Good morning / Good Afternoon / Good Evening

My name is Sumant Chauhan

I am a Web developer at Tecnovos Technologies.

I have over 3 years of experience in web development, JavaScript, React Js, Redux, Node Js, Debugging ,HTML, CSS, Material UI, Ant-Design and so on.

I have working knowledge of different types of projects like Social Media, Management, Portals.

**Project:**

Currently I am working on a project called Form And Work Flow. It’s a kind of portal. We called as Admin-Portal and Customer Portal.

In Admin portal you can create Forms, Workflows, Assign Permission to Users, Create Menus, Assign Menus.

In Customer portal you can use that form what we created on admin portal.

Example: If you want to create a user, then first you need to create a workflow and configure user creation api in that workflow. Then you need to create a form with required field and configure user creation workflow to this form on submit. And then we need to create a menu and assign that menu to particular role or group.

If you login with that role In customer side you will find that form what we created in Admin side.

Once we fill all form fields and on submit user will create.

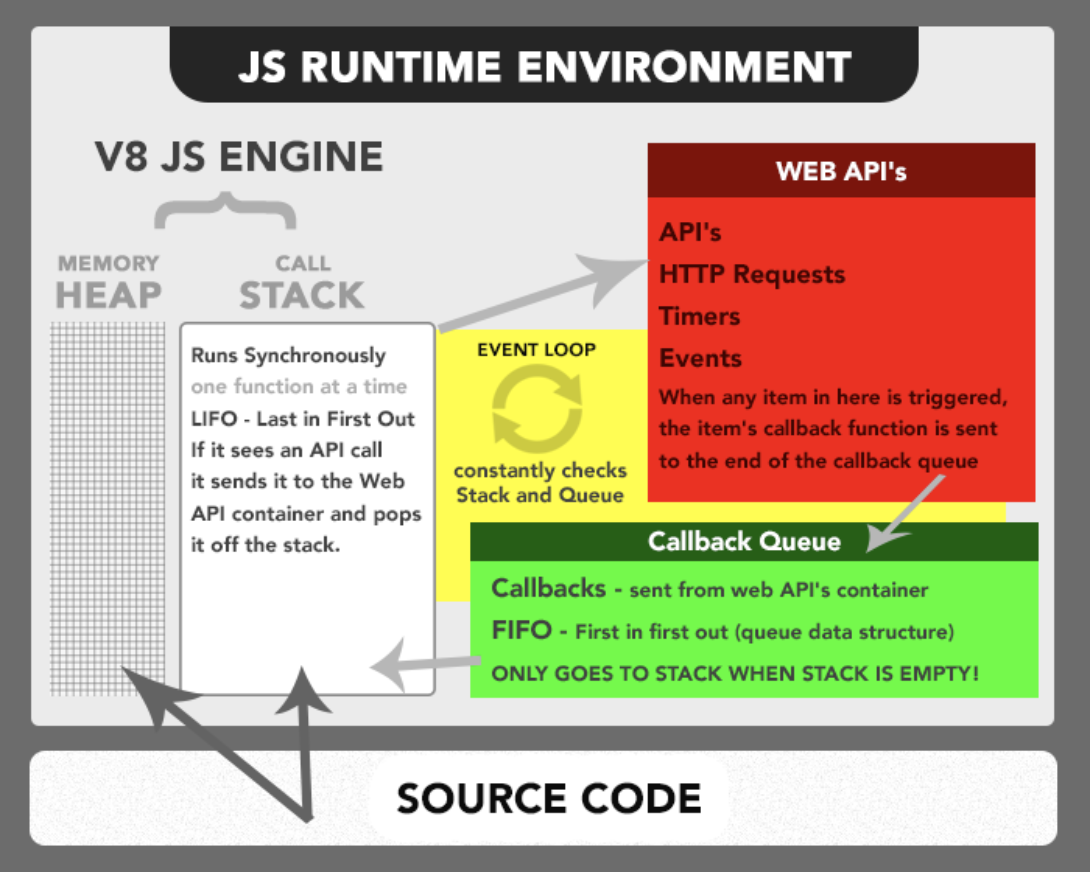
So it is completely dynamic.

You can configure any APIs in workflows.

**What is JavaScript engine and Runtime?**

**JavaScript runtime** refers to where your **javascript** code is executed when you run it. That said, **javascript** can be executed on google chrome, in which case your **javascript engine**is v8, if on mozilla — it is spidermonkey, if IE — then its chakra, if Safari — it’s nitro and if on node, again its v8. Now what is JS engine and what is JS runtime.

Engine converts the javascript we write into machine code. All JavaScript engines implement [**specification of the language** provide by ECMAScript](https://www.ecma-international.org/publications/standards/Ecma-262.htm). Standardisation facilitates the development of independent engines and ensures your scripts give the same results no matter where you run them. In order to obtain speed, V8 translates JavaScript code into more efficient machine code instead of using an interpreter. It compiles JavaScript code into machine code at execution by implementing a **JIT (Just-In-Time) compiler** like a lot of modern JavaScript engines do such as SpiderMonkey or Rhino (Mozilla). The main difference here is that V8 doesn’t produce bytecode or any intermediate code. JavaScript engine is just a building block of a bigger concept. This engine works inside an environment called Javascript Runtime which provides additional features to our scripts. These features can be making a call to web, catching mouse/keyboard events, etc.



## Single Threaded, What??

Javascript code is executed in a single thread but Javascript runtime is not run in single thread. Thread pool exists in JS runtime but we don’t have to worry about it as Runtime takes care of it. But how does it does that? **Event loop** to the rescue.

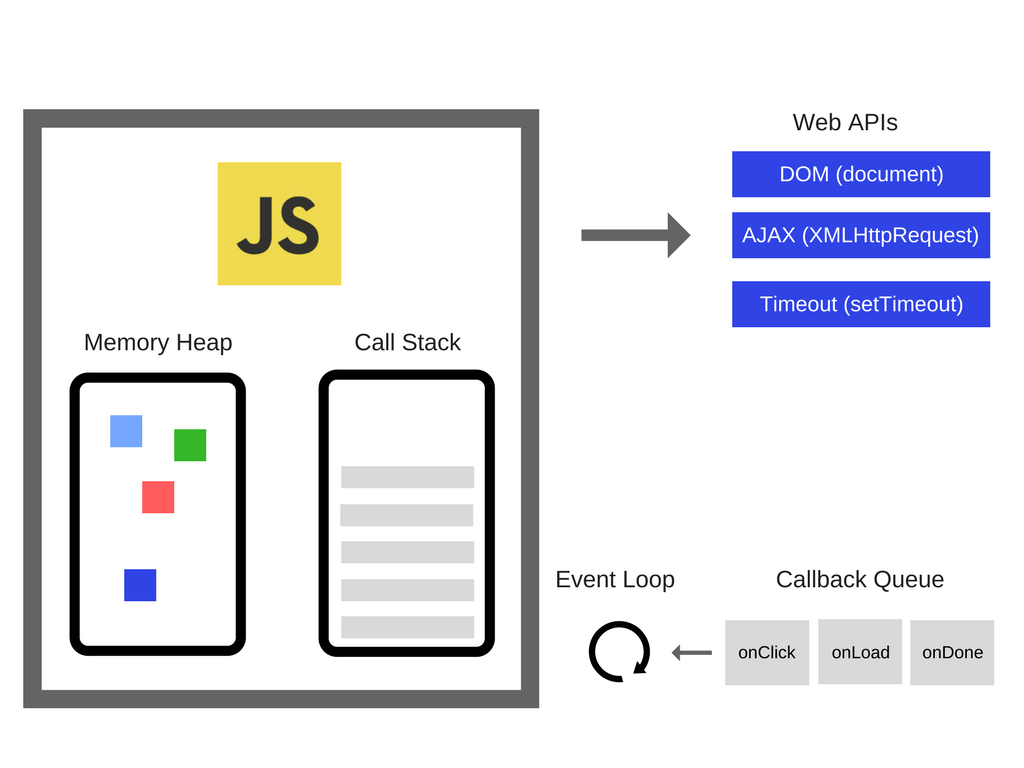
Lets understand what is Heap and Call stack in the Runtime (or in JS enginee which belongs to Runtime). The javascript code is first converted to machine code. Heap stores all the variables and call stack performs the operations.

We can divide the scripts in 2 types namely — immediately invoked and invoked for later.

What happens when asynchronous tasks come? Tasks which take time to run. Say making an API call or Timer, etc. There is a concept called callback. It is the function to be executed when this task is done.

Well they go into the call stack as any normal functions but we make a call to WebAPIs as this task resided in WebAPIs. It stores the callback function for the task and does the task for us (using threading / multi processing depending on the runtime). And when the task is finished, it sends the callback to callback queue.

Visualise again here.



Now what is event loop? Event loop runs continuously (in Browser runtime which it doesn’t always runs in node) to check if the call stack is empty and if it is empty, it picks up the first item from the callback queue and moves it to call stack and executes the callback function. Until the stack is not empty, no function is added from callback queue.

Callbacks are **A L W A Y S**executed completely. **The Event loop runs one callback at a time**. No context switching. All callbacks in the queue have to wait until the current one is finished. If a script runs too long, it blocks others. That’s why **callbacks should be relatively short and simple**.

Pretty simple right! But in reality it’s much more complex. There are multiple queues depending on the runtime and their priorities are different. There is something as rendering queue. Whose job is to render the screen.

# **If Javascript Is Single Threaded, How Is It Asynchronous?**

Javascript is a single threaded language. This means it has one call stack and one memory heap. As expected, it executes code in order and must finish executing a piece code before moving onto the next. It's synchronous, but at times that can be harmful. For example, if a function takes awhile to execute or has to wait on something, it freezes everything up in the meanwhile.

A good example of this happening is the window alert function. alert("Hello World")

You can't interact with the webpage at all until you hit OK and dismiss the alert. You're stuck.

So how do we get asynchronous code with Javascript then?

Well, **we can thank** the Javascript engine (V8, Spidermonkey, JavaScriptCore, etc...) for that, which has Web API that handle these tasks in the background. The call stack recognizes functions of the Web API and hands them off to be handled by the browser. Once those tasks are finished by the browser, they return and are pushed onto the stack as a callback.

Open your console and type window then press enter. You'll see most everything the Web API has to offer. This includes things like ajax calls, event listeners, the fetch API, and setTimeout. Javascript uses low level programming languages like C++ to perform these behind the scenes.

console.log("first")

setTimeout(() => {

console.log("second")

}, 1000)

console.log("third")

Output:

first

third

undefined

second

Feels odd, right? Well, let's break this down line by line:

console.log("first") is on the stack first, so it gets printed. Next, the engine notices setTimeout, which isn't handled by Javascript and pushes it off to the WebAPI to be done asynchronously. The call stack moves on without caring about the code handed off to the Web APIs and console.log("three") is printed.

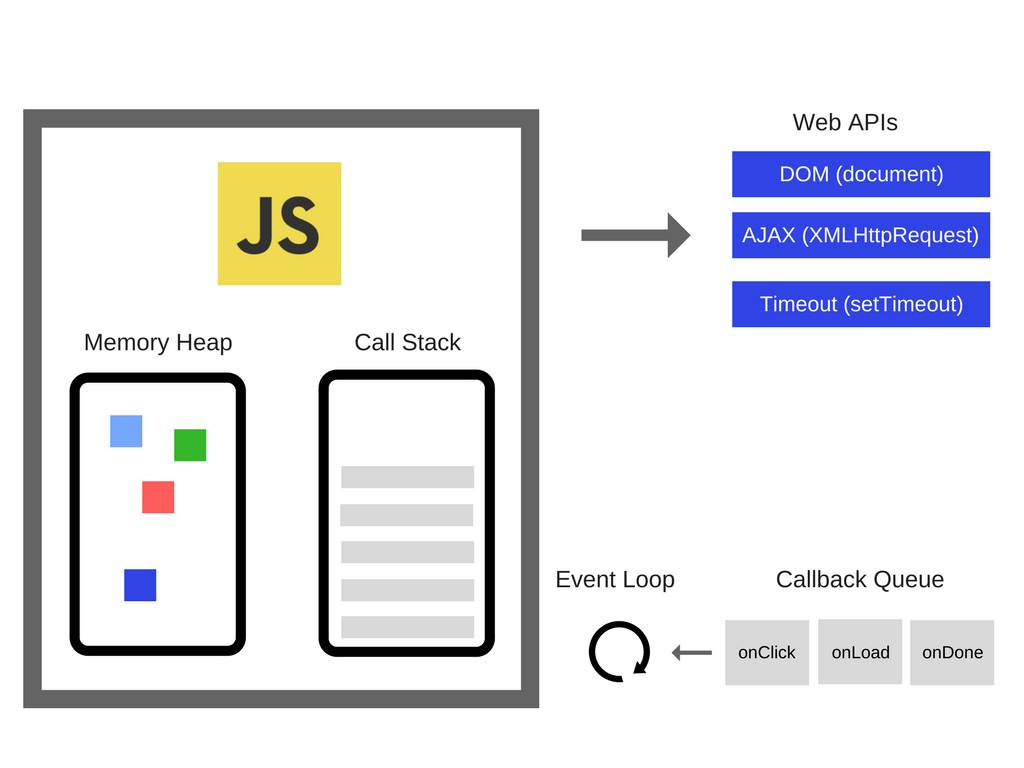
Next, the Javascript engine's event loop kicks in, like a little kid asking "Are we there yet?" on a road trip. It starts firing, waiting for events to be pushed into it. Since the setTimeout isn't finished, it returns undefined, as the default, well because it hasn't been given the value yet. Once the callback finally does hits we get console.log("second") printed.

**JavaScript is a single-threaded language and, at the same time, also non-blocking, asynchronous and concurrent.**

## ****RUNTIME****

JavaScript is an interpreted language, not a compiled one. This means that it needs an interpreter which converts the JS code to a machine code. There are several types of interpreters (known as engines). The most popular browser engines are V8 (Chrome), Quantum (Firefox) and WebKit (Safari). Incidentally, V8 is also used in a popular non-browser runtime, Node.js.

Each engine contains a memory heap, a call stack, an event loop, a callback queue and a WebAPI with HTTP requests, timers, events, etc., all implemented in its own way for faster and safer interpretation of the JS code.



## ****SINGLE THREAD****

A single-thread language is one with a single call stack and a single memory heap. It means that it runs only one thing at a time.

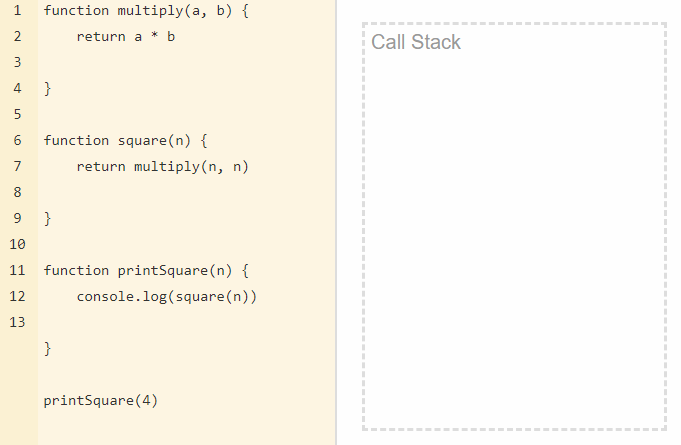
A stack is a continuous region of memory, allocating local context for each executed function.

A heap is a much larger region, storing everything allocated dynamically.

A call stack is a data structure which basically records where we are in the program.

### **CALL STACK**

Let's write a simple code and track what's happening on the call stack.



As you can see, the functions are added to the stack, executed and later deleted. It's the so-called LIFO way - Last In, First Out. Each entry in the call stack is called a stack frame.

Knowledge of the call stack is useful for reading error stack traces. Generally, the exact reason for the error is at the top in first line, though the order of code execution is bottom-up.

Sometimes you can deal with a popular error, notified by Maximum call stack size exceeded. It is easy to get this using recursion:

function foo() {

foo()

}

foo()

and our browser or terminal freezes. Each browser, even their different versions, has a different call stack size limit. In the vast majority of cases, they are sufficient and the problem should be looked for elsewhere.

### **EVENT LOOP**

The question is, how does the runtime know that the call stack is empty and how is the event in the callback queue invoked? Meet event loop. It is a part of the JS engine. This process constantly checks if the call stack is empty and, if it is, monitors whether there is an event in the callback queue waiting to be invoked.

That's all the magic behind the scenes!

## What is a Lexical Scope?

## Every inner level can access its outer levels.

## **The inner functions have access to the variables and other resources of their parent scope**. This means that the child functions are lexically bound to the execution context of their parents. Lexical scope is sometimes also referred to as static scope.

## Lexical scope or static scope is a variable defined outside your scope or upper scope is automatically available inside your scope which means you don't need to pass it there.

A lexical scope or static scope in JavaScript refers to the accessibility of the variables, functions, and objects based on their physical location in the source code. For example:

let a = 'global';

function outer() {  
 let b = 'outer'; function inner() {  
 let c = 'inner'  
 console.log(c); // prints 'inner'  
 console.log(b); // prints 'outer'  
 console.log(a); // prints 'global'  
 }  
 console.log(a); // prints 'global'  
 console.log(b); // prints 'outer'  
 inner();  
 }  
outer();  
console.log(a); // prints 'global'

Here the inner function can access the variables defined in its own scope, the outer function’s scope, and the global scope. And the outer function can access the variable defined in its own scope and the global scope.

# **Closures**

A closure gives you access to an outer function’s scope from an inner function **even after the outer function has returned.** In JavaScript, closures are created every time a function is created, at function creation time. All functions in JavaScript are closures.

This means a closure can remember and access variables and arguments of its outer function even after the function has finished.

## Execution Context

An execution context is an abstract environment where the JavaScript code is evaluated and executed. When the global code is executed, it’s executed inside the global execution context, and the function code is executed inside the function execution context.

There can only be one currently running execution context (Because JavaScript is single threaded language), which is managed by a stack data structure known as Execution Stack or Call Stack.

An execution stack is a stack with LIFO (Last in, first out) structure in which items can only be added or removed from the top of the stack only.

The currently running execution context will be always on the top of the stack, and when the function which is currently running completes, its execution context is popped off from the stack and the control reaches to the execution context below it in the stack.



When this code is executed, the JavaScript engine creates a global execution context to execute the global code, and when it encounters the call to first() function, it creates a new execution context for that function and pushes it to the top of the execution stack.

## Lexical Environment

## Every time the JavaScript engine creates an execution context to execute the function or global code, it also creates a new lexical environment to store the variable defined in that function during the execution of that function.

# **Hoisting**

In JavaScript, variable and function names can be used before declaring it. The JavaScript compiler moves all the declarations of variables and functions at the top so that there will not be any error. This is called hoisting.

Hoisting is only possible with declaration but not the initialization. JavaScript will not move variables that are declared and initialized in a single line.

console.log(counter); // undefined

var counter = 1;

console.log(counter1); //ReferenceError: Cannot access 'counter1' before initialization

let counter1 = 2;

log(counter3); //undefined

var counter3;

counter3 = 3;

counter4 = 4;

log(counter4); //4

var counter4;

* JavaScript hoisting occurs during the creation phase of the execution context that moves the variable and function declarations to the top of the script.
* The JavaScript engine hoists the variables declared using the let keyword, but it doesn’t initialize them as the variables declared with the var keyword.
* Function expressions and arrow functions aren’t hoisted.

## What's the difference between a Controlled component and an Uncontrolled one in React?

* A [Controlled Component](https://facebook.github.io/react/docs/forms.html#controlled-components) is one that takes its current value through props and notifies changes through callbacks like onChange. A parent component "controls" it by handling the callback and managing its own state and passing the new values as props to the controlled component. You could also call this a "dumb component".
* A [Uncontrolled Component](https://facebook.github.io/react/docs/uncontrolled-components.html) is one that stores its own state internally, and you query the DOM using a ref to find its current value when you need it.

The difference between them is that components that their value is set/passed and have a callback are called controlled components (<input type="text" value="value" onChange={handleChangeCallbackFn} />) vs. traditional HTML where an input element handle their own value and can be read via refs called uncontrolled components (<value type="text" />). Controlled components are managing their own state via setState or getting it from their parent component as props.

# **Pure Components in React**

Pure Components are introduced for performance enhancement. You can use this optimization to improve the performance of your components.

**Pure Components** in React are the components which do not re-renders when the value of state and props has been updated with the same values. If the value of the previous state or props and the new state or props is the same, the component is not re-rendered. Pure Components restricts the re-rendering ensuring the higher performance of the Component.

**Features of React Pure Components**

* Prevents re-rendering of Component if props or state is the same
* Takes care of “shouldComponentUpdate” implicitly
* State and Props are Shallow Compared
* Pure Components are more performant in certain cases

class ShalloWCompareComponent extends React.PureComponent {

constructor() {

super();

this.state = {

userArray: [1, 2, 3, 4, 5]

}

// The value of Counter is updated to same value during continues interval

setInterval(() => {

this.setState({

userArray: userArray.push(6)

});

}, 1000);

}

render() {

return <b>Array Length is: {this.state.userArray.length}</b>

}

}

In the code above, before re-rendering, the initial and the final value of state and props object are compared with Shallow Comparison. Since we’re adding value to the same array object, the reference this.state.userArray remains the same. React Component will compare the reference of previous and the new state variable userArray. As they are point to the same reference, no change will be detected and the component will not be re-rendered, leading to the unexpected result in the user interface.

Use Pure Components, in the case when the props and state changes are made to primitive type variable, state and props changes to reference variable may lead to incorrect results and inconsistent rendering.

# **React Portal:** React Portal provide the way to render children into a DOM node that exists outside the DOM hierarchy of the parent component.

ReactDOM.createPortal(child, container)

The first argument (child) is any [renderable React child](https://reactjs.org/docs/react-component.html#render), such as an element, string, or fragment. The second argument (container) is a DOM element.

where the child is the [Component](https://reactjs.org/docs/portals.html) which will be rendered at the place of the container which is dom node.

### **The Component Lifecycle**

Each component has several “lifecycle methods” that you can override to run code at particular times in the process.

#### **Mounting**

* [**constructor()**](https://reactjs.org/docs/react-component.html#constructor)
* [static getDerivedStateFromProps()](https://reactjs.org/docs/react-component.html#static-getderivedstatefromprops)
* [**render()**](https://reactjs.org/docs/react-component.html#render)
* [**componentDidMount()**](https://reactjs.org/docs/react-component.html#componentdidmount)

#### **Updating**

An update can be caused by changes to props or state. These methods are called in the following order when a component is being re-rendered:

* [static getDerivedStateFromProps()](https://reactjs.org/docs/react-component.html#static-getderivedstatefromprops)
* [shouldComponentUpdate()](https://reactjs.org/docs/react-component.html#shouldcomponentupdate)
* [**render()**](https://reactjs.org/docs/react-component.html#render)
* [getSnapshotBeforeUpdate()](https://reactjs.org/docs/react-component.html#getsnapshotbeforeupdate)
* [**componentDidUpdate()**](https://reactjs.org/docs/react-component.html#componentdidupdate)

#### **Unmounting**

This method is called when a component is being removed from the DOM:

* [**componentWillUnmount()**](https://reactjs.org/docs/react-component.html#componentwillunmount)

Unmounting in function component

React.useEffect(() => {

return () => {

console.log(props.current);

};

}, [props.current]);

#### **Error Handling**

These methods are called when there is an error during rendering, in a lifecycle method, or in the constructor of any child component.

* [static getDerivedStateFromError()](https://reactjs.org/docs/react-component.html#static-getderivedstatefromerror)
* [componentDidCatch()](https://reactjs.org/docs/react-component.html#componentdidcatch)

### render()

The render() method is the only required method in a class component.

render() method will call every time when state or props will changes.

render() will not be invoked if [shouldComponentUpdate()](https://reactjs.org/docs/react-component.html#shouldcomponentupdate) returns false.

### constructor()

**If you don’t initialize state and you don’t bind methods, you don’t need to implement a constructor for your React component.**

The constructor for a React component is called before it is mounted. When implementing the constructor for a React.Component subclass, you should call super(props) before any other statement. Otherwise, this.props will be undefined in the constructor, which can lead to bugs.

Typically, in React constructors are only used for two purposes:

* Initializing [local state](https://reactjs.org/docs/state-and-lifecycle.html) by assigning an object to this.state.
* Binding [event handler](https://reactjs.org/docs/handling-events.html) methods to an instance.

You **should not call**setState() in the constructor(). Instead, if your component needs to use local state, **assign the initial state to this.state** directly in the constructor:

### componentDidMount()

componentDidMount() is invoked immediately after a component is mounted (inserted into the tree). Initialization that requires DOM nodes should go here. If you need to load data from a remote endpoint, this is a good place to instantiate the network request.

This method is a good place to set up any subscriptions. If you do that, don’t forget to unsubscribe in componentWillUnmount().

### componentDidUpdate()

componentDidUpdate(prevProps, prevState, snapshot)

componentDidUpdate() is invoked immediately after updating occurs. This method is not called for the initial render.

Use this as an opportunity to operate on the DOM when the component has been updated. This is also a good place to do network requests as long as you compare the current props to previous props (e.g. a network request may not be necessary if the props have not changed).

componentDidUpdate(prevProps) {

// Typical usage (don't forget to compare props):

if (this.props.userID !== prevProps.userID) {

this.fetchData(this.props.userID);

}

}

You **may call setState() immediately** in componentDidUpdate() but note that **it must be wrapped in a condition** like in the example above, or you’ll cause an infinite loop. It would also cause an extra re-rendering which, while not visible to the user, can affect the component performance. If you’re trying to “mirror” some state to a prop coming from above, consider using the prop directly instead.

componentDidUpdate() will not be invoked if [shouldComponentUpdate()](https://reactjs.org/docs/react-component.html#shouldcomponentupdate) returns false.

### shouldComponentUpdate()

shouldComponentUpdate(nextProps, nextState)

shouldComponentUpdate() is invoked before rendering when new props or state are being received. Defaults to true. This method is not called for the initial render or when forceUpdate() is used.

Compare this.props with nextProps and this.state with nextState and return false to tell React the update can be skipped. Note that returning false does not prevent child components from re-rendering when their state changes.

### static getDerivedStateFromProps()

static getDerivedStateFromProps(props, state)

getDerivedStateFromProps is invoked right before calling the render method, both on the initial mount and on subsequent updates. It should return an object to update the state, or null to update nothing.

### getSnapshotBeforeUpdate()

getSnapshotBeforeUpdate(prevProps, prevState)

getSnapshotBeforeUpdate() is invoked right before the most recently rendered output is committed to e.g. the DOM. It enables your component to capture some information from the DOM (e.g. scroll position) before it is potentially changed. Any value returned by this lifecycle method will be passed as a parameter to componentDidUpdate().

### Error boundaries

[Error boundaries](https://reactjs.org/docs/error-boundaries.html) are React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of the component tree that crashed. Error boundaries catch errors during rendering, in lifecycle methods, and in constructors of the whole tree below them.

A class component becomes an error boundary if it defines either (or both) of the lifecycle methods static getDerivedStateFromError() or componentDidCatch(). Updating state from these lifecycles lets you capture an unhandled JavaScript error in the below tree and display a fallback UI.

Only use error boundaries for recovering from unexpected exceptions; **don’t try to use them for control flow.**

Error boundaries only catch errors in the components **below** them in the tree. An error boundary can’t catch an error within itself.

### static getDerivedStateFromError()

static getDerivedStateFromError(error)

This lifecycle is invoked after an error has been thrown by a descendant component. It receives the error that was thrown as a parameter and should return a value to update state.

getDerivedStateFromError() is called during the “render” phase, so side-effects are not permitted. For those use cases, use componentDidCatch() instead.

### componentDidCatch()

componentDidCatch(error, info)

This lifecycle is invoked after an error has been thrown by a descendant component. It receives two parameters:

1. error - The error that was thrown.
2. info - An object with a componentStack key containing [information about which component threw the error](https://reactjs.org/docs/error-boundaries.html#component-stack-traces).

componentDidCatch() is called during the “commit” phase, so side-effects are permitted. It should be used for things like logging errors:

# UseMemo

Unlike useEffect, React.useMemo does not trigger every time you change one of its dependencies.

A memoized function will first check to see if the dependencies have changed since the last render. If so, it executes the function and returns the result. If false, it simply returns the cached result from the last execution.

This is good for expensive operations like transforming API data or doing major calculations that you don't want to be re-doing unnecessarily

const posts = Redux.useSelector(state => state.posts);

const tags = React.useMemo(() => {

return getTagsFromPosts(posts)

}, [posts]);

// Generates random colours any time it's called

const randomColour = () => '#'+(Math.random()\*0xFFFFFF<<0).toString(16);

# **What is “this” keyword in JavaScript**

**this** keyword refers to an object, that object which is executing the current function.

By default this refers to global in node and window in browses.

* If we are in strict mode then the default value of *this* keyword is undefined otherwise *this* keyword act as global object, it’s called default binding of *this* keyword. (default is window object in case of browser).

Every JavaScript function while executing has a reference to its current execution context, called **this**. Execution context means here is how the function is called.

#### **It takes different values depending upon the usage**

1. Inside a method.
2. Inside a function.
3. Alone.
4. In an event.
5. call(), and apply().

**Inside a method**

When this is used inside a method, it refers to the owner object.

Functions defined inside an object are called methods.

# **“this” Refers to a New Instance**

When a function is invoked with the new keyword, then the function is known as a constructor function and returns a new instance. In such cases, the value of this refers to a newly created instance.

function Person(fn, ln) {

this.first\_name = fn;

this.last\_name = ln;

this.displayName = function () {

console.log(`Name: ${this.first\_name} ${this.last\_name}`);

};

}

let person = new Person("John", "Reed");

person.displayName(); // Prints Name: John Reed

let person2 = new Person("Paul", "Adams");

person2.displayName(); // Prints Name: Paul Adams

In the case of person.displayName, this refers to a new instance person, and in case of person2.displayName(), this refers to person2 (which is a different instance than Person).

# **“this” With the Call and Apply Methods**

A function in JavaScript is also a special type of object. Every function has call, bind, and apply methods. These methods can be used to set a custom value to this in the execution context of the function.

function Person(fn, ln) {

this.first\_name = fn;

this.last\_name = ln;

this.displayName = function() {

console.log(`Name: ${this.first\_name} ${this.last\_name}`);

}

}

let person = new Person("John", "Reed");

person.displayName(); // Prints Name: John Reed

let person2 = new Person("Paul", "Adams");

person2.displayName(); // Prints Name: Paul Adams

person.displayName.call(person2); // Here we are setting value of this to be person2 object

//Prints Name: Paul Adams

The only difference between the call and apply methods is the way an argument is passed. In the case of apply, the second argument is an array of arguments, whereas in the case of the call method, the arguments are passed individually.

# **“this” With the Bind Method**

The bind method returns a new method with this referring to the first argument passed. We’re going to use the above example to explain the bind method.

function Person(fn, ln) {

this.first\_name = fn;

this.last\_name = ln;

this.displayName = function() {

console.log(`Name: ${this.first\_name} ${this.last\_name}`);

}

}

let person = new Person("John", "Reed");

person.displayName(); // Prints Name: John Reed

let person2 = new Person("Paul", "Adams");

person2.displayName(); // Prints Name: Paul Adams

let person2Display = person.displayName.bind(person2); // Creates new function with value of “this” equals to person2 object

person2Display(); // Prints Name: Paul Adams

# **“this” With the Fat-Arrow Function**

As part of ES6, a new way was introduced to define a function.

let displayName = (fn, ln) => {

console.log(Name: ${fn} ${ln});

};

When a fat arrow is used, it doesn’t create a new value for this. this keeps on referring to the same object it’s referring to outside of the function.

# **Call( ), Bind( ), and Apply( )**

We use call, bind and apply methods to set the this keyword independent of how the function is called.

JavaScript also provides some special methods and properties to every function object. So every function in JavaScript inherits those methods. Call, bind, and apply are some of the methods that every function inherits.

# **Bind( )**

The bind method creates a new function and sets the this keyword to the specified object or Function.

ES5 introduced the [bind()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind) method to [set the value of a function's this regardless of how it's called](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this#The_bind_method).

function.bind(thisArg, optionalArguments)

## Bind() can also accept arguments

We can also pass extra arguments to the bind method. The general syntax for this is function.bind(this, arg1, arg2, ...)

# **Call ( )**

The call method sets the this inside the function and immediately executes that function.

The difference between call() and bind() is that the call() sets the this keyword and executes the function immediately and it does not create a new copy of the function, while the bind() creates a copy of that function and sets the this keyword.

function.call(thisArg, arg1, agr2, ...)

function greeting() {

console.log(`Hi, I am ${this.name} and I am ${this.age} years old`);

}

const john = {

name: 'John',

age: 24,

};

const jane = {

name: 'Jane',

age: 22,

};

// Hi, I am John and I am 24 years old

greeting.call(john);

// Hi, I am Jane and I am 22 years old

greeting.call(jane);

Above example is similar to the bind() example except that call() does not create a new function. We are directly setting the this keyword using call().

# **Apply ( )**

The apply() method is similar to call(). The difference is that the apply() method accepts an array of arguments instead of comma separated values.

function.apply(thisArg, [argumentsArr])

function greet(greeting, lang) {

console.log(lang);

console.log(`${greeting}, I am ${this.name} and I am ${this.age} years old`);

}

const john = {

name: 'John',

age: 24,

};

const jane = {

name: 'Jane',

age: 22,

};

// Hi, I am John and I am 24 years old

greet.apply(john, ['Hi', 'en']);

// Hi, I am Jane and I am 22 years old

greet.apply(jane, ['Hola', 'es']);

# **Arrow functions**

Unlike regular functions, arrow functions do not have their own this. In the case of an arrow function, this refers to the values of this in the environment the arrow function is defined in (i.e. "outside" the arrow function) and that remains the same throughout the lifecycle of the function and is always bound to the value of this in the closest non-arrow parent function.

Arguments objects are not available in arrow functions, but are available in regular functions.

### **What is the purpose of the array slice method**

The **slice()** method returns the selected elements in an array as a new array object. It selects the elements starting at the given start argument, and ends at the given optional end argument without including the last element. If you omit the second argument then it selects till the end.

**Note:** Slice method won't mutate the original array but it returns the subset as a new array.

### **What is the purpose of the array splice method**

The **splice()** method is used either adds/removes items to/from an array, and then returns the removed item. The first argument specifies the array position for insertion or deletion whereas the optional second argument indicates the number of elements to be deleted. Each additional argument is added to the array.

**Note:** Splice method modifies the original array and returns the deleted array.

**Some of the major difference in a tabular form**

| **Slice** | **Splice** |
| --- | --- |
| Doesn't modify the original array(immutable) | Modifies the original array(mutable) |
| Returns the subset of original array | Returns the deleted elements as array |
| Used to pick the elements from array | Used to insert or delete elements to/from array |

# What is the Temporal Dead Zone (TDZ) in JavaScript?

difference between let / const and var is that if you access var before it's declared, it is undefined. But if you do the same for let and const, they throw a ReferenceError.

console.log(varNumber); // undefined

console.log(letNumber); // ReferenceError letNumber is not defined

var varNumber = 1;

let letNumber = 1;

They throw the error all because of the Temporal Dead Zone.

Temporal Dead Zone describe the state where variables are un-reachable. They are in scope, but they aren't declared.

The let and constvariables exist in the TDZ from the start of their enclosing scope until they are declared.

{

// This is the temporal dead zone for the age variable!

// This is the temporal dead zone for the age variable!

// This is the temporal dead zone for the age variable!

// This is the temporal dead zone for the age variable!

let age = 25; // Whew, we got there! No more TDZ

console.log(age);

}

### **What are lambda or arrow functions**

An arrow function is a shorter syntax for a function expression and does not have its own **this, arguments, super, or new.target**. These functions are best suited for non-method functions, and they cannot be used as constructors.

## Explain what a callback function is and provide a simple example

A callback function is a function that is passed to another function as an argument and is executed after some operation has been completed.

function modifyArray(arr, callback) {

// do something to arr here

arr.push(100);

// then execute the callback function that was passed

callback();

}

var arr = [1, 2, 3, 4, 5];

modifyArray(arr, function() {

console.log("array has been modified", arr);

});

## Given a string, reverse each word in the sentence

For example Welcome to this Javascript Guide! should be become emocleW ot siht tpircsavaJ !ediuG

var string = "Welcome to this Javascript Guide!";

// Output becomes !ediuG tpircsavaJ siht ot emocleW

var reverseEntireSentence = reverseBySeparator(string, "");

// Output becomes emocleW ot siht tpircsavaJ !ediuG

var reverseEachWord = reverseBySeparator(reverseEntireSentence, " ");

function reverseBySeparator(string, separator) {

return string.split(separator).reverse().join(separator);

}

duplicate([1, 2, 3, 4, 5]); // [1,2,3,4,5,1,2,3,4,5]

function duplicate(arr) {

return arr.concat(arr);

}

duplicate([1, 2, 3, 4, 5]); // [1,2,3,4,5,1,2,3,4,5]

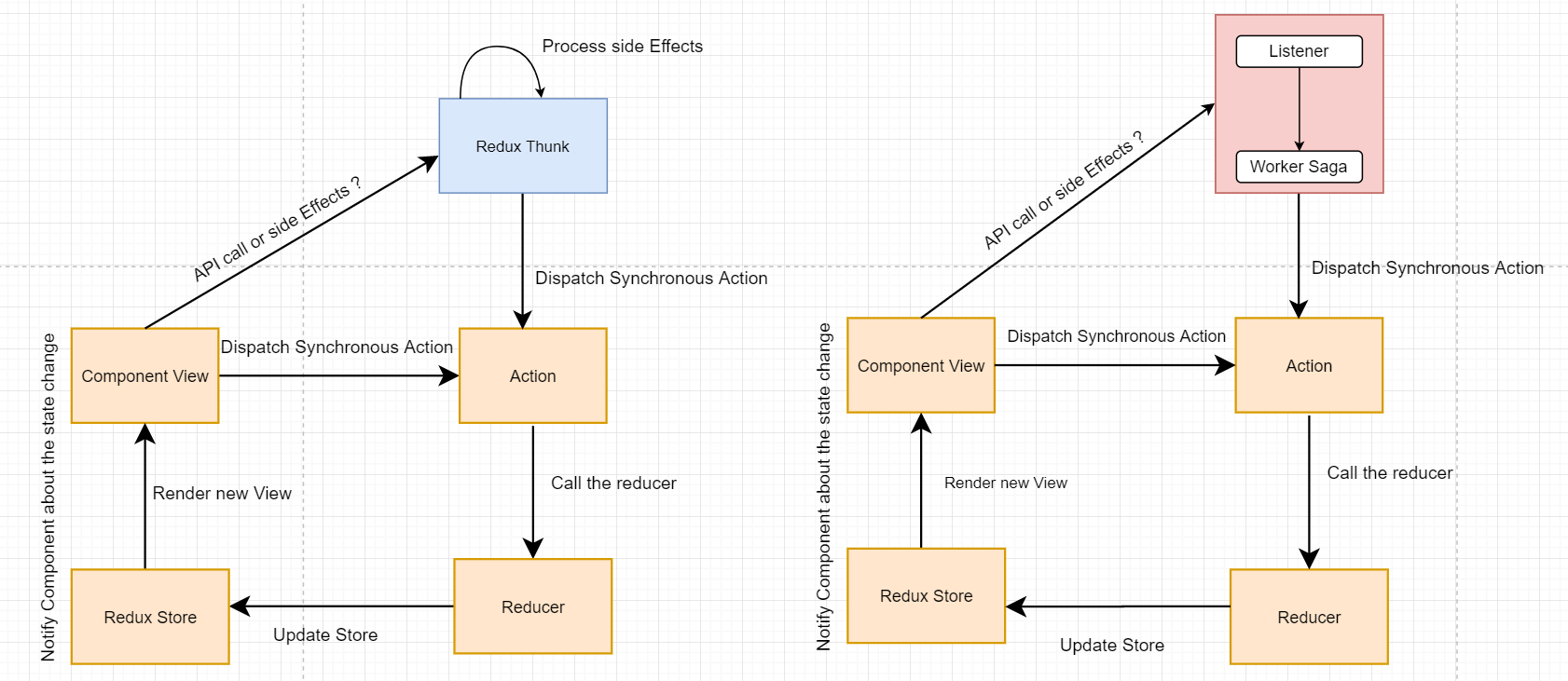
### What is a Redux middleware?

By default, Redux’s actions are dispatched synchronously, which is a problem to communicate with an external API or perform side effects. Redux also allows for middleware that sits between an action being dispatched and the action reaching the reducers.

Redux Thunk is a middleware that lets you call action creators that return a function instead of an action object. That function receives the store’s dispatch method, which is then used to dispatch regular synchronous actions inside the function’s body once the asynchronous operations have been completed.

Redux middleware is a function or a piece of code that sits between action and reducer and can interact with the dispatched action before reaching the reducer function.

A redux middleware can be used for many purposes such as logging (e.g. redux-logger), asynchronous API calls and so on.



Redux Thunk:

Redux Thunk is **middleware that allows you to return functions**, rather than just actions, within Redux.

One of the main use cases for this middleware is for handling actions that might not be synchronous, for example, using axios to send a GET request. Redux Thunk allows us to dispatch those actions asynchronously and resolve each promise that gets returned.

## What is a Closure?

A closure gives you access to an outer function’s scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

To use a closure, define a function inside another function and expose it. To expose a function, return it or pass it to another function.

The inner function will have access to the variables in the outer function scope, even after the outer function has returned.

Closures are important because they control what is and isn’t in scope in a particular function, along with which variables are shared between sibling functions in the same containing scope.

# useReducer:

useReducer is use to manage complex state in your components.

useReducer lets you manage local state of complex components with a reducer:

useReducer is usually preferable to useState when you have complex state logic that involves multiple sub-values or when the next state depends on the previous one. useReducer also lets you optimize performance for components that trigger deep updates.

const initialState = {count: 0};

function reducer(state, action) {

switch (action.type) {

case 'increment':

return {count: state.count + 1};

case 'decrement':

return {count: state.count - 1};

default:

throw new Error();

}

}

function Counter() {

const [state, dispatch] = useReducer(reducer, initialState);

return (

<>

Count: {state.count}

<button onClick={() => dispatch({type: 'decrement'})}>-</button>

<button onClick={() => dispatch({type: 'increment'})}>+</button>

</>

);

}

### useLayoutEffect:

useLayoutEffect fires synchronously after all DOM mutations. Use this to read layout from the DOM and synchronously re-render. Updates scheduled inside useLayoutEffect will be flushed synchronously, before the browser has a chance to paint.

### PropTypes

PropTypes can be used to make sure the data you receive is valid.

example, we’re using PropTypes.string. When an invalid value is provided for a prop, a warning will be shown in the JavaScript console. For performance reasons, propTypes is only checked in development mode.

The position Property

The position property specifies the type of positioning method used for an element.

There are five different position values:

* static
* relative
* fixed
* absolute
* sticky

## position: static;

HTML elements are positioned static by default.

Static positioned elements are not affected by the top, bottom, left, and right properties.

## position: relative;

An element with position: relative; is positioned relative to its normal position.

Setting the top, right, bottom, and left properties of a relatively-positioned element will cause it to be adjusted away from its normal position. Other content will not be adjusted to fit into any gap left by the element.

## position: fixed;

An element with position: fixed; is positioned relative to the viewport, which means it always stays in the same place even if the page is scrolled. The top, right, bottom, and left properties are used to position the element.

A fixed element does not leave a gap in the page where it would normally have been located.

## position: absolute;

An element with position: absolute; is positioned relative to the nearest positioned ancestor (instead of positioned relative to the viewport, like fixed).

However; if an absolute positioned element has no positioned ancestors, it uses the document body, and moves along with page scrolling.

**Note:** Absolute positioned elements are removed from the normal flow, and can overlap elements.

Here is a simple example:

The float Property

The float property is used for positioning and formatting content e.g. let an image float left to the text in a container.

The float property can have one of the following values:

* left - The element floats to the left of its container
* right - The element floats to the right of its container
* none - The element does not float (will be displayed just where it occurs in the text). This is default
* inherit - The element inherits the float value of its parent

In its simplest use, the float property can be used to wrap text around images.

## The CSS Box Model

In CSS, the term "box model" is used when talking about design and layout.

The CSS box model is essentially a box that wraps around every HTML element. It consists of: margins, borders, padding, and the actual content.

### **What is the purpose of the z-index and how is it used?**

The z-index helps to specify the stack order of positioned elements that may overlap one another. The z-index default value is zero and can take on either a positive or negative number.

An element with a higher z-index is always stacked above than a lower index.

## **What are the differences between relative and absolute in CSS?**

### **Relative Position**

An element with position: relative; is positioned relative to its normal position.

Setting the top, right, bottom, and left properties of a relatively-positioned element will cause it to be adjusted away from its normal position. Other content will not be adjusted to fit into any gap left by the element.

## CSS rule

### **Absolute Position**

An element with position: absolute; will cause it to adjust its position with respect to its parent. If no parent is present, then it uses the document body as parent.

## CSS rule

What is the difference between border-box and content-box?

* **content-box** – Default box-sizing property. The width and height properties (and min/max properties) includes only the content. Border and padding are not included
* **border-box** – The width and height properties (and min/max properties) includes content, padding and border.

## **What is the float property and what float do?**

## The float CSS property places an element on the left or right side of its container, allowing text and inline elements to wrap around it.

What does \* { box-sizing: border-box; } do? What are its advantages?

* Make every element in the document include the padding and border in the element's inner dimensions; making it easier to reason about the layout of elements on the page.
* By default, elements have box-sizing: content-box applied, and only the content size is being accounted for.
* box-sizing: border-box changes how the width and height of elements are being calculated, border and padding are also being included in the calculation.
* The height of an element is now calculated by the content's height + vertical padding + vertical border width.
* The width of an element is now calculated by the content's width + horizontal padding + horizontal border width.
* Taking into account paddings and borders as part of our box model resonates better with how designers actually imagine content in grids.

## **What is the difference between padding and margin?**

**1) Margin** is applied to the outside of you element hence effecting how far your element is away from other elements.  
**2) Padding** is applied to the inside of your element hence effecting how far your element's content is away from the border.

## **What is CSS opacity?**

The opacity CSS property sets the opacity of an element. Opacity is the degree to which content behind an element is hidden, and is the opposite of transparency.

## **What is word-wrapping in CSS?**

The word-wrap property in CSS is used to break long word and wrap into the next line. It defines whether to break words when the content exceeds the boundaries of its container.

# [What is the difference between visibility:hidden and display:none?](https://stackoverflow.com/questions/133051/what-is-the-difference-between-visibilityhidden-and-displaynone)

[Ask Question](https://stackoverflow.com/questions/ask)

**visibility:hidden** will keep the element in the page and occupies that space but does not show to the user.

**display:none** will not be available in the page and does not occupy any space.

### **What are synthetic events in React?**

* Synthetic events combine the response of different browser's native events into one API, ensuring that the events are consistent across different browsers.
* The application is consistent regardless of the browser it is running in. Here, **preventDefault**is a synthetic event.

### **What is a higher-order component in React?**

A higher-order component acts as a container for other components. This helps to keep components simple and enables re-usability. They are generally used when multiple components have to use a common logic.

### Arrow Functions:

Arrow functions, introduced in ES6, provides a short way to write functions in JavaScript.

Another significant advantage it offers is the fact that it does not bind its own **this**. In other words, the context inside arrow functions is lexically or statically defined.

### What is Hoisting

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution. Remember that JavaScript only hoists declarations, not initialisation. Let's take a simple example of variable hoisting,

console.log(message); //output : undefined

var message = 'The variable Has been hoisted';

The above code looks like as below to the interpreter,

var message;

console.log(message);

message = 'The variable Has been hoisted';

### What is a callback function

A callback function is a function passed into another function as an argument. This function is invoked inside the outer function to complete an action. Let's take a simple example of how to use callback function

function callbackFunction(name) {

console.log('Hello ' + name);

}

function outerFunction(callback) {

let name = prompt('Please enter your name.');

callback(name);

}

outerFunction(callbackFunction);

### What is an event flow

Event flow is the order in which event is received on the web page. When you click an element that is nested in various other elements, before your click actually reaches its destination, or target element, it must trigger the click event for each of its parent elements first, starting at the top with the global window object. There are two ways of event flow

1. Top to Bottom(Event Capturing)
2. Bottom to Top (Event Bubbling)

### What is event bubbling

Event bubbling is a type of event propagation where the event first triggers on the innermost target element, and then successively triggers on the ancestors (parents) of the target element in the same nesting hierarchy till it reaches the outermost DOM element.

### What is event capturing

Event capturing is a type of event propagation where the event is first captured by the outermost element, and then successively triggers on the descendants (children) of the target element in the same nesting hierarchy till it reaches the innermost DOM element.

### What is the use of stopPropagation method

The stopPropagation method is used to stop the event from bubbling up the event chain. For example, the below nested divs with stopPropagation method prevents default event propagation when clicking on nested div(Div1)

# Prototype in JavaScript

JavaScript is a prototype based language, so, whenever we create a function using JavaScript, JavaScript engine adds a prototype property inside a function.

**Prototype property** is basically an object (also known as Prototype object), where we can attach methods and properties in a prototype object, which enables all the other objects to inherit these methods and properties.

console.log(Object.prototype.constructor === Object); // true

JavaScript is a dynamic language. You can attach new properties to an object at any time as shown below.

function Student() {

this.name = 'John';

this.gender = 'Male';

}

var studObj1 = new Student();

studObj1.age = 15;

alert(studObj1.age); // 15

var studObj2 = new Student();

alert(studObj2.age); // undefined

As you can see in the above example, age property is attached to studObj1 instance. However, studObj2 instance will not have age property because it is defined only on studObj1 instance.

So what to do if we want to add new properties at later stage to a function which will be shared across all the instances?

The answer is **Prototype**.

The prototype is an object that is associated with every functions and objects by default in JavaScript, where function's prototype property is accessible and modifiable and object's prototype property (aka attribute) is not visible.

Every function includes prototype object by default.

The prototype object is special type of enumerable object to which additional properties can be attached to it which will be shared across all the instances of it's constructor function.

So, use prototype property of a function in the above example in order to have age properties across all the objects as shown below.

function Student() {

this.name = 'John';

this.gender = 'M';

}

Student.prototype.age = 15;

var studObj1 = new Student();

alert(studObj1.age); // 15

var studObj2 = new Student();

alert(studObj2.age); // 15

The prototype property is special type of enumerable object which cannot be iterate using for..in or foreach loop.

**Prototype chain:**

**When ever we try to access any object, it looks to object prototype and find that key and value. If that key and value not found it will again look to its own prototype and so on until it reached with null. This process is called Prototype Chain.**

Each object has a private property which holds a link to another object called its **prototype**. That prototype object has a prototype of its own, and so on until an object is reached with null as its prototype. By definition, null has no prototype, and acts as the final link in this **prototype chain**.

# JavaScript | Hoisting

In JavaScript, Hoisting is the default behavior of moving all the declarations at the top of the scope before code execution. Basically, it gives us an advantage that no matter where functions and variables are declared, they are moved to the top of their scope regardless of whether their scope is global or local.

**Note:** JavaScript only hoists declarations, not the initializations.

JavaScript allocates memory for all variables and functions defined in the program before execution.

**Note:** Always remember that in the background the Javascript is first declaring the variable and then initializing them. It is also good to know that variable declarations are processed before any code is executed.

When we talk about ES5, the variable that comes into our minds is var. Hoisting with var is somewhat different as when compared to let/const. Let’s make use of var and see how hoisting works:

// var code (global)

console.log(name); // undefined

var name = 'Mukul Latiyan';

**Let**   
We know that variables declared with let keywords are block scoped not function scoped and hence it is not any kind of problem when it comes to hoisting.

//let example(global)

console.log(name);

let name='Mukul Latiyan'; // ReferencError: name is not defined

Like before, for the var keyword, we expect the output of the log to be undefined. However, since the es6 let doesn’t take kindly on us using undeclared variables, the interpreter explicitly spits out a Reference error. This ensures that we always **declare**our variable first.

**const** behaves similar to let when it comes to hoisting.

# **JavaScript Event Loop**

The **event loop** is the secret behind JavaScript’s asynchronous programming.

The **call stack** is responsible for keeping track of all the operations in line to be executed. Whenever a function is finished, it is popped from the stack.

The **event queue** is responsible for sending new functions to the track for processing. It follows the queue data structure to maintain the correct sequence in which all operations should be sent for execution.

The event loop constantly checks whether or not the call stack is empty. If it is empty, new functions are added from the event queue. If it is not, then the current function call is processed.

The Event Loop has one simple job — to monitor the Call Stack and the Callback Queue. If the Call Stack is empty, the Event Loop will take the first event from the queue and will push it to the Call Stack, which effectively runs it.

### Event Loop

“How is JavaScript asynchronous and single-threaded ?” The short answer is that JavaScript language is single-threaded and the asynchronous behaviour is not part of the JavaScript language itself, rather they are built on top of the core JavaScript language in the browser (or the programming environment) and accessed through the browser APIs.

**Arrow function**

ES6 arrow functions provide you with an alternative way to write a shorter syntax compared to the function expression.

Arrow function does not have its own this. So whenever you call this, it refers to its parent scope.

With an arrow function, this is always inherited from the outer scope.

An arrow function captures the this value of the enclosing context instead of creating its own this context.

Although JavaScript is a single thread language, we have a strong helper which is browser that has ability to manage complex operations. Web API, Callback Queue and Event Loop mechanisms are part of browsers.

## Call Stack:

* Call Stack is the place where the code execution has been tracked.
* Every data in the call stack will be pointed to the memory heap.
* Follows Last In First Out (LIFO).

## Memory Heap:

* Memory heap is the place where the memory is allocated for the variables and functions etc.

## Stack Overflow:

* When the function runs inside and inside, the call stack will be filled and overflows.
* When the stack overflows, Maximum call stack size exceeded error will be thrown.
* The below function runs inside and inside and the stack will be overflowed.

**JavaScript Engine:**

Js engine is divided into two parts.

**Heap Memory**: Where memory allocates for variables.

**Call Stack**: Where code executes. Last In First Out(LIFO).

Diagram

Description automatically generated

**Stack:**This is where all your javascript code gets pushed and executed one by one as the interpreter reads your program, and gets popped out once the execution is done. If your statement is asynchronous: setTimeout, ajax(), promise, or click event, then that code gets forwarded to Event table, this table is responsible for moving your asynchronous code to callback/event queue after specified time.

**Heap:**This is where all the memory allocation happens for your variables, that you have defined in your program.

**Callback Queue:**This is where your asynchronous code gets pushed to, and waits for the execution.

**Event Loop:**Then comes the Event Loop, which keeps running continuously and checks the Main stack, if it has any frames to execute, if not then it checks Callback queue, if Callback queue has codes to execute then it pops the message from it to the Main Stack for the execution.

**Job Queue:**Apart from Callback Queue, browsers have introduced one more queue which is “Job Queue”, reserved only for new Promise() functionality. So when you use promises in your code, you add .then() method, which is a callback method. These `thenable` methods are added to Job Queue once the promise has returned/resolved, and then gets executed.

console.log('Message no. 1: Sync');

setTimeout(function() {

console.log('Message no. 2: setTimeout');

}, 0);

var promise = new Promise(function(resolve, reject) {

resolve();

});

promise.then(function(resolve) {

console.log('Message no. 3: 1st Promise');

})

.then(function(resolve) {

console.log('Message no. 4: 2nd Promise');

});

console.log('Message no. 5: Sync');

setTimeout was pushed to Callback Queue first, then promise was pushed. But this is not the case, the output will be:

// Message no. 1: Sync

// Message no. 5: Sync

// Message no. 3: 1st Promise

// Message no. 4: 2nd Promise

// Message no. 2: setTimeout

# Redux Thunk

With a plain basic Redux store, you can only do simple synchronous updates by dispatching an action. Middleware extends the store's abilities, and lets you write async logic that interacts with the store.

Redux Thunk is middleware that allows you to return functions, rather than just actions, within Redux. This allows for delayed actions, including working with promises.

One of the main use cases for this middleware is for handling actions that might not be synchronous, for example, using axios to send a GET request. Redux Thunk allows us to dispatch those actions asynchronously and resolve each promise that gets returned.

useReducer: The useReducer Hook is the better alternative to the useState hook and is generally more preferred over the useState hook when you have complex state-building logic or when the next state value depends upon its previous value or when the components are needed to be optimized.

The useReducer hook takes three arguments including reducer, initial state, and the function to load the initial state lazily.

### useCallback

Pass an inline callback and an array of dependencies. useCallback will return a memoized version of the callback that only changes if one of the dependencies has changed. This is useful when passing callbacks to optimized child components that rely on reference equality to prevent unnecessary renders (e.g. shouldComponentUpdate).

const memoizedCallback = useCallback(

() => {

doSomething(a, b);

},

[a, b],

);

useCallback and useMemo both expect a function and an array of dependencies. The difference is that useCallback returns its function when the dependencies change while useMemo calls its function and returns the result.

So what is the difference? useCallback returns its function uncalled so you can call it later, while useMemo calls its function and returns the result.

function foo() {

return 'bar';

}

const memoizedCallback = useCallback(foo, []);

const memoizedResult = useMemo(foo, []);

memoizedCallback;

// ƒ foo() {

// return 'bar';

// }

memoizedResult; // 'bar'

memoizedCallback(); // 'bar'

memoizedResult(); // 🔴 TypeError

# **ES7 - New Features**

## Exponentiation Operator

ES7 introduces a new mathematical operator called exponentiation operator. This operator is similar to using Math.pow() method. Exponentiation operator is represented by a double asterisk \*\*. The operator can be used only with numeric values.

2\*\*2 = 4

2\*\*10 = 1024

## Array Includes

The Array.includes() method introduced in ES7 helps to check if an element is available in an array.

# **ES8 - New Features**

## Padding a String

ES8 introduces two string handling functions for padding a string. These functions can be used to add space or any desired set of characters to the beginning and end of a string value.

## String. padStart()

This function pads the current string with a given input string repeatedly from the start, till the current string reaches the given length.

"Hello".padStart(10)

' Hello'

## String.padEnd()

This function pads the current string with a given input string repeatedly from the end, till the current string reaches the specified length.

"Hello".padEnd(10)

'Hello '

## Object:entries() and values()

ES8 introduces the following new methods to the built-in Object type −

* **Object.entries** − The Object.entries() method can be used to access all the properties of an object.
* **Object.values()** − The Object.values() method can be used to access values of all properties of an object.

## Async and Await

Async/Await is a very important feature in ES8.It is a syntactic sugar for Promises in JavaScript. The await keyword is used with promises. This keyword can be used to pause the execution of a function till a promise is settled. The await keyword returns value of the promise if the promise is resolved while it throws an error if the promise is rejected. The await function can only be used inside functions marked as async. A function that is declared using the async keyword always returns a promise.

**What is the purpose of using super constructor with props argument?**

A child class constructor cannot make use of this reference until super() method has been called. The same applies for ES6 sub-classes as well. The main reason of passing props parameter to super() call is to access this.props in your child constructors.

### **What is reconciliation?**

When a component's props or state change, React decides whether an actual DOM update is necessary by comparing the newly returned element with the previously rendered one. When they are not equal, React will update the DOM. This process is called reconciliation.

### **What are fragments?**

It's a common pattern in React which is used for a component to return multiple elements. Fragments let you group a list of children without adding extra nodes to the DOM.

### **What are portals in React?**

Portal is a recommended way to render children into a DOM node that exists outside the DOM hierarchy of the parent component.

ReactDOM.createPortal(child, container)

The first argument is any render-able React child, such as an element, string, or fragment. The second argument is a DOM element.

### **What is prop drilling?**

Prop Drilling is the process by which you pass data from one component of the React Component tree to another by going through other components that do not need the data but only help in passing it around.

# **Redux**

Redux is a predictable state container for JavaScript apps.

It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test.

It serves as a centralized store for state that needs to be used across your entire application.

Redux helps you manage "global" state - state that is needed across many parts of your application.

### **Data flow in a React-Redux application**

Redux follows the unidirectional data flow. It means that your application data will follow in one-way binding data flow.

There are four fundamental concepts that govern the flow of data in React-Redux applications.

**1). Redux store:**The Redux store, simply put, is an object that holds the application state. A redux store can consist of small state objects which are combined into one large object. Any component in the application can easily access this state (store) by hooking up to it through the connect method.

**2). Action creators:** Action creators, as the name suggests, are functions that return actions (objects). Action creators are invoked when the user interacts with the application through its UI (button click, form submission,  etc) or at certain points in a component’s lifecycle (component mounts, component un-mounts, etc).

**3). Actions** Actions are simple objects which conventionally have two properties- type and payload. The type property is usually a string that specifies identifies the action, and the payload is an optional property that contains some data that is required to perform any particular task. The main function of action is to send data from the application to the Redux store.

**4). Reducers:** Reducers are pure functions that update the state of the application in response to actions. Reducers take a previous state and an action as the input and return a modified version of the state. Since the state is immutable, a reducer always returns a new state, which is an updated version of the previous state.

Diagram

Description automatically generated

* The flow of data in a React-Redux application begins at the component level when the user interacts with the application UI. This interaction leads to the action creators dispatching an action.
* When an action is dispatched, it is received by the root reducer of the application and is passed on to all the reducers. Thus, it becomes the reducer’s task to determine if it needs to update the state based on the dispatched action.
* This is checked by using a simple switch statement to filter out the required actions. Each (smaller) reducer in the application accepts the dispatched action and if the type of the dispatched action matches, it returns a newly updated state.
* It is essential to note here that the state never actually changes in redux. Instead, the reducer always generates a new state which is a copy of the old state, but with some modifications.
* The store then informs the component about the new state which in turn retrieves the updated state and re-renders the component.
* Another important observation here is that flow of data in a React-Redux application is unidirectional, i.e., it only goes in one direction.

### **Actions**

An **action** is a plain JavaScript object that has a type field.

**You can think of an action as an event that describes something that happened in the application.**

### **Reducers**

A **reducer** is a function that receives the current state and an action object, decides how to update the state if necessary, and returns the new state.

**(state, action) => newState**

* They must not do any asynchronous logic, calculate random values, or cause other "side effects"
* They should only calculate the new state value based on the state and action arguments

### **Dispatch**

The Redux store has a method called dispatch. **The only way to update the state is to call store.dispatch() and pass in an action object**. The store will run its reducer function and save the new state value inside, and we can call getState() to retrieve the updated value:

### **Selectors**

**Selectors** are functions that know how to extract specific pieces of information from a store state value. As an application grows bigger, this can help avoid repeating logic as different parts of the app need to read the same data:

### **redux-thunk**

Redux-thunk

**getDerivedStateFromProps**

getDerivedStateFromProps is **invoked right before calling the render method**, both on the initial mount and on subsequent updates. It should return an object to update the state, or null to update nothing. This method exists for rare use cases where the state depends on changes in props over time.

### getSnapshotBeforeUpdate()

getSnapshotBeforeUpdate() is invoked right before the most recently rendered output is committed to e.g. the DOM. It enables your component to capture some information from the DOM (e.g. scroll position) before it is potentially changed. Any value returned by this lifecycle method will be passed as a parameter to componentDidUpdate().

This use case is not common, but it may occur in UIs like a chat thread that need to handle scroll position in a special way.

**getSnapshotBeforeUpdate(prevProps, prevState) {**

**// Are we adding new items to the list?**

**// Capture the scroll position so we can adjust scroll later.**

**if (prevProps.list.length < this.props.list.length) {**

**const list = this.listRef.current;**

**return list.scrollHeight - list.scrollTop;**

**}**

**return null;**

**}**

## Controlled Components

A [controlled component](https://reactjs.org/docs/forms.html#controlled-components) is a component that renders form elements and controls them by keeping the form data in the component's state.

In a controlled component, the form element's data is handled by the React component (not DOM) and kept in the component's state. A controlled component basically overrides the default behavior of the HTML form elements.

const { useState } from 'react';

function Controlled () {

const [email, setEmail] = useState();

const handleInput = (e) => setEmail(e.target.value);

return <input type="text" value={email} onChange={handleInput} />;

}

## Uncontrolled Components

An [uncontrolled component](https://reactjs.org/docs/uncontrolled-components.html) is a component that renders form elements, where the form element's data is handled by the DOM (default DOM behavior). To access the input's DOM node and extract its value you can use a [ref](https://reactjs.org/docs/refs-and-the-dom.html).

const { useRef } from 'react';

function Example () {

const inputRef = useRef(null);

return <input type="text" defaultValue="bar" ref={inputRef} />

}

# Refs

Refs provide a way to access DOM nodes or React elements created in the render method.

# Render Props

The term “render prop” refers to a technique for sharing code between React components using a prop whose value is a function.

A component with a render prop takes a function that returns a React element and calls it instead of implementing its own render logic.

<DataProvider render={data => (

<h1>Hello {data.target}</h1>

)}/>

The position Property

The position property specifies the type of positioning method used for an element.

There are five different position values:

* static
* relative
* fixed
* absolute
* sticky

## position: static;

HTML elements are positioned static by default.

Static positioned elements are not affected by the top, bottom, left, and right properties.

## position: relative;

An element with position: relative; is positioned relative to its normal position.

Setting the top, right, bottom, and left properties of a relatively-positioned element will cause it to be adjusted away from its normal position. Other content will not be adjusted to fit into any gap left by the element.

## position: fixed;

An element with position: fixed; is positioned relative to the viewport, which means it always stays in the same place even if the page is scrolled. The top, right, bottom, and left properties are used to position the element.

A fixed element does not leave a gap in the page where it would normally have been located.

## position: absolute;

An element with position: absolute; is positioned relative to the nearest positioned ancestor (instead of positioned relative to the viewport, like fixed).

However; if an absolute positioned element has no positioned ancestors, it uses the document body, and moves along with page scrolling.

**Note:** Absolute positioned elements are removed from the normal flow, and can overlap elements.

Here is a simple example:

The float Property

The float property is used for positioning and formatting content e.g. let an image float left to the text in a container.

The float property can have one of the following values:

* left - The element floats to the left of its container
* right - The element floats to the right of its container
* none - The element does not float (will be displayed just where it occurs in the text). This is default
* inherit - The element inherits the float value of its parent

In its simplest use, the float property can be used to wrap text around images.

## The CSS Box Model

In CSS, the term "box model" is used when talking about design and layout.

The CSS box model is essentially a box that wraps around every HTML element. It consists of: margins, borders, padding, and the actual content.

### **What is the purpose of the z-index and how is it used?**

The z-index helps to specify the stack order of positioned elements that may overlap one another. The z-index default value is zero and can take on either a positive or negative number.

An element with a higher z-index is always stacked above than a lower index.

## **What are the differences between relative and absolute in CSS?**

### **Relative Position**

An element with position: relative; is positioned relative to its normal position.

Setting the top, right, bottom, and left properties of a relatively-positioned element will cause it to be adjusted away from its normal position. Other content will not be adjusted to fit into any gap left by the element.

## CSS rule

### **Absolute Position**

An element with position: absolute; will cause it to adjust its position with respect to its parent. If no parent is present, then it uses the document body as parent.

## CSS rule

What is the difference between border-box and content-box?

* **content-box** – Default box-sizing property. The width and height properties (and min/max properties) includes only the content. Border and padding are not included
* **border-box** – The width and height properties (and min/max properties) includes content, padding and border.

## **What is the float property and what float do?**

## The float CSS property places an element on the left or right side of its container, allowing text and inline elements to wrap around it.

What does \* { box-sizing: border-box; } do? What are its advantages?

* Make every element in the document include the padding and border in the element's inner dimensions; making it easier to reason about the layout of elements on the page.
* By default, elements have box-sizing: content-box applied, and only the content size is being accounted for.
* box-sizing: border-box changes how the width and height of elements are being calculated, border and padding are also being included in the calculation.
* The height of an element is now calculated by the content's height + vertical padding + vertical border width.
* The width of an element is now calculated by the content's width + horizontal padding + horizontal border width.
* Taking into account paddings and borders as part of our box model resonates better with how designers actually imagine content in grids.

## **What is the difference between padding and margin?**

**1) Margin** is applied to the outside of you element hence effecting how far your element is away from other elements.  
**2) Padding** is applied to the inside of your element hence effecting how far your element's content is away from the border.

## **What is CSS opacity?**

he opacity CSS property sets the opacity of an element. Opacity is the degree to which content behind an element is hidden, and is the opposite of transparency.

## **What is word-wrapping in CSS?**

The word-wrap property in CSS is used to break long word and wrap into the next line. It defines whether to break words when the content exceeds the boundaries of its container.

# [What is the difference between visibility:hidden and display:none?](https://stackoverflow.com/questions/133051/what-is-the-difference-between-visibilityhidden-and-displaynone)

[Ask Question](https://stackoverflow.com/questions/ask)

**visibility:hidden** will keep the element in the page and occupies that space but does not show to the user.

**display:none** will not be available in the page and does not occupy any space.

### **What are synthetic events in React?**

* Synthetic events combine the response of different browser's native events into one API, ensuring that the events are consistent across different browsers.
* The application is consistent regardless of the browser it is running in. Here, **preventDefault**is a synthetic event.

### **What is a higher-order component in React?**

A higher-order component acts as a container for other components. This helps to keep components simple and enables re-usability. They are generally used when multiple components have to use a common logic.

### Arrow Functions:

Arrow functions, introduced in ES6, provides a short way to write functions in JavaScript.

Another significant advantage it offers is the fact that it does not bind its own **this**. In other words, the context inside arrow functions is lexically or statically defined.

### What is Hoisting

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution. Remember that JavaScript only hoists declarations, not initialisation. Let's take a simple example of variable hoisting,

console.log(message); //output : undefined

var message = 'The variable Has been hoisted';

The above code looks like as below to the interpreter,

var message;

console.log(message);

message = 'The variable Has been hoisted';

### What is a callback function

A callback function is a function passed into another function as an argument. This function is invoked inside the outer function to complete an action. Let's take a simple example of how to use callback function

function callbackFunction(name) {

console.log('Hello ' + name);

}

function outerFunction(callback) {

let name = prompt('Please enter your name.');

callback(name);

}

outerFunction(callbackFunction);

### What is an event flow

Event flow is the order in which event is received on the web page. When you click an element that is nested in various other elements, before your click actually reaches its destination, or target element, it must trigger the click event for each of its parent elements first, starting at the top with the global window object. There are two ways of event flow

1. Top to Bottom(Event Capturing)
2. Bottom to Top (Event Bubbling)

### What is event bubbling

Event bubbling is a type of event propagation where the event first triggers on the innermost target element, and then successively triggers on the ancestors (parents) of the target element in the same nesting hierarchy till it reaches the outermost DOM element.

### What is event capturing

Event capturing is a type of event propagation where the event is first captured by the outermost element, and then successively triggers on the descendants (children) of the target element in the same nesting hierarchy till it reaches the innermost DOM element.

### What is the use of stopPropagation method

The stopPropagation method is used to stop the event from bubbling up the event chain. For example, the below nested divs with stopPropagation method prevents default event propagation when clicking on nested div(Div1)

# Prototype in JavaScript

JavaScript is a prototype based language, so, whenever we create a function using JavaScript, JavaScript engine adds a prototype property inside a function.

**Prototype property** is basically an object (also known as Prototype object), where we can attach methods and properties in a prototype object, which enables all the other objects to inherit these methods and properties.

console.log(Object.prototype.constructor === Object); // true

JavaScript is a dynamic language. You can attach new properties to an object at any time as shown below.

function Student() {

this.name = 'John';

this.gender = 'Male';

}

var studObj1 = new Student();

studObj1.age = 15;

alert(studObj1.age); // 15

var studObj2 = new Student();

alert(studObj2.age); // undefined

As you can see in the above example, age property is attached to studObj1 instance. However, studObj2 instance will not have age property because it is defined only on studObj1 instance.

So what to do if we want to add new properties at later stage to a function which will be shared across all the instances?

The answer is **Prototype**.

The prototype is an object that is associated with every functions and objects by default in JavaScript, where function's prototype property is accessible and modifiable and object's prototype property (aka attribute) is not visible.

Every function includes prototype object by default.

The prototype object is special type of enumerable object to which additional properties can be attached to it which will be shared across all the instances of it's constructor function.

So, use prototype property of a function in the above example in order to have age properties across all the objects as shown below.

function Student() {

this.name = 'John';

this.gender = 'M';

}

Student.prototype.age = 15;

var studObj1 = new Student();

alert(studObj1.age); // 15

var studObj2 = new Student();

alert(studObj2.age); // 15

The prototype property is special type of enumerable object which cannot be iterate using for..in or foreach loop.

**Prototype chain:**

**When ever we try to access any object, it looks to object prototype and find that key and value. If that key and value not found it will again look to its own prototype and so on until it reached with null. This process is called Prototype Chain.**

Each object has a private property which holds a link to another object called its **prototype**. That prototype object has a prototype of its own, and so on until an object is reached with null as its prototype. By definition, null has no prototype, and acts as the final link in this **prototype chain**.

# JavaScript | Hoisting

In JavaScript, Hoisting is the default behavior of moving all the declarations at the top of the scope before code execution. Basically, it gives us an advantage that no matter where functions and variables are declared, they are moved to the top of their scope regardless of whether their scope is global or local.

**Note:** JavaScript only hoists declarations, not the initializations.

JavaScript allocates memory for all variables and functions defined in the program before execution.

**Note:** Always remember that in the background the Javascript is first declaring the variable and then initializing them. It is also good to know that variable declarations are processed before any code is executed.

When we talk about ES5, the variable that comes into our minds is var. Hoisting with var is somewhat different as when compared to let/const. Let’s make use of var and see how hoisting works:

// var code (global)

console.log(name); // undefined

var name = 'Mukul Latiyan';

**Let**   
We know that variables declared with let keywords are block scoped not function scoped and hence it is not any kind of problem when it comes to hoisting.

//let example(global)

console.log(name);

let name='Mukul Latiyan'; // ReferencError: name is not defined

Like before, for the var keyword, we expect the output of the log to be undefined. However, since the es6 let doesn’t take kindly on us using undeclared variables, the interpreter explicitly spits out a Reference error. This ensures that we always **declare**our variable first.

**const** behaves similar to let when it comes to hoisting.

## JavaScript hoisting

When you execute a piece of JavaScript code, the JavaScript engine creates the [global execution context](https://www.javascripttutorial.net/javascript-execution-context/).

The global execution context has two phases: creation and execution.

During the creation phase, the JavaScript engine moves the variable and function declarations to the top of your code. This feature is known as hoisting in JavaScript.

## Variable hoisting

Variable hoisting means the JavaScript engine moves the [variable declarations](https://www.javascripttutorial.net/javascript-variable-scope/) to the top of the script. The following example declares the counter variable and sets its value to 1:

console.log(counter); // undefined

var counter = 1;

### **The let keyword**

The following declares the variable counter with the [let](https://www.javascripttutorial.net/es6/javascript-let/) keyword:

console.log(counter); //"ReferenceError: Cannot access 'counter' before initialization

let counter = 1;

The error message explains that the counter variable is already in the heap memory. However, it hasn’t initialized.

Behind the scenes, the JavaScript engine hoists the variable declarations that use the let keyword. However, it doesn’t initialize those variables. Notice that if you access a variable that doesn’t exist, the JavaScript will throw a different error:

# **Arrow functions**

Unlike regular functions, arrow functions do not have their own this. In the case of an arrow function, this refers to the values of this in the environment the arrow function is defined in (i.e. "outside" the arrow function) and that remains the same throughout the lifecycle of the function and is always bound to the value of this in the closest non-arrow parent function.

Arguments objects are not available in arrow functions, but are available in regular functions.

# What is the Temporal Dead Zone (TDZ) in JavaScript?

difference between let / const and var is that if you access var before it's declared, it is undefined. But if you do the same for let and const, they throw a ReferenceError.

console.log(varNumber); // undefined

console.log(letNumber); // ReferenceError letNumber is not defined

var varNumber = 1;

let letNumber = 1;

They throw the error all because of the Temporal Dead Zone.

Temporal Dead Zone describe the state where variables are un-reachable. They are in scope, but they aren't declared.

The let and constvariables exist in the TDZ from the start of their enclosing scope until they are declared.

{

// This is the temporal dead zone for the age variable!

// This is the temporal dead zone for the age variable!

// This is the temporal dead zone for the age variable!

// This is the temporal dead zone for the age variable!

let age = 25; // Whew, we got there! No more TDZ

console.log(age);

}

## Given a string, reverse each word in the sentence

For example Welcome to this Javascript Guide! should be become emocleW ot siht tpircsavaJ !ediuG

var string = "Welcome to this Javascript Guide!";

// Output becomes !ediuG tpircsavaJ siht ot emocleW

var reverseEntireSentence = reverseBySeparator(string, "");

// Output becomes emocleW ot siht tpircsavaJ !ediuG

var reverseEachWord = reverseBySeparator(reverseEntireSentence, " ");

function reverseBySeparator(string, separator) {

return string.split(separator).reverse().join(separator);

}

# **Mutable**

Mutable is a type of variable that can be changed. In JavaScript, only objects and arrays are mutable, not primitive values.

A **mutable object** is an object whose state can be modified after it is created.

**Immutables are the objects** whose state cannot be changed once the object is created.

**Strings and Numbers are Immutable**.

Let’s say you have some things saved to variables, and you want to create copies.

var num1 = 42;

var str1 = 'Hello';

var arr1 = [1, 2, 3];

var obj1 = {greeting: 'hello', name: 'world'};

With numbers and strings, assigning the original variable to a new variable creates a new item.

// Copy items

var num2 = num1;

var str2 = str1;

// Update the copies

num2 = num2 - 10;

str2 += ' world';

// Logs 42

console.log(num1);

// Logos "Hello"

console.log(str1);

Objects and arrays work a big differently. Assigning the original variable to a new one creates a reference to the original rather than a new item.

// Copy items

var arr2 = arr1;

var obj2 = obj1;

// Update the copies

arr2.push(4, 5, 6);

obj2.name = 'universe';

// Logs [1, 2, 3, 4, 5, 6]

console.log(arr1);

// Logs {greeting: 'hello', name: 'universe'}

console.log(obj1);

Even though the copies were the ones that were modified, the original arrays and objects were also updated. The new variables point back to the original.

If an item is mutable, modifying the copy also modifies the original. If it’s immutable, modifying the copy does not affect the original.

# **Var, Let, and Const – What's the Difference?**

**var** declarations are globally scoped or function scoped while **let** and **const are block scoped**. var variables can be updated and re-declared within its scope; let variables can be updated but not re-declared; const variables can neither be updated nor re-declared. They are all hoisted to the top of their scope.

* **var** declarations are globally scoped or function scoped while **let** and **const** are block scoped.
* **var** variables can be updated and re-declared within its scope.
* **let** variables can be updated but not re-declared.
* **const** variables can neither be updated nor re-declared.
* They are all hoisted to the top of their scope. But while **var** variables are initialized with **undefined**.
* **let** and **const** variables are not initialized.
* While **var** and **let** can be declared without being initialized, **const** must be initialized during declaration.

# **Difference between CSS Grid and CSS Flexbox**

Grid: CSS Grid Layout, is a two-dimensional grid-based layout system with rows and columns, making it easier to design web pages without having to use floats and positioning. Like tables, grid layout allow us to align elements into columns and rows.

To get started you have to define a container element as a grid with **display: grid**, set the column and row sizes with grid-template-columns and grid-template-rows, and then place its child elements into the grid with grid-column and grid-row.

.main{

display: grid;

display: grid;

grid: auto auto / auto auto auto auto;

grid-gap: 10px;

background-color: green;

padding: 10px;

}

.gfg {

background-color: rgb(255, 255, 255);

text-align: center;

padding: 25px 0;

font-size: 30px;

}

Flexbox: The CSS Flexbox offers a one-dimensional layout. It is helpful in allocating and aligning the space among items in a container (made of grids). It works with all kinds of display devices and screen sizes.

To get started you have to define a container element as a grid with display: flex;

# **CSS Layout - The display Property**

The display property specifies how an element is displayed.

Every HTML element has a default display value depending on what type of element it is. The default display value for most elements is block or inline.

**Block-level Elements**

A block-level element always starts on a new line and takes up the full width available (stretches out to the left and right as far as it can).

**Inline Elements**

An inline element does not start on a new line and only takes up as much width as necessary.

This is an inline <span> element inside a paragraph.

Examples of inline elements:

<span>

<a>

<img>

**Inline:** Displays an element as an inline element (like <span>). An inline element does not start on a new line and only takes up as much width as necessary.

Any height and width properties will have no effect.

**Block:** Displays an element as a block element (like <p>). It starts on a new line, and takes up the whole width.

**inline-block:** Displays an element as an inline-level block container. The element itself is formatted as an inline element, but you can apply height and width values.

**position**

The position CSS property sets how an element is positioned in a document. The top, right, bottom, and left properties determine the final location of positioned elements.

**Position: relative;**

An element with position: relative; is positioned relative to its normal position.

Setting the top, right, bottom, and left properties of a relatively-positioned element will cause it to be adjusted away from its normal position. Other content will not be adjusted to fit into any gap left by the element.

**Position: fixed;**

An element with position: fixed; is positioned relative to the viewport, which means it always stays in the same place even if the page is scrolled. The top, right, bottom, and left properties are used to position the element.

A fixed element does not leave a gap in the page where it would normally have been located.

**Position: absolute;**

An element with position: absolute; is positioned relative to the nearest positioned ancestor.

However; if an absolute positioned element has no positioned ancestors, it uses the document body, and moves along with page scrolling.

Note: Absolute positioned elements are removed from the normal flow, and can overlap elements.

# **Currying:** Currying is a transformation of functions that translates a function from callable as f(a, b, c) into callable as f(a)(b)(c).

Currying doesn’t call a function. It just transforms it.

Currying is when you break down a function that takes multiple arguments into a series of functions that each take only one argument.

function foo(a) {

return function (b) {

return function (c) {

return a + b + c;

};

};

}

let res = foo(1)(2)(3);

console.log(res); //6

**Synthetic Events**

React has its own event handling system which is very similar to handling events on DOM elements. The react event handling system is known as Synthetic Events.

Handling events with react have some differences from handling events on DOM. These are:

1). React events are named as camelCase instead of lowercase.

2). With JSX, a function is passed as the event handler instead of a string.

3). In react, we cannot return **false** to prevent the **default** behavior. We must call **preventDefault** event explicitly to prevent the default behavior.

**Event declaration in plain HTML:**

<button onclick="showMessage()">

Hello JavaTpoint

</button>

**Event declaration in React:**

<button onClick={showMessage}>

Hello JavaTpoint

</button>

In plain HTML, to prevent the default link behavior of opening a new page, we can write:

<a href="#" onclick="console.log('You had clicked a Link.'); return false">

Click\_Me

</a>

In React, we can write it as:

function ActionLink() {

function handleClick(e) {

e.preventDefault();

console.log('You had clicked a Link.');

}

return (

<a href="#" onClick={handleClick}>

Click\_Me

</a>

);

}

In the above example, e is a Synthetic Event which defines according to the W3C spec.

**Angular**

### What are the key components of Angular?

Angular has the below key components,

* 1. **Component:** These are the basic building blocks of angular application to control HTML views.
  2. **Modules:** An angular module is set of angular basic building blocks like component, directives, services etc. An application is divided into logical pieces and each piece of code is called as "module" which perform a single task.
  3. **Templates:** This represent the views of an Angular application.
  4. **Services:** It is used to create components which can be shared across the entire application.
  5. **Metadata:** This can be used to add more data to an Angular class.

### **What are the differences between Component and Directive?**

In a short note, A component(@component) is a directive-with-a-template.

Some of the major differences are mentioned in a tabular form

| **Component** | **Directive** |
| --- | --- |
| To register a component we use @Component meta-data annotation | To register directives we use @Directive meta-data annotation |
| Components are typically used to create UI widgets | Directive is used to add behavior to an existing DOM element |
| Component is used to break up the application into smaller components | Directive is use to design re-usable components |
| Only one component can be present per DOM element | Many directives can be used per DOM element |
| @View decorator or templateurl/template are mandatory | Directive doesn't use View |

### **What are lifecycle hooks available?**

The description of each lifecycle method is as below,

1. **ngOnChanges:** When the value of a data bound property changes, then this method is called.
2. **ngOnInit:** This is called whenever the initialization of the directive/component after Angular first displays the data-bound properties happens.
3. **ngDoCheck:** This is for the detection and to act on changes that Angular can't or won't detect on its own.
4. **ngAfterContentInit:** This is called in response after Angular projects external content into the component's view.
5. **ngAfterContentChecked:** This is called in response after Angular checks the content projected into the component.
6. **ngAfterViewInit:** This is called in response after Angular initializes the component's views and child views.
7. **ngAfterViewChecked:** This is called in response after Angular checks the component's views and child views.
8. **ngOnDestroy:** This is the cleanup phase just before Angular destroys the directive/component.

### What is a data binding?

Data binding is a core concept in Angular and allows to define communication between a component and the DOM, making it very easy to define interactive applications without worrying about pushing and pulling data. There are four forms of data binding(divided as 3 categories) which differ in the way the data is flowing.

* 1. **From the Component to the DOM:**

**Interpolation:** {{ value }}: Adds the value of a property from the component

<li>Name: {{ user.name }}</li>

<li>Address: {{ user.address }}</li>

**Property binding:** [property]=”value”: The value is passed from the component to the specified property or simple HTML attribute

<input type="email" [value]="user.email">

* 1. **From the DOM to the Component:** **Event binding: (event)=”function”:** When a specific DOM event happens (eg.: click, change, keyup), call the specified method in the component

<button (click)="logout()"></button>

* 1. **Two-way binding:** **Two-way data binding:** [(ngModel)]=”value”: Two-way data binding allows to have the data flow both ways. For example, in the below code snippet, both the email DOM input and component email property are in sync

<input type="email" [(ngModel)]="user.email">

### What is a service?

A service is used when a common functionality needs to be provided to various modules. Services allow for greater separation of concerns for your application and better modularity by allowing you to extract common functionality out of components.

Let's create a repoService which can be used across components,

import { Injectable } from '@angular/core';

import { Http } from '@angular/http';

@Injectable({ // The Injectable decorator is required for dependency injection to work

// providedIn option registers the service with a specific NgModule

providedIn: 'root', // This declares the service with the root app (AppModule)

})

export class RepoService{

constructor(private http: Http){

}

fetchAll(){

return this.http.get('https://api.github.com/repositories');

}

}

The above service uses Http service as a dependency.

# **HTML5 - New Tags (Elements)**

|  |  |
| --- | --- |
| <audio> | Defines an audio file. |
| <canvas> | This is used for rendering dynamic bitmap graphics on the fly, such as graphs or games. |
| <footer> | Represents a footer for a section and can contain information about the author, copyright information, et cetera. |
| <header> | Represents a group of introductory or navigational aids. |
| <nav> | Represents a section of the document intended for navigation. |
| <article> | Represents an independent piece of content of a document, such as a blog entry or newspaper article |
| <section> | Represents a generic document or application section |
| <time> | Represents a date and/or time. |
| <video> | Defines a video file. |

## New types for <input> tag

|  |  |
| --- | --- |
| **Type** | **Description** |
| color | Color selector, which could be represented by a wheel or swatch picker |
| date | Selector for calendar date |
| datetime-local | Date and time display, with no setting or indication for time zones |
| datetime | Full date and time display, including a time zone. |
| email | Input type should be an email. |
| month | Selector for a month within a given year |
| number | A field containing a numeric value only |
| range | Numeric selector within a range of values, typically visualized as a slider |
| search | Term to supply to a search engine. For example, the search bar atop a browser. |
| tel | Input type should be telephone number. |
| time | Time indicator and selector, with no time zone information |
| url | Input type should be URL type. |
| week | Selector for a week within a given year |

# **Web Storage**

The **Web Storage API** provides mechanisms by which browsers can store key/value pairs.

**sessionStorage** maintains a separate storage area for each given origin that's available for the duration of the page session (as long as the browser is open, including page reloads and restores).

* Stores data only for a session, meaning that the data is stored until the browser (or tab) is closed.
* Data is never transferred to the server.
* Storage limit is larger than a cookie (at most 5MB).

**localStorage** does the same thing, but persists even when the browser is closed and reopened.

* + Stores data with no expiration date, and gets cleared only through JavaScript, or clearing the Browser cache / Locally Stored Data.
  + Storage limit is the maximum amongst the two.

# **Cookie:** A cookie is a small piece of information left on a visitor's computer by a website, via a web browser.

# Cookies are used to personalize a user’s web experience with a website. It may contain the user’s preferences or inputs when accessing that website. A user can customize their web browser to accept, reject, or delete cookies.

# Cookies can be set and modified at the server level using the Set-Cookie HTTP header, or with JavaScript using document.cookie.

**Ways to increase your page speed:**

1. Enable compression. ...
2. Minify CSS, JavaScript, and HTML. ...
3. Reduce redirects. ...
4. Remove render-blocking JavaScript. ...
5. Leverage browser caching. ...
6. Improve server response time. ...
7. Use a content distribution network. ...
8. Optimize images.

**Minify CSS, JavaScript, and HTML**

By optimizing your code (including removing spaces, commas, and other unnecessary characters), you can dramatically increase your page speed. Also remove code comments, formatting, and unused code. Google recommends using CSSNano and UglifyJS.

## Single-Page Applications (SPA)

An SPA (Single-page application) is a web app implementation that loads only a single web document, and then updates the body content of that single document via JavaScript APIs such as XMLHttpRequest and Fetch when different content is to be shown.

This therefore allows users to use websites without loading whole new pages from the server, which can result in performance gains and a more dynamic experience, with some tradeoff disadvantages such as SEO, more effort required to maintain state, implement navigation, and do meaningful performance monitoring.

A single-page application (SPA) is a website design approach where each new page's content is served not from loading new HTML pages but generated dynamically through JavaScript's ability to manipulate the DOM elements on the existing page itself.

An SPA approach allows the user to continue consuming and interacting with the page while new elements are being updated or fetched, and can result in much faster interactions and content reloading.

# **IIFE**

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

(function () {

statements

})();

Copy to Clipboard

It is a design pattern which is also known as a Self-Executing Anonymous Function and contains two major parts:

1). The first is the anonymous function with lexical scope enclosed within the Grouping Operator (). This prevents accessing variables within the IIFE idiom as well as polluting the global scope.

2). The second part creates the immediately invoked function expression () through which the JavaScript engine will directly interpret the function.