# Glossary

# Introduction

JavaScript is a

* Lightweight, cross platform, object-oriented
* Dynamic – compile just in time – while the browser is loading the JS and just before executing it (no compile 1st and then execute),
* weakly typed,
* prototype-based language with
* first-class functions.
* **Javascript** is always pass by value, but when a variable refers to an **object** (including arrays), the "value" is a **reference** to the **object**. Changing the value of a variable never changes the underlying primitive or **object**, it just points the variable to a new primitive or **object**.

JavaScript == ECMAScript == Jscript

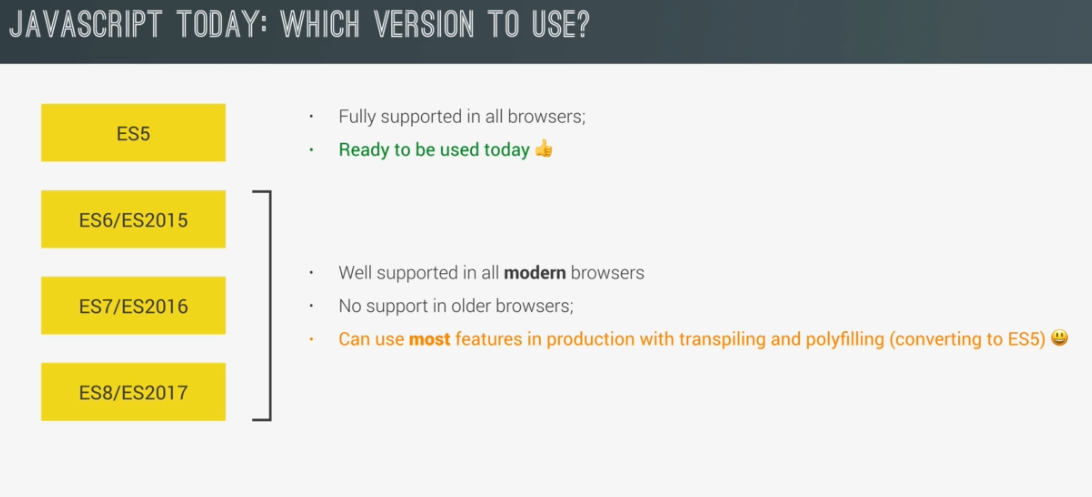
JavaScript != Document Object Model (DOM)

ECMAScript is a standardized version of JavaScript with the goal of unifying the language's specifications and features. As all major browsers and JavaScript-runtimes follow this specification, the term *ECMAScript* is interchangeable with the term *JavaScript*.

The most recent standardized version is called ECMAScript 6 (ES6), released in 2015.

## JavaScript Versions

* ECMAScript – the language standard
* JavaScript – the language name
* ES5 (ECMAScript 5) – is the main ‘old’ release
* ES6 (ES2015 : ECMAScript 2015) – is the main ‘new’ release with the biggest update to the language.
* 2015: Changed to smaller, annual release cycle.
* Future versions of JavaScript are called together **ESNext**



To check which feature is supported in which browser: <http://kangax.github.io/compat-table>

## Additional resources

<https://github.com/jonasschmedtmann/complete-javascript-course>

<http://codingheroes.io/resources/>

# Style Guide

The [Mozilla Developer Network (MDN)](https://developer.mozilla.org/en-US/) is a fantastic resource for all things web and [JavaScript](https://developer.mozilla.org/en-US/docs/Web/JavaScript)

Style Guide: <https://google.github.io/styleguide/javascriptguide.xml>

* Variable, function names: camelCase
* Files/modules names: lower\_case\_with\_underscore.js
* const IN\_ALL\_CAPITALS
* “use strict”; - as the 1st line in every file!!
* Use let or const. Never var!
* Always use ‘;’ at the end of java script command.
* Prefer // to /\*..\*/ for comments!

## Linter

### JSLint

<http://jslint.com/help.html>

The most common JS linter is jslint.

To install:

npm install -g jslint

#### Flags

To control the jslint behavior, add the following at the start of your files:

* Allow console outputs (console.log):

/\*jslint devel: true \*/

**NOTE:** Make sure to remove this from production code!!

* Allow whitespcaes at the end of the line:

/\*jslint white:true \*/

### ESLint

Allow better customization of all the rules used by the linter. This is more flexible than JSLint.  
ESLint recommend installing it locally, at each project.

##### Install:

npm install eslint –save-dev

##### Configure

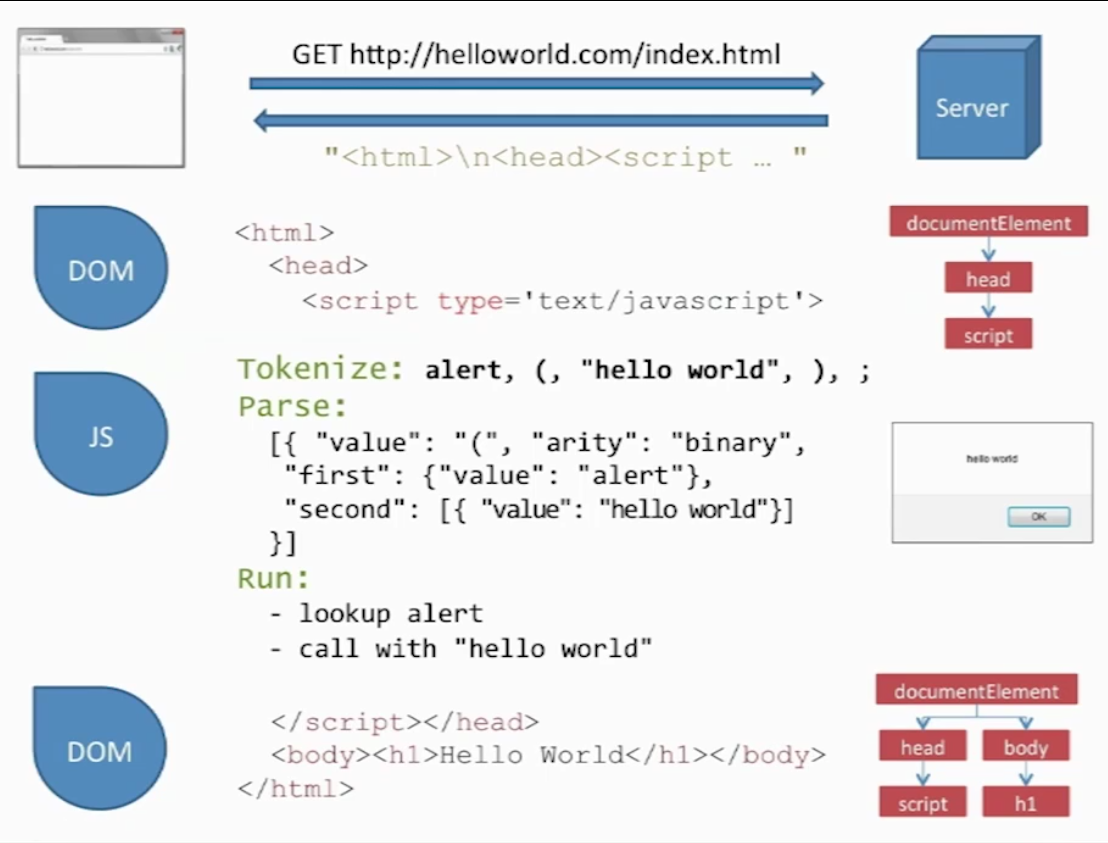
1. create a configuration file:   
   eslint --init

# Running In the DOM

When we have the following HTML:



What actually happens when we do to its site:



1. The browser sends an http request to the server
2. The server returns the html as a string
3. The browser pass it off to its web engine to process the html and build the DOM objects from it (documentElement etc)
4. When it see the java script, it will pass it to the JS engine (e.g. V8).  
   NOTE: **The browser hangs the processing of the page while it waits for the JS engine to process the script!**
5. The JS engine will tokenize and parse the script (i.e. ‘compile’ it to byte code) and execute it.
6. After the JS Engine has finished, it will return control to the DOM to keep parsing the html.
7. The DomContentReady will run only after the DOM finished processing the whole body of the html (after the </body>

## DOM Manipulation

* document – the DOM as an object we can query
* document.querySelector() – allow us to get access to an element in the DOM. Uses the same selectors as in CSS. Examples:

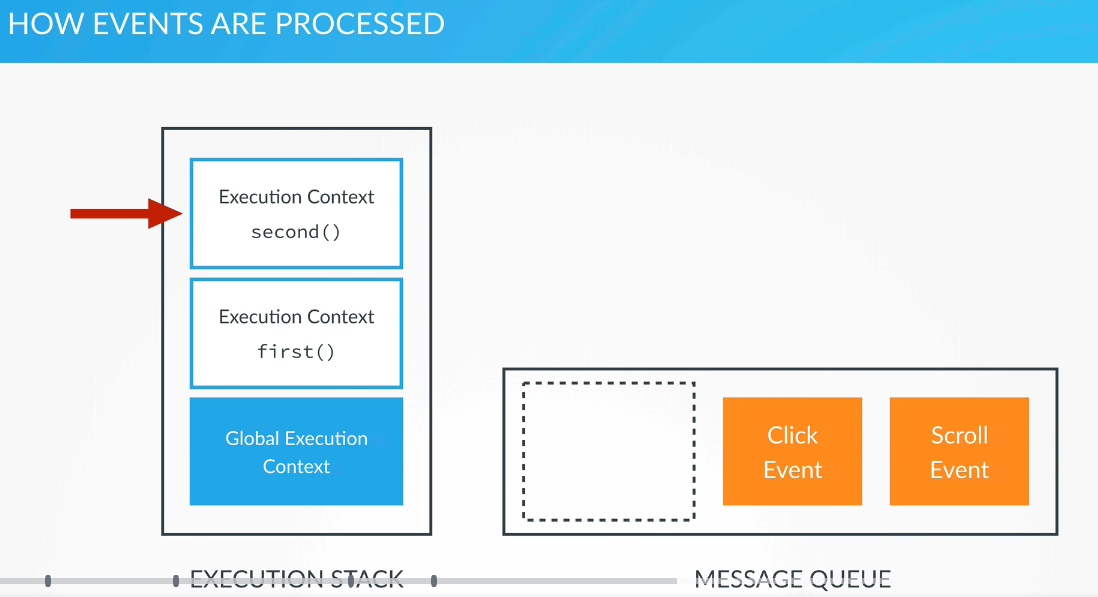
document.querySelector(  
'#current-' + activePlayer).innerHTML = `<em> ${dice} </em>`;

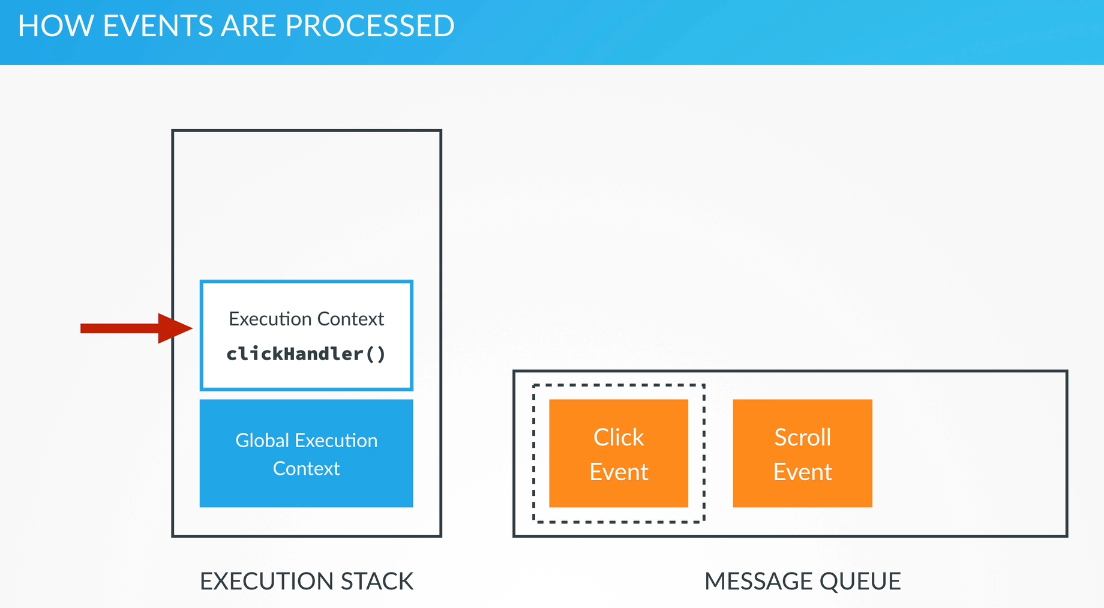
var x = document.querySelector(`#score-${activePlayer}`).textContent;

* getEelementById – a faster way to get elements by their Ids.
* Elements attributes can usually be accessed directly. For example:  
  document.querySelector('.dice').src = `dice-${dice}.png`;  
  document.getElementById(‘score-0’).style.display = ‘none’;
* Add/remove/toggle class to an element – use the classList property of the element:  
  document.querySelector(`.player-1-panel`)**.classList.toggle**('active');

## Event Handlers

* An event can **only** be handled when the **execution stack is empty**. Meaning there are no running functions!
* All the events in the page are entered into a **queue** and there they await execution in a FIFO order.





To add an event handle to elements:

let element = document.getElementById(“elementId”);  
element**.addEventListener**(“<event>”, eventHandlerFunction);

## MutationObserver

If you need to monitor the DOM for changes and register a callback function to handle them, you can use MutationObserver: <https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver>.

See some [rules of thumb](https://stackoverflow.com/questions/31659567/performance-of-mutationobserver-to-detect-nodes-in-entire-dom/39332340) for working with MutationObserver.

## Optimizations

* You should put your **JS at the bottom of the html** and not at the start of it so your users will see something while the JS is processing. Otherwise, the loading of the page will hang while the JS process the script.
* To make sure your code executes only after the page has finished loading, you can attached a JavaScript ‘DOMContentLoaded’ event to the document:

document.addEventListener('DOMContentLoaded',  
 function() {  
 ….   
// You can register additional event handlers here. For example: document.getElementById(‘getMessage’).onclick=function() {…};  
});

# Require & Import

## Require

Will import **all the functions and code** from the required module

In strict mode (“use strict”):

1. In the exporting module add the following at the buttom:

module.exports = {

<exported name>: <function/class/variable name>,  
 …

};

For example:

module.exports = {

palindrome: palindrome,

};

1. In the importing module add the following at the top:

let <local name> = require<path for module>.<exported name>;

for example:

let palindrome = require("../palindrome\_checker").palindrome;

Note: the require clause can also contain only the module’s name in order to include all the exported parameters.

## Import & Export – ES6 and up

### import

Allow us to chose which parts of a module/file to load into a given file:

**import** { <funct/var name> } **from** "<file\_path>";

to import everything from a file:

**import \* as** <imported object name> **from** “<file path>”;

**Notes:**

* The Import and Export are both non-browser features!!
* The whitespace surrounding the function inside the curly braces is a best practice - it makes it easier to read the import statement.
* In most cases, the file path requires a ./ before it; otherwise, node will look in the node\_modules directory first trying to load it as a dependency.

### Export

For each file, we need to declare which functions and variable can be imported from the outside, using the export statements:

const capitalizeString = (string) => {  
  return string.charAt(0).toUpperCase() + string.slice(1);  
}  
const foo = "bar";  
**export** { capitalizeString, foo }

We can also write a separate export statement for every function/variable separately:

export { capitalizeString } //How to export functions.  
export const foo = "bar"; //How to export variables.

#### Export Default

export default function add(x,y) {  
  return x + y;  
}

**Note:** Since export default is used to declare a fallback value for a module or file, you can only have **one value be a default export in each module or file**. Additionally, you cannot use export default with var, let, or const

To import an export default, you need to use this special syntax (without the curly brackets):

import add from “math\_functions”

# Data Types

undefined, null, boolean, string, symbol, number, and object.

We don’t declare the type of the variable explicitely.

## Primitive date types

The Primitive date types (that are not objects) in js are **value types**. They are:

* Number: floating point numbers, for decimals and integers
* String
* Boolean (see below)
* Undefined: data type of a variable that does not have any value yet
* Null:

## Type Coercion and Variable Mutation

* Type coercion – javascript ability to automatically convert the type of variables to the required type (for example: console.log(5);)
* Variable mutation – javascript ability to change the type of the variable dynamically according to its current value:  
  var varMutation = “now I’m a string”;  
  varMutation = 5;

## Boolean

**Truthy** values in JS are values that are not defined as Boolean but will evaluate to true when evaluated in a Boolean context.

Here are the **falsy** values:

* false
* null
* undefined
* The empty string ''
* The number 0
* The number NaN

All other values are truthy, including true, the string 'false', and all objects.

### Equality

* == (double ‘=’) – will compare that values and if they are of different types (for example, numbers and strings), it **will automatically convert one type to another**. This is known as "Type Coercion".
* **Strict equality** (**===**) is the counterpart to the equality operator (==). However, unlike the equality operator, which attempts to convert both values being compared to a common type, the strict equality operator does not perform a type conversion. Strict inequality (!==) is the opposite of ===.

## Number

## JavaScript has a single number type. Internally, it is represented as 64-bit floating point, the same as Java’s double. Unlike most other programming languages, there is no separate integer type, so 1 and 1.0 are the same value. This is a significant convenience because problems of overflow in short integers are completely avoided, and all you need to know about a number is that it is a number. A large class of numeric type errors is avoided.

The value NaN is a number value that is the result of an operation that cannot produce a normal result. NaN is not equal to any value, including itself. You can detect NaN with the isNaN( number ) function.

The floating point can cause precision errors unlike when working with decimals or integers in other languages. The ways to avoid this is:

* Write a round function that translate the number to whole number and then use Math.round to round it before translating it back:

function roundToCents(floatingValue) {

return (Math.round(floatingValue \* 100) / 100);

}

* Another option is to drop the decimal point completely, translate the problem to whole numbers problems and translate back at the very end to supply the answer.  
  For the example above, if we need to do arithmetic calculations on dollars and cents, we can translate the algorithm to work with cents only and translate the results to dollars and cents at the very end.

## Strings

* Immutable

JS accept both single-quoted and double-quoted strings (‘ and “).

* **concatenation**: + and +=
* **.length**

str**.split**(<split in substring>) e.g. “my string”.split(“ “) === arr[“my”,”string”]

* .replace(<regex>,<value or function to return the value>);
* . trim to remove whitespaces from start/end.
* All characters in JS are 16 bits wide. JS doesn’t have a character type. In order to represent one character, use a 1-length string.

### Template Literals

* **` (backtick)** allow you to add newlines in the code of the string instead of using \n
* ${variable} placeholder – can be used to pass on a variable to an expression
* To interpolate variables inside a string:

**`**String text **${**expression**}`**

### Performance

* In old browsers, it looks like there was a performance benefit for using ‘+’ instead of template literals. However,
* In more modern browsers, template literals are the fastest (as of start of 2020), followed by ‘+’ with concat function last. Therefore
* Recommendation:
  + **Use template literals where possible**. If not,
  + Use ‘+’ operator.

## Arrays

* **Mutable**

var myArray = **[**1, “string”, 7.4, [4]**]**;

var anotherArray = **new Array(**1,5,”hello”**)**;

* arr.length
* arr**.push**(val) – push val to the **end** of the array.
* val = arr**.pop**() – pop val from the **end** of the array.
* arr**.unshift**(val) – add val to the **beginning** of the array.
* val = arr**.shift**() – remove and return the **first** element of the array.
* arr.indexOf({val}) – will return the index of the val or -1 if doesn’t exist.
* arr.filter(test-function) - The filter() method creates an array filled with all array elements that pass the test-function. It does not change the original array!
* arr.map(func) - Creates a new array with the result of calling a function for each array element
* add/remove items to and array. Return the removed items   
  **arr.splice**(<from index>,<number of elements to remove>, <list of items to add>)
* **arr.slice**(<from index>,<until index exclusive>)  
  return the sub-array.
* **arr.forEach(<callback function>)**
* **Cloning Arrays:**let rainbowClone = […rainbow];

**Or:**

let rainbowClone = rainbow.slice();

### Array Comprehension

Javascript doesn’t support array comprehension like in Python (it is partially supported in a non-standardized way and not supported by Firefox). Instead, use:

Array.from({**length:** 5}, (\_, i) => i \* 2);

## Set

let animals = new Set(‘pig’, catObject,..);

## Objects

(almost) everything is an object in javascript. Everything except the primitive data types (e.g. int, etc).

Objects in JavaScript are mutable keyed collections. In JavaScript, arrays are objects, functions are objects, regular expressions are objects, and, of course, objects are objects.

All **objects** in js are **reference types**.

There are no classes in JS! We can define an object using **object literal notation** in the following way (similar to python’s dictionaties):

**var <object\_name> = {  
 <key (== property)> : <value>,  
 <key> : <value>  
};**

for example:

// object literal definition

var john = {  
 firstName: ‘John’,  
 birthyear: 1990  
};

we can also define a new object using the new Object() notation:

var jane = new Object();

jane[‘firstName’] = ‘Jane’;  
jane[‘birthyear’] = 1990;

* **Simple Fields:**  
  If we want to have the same value for the key and value, e.g.:  
  let myObj = {x: x};  
  We can remove the repetition and have JS do this under the hood. And we can write:  
  let myObj = {x};  
  instead
* We can **add additional keys** like in python:
  + <object name>.<new key> = <new value> or
  + <object name>[“<new key>”] = <new value>
* We can also **add functions**:  
  My\_object.my\_func = function(){….} or  
  <key> : function(){…}
* Get all keys:  
  **Object.keys(**<obj>**)**
* **Delete properties**:  
  delete obj.key; or delete obj[“key”]
* **Function in an object:**
  + **ES5** and below:  
    const person = {  
      name: "Taylor",  
      sayHello: function() {  
        return `Hello! My name is ${**this**.name}.`;  
      }  
    };
    - * **ES6:**const person = {  
          name: "Taylor",  
          sayHello() {  
            return `Hello! My name is ${**this.**name}.`;  
          }  
        };

### Object Creation Patterns

#### Constructor:

You can create an object by defining its constructor:

function Bird() {  
  this.name = "Albert";  
  this.color = "blue";  
  this.numLegs = 2;  
}

let bird = **new** Bird();

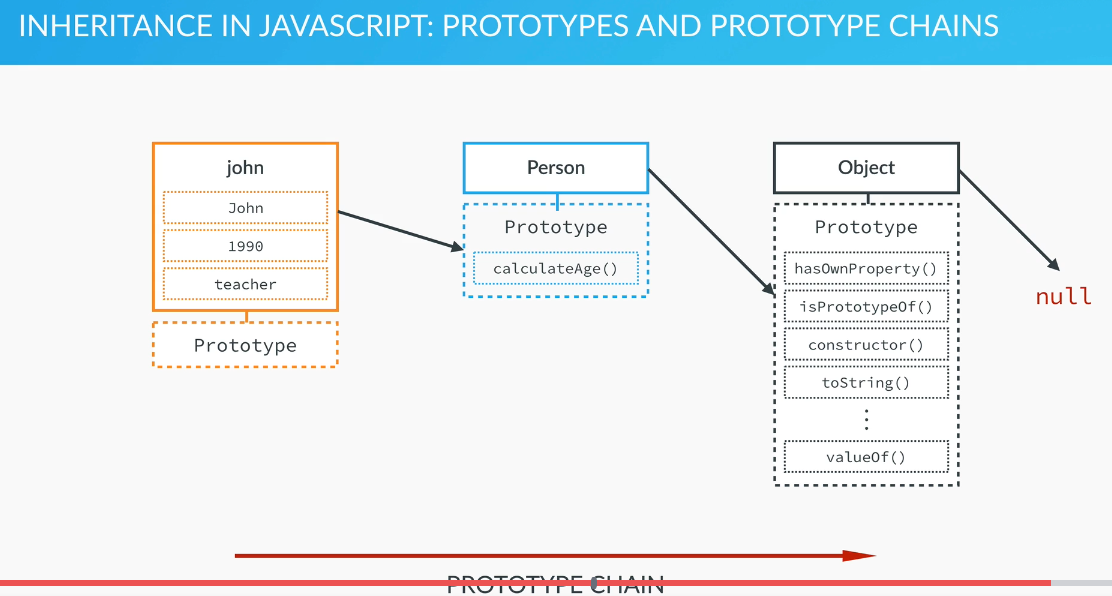
When the ‘new’ is called:

1. it creates a new empty ‘Bird’ object
2. It sets the new object’s ‘this’ parameter to the new object
3. It calls the object’s constructor with its ‘this’ parameter set.

#### Object.Create

### Inheritance and Prototypes

* Each JS object has a **prototype property,** which makes inheritance possible in js.
* We put all the **methods** and **parameters** that we want other objects to inherit in the object’s prototype.
* The constructor’s prototype property is **NOT** the prototype of the Constructor itself; it’s the prototype of **ALL** instances that are created through it.
* When a certain method (or property) is called, the search started in the object itself, and if it cannot be found, the search moves on the object’s prototype. This continue through the inheritance chain until the method/property is found in the **prototype chain**.



To use the prototype to define parameters and methods:

// If we define the functions as part of the PROTOTYPE, we define it only once for every object's prototypes and all the objects we create

// from it will be able to access and use it without having to duplicate

// the code in all of them:

Person.*prototype*.getJob =

function() {

    console.log(`${*this*.name} is a ${*this*.job}`);

    return *this*.job;

}

Note: if would define the function as part of the constructor:

var Person = function(name, yearOfBirth, job) {

*this*.name = name;

*this*.yearOfBirth = yearOfBirth;

*this*.job = job;

*this*.calculateAge = function() {

        var thisYear = new Date().getFullYear();

        return (thisYear - *this*.yearOfBirth);

    }

}

We will actually cause it to be duplicated for every object that gets created using this constructor, causing a lot of duplication!

* When we define a parameter in an object’s prototype, it too is inherited by all the objects that are created from the constructor and they all get the same **default** value.   
  Note: after they get the default value, they can change it and it will save a local parameter on the object itself (not on the prototype) with the same name that will hide the prototype’s parameter. The change will only effect their own object and not other objects that inherit from the same prototype. For example:

Person.*prototype*.lastName = 'Smith';

console.log(john.lastName); // Smith

john.lastName = 'Short';

console.log(john.lastName); // Short

console.log(jane.lastName); // Smith

console.info(john);

// john now have a property: lastName: ‘Short’ and

// also, in its Person.prototype, a property:

// lastName: ‘Smith’

##### Object keys:

Object’s keys can be strings, literals (strings without the quotation marks), numbers.

Note: in any case, JS will typecast them into string.

##### Object notation

Can use either . or [] to access object’s properties.

The [] notation can allow us to access the object’s fields dynamically, depending on the value in the variable.

##### Properties

* + - Own properties – these are the regular properties that are defined per object (i.e. this.property). Like normal C++ member variables.
    - Prototype properties – like C++ static member variables – the same properties are shared between all the objects of this type. This is not true!! When I ran a test, I saw that if I change the prototype property on one object, it doesn’t change it on different obects!!  
      This is implemented in javascript by having the prototype be actually an object that is shared among all the instances of the class.  
      Bird.**prototype**.numLegs = 2;

You can also add a whole bunch of properties to an already existing object (i.e. constructor):

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

Dog.prototype = {  
 constructor: Dog,  
 numOfLegs: 4,  
 eat() {…},  
 …   
};

Note: when setting the prototype to a new object, you must re-define the constructor in the prototype (see above in red), otherwise, it will be erased!!

* If the object has only prototype properties without any own properties, you can also define a new object of this type like this:  
  let myDogo = Object.create(Dog.prototype);  
  Note: this method will only initialize the prototype properties. All the member properties will remain undefined!!.
* Inheritance with objects:   
  **NOTE:** in ES6 and up you should use proper classes instead!!  
  Bird.prototype = Object.create(Animal.prototype);  
  Bird.prototype.constructor = Bird;

Now you can add additional properties and functions specific to the inherited object:  
Bird.prototype.myFunc = function() {…}

##### Functions

* + - **hasOwnProperty**(propname) – does the property exist in the object itself? Not: properties that are defined in the object’s prototype will return false here.
    - **Instanceof** – is the object an instance of the constructor:  
      if( bird instanceof Bird)…
    - **JSON.stringify**(object) – will print all the object’s fields and values for debugging.
    - **console.info**(object) – will print the object in a format that allows inspecting it’s internal structure.

##### Mixin

As you have seen, behavior is shared through inheritance. However, there are cases when inheritance is not the best solution. Inheritance does not work well for unrelated objects like Bird and Airplane. They can both fly, but a Bird is not a type of Airplane and vice versa.

For unrelated objects, it's better to use mixins. A mixin allows other objects to use a collection of functions.

let flyMixin = function(obj) {  
  obj.fly = function() {  
    console.log("Flying, wooosh!");  
  }  
};

**The flyMixin takes any object and gives it the fly method**.

let bird = {  
  name: "Donald",  
  numLegs: 2  
};  
  
let plane = {  
  model: "777",  
  numPassengers: 524  
};  
  
flyMixin(bird);  
flyMixin(plane);

Here bird and plane are passed into flyMixin, which then assigns the fly function to each object. Now bird and plane can both fly:

bird.fly(); // prints "Flying, wooosh!"  
plane.fly(); // prints "Flying, wooosh!"

Note how the mixin allows for the same fly method to be reused by unrelated objects bird and plane.

##### Closure

The simplest way to **make properties private** is by creating a variable within the constructor function. This changes the scope of that variable to be within the constructor function versus available globally. This way, the property can only be accessed and changed by methods also within the constructor function.

function Bird() {  
  let hatchedEgg = 10; // private property  
  
  this.getHatchedEggCount = function() { // publicly available method that a bird object can use  
    return hatchedEgg;  
  };  
}  
let ducky = new Bird();  
ducky.getHatchedEggCount(); // returns 10

Here getHachedEggCount is a privileged method, because it has access to the private variable hatchedEgg. This is possible because hatchedEgg is declared in the same context as getHachedEggCount. In JavaScript, a function always has access to the context in which it was created. This is called closure

## Classes

JS doesn’t support full-fledged class-based implementation of object-oriented paradigm like in C++ or python.

Instead, it has a syntax to allow re-using an object definition to instantiate new objects.

### ES5 and Below

var SpaceShuttle = function(targetPlanet){  
  this.targetPlanet = targetPlanet;  
 this.count = 0;  
}

**SpaceShuttle.prototype.countMyCalls** = function() {  
 this.count++;  
}

var zeus = **new** SpaceShuttle('Jupiter');

### ES6

class SpaceShuttle {  
 constructor(targetPlanet){  
 this.targetPlanet = targetPlanet;  
 this.count = 0;  
  
 // Note: for each function that you allow calling from   
 // outside (e.g. React callbacks) – you must bind the   
 // function to this whenever the constructor is called:  
 **this.countMyCalls = this.countMyCalls.bind(this);**

}

countMyCalls() {  
 this.count++;  
 }

// another option will be, using babel class-properties:  
 // this syntax will bind the method automatically to the   
 // class so we won’t have to bind it in the constructor.  
 countMyCalls = () {…}

}

const zeus = new SpaceShuttle('Jupiter');

#### This

Even though it appears this refers to the object where it is defined, it is not until an object invokes the this Function that this is actually assigned a value. And the value it is assigned is based **exclusively** on the **object** that invokes the this Function. this has the value of the invoking object in most circumstances. However, there are a few scenarios where this does not have the value of the invoking object:

1. When **passing the function through a Callback**:

// We have a simple object with a clickHandler method  
// that we want to use when a button on the page is   
// clicked

var user = {

count:0,

clickHandler:function (event) {  
 count += 1;  
 console.log(`Current count is: ${this.count}`);  
 }  
};

// The button is wrapped inside a jQuery $ wrapper,   
// so it is now a jQuery object  
// And the output will be undefined because there is  
// no data property on the button object

$ ("button").click (user.clickHandler);   
// Cannot read property count of undefined

**How to fix:**

Instead of this line:

$("button").click(user.clickHandler);

We have to **bind** the clickHandler method to the user object like this:

$("button").click(user.clickHandler.bind(user)); // count

1. When **passing to an inner method (closure):**

var user = {  
 tournament:"The Masters",  
 data :[ {name:"T. Woods", age:37},  
 {name:"P. Mickelson", age:43}],

clickHandler:function () {  
 // the use of this.data here is fine, because   
 // "this" refers to the user object, and data is   
 // a property on the user object.

this.data.forEach (function (person) {

// But here inside the anonymous function   
 // (that we pass to the forEach method),   
 // "this" no longer refers to the user object.  
 // This inner function cannot access the outer   
 // function's "this"

console.log("What is This referring to? " + this);   
 //[object Window]

console.log(person.name + " is playing at " +  
 this.tournament);

// T. Woods is playing at undefined  
 // P. Mickelson is playing at undefined

})  
 }  
};

user.clickHandler(); // What is "this" referring to? [object Window]

**How to fix:**

**Initialize a local variable to this before calling the anonymous function:**

var user = {  
 tournament:"The Masters",  
 data :[ {name:"T. Woods", age:37},  
 {name:"P. Mickelson", age:43} ] ,

clickHandler:function () {

**var theUserObj = this;**  
 this.data.forEach(function (person) {

console.log (person.name + " is playing at " +   
 theUserObj.tournament);

});  
 }  
};

1. **When assigning the method to a variable:**

var information = {  
 count: 0,  
 showCounter:function (event) {  
 console.log (`Current Counter is: ${this.count}`);  
 }  
};

// Assign the user.showData to a variable -   
// will cause the 'this' to be undefined in showUserCounter:

var showUserCounter = information.showCounter;  
showUserCounter(); // Current Counter is **undefined**

**how to fix:**

bind the new function to the object:

var fixedUserCounter = information.showCounter.**bind(information)**;

fixedUserCounter(); // Current Counter is: 0

##### 

#### Static Functions

called without [instantiating](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript#The_object_(class_instance)) their class and **cannot** be called through a class instance. Static methods are often used to create utility functions for an application.”:

class Point {

…

**static** distance(a, b) {  
 const dx = a.x - b.x;  
 const dy = a.y - b.y;  
 return Math.hypot(dx, dy);  
 }

}

#### Setters & Getters

By convention - \_ mark private member variable (although it’s not really private…)

class Book {  
  constructor(author) {  
    this.**\_**author = author;  
  }  
  // getter  
  **get writer**(){  
    return this.\_author;  
  }  
  // setter  
  **set writer**(updatedAuthor){  
    this.\_author = updatedAuthor;  
  }  
}  
const lol = new Book('anonymous');  
console.log(lol.writer);  // anonymous  
lol.**writer** = 'wut';  
console.log(lol.writer);  // wut

# Variables

Declaring:

var ourVariable; // ES5 and bellow. Should not use  
 // anymore!

Use **let** instead of var to declare variables! This will make sure that you can’t overwrite existing variables by mistake!

Use: **“use strict”;** to enable strict mode that will catch common coding mistakes and “unsafe” actions!

* Variable names can be made up of numbers, letters, and $ or \_, but may not contain spaces or start with a number.
* When declaring a new variable, its initial value is undefined
* If you do a mathematical operation on an undefined variable your result will be NaN which means *"Not a Number"*.
* If you concatenate a string with an undefined variable, you will get a literal *string* of "undefined".
* **const** <var> = <val> - define a const variable with the same advantages as let.  
  should be defined with ALL\_CAPS!
  + Note: if you define an array or object as const, you will still be able tot change it’s internal parameters. You just won’t be able to change it to point to a different array/object.
  + To prevent change the data, you can use **Object.freeze(obj)** function instead.
  + Best practice: unless you know that you’ll need to change the variable, **always define it as const**. This way to prevent accidental change!

# Statements vs Expressions

* Expressions return an immediate result for example:

2+5;

myFunc(7);

…etc

* Statement is code that defines something. For example:

Function myFunc {return 8;}

Or

If(true) {console.log(‘true’);}

Statements return ‘**undefined’**

# Useful Functions

* setInterval(() => ++myVariable, 500); - will add 1 to myVariable every 500 ms (timer)

# Functions

* Primitives are passed by value
* Reference types are passed by reference

**function** **funcName**(var1, var2) {

…

return result;

}

Is equal to:

let **funcName = function**(var1, var2) {…}

If the function doesn't have a return statement, when you call it, the function processes the inner code but the returned value is undefined.

##### Default Arguments

* JS support default arguments in functions.
* Unlike python, it re-define the default argument every time the function is called and

##### The Rest Operator (args)

…args – can accept a variable number of arguments:

function howMany(...args) {  
  return "You have passed " + args.length + " arguments.";  
}

##### Static Variables

Since functions in javascript are objects, you can define properties on them that will act as static variables. For example:

function someFunction(arg) {  
 if( **someFunction.staticCounter** == undefined) {  
 someFunction.staticCounter = 0;  
 }  
 else {  
 someFunction.staticCounter++;  
 }  
}

##### The Spread (unpack) Operator

When we need to unpack a string/array to individual characters/values in order to pass them as arguments to a function.

Example:

function sum(x, y, z) {  
 return x + y + z;  
}

const numbers = [1, 2, 3];

console.log(sum(...numbers));  
// expected output: 6

Another example:

const arr1 = ['JAN', 'FEB', 'MAR', 'APR', 'MAY'];  
let arr2 = [...arr1];

##### Destructuring Assignment

Unpacking objects into variables:

var voxel = {x: 3.6, y: 7.4, z: 6.54 };  
var x = voxel.x; // x = 3.6  
var y = voxel.y; // y = 7.4  
var z = voxel.z; // z = 6.54

Here's the same assignment statement with ES6 destructuring syntax:

const { x, y, z } = voxel; // x = 3.6, y = 7.4, z = 6.54

If instead you want to store the values of voxel.x into a, voxel.y into b, and voxel.z into c, you have that freedom as well.

const { x : a, y : b, z : c } = voxel;   
// a = 3.6, b = 7.4, c = 6.54

when we want to assign values as part of the code, we will enclose the destructing assignment in parentheses:

({x, y, z} = voxel);

**With Default values:**

({x, y, z, n=15} = voxel);

**Using different variable names:**

|  |
| --- |
| const person = {  name: ‘John Doe’,  country: ‘Canada’ };  // Assign default value of 25 to years  // if age key is undefined ({name: fullname, country: place, age: years = 25} = person; |

**Nested Object Destructuring:**

const student = {  
 name: 'John Doe',  
 age: 16,  
 scores: {  
 maths: 74,  
 english: 63  
 }  
};

// We define 3 local variables: name, maths, science  
const { name, scores: {maths, science = 50} } = student;

**Skipping**

Deconstructing objects doesn’t require skipping like in arrays since you can explicitly only set the variables you care about.

**Array Destructuring**

**:**

const rgb = [255, 200, 0];

// Array Destructuring  
const [red, green, blue] = rgb;

// Array Destructuring assignment –   
//doesn’t require the parentheses:  
[red, green] = rgb;  
  
// Skip the first two items:  
[,,blue] = rgb;

**Nested Array Destructuring:**

const color = ['#FF00FF', [255, 0, 255], 'rgb(255, 0, 255)'];

// Use nested destructuring to assign red, green and blue  
const [hex, [red, green, blue]] = color;

console.log(hex, red, green, blue); // #FF00FF 255 0 255

**Swapping variables:**

[width, height] = [height, width];

**Rest Items**

You can use the rest operator to assign the rest of the items:

const rainbow = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet'];

// Assign the first and third items to red and yellow  
// Assign the remaining items to otherColors variable  
// using the spread operator(...)

const [red,, yellow, **...otherColors**] = rainbow;

console.log(otherColors); // ['green', 'blue', 'indigo', 'violet']

**Cloning Arrays**

const [...rainbowClone] = rainbow;

**Mixed Deconstructing**

const person = {  
 name: 'John Doe',  
 age: 25,  
 location: {  
 country: 'Canada',  
 city: 'Vancouver',  
 coordinates: [49.2827, -123.1207]  
 }  
}

// Observe how mix of object and array destructuring is being used here  
// We are assigning 5 variables: name, country, city, lat, lng

const {name, location: {country, city, coordinates: [lat, lng]}} = person;

console.log(`I am ${name} from ${city}, ${country}. Latitude(${lat}), Longitude(${lng})`);

## Immediately Invoked Function Expression (IIFE)

A common pattern in JavaScript is to execute a function as soon as it is declared:

**(**function () {  
  console.log("Chirp, chirp!");  
}**)()**; // this is an anonymous function expression that executes right away  
// Outputs "Chirp, chirp!" immediately

Note that the function has no name and is not stored in a variable. The two parentheses () at the end of the function expression cause it to be immediately executed or invoked. This pattern is known as an immediately invoked function expression or IIFE.

### Create a Module Using IIFE

An immediately invoked function expression (IIFE) is often used to group related functionality into a single object or module. For example, an earlier challenge defined two mixins:

function glideMixin(obj) {  
  obj.glide = function() {  
    console.log("Gliding on the water");  
  };  
}  
function flyMixin(obj) {  
  obj.fly = function() {  
    console.log("Flying, wooosh!");  
  };  
}

We can group these mixins into a module as follows:

let motionModule = (function () {  
  return {  
    glideMixin: function (obj) {  
      obj.glide = function() {  
        console.log("Gliding on the water");  
      };  
    },  
    flyMixin: function(obj) {  
      obj.fly = function() {  
        console.log("Flying, wooosh!");  
      };  
    }  
  }  
}) (); // The two parentheses cause the function to be immediately invoked

Note that you have an immediately invoked function expression (IIFE) that returns an object motionModule. This returned object contains all of the mixin behaviors as properties of the object.

The advantage of the module pattern is that all of the motion behaviors can be packaged into a single object that can then be used by other parts of your code. Here is an example using it:

motionModule.glideMixin(duck);  
duck.glide();

## Pure Functions

A pure function in JavaScript is one that **given the same input, will always return the same output without side effects.** Put simply, pure functions only depend on their input arguments.

## Common Functions

* typeof
* Math.random() – return 0<=rand<1;
* Math.floor() – round the number down to its nearest whole number
* parseInt(<string>,<radix>) – convert the string to an int, or return NaN. Radix is the base 2-36.
* console.log(<text>); - print debug information
* console.clear() – clear the console.
* <string>.replace(<old\_string>,<new\_string>) find and replace string.

## Anonymous Functions

### ES5 and Below

const myFunc = function() {  
  const myVar = "value";  
  return myVar;  
}

### ES6

const myFunc = (args) => {  
  const myVar = "value";  
  return myVar;  
}

When there is no function body, and only a return value, arrow function syntax allows you to omit the keyword return as well as the brackets surrounding the code. This helps simplify smaller functions into one-line statements:

const myFunc = (args) => "value"

if we pass only one argument, we can remove the parentheses:

const myFunc = arg1 => "value"

## Curried Functions

The arity of a function is the number of arguments it requires. Currying a function means to convert a function of N arity into N functions of arity 1.

In other words, it restructures a function so it takes one argument, then returns another function that takes the next argument, and so on.

Here's an example:

//Un-curried function  
function unCurried(x, y) {  
  return x + y;  
}  
  
//Curried function  
function curried(x) {  
  return function(y) {  
    return x + y;  
  }  
}  
curried(1)(2) // Returns 3

This is useful in your program if you can't supply all the arguments to a function at one time. You can save each function call into a variable, which will hold the returned function reference that takes the next argument when it's available. Here's an example using the curried function in the example above:

// Call a curried function in parts:  
var funcForY = curried(1);  
console.log(funcForY(2)); // Prints 3

## Partial Application

partial application can be described as applying a few arguments to a function at a time and returning another function that is applied to more arguments.

Here's an example:

//Impartial function  
function impartial(x, y, z) {  
  return x + y + z;  
}  
var partialFn = impartial.bind(this, 1, 2);  
partialFn(10); // Returns 13

# Regular Expression

By default, the regex of JS is greedy e.g. find the longest possible match.

### Test method – on regex

let testStr = "freeCodeCamp";  
let testRegex = /Code/;  
testRegex.test(testStr);  
// Returns true

### Match method – on string

Returns an array with the matched string.

let ourStr = "Regular expressions";  
let ourRegex = /expressions/;  
ourStr.match(ourRegex);  
// Returns ["expressions"]

### Create a Regular Expression

To create a regular expression from a string:  
new RegExp(string, flags) where flags are g or i. For example:  
myRegex = new RegExp('god', 'i');

### JS Regex Patterns

* Find string1 or string 2: /string1**|**string2/
* **Flags:**  
  You can add flags for the regex after the closing / of the regex. For example: the ignore case flag /i:
  + Ignore case in the regex: /regex/**i**
  + Find all patterns in string: /regex/**g**
* **Lookaheads**will check for patterns **starting immediately in the current position** **without consuming them** (so after the search, the regex engine is still in the same position as before). This is useful for checking for two or more conditions on the same string. Since lookahead doesn’t consume the string, you can test for different conditions on the same string in the same regex. This is useful for checking validity conditions or for password checks:
  + **Positive lookahead: (?=**<regexp>**)** – will match if found the <regexp>
  + **Negative lookahead (?!**<regexp>**)** – will match if could not find the <regexp>
  + Example: check that a password have between 3-6 characters and at least one number:

let checkPass = /(?=\w{3,6})(?=\d)/;

* **Capture groups:**
  + In the regex: define with (<exp>) and   
     re-use with **\**<group num from 1>
  + In the match group (after the match): use with **$**<group numb from 1>

## $

$(window).bind('load', function() {

$('img.protect').protectImage();

});

There's nothing mysterious about the use of $ in JavaScript. $ is simply a valid JavaScript identifier.

JavaScript allows upper and lower letters, numbers, and $ and \_. The $ was intended to be used for machine-generated variables (such as $0001).

Prototype, jQuery, and most javascript libraries use the $ as the primary base object (or function). Most of them also have a way to relinquish the $ so that it can be used with another library that uses it. In that case you use jQuery instead of $. In fact, $ is just a shortcut for jQuery.

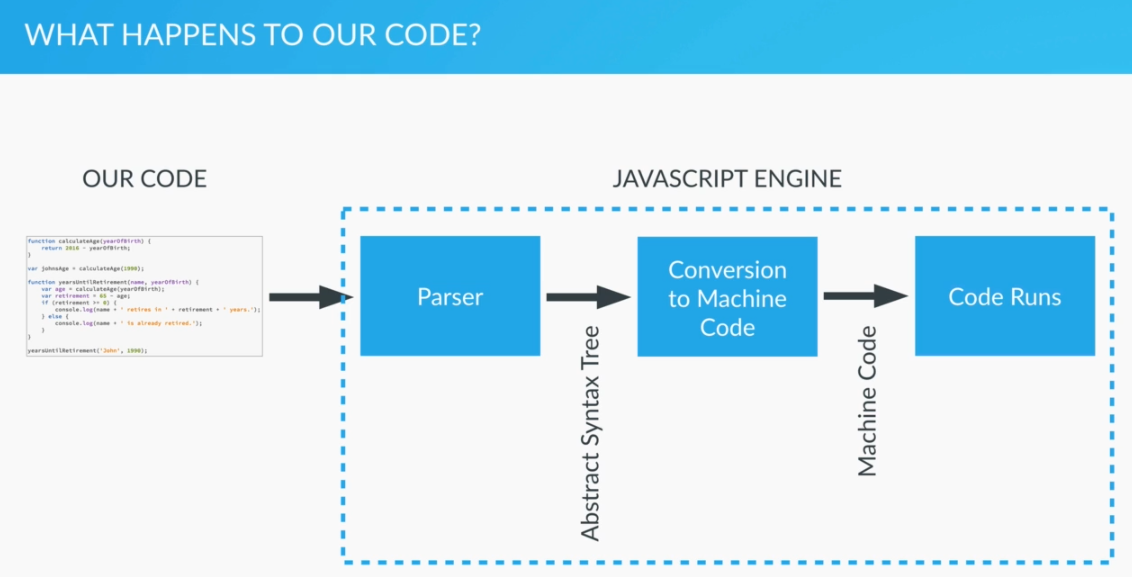
# External Functions

## Web API

* SetTimeout(callbackFunction, seconds) – will put the callbackFunction in the callbackQueue after {x} seconds. Note: this does not mean that the callbackFunction will actually run after {x} seconds – since it will have to wait until the stack is empty and it’s the 1st function in the callback queue before it’s picked up into the stack to run.

# Behind the Scenes

* JS code always run in a host environment. This can be:
  + Browser – for most client-side applications
  + Node JS Server – for node js code
  + Application – for applications that except js input.
* The JS Engine in the host execute the JS in the following case:



* + There are many different JS engines like the Chrome’s V8 or Mozila’s SpiderMonkey engine.

## Execution Context

The context in which js runs. We can look at it as a box/container/wrapper which stores variables and in which a piece of our code is evaluated and executed.

### Default Execution Context = Global Execution Context

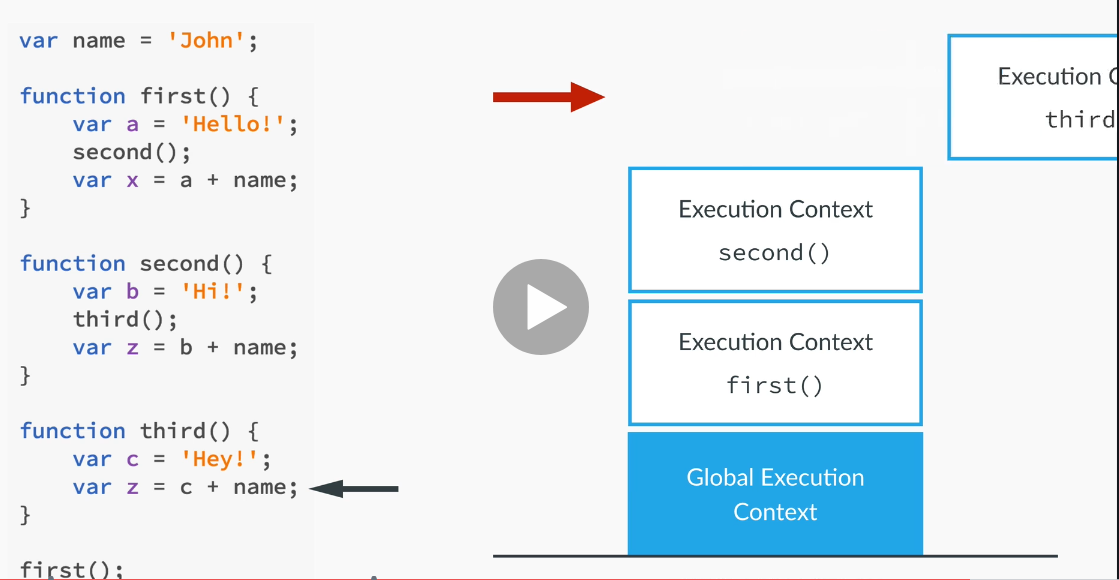
The default execution context is the Global execution context. This includes:

* + All code that is **not inside any function**
  + It’s associated with the **global object**
  + In the browser, that’s the window object.  
    So when we declare a variable outside of any function:  
    let lastName = “Smith”;  
    we actually define it inside the global object (in a browser -> the window):  
    lasteName === window.lastName; // true

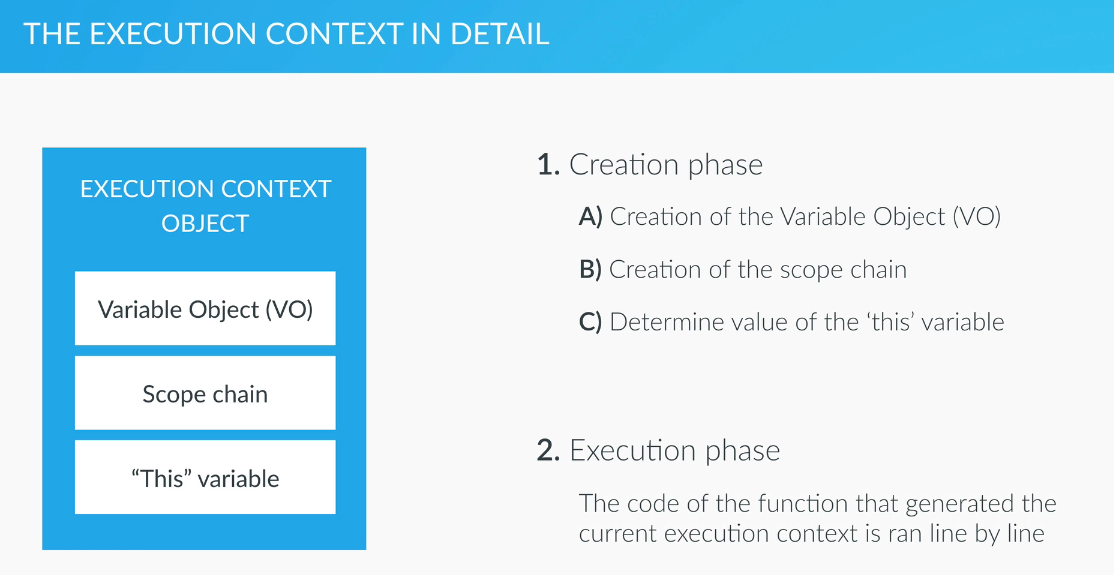
### Function Execution Context

* Whenever we declare a function, it creates a new execution context.
* Each execution context is isolated from the other execution context but it can access it’s containing execution context (e.g. global).

#### Execution Context Stack

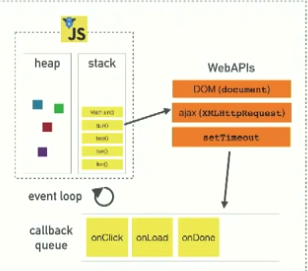


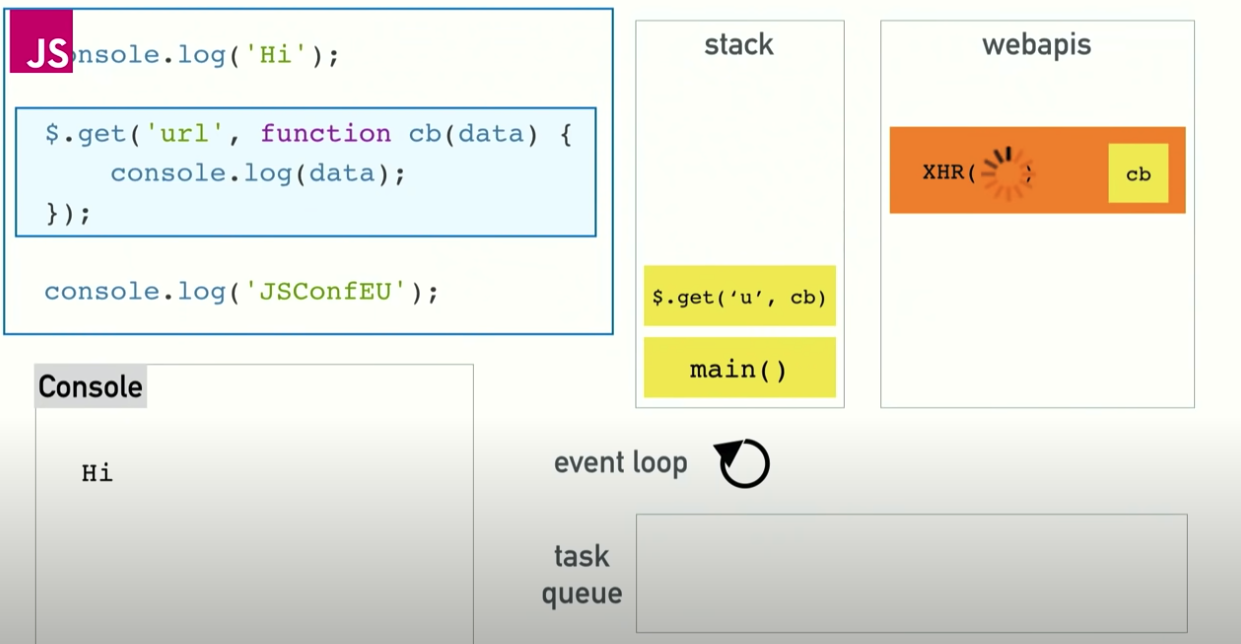
Every time a new execution context is created, it execute the following:



* **Variable Object (VO) creation:**
  + The argument object is created, containing all the arguments that were passed into the function
  + **Hoisting:  
    Note:** the hoisting happens in the creation phase, before the code is executed!
    - Code is scanned for **function declarations**: for each function, a property is created in the Variable Object, **pointing to the function**.
    - Code is scanned for **variable declarations**: for each variable, a property is created in the Variable Object, and **set to undefined**.
* **Creation of scope chain:**
  + Calculate the scope chain (who has access to what) in this execution context.
* **this:**
  + Regular function call: the this keyword points at the global object (the window object, in the browser)
  + Method call: the ‘this’ variable points to the object that is calling the method.  
    Note: the **‘this’ is not assigned a value until there is the first call** to an object’s method! Is it not assigned when the object is defined. Only when it actually calls a method!   
    This is because the ‘this’ variable is defined in the execution context of the method and that is created only when the method is being called!

### Event Loops





The call stack: javascript is a single-threaded language. This means that it has:  
one call stack that execute one thing at a time.

However,

#### Multithreading:

The WebAPIs are actually calling different threads that run concurrently to the JS runtime (e.g. V8). In Node JS, instead of WebAPIs, C++ code is handling the multi-threading of the server-side calls. In any case, these external APIs take callback functions so once they are finished, they put the callback function call in the callback queue

#### The Main Thread

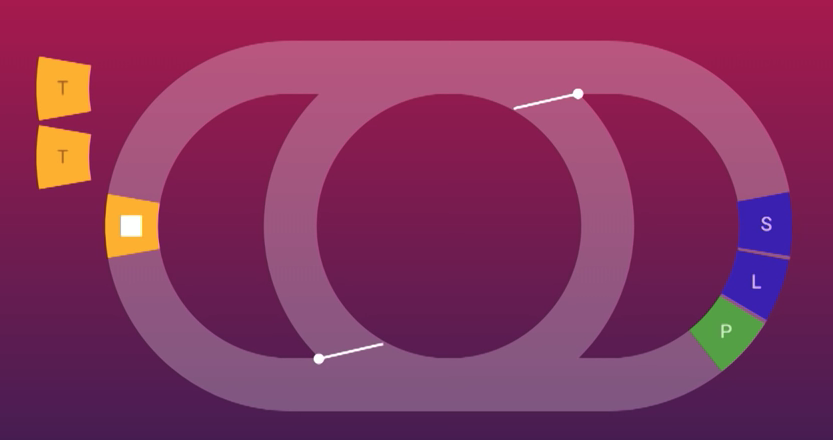
The **main thread** in the browsers is where the DOM lives and where user interaction happens including rendering. The fact that it’s single threaded means that sync is pretty straight forward. However, slow things (>=200ms) will cause a noticeable degredation in user experience.

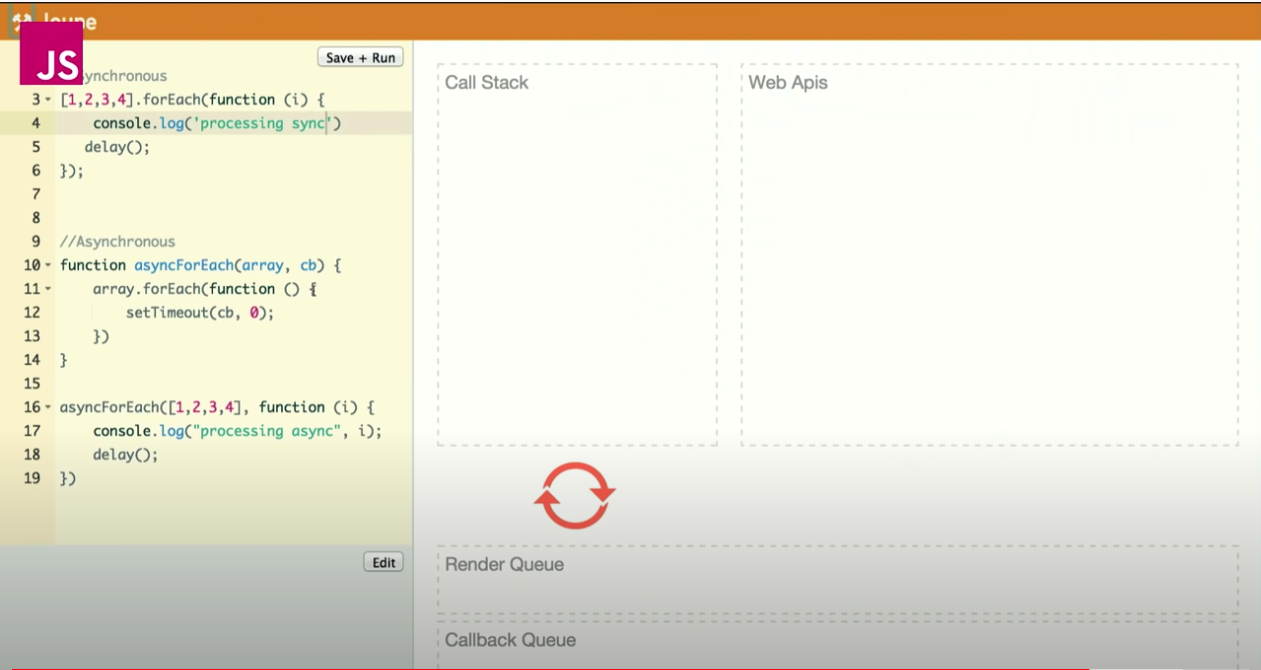
#### Event Loop

Whenever JS’s main stack is empty, the JS runtime look at the callback (task) queue and if there is anything there, it will put it in the stack to execute.

* + Note: this is also why we would call setTimeout(callbackFun,**0**) with 0 timeout – this will allow us to defer the callback’s execution until the stack is cleared.
  + Note:

#### Render Queue



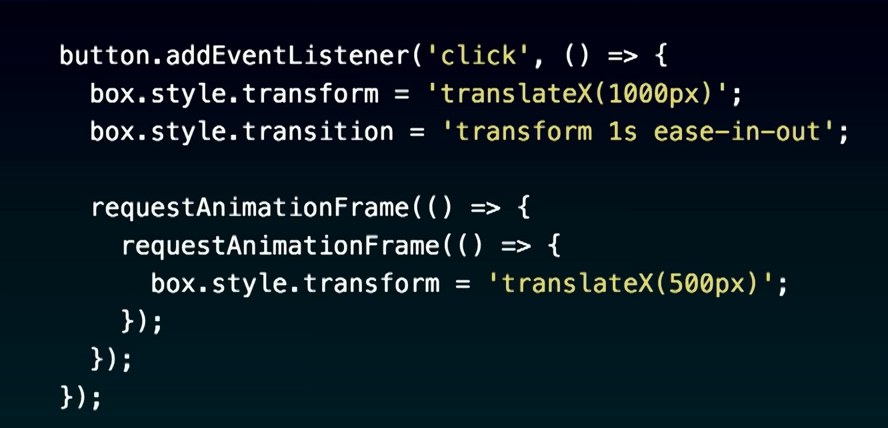
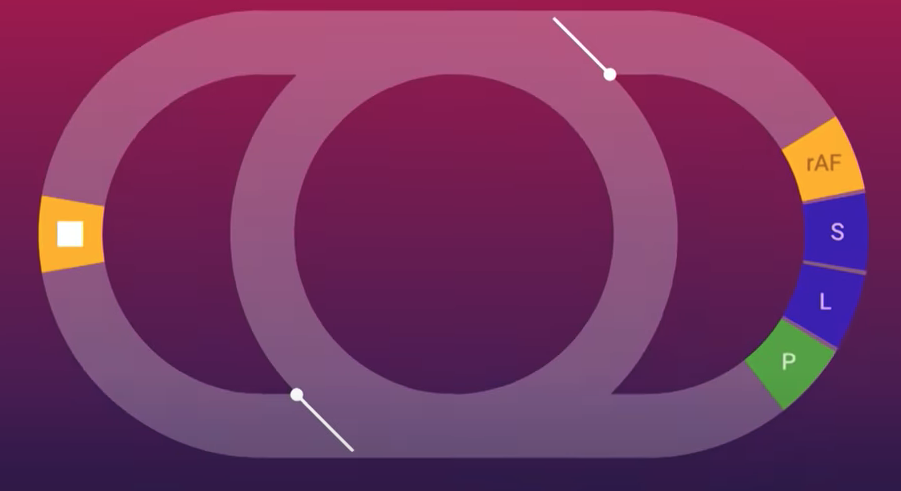


The page (browser tells the JS to do the rendering round) tries to render usually 60 times per second (depends on the browser + screen rate). However, it can’t render while there are calls in the call stack (v8). Therefore, it will put a rendering request in the Render Queue which had higher priority than the Callback Queue so that when the

* + - Call Stack is empty (this means that v8 is not actively working on any task)
    - If there is a rendering request, the page will render (allowing users to interact with it). This will happen even if there are callbacks waiting in the callback queue. Note: if there is a queue of rAFs already in the rendering queue when it executes, they will all get executed (except for ones that are added while the renderer is running)
  + If there is no rendering request in the queue, the next callback function will go on the call stack to execute. Note: the callback (task) stack execute one task at a time. If the other stacks (rendering/microtasks) have tasks in them, they will run after this callback and before the callback queue is checked again.
  + In every rendering round, the renderer do the following in this order:
    - requestAnimationFrame (**rAF**) – user callback that runs as part of the rendering queue. This is a great way to do all the work that’s required to update the visual rendering in a way that is synchronized with the rendering of the page ; see orange below
    - **S**tyle calculations – looking at the CSS and figuring out what style apply to each element; see blue below
    - **L**ayout – where each element need to be on the page; see green below
    - **P**ainting - Doing the actual rendering (pixels on screen)  
      
  + The Microtasks queue will run all the microtasks in it until it’s completely empty. Including all the microtasks that are added to it by currently running microtask. So it can prevent rendering and starve the js callback (tasks) queue :



**Notes:**

* This means that all the synchronous changes you make to the DOM in your methods, will run as one unit before the DOM gets rendered for the next time!! So you don’t need to worry about race conditions between two sequential lines of code!
* It also means that all your js code that affect the rendering will run in one block before the page is rendered. So if for example, you want to code some animation that needs to render, make sure that you split is across different rAFs. Otherwise, when the Style Calculation gets called, it only see the final elements values so it won’t see all the stages before it. This is an example of how to render an animation from position 1000 to 500. Note: if you don’t use the double rAFs, it will only render the box in position 500 without any animation:
*   
  
  + In Chrome, you can use the web animation API to avoid this issue but it’s not available in other browsers.
  + In Safari, the rAF is scheduled after the paint instead of at the start of the rendering queue so there is an additional delay for your rendering + Safari will display the state of the system before the rAF ran. That’s a bug and they have a ticket on it.

#### Microtasks Queue

The Microtasks Callbacks Queue runs every time the JS v8 execution stack is empty. Note: it has higher priority than the callback (tasks) queue or the rendering queue!!

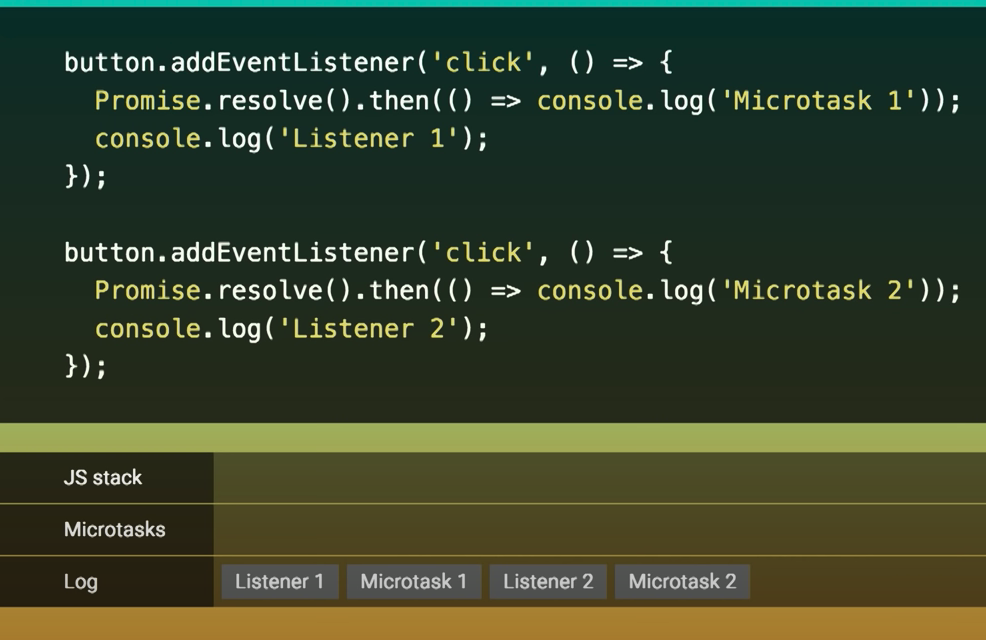
It used by:

* + - Promises
    - Mutation Observer

Promise example:

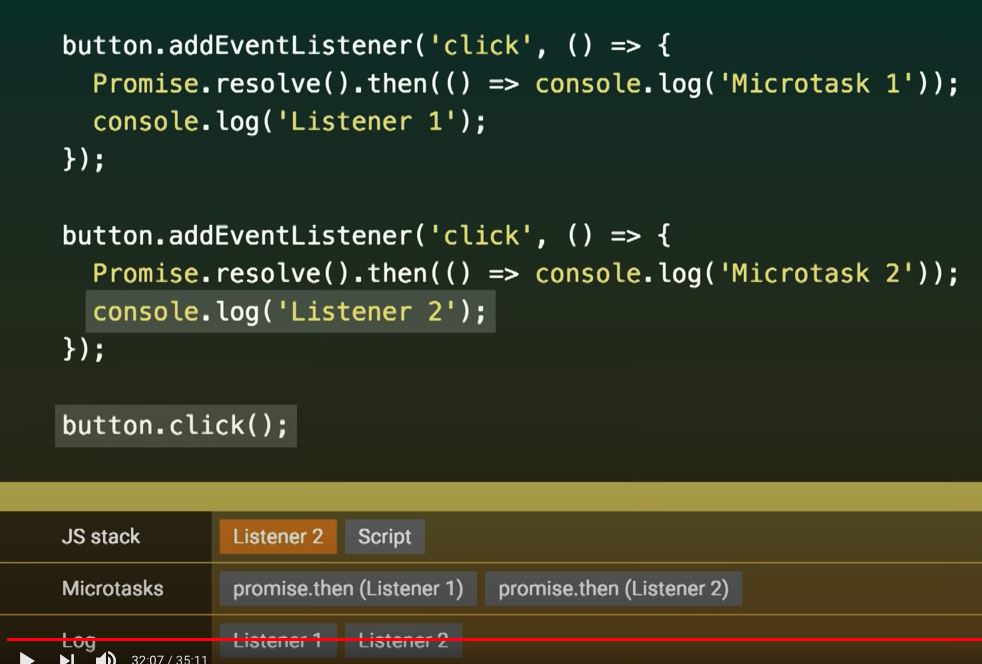


console.log(‘Hey!’); - will be queued in the microtask queue and will run after the console.log(‘Yo!’); has finished running.



In this example, although both listeners are triggered and put on the callback queue at the same time, since the microtasks have higher priority, the microtask of the 1st listener will run before the 2nd listener is executed!

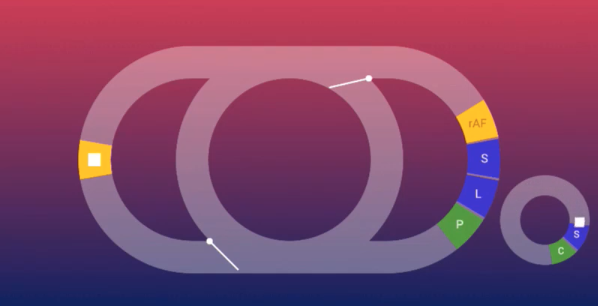
However, if instead of the user clicking the button, we have js code clicking it, the **button.click(); will only return after all the listeners have finished executing** (!!!) this means that the order of execution will be different:



Because ‘Script’ is still executing, the microtasks queue isn’t run and both listeners run before their respective microtasks.

#### Compositor

The compositor is another thread that runs independently from the main thread and is responsible to take the Painted bitmaps from the Rendering Loop (the ‘P’) and actually paint the screen:



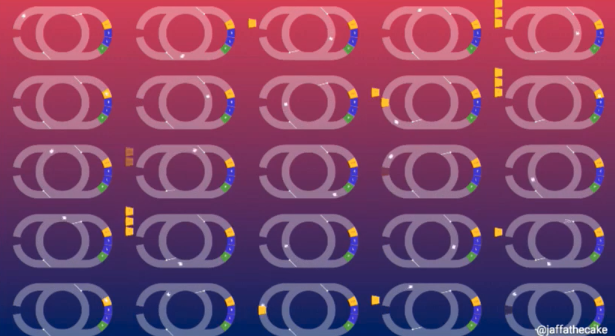
compositor

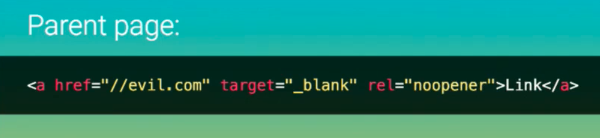
It can only do simple things like moving or rotating an image.

Transform actions for example can be done by the compositor

top or left actions can’t be done by it.

#### Event Loops



* On modern browsers, different tabs run on different event loops.
* The spec state that pages must share the same event loop if they can gain synchronous access to each other.  
    
  
* For hrefs, it’s recommended to add ‘rel=”noopener”’ for both security reasons (isolate the child page from the parent) and in some browser, it will open the child on a different even loop:  
    
  

## Scope

JS includes the following scopes (only):

* global
* function level   
  Note: a function that’s defined within another function gets access to the scope of the outer function (lexical scoping)

**let** (but not var) also support the following scopes:

* block  
  Note: when declaring a variable with var – any variable used/defined in the block has a global/function scope

In JavaScript, a name enters a scope in one of four basic ways:

1. **Language-defined:** All scopes are, by default, given the names this and arguments.
2. **Formal parameters:** Functions can have named formal parameters, which are scoped to the body of that function.
3. **Function declarations:** These are of the form function foo() {}.
4. **Variable declarations:** These take the form var foo;.

Note:

* Variables which are used without the var keyword are automatically created in the global scope. This can create unintended consequences elsewhere in your code or when running a function again. You should always declare your variables with var.

## Hoisting

* **Function** **declarations** and **variable declarations** are always moved (“hoisted”) invisibly to the top of their containing scope by the JavaScript interpreter.
* **Assignment** are **not hoisted**. Even if they are a part of a declaration – the declaration itself (e.g. var x) will be hoisted to the start of the scope but the assignment will stay in it’s original place (x=5;)
* **Note:** functions expressions (var myFunc = function(){…}) are not hoisted (since it’s actually assignment and not declaration).

Function parameters and language-defined names are, obviously, already there. This means that code like this:

var x = 5;

console.log(x); // 5

var y = 10;

is actually interpreted like this:

var x, y; // this line simply declares x and y at the same time.

x = 5;

console.log(x); // 5

y = 10;

Notice how the declaration of y moved to the top of the scope. And also notice how the first line doesn't set a value for neither x nor y. After var x, y; both x and y are undefined. This is because the **assignment** portion of the declarations were **not hoisted**. **Only the name is hoisted**.

**This is also true for functions**:

* When declaring a function as variable:

example2();

var example2 = function() {

console.log("Ran the example");

}

Is equal to:

var example2;

example2();

example2 = function() { => will return an errors.

console.log("Ran the example");

}

* When declaring a function as a function:

example2();

function example2() {

console.log("Ran the example");

}

Is equal to:

var example2;

example2 = function() {

console.log("Ran the example");

}

example2();

## Global Scope Issue

All the variables declared inside js files are considered to be **part of the global scope**. This causes a lot of problems with variables in different files overwriting each other and resulting in a lot of weirdness….

In order to work around this, it is common to wrap each file with an anonymous function that execute immediately:

**(function () {**

// all the file’s contents…

**})();**

This prevents the variables defined in this file (i.e. function) from leaking into the global scope.

When using JQuery, there is a better solution using the $(document).ready function call that both supply the function-scope and make sure that the script will only run after the html document is ready. See JQuery for more details.

# Program Flow

## Conditions and Loops

JS supports if-else statements, for, while loops like C++ (syntax).

In addition, it supports:

* **for..of:**

**for**( let user **of** users){

user.doSomething();

}

Creates a loop iterating over **any** [iterable objects](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols#The_iterable_protocol), including: built-in [String](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String), [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array), Array-like objects (e.g., [arguments](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments) or [NodeList](https://developer.mozilla.org/en-US/docs/Web/API/NodeList" \o "NodeList objects are collections of nodes, usually returned by properties such as Node.childNodes and methods such as document.querySelectorAll().)), [TypedArray](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypedArray), [Map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map), [Set](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set), and user-defined iterables. It invokes a custom iteration hook with statements to be executed for the value of each distinct property of the object.

*for(let* ***i*** *= 0 ;* ***i****<****stores****.length ;* ***i****++) {*

…

}

* for..in – only for enumerable, non-Symbol properties like properties of objects:

**for**( let key **in** users){

if(users.hasOwnProperty(key)) {   
 users[key].doSomething();   
}

}

**Note**:

* + for-in **return the keys/indexes** of the object/array and not the data itself!!
  + It is usually necessary to test object.hasOwnProperty(variable) to determine whether the property name is truly a member of the object or was found instead on the prototype chain.
* In order for JavaScript to compare two different data types (for example, numbers and strings), it must convert one type to another. This is known as "Type Coercion".
* **Strict equality** (**===**) is the counterpart to the equality operator (==). However, unlike the equality operator, which attempts to convert both values being compared to a common type, the strict equality operator does not perform a type conversion. Strict inequality (!==) is the opposite of ===.
* **Switch** statements is like in C. It uses the strict equality to compare. So if(val===value) is qual to switch(val){ case value:….}
* **if-else** – the same as in C++
* **while** , **for** , **do-while** loops – the same as in C++.
* **Conditional operator (ternary)** – the same as in C++.

## Evaluators

***Never* use ==. It's a frequent source of bugs.**

**Strict equality (**===**) vs Loose equality (**==**)**

When you use three equal signs, ===, no type conversion is done prior to the comparison. If the values are different types, for example, a String and a Number, they can't ever be equal. To return true, the values must be equal and the types must be the same.

Loose equality, ==, checks to see if the two values are the same type and if not, converts to a common type before the conversion. If the types are already the same, there is no difference between the result of === and ==. When they aren't it can cause unexpected results.

Check the [link](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_when_to_use_them) to an article on Mozilla Developer Network to see what values get converted into what.

According to Jacques Favreau, the lead front-end engineer at Udacity, you should *never* use ==. It's a frequent source of bugs. In fact, if a Udacity engineer tries to commit code with ==, it automatically gets rejected.

Though it wasn't mentioned in the video, **the same conditions apply for strict inequality (!==) and loose inequality (!=)**. Loose inequality is more forgiving than loose equality so you might not see strict inequality as often.

## Functions

There are two ways to define functions in Java Script:

var <function-name> = function(param 1, param 2){

// code

}

Or:

**let/const <function-name> = (params…) => {}**

Or:

function <function-name>(param1, param2){

// code

}

## Exceptions

### Throw:

throw <exception>;

<exception> can be a string, a number or any other expression:

throw 'Error2';   
throw 42;   
throw true; // generates an exception with the value  
 true  
throw new Error('Required'); // generates an error object with the message of Required

### Try-Catch:

try {  
 …  
} catch (<variable name that will receive the exception>) {  
…. handle the exception  
}

# Input/Output

* **console.log**("Hello World");
  + console.log(var1, var2, var3);
* **console.table(<array/JSON object>)** – for pretty table output to console.
* **alert**(“Hello World”); - will display a message box in the browser
* var name = **prompt**(“what is your name?”); - will display the message box to get the data and initialize the variable with it.

# Asynchronous Programing

## Callbacks

The simplest way to implement asynchronous programing in JS is with callbacks, like with setTimer function.

## Promise

<https://scotch.io/tutorials/javascript-promises-for-dummies>

A [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) is **an object representing the eventual completion or failure of an asynchronous operation**. Since most people are consumers of already-created promises, this guide will explain consumption of returned promises before explaining how to create them.

Essentially, a promise is a returned **object to which you attach callbacks, instead of passing callbacks into a function.**

Imagine a function, createAudioFileAsync(), which asynchronously generates a sound file given a configuration record and two callback functions, one called if the audio file is successfully created, and the other called if an error occurs.

Here's some code that uses createAudioFileAsync():

function successCallback(result) {

console.log("Audio file ready at URL: " + result);

}

function failureCallback(error) {

console.log("Error generating audio file: " + error);

}

createAudioFileAsync(audioSettings, successCallback, failureCallback);

…modern functions return a promise you can attach your callbacks to instead:

If createAudioFileAsync() were rewritten to return a promise, using it could be as simple as this:

**createAudioFileAsync(audioSettings).then(  
 successCallback, failureCallback);**

That's shorthand for:

const promise = createAudioFileAsync(audioSettings);

promise.then(successCallback, failureCallback);

Unlike "old-style", passed-in callbacks, a promise comes with some guarantees:

* Callbacks will never be called before the [completion of the current run](https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop#Run-to-completion) of the JavaScript event loop.
* Callbacks added with [then()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then) even after the success or failure of the asynchronous operation, will be called, as above.
* Multiple callbacks may be added by calling [then()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then) several times. Each callback is executed one after another, in the order in which they were inserted.

### Promises

Promises has three possible states:

* Pending – the asynchronous action was call but hasn’t finished yet
* Fulfilled – the asynchronous action was successful
* Rejected – the asynchronous action failed

### Promises Chaining

A common need is to execute two or more asynchronous operations back to back, where each subsequent operation starts when the previous operation succeeds, with the result from the previous step. We accomplish this by creating a **promise chain**.

Here's the magic: the then() function returns a **new promise**, different from the original:

const promise = doSomething();

const promise2 = promise.then(successCallback, failureCallback);

or

const promise2 = doSomething().then(successCallback, failureCallback);

This second promise (promise2) represents the completion not just of doSomething(), but also of the successCallback or failureCallback you passed in, which can be other asynchronous functions returning a promise. When that's the case, any callbacks added to promise2 get queued behind the promise returned by either successCallback or failureCallback.

Basically, each promise represents the completion of another asynchronous step in the chain.

Full example:

doSomething()  
.then(result => doSomethingElse(result))  
.then(newResult => doThirdThing(newResult))  
.then(finalResult => {  
 console.log(`Got the final result: ${finalResult}`);  
})  
.catch(failureCallback);

The arguments to then are optional, and catch(failureCallback) is short for then(null, failureCallback)

### Example

/\_ ES6 \_/

const isMomHappy = true;

// Promise

const willIGetNewPhone = new Promise(

(resolve, reject) => {

if (isMomHappy) {

const phone = {

brand: 'Samsung',

color: 'black'

};

resolve(phone);

} else {

const reason = new Error('mom is not happy');

reject(reason);

}

}

);  
// Promise 2 - since it only has Resolve - we can shorten it to   
// contain only the data for the Promise.resolve call

const showOff = function (phone) {

const message = 'Hey friend, I have a new ' +

phone.color + ' ' + phone.brand + ' phone';

return Promise.resolve(message);

};

// call our promise

const askMom = function () {

willIGetNewPhone

.then(showOff)

.then(fulfilled => console.log(fulfilled)) // fat arrow

.catch(error => console.log(error.message)); // fat arrow

};

askMom();

# AJAX – Asynchronous JavaScript & XML

* Technology for making asynchronous requests to a server to transfer data, then load any returned data into the page. An asynchronous process has a couple key properties.
* The browser does not stop loading a page to wait for the server's response. Also, the browser inserts updated data into part of the page without having to refresh the entire page.
* Note: the page rendering can’t happen in parallel to the V8 call stack running. Therefore, like callback functions, it has to wait until the stack it empty from it can run! Since the ideal page-rendering speed is every 60 seconds

User experience benefits from asynchronous processes in several ways:

* Pages load faster since the browser isn't waiting for the server to respond in the middle of a page render.
* Requests and transfers happen in the background, without interrupting what the user is doing.
* When the browser receives new data, only the necessary area of the page refreshes.

These qualities especially enhance the user experience for single page applications.  
  
The data transferred between the browser and server is often in a format called JavaScript Object Notation (JSON). JSON resembles JavaScript object literal syntax, except that it's transferred as a string. Once received, it can be converted into an object and used in a script.

## Connect to DOM

* To make sure you’re your code executes only after the page has finished loading, you can attached a JavaScript ‘DOMContentLoaded’ event to the document:

document.addEventListener('DOMContentLoaded',  
 function() {  
 ….   
// You can register additional event handlers here. For example: document.getElementById(‘getMessage’).onclick=function() {…};  
});

## Change Element Content

document.getElementsByClassName('message')[0].textContent="Here is the message";

## XMLHttpRequest

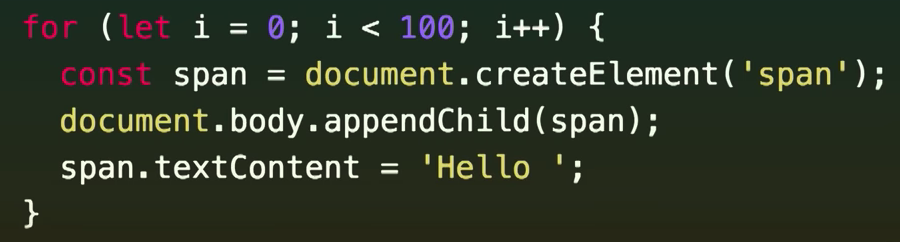
You can also request data from an external source. This is where APIs come into play.

Most web APIs transfer data in a format called JSON. JSON stands for JavaScript Object Notation.

However, JSON transmitted by APIs are sent as bytes, and your application receives it as a string. These can be converted into JavaScript objects, but they are not JavaScript objects by default. The **JSON.parse** method parses the string and constructs the JavaScript object described by it.

## Event Listeners

Note: every time you change the DOM, a change event is triggered. It doesn’t just trigger on rendering but on every line that changes the DOM. So if you’re listening to changes, you might get more events than you expect.   
For example:



Will trigger 200 events (on the appendChild and on the spac.textContent)

**Mutation Observers** came to solve the issue of listening on specific events and getting triggered for every single event. Instead, they will be called just once for all these events. They solve it by using a new **microtasks queue**

# JSON

## Get and manipulate elements on a page:

In the HTML document:

<script type="text/javascript">

….

document.getElementsByClassName(<class name>)[0]

and then to manipulate it’s properties:

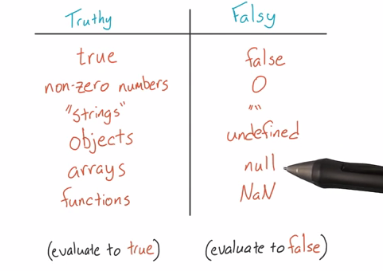
document.getElementsByClassName(<class name>)[0].style.<set any CSS property as you would in CSS file>

</script>

**‘document’** here is the DOM (the web page at its current state)

NOTE: Some property names are different between CSS and JS!! See Table in CSS: <https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Properties_Reference>

**True and False values in JS:**



JSON is a popular and simple format for storing and transferring nested or hierarchal data. It's so popular that most other programming languages have libraries capable of parsing and writing JSON (like Python's [JSON library](https://docs.python.org/2/library/json.html)).

Internet GET and POST requests frequently pass data in JSON format.

JSON allows for objects (or data of other types) to be easily encapsulated within other objects. See the [MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON) or [JSON.org](http://json.org/) for more details.

[This](http://www.copterlabs.com/blog/json-what-it-is-how-it-works-how-to-use-it/) is a fantastic deep dive from Jason Lengstorf about JSON and its ubiquitous use in the form of [AJAX requests](http://en.wikipedia.org/wiki/Ajax_%28programming%29).

## Always Lint your JSON!

With a mix of nested curly braces, square brackets and commas, it's easy to make mistakes with JSON. And mistakes mean bugs.

If you're generating JSON by hand, you should copy and paste your code into a JSON linter like [jsonlint.com](http://jsonlint.com) to quickly and easily find syntax errors. A linter is a piece of software that analyzes code for syntax errors. Some text editors, like Sublime Text, will automatically lint (or highlight) most syntax errors. But a JSON linter won't miss any syntax errors and you can rest assured that your JSONs will be properly formatted.

# TypeScript

A superset of JavaScript that add new features and type-safety to JavaScript.

Typescript is compiled (transcripted) into plain JavaScript before running in the browser.

Note: the type-checking is done only in compile time. Once the typescript code becomes plain javaScript – no type checking is performed (i.e. no type checking in run time).

Typescript files have the extension .ts

## Install

npm install -g typescript

## Compile to JS

tsc helloworld.ts

## Create New React-Redux-Typescript Project

create-react-app my-app --scripts-version=react-scripts-ts

* tsconfig.json contains TypeScript-specific options for our project.
  + We also have a tsconfig.prod.json and a tsconfig.test.json in case we want to make any tweaks to our production builds, or our test builds.
* tslint.json stores the settings that our linter, [TSLint](https://github.com/palantir/tslint), will use.
* package.json contains our dependencies, as well as some shortcuts for commands we'd like to run for testing, previewing, and deploying our app.
* public contains static assets like the HTML page we're planning to deploy to, or images. You can delete any file in this folder apart from index.html.
* src contains our TypeScript and CSS code. index.tsx is the entry-point for our file, and is mandatory.
* images.d.ts will tell TypeScript that certain types of image files can be import-ed, which create-react-app supports.

## Syntax

<https://github.com/Microsoft/TypeScript-React-Starter#typescript-react-starter>

### Type Annotations

Type annotations in TypeScript are lightweight ways to record the intended contract of the function or variable. In this case, we intend the greeter function to be called with a single string parameter:

function greeter(person**: string**) {  
 return "Hello, " + person;  
}

If we’ll call the function with the wrong parameter:

let user = [0, 1, 2];  
document.body.innerHTML = greeter(user);

Compiling the TypeScript code will return an error. **Note:** although the compilation returns an error, the javascript file is still created and can be used. The TypeScript compiler just warns us that our code likely has bugs in it.

### Interfaces

Here we use an interface that describes objects that have a firstName and lastName field. In TypeScript, two types are compatible if their internal structure is compatible. This **allows us to implement an interface just by having the shape the interface requires, without an explicit implements clause**.

**interface** **Person** {  
 firstName: string;  
 lastName: string;  
}

function greeter(person: **Person**) {  
 return "Hello, " + person.firstName + " " + person.lastName;  
}

let user = { firstName: "Jane", lastName: "User" };  
document.body.innerHTML = greeter(user);

### Classes

TypeScript supports new features in JavaScript, like support for class-based object-oriented programming.

##### Public/Private/Protected

Class’ fields/functions can be public/private/protected like in C#:

public visits: number = 0;

private ourName: string;

##### ctor

constructor(value: type){…}

in TypeScript, the constructor can also create fields automatically for us when the parameters include the access indicator (e.g. private/protected/public). For example:

**constructor**(**private** firstName: string,

**private** lastName: string) {}

public showName() {

alert(this.firstName + " " + this.lastName);

}

##### Functions

<access indicator> functionName(param: ofType,…): returnedType {…}

For example:

public showName(name: string): boolean {

alert(name);

return true;

}

##### Accessors (Properties in C#)

Similar to properties in C#:

* Setters:

**set** propertyName( value: Type) {

this.privateField = value;

}

* Getters:

**get** properyName() {

**return** this.privateProperty;

}

Here we’re going to create a Student class with a constructor and a few public fields. Notice that classes and interfaces play well together, letting the programmer decide on the right level of abstraction.

Also of note, the use of *public on arguments to the constructor is a shorthand that allows us to automatically create properties with that name*.

class Student {  
 public **fullName: string;**  
  
 constructor(**public** firstName**: string**, public middleInitial: string, public lastName: string) {

**this.fullName** = firstName + " " + middleInitial + " " + lastName;

}  
}

interface Person {  
 firstName: string;  
 lastName: string;  
}

function greeter(person : Person) {  
 return "Hello, " + person.firstName + " " + person.lastName;  
}

let user = new Student("Jane", "M.", "User");

document.body.innerHTML = greeter(user);

### Extend React.Component

export **interface Props** {  
 name: string;  
 enthusiasmLevel?: number;  
}

class Hello extends React.Component**<Props, object>** {

render() {  
 const { name, enthusiasmLevel = 1 } = this.props;  
 if (enthusiasmLevel <= 0) {

throw new Error('You could be a little more enthusiastic. :D');

}

return (  
 <div className="hello">  
 <div className="greeting">  
 Hello {name + getExclamationMarks(enthusiasmLevel)}  
 </div>  
 </div>

);  
 }  
}

Notice that we defined a type named Props that specifies the properties our component will take. name is a required string, and enthusiasmLevel is an optional number (which you can tell from the ? that we wrote out after its name).

the class extends React.Component<Props, object>. The TypeScript-specific bit here are the type arguments we're passing to React.Component: Props and object. Here, Props is the type of our class's this.props, and object is the type of this.state.

#### Type Assertions

ReactDOM.render(  
 <Hello name="TypeScript" enthusiasmLevel={10} />,  
 document.getElementById('root') **as** HTMLElement  
);

This syntax is called a *type assertion*, sometimes also called a *cast*. This is a useful **way of telling TypeScript what the real type of an expression is** when you know better than the type checker.

The reason we need to do so in this case is that getElementById's return type is HTMLElement | null. Put simply, getElementById returns null when it can't find an element with a given id. We're assuming that getElementById will actually succeed, so we need to convince TypeScript of that using the as syntax.

TypeScript also has a trailing "bang" syntax (!), which removes null and undefined from the prior expression. So we *could* have written document.getElementById('root')**!**, but in this case we wanted to be a bit more explicit.

## Testing

### Enzyme

[Enzyme](http://airbnb.io/enzyme/) is a common tool in the React ecosystem that makes it easier to write tests for how components will behave. By default, our application includes a library called jsdom to allow us to simulate the DOM and test its runtime behavior without a browser. Enzyme is similar, but builds on jsdom and makes it easier to make certain queries about our components.

Let's install it as a development-time dependency:

npm install -D enzyme @types/enzyme enzyme-adapter-react-16 @types/enzyme-adapter-react-16 react-test-renderer

# Libraries

To know which library is supported for which browser, version etc – see <http://www.caniuse.com>

## Useful Libraries

* seamless-immutalbe – a library for creating and working with immutable objects. Very useful when working with redux.

To install: npm install seamless-immutable

* lodash – a library for iterating over and manipulating arrays, objects and strings. Contains a lot of syntactic sugar for data manipulations. Nicknamed Linq for JS.

Examples:

import \* as \_ from "lodash";

…

get subtotal() : number {

return **\_.sum**(this.items, i =>

(i.unitPrice \* i.quantity));

}

## JQuery

A very popular library in java script that allow easy manipulation of web pages.

Check out the [jQuery website](http://jquery.com/) to learn more

JQuery methods: <http://www.w3schools.com/jquery/jquery_ref_html.asp>

**Directly work on elements in a DOM:**

Using JQuery, we can directly access and manipulate DOM elements using CSS syntax to refer to elements. For example:

* $(“#element-id”)
* $(“.element-class”)

# Tools

Json2ts – generate TS types declarations from json data.

## Bundling

Webpack, browserfy – boundle all the javascript files into one, compressed file.

## Compilers

Babel, jscodeshift – compile code to make it runnable (e.g. from typescript or ES6)

## Toolkits

Can create initial custom-project/code. For example, the create-react-app

# NodeJs

# Testing

## Unit Testing

### Mocha

// "use strict";

var assert = require('assert');

// require("js\_algorithms.js");

describe('Array',function() {

describe('#indexOf()', function() {

it('should return -1 when the value is not present', function() {

assert.equal(-1, [1,2,3].indexOf(4));

});

});

});

// Create a test suite (group) called Math

describe('Math', function() {  
  
 // add a test hook  
 beforeEach(function() {  
 // ...some logic before each test is run  
 })

// Test One: A string explanation of what we're testing

it('should test if 3\*3 = 9', function(){

// Our actual test: 3\*3 SHOULD EQUAL 9

assert.equal(9, 3\*3);

});

// Test Two: A string explanation of what we're testing

it('should test if (3-4)\*8 = -8', function(){

// Our actual test: (3-4)\*8 SHOULD EQUAL -8

assert.equal(-8, (3-4)\*8);

});

});

##### Install:

$ npm install // will install all the required dependencies according to the package.json file.

##### Set Up:

In the package.json add:

"scripts": {

"test": "mocha"

}

##### Structure

1. **describe()** is simply a way to group our tests in Mocha. We can nest our tests in groups as deep as we deem necessary. describe() takes two arguments, the first is the name of the test group, and the second is a callback function.

describe('string name', function(){  
 // can nest more describe()'s here, or tests go here  
});

* **it()** is used for an individual test case. it() should be written as if you were saying it out loud: “It should equal zero”, “It should log the user in”, etc. it() takes two arguments, a string explaining what the test should do, and a callback function which contains our actual test:

it('should blah blah blah', function(){  
 // Test case goes here  
});

##### Assertion Libraries (e.g. Chai):

Within our testing framework (Mocha), we can use assertion libraries. **An assertion library is a tool to verify things are correct -** It’s what actually verifies the test results.

Note that we don’t need to use an assertion library, but they make testing **way** easier. Mocha allows us to use any assertion library we wish. In the above example (and for all of the other examples), we’re using Node.js’ built-in [assert](https://nodejs.org/api/assert.html) module. Hence this line of code where we require the assert module:

##### Running

Form the command-line:

$ npm test

To run a single test, from the root of the project:

$ mocha -g <it/describe name as a regular expression>

Note: if you changed the names, you might need to run the npm test first to compile the whole names list.

Example:

it('logs a', function(done) {  
 console.log('a');  
 done();  
});  
…

$ mocha -g 'logs a'

##### Debugging

* 1. run the test(s) with inspect-brk flag: $ mocha --inspect-brk
  2. in VS-Code:
     1. choose the “Debug Mocha Test” debugging configuration:  
        in launch.json in your project’s root directory:

{

"type": "node",

"request": "launch",

"name": "Debug Mocha Tests",

"address": "localhost",

"port": 9229,

"sourceMaps": false

}

* + 1. Set break point(s) in your test
    2. Press the green run button

## Chai

Chai is a BDD / TDD assertion library for [node](http://nodejs.org) and the browser that can be delightfully paired with any javascript testing framework.

### Installation

##### Node.js

1. $ npm install chai
2. See js\_sandbox/package.json for additional settings

##### Browser

Include the chai browser build in your testing suite.

<script src="chai.js" type="text/javascript"></script>

This will provide chai as a global object, or define it if you are using AMD.

### Test

## End-To-End Testing

### Cypress

##### Install:

1. Navigate to your project’s root directory
2. Install cypress locally as a dev dependence for your project:  
   $ npm install cypress --save-dev

# How to debug

* Get to the page/scenario you want to test
* Right click an element->inspect Element
* Copy one of the element’s properties and search for them in the Debugger tab (in all files)
* This is where the code should go on choosing the address. If it doesn’t there is a problem with wiring of the function to the button.
* You can also try looking at the EventListeners
* $0. – the selected element in the page.
* When the script runs in the browser, we have access to its functions and we can call them. When you breakpoint on a line, you can see all its internal variables and functions and you can call them from the console.

A.on("a:button-group", function () { console.log(arguments()) })

A.on("a:button-group:toggle", function () { console.log(arguments()) })

var el = document.createElement("span")

el.className = "class1 class2"

el.setAttribute("data-a-button-group-name", '{name: "asdf"}')

el.outerHTML

## Chrome Debugging

* In Dev-tools: Cmd-Shift-P – opens a commands panel that you can types a lot of commands.
* If you select an element in the Elements tab, you can get the JS reference to it through $0 so you can inspect or manipulate it.
* Breakpoints:
  + XHR/fetch – will break when the browser send a request to URL. You can add a URL that contains some partial path and it will break there.
  + On the DOM elements themselves, you can set breakpoints when the DOM is changing (in the elements tab -> right click)
* Search:
  + In the Console: Cmd-opt-F (Ctrl-shift-F) – search for string in any file (js/html etc)
* Copy full network request in any format you want (like curl request): choose request in network tab, right click…
* Console:
  + Cmd+L – clear console
* You can use snippets to write partial code you can run in the debugger (need to figure out how – a bit clunky..)

# Packaging

## Npm

**npm** is a [package manager](https://en.wikipedia.org/wiki/Package_manager) for the [JavaScript](https://en.wikipedia.org/wiki/JavaScript) programming language. It is the default package manager for the JavaScript runtime environment [Node.js](https://en.wikipedia.org/wiki/Node.js). It consists of a command line client, also called npm, and an [online database](https://en.wikipedia.org/wiki/Online_database) of public and paid-for private packages, called the npm registry.

### Init

npm init -y

will initialize the project and will create the initial package.json file.

### Install Dependencies

npm install

will check the package.json and will install all the required dependencies.

### Install Package

npm install [-g] <package name>

Can also use: npm i…

-g – install the package globally. Otherwise, will install it on the current folder only.

--save-dev – will save as a development-dependency in package.json

### Useful Commands

* The latest available version on the npm registry:

npm view <package-name> version

* The versions of all local [or -g for global] installed packages:

npm list [-g]

## Yarn

Another package manager

## Package.json

* “dependencies”: {  
   all the dependencies that will be used by all versions of the app (both development and deployed).  
  }
* “devDependencies” : {  
   dependencies for development-environment only.  
  }