stack



```
function add(a, b) {
    return a + b;
}

function average(a, b) {
    return add(a, b) / 2;
}

let x = average(10, 20);
```

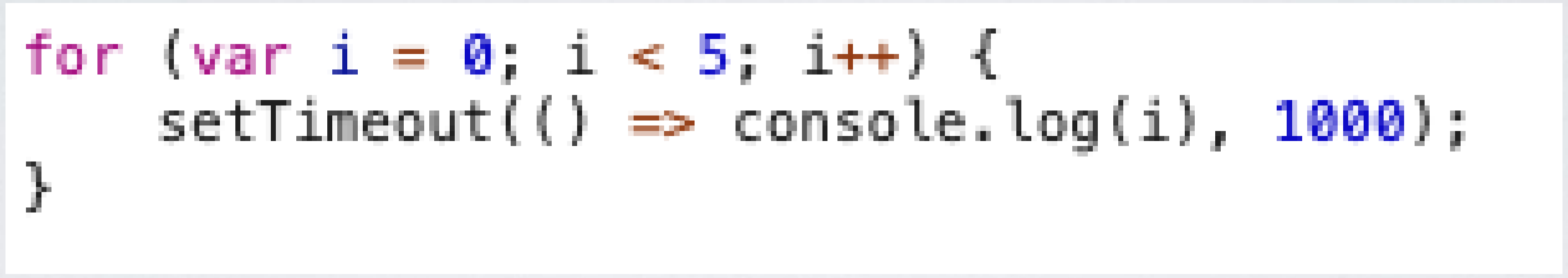
The following picture illustrates the overall status of the Call Stack in all steps:



ASYNCHRONOUS

- There are several ways to achieve asynchronous in JavaScript
  - Callback functions
  - Promise
  - async / await

ASYNCHRONOUS

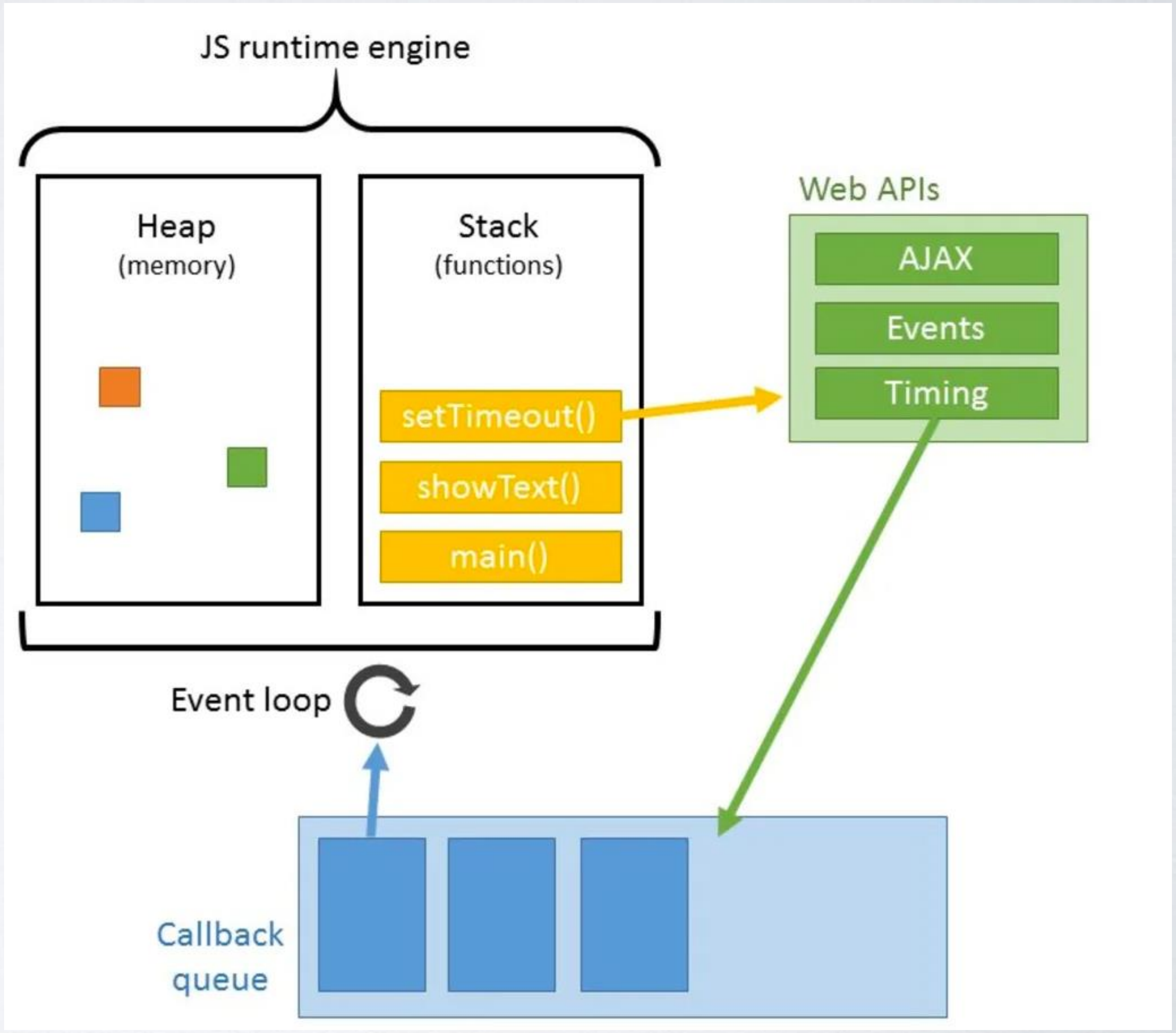


```
for (var i = 0; i < 5; i++) {
    setTimeout(() => console.log(i), 1000);
}
```

- JavaScript is a single threaded single concurrent language which means it can only handle one task or piece of code at a time
- How is it we can write asynchronous code with it, such as using setTimeout() in our example?
  - What is the output?
  - Why?

ASYNCHRONOUS

- JavaScript Runtime Engine
  - Heap
  - Stack
  - Web API Container
  - Callback Queue
  - Event Loop



ASYNCHRONOUS

- Heap
    - When JS runtime engine comes across variables and function *declarations* in the code it stores them in the *Heap*
- Stack
    - As the JS Engine comes across an actionable item, like a function call, it adds it to the *Stack*
    - Once a function is added to the Stack the JS engine jumps right in and starts parsing its code, adding variables to the Heap, adding new function calls to the top of the stack, or sending itself to the third container where Web API calls go
    - This process of parsing a function and popping it off the stack is what they mean when they say Javascript runs *synchronously*

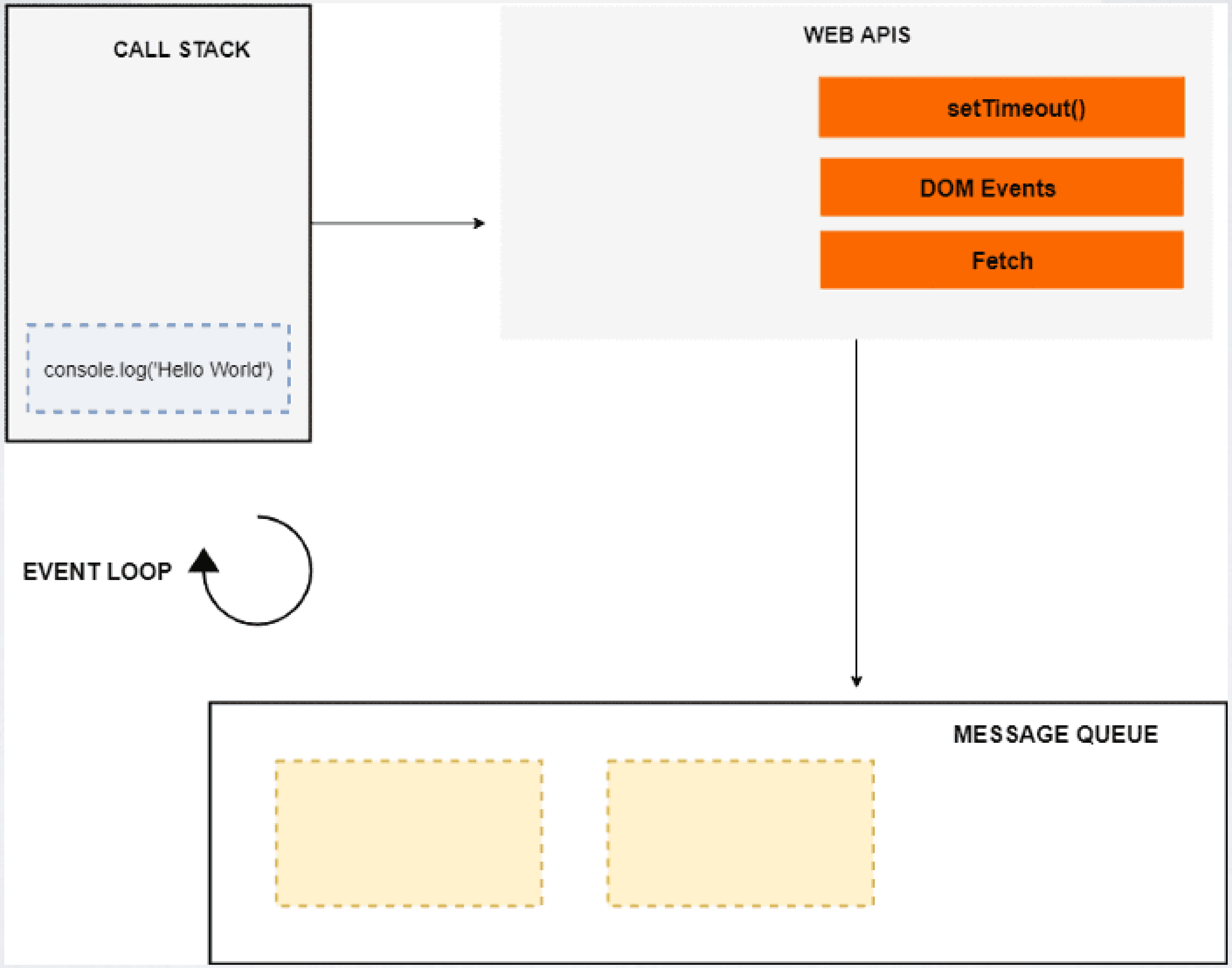
ASYNCHRONOUS

- Web API Container
  - The Web API calls that were sent to the Web API container from the Stack, like event listeners, HTTP/AJAX requests, or timing functions, sit there until an action is triggered
    - Either a ‘click’ happens, or the HTTP request finishes getting its data from its source, or a timer reaches its set time
  - In each instance, once an action is triggered, a ‘callback function’ is sent to the fourth and final container, the ‘callback queue.’

ASYNCHRONOUS

- Callback Queue
    - The *Callback Queue* will store all the callback functions in the order in which they were added.
    - It will ‘wait’ until the Stack is completely empty. When the Stack is empty it will send the callback function at the beginning of the queue to the Stack. When the Stack is clear again, it will send over its next callback function.
- Event Loop
    - Its job is to constantly look at the Stack and the Queue. If it sees the Stack is empty, it will notify the Queue to send over its next callback function.

```
const networkRequest = () => {
  setTimeout(() => {
    console.log('Async Code');
  }, 2000);
};


console.log('Hello World');


networkRequest();


console.log('The End');
```



https://blog.bitsrc.io/understanding-asynchronous-javascript-the-event-loop-74cd408419ff

ASYNCHRONOUS

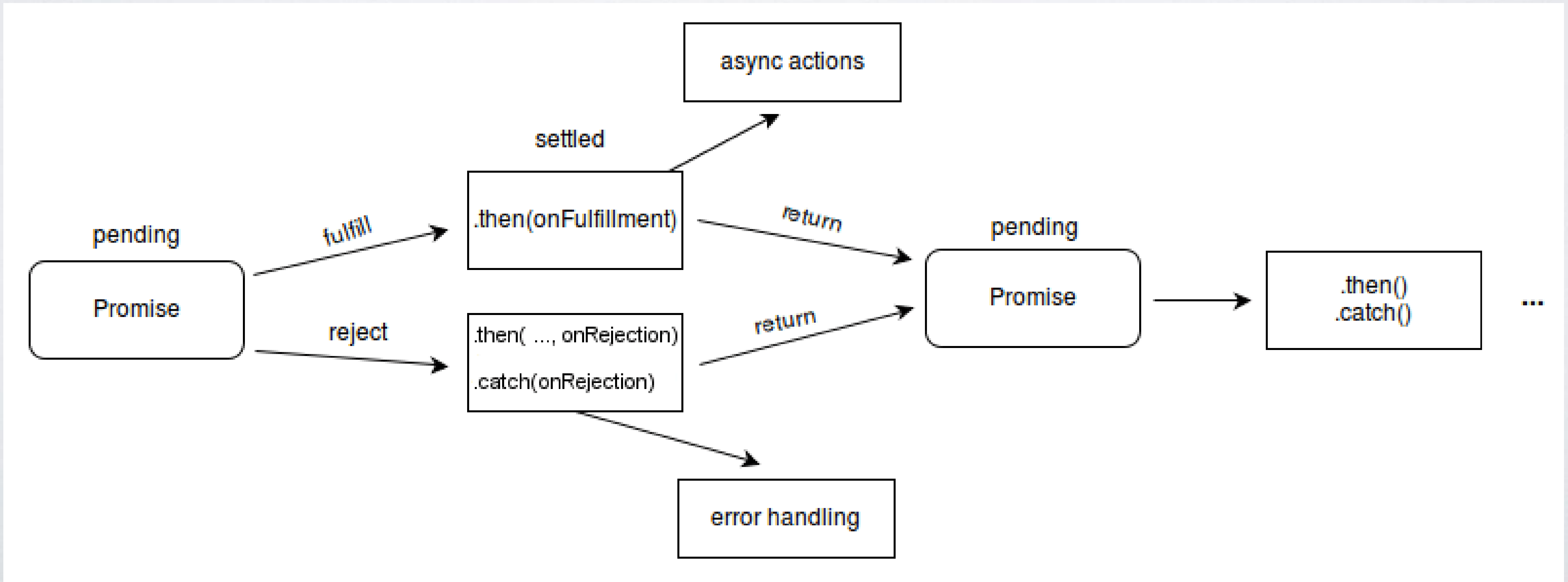
- Because of Web API Container, Callback Queue and Event Loop, JavaScript can run *Asynchronously*.
- It isn’t actually true, it just seems true. Javascript can only ever execute one function at a time, whatever is at top of the stack, it is a synchronous language
- But because the Web API container can forever add callbacks to the queue, and the queue can forever add those callbacks to the stack, we think of javascript as being asynchronous
- What is the output for following code and why?

```
setTimeout(function(){
  console.log('Hey, why am I last?');
}, 0);

function sayHi(){
  console.log('Hello');
}

function sayBye(){
  console.log('Goodbye');
}

sayHi();
sayBye();
```

ASYNCHRONOUS

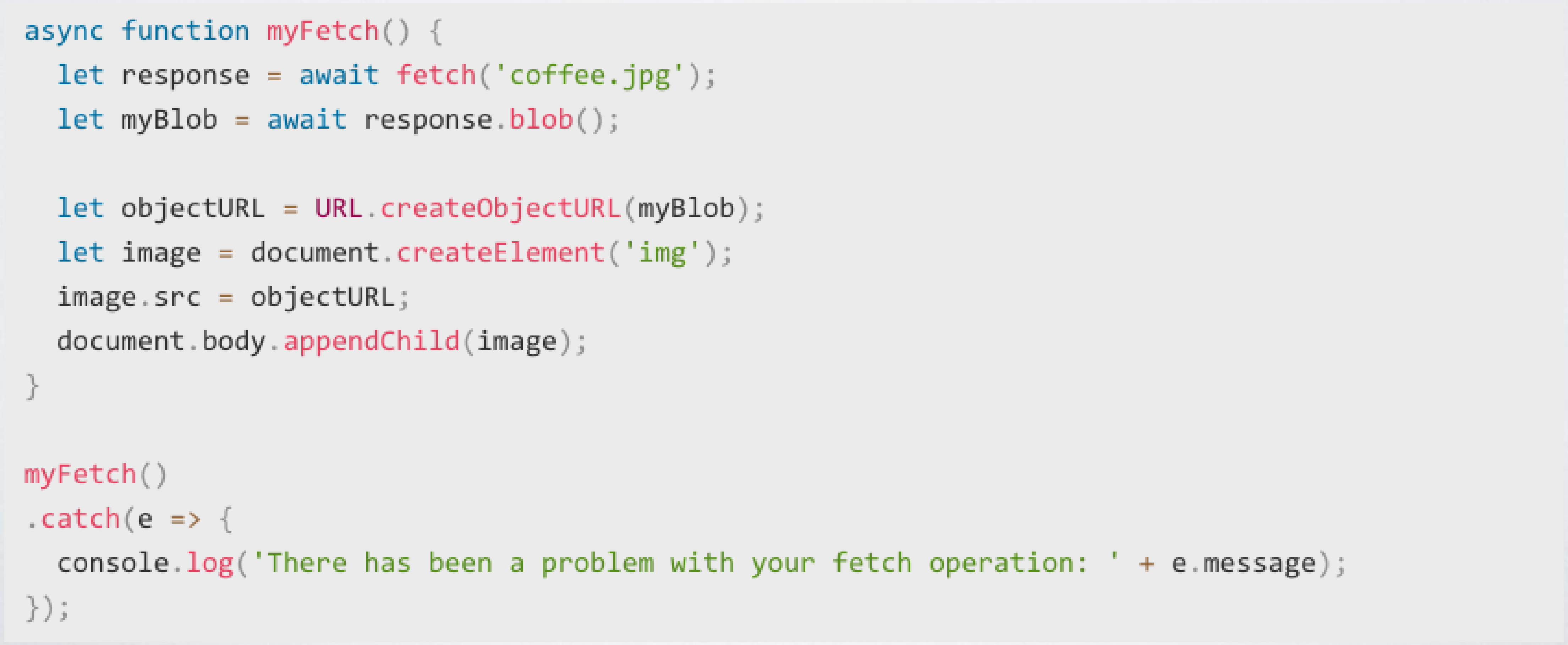
- Promise — A promise is an object that may produce a single value some time in the future: either a resolved value, or a reason that it's not resolved

- A promise may be in one of 3 possible states: fulfilled, rejected, or pending. Promise users can attach callbacks to handle the fulfilled value or the reason for rejection



ASYNCHRONOUS

- async — the async keyword is added to functions to tell them to return a promise rather than directly returning the value
- await — this can be put in front of any async promise-based function to pause your code on that line until the promise fulfills, then return the resulting value. In the meantime, other code that may be waiting for a chance to execute gets to do so



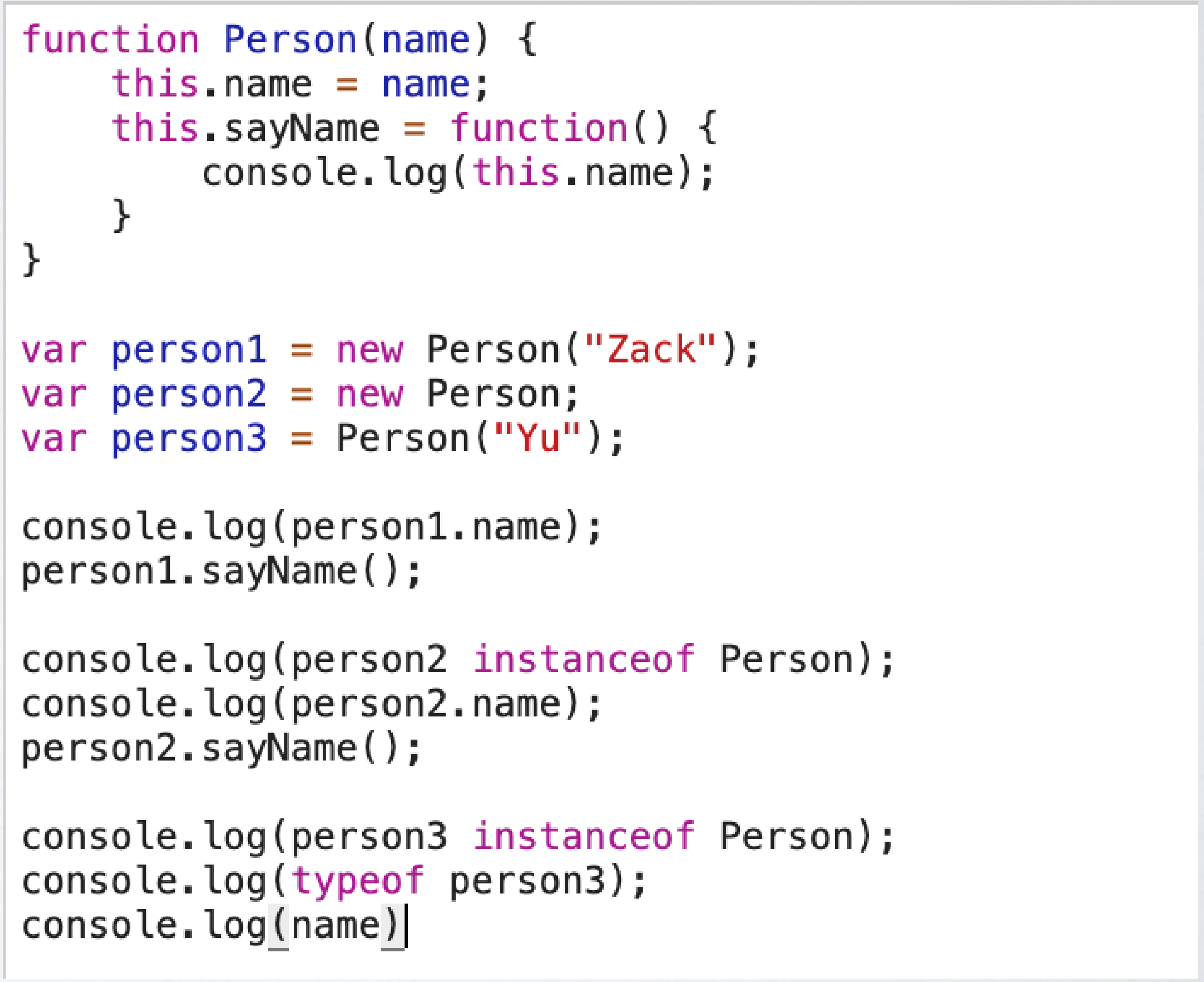
```
async function myFetch() {
  let response = await fetch('coffee.jpg');
  let myBlob = await response.blob();

  let objectURL = URL.createObjectURL(myBlob);
  let image = document.createElement('img');
  image.src = objectURL;
  document.body.appendChild(image);
}

myFetch()
.catch(e => {
  console.log('There has been a problem with your fetch operation: ' + e.message);
});
```

CONSTRUCTOR

- A *constructor* is simply a function that is used with *new* to create an object.

- Because a constructor is just a function, you define it in the same way.

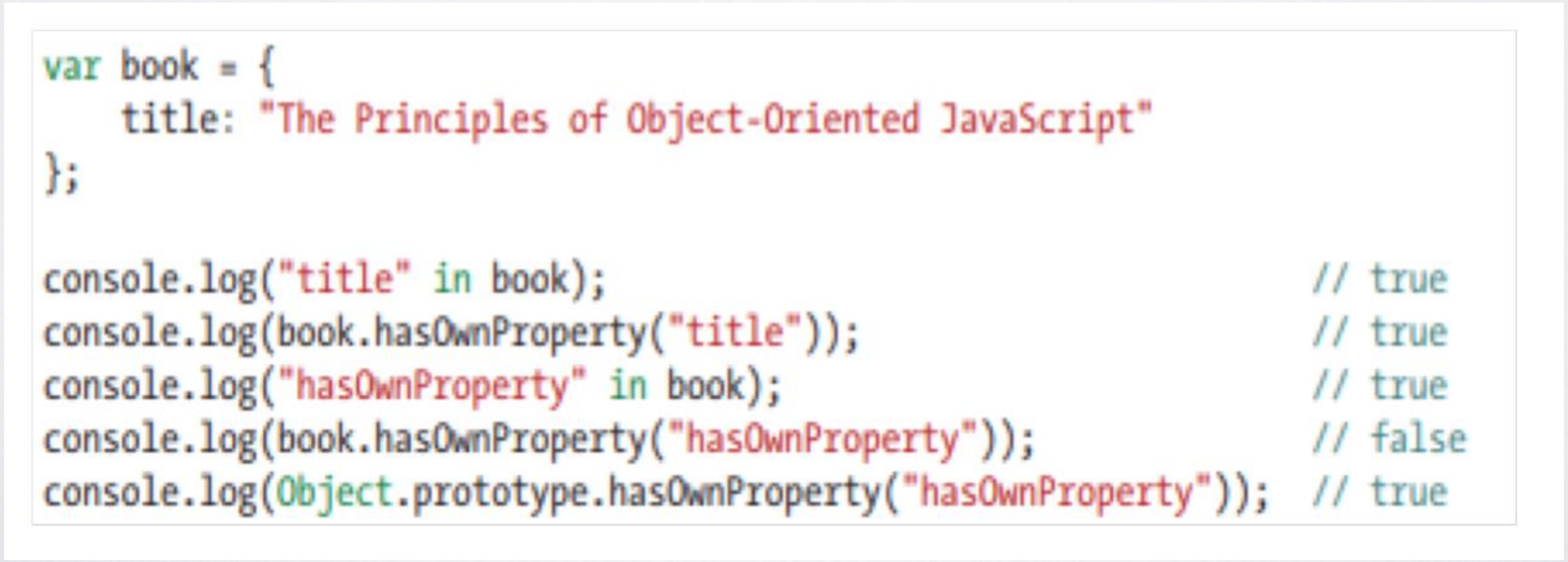
  - The difference is that constructor names should begin with a capital letter, to distinguish them from other functions.



```
function Person(name) {
    this.name = name;
    this.sayName = function() {
        console.log(this.name);
    }
}

var person1 = new Person("Zack");
var person2 = new Person;
var person3 = Person("Yu");

console.log(person1.name);
person1.sayName();

console.log(person2 instanceof Person);
console.log(person2.name);
person2.sayName();

console.log(person3 instanceof Person);
console.log(typeof person3);
console.log(name)
```

PROTOTYPE

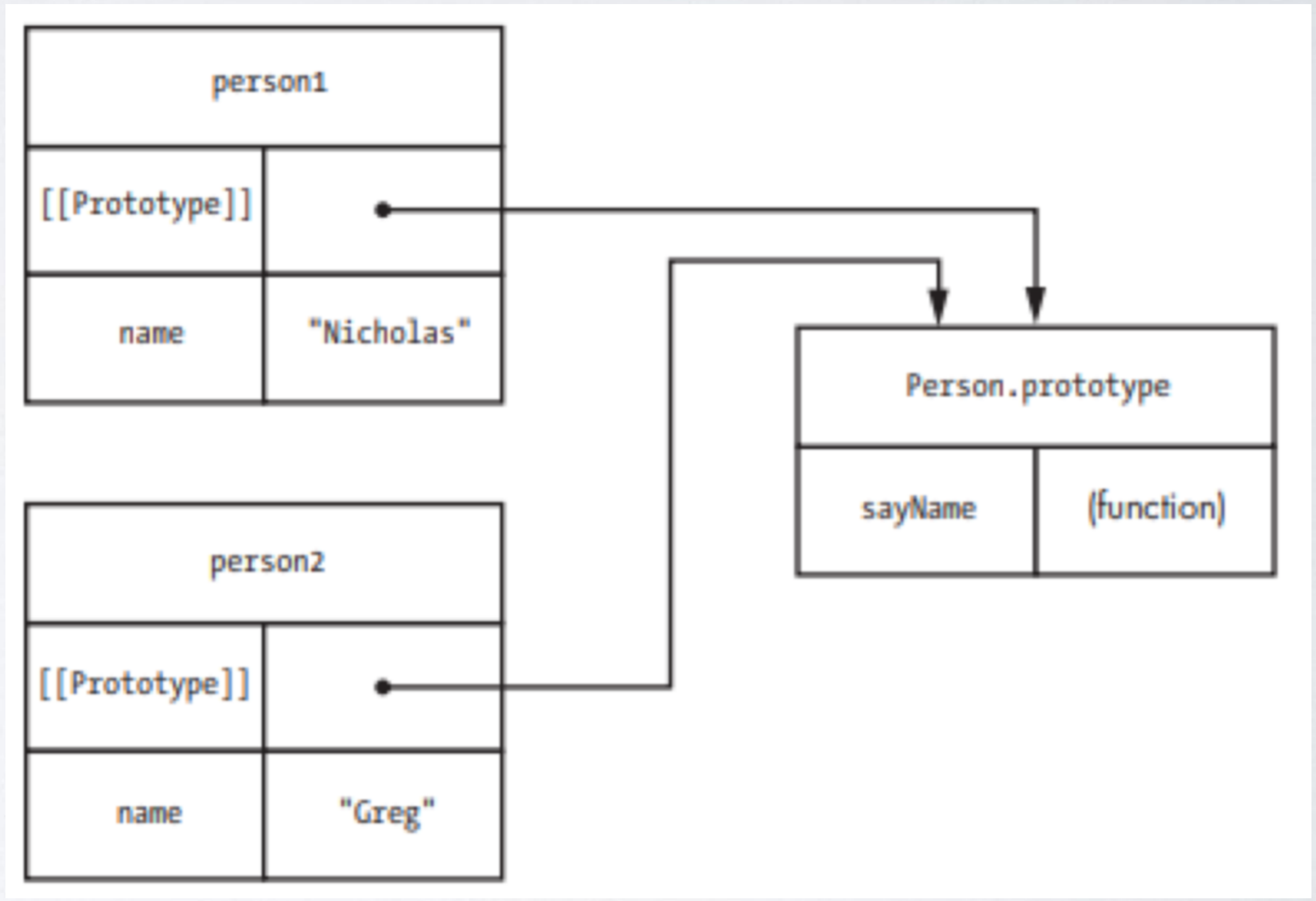
- JavaScript is often described as a *prototype-based* language — to provide *inheritance*, objects can have a *prototype* object, which acts as a template object that it inherits methods and properties from.



```
var book = {
    title: "The Principles of Object-Oriented JavaScript"
};

console.log("title" in book);                                   // true
console.log(book.hasOwnProperty("title"));                      // true
console.log("hasOwnProperty" in book);                          // true
console.log(book.hasOwnProperty("hasOwnProperty"));             // false
console.log(Object.prototype.hasOwnProperty("hasOwnProperty")); // true
```

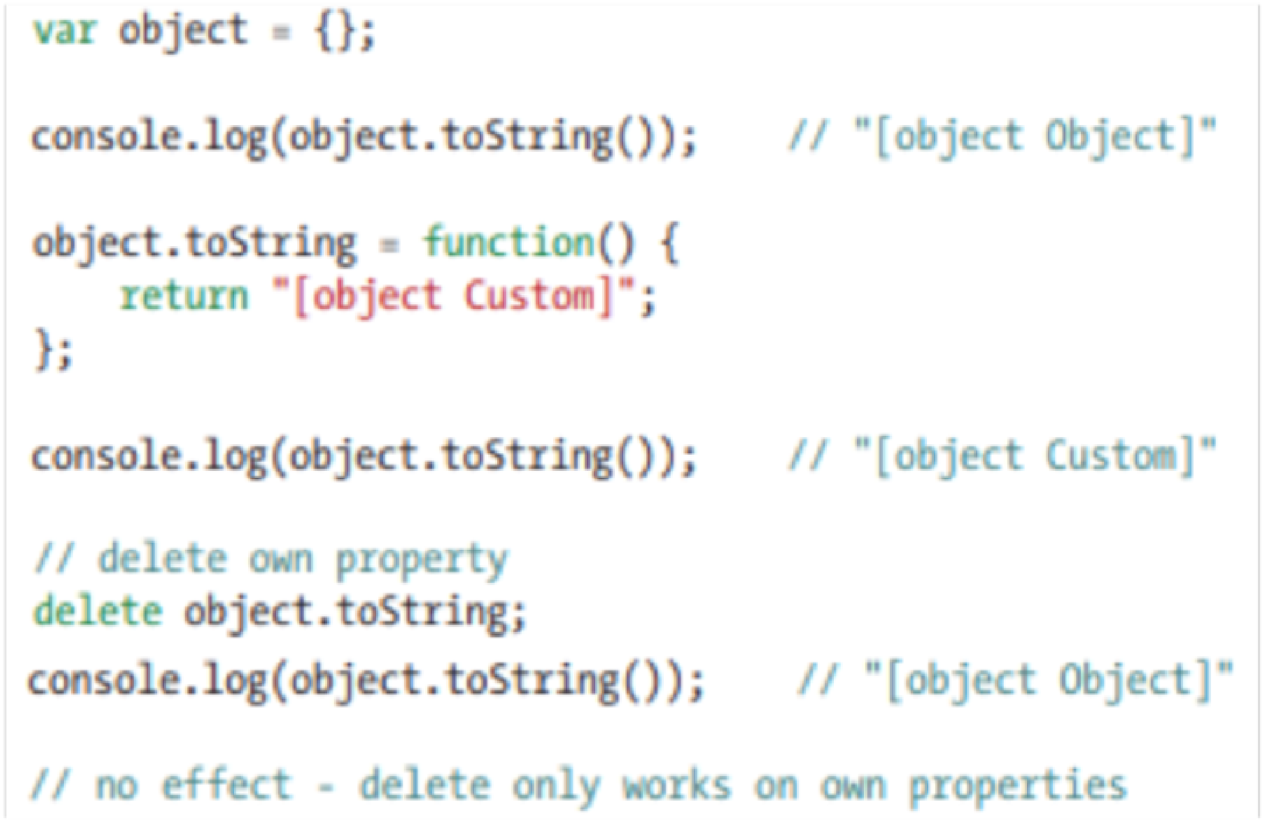
PROTOTYPE

- An instance keeps track of its prototype through an internal property called [[*Prototype*]]
- This property is a pointer back to the prototype object that the instance is using.
- When you create a new object using new, the constructor's prototype property is assigned to the [[Prototype]] property of that new object.

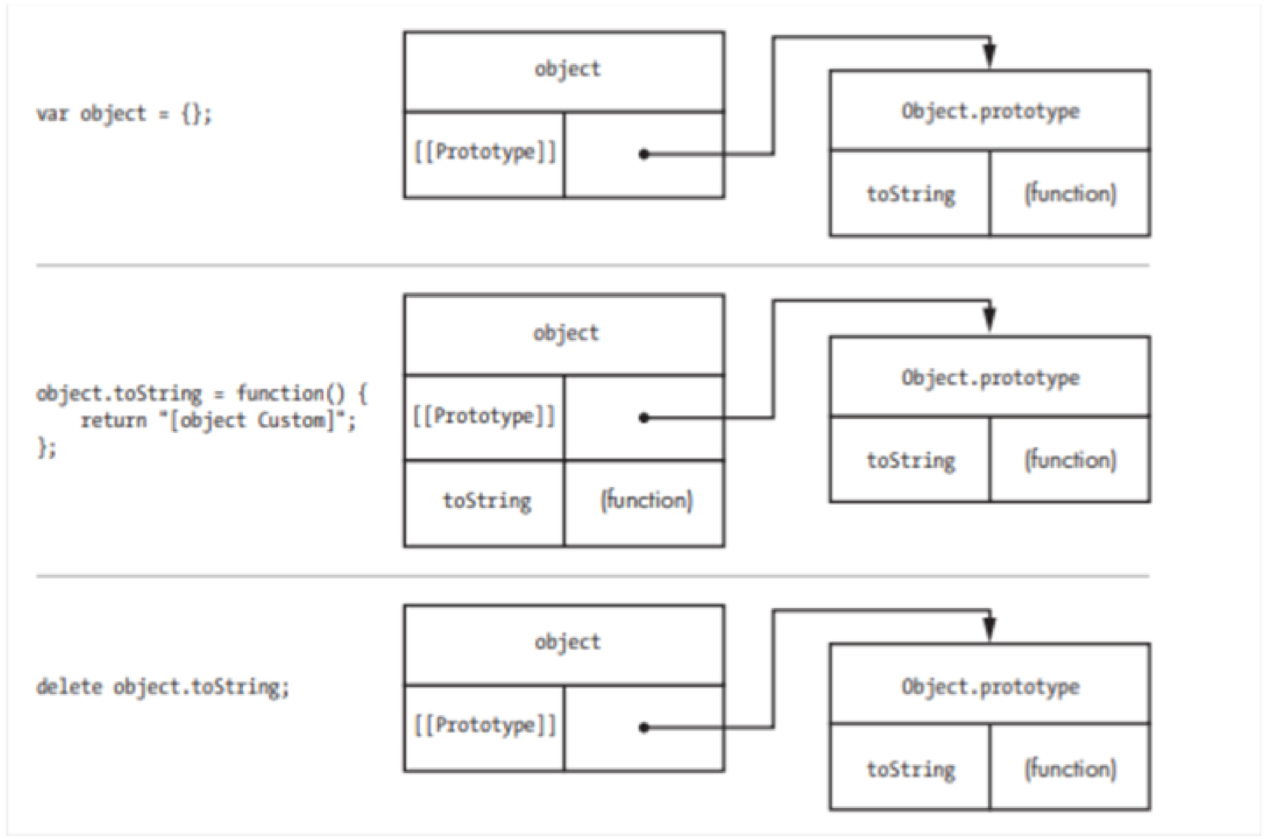


PROTOTYPE

- How *[[Prototype]]* Property read/write works



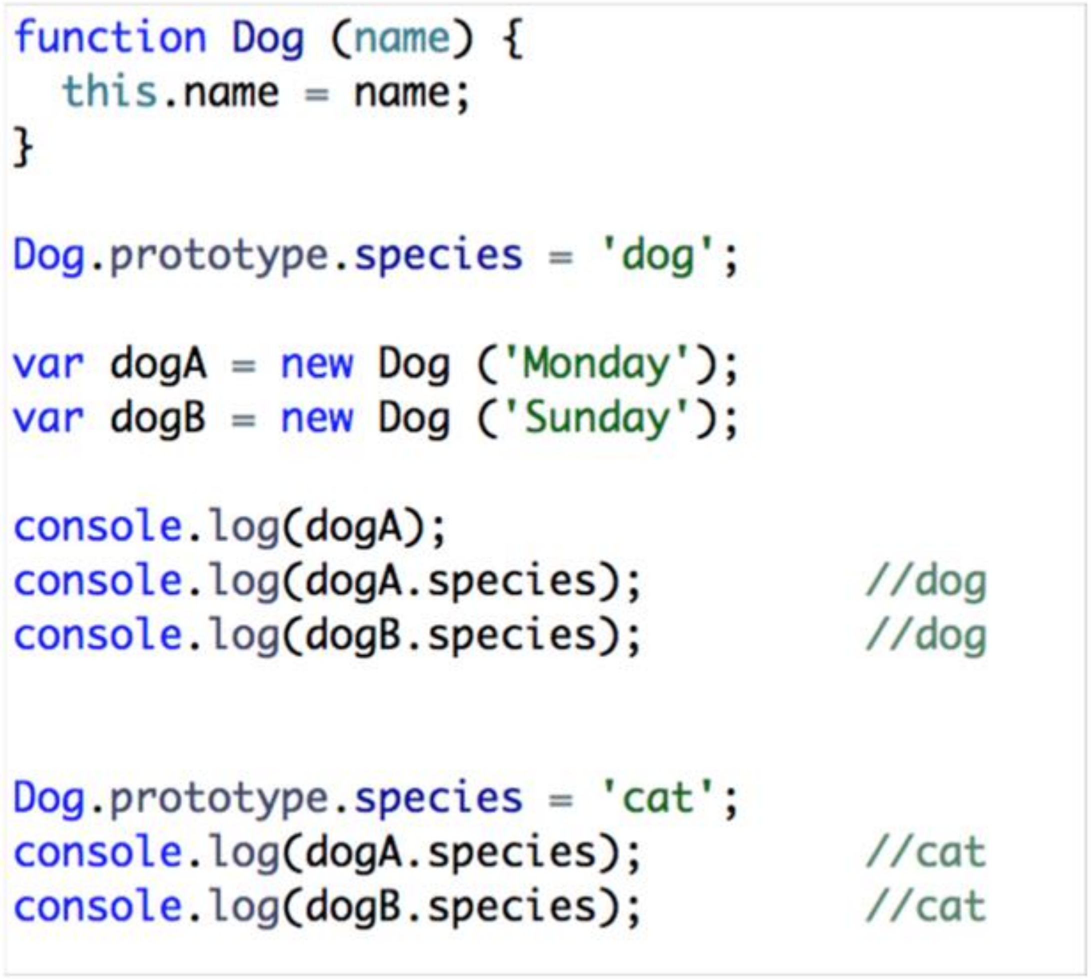
```
var object = {};

console.log(object.toString());    // "[object Object]"

object.toString = function() {
    return "[object Custom]";
};

console.log(object.toString());    // "[object Custom]"

// delete own property
delete object.toString;
console.log(object.toString());    // "[object Object]"

// no effect - delete only works on own properties
```



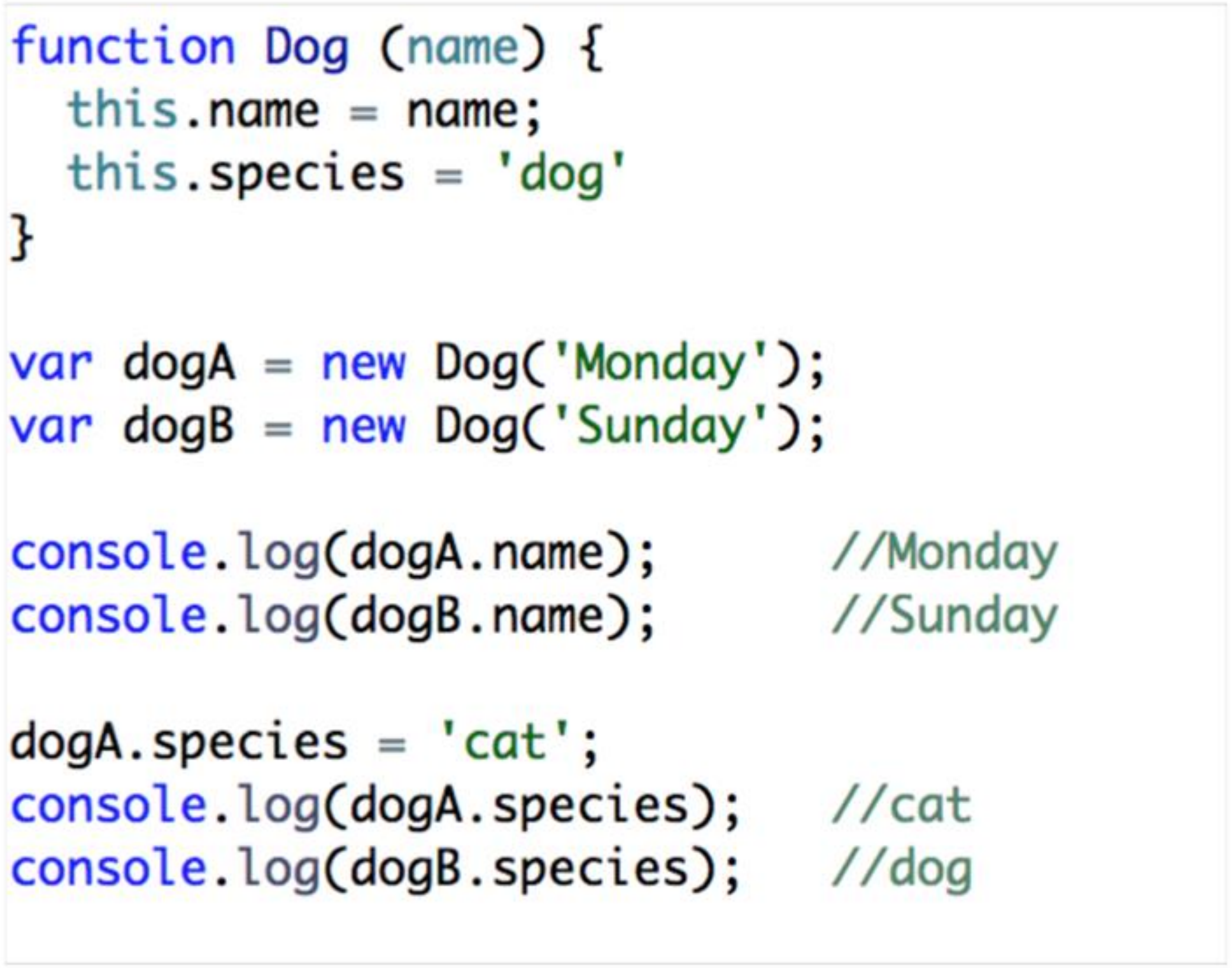
INHERITANCE

- When you attempt to access a property or method of an object
  - JavaScript will first search on the object itself, and if it is not found, it will search the object's [[Prototype]], until an object is reached with null as its prototype.
  - By definition, null has no prototype, and acts as the final link in this prototype chain.
- There are basically two ways to implement the inheritance in JavaScript
  - Prototype
  - Object

PROTOTYPE INHERITANCE

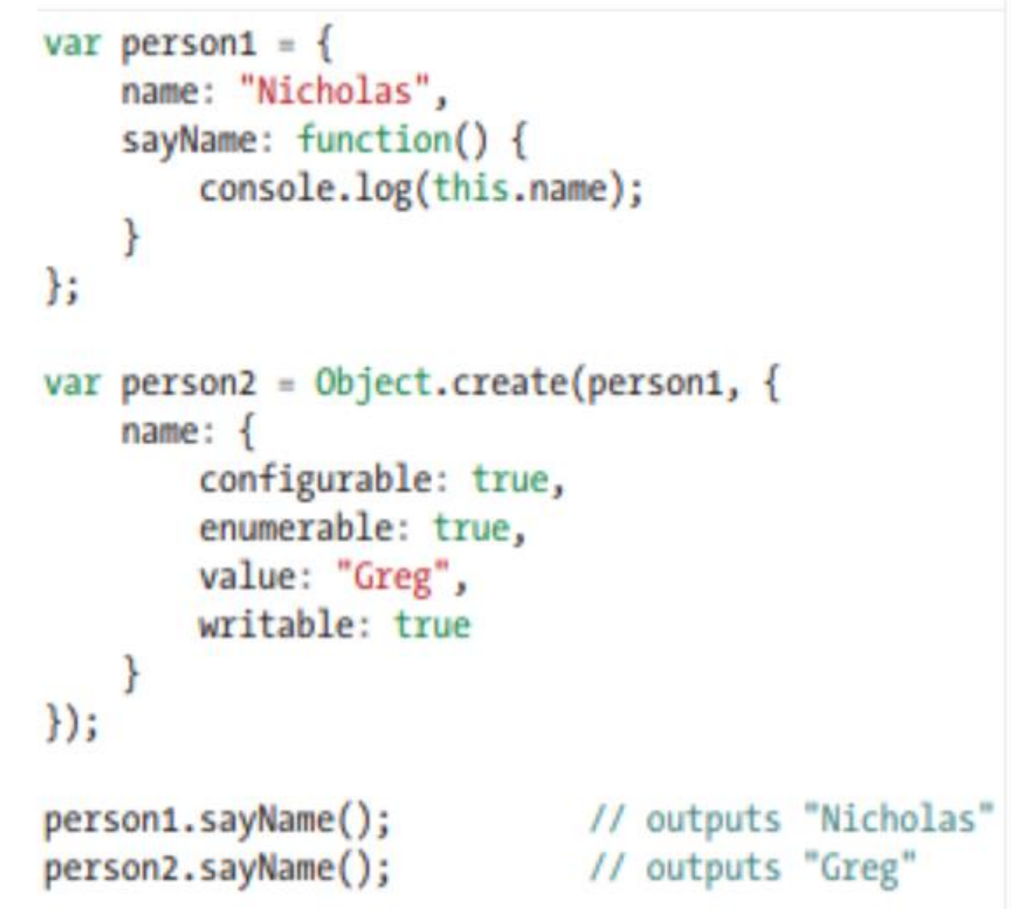


```
function Dog (name) {
  this.name = name;
}

Dog.prototype.species = 'dog';

var dogA = new Dog ('Monday');
var dogB = new Dog ('Sunday');

console.log(dogA);
console.log(dogA.species);          //dog
console.log(dogB.species);          //dog


Dog.prototype.species = 'cat';
console.log(dogA.species);          //cat
console.log(dogB.species);          //cat
```

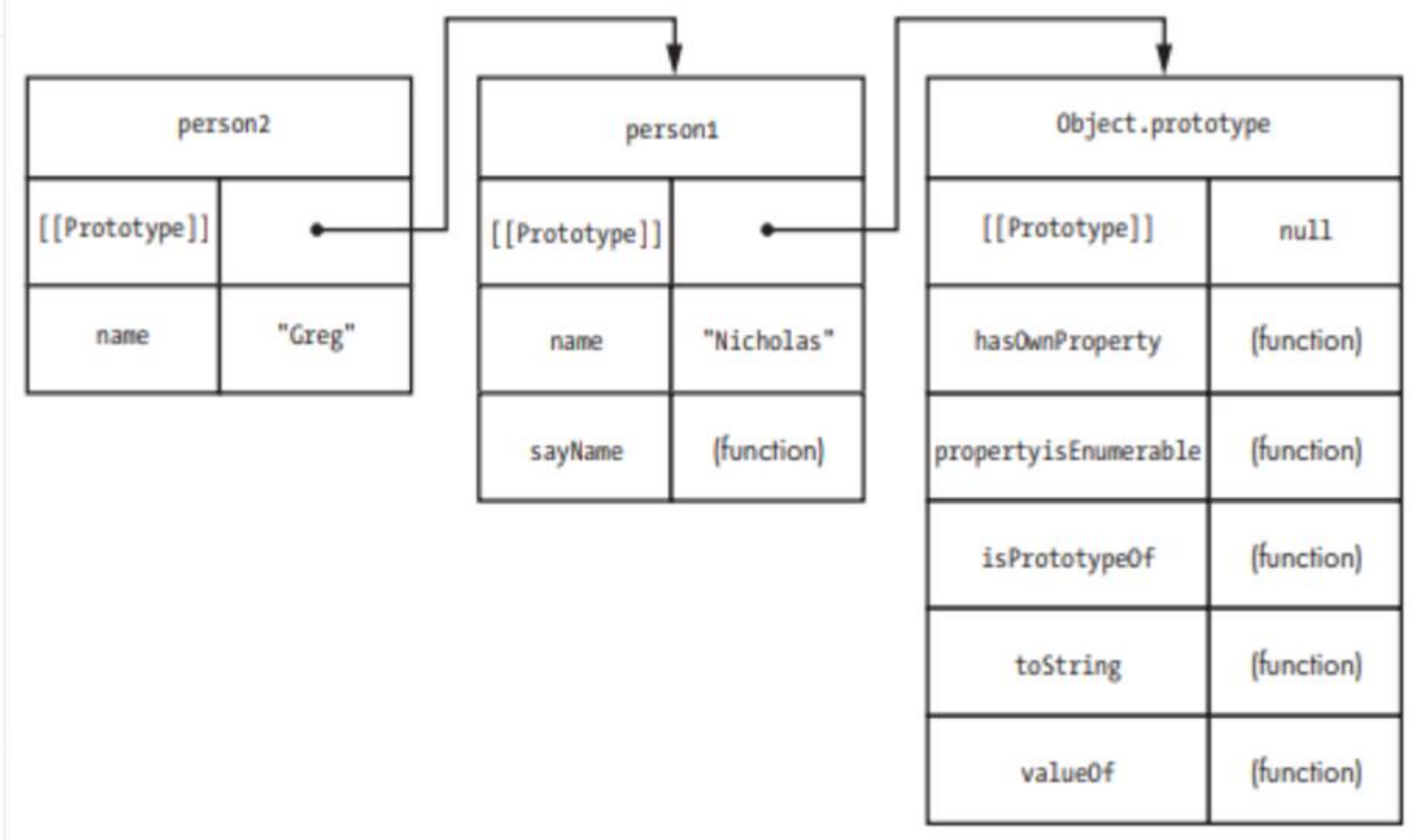


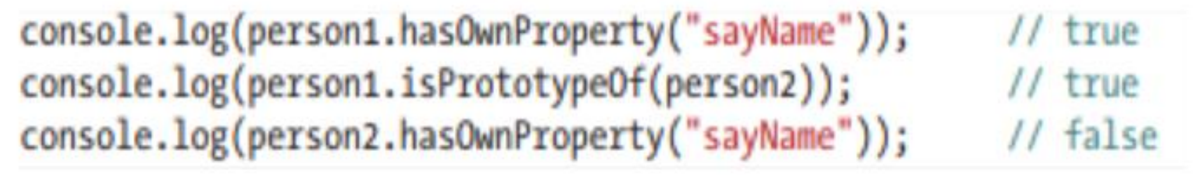
```
function Dog (name) {
  this.name = name;
  this.species = 'dog'
}

var dogA = new Dog('Monday');
var dogB = new Dog('Sunday');

console.log(dogA.name);      //Monday
console.log(dogB.name);      //Sunday

dogA.species = 'cat';
console.log(dogA.species);   //cat
console.log(dogB.species);   //dog
```

OBJECT INHERITANCE



```
var person1 = {
    name: "Nicholas",
    sayName: function() {
        console.log(this.name);
    }
};

var person2 = Object.create(person1, {
    name: {
        configurable: true,
        enumerable: true,
        value: "Greg",
        writable: true
    }
});

person1.sayName();          // outputs "Nicholas"
person2.sayName();          // outputs "Greg"
```





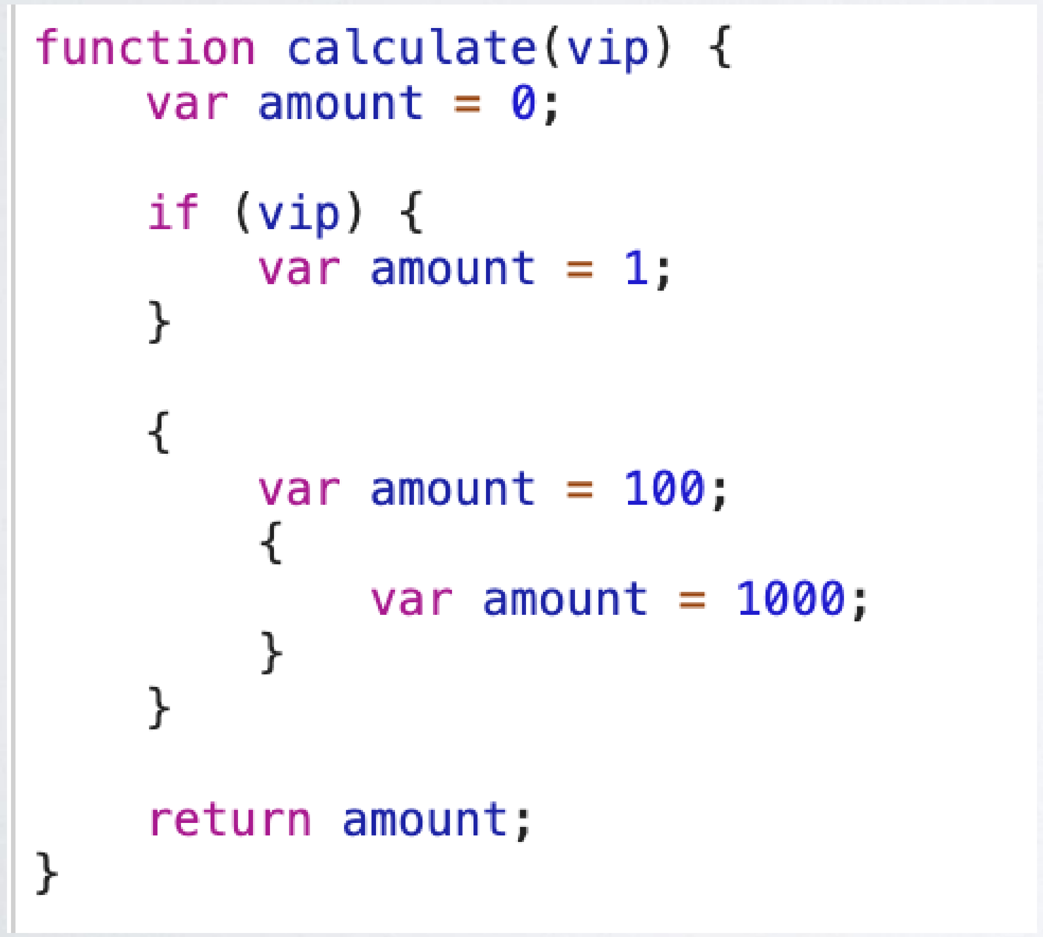
```
console.log(person1.hasOwnProperty("sayName"));     // true
console.log(person1.isPrototypeOf(person2));        // true
console.log(person2.hasOwnProperty("sayName"));     // false
```

ES6 FEATURES

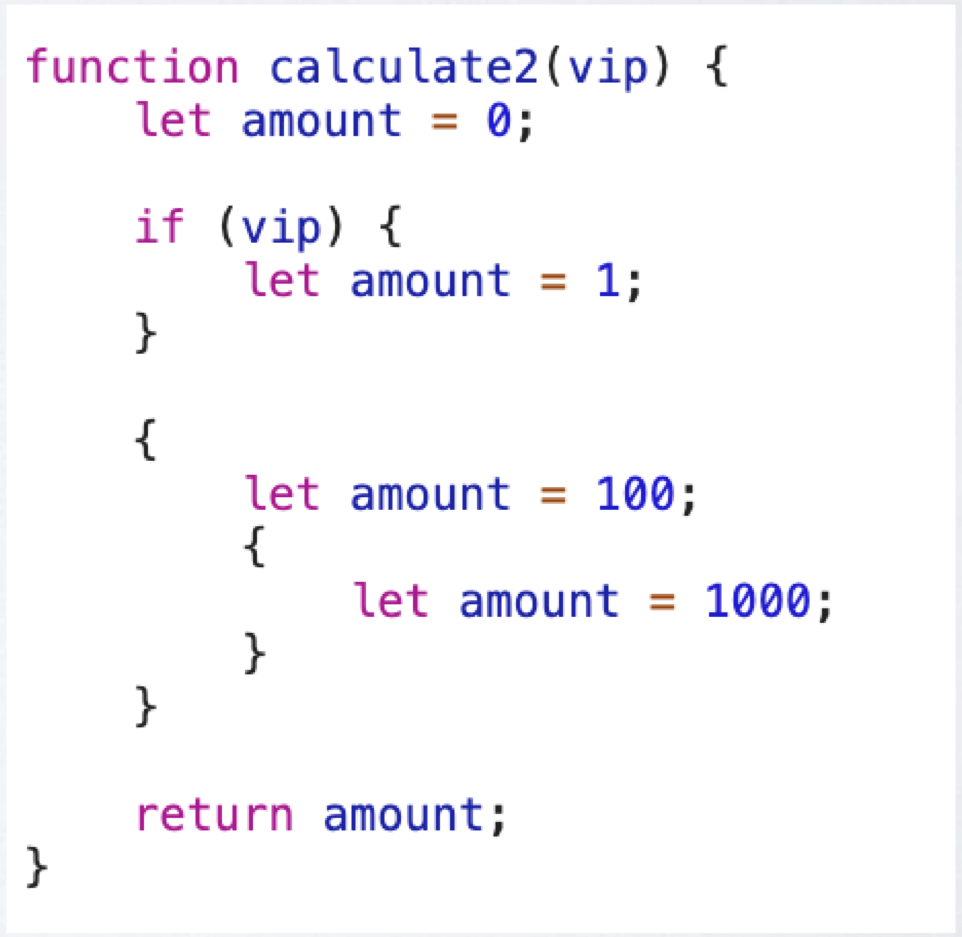
- There are some new features added in ES6
  - JavaScript let
  - JavaScript const
  - JavaScript Arrow Functions
  - JavaScript Classes
  - Default parameter values
  - Array.find()
  - Array.findIndex()
  - Exponentiation (**)

LET & CONST

- These two keywords provide Block Scope variables (and constants) in JavaScript.



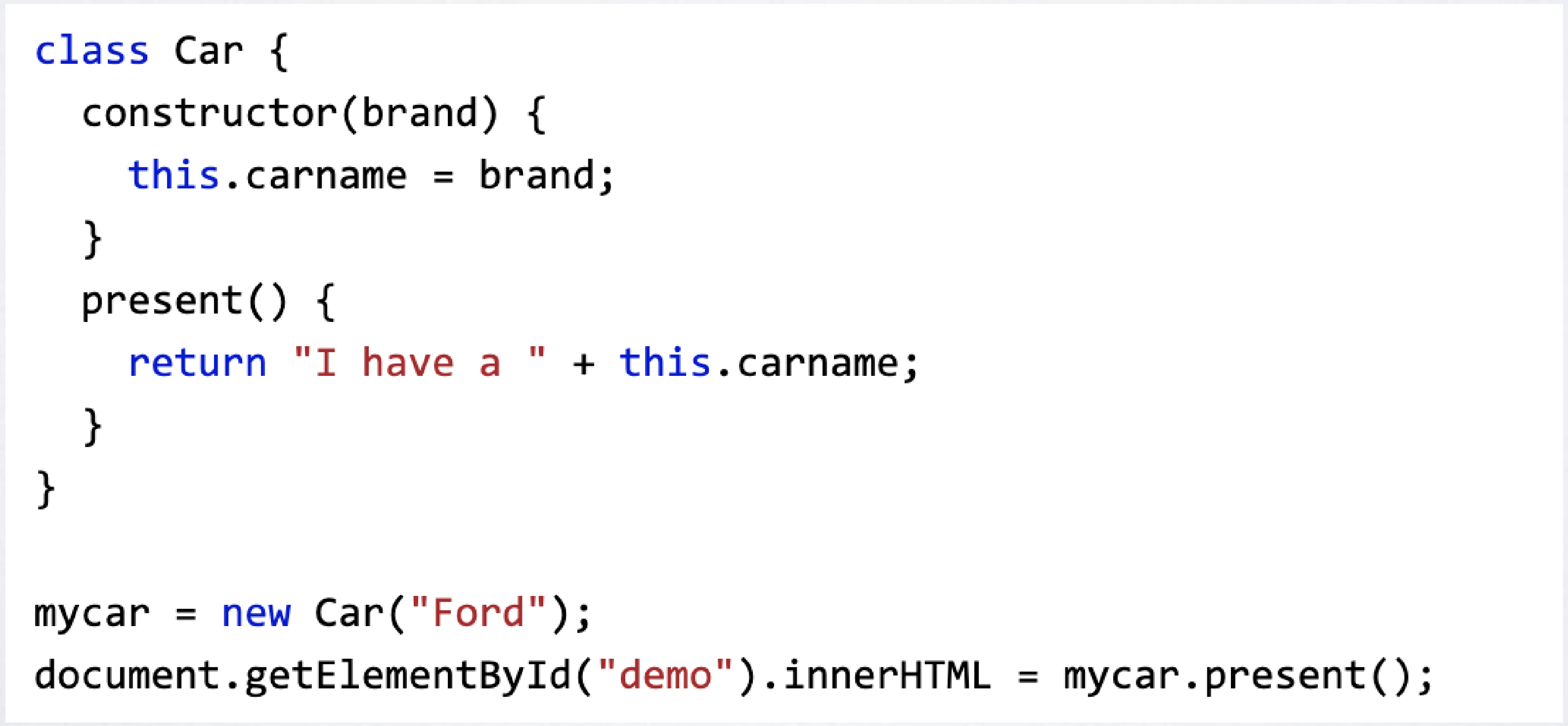
```
function calculate(vip) {
    var amount = 0;

    if (vip) {
        var amount = 1;
    }

    {
        var amount = 100;
        {
            var amount = 1000;
        }
    }

    return amount;
}
```



```
function calculate2(vip) {
    let amount = 0;

    if (vip) {
        let amount = 1;
    }

    {
        let amount = 100;
        {
            let amount = 1000;
        }
    }

    return amount;
}
```

CLASS

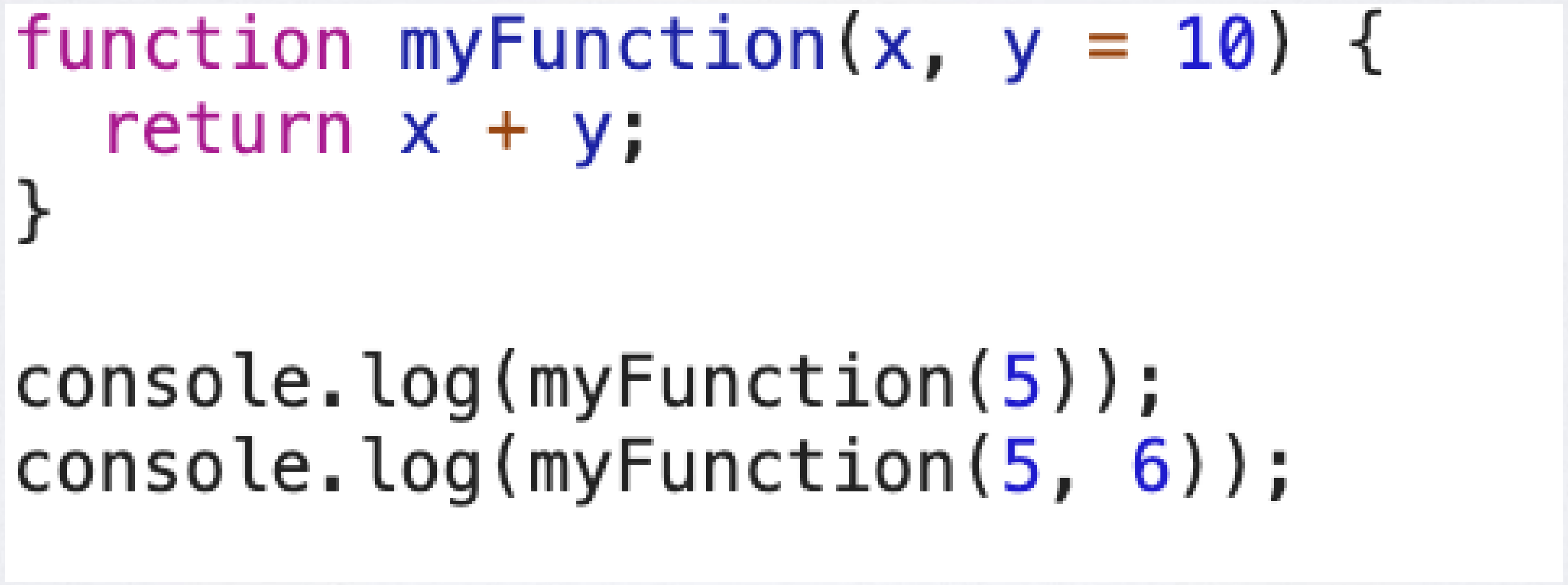
- A class is a type of function, but instead of using the keyword function to initiate it, we use the keyword class, and the properties are assigned inside a constructor() method.



```
class Car {
  constructor(brand) {
    this.carname = brand;
  }
  present() {
    return "I have a " + this.carname;
  }
}

mycar = new Car("Ford");
document.getElementById("demo").innerHTML = mycar.present();
```

DEFAULT PARAMETER

- ES6 allows function parameters to have default values.



```
function myFunction(x, y = 10) {
  return x + y;
}

console.log(myFunction(5));
console.log(myFunction(5, 6));
```

ANY QUESTIONS?