# Types

**Statically typed** means that types are enforced and then compiled to .js as in Typescript

**Example:** int x = 5

**Dynamically Typed** means that the compiler infers the types of your variables at *runtime* like in Javascript. Types are still enforced…it’s just the that compiler decides what it is.

**Example:** var a = 1 // compiler infers that a is an int

**Weakly Typed** allows languages to be inferred as another type. Think about in JS how 1 can be either a number or a string.

# Primitives

**Primitive data** types in JS: Strings, Number, Boolean, Undefined, bigint, symbol

**Structural data** types in JS: Objects, functions, null

All primitives are immutable.

A variable assigned a primitive value is not itself a primitive.

Interestingly, primitives have NO methods. ‘abc’.length may APPEAR to have a method but in reality when .ltngth is called, a constructor makes an object out of the primitive then gets garbage collected (removed from memory).

# Double Equality vs Triple Equality

Double equals (==) only compares values.

Triple equals (===) compares both values and data types. More strict. For non primitives, triple equals compare whether the two things are at the same location in memory.

# Hoisting

Where all variables and function declarations (but not assignment/initialization!!!) move to the top.

# Event Delegation

Instead of adding addEventListener to every element, just add it once to the parent element and use an if/match statement. This is better for performance.

If you don’t specify for an addEventListener to “capture”, it will only bubble. Capture happens before bubble. Capture happens from the most outer element down to the lowest element.

You can stop propagation of capture/bubbling by using e.stopPropagation();

# Scoping for var, let, const

For strict mode: var, let, const are locally scoped when in a block:

Let x = 10;

{

X = 5;

}

Console.log(x); // Will print 10

# Prototypical Inheritance

When we read a property from an object and it’s missing, Javascript automatically takes it from the prototype. That is prototypical inheritance.

let animal = {

eats: true

walk() {

alert(“Animal is walking”);

}

};

let rabbit = {

jumps: true

\_\_proto\_\_: animal

};

// we can find both properties in rabbit now:

alert( rabbit.eats ); // true (\*\*)

alert( rabbit.jumps ); // true  
rabit.walk();

Prototypical inheritance can go on for on for many levels.

An object can only have one prototype at a time, nothing more.

\_\_proto\_\_ is actually a getter/setter for [[Prototype]]

This is not affected by prototypes. That’s why we can use a setter on the object without affecting the prototype.

# Inheritance in JavaScript (ES6)

Inheritance in JavaScript (ES6) is very similar to inheritance in Java.

Uses the ‘extends’ keyword to inherit from the super class.

Subclass can (like Java) override a super class method by declaring its own.

# Polymorphism in JavaScript

This principle is more applicable to strongly typed languages like Java. Not too relevant to JavaScript as it is a weakly typed object.

# Asynchronous vs Synchronous Programming

Synchronous code starts from the top of the file and executes downwards in order.

Asynchronous code also starts from the top of the file and executes downwards – until it hits an asynchronous piece of code that needs to split off and run on its own independently – or asynchronously (hence the name). We use asynchronous code because that part of the program needs to wait or do some operation that takes a long time (like making API calls, pushing to database, getting from server.

# Threads

The **main thread** is the one used by the browser to handle user events, rendering, and to run the majority of the code of a typical web page/web app.

**Worker threads** can be spun off to run sub-programs concurrent to the main thread. Worker threads can not touch the DOM or the UI (that is handled by the main thread). Example: Async function.

# Encapsulation

Encapsulation hides an object’s data from the outside world. This is done by wrapping an object and its method inside a function. The methods and data can still talk to each other because all data and method are within the function closure.

# Array.forEach() vs Map()

The main difference between forEach and Map() is that Map() returns a result without actually modifying the input whereas forEach() actually modifies the input.

**Want to make a program that utilize:**

Inheritance and objects in ES6. Async and promise. Event Delegation. Lots of html/css.

**HTTP**

Hypertext Transfer Protocol. Verbs: GET, POST, PUT, DELETE.

How frontend communicate to web server which communicate with a database or a API server.

**SSL**

Web server sends a certificate to authenticate itself to the web client