* <https://reactjs.org/tutorial/tutorial.html#why-immutability-is-important>

Immutability helps in tracking changes, undo\refresh, and performance by determining when to re-render the component

* Optimize performance

<https://reactjs.org/docs/optimizing-performance.html#examples>

* Callback functions – When a function is passed as argument. In DCA in Api.js we pass handler function which is called based on the response. It is also known as **higher order function**
* Callback Hell – is nesting of callback functions. Promises were introduced to get rid of callback hell
* Promise – states pending, fulfilled and rejected. Promise.all() enables you to be notified once all results are in from parallel functions
* Promise benefits –

1. Guaranteed Future- once you have made a promise you would definitely get result or error. There won’t be any cancellation in between
2. Immutable –
3. Single Value – you send request and receive response. This is particularly useful in ajax request, but suffers in user interactions like search, websockets (stream of values), animations, real time graph updates
4. Caching – once you have made a promise and you listen to it multiple times you will always get same value each time
5. Not lazy loaded. The moment you call a function returning a promise it is executed

* Observables benefits – Promises are either fulfilled or rejected. There is no way to cancel it. You can unsubscribe to cancel observable for example in componentWillUnMount. Promises always return one data whereas Observables return stream of data. If you have very simple app where you don’t need stream of data then Promises are good enough as observables are not native to Javascript and you will have to import RxJS which would increase bundle size. With observables you have three things - complete, error and next

1. Stream of zero, one or more values (using subscribe.next)
2. Over any period of time stream of value
3. Cancellable
4. Lazy loaded – If you call a function which returns observable it will not execute till you subscribe to it. This helps to retry and repeat

* Closures - A closure is the combination of a function and the lexical environment within which that function was declared.
* Closure and callback difference -- In simple words: a callback using context variables is a closure. A callback depending on a context variable aka bound variables (== object state) will be a closure.
* Arrow functions – You cannot use new. Second, **\*this\*** is picked up from surroundings (lexical). Therefore, you don’t need bind() or that = this
* Virtual DOM – is react dom and copy of real dom. It is represented as tree. Whenever a component (node) changes in react don it updates only that part of real DOM. This increases performance as browser doesn’t have to rewrite entire DOM, which means it saves time on drawing the entire CSS, margin etc. again
* Higher Order Components
* Refs – for quick access to DOM elements

class CustomTextInput extends React.Component {

constructor(props) {

super(props);

// create a ref to store the textInput DOM element

this.textInput = React.createRef();

this.focusTextInput = this.focusTextInput.bind(this);

}

focusTextInput() {

// Explicitly focus the text input using the raw DOM API

// Note: we're accessing "current" to get the DOM node

this.textInput.current.focus();

}

render() {

// tell React that we want to associate the <input> ref

// with the `textInput` that we created in the constructor

return (

<div>

<input

type="text"

ref={this.textInput} />

<input

type="button"

value="Focus the text input"

onClick={this.focusTextInput}

/>

</div>

);

}

}

* Keys – are for react to find the differences in virtual dom nodes to find when to update real DOM. Refs are for developers to access dom elements and perform manipulations. Keys are used for reconciliation (diffing algorithm)
* React code is isomorphic which means it doesn’t contain browser specific code so it can run either on server or browser
* Webpack – bundle javascript
* Babel – compile JSX to javascript
* Provider is a React component given to us by the “react-redux” library. It serves just one purpose : to “provide” the store to its child components.
* Connect - map the stores state and maps dispatch to the props of a component
* Store.subscribe has the callback method which is called by Redux every time the state is changed. For example getState() can be called inside subscribe, so that it gets State everytime action is triggered.
* Javascript patterns
* Shadowing – When same variable is redefined in a function it shadows the function in outer context, this is called shadowing.

var shadow = ‘ test’;

function(){

var shadow = ‘inner test’

console.log(shadow); //Prints inner test

}

* Let and var – Difference is in scope. let is only visible in the for() loop and var is visible to the whole function.If using strict mode then let is not forgiving but var is. For example

'use strict';

let me = 'foo';

let me = 'bar'; // SyntaxError: Identifier 'me' has already been declared

'use strict';

var me = 'foo';

var me = 'bar'; // No problem, `me` is replaced.

the purpose of let statements only to free up memory when not needed in a certain block?

<https://stackoverflow.com/questions/762011/whats-the-difference-between-using-let-and-var-to-declare-a-variable>

* React patterns – Composability (combining components), higher order components (decorator design pattern), one way flow. Decorator pattern is an object which adds functionality to another object dynamically.
* Prototype -- Inheritance in Javascript can be achieved by Prototype. A prototype is an object (no surprises) and every function you create automatically gets a prototype property that points to a new blank object. You can add members to this blank object and later have other objects inherit from this object and use its properties as their own.
* Object.assign() copies the values (of all enumerable own properties) from one or more source objects to a target object.
* template literals in Es6 -- Template literals are the string with embedded code and variables inside. let c=`${a} ${b}`;
* Spread Operator: Spread Operator provides a new way to manipulate array and objects in Es6.A Spread operator is represented by … followed by the variable name.
* Bind: You would do this because when a function is used as a DOM event handler, the “this” is set to be the element that dispatched the DOM event. You want to guarantee that the this is the one you expect, since it's used in tick()

var logPokemon = pokemonName.bind(pokemon); // creates new object and binds pokemon. When you call this.getPokeName() in pokemonName then it will pick getPokeName method of pokemon

var pokemonName = function(snack, hobby) {

console.log(this.getPokeName() + 'I choose you!');

console.log(this.getPokeName() + ' loves ' + snack + ' and ' + hobby);

};

In above pokemonName this = pokemon because of the bind

* Call() -- The main differences between bind() and call() is that the call() method:

Accepts additional parameters as well

Executes the function it was called upon right away.

The call() method does not make a copy of the function it is being called on.

var pokemon = {

firstname: "Ibha",

lastname: "Gandhi",

getPokeName: function() {

return(this.firstname + this.lastname);

}

}

var pokemonName = function(snack, hobby) {

console.log(this.getPokeName() + ' loves ' + snack + ' and ' + hobby);

console.log(this.firstname);

};

pokemonName.call(pokemon,'sushi', 'algorithms');

* Apply -- The only difference between how they work is that call() expects all parameters to be passed in individually, whereas apply() expects an array of all of our parameters.

var pokemonName = function(snack, hobby) {

console.log(this.getPokeName() + ' loves ' + snack + ' and ' + hobby);

};

pokemonName.call(pokemon,'sushi', 'algorithms'); // Pika Chu loves sushi and algorithms

pokemonName.apply(pokemon,['sushi', 'algorithms']); // Pika Chu loves sushi and algorithms

**Call** and **Apply** are the only way to change the execution context when a function is invoked. You can pass an object as the first parameter of the **Call** and **Apply** and the **this** reference will point to the object passed the function.

Very good explanation on <https://www.undefinednull.com/2014/06/26/explaining-call-and-apply-in-javascript-through-mr-dot-dave/>

**Use .bind() when you want that function to later be called with a certain context, useful in events. Use .call() or .apply() when you want to invoke the function immediately, and modify the context.**

* V8: Javascript runtime in Google Chrome
* Web apis are what browser provides. Like DOM, asyn calls, setTimeout
* Call stack is data structure which tells where in program we are. Push and Pop method. When we enter the function we push to stack and when we return we pop out of the stack
* MVVM
* Browser will render only when stack is clear. Render has higher priority than callback queue. That is why statements like console.log will slow down performance as it will enter the stack and browser will be blocked from rendered till then.
* Target and CurrentTarget: Target is actual element that triggered event and currentTarget is the element to which you attached event listener to. For example if you attached event listener to form and clicked a button then form is currentTarget and button is target
* IIFE (Immediately Invoked Function Expression) is a JavaScript function that runs as soon as it is defined. This is to control variable scope
* Why not global variables: You may pull in other library which uses same name and there will be name collision. So a) Reduce Collision b) Maintain independence c) Easier to write your own code.
* Hoisting: Const and let are not hoisted. Their scope is well contained. Hoisting is like variables being forward declared at top of function. For Example

console.log(t); //undefined is output

var t = "var";

prints t as variables are considered to be declared at top of function.

* Event delegation and bubbling

<div id=‘elementOne’>

<div id=‘elementTwo’></div>

</div>

Event Capturing – This school of thought says that events always move from top to bottom i.e. in our case, event listener of elementOne fires first followed by that of elementTwo.

Event Bubbling – this theory says that events always move from bottom to top, i.e. event listener of elementTwo fires first and then that of elementOne.

* Functions as statements\declarations versus expressions -- Functions as statements\declarations versus expressions. The difference lies in how the browser loads them into the execution context. Function declarations load before any code is executed. Function expressions load only when the interpreter reaches that line of code.

**Example: Function Expression**

alert(foo()); // ERROR! foo wasn't loaded yet

var foo = function() { return 5; }

**Example: Function Declaration**

alert(foo()); // Alerts 5. Declarations are loaded before any code can run.

function foo() { return 5; }

Advantage of function expression is that you can declare functions based on certain conditions. For example:

if(testCondition) {// If testCondition is true then

var foo = function(){

console.log("inside Foo with testCondition True value");

};

}else{

var foo = function(){

console.log("inside Foo with testCondition false value");

};

}

* Triple equal checks for type and equality
* ES6 Generators:

It’s useful for iterating through infinite sequences, where you may want to pause and do something until next item is ready to be processed

function\* nextGen() {

let n = 0;

while (n < 3) {

yield n++;

}

let gen = nextGen();

console.log(gen.next().value);

console.log(gen.next().value);

console.log(gen.throw(new Error(‘oops!’) ));

}

In above example nextGen is not called at line number 6 but at line 7, 8 and 9 with next();

Gen.next() returns an object with {value: value, done: true\false}

If there are multiple yields each gen.next will return value of one yield. For example

1. function\* f(){
2. let n = 0;
3. while (n < 3) {
4. yield n++;
5. yield n + 10;
6. }
7. }
8. let gen = f();
9. console.log(gen.next().value);
10. console.log(gen.next().value);

Line 9 will return 0 and line 9 will return 11

You can call generators from generators using yield\*

Example: Continuously test water flow rate in a pipe. The pipe is connected to the fire rose you see in the ceiling all over offices.

- You iterate over the flow rate continuously, which is normally zero.

- If heat from a fire breaks the glass, the water flow suddenly goes up. The rose is now spraying water onto a "supposed" fire.

- Suddenly flow is 7, as in 7 cubic feet per second. Danger!

- When we get 7 (or > 0) we want to check adjacent "rose" outlets in the area, (e.g. same floor). If they are still zero, then maybe it's not a fire, just a clumsy workman who broke the glass.

- So, call another generator (nested) to check say the 5 rose outlets around the first rose, which caused the alarm. If any return a value > 0 ... call 911 - Fire Dept.

- If all others are zero, ring for IMMEDIATE visual inspection (N1) to the bldg maintenance and designated floor captain. (N1: Some places have closed circuit TV to see the exact spot.)

- Example here is that one continuous monitoring of water flow to one spot, can cause a call to a "check nearby" function and send back a report from each nearby spot, before responding.﻿

* Gulp or grunt - Gulp has a major advantage when it comes to speed
* 2 known libraries for React Unit test – Jest and Enzyme -- Jest is a framework and not a library. It comes with a test runner, assertion library, and good mocking support. Jest is built on top of Jasmine. Enzyme is not a unit testing framework. It does not have a test runner or an assertion library. It makes it easier to assert, manipulate, and traverse your React Components' output

**Test runner** — a tool that picks up files that contain unit tests, executes them, and writes the test results to the console or log files. Mocha and Jasmine are two popular test runners used within the JavaScript community. You can read a comparison between the two [here](https://www.codementor.io/javascript/tutorial/javascript-testing-framework-comparison-jasmine-vs-mocha).

**Assertion library** — verifies the results of a test. Chai, Should, and Expect are examples of JavaScript assertion libraries.

**Mocks** — used in unit testing a component. A component under test has many dependencies. These dependencies are usually replaced by stubs or mocks. Stubs simulate a dependent object. Mocks offer an additional feature over stubs. With mocks, tests can be written to verify if the component under test has called the mocks as expected.

**Mocking library** — facilitates the usage of mocks in unit testing. Sinon and TestDouble are commonly used JavaScript mocking libraries.

* Undefined and Not Defined –

var x; console.log(x) // returns undefined

console.log(y)//y is not defined

<https://www.codementor.io/nihantanu/21-essential-javascript-tech-interview-practice-questions-answers-du107p62z>

* Object is collection of properties and each property associated with the name-value pairs. Objects can be created in two types using literals or constructor

Using literals

var emptyObj ={

empId:”Red”,

empCode: “X0091”,

empDetail : function(){

alert(“Hi”);

};

};

Using constructor

var obj = new Object();

Obj.empId=”001”;

Obj.empCode=”X0091”;

* New features of HTML5 – There are new semantic elements like header, nav, footer, support for svg and video etc.
* New features of CSS3 – media query, animations, flex-box
* CSS resetting versus CSS normalizing –

CSS resets aim to remove all built-in browser styling. Standard elements like H1-6, p, strong, em, et cetera end up looking exactly alike, having no decoration at all. You're then supposed to add all decoration yourself.

Normalize CSS aims to make built-in browser styling consistent across browsers. Elements like H1-6 will appear bold, larger et cetera in a consistent way across browsers. You're then supposed to add only the difference in decoration your design needs.

* Define inheritance in Javascript

//This is a parent class.

var parent = {

sayHi: function () {

alert('Hi, I am parent!');

},

sayHiToWalk: function () {

alert('Hi, I am parent! and going to walk!');

}

};

//This is child class and the parent class is inherited in the child class.

var child = Object.create(parent);

child.sayHi = function () {

alert('Hi, I am a child!');

};

* Add/Remove Properties to Object in run-time in JavaScript?

var objectJSON = {

    id: 1,

    name: "Anil Singh",

    dept: "IT"

};

//This is the process to delete

delete objectJSON.dept;

console.log(objectJSON.dept); // Answer: undefined

//This is used to add the property.

objectJSON.age = 30;

* **eval()** function used in execute an argument as expression

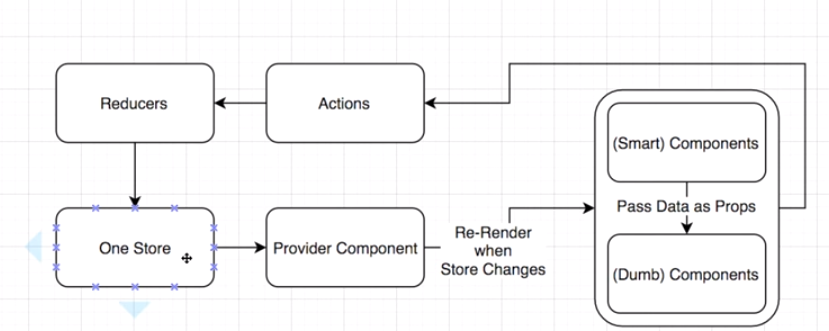
var x = 14;

eval('x + 10'); //The output is 24.

* The**floor ()** function is a static method of Math and we can write as **Math.floor()** and used to round the number of downwards i.e.

Math.floor(1.6);//The output is 1.

* Event Loop is a queue of callback functions. http://latentflip.com/loupe/



Reducer is a pure function that takes previous state and an action and returns a new state.

Provider makes the store available throughout the component tree

Actions describe that something has happened but don’t describe how the state changes

Architectural Challenges: Multiple products coming together.

1. First challenge Single workflow spanning across products
2. Interaction of UI with other components\products
3. Scaling thousands of (80k) resources
4. Performance while loading 1000’s of resources
5. Responsiveness with big number of resources and different age customers
6. Pressure of more and more features in less time

Where should you do input validation – thunk, react or redux

Ans: Thunk

Validatons should be done like:

fetchPostsIfNeeded is action creators that return a function instead of an action. The thunk can be used to delay the dispatch of an action, or to dispatch only if a certain condition is met. T

function shouldFetchPosts(state, reddit) {

const posts = state.postsByReddit[reddit]

if (!posts) {

return true

} else if (posts.isFetching) {

return false

} else {

return posts.didInvalidate

}

}

// This is closure function

export function fetchPostsIfNeeded(reddit) {

return (dispatch, getState) => {

if (shouldFetchPosts(getState(), reddit)) {

return dispatch(fetchPosts(reddit))

}

}

}

Promises interview questions

<https://medium.com/@ivantsov/promises-in-js-one-more-interview-question-6d59634a3463>

* filter and reduce 🡺

filter syntax is same as map

var arr = [

{name:"tiya", age:7},

{name:"ibha", age:40},

{name:"chiya", age:10},

{name:"navdeep", age:47},

]

console.log(arr.filter(elem => elem.age>39));

var num = [2,3,4,5];

console.log(num.reduce((prev,curr) => prev+curr));

* Prototype versus constructor methods

var YourClass = function(){

var privateField = "somevalue";

this.publicField = "somevalue";

this.instanceMethod1 = function(){

//you may access both private and public field from here:

//in order to access public field, you must use "this":

alert(privateField + "; " + this.publicField);

};

}

YourClass.prototype.instanceMethod2 = function(){

//you may access only public field 2 from this method, but not private fields:

alert(this.publicField);

//error: drawaback of prototype methods:

alert(privateField);

};

Debounce --- Run a method after a delay. Common use case is search. We don’t want to run a function on every key down but after few seconds delay

Javascript Best Practices

Avoid Global Variables

Always Declare Local Variables

Declarations on Top

Initialize Variables

if(function(){}) {

console.log("Entered If");

} else {

console.log("Entered else");

}

Answer: Entered If as function(){} when converted to boolean return true

0 when converted to Boolean returns false

“0” and “000” when converted to Boolean return true

HTML5 new features

Semantic Tags like nav, footer, header, sections

Tags to embed graphics, audio, video

Web workers

New form controls, search,calendar,date,time,email etc.

Variable Hoisting

<https://www.youtube.com/watch?v=QyUFheng6J0>

function makeaddr(n) {

var inc = n;

var sum = 0;

return function () {

sum = sum + inc;

console.log(sum);

return sum;

}

}

var addr3 = makeaddr(3);

addr3(); //prints 3

addr3();// prints 6

* At a fundamental level, functional programming specifies the use of functions as arguments
* Push -- The push() method can append one or more elements to the end of an array.
* Pop -- The pop() method removes one or more elements from the end of an array.
* shift -- The shift() method is like the pop() method, only it works at the beginning of the array.
* unshift -- The unshift() method is like the push() method, only it works at the beginning of the array.

Cache busting --- Cache busting solves the browser caching issue by using a unique file version identifier to tell the browser that a new version of the file is available. Therefore the browser doesn't retrieve the old file from cache but rather makes a request to the origin server for the new file.

Deep Clone vs Shallow Copy - <https://www.freecodecamp.org/news/copying-stuff-in-javascript-how-to-differentiate-between-deep-and-shallow-copies-b6d8c1ef09cd/>

A deep copy means that all of the values of the new variable are copied and **disconnected from the original**variable. A shallow copy means that certain (sub-)values are **still connected** to the original variable.

int b = a //deep clone as original is not impacted

const a = {en:”English”,gm:”German”}

let b = a; // shallow clone, creates a reference

b.gm = “Spanish”

console.log(a.gm) //Spanish

using ES6

let b = Object.assign({},a) // creates new object doesn’t change a when b is changed // shallow copy

if a={food:{veg:”cheese”,nonveg:”chicken”}}

b = Object.assign({},a) then as food is nested it is copied as reference.

b.food.veg = “pasta”

console.log(a.food.veg); //pasta

To actual deepclone an object use lodash.[cloneDeep](https://lodash.com/docs#cloneDeep) or angular.copy

Object.keys(obj) – returns an array of keys.

Object.values(obj) – returns an array of values.

Object.entries(obj) – returns an array of [key, value] pairs.

Object.fromEntries(array) on the array to turn it into an object.

Objects lack many methods that exist for arrays, e.g. map, filter to apply them, Object.entries should be used to convert it to Array and after operations are performed it can be changed to bject using Object.fromEntries

Webpack as part of angular-cli

<https://hackernoon.com/webpack-for-angular-developers-c8584a60e627>

Diamond Problem if multi inheritance is allowed

The "diamond problem" (sometimes referred to as the "Deadly Diamond of Death"[6]) is an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C. If there is a method in A that B and C have overridden, and D does not override it, then which version of the method does D inherit: that of B, or that of C?

JSX- Javascript XML. It allows us to write HTML inside JavaScript and place them in the DOM without using functions like appendChild( ) or createElement( ) like we do in render methos under return

Compilers output binary and Transpilers outputs new code

Diagram

Description automatically generated

Babel is a transpiler converts from ECMAScript to Javascript

MonoRepo – advantages one version for all projects so no conflicts in versions.

Google and Facebook maintain one company wide mono repo so that they update versions at one go

DDD-Domain driven design

Microfrontends – first and foremost reason is to scale it to various teams so that they can work simultaneously

Separate development

Separate deployment

Own architectural designs

Ow technology decision – which is anti pattern

**Design Patterns**

Decorator pattern: In Angular @Input and @Component are decorators

In React HOC are decorator pattern example

I have code I want to extend it. How do I add behaviour without touching real implementation. Composition is kind of decorator pattern.

Observer Pattern: Angular has observables

Change detection uses this pattern.

Redux store uses this pattern

Strategy Pattern: The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

In my own words the behavior is encapsulated in other class and the implementation details are hidden from the class using it.

**Monkey and Banana Problem** in inheritance. Even though you just need banana you get monkey also due to inheritance.

Example duck problem if the duck class had swim, quack and fly methods then all the types of ducks even those who cannot fly will not to implement this method and will have to say return nothing unneccesarily.

Private readonly test; Difference between constant and readonly

they effectively both do the same thing, but one is for variables and the other is for properties.

--------------------

const myArr = [1, 2, 3];

// Not allowed

myArr = [4, 5, 6]

// Perfectly fine

myArr.push(4);

// Perfectly fine

myArr[0] = 9;

myArr is a constant reference to a mutable object. You can't point it at some other array, but you can change the contents of the array.

--------------------

Typescript Function overloading with different number of parameters and types with same name is not supported.

**Invalid**

function display(a:string, b:string):void //Compiler Error: Duplicate function implementation

{

console.log(a + b);

}

function display(a:number): void //Compiler Error: Duplicate function implementation

{

console.log(a);

}

**Valid**

function add(a:string, b:string):string;

function add(a:number, b:number): number;

Difference between forEach and map

**Table

Description automatically generated**

**Singleton: example for database connection pooling**

The isNan() function returns true if the variable value is “not a number”.

In JavaScript, the event.preventDefault() method is used to prevent the default behavior of an element. **For example:** If you use it in a form element, it prevents it from submitting. If used in an anchor element, it prevents it from navigating. If used in a contextmenu, it prevents it from showing or displaying. On the other hand, the event.stopPropagation() method is used to stop the propagation of an event or stop the event from occurring in the bubbling or capturing phase.

"View state" is specific to a page in a session whereas "Session state" is specific to a user or browser that can be accessed across all pages in the web application.

**Worst case time complexity**: expressed in Big O. Example if we have array of 5 elements and the element to be searched is matched is the last one then it is worst case of that algo

**Best case time complexity**: expressed in [Big omega notation](https://www.geeksforgeeks.org/analysis-of-algorithms-set-3asymptotic-notations/). Example if we have array of 5 elements and the element to be searched is matched is the first one then it is best case of that algo

**Average case time complexity**: expressed in [Big theta notation](https://www.geeksforgeeks.org/analysis-of-algorithms-set-3asymptotic-notations/). Example if we have array of 5 elements and the element to be searched is matched is in middle then it is average case of that algo

**Space complexity:** memory space needed to carry out the algorithm. expressed in Big O.

* JavaScript by default uses insertion sort for the sort() method. This means that it is not appropriate when sorting large data sets. When dealing with large data sets, one should consider other sorting algorithms such as merge sort.

Time complexity of Recursion versus Iteration

Usage of either of these techniques is a trade-off between time complexity and size of code. If time complexity is the point of focus, and number of recursive calls would be large, it is better to use iteration. However, if time complexity is not an issue and shortness of code is, recursion would be the way to go.

* **Recursion**: Recursion involves calling the same function again, and hence, has a very small length of code. However, as we saw in the analysis, the time complexity of recursion can get to be exponential when there are a considerable number of recursive calls. Hence, usage of recursion is advantageous in shorter code, but higher time complexity.
* **Iteration**: Iteration is repetition of a block of code. This involves a larger size of code, but the time complexity is generally lesser than it is for recursion.

Good article on time complexity of sorting techniques

[Sorting Algorithms in JavaScript | Engineering Education (EngEd) Program | Section](https://www.section.io/engineering-education/sorting-algorithms-in-js/#merge-sort)

Typeof array is object.To find if it is array

        function isArray(arr) {

          return arr.constructor === Array;

        }

        console.log("It is array ",isArray([2,4,3,5,6]));

typeof null is also object

Must read: [JavaScript typeof (w3schools.com)](https://www.w3schools.com/js/js_typeof.asp)

[25+ JavaScript Coding Interview Questions (SOLVED with CODE) | FullStack.Cafe](https://www.fullstack.cafe/blog/javascript-code-interview-questions)

Coding Guidelines

1. Easy to read
2. Scalable
3. Maintainable
4. Modular
5. Reusable