JavaScript: The Advanced Concepts

# Cheat Sheet

<https://zerotomastery.io/cheatsheets/javascript-cheatsheet-the-advanced-concepts>

# Foundation

## JavaScript Engine

It helps to compile the JavaScript so that the machine/browser can understand.

E.g:- V8, Chakra etc.

**Inside the Engine**

|  |
| --- |
|  |

Interpreters and compilers

Babel + TypeScript

Have you heard of Babel or TypeScript? They are heavily used in the JavaScript ecosystem and you should now have a good idea of what they are:  
  
[Babel](https://babeljs.io/) is a JavaScript compiler that takes your modern JS code and returns  browser compatible JS (older JS code).  
[Typescript](https://www.typescriptlang.org/) is a superset of JavaScript that compiles down to JavaScript.  
  
Both of these do exactly what compilers do: Take one language and convert into a different one!

## JIT Compiler

It is the combination of interpreter and compiler.

## Stack Overflow

When the Call stack size exceeded it leads to stack flow. Just try with recursion function.

## Call Stack and Memory Heap

**Memory Heap**: - Where the memory allocation happens. Whenever we are declaring variable, array, objects memory is allocated to memory heap.

**Call Stack**: - Where the JS engine keeps tracks of where the code is in its execution. It stores function, values as when the program execute

## Garbage Collection

JavaScript is garbage collective language means it automatically clean up the memory which no longer needs.

Garbage collection uses the **marks and sweep** algorithm.

## Memory Leaks

**Memory Leaks** happens when we have unused memory just laying around in Memory Heap it fills up the memory.

Memory Leaks can happens due to:-

1. Global variable.
2. Event listeners
3. Using setInterval()

<https://developers.soundcloud.com/blog/garbage-collection-in-redux-applications>

## Single Threaded

JavaScript is single threaded programming language because it has one call stack because of this JavaScript is synchronous.

## JavaScript Runtime

**Call Stack** its runs all the JavaScript code one by one and if any **asynchronous** method (like modify DOM, calling http function etc.) finds it sends to the **web API** then execute the function and return to **Callback Queue** which then pushed to call stack by **Event loop** when the call stack is empty**.**

|  |
| --- |
|  |

<http://latentflip.com/loupe/?code=ZnVuY3Rpb24gcHJpbnRIZWxsbygpIHsNCiAgICBjb25zb2xlLmxvZygnSGVsbG8gZnJvbSBiYXonKTsNCn0NCg0KZnVuY3Rpb24gYmF6KCkgew0KICAgIHNldFRpbWVvdXQocHJpbnRIZWxsbywgMzAwMCk7DQp9DQoNCmZ1bmN0aW9uIGJhcigpIHsNCiAgICBiYXooKTsNCn0NCg0KZnVuY3Rpb24gZm9vKCkgew0KICAgIGJhcigpOw0KfQ0KDQpmb28oKTs%3D!!!PGJ1dHRvbj5DbGljayBtZSE8L2J1dHRvbj4%3D>

e.g :-

console.log("1");

setTimeout(() => console.log("2"),0);

console.log("3");

It will print:- 1 3 2

## Node.js

Node.js uses V8 engine to run JavaScript outside of the browser. Node.js is most popular for building servers, it’s fast. It also good for chat application.

**Node.js also uses single threaded process but any asynchronous method can be non-blocking that it can be passed on to the worker thread.**

# Foundation 2

## Execution Context

Anytime we run code in JavaScript Engine a global execution context **(global object/window and this)** is created and when we run a function a new execution context is added until all the code pops out of the Call Stack.

|  |
| --- |
|  |

Execution context tells us which lexical environment currently running.

## Lexical Environment

Lexical Environment where code/function is written, other word in compile time.

## Hoisting

It is the JavaScript Engine allocating memory for the variable and function that is sees in the code during the creation phase before it executes.

Var and function used as hoisted.

* Function declarations are scanned and made available.
* Variable declarations are scanned and made undefined.

**We should avoid hoisting, it can be done using let/const instead of var.**

|  |
| --- |
|  |

## Function Invocation

When we call/invoke/execute a function (hello()), as and when JavaScript Engine see the invoke () it will start the execution context of that function.

**Function expression**

|  |
| --- |
| var Hello = () => {  console.log("Hello")  } |

Function expression is defined at runtime means when we execute/invoke the function.

**Function Declaration**

|  |
| --- |
| function Hello (){  console.log("Hello")  } |

Function declaration get defined at parse time when we invoke/execute the function.

## arguments keyword

It is present inside the execution context when we run a function, it a predefined keyword. **With modern JavaScript we should not use the arguments keyword**

## Variable Environment

It is memory available in each execution context to store the local variable of that function. It is storing in call stack or it could be a reference to memory heap

|  |
| --- |
|  |

## Scope Chain

### Global Lexical Environment

It is written in the global space means it attached to the window object

All the function has their own variable environment, but they also access to parent environment/global environment via scope chain.

|  |
| --- |
|  |

### Function Lexical Environment

|  |
| --- |
|  |

|  |
| --- |
|  |

[[Scopes]]

**Declaring a variable without let, var, const inside function will treated to store the variable in global scope. By using ‘use strict’ it will throw error.**

|  |
| --- |
|  |

## Block Scope vs function scope

JavaScript always use the function scope but it other programming language it uses the block scope.

**But in ES6 JavaScript let and const keyword which restrict block scope variable to use outside.**

## Global Variables

Issue with global variables it leads to variable collision

## IIFE (Immediately invoke function expression)

IIFE is a common JavaScript design pattern use by a lot of popular library like jquery, backbone.js

An IIFE (Immediately Invoked Function Expression) is a JavaScript function that runs as soon as it is defined.

|  |
| --- |
| (function () {  statements  })(); |

## this keyword

It has different values depending on where it is used:

* In a method, this refers to the **owner object**.
* Alone, this refers to the **global object**.
* In a function, this refers to the **global object**.
* In a function, in strict mode, this is **undefined**.
* In an event, this refers to the **element** that received the event.
* Methods like call(), and apply() can refer this to **any object**.

this is an object that the function is property of. Benefits: -

1. It gives methods access to their objects

|  |
| --- |
| const obj = {  name: 'Billy',  sing: function() {  return 'llala ' + this.name + '!'  },  singAgain: function() {  return this.sing()  }  } |

1. Execute the same code for multiple object

|  |
| --- |
| function importantPerson() {  console.log(this.name)  }  const name = 'Sunny';  const obj1 = { name: 'Cassy', importantPerson: importantPerson}  const obj2 = { name: 'Jacob', importantPerson: importantPerson}  obj2.importantPerson() |

**Always use arrow function inside another to use this, otherwise it will be treated as window object.**

|  |
| --- |
| //JS is weird:  const obj = {  name: 'Billy',  sing: function() {  console.log(this) // in this case, it's a method on an object.  var anotherFunc = function() {  console.log(this)// this points to windows! // change to arrow function  }  }  } |

### call()

When we use () to invoke any function, all function when created have the property called **call()** that allow us to call the function.

Instead function.call() we use the short hand function()

### apply()

Same as call() it also do the same.

Also **we can use call() and apply() to borrow methods from an object**. Also parameter for apply() to pass as an array.

|  |
| --- |
| const Wizard = {  name:"Jack",  health:50,  heal(num1,num2){  return this.health += num1 + num2 ;  }  }  const Archer = {  name:'John',  health:30  }  console.log('1',Archer);  Wizard.heal.call(Archer,2,4);  Wizard.heal.apply(Archer,[2,4]);  console.log('2',Archer); |

### bind()

Unlike call() and apply() immediately runs a function , bind() returns a new function with a certain context and parameter. Bind allow us to this keyword or this function browning for later user

**bind() is use to call a function later on with a certain context**

|  |
| --- |
| const wizard = {  name: 'Merlin',  health: 100,  heal: function(num1, num2) {  this.health += num1 + num2;  }  }  const archer = {  name: 'Robin Hood',  health: 50  }  wizard.heal.call(archer, 50, 60)  wizard.heal.apply(archer, [20, 30])  archer  // function borrowing  const healArcher = wizard.heal.bind(archer, 50, 60);  console.log(archer)  healArcher()  console.log(archer) |

#### currying using bind

Currying basically means instead of passing multiple parameter in a function we can pass one at a time

|  |
| --- |
| function multiply(a, b) {  return a\*b;  }  var multipleByTwo = multiply.bind(this, 2);  console.log(multipleByTwo(4));  var multipleByThree = multiply.bind(this, 3);  console.log(multipleByThree(4)); |

## Scope vs Context

Scope means accessing variable inside a function.

Context is more about object (means what the value of this keyword).

# Types in JavaScript

## JavaScript Types

### Primitive Type

It ‘s the data that only represent the single value, immutable in nature.

1. number
2. Boolean
3. String
4. Undefined
5. null
6. symbol

**Undefined** is an absence of the definition, default values when the variables are initialized.

**Null** is an absence of the value. Issue in JavaScript **typeof** null returns object

### Non-Primitive Type

It does not contain the actual value directly, means it will have the reference like pointer somewhere in memory where the object is stored.

1. {}
2. []
3. Function(){}

Array in JavaScript is object.

|  |
| --- |
|  |

**Array.isArray([])** // true

## Pass by Value vs Pass by Reference

Primitive Data Types are Pass by Value.

Non-Primitive Data Types like object are Pass by reference.

|  |
| --- |
|  |

**Cloning of array and object instead of pointing to same memory :-**

1. , **let array2 = [].concat(array1)**;
2. let obj2 = **Object.assign(obj1)**; or **let obj2 = {…obj1} // shallow clone will not work in object inside another object.**
3. **Deep clone is required**

|  |
| --- |
| let obj = {  a: 'a',  b: 'b',  c: {  deep: 'try and copy me'  }  };  let clone = Object.assign({}, obj);  let clone2 = {...obj}  let superClone = JSON.parse(JSON.stringify(obj))  obj.c.deep = 'hahaha';  console.log(obj)  console.log(clone)  console.log(clone2)  console.log(superClone) |

## Type Coercion

Means the language converting the certain type to another type.

1 == ‘1’ // true

1 === ‘1’ // false

Never use == to compare anything in JavaScript instead use === to prevent type coercion

## Static Typing vs Dynamic Typing

JavaScript is dynamic type language.

Static typing usually prevents bugs and helps keep error to happen, but dynamic typing allows to be more flexible and write software faster.

Type Script allows to make the JavaScript code to be written in static typing.

## Strong vs Weekly Type Language

JavaScript is weekly typed language.

# The 2 pillars: closures and Prototypal Inheritance

## Functions are Objects

Functions are special type of objects that is callable object with ()- for invoking, it has name and also contains properties (call, apply, bind).

Functions are objects in JS that means we can pass them around like objects means we can also store them as data.

|  |
| --- |
|  |

## First Class Citizens

Functions are first class citizens in JS because: -

1. We can declare function in a variable

|  |
| --- |
| var a = function(){  console.log('Hiiiiii');  }  a(); |

1. Function can be passed as variable.

|  |
| --- |
| function b (fn){  fn();  }  b(function(){console.log('Hello')}); |

1. We can return function as a value from other function

|  |
| --- |
| function c(){  return function d(){  console.log('27000000');  }  }  c()(); |

## Higher Order Function

A function that can take a function as an argument or a function that return another function.

1. Passing a function into another function.

|  |
| --- |
| const giveAccessTo = (name) =>  'Access Granted to ' + name;  function authenticate(person) {  let array = [];  // you can add checks here for person.level  for (let i = 0; i < 50000; i++) {  array.push(i)  }  return giveAccessTo(person.name)  }  function letPerson(person, fn) {  // ++ We now tell the function what data to use when we call it not when we define it + tell it what to do.  if (person.level === 'admin') {  return fn(person)  } else if (person.level === 'user') {  return fn(person)  }  }  function sing(person) {  return 'la la la my name is ' + person.name  }  letPerson({level: 'user', name: 'Tim'}, sing) |

1. Return a function from another function

|  |
| --- |
| const multiplyBy = num1 => {  return function (num2) {  return num1 \* num2;  }  }  const multiplyByTwo = multiplyBy(2);  multiplyByTwo(4)  //Short cut using ES6  const multiplyBy = num1 => (num2) => num1 \* num2;  multiplyBy(2)(5); |

## Closure (function + lexical scope)

A closure is a feature in JavaScript where an inner function has access to the outer (enclosing) function’s variables — a scope chain.

The closure has three scope chains:

* it has access to its own scope — variables defined between its curly brackets
* it has access to the outer function’s variables
* it has access to the global variables.

|  |
| --- |
| function a(){  let grandpa = 'grandpa';  return function b(){  let father = 'father';  return function c(){  let son = 'son';  return `${grandpa} > ${father} > ${son}`  }  }  }  a()()(); |

### Benefits of Closure

#### Memory Efficient

|  |
| --- |
| //With out using closure  function heavyDuty(item) {  const bigArray = new Array(7000).fill('😄')  console.log('created!');  return bigArray[item]  }  heavyDuty(699)  heavyDuty(699)  heavyDuty(699)  const getHeavyDuty = heavyDuty2();  getHeavyDuty(699)  getHeavyDuty(699)  getHeavyDuty(699)  // but i dont want to pollute the global namespace.. and using closure  function heavyDuty2() {  const bigArray = new Array(7000).fill('😄') // it will create once in closure memory  console.log('created Again!')  return function(item) {  return bigArray[item]  }  } |

#### Encapsulation

Hiding of unnecessary things to the outside world.

## Prototypal Inheritance

1. **JavaScript uses prototypal Inheritance which is not popular in other language (Java, c#) those language uses the classical inheritance.**
2. **Even though JavaScript is having the class keyword it is nothing but the synthetic sugar, actually JavaScript don’t have any class it only has prototypal inheritance.**
3. **In JavaScript array and function are prototypal inheritance of Objects means they can access the functionality of the object.**
4. **An object get access to the properties and methods of another object through prototype chain.**

|  |
| --- |
|  |

|  |
| --- |
| let array = [];  array.\_\_proto\_\_ // it gives access to method of array object  array.\_\_proto\_\_.\_\_proto\_\_ // it gives access to method of object |

Example of Prototypal inheritance

|  |
| --- |
| let dragon = {  name: 'Tanya',  fire: true,  fight() {  return 5  },  sing() {  if (this.fire) {  return `I am ${this.name}, the breather of fire`  }  }  }  let lizard = {  name: 'Kiki',  fight() {  return 1  }  }  // Don't do this, bad performance. Show with bind.  lizard.\_\_proto\_\_ = dragon;  dragon.isPrototypeOf(lizard);  console.log(lizard.fire)  console.log(lizard.sing())  const lizardFire =dragon.sing.bind(lizard)  console.log(lizardFire()) |

1. **Never use the \_\_proto\_\_ in real application as it lead to performance issue which we will discuss in object oriented programming section**
2. **Instead of use \_\_proto\_\_ we can use Object.create to inheriting from other object.**

|  |
| --- |
| // Create our own prototypes:  var human = {mortal: true}  var socrates = Object.create(human);  human.isPrototypeOf(socrates); // true |

1. **Only functions have the prototype property.**

|  |
| --- |
|  |

**Summary:- Everything in JavaScript is object . array and functions are object they inherit through the prototype chain from the base object, we can go up the prototype chain looking for property on the prototype property.**

**Benefits of prototypes :- We avoid repeating the code means avoiding the same code over and over and being inefficient with our memory**

# Object Oriented Programming

|  |
| --- |
|  |

## OOP

Object oriented programming is all about modeling real world object and relation.

Two types of OOP language.

1. Prototype based programming languages. (JavaScript)
2. Class based programming languages. (C#, Java etc.

### Factory Function

Function that helps in creating object for us.

Disadvantages: - **It need more memory once the object get bigger and bigger because same functionality is being by multiple object.**

|  |
| --- |
| // factory function make/create  function createElf(name, weapon) {  //we can also have closures here to hide properties from being changed.  return {  name: name,  weapon: weapon,  atack() {  return 'atack with ' + weapon  }  }  }  const sam = createElf('Sam', 'bow');  const peter = createElf('Peter', 'bow');  sam.atack() |

### Object.create()

Object.create() nothing but using the prototypal inheritance to get the properties/method of another object.

In general, we don’t use this in most code bases

|  |
| --- |
| const elfFunctions = {  attack: function() {  return 'atack with ' + this.weapon  }  }  function createElf(name, weapon) {  //Object.create creates \_\_proto\_\_ link  newElf = Object.create(elfFunctions)  newElf.name = name;  newElf.weapon = weapon  return newElf  }  const sam = createElf('Sam', 'bow');  const peter = createElf('Peter', 'bow');  sam.attack() |

### Constructor function

Any function that is invoke using keyword is called constructor function. Also as a naming standard all constructor function name should be pascal case.

|  |
| --- |
| //Constructor Functions  function Elf(name, weapon) {  this.name = name;  this.weapon = weapon;  }  Elf.prototype.attack = function() {  return 'atack with ' + this.weapon  }  const sam = new Elf('Sam', 'bow');  const peter = new Elf('Peter', 'bow');  sam.attack() |

### ES6 Classes

Class is not a synthetic sugar of prototypal inheritance in JavaScript.

|  |
| --- |
| class Elf {  constructor(name, weapon) {  this.name = name;  this.weapon = weapon;  }  attack() {  return 'atack with ' + this.weapon  }  }  const fiona = new Elf('Fiona', 'ninja stars');  console.log(fiona instanceof Elf) //  const ben = new Elf('Ben', 'bow');  fiona.attack() |

### Object.create() vs Class

Some developer like Object.create() and some Class at the end of the day it up to you which one to use.\

### this keyword uses in 4 ways

|  |
| --- |
| // new binding  function Person(name, age) {  this.name = name;  this.age =age;  console.log(this);  }  const person1 = new Person('Xavier', 55)  //implicit binding  const person = {  name: 'Karen',  age: 40,  hi() {  console.log('hi' + this.name)  }  }  person.hi()  //explicit binding  const person3 = {  name: 'Karen',  age: 40,  hi: function() {  console.log('hi' + this.setTimeout)  }.bind(window)  }  person3.hi()  // arrow functions to lexical bind the name property.  const person4 = {  name: 'Karen',  age: 40,  hi: function() {  var inner = () => {  console.log('hi ' + this.name)  }  return inner()  }  }  person4.hi() |

### Inheritance

If we want to extend some functionality from parent class via inheritance, we can achieve that.

In JavaScript Object inherit from another Object, but other languages (Java, C#) class inherit from another class.

In the below it just synthetic sugar of prototypal inheritance.

|  |
| --- |
| class Character {  constructor(name, weapon) {  this.name = name;  this.weapon = weapon;  }  attack() {  return 'atack with ' + this.weapon  }  }  class Elf extends Character {  constructor(name, weapon, type) {  // console.log('what am i?', this); this gives an error  super(name, weapon)  console.log('what am i?', this);  this.type = type;  }  }  class Ogre extends Character {  constructor(name, weapon, color) {  super(name, weapon);  this.color = color;  }  makeFort() { // this is like extending our prototype.  return 'strongest fort in the world made'  }  }  const houseElf = new Elf('Dolby', 'cloth', 'house')  //houseElf.makeFort() // error  const shrek = new Ogre('Shrek', 'club', 'green')  shrek.makeFort() |

### 4 pillars of OOP

1. Encapsulation: -
2. Abstraction: - Hiding the complexity of the code from the user. That is creating simple interfaces i.e. User just need create the object and use the property/method of that class. It helps via private and public keywords.
3. Inheritance:
4. Polymorphism: Ability to appear a method in many forms e.g. Method overloading (different method parameter in same class) and Method overriding (different method parameter in parent and child class)

# Functional Programming

## Introduction

Functional Programming is all about separation of concern which OOP does as well. It all about packing of code in separate chunks so that all the things are well organized.

**But it also separate data of a program and function/behavior of a program means Pure function.**

**Functional Programming is really good in one to one data transformation.**

Benefits is also same as OOP.

|  |
| --- |
|  |

## Pure Function

1. It should not have no side effects (means it should not change anything to the outside world)
2. Same Input should give the same output.

|  |
| --- |
| //Side effects:  const array = [1,2,3];  function mutateArray(arr) {  arr.pop()  }  function mutateArray2(arr) {  arr.forEach(item => arr.push(1  ))  }  //The order of the function calls will matter.  mutateArray(array)  mutateArray2(array)  array  // map and concat methods can fix this issue of mutation |

Benefits of Pure function: -

1. Very easy to test.
2. Very easy to compose.
3. Avoid lot of bug (because of no mutation means no shared state).

## Can Everything be pure?

No, as we need to interact with the outside world like change the DOM, http calls etc.

Goal of functional programming is not to make everything pure function, the goal is to minimize side effects

|  |
| --- |
|  |

## Idempotence

A function that always returns or those what we expected to do.

## Imperative vs Declarative

## Immutability

Means not changing the data. For example, don’t change the object in the outside world inside change in the inside world/function, so that if that object used by other function it can use.

|  |
| --- |
| const obj = {name: 'Andrei'}  function clone(obj) {  return {...obj}; // this is pure  }  function updateName(obj) {  const obj2 = clone  (obj);  obj2.name = 'Nana'  return obj2  }  const updatedObj = updateName(obj)  console.log(obj, updatedObj) |

## Higher Order function and Closures

**Closures**: - Just like object, closures in JavaScript is a mechanism for containing some sort of state.

We simply define a function inside another function and expose the inner function either by returning it or passing it another function, so the we can use that variable.

|  |
| --- |
| //HOF  const hof = (fn) => fn(5);  hof(function a(x){ return x})  //Closure  const closure = function() {  let count = 55;  return function getCounter() {  return count;  }  }  const getCounter = closure()  getCounter()  getCounter()  getCounter() |

## Currying

A function that take multiple parameters, instead currying we can modify the function that take one parameter at a time.

|  |
| --- |
| //Currying  const multiply = (a, b) => a \* b  const curriedMultiply = (a) => (b) => a \* b  curriedMultiply(5)(20)  const multiplyBy5 = curriedMultiply(5)  multiplyBy5(20) |

## Partial Application

It is the process of producing the function with smaller number of parameters. In partial application on the second function call it except all the argument.

|  |
| --- |
| //Partial Application  const multiply = (a, b, c) => a \* b \* c  const partialMultiplyBy5 = multiply.bind(null, 5)  partialMultiplyBy5(10, 20) |

## Memoization

In computing, memoization or memoisation is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again

|  |
| --- |
| //learn to cache  function addTo80(n) {  return n + 80;  }  addTo80(5)  let cache = {};  function memoizeAddTo80(n) {  if (n in cache) {  return cache[n];  } else {  console.log('long time');  const answer = n + 80;  cache[n] = answer;  return answer;  }  }  // console.log(1, memoizeAddTo80(6))  // // console.log(cache)  // // console.log('-----------')  // console.log(2, memoizeAddTo80(6))  // let's make that better with no global scope. This is closure in javascript so.  function memoizeAddTo80(n) {  let cache = {};  return function(n) {  if (n in cache) {  return cache[n];  } else {  console.log('long time');  const answer = n + 80;  cache[n] = answer;  return answer;  }  }  }  const memoized = memoizeAddTo80();  console.log(1, memoized(6))  // console.log(cache)  // console.log('-----------')  console.log(2, memoized(6)) |

## Compose

Compose is a system design principle that deals with the relationship between component, that can be selected and assembled various combination.

|  |
| --- |
| fn1(fn2(fn3(50)));  compose(fn1, fn2, fn3)(50) //Right to lext  pipe(fn3, fn2, fn1)(50)//left to right  const compose = (f, g) => (a) => f(g(a))  const pipe = (f, g) => (a) => g(f(a))  const multiplyBy3AndAbsolute = compose((num) => num\*3, Math.abs)  console.log(multiplyBy3AndAbsolute(-50)) |

## Arity

Arity means how many arguments a function takes.

## Amazon Problem solution

|  |
| --- |
| const user = {  name: 'Kim',  active: true,  cart: [],  purchases: []  }  const history1 = [];  const compose = (f, g) => (...args) => f(g(...args))  const pipe = (f, g) => (...args) => g(f(...args))  const purchaseItem = (...fns) => fns.reduce(compose);  const purchaseItem2 = (...fns) => fns.reduce(pipe);  purchaseItem2(  addItemToCart,  applyTaxToItems,  buyItem,  emptyUserCart,  )(user, {name: 'laptop', price: 60})  // purchaseItem(  // emptyUserCart,  // buyItem,  // applyTaxToItems,  // addItemToCart  // )(user, {name: 'laptop', price: 50})  function addItemToCart(user, item) {  history1.push(user)  const updatedCart = user.cart.concat(item)  return Object.assign({}, user, {cart: updatedCart});  }  function applyTaxToItems(user) {  history1.push(user)  const {cart} = user;  const taxRate = 1.3;  const updatedCart = cart.map(item => {  return {  name: item.name,  price: item.price\*taxRate  }  })  return Object.assign({}, user, { cart: updatedCart });  }  function buyItem(user) {  history1.push(user)  const itemsInCart = user.cart;  return Object.assign({}, user, { purchases: itemsInCart });  }  function emptyUserCart(user) {  history1.push(user)  return Object.assign({}, user, { cart: [] });  }  function refundItem() {  }  function getUserState() {  }  function goBack() {  }  function goForward() {  } |

# OOP vs FP

## Inheritance vs Composition

Inheritance is a super class that is extended to smaller pieces that add or overwrite things.

Composition is about smaller pieces that are combined to create something bigger

## OOP vs FP

* + - FP: -

1. Avoiding side effects and writing pure would be call functional programming.
2. Many Operation on fixed data.
3. Stateless.
4. Pure no side effect.
5. Declarative.
6. Data and behavior are kept separately,
7. When to use: - Functional Programming is quite good at processing large data for an application.
   * + OOP
8. Organizing the code into units would be called OOP.
9. Few Operation on common data
10. Stateful.
11. Side effect.
12. Imperative.
13. Data and behavior are kept together using class.
14. When to use: - Where we can multiple object interact with each other.

# Asynchronous JavaScript

## Job Queue

When promises were introduce in ECMAScript, the processing of promises was not done in Callback Queue/Task Queue.

Instead a new queue was introduced called Job Queue/Microtask Queue, this queue is similar to Callback Queue but little smaller.

Job Queue has priority than Callback Queue means Event loop will check the Job queue first if it is empty then it will see the callback queue.

|  |
| --- |
|  |

## Parallel, Sequence, and Race

**Parallel**: - If we want to execute multiple promises at a time.

**Sequential**: - If we want to execute multiple promises sequentially one by one.

**Race**: - If we want to execute multiple promises but whichever promises comes first execute that than ignore other.

**Fastest to execute :- race > parallel > sequential**

|  |
| --- |
| const promisify = (item, delay) =>    new Promise((resolve) => setTimeout(() => resolve(item), delay));  const a = () => promisify("a", 100);  const b = () => promisify("b", 5000);  const c = () => promisify("c", 3000);  async function parallel() {    const promises = [a(), b(), c()];    const [output1, output2, output3] = await Promise.all(promises);    return `prallel is done: ${output1} ${output2} ${output3}`;  }  async function race() {    const promises = [a(), b(), c()];    const output1 = await Promise.race(promises);    return `race is done: ${output1}`;  }  async function sequence() {    const output1 = await a();    const output2 = await b();    const output3 = await c();    return `sequence is done ${output1} ${output2} ${output3}`;  }  sequence().then(console.log);  parallel().then(console.log);  race().then(console.log); |

## Threads, concurrency and parallelism

When we open a new tab in browser it creates a one thread per tab, as soon we close the tab that thread dies.

Some operation we need to do in background for that Browser has something called web worker that work in the background.

In Node.js we use V8 engine which is single thread but for multiple operation we have LIBUV (which is written in C++) which contain worker thread to do multiple task like file operation, network, etc.

<https://www.simplilearn.com/understanding-node-js-architecture-article>

|  |
| --- |
|  |

|  |
| --- |
|  |

In browser we have web worker.

|  |
| --- |
|  |

# Modules

Modules is created to organize the code nicely and also think about separation of concern

Simply putting multiple script file in the html file while not treated a separated module, instead while loading in browser multiple script file will be treated as single script file.

## What is a module

Back in the day we used to have only one JS file, but things started bigger and bigger we can not have one big JS file.

Module give us to better way to organize the variable and function, so that we can group those variables and functions that make sense together.

## Module Pattern

We just return whatever we need, below example we make public only to the fight function by returning it and assigning to a variable, so that we can access only that function or variable.

other variable we cann’t access from outside.

|  |
| --- |
|  |

## Pros and cons of Module pattern

**Pros:-**

1. Individual developer can work on separate module.
2. Reusability.

Cons:-

1. We technically polluting the global namespace

## CommonJS, AMD, UMD

CommonJs and AMD has solved the problem of dependency resolution and the pollution of the global scope.

All the module we can bundle in js file using <http://browserify.org/>

## ES6 Modules

Using import and export.

And also remember to use **type=”module”** in the script tag, so the import and export syntax should work in browser.

# Error Handling

## Errors in JavaScript

In JavaScript we have **Error** constructor which helps to create a error. But for throwing a error we use the **throw** keyword.

Also, when we throw something the script which is currently running stop execute means executing of the current function will stop and control will be passed to next part of the call stack.

|  |
| --- |
|  |

|  |
| --- |
| throw "Error2"; // String type  throw 42; // Number type  throw true; // Boolean type  throw Error;  throw new Error(); // will create an instance of an Error in JavaScript and stop the execution of your script.  function a() {    const b = new Error("what?");    return b;  }  a().stack;  let error = new Error(message);  let error2 = new SyntaxError(message);  let error3 = new ReferenceError(message); |

## Try Catch and Finally

|  |
| --- |
| function fail() {    try {      console.log("this works");      throw new Error("oopsie");    } catch (e) {      console.log("error", e);    } finally {      console.log("still good");      return "returning from fail";    }    console.log("never going to get here"); // not reachable  }  fail(); |

## Async Error Handling

|  |
| --- |
| Promise.resolve("asyncfail")    .then((response) => {      console.log(response);      throw new Error("#1 fail");    })    .then((response) => {      console.log(response);    })    .catch((err) => {      console.error("error", err.message);    })    .then((response) => {      console.log("hi am I still needed?", response);      return "done";    })    .catch((err) => {      console.error(err);      return "failed";    }); |

Using try catch for async code

|  |
| --- |
| (async function () {    try {      await Promise.reject("oopsie");    } catch (err) {      console.error(err);    }    console.log("This is still good!");  })(); |