# 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 .length 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.

# Object-Oriented Programming

1. Abstraction: hide away implementation details, only need input/output. Hides entire class. Model of more complex idea. Person is super class. Teacher is a subclass.
2. Encapsulation: hide method in single entity to protect from outside. Hides methods.
3. Inheritance: prototypical/classical inheritance
4. Polymorphism: functions with same name can have different functions

# Abstraction

Hiding implementation detail from the users and other parts of the program at the class level. All you need to know is the input/output so that it can interact with other parts of the program. Model the real world. Benefits: modularity, replaceable, won’t break the system.

# Encapsulation

Encapsulation hides/protects an object’s data from the outside world. This is done by wrapping an object and its method inside a function so that the program can’t access the object data and methods directly. The program would need to use the getter and setter methods to change an object’s variables. By convention, you can add \_ before a variable to let other programmers know that this variable is meant to be private. The methods and data can still talk to each other because all data and method are within the function closure. Hide the implementation details.

# 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.

Polymorphism means that in Java, you can have multiple functions with the SAME name as long as the number of arguments required is different or the data type of the argument is different.

Not as relevant in JS. Polymorphism is when you let a subclass define its own implementation of an otherwise inherited property. This is usually done when you have a subclass and you want to change how one of the method is done from the super class

# 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.

# 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.

**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

# [Functional Programming](https://www.youtube.com/watch?v=LnX3B9oaKzw&list=WL&index=12&t=16s&ab_channel=Computerphile)

Essentially programming where you **don’t have side effects** and the **inputs are not modified** by the function. The benefits of functional programming is that it avoids introducing bugs.

Another way of saying it: Functional languages try to isolate the transformations of data from the definition of the data.

# Pure Functions

Producing a program where your functions:

1. Avoids side effects (eg. performing a function won’t modify a global variable)
2. Given an input, you will always get the same output. It does not modify the inputs. (eg. If the parameters/inputs are x and y, then z will always be returned)

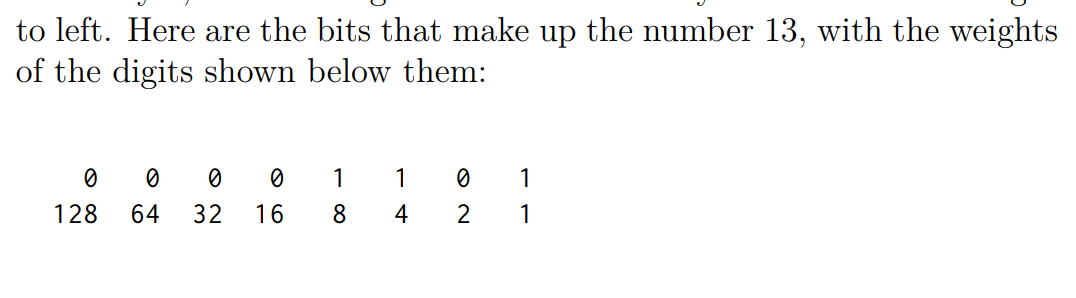
Example of a pure function, note that after the function is executed, the inputs are not modified.

**Function addElementToArray (element, array)**

**Return […array].push(element)**

# Easy Way to Convert Digits to Binary in Your Head

If you want to know 13 (which is a digit) in binary, just add up the hexadecimals (second row of image). Notice that 8 + 4 + 1 = 13.



# Software Development Life Cycle

Purpose is to produce highest quality software w/ lowest cost possible in shortest time possible. SDLC used in water and agile.

Reduces cost, cuts time, and produces quality code by getting client's requirement sign off.

1. Requirement’s analysis (Identify the problems from all stakeholders.)

2. Planning (Determines feasibility of resources required, cost, and risks for implementing requirements)

3. Software design (Turning requirements into design plan. Get signoff from stakeholders on the plan.)

4. Software development (Build what we want)

5. Testing (Test for defects and deficiencies, verify code meets requirements)

6. Deployment (Deploy software to different environments like UAT and PROD)

Agile is when we do the SDLC for each product. Focus on rapid deployment and use the results to feed into the next cycle.

# React Component Life Cycle

componentWillMount() (Now deprecated): Executed right before component render. At this time, there may not even be any component on the DOM. This method gives flexibility for users to execute before render.

componentDidMount(): Right when component did mount on DOM/shows up on screen. Super common method. As soon as a DOM is loaded, do something.

componentWillReceiveProps(): Run every time a component receives new props.

shouldComponentUpdate(): Tell React if Component should or should not re-render. If Component re-renders everytime, it is a tax on the system.

componentDidUpdate(): Executed when a Component has been updated on the DOM.

componentWillUnmount(): Called right before component unmount from DOM.

## REST API

|  |  |
| --- | --- |
| **SOAP** | **REST** |
| XML format | XML or JSON format |
| Transfer with HTTP, SFTP, FTP, and more | Transfer with HTTP |
| Machine Readable | Human Readable |
| Harder to make calls with JavaScript | Easy to make calls with JavaScript |

REST API is API that conforms to the rules of REST. REST is stateless. This means that the server/instances do NOT store any state about the client session on the server-side. Stateful means that the server stores states (such as login status state) of the user. The request from the client to server must contain all the information (such as authorization token) and cannot take advantage of any stored context on server. The server never relies on information from previous request.

Benefits of stateless is that scaling is easier: When a client makes a call, a load balancer directs the request to an available instance/server. It would be bad if every server needed to hold information about the state of the client.

# Closure

Amazing explanation: https://www.youtube.com/watch?v=XTAzsODSCsM

The main benefit of Closure is creating persistent storage – without using global memory.

When an inner function is inside of an outer function. The inner function has access to variables in the outer function including all variables – this is called the environment. However, the most important point is that, the inner function has access to the variables of the outer function even **after** the outer function has already executed.

Function outerFunction(url) {

Function getJSON (Fetch(url).then(() => {

Console.log(url)

})

Return getJSON;

}

Const newFunction = outerFunction(‘outside)

newFunction(‘inner)

---