

REACT

1) What is React?

- 1) React is an open-source JavaScript library for building user interfaces (UIs).
- 2) It was developed by Facebook.
- 3) React is a component based library. Components are re-usable.
- 4) React is used for developing dynamic web applications, react can update and render data efficiently when the underlying data changes.
- 5) React uses virtual representation of DOM to efficiently update and render data in UI. Instead of directly manipulating the actual DOM, React calculates the minimal set of changes needed and updates the virtual DOM, which is efficiently applied to actual DOM.
- 6) React utilizes JSX (JavaScript XML), It allows developers to write HTML-like syntax within JavaScript code. This helps developers for developing UI fast and easily.
- 7) React follows a uni-directional data flow pattern, where data flows from parent components to child components via props.
- 8) React can do only one thing that is it can render data fastly and efficiently in webpages.
- 9) React provides the facility of integrating with other libraries, frameworks and languages easily to develop additional functionality.
- 10) React works as a declarative library.
- 11) React can develop single page applications by using react-router-dom library.

2) DOM Manipulation

In Javascript

```
<script>
  var h1 = document.createElement('h1')
  h1.innerHTML = 'Hello react'
  var root = document.getElementById('root')
  root.appendChild(h1)
</script>
```

In React

```
<script>
  var h1 = React.createElement('h1', {}, 'hello world')
  var root = ReactDOM.createRoot(document.getElementById('root'))
  root.render(h1)
</script>
```

3) How to install react

Use React and ReactDOM library

1) How to use react by using CDN links

```
<script crossorigin src="https://unpkg.com/react@18/umd/react.development.js"></script>
```

```
<script crossorigin src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"></script>
```

2) How to install react by using create-react-app tool

=> Here by default webpack will be installed (This is also a build tool)

=> `npx create-react-app app-name`

=> `cd app-name`

=> `npm start`

3) How to install react by using vite tool

- Vite is one of the most popular build tools out there in market.

- Vite is a build tool that aims to provide a faster development environment.

- Vite makes Hot module reload.

- Vite also allows us to select the framework we want to work.

=> `npm create vite@latest.`

=> `npm i`

=> `npm run dev`

4) Folder structure in react

node_modules

- "node_modules" contains external libraries and packages that your React application depends on. These modules are managed by a package manager like npm.

- With a package manager like npm, you can use a single command (`npm install`) to automatically download and install all the dependencies specified in your project's package.json file.

- No need to send this file to github, it can be generated by using `npm install` command. So put this folder in .gitignore file.

public folder

- In React.js, the "public" folder is a special directory that contains static assets and files that should be publicly accessible to the client-side of your application.

- The purpose of the "public" folder is to provide a place for assets that don't need to go through the build process. Unlike the "src" folder, which contains the source code of your React application and gets processed by Webpack and Babel, the "public" folder assets are copied as-is to the build output directory during the build process.

index.html file:

- This is the main HTML file that serves as the entry point to your React application. It contains a `<div>` element with an `id='root'`, which acts as the mount point for the React components.

src folder

- It's important to note that the "src" folder contains the source code that will be processed and bundled by build tools like webpack, vite and others to create a production-ready version of your application
- During development, you work in the "src" folder, and the build process outputs the optimized and minified code into a separate "build" or "dist" folder that you can deploy.

- App.js file:

It is the root/ parent component created by default.

- main/index.js file:

This is the linking file, in this file we link `root(App)` component to index.html file.

App component will be rendered in div element of index.html file.

package.json

- It contains a dependencies that lists all the external libraries and packages that your project depends on.
- When you or someone else clones your project and runs `npm install`, npm reads the package.json file and installs all the listed dependencies along with their specified versions.

package.lock.json

- To summarize, the package-lock.json file and the package.json file work together to manage dependencies, ensure version consistency, provide reproducibility, and enhance the security and integrity of your React.js project.
- It's important to commit both the package-lock.json and the package.json files to version control systems like Git, so others can have a consistent development environment when working on the project.

.gitignore

- The .gitignore file in a React project is used to specify which files and directories should be ignored by the version control system, such as Git.

In a React project, some typical entries you might find in the .gitignore file include:

- Build Output: Ignore the output directories where the bundled and minified code is generated. These directories are usually named "build" or "dist."
- Node_Modules: Ignore the "node_modules" directory, which contains all installed dependencies. Since dependencies can be easily re-installed using the package manager there is no need to include them in the version control system.

- Environment Variables: Ignore files that store sensitive information, such as API keys, passwords, or configuration files specific to your development environment.

5) Components

1) What is component?

- 1) A component is a reusable block of code, it contains a piece of user interface (UI).
- 2) User Interface (UI) is a collection of components in react.
- 3) Components are re-usable.
- 4) Components can maintain state in it and can receive props from parent and return JSX.
- 5) Components return JSX, JSX contains UI.
- 6) Components can render dynamic data in UI by using props and state.
- 7) Components can be composed together by nesting them within each other or passing them as props to other components, creating a hierarchy of UI elements. This modular approach allows for reusability, maintainability, and separation of concerns in React applications.

2) Types of components in react?

In React there are two types of components.

1) Class component:

- A class component is a type of component that is defined using ES6 classes and extends the `React.Component` class from React.
- It can maintain state in it and can receive props from parent and return JSX.
- It has a constructor where the initial state can be defined.
- The render method is a compulsory method in a class component. Render method returns the JSX that defines the component's UI.
- In class components we have to bind "this" keyword when we handle with events.
- Class components can't be understood by browser which need to be converted into pure javascript by using Babel (Transpiler).

2) Functional component:

A functional component is a type of component that is defined using JavaScript function.

- It can maintain state in it and can receive props from parent and return JSX.
- The function body returns the JSX that defines the component's UI.
- No need of constructor and `render()` method and "this" key word in functional components.
- Functional component is a javascript function which can be understood by browser easily. No need of conversion.

- After introduction of hooks functional components are not stateless. By using useSate hook we can maintain state in functional component.
- Functional components are simpler and more lightweight compared to class components.

6) JSX (Javascript & XML)

- JSX (JavaScript & XML) is used in React.js for defining and rendering the UI.
 - It allows developers to write HTML-like syntax within JavaScript code. This helps developers for developing UI fast and easily.
 - Under the hood, JSX is transformed into regular JavaScript code by a process called transpiling. Tools like Babel are used to transpile JSX code into JavaScript code that the browser can understand.
 - The transpiled code uses `React.createElement()` function to create and update the actual DOM elements.
 - JSX allows you to embed JavaScript expressions within curly braces `{}`.
 - Using of JSX is a common practice in React development, it is not mandatory.
- React can also work without JSX by using the `React.createElement()` function directly.

- Take a look at the below code:

```
let jsx = <h1>This is JSX</h1>
```

- This is simple JSX code in React. But the browser does not understand this JSX because it's not valid JavaScript code. This is because we're assigning an HTML tag to a variable that is not a string but just HTML code.

- So to convert it to browser understandable JavaScript code, we use a tool like Babel which is a JavaScript transpiler.

The `React.createElement` has the following syntax:

```
React.createElement(type, [props], [...children])
```

Let's look at the parameters of the `createElement` function.

*type can be an HTML tag like `h1`, `div` or it can be a React component.

*props are the attributes you want the element to have.

*children contain other HTML tags or can be a component.

- When we have two or more jsx sibling elements, JSX will through an error.

```
const App = () => {
  return (
    <p>This is first JSX Element!</p>
```

```
<p>This is another JSX Element</p>
```

```
);
```

```
};
```

Here We will get an error

Solutions to resolve issue:

1) To make it work, the obvious solution is to wrap both of them in some parent element, most probably a div.

2) You can try returning it as an array as shown below:

```
const App = () => {  
  return (  
    [<p>This is first JSX Element!</p>,<p>This is another JSX Element</p>]  
  )  
};
```

3) The other way to fix it is by using the React.Fragment component:

```
const App = () => {  
  return (  
    <React.Fragment>  
      <p>This is first JSX Element!</p>  
      <p>This is another JSX Element</p>  
    </React.Fragment> );  
};
```

- Fragments let you group a list of children without adding extra nodes to the DOM.

****Following are the valid things you can have in a JSX Expression:**

- A string like "hello"
- A number like 10
- An array like [1, 2, 4, 5]
- An object property that will evaluate to some value
- A function call that returns some value which may be JSX
- A map method that always returns a new array
- JSX itself

****Following are the invalid things and cannot be used in a JSX Expression:**

- Loops
- variable declaration
- function declaration
- An object
- undefined, null, and boolean are not displayed on the UI when used inside JSX.

Summary:

- Every JSX tag is converted to `React.createElement` call and its object representation.
- JSX Expressions, which are written inside curly brackets, allow only things that evaluate to some value like string, number, array map method and so on.
- In React, we have to use `className` instead of `class` for adding classes to the HTML element
- All attribute names in React are written in camelCase.
- undefined, null, and boolean are not displayed on the UI when used inside JSX.

2) Redux:

- Redux is a popular state management library for React applications. It provides a global store that holds the application's state, and any component can access the state and dispatch actions to update the state. Redux follows a unidirectional data flow, making it easier to manage and track state changes in large applications.

Core principles in redux:

- 1) Action: Action is an object with type property.
- 2) Reducer: Reducer is a function it receives `initialState` and action and it return new state depends upon action.
- 3) Store: We can create store by using `createStore ()` method which is from redux.

React-redux

React-Redux is a library for integrating the React library with the Redux state management library in a React application. React is a JavaScript library for building user interfaces, and Redux is a predictable state container for JavaScript apps, commonly used with React for managing the state of a web application in a more organized and scalable way.

Here are the main components and concepts in React-Redux:

Provider:

Wraps the entire React application and makes the Redux store available to all components in the component tree.

`connect()` Function: React-Redux provides a `connect()` function that creates container components. These container components are responsible for connecting React components to the Redux store.

The `connect()` function takes two main arguments: `mapStateToProps` and `mapDispatchToProps`. These functions define how to retrieve state from the Redux store and how to dispatch actions.

The `connect` function is a crucial part of the React-Redux library, and it is used to connect a React component to the Redux store. It is a higher-order function (HOC) that wraps your component and provides it with the necessary props to interact with the Redux store.

Here's how the `connect` function is typically used:

```
import { connect } from 'react-redux';

// Define a React component
class MyComponent extends React.Component {
  // Your component logic here
}

// Define a function to map state from the Redux store to component props
const mapStateToProps = (state) => {
  return {
    // Map state properties to component props
    someProp: state.someProp,
    anotherProp: state.anotherProp,
  };
};

// Define a function to map dispatch actions to component props
const mapDispatchToProps = (dispatch) => {
  return {
    // Map action creators to component props
    someAction: () => dispatch(someAction()),
    anotherAction: () => dispatch(anotherAction()),
  };
};

// Connect the component to the Redux store
export default connect(mapStateToProps, mapDispatchToProps)(MyComponent);
```

Here's a breakdown of the key parameters passed to the `connect` function:

mapStateToProps: This function is used to map the state from the Redux store to the props of your component. It takes the current state as an argument and returns an object that defines the props your component needs.

mapDispatchToProps:

It is used to map action creators to the props of your component. It takes the dispatch function as an argument and returns an object with the mapped actions.

connect(mapStateToProps, mapDispatchToProps)(MyComponent):

The connect function returns a new function that you can use to wrap your React component. The resulting connected component will receive the state and actions as props.

Once your component is connected using connect, it can access the Redux state and dispatch actions as props. For example:

```
// Inside MyComponent  
  
console.log(this.props.someProp); // Accessing state  
  
this.props.someAction(); // Dispatching an action
```

This pattern helps in creating components that are connected to the global Redux state, making it easy to manage and update the state in a React application.

In functional component

useSelector and useDispatch are two hooks provided by the React Redux library, which is commonly used for state management in React applications.

useSelector:

- useSelector is used to extract data from the Redux store.
- It takes a selector function as an argument, which is used to select a specific piece of data from the Redux store's state.

useDispatch:

- useDispatch is used to dispatch actions to the Redux store.
- It returns a reference to the dispatch function from the Redux store.
- The dispatch function is used to send actions to the Redux store, triggering state changes.

In summary, useSelector is used to read data from the Redux store, and useDispatch is used to dispatch actions to update the state in the Redux store. Together, these hooks facilitate the integration of React components with the Redux state management system.

Global state

1) class components

- context API
- redux

2) functional components

- Context API
- redux

7) Props

1. In ReactJS, "props" short for properties, it is a way for passing data from a parent component to its child components.

2. Props allow you to make your components dynamic and reusable by providing them with the necessary data from their parent components.

3. Props are read-only (immutable) meaning that the child components should not modify the props directly. If a child component needs to modify the data, ((it should be done by sending a callback function from the parent component as a prop.))

4. Props promote the flow of data from top to bottom in the component hierarchy, following the unidirectional data flow principle of React.

Here's how it works:

- Parent Component: In the parent component, you define a child component and pass data to it using attributes. These attributes are referred to as props in the child component.

- Child Component: In the child component, you can access the data passed from parent through the props object.

1) Parent to child

2) Child to parent

3) Child to child (between siblings)

1) Child to parent & parent to child

2) Context API

3) Redux

8) State

1. In React, "state" is a pre-defined variable for storing data within a component. Unlike props, which are passed from parent to child components and props are read-only, where as state is used for storing data that can change over time and is maintained and managed within the component itself.

2. State is an essential concept in React and is widely used for creating dynamic and interactive user interfaces.

3. A React component's state is used to store and manage data that can change over time. When the state of a component changes, React will automatically re-render the component, updating the user interface to reflect the new state.

4. To use state in a class-based component, you can define the initial state in the constructor using this.state. In functional components, you can use the useState hook to create and manage state variables.

1) What is local state?

- In React, "local state" refers to the state that is confined and managed within a specific component. It means that the state data is not accessible by other components in the application.

- useState:

useState is a built-in React Hook that allows functional components to have local state.

It is used for managing simple state within a component without the need for complex state management solutions.

Use useState when you have local state that is limited to a single component.

1) class components

```
this.state = {}
```

```
this.setState()
```

2) functional components

simple data

useState hook

complex data

useReducer hook

2) What is state lifting?

- State lifting, also known as "lifting state up," is a pattern in React where the state data is moved from a child component to its parent component in the component hierarchy. This is done to share the state between multiple child components

- The need for state lifting typically arises when two or more components need to share the same state. Instead of maintaining the state separately in each child component, the state is moved to a common parent component.

- The parent component then passes down the state data and any necessary callback functions as props to the child components, allowing them to interact with and update the shared state.

3) What is props drilling?

- Props drilling is a term used in React to describe the process of passing data (props) from a higher-level component down to one or more nested child components through multiple levels of the component tree.

- This process of passing the same prop through multiple intermediary components is known as props drilling.

Props drilling typically occurs when ?

- Data needs to be shared between distant components: If two components are not directly connected through parent-child relationships, but they need to exchange data, the data must be passed through intermediary components using props drilling.

- Intermediate components do not use the data themselves: Sometimes, components in the middle of the component tree don't need the data they receive as props. However, since they are required to pass the data down to their child components, they act as channel for passing data, leading to props drilling.

While props drilling is a natural and common pattern in React applications, it can lead to a couple of issues:

1) Prop drilling can make the code harder to maintain and read, especially when there are many levels of nesting.

2) If the data needs to be accessed by a deeply nested component, all the intermediate components must receive and pass down the data, even if they don't use it, which can be inefficient.

4) How to avoid props drilling in react?

- By maintaining global state we can avoid props drilling.

- To address these issues, you can consider using state management libraries like Redux or the React Context API to manage and share data across components without the need for explicit props drilling.

- These libraries offer centralized stores or contexts that allow components to access data without passing it through every level of the component tree.

5) What is Global state?

- In ReactJS, "global state" refers to a centralized state management approach where the application's state is stored and managed in a global container and made accessible to any component in the application.

- This global state can be accessed from any component without the need to pass data through props or props drilling.

- The motivation behind using global state is to simplify the process of sharing data between different components, especially when multiple components need access to the same data. It helps avoid the complexity arise from excessive props drilling.

There are various libraries and approaches available in React to implement global state:

1) React Context API:

- The React Context API is a built-in feature that allows you to create a context and share data through the component tree without explicit props passing.

- The Context API in React allows you to store and share any value that you want, including primitive data types (like numbers and strings), objects, arrays, functions, or even React components. It's not limited to specific data types.

- It enables you to define a Provider component at the top level of the application to provide data, and then any component within the provider's scope can consume that data using the useContext hook.

- When you create a context using `React.createContext()`, you can provide an initial value that will be used when a component accesses the context without a matching provider.

- Basically, Context API consists of two main components: the context provider and the context consumer. The provider is responsible for creating and managing the context, which holds the data to be shared between components. On the other hand, the consumer is used to access the context and its data from within a component.

Steps:

1) Create a Context Object:

First, you need to create a context object using the `createContext` function from the 'react' library. This context object will hold the data that you want to share across your application.

2) Wrap Components with a Provider:

Once you've created a context object, you need to wrap the top level components that need access to the shared data with a `Provider` component. The `Provider` component accepts a "value" prop that holds the shared data, and any component that is a child of the `Provider` component can access that shared data.

3) Consume context value:

In class components: `render props` pattern

In functional components: `useContext` hook

Note: Avoid using it for state that only needs to be accessed within a single component, as it can lead to unnecessary complexity and performance issues.

2) Redux:

- Redux is a popular state management library for React applications. It provides a global store that holds the application's state, and any component can access the state and dispatch actions to update the state.

- Redux follows a unidirectional data flow, making it easier to manage and track state changes in large applications.

Core principles in redux:

1) Action: Action is an object with type property.

2) Reducer: Reducer is a function it receives `initialState` and action and it return new state depends upon action.

3) Store: We can create store by using `createStore ()` method which is from redux.

React-redux

React-Redux is a library for integrating the React library with the Redux state management library in a React application. React is a JavaScript library for building user interfaces, and Redux is a predictable state container for JavaScript apps, commonly used with React for managing the state of a web application in a more organized and scalable way.

Here are the main components and concepts in React-Redux:

Provider:

Wraps the entire React application and makes the Redux store available to all components in the component tree.

`connect()` Function:

React-Redux provides a `connect()` function that creates container components. These container components are responsible for connecting React components to the Redux store. The `connect()` function takes two main arguments: `mapStateToProps` and `mapDispatchToProps`. These functions define how to retrieve state from the Redux store and how to dispatch actions.

The `connect` function is a crucial part of the React-Redux library, and it is used to connect a React component to the Redux store. It is a higher-order function (HOC) that wraps your component and provides it with the necessary props to interact with the Redux store.

Here's how the `connect` function is typically used:

```
import { connect } from 'react-redux';

// Define a React component
class MyComponent extends React.Component {
  // Your component logic here
}

// Define a function to map state from the Redux store to component props
const mapStateToProps = (state) => {
  return {
    // Map state properties to component props
    someProp: state.someProp,
    anotherProp: state.anotherProp,
  };
};

// Define a function to map dispatch actions to component props

const mapDispatchToProps = (dispatch) => {
  return {
    // Map action creators to component props
    someAction: () => dispatch(someAction()),
    anotherAction: () => dispatch(anotherAction()),
  };
};

// Connect the component to the Redux store
export default connect(mapStateToProps, mapDispatchToProps)(MyComponent);
```

Here's a breakdown of the key parameters passed to the `connect` function:

mapStateToProps:

This function is used to map the state from the Redux store to the props of your component. It takes the current state as an argument and returns an object that defines the props your component needs.

mapDispatchToProps:

It is used to map action creators to the props of your component. It takes the dispatch function as an argument and returns an object with the mapped actions.

connect(mapStateToProps, mapDispatchToProps)(MyComponent):-

The connect function returns a new function that you can use to wrap your React component. The resulting connected component will receive the state and actions as props.

Once your component is connected using connect, it can access the Redux state and dispatch actions as props. For example:

```
// Inside MyComponent
```

```
console.log(this.props.someProp); // Accessing state
```

```
this.props.someAction(); // Dispatching an action
```

This pattern helps in creating components that are connected to the global Redux state, making it easy to manage and update the state in a React application.

In functional component

useSelector and useDispatch are two hooks provided by the React Redux library, which is commonly used for state management in React applications.

useSelector:

- useSelector is used to extract data from the Redux store.
- It takes a selector function as an argument, which is used to select a specific piece of data from the Redux store's state.

useDispatch:

- useDispatch is used to dispatch actions to the Redux store.
- It returns a reference to the dispatch function from the Redux store.
- The dispatch function is used to send actions to the Redux store, triggering state changes.

In summary, useSelector is used to read data from the Redux store, and useDispatch is used to dispatch actions to update the state in the Redux store. Together, these hooks facilitate the integration of React components with the Redux state management system.

Global state

1) class components

-context API

- redux

2) functional components

- Context API

- redux

8) Life cycle methods in class components

1) Mounting phase

2) Updating phase

3) Unmounting phase

4) Error boundry

In React.js, components have a lifecycle that consists of different phases during their creation, updating, and destruction. However, with the introduction of React Hooks, some of the traditional lifecycle methods have been replaced or complemented by new functions. Here is an overview of both the traditional class component lifecycle methods and the corresponding React Hook equivalents:

Class Component Lifecycle Methods:

1) Mounting Phase:

1) constructor(props): Initializes the component's state and binds event handlers.

2) static getDerivedStateFromProps(props, state): Used to update the state based on changes in props. Rarely used.

3) render(): Renders the component's UI.

4) componentDidMount(): Invoked immediately after the component is mounted to the DOM. Perfect for side effects like network requests.

2) Updating Phase:

1) static getDerivedStateFromProps(nextProps, prevState): Similar to the mounting phase method, but for updates.

2) shouldComponentUpdate(nextProps, nextState): Determines if the component should re-render, optimizing performance.

3) render(): Renders the updated UI.

4) getSnapshotBeforeUpdate(prevProps, prevState): Captures some information before the component update.

5) componentDidUpdate(prevProps, prevState, snapshot): Invoked after the component update is flushed to the DOM. Useful for side effects.

3) Unmounting Phase:

1) `componentWillUnmount()`: Invoked just before the component is unmounted and destroyed. Clean up resources here.

4) Error Handling Phase:

`componentDidCatch(error, info)`: Used for error boundaries to catch and handle errors within a component's children.

React Hook Equivalents:

Mounting Phase:

`useState(initialState)`: Equivalent to setting state in the constructor.

`useEffect(callback, dependencies)`: Equivalent to `componentDidMount`.

Updating Phase:

`useEffect(callback, dependencies)`: Still used for updates and side effects.

Unmounting Phase:

`useEffect(() => () => cleanup, [])`: Cleanup function to be executed when the component unmounts.

Error Handling Phase:

In React functional components, error handling is typically done using a combination of error boundaries and standard JavaScript error handling techniques. Error boundaries are components that catch JavaScript errors anywhere in their child component tree and display an error UI instead of crashing the whole application. Here's how you can handle errors in functional components in React:

Use try...catch Blocks: You can use standard JavaScript try...catch blocks within your functional component to catch errors that occur during rendering or other operations.

Use Error Boundaries: Error boundaries are special components that catch errors from their child components. You can define an error boundary component and wrap your functional component with it.

Use Custom Error Handling Hooks: You can create custom hooks to handle errors in functional components and provide a consistent way of handling errors throughout your application.

```
import React, { useState } from 'react';

function useErrorHandling() {
  const [error, setError] = useState(null);

  const handleError = (error) => {
    setError(error);
    console.error(error);
  };
}
```

```

    return { error, handleError };
}function My
Component() {
  const { error, handleError } = useErrorHandling();
  try {
    // Your component logic here
    return <div>{someData}</div>;
  } catch (error) {
    handleError(error);
    return <div>An error occurred: {error.message}</div>;
  }
}

```

Remember that error boundaries can only catch errors in their child components during rendering. They won't catch errors in event handlers, async code (e.g., `setTimeout`), or when errors are thrown outside the component tree. For those cases, you may still need to use traditional `try...catch` blocks or other error-handling techniques.

Additionally, make sure to provide helpful error messages and notifications to users when errors occur, but avoid showing too much technical detail that could potentially expose vulnerabilities.

9) Side Effects (DOM interaction/ Browser interaction/ API calls)

1) class components

```

=>componentDidMount
=>componentDidUpdate
=>componentWillUnmount

```

2) functional components

useEffect hook:

The `useEffect` hook is a fundamental part of React functional components that allows you to perform side effects in your components. Side effects can include things like data fetching, DOM manipulation. It's used to handle the lifecycle aspects of a component, similar to how lifecycle methods (`componentDidMount`, `componentDidUpdate`, and `componentWillUnmount`) work in class components.

10) refs

In class components we use `React.createRef()`

- In functional components we use `useRef` hook
- It is used to access/refer DOM elements in react component.

- By using ref we can also store data in react and it can also update value between re-renders without re-rendering the component.
- The updated value will be tracked by react but component will not be re-rendered. For storing value in react we have state variable.
- The value will persist through the re-renders while also not causing any additional renders when its value changes.
- The main purpose of ref variable in react to interact with DOM element and to refer value somewhere.

11) Portals

- Portal is an advanced concept in react which provides a way to render child element outside of parent DOM hierarchy.
- If any child component having modal/popup/tooltip in its JSX then its parent component's css will effect the child's css.
- To implement a portal in React, you use the ReactDOM.createPortal() method. This method allows you to specify the content you want to render and the target DOM element where it should be inserted.

- Create one more div in index.html file to render child component in which modal develops.

```
<div id='model-root'></div>
```

- In child component JSX use the below to create portal.

```
=> ReactDOM.createPortal(modal, document.getElementById(model-root))
```

- By using portals, you can achieve more flexibility in rendering elements and create better user experiences when it comes to managing overlapping or out-of-flow components in React applications.

```
.modal-overlay {
  position: absolute;
  top:0;
  left:0;
  width: 100%;
  height: 100%;
  background-color:rgba(0, 0, 0, 0.5);
  z-index: 2;
  display: flex;
  justify-content: center;
  align-items: center;
```

```

}

.content {
  background-color: white;
  padding: 20px;
  border-radius: 5px;

  box-shadow: 0 2px 8px rgba(0 0 0 0.5)
}

/* .app {
  position: relative
} */

```

12) How to handle with CSS?

- 1) Inline css
- 2) style sheet
- 3) module style sheet
- 3) styled component

Sure, let's take a look at how module styles are scoped locally and how regular styles are scoped globally using code examples.

Module Styles (Scoped Locally):

With module styles, each component has its own CSS module file, and the class names within that file are automatically scoped to that component. Here's an example:

```

// Button.module.css

.button {
  background-color: blue;
  color: white;
  padding: 10px 20px;
}

// Button.js

import React from 'react';
import styles from './Button.module.css';

const Button = () => {
  return <button className={styles.button}>Click me</button>;
}

```

```
};
```

```
export default Button;
```

In this example, the `.button` class defined in `Button.module.css` is scoped locally to the `Button` component. It won't clash with any other `.button` classes in different components because the class name is automatically transformed to be unique.

Regular Styles (Scoped Globally):

With regular styles, if you define a class in an external stylesheet, it can be accessed and used across multiple components or elements. Here's an example:

```
<!-- styles.css -->
```

```
.button {  
  
  background-color: red;  
  
  color: white;  
  
  padding: 10px 20px;  
}
```

```
// Button.js
```

```
import React from 'react';  
  
import './styles.css'; // Import the global stylesheet  
  
const Button = () => {  
  
  return <button className="button">Click me</button>;  
};
```

```
export default Button;
```

In this example, the `.button` class defined in `styles.css` is not locally scoped to the `Button` component. It's globally available and can be used by other components as well. This can lead to potential naming conflicts if not managed carefully, especially in larger projects.

In summary, module styles provide local scoping by automatically generating unique class names for each component, preventing conflicts. Regular styles are globally scoped and can be used across multiple components, which might lead to potential naming clashes. Using module styles is generally considered a better practice for managing styles in larger React applications to ensure encapsulation and avoid global style pollution.

13) Destructuring

- Destructuring is a powerful technique in JavaScript, including when working with React components to handle props and state more effectively. It allows you to extract values from objects and arrays into distinct variables, making your code cleaner and more readable.

Destructuring Props:

When destructuring props in a functional component, you can do so within the function's argument list. Here's how you can destructure props:

Alternatively, you can destructure props within the function body:

```
function Child({ prop1, prop2 }) {  
  //const { prop1, prop2 } = props;//  
  return (  
    <div>  
      <p>Prop 1: {prop1}</p>  
      <p>Prop 2: {prop2}</p>  
    </div>  
  );  
}
```

Destructuring State (Class Component):

When working with class components, you can destructure state in the render method or any other class method:

```
const { stateProp1, stateProp2 } = this.state;
```

Destructuring State (Functional Component with Hooks):

With functional components and Hooks, you can destructure state using the useState Hook:

14) List rendering

- In React, list rendering is the process of rendering a collection of items (such as an array) as a list of React elements. This is a common scenario in web applications where you want to display a dynamic list of items, such as posts, comments, products, etc. React provides various ways to achieve list rendering. Here are some approaches:

Using map(): The map() function is a commonly used method to iterate over an array and create React elements for each item.

```
function App(props) {  
  const items = props.items;  
  return (  
    <ul>  
      {items.map((item, index) => (  
        <li key={index}>{item}</li>  
      ))}  
    </ul>  
  )  
}
```

```
);  
}
```

- Remember that the key prop is important to provide when rendering lists, as it helps React keep track of each element's identity and optimize updates. The key should be a unique identifier for each item in the list. Avoid using the array index as the key when possible, as it can lead to performance issues in certain cases.

15) Events handling

- In React, event handling is essential for creating interactive user interfaces. You can attach event handlers to React elements to respond to various user actions, such as clicks, input changes, and more. Here's how you can handle events in React:

Inline Event Handlers:

- You can define event handlers directly in the JSX element's attributes. Use the onEventName attribute where EventName is the specific event you want to handle (e.g., onClick, onChange)

Binding Event Handlers:

- When using class components, you need to bind the event handlers to the class instance to access this correctly. In functional components, this is not required.

Passing Arguments to Event Handlers:

- If you need to pass additional data to an event handler, you can do so using arrow functions or the bind method.

Preventing Default Behavior:

- You can prevent the default behavior of an event using the preventDefault method.

16) Routing

=> MPA vs SPA

-With MPAs, every time we want to see something new, the whole web page has to reload

With SPAs no need to reload the whole page when we see some new content.

-MPAs will have the entire page getting reloaded Where as SPAs will have the main page loaded once and then sub page sections are changing without refresh and this is handled by JS inside browser.

-MPAs are traditional where as SPAs are modern approach.

-To make a single page application we have to implement the client side routing by using react-router-dom library.

=> Install react-router-dom

Need to install the react-router-dom library to create routing in react, it is dom version.

=> npm i react-router-dom

=> Configure Routing

First step is to configure the router is giving top level or height order component named

`<BrowserRouter/>`

All the routes and components will be wrapped inside the `<BrowserRouter/>` component, as it will store the URLs internally, using which you actually route through all the components

or basically the entire application. It keeps track of changes and navigation history.

Routes and Route =>

When we click on the link which will be in the URL and The Routes component will look for the related child route configured with the same URL value.

To implement the SPA, react router provided us Link component.

=> Link

Link component is used to create SPA in react. When you click on any link it will not refresh or reload entire page. It will load only particular section of data from server.

=> Dynamic routing (useParams)

When we have lot of products from database then creating routes for each product is not easy. Then we create only one route for all products by using dynamic routing concept.

=> Not matching route

If user is typed the URL for route which is not created then we have to handle that case by creating one component which is rendered for not matched route.

give * for path while configure route.

=> Nested routes (Outlet)

When we click on link on menu then child component will be rendered but the menu will be hidden, to persist the menu we have to create nested route for child component. To render child component we have to use Outlet component.

=> useLocation hook (sharing of data)

=> useNavigate hook (page re-direct)

=> Protected routes

=> lazy loading

Lazy loading in React.js is a technique used to improve the performance of web applications by loading certain components, routes, or modules only when they are needed, rather than loading them all at once when the application initially loads. This can significantly reduce the initial load time and resource usage of your application, especially for larger and more complex applications.

Lazy loading is typically achieved using two primary mechanisms in React:

Dynamic Imports: You can use JavaScript's `dynamic import()` function, which allows you to load modules or components asynchronously. By dynamically importing a module, you can specify when and where to load the code.

9) Performance?

What is Conditional rendering?

- Conditional rendering in React.js is the process of displaying different components or content based on certain conditions or state values.

- It allows developers to control the output of the user interface based on specific conditions, such as user interactions, data availability, or component state changes.

- 1) If statements: You can use regular JavaScript if else statements to conditionally render components or elements.

- 2) Ternary operator: The ternary operator is a concise way to conditionally render content based on condition. It has condition ? expression1:expression2

- 3) Logical && operator: In cases where you want to conditionally render content only when a certain condition is true, you can use the && operator.

- Conditional rendering is a powerful concept in React because it enables developers to build dynamic and interactive user interfaces by showing different content based on the application's state and user interactions

What is Fragment?

- In React.js, a Fragment is a special type of component that allows you to group multiple child elements without adding an extra DOM element.

- It was introduced as a feature in React 16.2 to address the common use case where developers needed to return multiple elements from a component's render method without wrapping them in a parent element.

- Prior to the introduction of Fragments, if you wanted to return multiple elements from a component, you had to wrap them inside a single parent element.

- With Fragments, you can achieve the same result without introducing an extra `<div>` element.

- By using Fragments, you keep your JSX cleaner and reduce the number of DOM elements, which can lead to better performance in some cases.

- It's important to note that using Fragment doesn't add any additional functionality to your components; they are purely a syntax feature to make your JSX more concise and organized.

What is HOC?

- In simple terms, an HOC is a function that takes a component as input and returns a new component with additional props or behaviors.

- The purpose of HOCs is to encapsulate and share common functionality across different components without the need for code duplication. It allows you to reuse component logic in your application.
- They are commonly used for tasks like logging, authentication, authorization, and sharing state or behavior among multiple components.
- It's worth noting that the React team has recommended using hooks (e.g., `useState`, `useReducer`, etc.) for managing local state within components, which makes HOCs less commonly used for state management in modern React applications. Hooks provide a more concise and straightforward approach for managing local state without the need for higher-order components. However, HOCs can still be valuable for other scenarios like code reuse, logic separation, and wrapping third-party components.
- HOCs can offer a more localized approach to state management, allowing you to encapsulate state logic within the component tree without relying on a global state container.
- Global state management in React can be achieved using various tools and libraries like Redux, MobX, or React Context API. However, using HOCs is another approach that allows you to share state and functionality across multiple components in a more localized manner.
- By using this approach, you can manage the state in a more localized and composable way, making it easier to understand and maintain your code. Keep in mind that if your application's state management requirements grow complex, using dedicated state management libraries like Redux or MobX might still be a more suitable solution.

What is render props pattern?

- The term 'render prop' refers to a technique for sharing common logic between react components by using prop whose value is a function.

What is custom hook?

- In React.js, a "Custom Hook" is a JavaScript function that allows you to encapsulate reusable logic and stateful behavior to be shared across different components.

- Custom Hooks are powerful because they allow you to abstract complex logic and state management into separate units of functionality, making your components cleaner, more focused, and easier to maintain.
- They also promote code sharing and prevent duplication of logic across different components.
- Custom Hooks follow a specific naming convention by starting the function name with the prefix "use." For example, a custom hook for handling state might be named `useCustomState`, and one for fetching data might be named `useCustomFetch`.

What is Virtual DOM?

- In React.js, the Virtual DOM is a lightweight copy of real DOM. It is a key concept in React's rendering process and plays a crucial role in optimizing the performance of React applications.
- When you create a React component, you define its structure using JSX, React takes these JSX components and converts them into Virtual DOM representation.

Here's how the Virtual DOM works:

1) Component rendering (Create the Virtual DOM Tree:): When you render a React component, React creates a Virtual DOM tree that mirrors the structure of your JSX components.

2) State/props changes: When there's a change in the application's state or props, a new Virtual DOM tree is generated for the updated component and its children. This represents the desired state of the UI.

3) Virtual DOMs comparison (Diffing:): React then compares the new Virtual DOM tree with the previous one to determine the minimal number of changes needed to update the actual DOM.

4) Recursive Comparison: During the diffing process, React compares nodes in both Virtual DOM trees. It checks whether a node is different between the two trees based on its type (e.g, HTML element, functional component, class component) and its key (if provided). If the type or key is different, React considers the nodes to be different and proceeds to update that part of the tree.

4) Reconciliation: After calculating the differences, React applies the necessary updates only to the real DOM nodes that require changes. This process of updating only the specific parts of the DOM is known as "reconciliation" and is much more efficient than directly updating the entire DOM.

5) Actual DOM update: Finally, React applies the "diff" to the real DOM in a single batch update, avoiding excessive direct manipulation of the DOM, which can be slow and resource-intensive.

- In summary, the Virtual DOM is an essential part of React's rendering process that acts as a middle layer between your React components and the actual DOM, optimizing performance by efficiently updating only the necessary parts of the DOM.

Let's start with the Virtual DOM.

Virtual DOM

It is a virtual representation of the UI (copy of DOM) kept in memory and synced with the real DOM by a library like ReactDOM.

Why do we need a virtual DOM when there is an actual DOM?

Well, DOM operations are expensive, and updating the whole DOM on every prop/state change is very inefficient. Here's how the virtual DOM deals with this inefficiency:

A component props/state changes

-React triggers a re-render.

-React compares the virtual DOM (virtual DOM before the update) with the updated virtual DOM (virtual DOM after update).

-React determines the best possible way to reflect the changes in the UI with minimal operations on the real DOM.

-So its virtual DOM helps React update the UI to match the most recent tree.

Reconciliation

The process of keeping virtual DOM in sync with the real DOM is called reconciliation.

-So the whole process we discussed above is known as reconciliation.

Diffing

The comparison between the virtual DOMs (to figure out what needs to be updated in the UI) is referred to as diffing, and the algorithm that does it is called diffing algorithm.

How does the diffing algorithm work?

The diffing algorithm compares the two trees by comparing their root nodes first.

1. Different root nodes

```
<div>
  <p>Hello</p>
</div>
<section>
  <p>Hello</p>
</section>
```

Comparing the two trees above will lead to a full rebuild because the root elements `<div>` and `<section>` are different. Any components below root will also be rebuilt.

2. Same root nodes

When DOM elements of the same type are compared, React compares the attributes of both, keeps the same underlying DOM node, and updates the changed attributes.

```
<div className="one" />
<div className="two" />
```

React modifies the `className` on the underlying DOM node.

3. Diffing lists

When diffing children of a DOM node, React compares children of both lists and generates a mutation whenever there's a difference.

Consider the list of todos below:

```
<ul> <li>sachin</li> <li>kohli</li></ul>
```

Now, let's add a new item at the beginning of the list:

```
<ul>
  <li>gill</li>
  <li>sachin</li>
  <li>kohli</li>
</ul>
```

When a new item is added to the list, the following happens:

React compares `gill` in the virtual DOM with `sachin` in the updated virtual DOM

The comparison tells React that the list items are changed.

React will repaint all the list items in the DOM

This is very inefficient because React has to repaint all the list items. To solve this, we can give a key prop a value that uniquely identifies a list item among its siblings.

Keys

The key prop lets React identify each element in a list. It is used to keep track of items that are changed, added, or removed. Key should be unique among its siblings not globally.

Now let's add a key prop to our above example:

```
<ul>
  <li key="sa">sachin</li>
  <li key="ko">kohli</li>
</ul>

<ul>
  <li key="gi">gill</li>
  <li key="sa">sachin</li>
  <li key="ko">kohli</li></ul>
```

Now React knows that the element with key 'gi' is the new one, and React will just add the new element to the DOM instead of manipulating all items again in the DOM.

What To Pick As the Key?

Usually, the data set you are iterating already has an ID or something that can be used as an ID. So you can just use that.

```
<li key={item.id}>{item.name}</li>
```

When you don't have a stable ID in your data set, you can use libraries like nanoid that can generate unique ids for you.

```
import { nanoid } from 'nanoid';

{items.map(item => (
  <div key={nanoid()}>{item.name}</div>
))}
```

However, an index is not always the best choice and can lead to poor performance and unexpected rendering problems.

Now, let's add a new item at the ending of the list:

React will insert the new child at the end of the children.

It renders/inserts only last child in list in DOM instead of all childs in DOM. It does not re-render remaing childs in list.

What is Pure component?

- It is a specific type of React component that automatically implements `shouldComponentUpdate()` with shallow comparison of props and state to determine whether the component should re-render or not.

- By default, when a component receives new props or updates its state, React will trigger a re-render of that component and its child components.

- However, in some cases, the new props or state might be the same as the previous ones, and the component doesn't need to re-render because it would produce the same output.

- A Pure Component optimizes this process by doing a shallow comparison of the current and previous props and state.

- If React determines that the new props and state are equal to the previous ones, it will skip the re-rendering process for that component and its children, saving unnecessary rendering cycles.

- To create a Pure Component in React, you can extend the `React.PureComponent` class instead of `React.Component`.

What is `React.memo()` in react?

- `React.memo` is a higher-order component (HOC) provided by React that is used for optimizing functional components by preventing unnecessary re-renders.

- It is similar to `React.PureComponent` for class components, but it is specifically designed to work with functional components.

- By default, when a functional component is re-rendered, it re-executes its entire function body, potentially recalculating values and causing re-renders for its child components.

In some cases, this can lead to performance issues, especially when the component receives the same props and its rendering output would be identical.

- `React.memo` optimizes this process by memoizing the component's result and re-rendering it only when its props change. It performs a shallow comparison of the current and previous props, and if they are equal, the component is not re-rendered, and the previously memoized result is used.

- When you wrap a functional component with `React.memo`, it will only re-render if its props have changed since the last render. If the props are the same, React will skip the rendering process for that component, preventing unnecessary re-renders and improving performance.

What is `useMemo()` hook?

- In `React.js`, `useMemo` is a hook that is used for memoizing expensive computations, preventing unnecessary recalculation of values on each render.

- It allows you to cache the result of a function call and return the cached result when the inputs (dependencies) to that function have not changed since the last render.

- The useMemo hook takes two arguments: the first argument is the function that computes the value you want to memoize, and the second argument is an array of dependencies.
- The hook will recompute the value only when one of the dependencies in the array has changed. If the dependencies remain the same between renders, useMemo will return the cached value, avoiding redundant calculations.

What is useCallback hook?

- In React.js, useCallback is a hook that is used to memoize functions in order to avoid unnecessary re-creations of those functions on every render.
- It is particularly useful when dealing with child components that receive functions as props, as re-creating functions can cause those child components to re-render unnecessarily, leading to performance issues.

The useCallback hook takes two arguments:

- 1) The function that you want to memoize.
- 2) An array of dependencies. The memoized callback will only be re-created when any of the dependencies in the array change.

- Using useCallback is beneficial when dealing with components that receive functions as props, as it can prevent those components from re-rendering needlessly and improve overall performance.

While both useMemo and useCallback remember something between renders until the dependencies change, the difference is just what they remember.

=> useMemo will remember the returned value from your function.

=> useCallback will remember your actual function.

=> useMemo is to memoize a calculation result between a function's calls and between renders.

=> useCallback is to memoize a callback itself (referential equality) between renders/

=> useRef is to keep data between renders (updating does not fire re-rendering)

=> useState is to keep data between renders (updating will fire re-rendering)

What is the difference between useCallback and useMemo?

=> useCallback returns a memoized callback function,

=> while useMemo returns a memoized value.

=> Both hooks can be used to optimize the performance of your React components by avoiding unnecessary re-creations of functions or values

React.memo vs useCallback vs useMemo:

1) Without passing props:

1) Initial Rendering:

=> By default, if parent rendered then all its child components will be rendered.

2) Re-rendering:

=> By default, every time parent is re-rendered then all its child components also will be re-rendered as well regardless of props changed.

2) with passing same props:

1) Primitives types as prop:

=> This can be handled by `react.Memo()`

2) Reference types as prop (array, object):

=> This can not be handled by `react.Memo()`

1) this can be handled by `useMemo()` hook (if having dependence)

2) move array/ object outside(above) of component then no need of `useMemo()` hook.

3) function as prop:

=> This can not be handled by `react.Memo()`

1) this can be handled by `useCallback()` hook (if having dependence)

2) move function outside(above) of component then no need of `useCallback()` hook.

3) previous and current props are different:

1) Primitives types as prop:

=> component will be re-rendered

2) Reference types as prop (array, object):

=> component will be re-rendered

3) function as prop:

=> component will be re-rendered

Note: `useCallback` and `useMemo` both expect a function and an array of dependencies.

JAVASCRIPT

1) Introduction

1) what is JS?

JS is a programming language which is used to develop dynamic web and mobile apps.

By using JS we can manipulate (add, update, delete) DOM.

By using JS we can perform validations.

2) What is ES?

Ecma Script is a specification for JS.

3) What is nodejs?

Node js is a run time environment for JS. By using nodejs we can create APIs.

Node js is not a programming language. Node js is not a library.

Node js is not a framework.

2) Setup Development environment?

2) Front-end

- We need a Browser (Chrome).
- We need an IDE to write code (vs code).
- We need a HTML file to run js code in front-end.
- Every browser has js engine in it to run js code.

3) Back-end

- We need to install Nodejs to run js code in back-end.
- We need an IDE to write code.
- Nodejs is a runtime environment for js.
- By using nodejs js we can create APIs.
- How to run js file in nodejs

=> node filename

3) Basics

=> When to store data?

We need to store data in memory, when there is need in app.

=> How to store data in memory?

By using var, let, const, we can store data in memory.

We can perform some operations on the data which is stored in memory

=> What to store in memory?

We have to store values (data) by using data types. These are the real values to do any functionality.

1) variables (we need variables to manipulate/perform some operation/update/delete/save)

We will declare variables by using var, let, const keywords.

2) data types

Primitive data types: we use primitives data types to store single value in memory.

Note: All primitive data types will store value directly in memory.

1) string: String is collection of characters to be stored in memory.

By using " ' " ` we can store string in memory.

2) number: We can store numbers in memory by using number data type.

whether it is integer or float number.

3) boolean:

We can store true or false in memory by using boolean datatype.

4) undefined: undefined means a variable has been declared but its value has not been assigned.

5) null: Null means an empty value. The variable which has been assigned as null contains no value.

Reference data types: we use reference data types to store multiple values in memory.

Note: All reference data types will store value somewhere in memory location. The stored memory location address will be stored in main memory.

1) Object:

By using object we can store multiple values in single memory location in the form of key & value pair. By using dot operator we can access the object values in app.

Objects are often used to model real-world entities such as a person, car, or any other entity that has properties and behaviors.

2) Array:

By using array we can store collection of values in single memory location. It stores only values. Internally js attaches index numbers to the values in array. By using index numbers we can access array values in app.

3) Function:

- Function is block of code. By using a function we can do some task and return some value.
- For every function call separate execution context will be created.
- For every execution context, there are memory creation phase and code execution phase.
- We can store multiple values in functional scope.

ES-6

4) Map

5) WeakMap

6) Set

7) WeakSet

3) typeof operator

- By using typeof operator we can find data type for the value which is stored in memory.

value : datatype

'sachin': string

40: number

true/false: boolean

undefined: undefined

null: object

{}: object

[]: object

function({}): function

4) operators:

why operators?

we use operators to develop some logic or expression in combination with variables.

1) Arithmetic

We use arithmetic operators to perform some mathematical operations.

+ add

- subtraction

* multiplication

/ division

% remainder

** exponential

++ incremental : It increases 1 at a time.

-- decremental : It decreases 1 at a time.

2) Assignment (=)

By using assignment operator we can assign/store value (data) to a variable.

3) Comparison

It compares the two values, the result from this operators will be true/false.

1) Rational/Relational

>

>=

<

<=

2) Equality

1) Loose equality (==) It compares only value of variables

2) Strict equality (===) It compares value and data type of variables

`undefined == null`

`undefined === null`

3) Not equality

1) Loose inequality (!=)

The != operator is the inequality operator. It checks whether two values are not equal, regardless of their types. If the values are different, it returns true. If the values are the same, it returns false.

2) Strict inequality (!==)

The !== operator is the strict inequality operator. It checks whether two values are not equal and whether they are of the same type. If the value or the type are different, it returns true; otherwise, it returns false.

4) Ternary operator

We use ternary operator to render content conditionally.

`let age = 15;`

`let vote = age >= 18 ? 'Having vote' : 'Not having vote';`

`console.log(vote);`

5) Logical operator

In JavaScript, logical AND (&&) and logical OR (||) are operators used to perform logical operations on boolean values or expressions.

1) logical and &&

2) logical or ||

3) ! Operator

The exclamation mark (!) is the logical NOT operator.

When used, it converts a true value to false and vice versa.

! is the logical NOT operator, used for negating boolean values.

6) Control statements

We use control statements to develop some logic or functionality when we have multiple conditions.

1) if else - In JavaScript, the if...else statement is used for conditional execution of code. It allows you to perform different actions based on a specific condition.

The syntax of the if...else statement is as follows:

```
if (condition) {  
    // Code block to be executed if the condition is true  
} else {  
    // Code block to be executed if the condition is false  
}
```

2) switch case (It does not work for step value)

- In JavaScript, the switch statement is another way to perform conditional execution of code based on the value of an expression. It is often used as an alternative to multiple if...else statements.

- The switch statement evaluates an expression once and then matches the value of the expression to a case label. If a matching case label is found, the corresponding block of code is executed.

7) Loops

- We use loops to do same task again and again simply.

- We use loops to access memory value multiple times in an application.

Conditional Loops

1) for loop:

In JavaScript, a for loop is used to execute a block of code repeatedly for a specified number of times.

2) while loop:

In JavaScript, a while loop is used to execute a block of code repeatedly as long as a specified condition is true. The loop continues until the condition evaluates to false.

3) do while loop:

In JavaScript, a do-while loop is similar to a while loop, but with a slight difference. The primary difference is that in a do-while loop, the loop body is executed at least once before the loop condition is checked.

This ensures that the loop body is executed at least once, regardless of whether the condition is initially true or false. The do-while

4) infinity loop:

An infinite loop in JavaScript is a loop that runs infinitely, continuously executing the same code block without ever stopping. This usually happens when the loop condition always evaluates to true

****break=>** by using Break keyword we can break the loop and make exit the loop.

****continue=>** by using continue keyword we can make jump the loop.

Non conditional loops

5) for in loop:

We use this loop to iterate keys in object.

In JavaScript, the for...in loop is used to iterate over the enumerable properties of an object. It allows you to loop through the keys (property names) of an object and access their corresponding values.

6) for of loop:

In JavaScript, the for...of loop is used to iterate over the values of iterable objects(array/string object/set/map/arguments object/generator object/).. It provides an easy and concise way to loop through arrays, strings, sets, maps, and other objects that are iterable.

Note:

- It's important to note that the for...in loop should be used for iterating over object properties. If you want to loop through elements of an array, it is recommended to use the for...of loop or a simple for loop with an index.

- keep in mind that the for...of loop is not suitable for iterating over regular objects (objects created with {}) since they are not iterable by default. For iterating through object properties, you should use the for...in loop.

8) Functions

1) What is function?

=>Function is a block of code which is used to do some task and return value.

=>It stores multiple values in functional scope.

=>For every function call separate execution context will be created.

For every execution context there are memory creation phase and code execution phase.

2) How can we define function?

1) Function declaration

2) Function expression

1) named function expression

2) anonymous function expression

3) arrow function (ES-6)

3) parameters vs arguments

Parameters

Parameters are the placeholders or variables defined in the function's declaration.

- They are like local variables that store the values passed to the function when it is called.

- They are defined within the function's parentheses in the function declaration and serve as placeholders for the arguments that will be provided when the function is called.

Arguments

Arguments are the actual values or expressions passed to a function when it is called.

- They represent the data that you want to work with within the function.
- The number of arguments should match the number of parameters defined in the function.
- Arguments are provided in the function call and are placed inside the parentheses.

4) what is default parameter

- Default parameters were introduced in ECMAScript 6 (ES6) and have since become a common feature in modern JavaScript.

- To define default parameters in a function, you assign a default value to a parameter in the function's declaration.

- Default parameters in JavaScript allow you to specify default values for function parameters.

5) Varying no of parameters or arguments

If we have varying no of params and arguments then we have to handle with below concepts.

=> arguments object (ES-5)

It takes all values at a time and stores in memory.

This is available in all functions except arrow function.

arguments object is an iterable object, it has `symbol.iterator()` method.

=> rest parameter (ES-6)

It starts with ...

It takes all values and stores in array.

It should be last parameter in parameter list.

6) Scope

1) Global scope

2) Function / local scope

3) Block scope

Note:

1) var is functionl scope

2) let, const is block scope

- In Js, "scope" refers to the context in which variables, functions are stored and can be accessed.

- The scope determines the accessibility and lifetime of these variables and functions.

- Every execution context will create a new scope.
- Every function call will create new execution context.

JavaScript has two main types of scope:

Global Scope:

Variables declared or stored outside of any function or block have global scope.

They can be accessed from any part of the code, including inside functions and blocks.

Local Scope:

Variables declared or stored within a function or a block have local scope. They are only accessible within that specific function or block.

Note:

Local variables take precedence over global variables if they share the same name.

Function scope, which means that variables declared with `var` are only scoped within the function where they are defined.

However, with the introduction of `let` and `const` in ES6, block scope was introduced. Variables declared with `let` or `const` are scoped to the nearest enclosing block (defined by curly braces `{}`).

7) Hoisting:

=> Hoisting is default behaviour of javascript of moving all variable declarations and function declarations to top of current scope.

=> Hoisting lets you allow to access memory value even before execution of code.

In JavaScript, both `let` and `const` declarations are hoisted, but they behave differently compared to `var` declarations.

When you declare a variable using `var`, it gets hoisted to the top of its scope and is initialized with `undefined`. However, `let` and `const` are also hoisted to the top of their scope, but they are not initialized. This means that you cannot access them before the actual declaration (you'll get a `ReferenceError`), which is known as the "temporal dead zone."

```
console.log(x); // undefined
```

```
var x = 5;
```

```
console.log(y); // ReferenceError:
```

```
let y = 10;
```

```
console.log(z); // ReferenceError: z
```

```
const z = 15;
```


In the above example, x is declared using var, so it gets hoisted and initialized with undefined. y and z are declared using let and const, respectively. While they are hoisted, accessing them before the actual declaration results in a ReferenceError due to the temporal dead zone.

both let and const declarations are hoisted in JavaScript. However, unlike var, they are not initialized with undefined. Instead, they enter a "temporal dead zone" until their declaration is encountered in the code. Attempting to access a variable declared with let or const before its declaration will result in a ReferenceError due to being in this temporal dead zone. This behavior ensures that variables declared with let or const are not accessible before they are defined in the code execution flow.

8) var vs let vs const

	var	let	const
1) scope	functional	block	block
2) hoisting	yes	yes but differently	yes but differently
3) re-declaration	yes	no	no
4) re-assignment	yes	yes	no
5) initialization	no need	no need	need

9) Closure:

=>Closure is a concept in js, it allows to inner function to access outer scope variables even after the outer function execution content is over.

9) Objects

1) What is an object?

- In JavaScript, an object is a reference data type that stores a collection of related data and functionality.
- It is a container for key-value pairs, where each key is a unique identifier and each value can be of any data type, including other objects, arrays, functions, primitives (such as numbers, strings, and booleans)
- Objects in JavaScript are often used to model real-world entities such as a person, car, or any other entity that has properties and behaviors.
- JavaScript also has some built-in objects like Array, Date, Math. These built-in objects provide specialized functionality for specific use cases.
- You can access the properties and methods of an object using dot notation or bracket notation.

2) How to create object:

1) Object literal way

3) Factory function

4) Constructor function

5) Classical way (ES-6)

3) Objects are dynamic:

You can add, modify, or delete properties and methods of an object at runtime, making objects dynamic and flexible in JavaScript.

-add

-update

-delete

4) Iterating object:

- for in loop

- Object.keys/ Object.entries /Object.values

Object.keys => This method will create array with looped keys in object.

Object.values => This method will create array with looped values in object.

Object.entries => This method will create array with looped keys and values in object.

5) Copy primitives and reference types:

=>primitives copy

By default deep copy

=>Object copy

normal copy => = (assignment operator)

swallow copy => Object.assign({}, source object) or spread operator.

deep copy => JSON.parse(JSON.stringify(obj))

(it will copy only pimitives and obj, it will ignore method copy)

cloneDeep() => lodash

(it will copy all pimitives and obj and method copy)

6) Math object

This is predefined object in javascript to do math related functionality.

1) min => It gives min value.

2) max => It gives max value

3) ceil => It rounds to up value when we have float value.

4) floor => It rounds to down value when we have float value.

5) round => It rounds to up value when we have 0.5 or above and down value when we have below 0.5.

6) random => It generates random value.

- Math.random() always returns a number lower than 1.

- Math.floor(Math.random() * 10);

- Returns a random integer from 0 to 9:

- Math.floor(Math.random() * 100);

- Returns a random integer from 0 to 99:

7) pow => In JavaScript, the Math.pow() function is used to calculate the power of a number. It takes two arguments: the base number and the exponent. Both Math.pow() and the exponentiation operator have the same functionality, but using the exponentiation operator can be more concise and readable in many cases.

8) sqrt => In JavaScript, the Math.sqrt() function is used to calculate the square root of a given number. It takes a single argument, which is the number whose square root you want to find.

Math.sqrt(number);

7) Date object

This is predefined object in javascript.

By using this we can create dates.

1) var now = new Date() ==> It gives current date.

2) var x = new Date(2040, 0, 30, 9:00) ==> It is number date type we can create custom dates.

3) var x = new Date('2040 0 30 9') ==> It is string date type we can create custom dates.

8) Garbage collector

=> In low level languages like c, c++ when we create variable we need to allocate memory to it.

=> when we use it we have to de-allocate memory. we dont have this concept in javascript. we can easily create a variable at the time we initialized this variable the memory is automatically allocated to this variable next we can use that when we are done using we dont have to de-allocate memory.

=> So JS engine has garbage collector. The job of this garbage collector is to find the variable which are no longer used and then de-allocate the memory. Memory allocation and de-allocation allomatically done behind the scene you have no control on it. You cant tell garbage collector when to run and what varibales to remove from the memory.

=> Based on some complex algorithms(mark and sweep) the garbage collector run in the background it figurs out what variables are not used and then it automatically removes from memory.

9) Template literal (ES-6)

Synonyms:

Template Literals

Template Strings

String Templates

=> Template strings are a powerful feature of modern JavaScript released in ES6.

=> It lets you insert variables and expressions into strings without needing to concatenate + like in older versions of JavaScript.

=> It allows us to create strings that are complex and contain dynamic elements.

=> Template Literals use back-ticks (`) rather than the quotes (") to define a string.

=> Template literals allows multiline strings.

syntax: `\${}`

10) String Object

1) String length ==> It gives string length.

2) Removes white spaces

=>trim() ==> It removes white space both sides.

=>trimStart() ==> It removes white space at starting

=>trimEnd() ==> It removes white space at ending

3) Extracting part of string

=> slice(startindex, endindex)

=> substring(startindex, endindex)

=> substr(startindex, length)

slice()

-The slice() method extracts a part of a string.

-The start and end parameters specifies the part of the string to extract.

-A negative number selects from the end of the string.

subString()

-The substring() method extracts characters from start to end (exclusive).

-Start or end values less than 0, are treated as 0. It does not work for negative values.

subStr() -The substr() method extracts a part of a string.

-The substr() method begins at a specified position, and returns a specified number of characters.

-To extract characters from the end of the string, use a negative start position.

4) Extracting string characters.

=> charAt(index) - It returns the character at specified index.

=> charCodeAt(index) - It returns unicode of character at specified index.

5) Replacing string content

=> replace(exitingstring, newstring).

=> by default replace method replaces only first match.

=> replace method is case sensitive.

6) Converting to upper and lower case

=> toUpperCase ==> It converts string to UPPERCASE.

=> toLowerCase ==> It converts string to lowercase.

7) To Join two or more strings.

=> we use concat() method.

=> It is used to concat two or more strings.

=> concat method is used instead of plus operator.

8) String padding

=>padStart(target_length, pad_string)

=>padEnd(target_length, pad_string)

=>target_length - The desired length of the resulting string after it has been padded.

9) Converting string into array.

=>split("")

-The split() method splits a string into an array of substrings.

-The split() method returns the new array.

-The split() method does not change the original string.

-If (" ") is used as separator, the string is split between words.

10) Methods for finding specific string.

=>indexOf() ==> It gives first occurrence index number. if not return -1

=>lastIndexOf() ==> It gives last occurrence index number. if not return -1

=>includes() ==> It gives true if string has that that.

=>startsWith() ==> It gives true if it has started with same string as you asked.

=>endsWith() ==> It gives true if it has ended with same string as you asked.

1) What is an array.

- In JavaScript, an array is a data structure used to store a collection of values.

Arrays allow you to store multiple values of different types or the same type into a single variable, making it easier to manage and manipulate collections of data.

Each value in an array is referred to as an "element," and each element has an associated index, starting from 0 for the first element.

Here's how you can create an array in JavaScript:

Creating an array of numbers

```
const numbers = [1, 2, 3, 4, 5];
```

Creating an array of strings

```
const fruits = ['apple', 'banana', 'orange'];
```

Creating an array of mixed types

```
const mixedArray = [1, 'hello', true, null, undefined];
```

You can access elements in an array using their index:

```
console.log(numbers[0]); // Outputs: 1
```

```
console.log(fruits[1]); // Outputs: banana
```

- Arrays have a variety of methods that you can use to manipulate their contents, such as adding or removing elements, iterating through elements.

Here are a few common array methods:

push: Adds an element to the end of the array.

pop: Removes the last element from the array.

shift: Removes the first element from the array.

unshift: Adds an element to the beginning of the array.

length: Returns the number of elements in the array.

splice(): Can be used to add, remove, or replace elements at a specific position in the array.

concat(): Combines two or more arrays to create a new array.

slice(): Creates a new array by extracting a portion of the existing array. Creates a shallow copy of a portion of an array.

forEach: Executes a function for each element in the array.

map: Creates a new array by applying a function to each element in the array.

filter: Creates a new array with elements that pass a test condition.

2) adding of elements to an array

push: It adds new item to end of array.

unshift: It adds new item to start of array.

splice: It adds new item at any place in array.

3) removing specific element from an array

pop: It removes end value in array.

shift: It removes start value in array.

splice: It removes value in any where in array.

4) removing all elements from an array

`[]`

`length = 0`

`splice()`

5) finding of elements in an array

primitives:

indexOf: It finds index of value at first occurrence in array.

If that value is not found in array it gives -1

lastIndexOf: It finds index of value at last occurrence in array.

If that value is not found in array it gives -1

includes: If given is in array then it gives true as output.

If given value is not found in array it gives false as output.

reference:

find:

It finds whether the given value is in array then it returns the same value.

If the given value is not there in array then it returns undefined.

findIndex:

It finds whether the given value is in array then it returns the index number

of that value. If the given value is not there in array then it returns -1.

6) combining two or more arrays as one array.

1) concat() => es-5

`var combineArr = arr1.concat(arr2)`

2) spread operator => es-6

```
var combineArr = [...arr1, ...arr2]
```

7) copy/clone an array.

1) Normal copy => (= assignment operator)

2) Shallow copy

=> ES-5 slice()

=> ES-6 spread operator

3) Deep copy

=> without method JSON.parse(JSON.stringify(array))

=> with method lodash (cloneDeep(array))

1) Normal copy

2) Shallow copy

1. Using the slice() Method:

- The slice() method can be used to create a shallow copy of an array. It takes two optional arguments: the starting index and the ending index. If no arguments are provided, it copies the entire array.

```
const originalArray = [1, 2, 3, 4, 5];
```

```
const shallowCopy = originalArray.slice();
```

2. Using the Spread Operator ([...]):

- The spread operator allows you to create a shallow copy of an array in a concise way.

```
const originalArray = [1, 2, 3, 4, 5];
```

```
const shallowCopy = [...originalArray];
```

- Both of these methods create a new array that contains the same elements as the original array. However, keep in mind that the elements themselves are still references to the same objects. If the array contains objects or arrays, changes made to those objects or arrays will be reflected in both the original array and the shallow copy.

```
const originalArray = [{ value: 1 }, { value: 2 }, { value: 3 }];
```

```
const shallowCopy = originalArray.slice();
```

```
shallowCopy[0].value = 100;
```

```
console.log(originalArray[0].value); // Output: 100
```

```
console.log(shallowCopy[0].value); // Output: 100
```

- As seen in the example, modifying an object within the shallow copy also affects the corresponding object in the original array. If you need to create an independent copy where changes to one array don't affect the other, you would need a deep copy.

3) Deep copy

- you can create a deep copy of an array using the lodash library in JavaScript. Lodash provides a convenient method called `_cloneDeep()` that allows you to create a deep copy of an array or any nested data structure, handling all the complexities of deep copying effectively.

- `cloneDeep()` function from the lodash library takes care of creating a complete and independent copy of the array and its nested elements, ensuring that modifications to the deep copy do not affect the original array.

8) iterating of elements in an array

1) for of:

=>for of loop iterates an array values only.

2) forEach:

=>The `forEach()` method calls a function for each element in an array.

=>`forEach()` method is used to loop through values and index of an array.

`forEach` method does not return any value. It returns undefined.

9) Joining of an array

`join()`

=>The `join()` method returns an array as a string.

=>The `join()` method does not change the original array.

=>Any separator can be specified. The default is comma (,).

10) testing of an array elements

1) `some()`:

=>`some` method in JavaScript is used to check whether at least one of the elements of the array satisfies the given condition or not.

=>The only difference is that the `some()` method will return true if any predicate is true while `every()` method will return true if all predicates are true.

2) `every()`:

=>`every` method in JavaScript is used to check whether all the elements of the array satisfy the given condition or not.

=>The output will be false if even one value does not satisfy the element,

else it will return true, and it opposes the `some()` function.

11) filtering of array

`filter()`

=>The `filter()` method creates a new array filled with elements that pass a test provided by a function.

=>The filter() method does not execute the function for empty elements.

=>The filter() method does not change the original array.

12) Sorting of an array

1) Ascending order

2) Descending order

primitives:

sort()

reverse()

reference:

sort + comparison function

=> nested for loop

13) mapping of array

map()

=>The map() method creates a new array populated with the results of calling a provided function on every element in the calling array.

=>It returns new array with transforming values.

14) reduce of array

reduce()

=>The reduce() method executes a reducer function for array element.

=>The reduce() method returns a single value: the function's accumulated result.

=>The reduce() method does not change the original array.

15) Flattening of array(converting multi dimensional array into single dimension array).

1) flat() method

2) Array.isArray()

16) How to remove duplicate elements in an array.

1) Set with spread operator

2) indexOf()

17) map() vs forEach()

- The map() method returns a new array, whereas the forEach() method does not return a new array

- The map() method is used to transform the elements of an array, whereas the forEach() method is used to loop through the elements of an array.

18) Array.from()

- The Array.from() method returns an array from any object with a length property.
- The Array.from() method returns an array from any iterable object.
- Array.from() is an ECMAScript6 (ES6) feature.
- Array.from(object, ()=>{})
- object: This parameter holds an object that will convert into an array.

This method can create an array from:

- array-like objects - The objects that have length property and have indexed elements like String.
- Iterable objects like Map or Set.

Return Value

- Returns a new Array instance.

The Array.from() method, which is new in ES6, creates a new instance of the Array from an object that acts like an array or is iterable. The syntax for the Array.from() method is shown below:

```
Array.from(target [, mapFn[, thisArg]])
```

target is an object to be converted into an array that is iterable or array-like. The map function, or mapFn, should be used on each element of the array. The value used to call the mapFn method is thisArg.

A new instance of Array is returned by the Array.from() method, and it contains every element of the source object.

19) at()

The at() method takes an integer value and returns the item at that index, allowing for positive and negative integers. Negative integers count back from the last item in the array.

20) fill()

The fill() method in JavaScript is used to fill all the elements of an array from a start index to an end index with a static value.

It mutates the original array and returns the modified array.

The fill() method in JavaScript is used to fill all the elements of an array from a start index to an end index with a static value.

It mutates the original array and returns the modified array.

21) Array.isArray()

The Array.isArray() static method determines whether the passed value is an Array or not.

11) DOM

1) What is DOM? - DOM stands for Document Object Model. It is a programming interface for web documents, primarily used to access and manipulate the content and structure of web pages. The

DOM represents the page as a tree-like structure where each element on the web page, such as HTML tags, text, and attributes, is treated as an object in the tree.

Here are some key points about the DOM:

- The DOM represents the structure of HTML document as a tree, where each node in the tree corresponds to an element or part of the document.
- The DOM is dynamic, You can use JavaScript to interact with the DOM, change content, add or remove elements, and respond to user actions.
- The DOM allows you to attach event listeners to elements on a web page so that you can respond to user interactions like clicks, mouse movements, and keyboard input.
- Web browsers use the DOM to render web pages. When a web page is loaded, the browser parses the HTML and constructs the DOM tree, which it uses to display and render the page.
- Here's a simple example in JavaScript of how you can use the DOM to manipulate a web page:
- Get a reference to an DOM element with an 'id' attribute of 'myId'

```
var element = document.getElementById("myId");
```

- Change the text content of the element

```
element.textContent = "Hello, World!";
```

- Add a new element to the page

```
var newElement = document.createElement("p");  
newElement.textContent = "This is a new paragraph."  
document.body.appendChild(newElement);
```

In this example, we use the DOM to select an element with the ID "myId" change its text content, and create a new paragraph element that is appended to the document body. This demonstrates how the DOM can be used to manipulate the content a web page dynamically.

2) How to examine DOM

`console.dir(document)` => It gives whole DOM with lot of properties.

3) How to read DOM properties

`document.propertyname` => We can read DOM properties by using dot operator or []

4) How to select DOM elements

=> id

```
var para1 = document.getElementById('one');
```

=> class

```
var x = document.getElementsByClassName('one');
```

=> tag

```
var x = document.getElementsByTagName('p');
```

=> querySelector

```
var x = document.querySelector('#one');
```

```
var x = document.querySelector('.one');
```

```
var x = document.querySelector('p');
```

=> querySelectorAll

```
var x = document.querySelectorAll('#one');
```

```
var x = document.querySelectorAll('.one');
```

```
var x = document.querySelectorAll('p');
```

Note: By using id we can get only one element. id is unique.(single element as result)

By using querySelector we can get first element only.(single element as result)

By using className we will get html collection in array.

By using tagName we will get html collection in array.

By using querySelectorAll we will get node list in array.

5) Traversing of DOM

element to parent

=> element.parentElement

element to child

=> element.firstChild

=> element.lastElementChild

=> element.children

element to siblings

=> element.previousElementSibling

=> element.nextElementSibling

6) Adding / removing / replacing DOM elements

1) Adding

parent.appendChild(child)

parent.insertBefore(new child, existing child)

2) Replacing

parent.replaceChild(new child, existing child)

3) Remove

```
element.remove()
```

```
parent.removeChild(child)
```

7) Adding events to DOM elements from Javascript.

```
var button = document.getElementById('btn')
```

```
DOMElement.addEventListener('eventname', function(event handler), false/true)
```

- false: Bubbling phase
- true: Capturing phase

8) Events handling in JS.

1) Event bubbling:

- Event bubbling means propagation of an event is done from child element to ancestor elements in the DOM.

- When an event happens on an element, event first runs the handlers on it, then on its parent, then all the way up on other ancestors. Propagation path is bottom to top.

2) Event capturing:

- Event capturing means propagation of event is done from ancestor elements to child element in the DOM. Propagation path is top to bottom.

3) Event delegation:

- Event delegation in JavaScript is a pattern that efficiently handles events.
- Events can be added to a parent element instead of adding to every single element.
- It refers to the process of using event propagation to handle events at a higher level in the DOM than the element on which the event originated.
- This can be done on particular target element using the .target property of an event object.
- when you click the td, the event bubbles up to the table which handles the event.

Why Event Delegation?

- It is useful because the event can be listened to on multiple elements by using just one event handler.
- It also uses less memory and gives better performance.
- Apart from this, it also requires less time for setting up the event handler on element.

9) How to improve JS app performance when we firing event in app:

1) Normal event: It is about firing event every time.

2) Throttling event: Throttling is about firing event after every certain time.

3) Debouncing event: Debouncing is about firing event after every certain time, provided in between there was no event firing.

12) Asynchronous JS

Synchronous JavaScript:

As the name suggests synchronous means to be in a sequence, i.e. every statement of the JS code gets executed one by one. So, basically a statement has to wait for the earlier statement to get executed.

Let us understand this with the help of an example.

```
console.log("Hi"); // First  
  
document.write("Hari") ;// Second  
  
document.write("How are you"); // Third
```

In the above code snippet, the first line of the code 'Hi' will be logged first then the second line 'Hari' will be logged and then after its completion, the third line would be logged 'How are you'.

So as we can see the codes work in a sequence. Every line of code waits for its previous one to get executed first and then it gets executed.

Asynchronous JavaScript:

Asynchronous code allows the program to be executed immediately where the synchronous code will block further execution of the remaining code until it finishes the current one. This may not look like a big problem but when you see it in a bigger picture you realize that it may lead to delaying the User Interface.

```
console.log("Hi");  
  
setTimeout(() => {  
    console.log("Hello here what is go to happen");  
}, 2000);  
  
console.log("End");
```

- So, what the code does is first it logs in 'Hi' then rather than executing the setTimeout function it logs in End and then it runs the setTimeout function. At first, as usual, the Hi statement got logged in. As we use browsers to run JavaScript, there are the web APIs that handle these things for users.

- So, what JavaScript does is, it passes the setTimeout function in such web API and then we keep on running our code as usual. So it does not block the rest of the code from executing and after all the code its execution, it gets pushed to the call stack and then finally gets executed. This is what happens in asynchronous JavaScript.

```
// console.log('Hello');  
  
// setTimeout(() => {  
  
//   console.log('one');  
  
// }, 4000);
```

```
// Promise.resolve()
// .then(() => console.log('PR-1'))
// .then(() => console.log('PR-2'));
// console.log('bye');
// setTimeout(() => {
//   console.log('two');
// }, 0);
```

1) Callback functions

- A JavaScript callback is a function which is to be executed after another function has finished execution.

- A more formal definition would be - Any function that is passed as an argument to another function so that it can be executed in that other function is called as a callback function.

- When we have to process the data further then callback syntax will create callback hell situation. we can not read the callback code.

Why use callback function in JavaScript?

Asynchronous programming:

- Callbacks are used to handle the results of asynchronous operations, which means that the operation does not block the execution of the rest of the program.

- Instead, the program continues to run and the callback function is executed when the operation is complete.

2) Promises

Promises are a way to implement asynchronous programming in JavaScript(ES6 which is also known as ECMAScript-6). A Promise acts as a container for future values.

- A promise is a JavaScript object that allows you to make asynchronous calls. It produces a value when the async operation completes successfully or produces an error if it doesn't complete.

- let promise = new Promise(function(resolve, reject) { Do something and either resolve or reject})

- Promise has 3 states 1) pending 2) success 3) failure.

a) Promise.all()

Promise.all() will reject as soon as one of the Promises in the array rejects.

b) Promise.allSettled()

- whereas Promise.allSettled() waits for all the promises to settle (either resolve or reject) before returning an array of objects representing each promise's outcome.

- Promise.allSettled will never reject - it will resolve once all Promises in the array have either rejected or resolved.

c) Promise.race()

- The Promise.race() method returns a promise that fulfills or rejects as soon as one of the promises in an iterable fulfills or rejects, with the value or reason from that promise.

- It returns the winning promise only.

example:

```
var p1 = new Promise((res, rej) => {
  setTimeout(() => {
    res('pr-1');
  }, 3000);
});

var p2 = new Promise((res, rej) => {
  setTimeout(() => {
    res('pr-2');
  }, 2000);
});

var p3 = new Promise((res, rej) => {
  setTimeout(() => {
    res('pr-3');
  }, 5000);
});

Promise.all([p1, p2, p3]).then((res) => {
  console.log(res);
});

Promise.allSettled([p1, p2, p3]).then((res) => {
  console.log(res);
});

Promise.race([p1, p2, p3]).then((res) => {
  console.log(res);
});
```

3) async and await

- Async and await are built on promises. The keyword “async” accompanies the function, indicating that it returns a promise.

- Within this function, the await keyword is applied to the promise being returned.
- The await keyword ensures that the function waits for the promise to resolve.

API CALLS

The fetch() method in JavaScript is used to request data from a server. The request can be of any type of API that returns the data in JSON . The fetch() method requires one parameter, the URL to request, and returns a promise.

api for the get request:

```
fetch('url')  
  
.then(response => response.json())  
  
.then(data => console.log(data));
```

URL: It is the URL to which the request is to be made.

Return Value:

It returns a promise whether it is resolved or not. The return data can be of the JSON format. It can be an array of objects or simply a single object.

NOTE: Without options, Fetch will always act as a get request.

Making Post Request using Fetch:

Post requests can be made using fetch by giving options as given below:

```
var obj = {  
  userId: 101,  
  title: 'my title',  
  body: 'my body',  
};  
  
fetch('https://jsonplaceholder.typicode.com/posts', {  
  method: 'POST',  
  headers: {  
    'Content-type': 'application/json',  
  },  
  body: JSON.stringify(obj),  
}).then((res) => {  
  console.log(res);  
});
```

13) iterators and generators

- Iterator is a new concept introduced in ES-6. It's a kind of new mechanism to iterate or traverse through data structures.

- Arrays, strings, Maps, Sets all these data collections are iterable.

Iterable:(object)

An iterable is any object that implements a method whose key is symbol.iterator and symbol.iterator method is going to return an iterator object.

Iterator:(object) It returns by symbol.iterator()

- What is an iterator? An iterator is an object that is going to implement next method.

- This next method knows how to access elements in a collection, next method returns an object(IteratorResult object).

- That IteratorResult object contains two properties {value: any datatype, done: boolean}

```
var iterable = [1, 2, 3, 4, 5];
```

```
let iterator = iterable[Symbol.iterator]();
```

```
console.log(iterator);
```

```
console.log(iterator.next());
```

```
console.log(iterator.next());
```

```
console.log(iterator.next());
```

```
console.log(iterator.next());
```

```
console.log(iterator.next());
```

```
console.log(iterable.next());
```

```
let person = {
```

```
  fname: 'Hari',
```

```
  lname: 'Ravilla',
```

```
};
```

```
person[Symbol.iterator] = function () {
```

```
  var properties = Object.keys(person);
```

```
  var count = 0;
```

```
  var isDone = false;
```

```
  var next = () => {
```

```
    if (count >= properties.length) {
```

```

    isDone = true;
  }
  return { done: isDone, value: this[properties[count++]] };
};
return { next };
};
for (var v of person) {
  console.log(v);
}

```

By default iterable data structures

-array, -Map, -Set, -String, -Generator object, -arguments object

By default not iterable data structures

-object

Generator:

-Generators can help you to pause & resume the code.

-It is a function which can return multiple values in phases.

-The function* is the keyword used to define a generator function.

-Yield is an operator which pauses the generator.

-Yield itself is capable of returning any value.

-Generators are iterable.

-Next () method returns an object, which has two keys.

1) value

2) boolean

-Generator function returns an iterator object.

-To execute generator function we have to call next() method.

-When you want to come out of generator function or terminate it you can write return() method,

Writing yield inside finally will not allow the return() to terminate gen function.

Note: When a generator function is called, it does not call the function instead it returns a generator object.

Note: next method will start the execution till the yield operator. Next method returns an object which has two keys {value: any datatype, done: true/false}

14) ES-6 Modules

- Consider a scenario where parts of JavaScript code need to be reused. ES6 comes to rescue with the concept of Modules.
- A module organizes a related set of JavaScript code. A module can contain variables and functions.
- A module is nothing but a chunk of JavaScript code written in a file.
- By default, variables and functions of a module are not available for use.
- Variables and functions within a module should be exported so that they can be accessed (imported) from within other files.

Named Exports

Named exports are distinguished by their names. There can be several named exports in a module. A module can export selected components using the syntax given below –

Syntax 1

-using multiple export keyword

export component1

export component2

...

...

export componentN

Syntax 2

Alternatively, components in a module can also be exported using a single export keyword with {} binding syntax as shown below –

-using single export keyword

export {component1,component2,...,componentN}

Default Exports

Modules that need to export only a single value can use default exports. There can be only one default export per module.

Syntax

export default component_name

However, a module can have one default export and multiple named exports at the same time.

Importing Named Exports

While importing named exports, the names of the corresponding components must match.

Syntax

import {component1,component2..componentN} from module_name

However, while importing named exports, they can be renamed using the as keyword.

```
import {original_component_name as new_component_name }
```

Importing Default Exports

Unlike named exports, a default export can be imported with any name.

Syntax

```
import any_variable_name from module_name
```

Note: We have to do extra 2 tasks 1) type: "module" 2) add .js to file extension

15) ES-6 Tooling

1) Babel

=> Babel is a transpiler for ES6 to ES5, so that browser can understand the JS.

=> Babel can convert the advanced js and jsx into pure js, which is understood by browser.

=> It allows web developers to take advantage of the newest features of the JS language.

This is just for example to show how babel works

==> npm init --yes

==> npm install babel-cli@6.26.0 babel-core@6.26.0 babel-preset-env@1.6.1 --save-dev

==> Create build folder

==> In scripts: "babel": "babel --presets env index.js -o build/index.js"

==> npm run babel

Note: It will create new js file in build folder with ES-5 syntax, We mentioned index.js file in script to run, But if we have more files to run we need to add more files. Solution for this is we have to use Webpack it will take all files at a time and convert into ES-5 then make these files into bundle. Before creating bundle, each file will be taken by babel to convert into ES-5. After completion of conversion, all files will be bundled by webpack.

2) Webpack

=> Webpack is an open-source JavaScript module bundler. It is a build tool that is primarily used for bundling assets of a web application, such as JavaScript files, CSS stylesheets, and images. Webpack takes various modules and their dependencies and bundles them into a single or multiple output files, typically optimized for deployment in a web browser.

=> Webpack can also optimize code and assets for production, reducing the file size of the application and improving performance.

Note: Webpack runs our code through babel and it converted to ES-5.

==> Finally make changes in index.html:

- remove type="module" in script tag and
- change src path, serve the "dist/main.bundle.js"

- In scripts: "build": "webpack -w" This is for automatic build for every changes made.

- run everytime to build: num run build

16) ES-6 Destructuring

- The destructuring is a JavaScript expression that makes it possible to unpack values from arrays or unpack properties from objects, into individual variables.

1) Objects

2) Arrays

17) Browser APIs (Timer functions)

1) setTimeout

`setTimeout(function, milliseconds)`

Executes a function, after waiting a specified number of milliseconds.

2) setInterval

`setInterval(function, milliseconds)`

Same as `setTimeout()`, but repeats the execution of the function continuously.

3) clearTimeout

The `clearTimeout()` method clears a timer set with the `setTimeout()` method.

```
var myTimeout = setTimeout(function, milliseconds);
```

```
clearTimeout(myTimeout);
```

4) clearInterval

The `clearInterval()` method clears a timer set with the `setInterval()` method.

```
let myVar = setInterval(function, milliseconds);
```

```
clearInterval(myVar);
```

18) Miscellaneous topics (related to functions)

1) IIFE (Immediately Invocable Function Expression) (ES-5)

=>An IIFE is a way to excute functions immediately as soon as they are created.

=>The most usecase of an IIFE is to restrict the scope of variables to local so that they don't pollute the global context.

=>It helps in making our variables and methods private.

=>Any function or variable defined inside IIFE can not be accessed outside of the IIFE block, thus preventing global scope from getting polluted.

=>It's often used to create private variables and functions.

2) Function currying?

=>Function currying is a technique of transforming (converting) a function with multiple parameters into multiple functions with single parameter.

=>It helps to avoid passing same value again and again.

3) first class function?

=>A programming language is said to have First-class functions when functions in that language are treated like any other variable.

=>For example, in such language, a function can be passed as an argument to other functions, can be returned by another function and can be assigned as a value to a variable.

4) higher order function?

=>A function that receives another function as an argument or that returns a new function or both is called Higher-order function. Higher-order functions are only possible because of the First-class functions

5) pure function?

=>Pure function always returns the same result if the same arguments are passed in.

=>It does not depend on any state, or data, change during a program's execution. It must only depend on its input arguments .

=>They do not have any side effects like network or database calls and do not modify the arguments which are passed to them.

=>We can predict the output from pure function.

6) impure function?

=>Any function that changes the internal state of one of its arguments or the value of some external variable is an impure function.

=>They may have any side effects like network or database calls and it may modify the arguments which are passed to them.

=>We can not predict output from impure function.

7) Recursion function:

- A function that calls itself is recursion. Recursion in JavaScript is a recursive call of the function to itself, where the function calls itself, again and again, (recursively) until given condition becomes false.

calculate factorial:

```
function factorial(n) {  
  if (n <= 0) {  
    return 1;  
  } else {
```



```
    return n * factorial(n - 1);  
  }  
}
```

```
let result = factorial(5);
```

```
console.log(result);
```

19) Types of errors in javascript

1) Syntax error:

The error occurs when you use a predefined syntax incorrectly.

ex :

```
var obj = { name = 'hari'}
```

2) Reference error:

A ReferenceError occurs when you try to access a variable that doesn't exist in memory.

ex:

```
console.log(x);
```

3) Type error:

A TypeError occurs when the variable exists, but the operation you're trying to perform on it is not appropriate for the type of value it contains.

ex :

```
var x = 100;
```

```
x.push(10);
```

20) Errors handling in JavaScript:

- Handling errors in JavaScript is essential to ensure your code is clean and provides a good user experience. JavaScript provides several mechanisms to handle errors effectively. Here are some common methods:

Using try...catch:

- The try...catch block allows you to handle exceptions (errors) that occur within a specific block of code. You can place the code that might throw an error inside the try block, and if an error occurs, it will be caught and processed in the catch block.

try => try statement lets you test a block of code for error.

catch => catch statement lets you handle the error.

throw => throw statement lets you create custom errors.

```
try{
```

```
  code to be tested for error
```

```
}  
  
catch(error){  
    error handling  
}
```

5) Type conversion

Type conversion is the process of converting data of one type to another.

For example: converting String data to Number.

There are two types of type conversions in JavaScript.

Implicit Conversion - automatic type conversion by js.

Explicit Conversion - manual type conversion by developer.

JavaScript Implicit Conversion

In certain situations, JavaScript automatically converts one data type to another. This is known as implicit conversion.

Example 1: Implicit Conversion to String

1) numeric string used with + gives => string type as output

```
let result;  
  
result = '3' + 2;  
console.log(result) // "32"  
  
result = '3' + true;  
console.log(result); // "3true"  
  
result = '3' + undefined;  
console.log(result); // "3undefined"  
  
result = '3' + null;  
console.log(result); // "3null"
```

Note: When a number is added to a string, JavaScript converts the number to a string before concatenation.

Example 2: Implicit Conversion to Number

1) numeric string used with -, /, * results => number type as output.

```
let result;  
  
result = '4' - '2';  
console.log(result); // 2  
  
result = '4' * 2;
```

```
console.log(result); // 8
```

```
result = '4' / 2;
```

```
console.log(result); // 2
```

Example 3: Non-numeric String Results to NaN

=> non-numeric string used with -, /, * results to NaN type as output.

```
let result; result = 'hello' - 'world';
```

```
console.log(result); // NaN
```

```
result = '4' - 'hello';
```

```
console.log(result); // NaN
```

Example 4: Implicit Boolean Conversion to Number

=> if boolean is used, true is 1, false is 0

```
let result;
```

```
result = '4' - true;
```

```
console.log(result); // 3
```

```
result = 4 + true;
```

```
console.log(result); // 5
```

```
result = 4 + false;
```

```
console.log(result); // 4
```

Note: JavaScript considers 0 as false and all non-zero number as true. And, if true is converted to a number, the result is always 1.

Example 5: null Conversion to Number

=> null is 0 when used with number

```
let result;
```

```
result = 4 + null;
```

```
console.log(result); // 4
```

```
result = 4 - null;
```

```
console.log(result); // 4
```

Example 6: undefined used with number, boolean or null

=> Arithmetic operation of undefined with number, boolean or null gives NaN

```
let result;
```

```
result = 4 + undefined;
```

```
console.log(result); // NaN
result = 4 - undefined;
console.log(result); // NaN
result = true + undefined;
console.log(result); // NaN
result = null + undefined;
console.log(result); // NaN
```

JavaScript Explicit Conversion

You can also convert one data type to another as per your needs. The type conversion that you do manually is known as explicit type conversion.

In JavaScript, explicit type conversions are done using built-in methods.

1. Convert to Number Explicitly

To convert numeric strings and boolean values to numbers, you can use `Number()`. For example,

```
let result;
// string to number
result = Number('324');
console.log(result); // 324
// boolean to number
result = Number(true);
console.log(result); // 1
result = Number(false);
console.log(result); // 0
```

In JavaScript, empty strings and null values return 0. For example,

```
let result;
result = Number(null);
console.log(result); // 0
let result = Number(' ');
console.log(result); // 0
```

If a string is an invalid number, the result will be NaN

```
let result;
result = Number('hello');
```

```
console.log(result); // NaN
result = Number(undefined);
console.log(result); // NaN
result = Number(NaN);
console.log(result); // NaN
```

You can also generate numbers from strings using `parseInt()`, `parseFloat()`, unary operator `+` and `Math.floor()`. For example,

```
let result;
result = parseInt('20.01');
console.log(result); // 20
result = parseFloat('20.01');
console.log(result); // 20.01
result = +'20.01';
console.log(result); // 20.01
result = Math.floor('20.01');
console.log(result); // 20
```

To convert other data types to strings, you can use either `String()` or `toString()`. For example,

```
//number to string
let result;
result = String(324);
console.log(result); // "324"
result = String(2 + 4);
console.log(result); // "6"
//other data types to string
result = String(null);
console.log(result); // "null"
result = String(undefined);
console.log(result); // "undefined"
result = String(NaN);
console.log(result); // "NaN"
result = String(true);
```

```
console.log(result); // "true"
result = String(false);
console.log(result); // "false"
// using toString()
result = (324).toString();
console.log(result); // "324"
result = true.toString();
console.log(result); // "true"
```

To convert other data types to a boolean, you can use Boolean().

In JavaScript, undefined, null, 0, NaN, "" converts to false. For example,

```
let result;
result = Boolean("");
console.log(result); // false
result = Boolean(0);
console.log(result); // false
result = Boolean(undefined);
console.log(result); // false
result = Boolean(null);
console.log(result); // false
result = Boolean(NaN);
console.log(result); // false
```

All other values give true. For example,

```
result = Boolean(324);
console.log(result); // true
result = Boolean('hello');
console.log(result); // true
result = Boolean(' ');
console.log(result); // true
```

Truthy values and falsy values in javascript

- 1) (false); // false
- 2) (undefined); // false

```
3) (null);    // false
4) ("");      // false
5) (NaN);     // false
6) (0);       // false
=> (true);    // true
=> ('hi');    // true
=> (1);       // true
=> ([]);      // true
=> ([0]);     // true
=> ([1]);     // true
=> ({});      // true
=> ({ a: 1 }); // true
```

23) Data structures

In JavaScript, keys of objects are always strings or symbols. Even if you use a number as a key, it gets converted to a string. Here's an example to illustrate this:

```
let obj = {
  name: "Alice",
  age: 25,
  1: "one",
  true: "boolean",
  null: "null",
  undefined: "undefined",
  symbolKey: Symbol("symbolValue")
};

for (let key in obj) {
  console.log(`${key}: ${typeof key}`);
}
```

In this example, even though some keys appear to be numbers, booleans, null, or undefined, they are all converted to strings. The output will be:

```
1: string
name: string
age: string
true: string
```

null: string

undefined: string

symbolKey: string

Symbols are a special case. If a symbol is used as a key, it does not show up in a regular for...in loop or Object.keys() method. To handle symbols, you can use Object.getOwnPropertySymbols.

Here's how you can identify both string and symbol keys in an object:

```
let obj = {
  name: "Alice",
  age: 25,
  [Symbol("symbolKey")]: "symbolValue"
};

// Check string keys
for (let key in obj) {
  console.log(`${key}: ${typeof key}`);
}

// Check symbol keys
let symbolKeys = Object.getOwnPropertySymbols(obj);
for (let sym of symbolKeys) {
  console.log(`${sym.toString()}: ${typeof sym}`);
}
```

In JavaScript, keys of objects are always strings or symbols. Even if you use a number as a key, it gets converted to a string. Here's an example to illustrate this:

javascript

Copy code

```
let obj = {
  name: "Alice",
  age: 25,
  1: "one",
  true: "boolean",
  null: "null",
  undefined: "undefined",
  symbolKey: Symbol("symbolValue")
}
```



```
};

for (let key in obj) {
  console.log(`${key}: ${typeof key}`);
}
```

In this example, even though some keys appear to be numbers, booleans, null, or undefined, they are all converted to strings. The output will be:

```
1: string
name: string
age: string
true: string
null: string
undefined: string
symbolKey: string
```

Symbols are a special case. If a symbol is used as a key, it does not show up in a regular for...in loop or Object.keys() method. To handle symbols, you can use Object.getOwnPropertySymbols.

Here's how you can identify both string and symbol keys in an object:

```
let obj = {
  name: "Alice",
  age: 25,
  [Symbol("symbolKey")]: "symbolValue"
};

// Check string keys
for (let key in obj) {
  console.log(`${key}: ${typeof key}`);
}

// Check symbol keys
let symbolKeys = Object.getOwnPropertySymbols(obj);
for (let sym of symbolKeys) {
  console.log(`${sym.toString()}: ${typeof sym}`);
}
```

This will output:

```
name: string
```

age: string

Symbol(symbolKey): symbol

In summary, JavaScript object keys are either strings or symbols. Regular object keys (including those that look like numbers or booleans) are converted to strings, and symbol keys need to be handled separately using `Object.getOwnPropertySymbols()`.

When to use map:

With map, you can use any type (and values) as keys.

Map provides better performance for large quantities of data.

Use a map for better performance when adding and removing data frequently.

When to use an object:

Objects can only use strings, and symbols as keys.

Objects are perfect for small to medium-sized sets of data.

Objects have better performance and are easier to create.

Map

Map is a collection of keyed data items, just like an Object. But the main difference is that Map allows keys of any type.

how to create empty

```
var obj = {}
```

```
var map = new Map();
```

how to initialize with values

```
var obj = {name: 'sachin', age: 40}
```

```
const map = new Map([ [ 'one', 1 ], [ 'two', 2 ] ]);
```

how to add new properties

```
obj[color] = 'red'
```

```
obj.color = 'red'
```

`map.set(key, value)` -- stores the value by the key.

how to get or access values

```
obj.key
```

```
obj['key']
```

`map.get(key)` -- returns the value by the key, undefined if key doesn't exist in map.

how to check specific key is there

The `in` operator checks if a key exists in the object, including keys from the prototype chain.

The `hasOwnProperty` method checks if a key exists directly on the object, not considering keys from the prototype chain.

```
console.log('name' in obj)
```

```
console.log(obj.hasOwnProperty('name'))
```

`map.has(key)` -- returns true if the key exists, false otherwise.

how to delete specific key

```
delete obj.name;
```

`map.delete(key)` -- removes the value by the key.

how to delete all keys

Using a loop with the delete operator

```
for (let key in obj1) {  
  if (obj1.hasOwnProperty(key)) {  
    delete obj1[key];  
  }  
}
```

Setting the object to an empty object

```
obj = {};
```

```
console.log(obj); // {}
```

`map.clear()` -- clears the map

how to find length

alt:1

```
let length = Object.keys(obj).length;
```

alt:2

```
let length = 0;
```

```
for (let key in obj) {  
  if (obj.hasOwnProperty(key)) {  
    length++;  
  }  
}
```

alt:3

```
let length = Object.getOwnPropertyNames(obj).length;
```

map.size -- returns the current element count.

how to loop keys

alt:1

```
for (let key in obj) {  
  if (obj.hasOwnProperty(key)) {  
    console.log(`${key}: ${obj[key]}`);  
  }  
}
```

alt:2

```
let keys = Object.keys(obj);  
for (let i = 0; i < keys.length; i++) {  
  let key = keys[i];  
  console.log(`${key}: ${obj[key]}`);  
}
```

alt:3

```
Object.keys(obj).forEach(key => {  
  console.log(`${key}: ${obj[key]}`);  
});
```

Similar to objects, there are three methods you can use for looping over Maps:

map.keys() returns an iterable containing the keys

map.values() returns an iterable containing the values

map.entries() returns an iterable containing the [key, value] pairs

how to convert

```
let map = new Map(Object.entries(obj));
```

```
let obj = Object.fromEntries(map);
```

how to destructure

```
let { one, two } = obj;
```

```
let { one, two } = map;
```

```
console.log(one) // undefined
```

// But you can destructure it similar to an array where you destructure by the order items were added into the map

```
let [ firstEntry, secondEntry ] = map;  
console.log(firstEntry) // ["one", 1]  
console.log(secondEntry) // ["two", 2]
```

- remember the regular Object? it would convert keys to string.
- Map keeps the type, so these two are different.
- As we can see, unlike objects, keys are not converted to strings. Any type of key is possible.
- Map can also use objects as keys.

map vs weakMap

In a Weak Map, every key can only be an object and function. It is used to store weak object references.

One main difference when using a WeakMap is that the keys have to be objects, not primitive values. Which means they will pass by reference.

So why use a WeakMap? The major advantage of using a WeakMap over a Map is memory benefits.

Objects that are not-reachable get garbage collected, but if they exist in as a key in another reachable structure then they won't get garbage collected.

WeakMap does not prevent garbage-collection of it's key objects.

WeakMaps only have the following methods: get, set, delete, has.

It does not support for clear, size, keys, values, forEach

set vs weekSet

It is used to store a collection of objects similar to that of set the only difference is that these values are only object and not some particular data type.

Sets can store any value. WeakSets are collections of objects only.

WeakSet does not have size property.

WeakSet does not have clear, keys, values, entries, forEach methods.

WeakSet is not iterable.

summary

- Use Map when you need to associate values with keys.
- Use WeakMap when you want to associate values with keys and the keys should not prevent garbage collection.
- Use Set when you need a collection of unique values.
- Use WeakSet when you need a collection of unique objects, and the objects should not prevent garbage collection.

24) Browser Object Model (Bom)

The Browser Object Model (BOM) allows JavaScript to "talk to" the browser. The Browser Object Model (BOM) is a set of objects provided by web browsers that allows JavaScript to interact with the browser environment.

Here are some key objects in the Browser Object Model:

The Browser Object Model (BOM) is used to interact with the browser.

The default object of browser is window means you can call all the functions of window by specifying window or directly. For example:

You can use a lot of properties (other objects) defined underneath the window object like document, history, screen, navigator, location.

The window object represents a window in browser. An object of window is created automatically by the browser.

Window is the object of browser, it is not the object of javascript. The javascript objects are string, array, date, math etc.

1. window Object:

- A window object is created automatically by the browser itself.
- The window object is the top-level object in the BOM hierarchy and represents the browser window.
- It contains properties and methods that allow to interaction with the browser window.
- The window object is supported by all browsers. It represents the browser's window.
- All global JavaScript objects, functions, and variables automatically become members of the window object.
- Global variables are properties of the window object.
- Global functions are methods of the window object.
- Even the document object (DOM) is a property of the window object.
- Two properties can be used to determine the size of the browser window.

window.innerHeight - the inner height of the browser window (in pixels)

window.innerWidth - the inner width of the browser window (in pixels)

-timing functions:

window.setTimeout()

window.setInterval()

window.clearTimeout()

window.clearInterval()

-popup functions:

window.alert()

`window.confirm()`

`window.prompt()`

`window.open()` - open a new window

`window.close()` - close the current window

`window.moveTo()` - move the current window

`window.resizeTo()` - resize the current window

2. document Object:

The document object represents the HTML document loaded in the browser window.

It provides methods and properties to manipulate the content and structure of the document.

// Examples of document object properties

`console.log(document.title);` // Title of the document

`console.log(document.body);` // Body element of the document

// Examples of document object methods

`document.getElementById('myElement');` // Returns the element with the specified ID

`document.createElement('div');` // Creates a new HTML element

3. navigator Object:

The navigator object provides information about the browser and the user's system.

Examples of navigator object properties

`console.log(navigator.appName);`

The appName property returns the application name of the browser.

`console.log(navigator.platform);`

Operating system platform

`navigator.language`

The language property returns the browser's language:

Is The Browser Online?

`navigator.onLine;`

The onLine property returns true if the browser is online:

4. screen Object:

The screen object represents the user's screen and provides information about its size and resolution.

// Examples of screen object properties

```
console.log(screen.width); // Width of the screen
```

```
console.log(screen.height); // Height of the screen
```

5. location Object:

The location object provides information about the current URL and allows you to navigate to different URLs.

The window.location object can be used to get the current page address (URL) and to redirect the browser to a new page.

```
// Examples of location object properties
```

```
console.log(location.href); // Full URL of the current page
```

```
console.log(location.pathname); // Path of the current URL
```

```
// Example of location object method
```

```
location.assign('https://www.example.com'); // Navigates to the specified URL
```

window.location.href returns the href (URL) of the current page

window.location.hostname returns the domain name of the web host

window.location.pathname returns the path and filename of the current page

window.location.protocol returns the web protocol used (http: or https:)

The window.location.assign() method loads a new document.

```
<h3>The window.location object</h3>
```

```
<script>
```

```
function newDoc() {
```

```
    window.location.assign("https://www.youtube.com")
```

```
}
```

```
</script>
```

6) History object

The history object contains the URLs visited by the user (in the browser window).

The history object is a property of the window object.

The history object is accessed with:

window.history or just history:

back() Loads the previous URL (page) in the history list

(Will only work if a previous page exists in your history list)

forward() Loads the next URL (page) in the history list

(Will only work if a next page exists in your history list)

`go()` Loads a specific URL (page) from the history list
(Will only work if the previous pages exist in your history list)

`length` Returns the number of URLs (pages) in the history list

`history.go(0)` reloads the page.

`history.go(-1)` is the same as `history.back()`.

`history.go(1)` is the same as `history.forward()`.

HTML

What is HTML

HTML stands for Hypertext Markup Language, which is used to structure content on the web page. HTML uses tags to structure content on the web. HTML tags are enclosed in angle brackets and come in pairs: an opening tag and a closing tag. HTML is not a programming language, it's a markup language that uses tags to format text, images, audio, video on a webpage.

Here are some key points about HTML:

1) Markup Language:

HTML is a markup language, not a programming language. It uses tags to display text, images, and other content on a web page.

2) Structural Hierarchy:

HTML documents are organized hierarchically. They have a root element `<html>` that contains two child sections: `<head>` (which contains meta-information and links to external resources) and `<body>` (which contains main content displayed in web browser).

3) Elements and Tags:

HTML documents are made up of elements. Elements are defined using tags. For example, a paragraph is defined using the `<p>` tag: `<p>This is a paragraph.</p>`. Tags typically come in pairs, with an opening tag `<p>` and a closing tag `</p>`. The content is placed between these tags.

4) Attributes:

Many HTML elements can have attributes that provide additional information about the element or modify its behavior. For example, the `` element can have an "src" attribute to specify the image file source.

-Web developers use HTML to create the structure and content of web pages, and then they use CSS to style those pages and JavaScript to add interactivity and functionality.

Basic tags

`!DOCTYPE>`: Specifies the document type and version of HTML being used.

`<html>`: The root element that contains all other HTML elements on the page.

`<head>`: Contains metadata about the document, such as the title, and linked stylesheets or scripts and meta tags.

<title>: Defines the title of the webpage, which appears in the browser's title bar or tab.

<meta>: Provides metadata about document.

<link>: Links external resources such as stylesheets and icons.

<script>: Embeds or references JavaScript code/file.

<style>: Defines CSS for styling the page.

<body>: Contains the main content of the webpage, including text, images, audio, video and other elements.

meta tags

The <meta> tag defines metadata about an HTML document. Metadata is data (information) about data.

<meta> tags always go inside the <head> element, and are typically used to specify character set, page description, keywords, author of the document, and viewport settings.

Metadata will not be displayed on the page, but is machine parsable.

Metadata is used by browsers (how to display content or reload page), search engines (keywords), and other web services.

1) <meta charset="UTF-8">:

Specifies the character encoding for the document. UTF-8 is widely used for encoding text in various languages. To display an HTML page correctly, a web browser must know which character set to use. The HTML5 specification encourages web developers to use the UTF-8 character set.

UTF-8 covers almost all of the characters and symbols in the world!

2) <meta name="description" content="Your page description">:

Provides a brief summary or description of the web page's content. Search engines often use this information in search results.

3) <meta name="keywords" content="keyword1, keyword2, keyword3">:

Specifies a list of keywords or key phrases related to the page's content. However, this tag is no longer widely used by search engines to determine rankings.

4) <meta name="author" content="Author Name">:

Identifies the author of the web page's content.

5) <meta name="viewport" content="width=device-width, initial-scale=1.0">:

Sets the initial scale and width of the viewport, important for responsive design.

6) <meta http-equiv="refresh" content="20">:

Automatically redirects the browser to another page after a specified time interval (in this case, 20 seconds).

=> html tags vs html elements vs html attributes.

HTML Attributes

Html attributes are the properties of html element and it is used to give extra information about element to the browser. Every attribute will be in the key value pair.

Block-level and Inline elements

-A block-level element always starts on a new line and takes up the full width available.

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

HTML Elements

1) Heading elements

`<h1>content</h1>`

h1 to h6

- Heading tags are used to define headings or titles for sections of a web page.
- We have different levels of headings from h1 to h6.
- It is a block-level element, which means it starts in a new line and takes 100% width.

2) Paragraph element

`<p>content</p>`

- The `<p>` tag is used to define and format text as a paragraph.
- It is a block-level element, which means it starts on a new line and takes 100% width.

3) image element

``

- The `` tag is used to display images on a web page. It is a self-closing tag, which means it doesn't have a closing tag.
- It is inline element, It takes only required space for image.
- Html attributes are the properties of html element and it is used to give extra information about element to the browser. Every attribute will be in the key value pair.

Attributes:

src alt width height title

4) List elements:

``: Defines an unordered (bulleted) list.

``: Defines an ordered (numbered) list.

``: Represents a list item.

`<dl>`: Defines a description list.

`<dt>`: Represents a term in a description list.

<dd>: Provides the description for a term.

Attributes:

- type: we can use type attribute for only ordered and unordered elements.
 - start and reversed attributes are only for ordered list.
 - start will only work for numbers.
 - These are only for ordered list
 - Number (default)
 - type = 'a/A/I/i'
 - reversed
 - start
 - Alpha
 - lower case (a)
 - UPPER CASE (A)
 - Roman
 - lower case (i)
 - UPPER CASE (I)
 - These are only for unordered list
- type:
- disc (default) - circle - square - none

5) Anchor element

content

- The <a> tag (anchor tag) in HTML is used to create a links in the webpage.
- By using links we can navigate from one webpage to other web page(inter navigation).
- By using links we can navigate from one section to another section with in the same webpage (intra navigation).

Navigation:

- 1) Inter navigation (navigation between documents)
- 2) Intra navigation (navigation between sections with in document)

Attribute:

href => hyper reference

href = 'path of another webpage' or '#id value of section in same page'

target => Specifies where to open the linked document.

_blank

_parent

_self (default)

_top

_blank => Opens the linked document in a new window or tab.

_self => Opens the linked document in the same tab as it was clicked (default).

frameName => Opens the linked document in the named iframe.

6) Button element

- The <button> tag in HTML is used to define the clickable button. <button> tag is used to submit the content. The images and text content can be used inside <button> tag.

Attributes

disabled

autofocus

type

Following are the three values of type attribute:

Button: The button value is used for the clickable button.

Submit: The submit value is used for submitting the form-data.

Reset: The reset value is used to reset the values of a form to its initial values.

7)

- The
 tag in HTML document is used to create a line break in a text. It is self closing.

- Don't use br tag for the margin between two paragraphs, use CSS margin property instead.

8) <hr>

The <hr> tag in HTML stands for horizontal rule and is used to insert a horizontal rule to divide or separate document sections. The <hr> tag is an empty tag, and it does not require an end tag.

Attributes

width, color, size, align.

9) Center element

- The <center> HTML element is a block-level element that displays its block-level or inline contents centered horizontally within its containing element. - This tag is not supported in HTML5. CSS's property is used to set the alignment of the element instead of the center tag in HTML5.

10) HTML entities

=> A commonly used HTML entity is the non-breaking space.

=> © It will show copy right symbol.

=> ® It will show trademark symbol.

11) Division element

<div> </div>

- It is used to create divisions or sections inside the html documents.

- The <div> tag is used as a container for HTML elements.

- Div tag is Block level tag.

- It is a generic container tag.

- It is used to group various tags of HTML so that sections can be created.

- As we know Div tag is block-level tag, the div tag contains entire width.

Hence, every div tag will be started from a new line, and not the same line.

12) Span element

- The tag is an inline container used to mark up a part of a text, or a part of document.

Note:

Div and Span Tags: These are generic containers that can be used to group and style content. <div> is a block-level element, and is an inline element.

- <div>This is a block-level container. </div>

- This is an inline container.

13) Formatting Elements

Formatting elements were designed to display special types of text with css styles:

Bold text: or

HTML5 provides two tags to make text bold: and .

Both tags have the same effect of making the enclosed text bold, but is preferred because it indicates that the text is important for semantic purposes.

Italicized text: <i> or

HTML5 provides two tags to make text italic: `<i>` and ``. Both tags have the same effect of making the enclosed text italic, but `` is preferred because it indicates that the text has emphasis for semantic purposes.

Strike-through text: `` or `<s>`

HTML5 provides two tags to make text strike-through: `<s>` and ``. Both tags have the same effect of making the enclosed text strike-through, but `` is preferred because it indicates that the text has been deleted for semantic purposes.

Superscript and subscript text: `<sup>` or `<sub>`

HTML5 provides two tags to make text superscript and subscript: `<sup>` and `<sub>`, respectively.

u and ins

HTML5 provides two tags u and ins. Both tags have the same effect of making underline.

`<ins>` is used for semantic purpose. The `<ins>` tags means content inserted after it was first published. The `<u>` tag is simply for underlining and has no meaning.

small vs big

In HTML, the `<small>` tag is used to emphasize small text or information that is of secondary significance, such as copyright and legal disclaimer.

The `<small>` tag can be used to mark up a copyright notice or license agreement.

HTML `<big>` tag was used to increase the text font size one level bigger than the document's base font size.

`` - Bold text

`` - Important text

`<i>` - Italic text

`` - Emphasized text

`<mark>` - Highlighted Text

`` - Deleted text

`<s>` - text strick

`<ins>` - Inserted text

`<u>` - Underline text

`<sub>` - Subscript text

`<sup>` - Superscript text

`<small>` - Smaller text

`<big>` tag - Big Text

14) Table elements

- <table> - Defines a table
- <caption> - Defines a table caption
- <thead> - Groups the header content in a table
- <tbody> - Groups the body content in a table
- <tfoot> - Groups the footer content in a table
- <tr> - Defines a row in a table (rows)
- <th> - Defines a header cell in a table (columns)
- <td> - Defines a data cell in a table (columns)

Attributes:

1) border = '1' | '5' | '10' => if we use border then no need of frame and rules.

frame = box | border | above | below | hside | vside | lhs | rhs | void

rules = all | rows | cols | groups | none

2) width => It will effect the table header & table body & table footer.

height => It will give effect only to table body but not effect the table header(thead) & (tfoot) footer.

3) cellspacing => It gives space between all cells.

cellpadding => It gives space with in cells.

4) colspan => merging columns in specific row.

rowspan => merging rows in specific column.

5) text-align => text align horizontally.

vertical-align => text align vertically.

6) align = left | center | right

7) bgcolor => bg color to table.

Colspan

Colspan is an attribute which assigns multiple columns to a cell of a table. The number of columns depends on the value entered by you in colspan="" attribute.

rowspan

Rowspan in table, works similar to the colspan for columns, but here, we assign multiple rows to a cell using an attribute rowspan=""

horizontal alignment (text in table)

The text-align property sets the horizontal alignment (like left, right, or center) of the content in <th> or <td>.

By default, the content of `<th>` elements are center-aligned and the content of `<td>` elements are left-aligned.

vertical alignment (text in table)

The vertical-align property sets the vertical alignment (like top, bottom, or middle) of the content in `<th>` or `<td>`.

By default, the vertical alignment of the content in a table is middle (for both `<th>` and `<td>` elements).

15) Form element

Form element is used to take input from an user to send data to server and then db.

The `<form>` element is a container for different types of input elements, such as: text fields, checkboxes, radio buttons, submit buttons, etc.

Form Elements:

The HTML `<form>` element can contain one or more of the following form elements.

`<form>` => Defines an HTML form for user input.

`<input>` => `<input>` element can be displayed in many ways, depending on the type attribute.

`<label>` => Defines a label for an `<input>` element.

The for attribute of the `<label>` tag should be equal to the id attribute of the `<input>` element to bind them together.

`<textarea>` => Defines a multiline input control (text area).

The `<textarea>` element is often used in a form, to collect user inputs like comments or reviews. The size of a text area is specified by the cols and rows attributes.

`<button>` => Defines a clickable button.

`<fieldset>` => Groups related elements in a form.

`<legend>` => Defines a caption for a `<fieldset>` element.

`<select>` => Defines a drop-down list.

`<option>` => Defines an option in a drop-down list. (The `<option>` tags inside `<select>` element define the available options in the drop-down list). (selected attribute)

`<optgroup>` => Defines a group of related options in a drop-down list.

`<datalist>` => Specifies a list of pre-defined options for input controls.

-The `<datalist>` tag is used to provide an "autocomplete" feature for `<input>` elements. Users will see a drop-down list of pre-defined options as they input data.

-The `<datalist>` element's id attribute must be equal to the `<input>` element's list attribute (this binds them together).

Form attributes:

- autocomplete: 'on' (in form opening tag).
- novalidate => form-data (input) should not be validated when submitted.
- onSubmit: form data will be submitted when click on submit button.

Input element Attributes

Input element attributes

1.type: text | password | email | button | radio | checkbox | number | color | date | reset | submit | tel | range | time | month

The type attribute is used to specify the type of the input element. Its default value is text.

2.placeholder: Placeholder attribute is used to specify hint that describes the expected value of an input field.

3.value: The value attribute is used to specify the value of the input element.

4.name: The name attribute is used to specify the name of the input element.

5.maxlength: This property is used to specifies the maximum number of characters allowed in an <input> element

6.size: This property is used to specifies the width of <input> element(default 20 chars)

7.required: The required attribute specifies that an input field must be filled out before submitting the form.

8.disabled: The disabled attribute specifies that the element should be disabled. The disabled attribute can be set to keep a user from using the < input > element until some other condition has been fulfilled.

9.readonly: The readonly attribute specifies that an input field is read-only. A read-only input field cannot be modified. A form will still submit an input field that is readonly, but will not submit an input field that is disabled.

Note:

The default width of an input field is 20 characters.

- The for attribute of the <label> tag should be equal to the id attribute of the <input> element to bind them together.
- Name attribute should be same value for all radio buttons to select any one of them.
- The <input type="submit"> defines button for submitting the form data to form-handler(function).

16) marquee element

The <marquee> tag is a container for creating scrollable text or images within a web page from either left to right or vice versa, or top to bottom or vice versa. But this tag has been deprecated in the new version of HTML, i.e., HTML 5

Attributes

width

height

direction : left (default right to left) | right | up | down

behaviour : slide | scrolling (default) | alternate

scrollDelay : default 85 (to make slow)

scrollAmount : default 6 (to make speed)

bgColor

loop : times of loop (The default value of loop is INFINITE)

17) semantic elements

Semantic elements = elements with a meaning.

A semantic element clearly describes its meaning to both the browser and the developer.

Examples of non-semantic elements: <div> and - Tells nothing about its content.

Examples of semantic elements: <form>, <table>, and <article> - Clearly defines its content.

<table> : Defines table.

<form> : Defines form.

<details> : Defines additional details that the user can view or hide.

<summary> : Defines a visible heading for a <details> element.

<header> : Defines a header for a document or section.

<nav> : Defines navigation links (nav contains <a> tag).

<footer> : Defines a footer for a document or section.

<main> : Specifies the main content of a document.

<section> : The <section> element defines a section in a document.

<article> : Defines an article. Examples of where the <article> element can be used.

(Forum posts, Blog posts, User comments, Product cards, Newspaper articles)

<aside> : Defines content aside from the page content. The <aside> content should be indirectly related to the surrounding content.

<figure> : Specifies self-contained content, like illustrations, diagrams, photos etc.

<figcaption> : Defines a caption for a <figure> element.

Note: All semantic tags contain no default styles as like div.

18) iframe

The iframe in HTML stands for Inline Frame.

- Use the <iframe> tag to embed another document within the current HTML document.

- The HTML iframe "name" attribute is used to specify a reference for an <Iframe> element.
- The iframe is basically used to show a webpage inside the current web page.
- The "src" attribute is used to specify the URL of the document that occupies the iframe.

Attributes

src

width

height

name

frameBorder

-An iframe can be used as the target frame for a link. The target attribute of the link must refer to the name attribute of the iframe.

-By default iframe has border, we can remove border by using css also.

19) Media Elements

- HTML5 introduces media elements like <audio> and <video> for displaying and playing audio and video content.
- The <video> tag is used to embed video content in a document, such as some videos.
- The <audio> tag is used to embed sound content in a document, such as some audios.

<audio controls autoplay muted loop>

<source src="">

</audio>

<video width="320" height="240" controls autoplay muted loop>

<source src="">

</video>

Attributes

width

height

controls

autoplay

muted

poster

loop

- The <source> element allows you to specify alternative audio files.
- controls: attribute adds audio controls, like play, pause, and volume.
- autoplay: it will work with muted for videos.
- loop: Specifies that the video will start over again, every time it is finished.
- muted: Specifies that the audio output of the video should be muted.
- poster: Specifies an image to be shown while the video is downloading, or until the user hits the play button.

20) Web storage

=> Local storage

=> Session storage

With web storage, web applications can store data locally within the user's browser.

Before HTML5, application data had to be stored in cookies, included in every server request.

Web storage is more secure, and large amounts of data can be stored locally, without affecting website performance.

Unlike cookies, the storage limit is far larger (at least 5MB) and information is never transferred to the server.

It is used to store data on the client side. It has no expiration time, so the data in the LocalStorage exists always till the user manually deletes it.

HTML web storage provides two objects for storing data on the client:

window.localStorage - stores data with no expiration date

window.sessionStorage - stores data for one session (data is lost when the browser tab is closed)

=> Store

```
localStorage.setItem("name", "Hari");
```

=> Retrieve

```
document.getElementById("result").innerHTML = localStorage.getItem("name");
```

=> remove

```
localStorage.removeItem('name')
```

```
localStorage.clear()
```

22. Block-level vs Inline

Block

| | | | |
|-----------|------|-----------|----------|
| <address> | <hr> | <abbr> | <map> |
| <article> | | <acronym> | <object> |

| | | | |
|--------------|------------|----------|------------|
| <aside> | <main> | | <output> |
| <blockquote> | <nav> | <bdo> | <q> |
| <canvas> | <noscript> | <big> | <samp> |
| <dd> | |
 | <script> |
| <div> | <p> | <button> | <select> |
| <dl> | <pre> | <cite> | <small> |
| <dt> | <section> | <code> | |
| <fieldset> | <table> | <dfn> | |
| <figcaption> | <tfoot> | | <sub> |
| <figure> | | <i> | <sup> |
| <footer> | <video> | | <textarea> |
| <form> | Inline | <input> | <time> |
| <h1>-<h6> | <a> | <kbd> | <tt> |
| <header> | | <label> | <var> |

CSS CONTENT

1. What is CSS

CSS stands for Cascading Style Sheet. It is a styling language, by using CSS we can style the html content or elements on webpage by using CSS properties. CSS properties will be in key value pair.

2. How many ways we can add css to html elements

1) Inline

- By using style attribute we can apply css directly to that html element. By using inline style we can apply only css to only one element at a time.

2) Internal

- By using style tag in head tag we can apply css to many html elements in the same html file at a time html elements.

3) External

- By using external style sheet we can apply css to many html elements in different html files at a time. We can link external css file to html file by using link tag in head tag.

3. CSS selectors

CSS selectors are used to select the HTML elements you want to style.

- We can divide CSS selectors into several categories:

1. Simple selectors (select elements based on tag, id, class) (3 selectors)

2. Combinator selectors (select elements based on a specific relationship between them)

a) Descendent selector => div p {} - all childs

b) Child selector => div>p {} - direct childs

c) Adjacent sibling => div+p {} - next one (sibling)

d) General sibling => div~p {} - next all siblings

3. Pseudo-class selectors (select elements based on a certain state)

4. Pseudo-elements selectors (select and style a part of an element)

5. Attribute selectors (select elements based on an attribute or attribute value) (4 selectors)

6. Universal Selector (The universal selector (*) selects all HTML elements on the page)

7. Grouping Selector (To group selectors, separate each selector with a comma)

Allows you to apply the same styles to multiple selectors.

Example: h1, h2, h3 { font-family: Arial, sans-serif; }

1.Type (Element) Selector:

Selects elements by their HTML tag name.

Example: p { color: blue; }

2.Class Selector:

Selects elements by their class attribute.

Example: .my-class { font-weight: bold; }

3.ID Selector:

Selects a single element by its unique ID attribute.

Example: #my-id { background-color: yellow; }

4.Universal Selector:

Selects all elements on the page.

Example: * { margin: 0; }

5.Descendant Selector (all childs):

Selects an element that is a descendant of another element.

Example: ul li { list-style: square; }

6.Child Selector (direct child):

Selects an element that is a direct child of another element.

Example: ul > li { font-style: italic; }

7.Adjacent Sibling Selector (next sibling):

Selects an element that is immediately preceded by another element.

Example: h2 + p { color: red; }

8.General Sibling Selector (next all siblings):

Selects elements that are siblings of a specified element.

Example: `h2 ~ p { font-size: 18px; }`

9.Pseudo-Class Selector:

Selects elements based on a specific state or interaction, such as `:hover`, `:active`, and `:focus`.

Example: `a:hover { text-decoration: underline; }`

10.Pseudo-Element Selector:

Selects and styles a part of an element's content, such as

`=> ::first-letter` `::first-line` and `::before` and `::after`

Example: `p::before { content: "Note: "; }`

Pseudo-elements in CSS are used to style a specific part of an element. They are denoted by double colons (`::`) and are often used with the `::before` and `::after` pseudo-elements to insert content before or after an element. Here's an example of using the `::before` pseudo-element to add content before an element:

```
.myclass::before {  
  content: "This is content";  
  font-weight: bold;  
  color: red;  
}
```

```
.myclass::after {  
  content: "This is content";  
  font-weight: bold;  
  color: white;  
}
```

11.Grouping Selector:

Allows you to apply the same styles to multiple selectors.

Example: `h1, h2, h3 { font-family: Arial, sans-serif; }`

12.Attribute Selector:

Selects elements based on the value of their attributes.

Example: `input[type="text"] { border: 1px solid #ccc; }`

13.Attribute Value Starts With Selector:

Selects elements whose attribute values start with a specified string.

Example: `[href^="https://"] { color: blue; }`

14.Attribute Value Ends With Selector:

Selects elements whose attribute values end with a specified string.

Example: `[src$=".jpg"] { border: 1px solid #ccc; }`

15.Attribute Value Contains Selector:

Selects elements whose attribute values contain a specified string.

Example: `[alt*="cat"] { border: 2px solid orange; }`

5. Text Properties

CSS (Cascading Style Sheets) provides a wide range of text properties that allow you to control the appearance and layout of text within HTML elements. A list of common text properties in CSS.

1.color:

Sets the color of the text.

Example: `color: red;`

2.direction:

Sets the text direction, such as ltr (left-to-right) or rtl (right-to-left).

Example: `direction: rtl;`

3.text-transform:

Modifies the capitalization of text, e.g., uppercase or lowercase or capitalize.

Example: `text-transform: uppercase;`

4.text-shadow:

It gives shadow to text.

`text-shadow: x-axis y-axis blur color`

`text-shadow: 10px 0px 10px red`

5.text-align:

Aligns the text horizontally within its container.

Example: `text-align: center;`

6.text-decoration:

Controls decorations like underline, overline, and line-through.

Example: `text-decoration: underline;`

7.text-indent:

Sets the indentation of the first line of text within an element.

Example: text-indent: 20px;

8.font-family:

Specifies the font family for the text.

Example: font-family: Arial, sans-serif;

9.font-size:

Sets the size of the text.

Example: font-size: 16px;

10.font-weight:

Defines the thickness or boldness of the text.

Example: font-weight: bold;

11.font-style:

Specifies the style of the text, such as italic.

Example: font-style: italic;

12.line-height:

Sets the height of a line of text, controlling spacing between lines.

Example: line-height: 1.5;

13.letter-spacing:

Adjusts the space between individual characters.

Example: letter-spacing: 2px;

14.word-spacing:

Controls the spacing between words in a block of text.

Example: word-spacing: 4px;

These are some of the essential text properties in CSS. By using these properties, you can customize the appearance and layout of text on your web pages, making it an integral part of web design and typography.

6. Background Properties

1.background-color:

<style>

.example {

background-color: #F0F0F0;

}

</style>

```
<div class="example">
```

This is a div with a colored background.

```
</div>
```

2.background-image:

```
<style>
```

```
.example {
```

```
background-image: url('image.jpg');
```

```
}
```

```
</style>
```

```
<div class="example">
```

This is a div with a background image.

```
</div>
```

3.background-repeat: (repeat (default) | repeat-x | repeat-y | no-repeat)

=> Repeating horizontally and vertically

```
.example-repeat {
```

```
background-image: url('repeating-pattern.png');
```

```
background-repeat: repeat;
```

```
}
```

=> Repeating horizontally only

```
.example-repeat-x {
```

```
background-image: url('horizontal-pattern.png');
```

```
background-repeat: repeat-x;
```

```
}
```

=> Repeating vertically only

```
.example-repeat-y {
```

```
background-image: url('vertical-pattern.png');
```

```
background-repeat: repeat-y;
```

```
}
```

=> No repetition

```
.example-no-repeat {
```

```
background-image: url('single-image.jpg');
```

```
background-repeat: no-repeat;
}
```

ex:

```
<style>
```

```
.example {
  background-image: url('image.jpg');
  background-repeat: no-repeat;
}
```

```
</style>
```

```
<div class="example">
```

This is a div with a non-repeating background image.

```
</div>
```

4.background-position: left top | left center | left bottom | right top | right center |
----- right bottom | center top | center center | center bottom

```
<style>
```

```
.example {
  background-image: url('image.jpg');
  background-position: center top;
}
```

```
</style>
```

```
<div class="example">
```

This is a div with a centered top-aligned background image.

```
</div>
```

5.background-size:

=> Original size (auto)

```
.example-auto {
  background-image: url('large-image.jpg');
  background-size: auto;
}
```

=> Scale to cover the content area

```
.example-cover {  
    background-image: url('image.jpg');  
    background-size: cover;  
}
```

=> Scale to fit within the content area

```
.example-contain {  
    background-image: url('image.jpg');  
    background-size: contain;  
}
```

=> Specific size in pixels

```
.example-specific-size {  
    background-image: url('image.jpg');  
    background-size: 200px 150px;  
}
```

=> Relative size with percentages

```
.example-percent-size {  
    background-image: url('image.jpg');  
    background-size: 50% 75%;  
}
```

Ex:

```
<style>
```

```
.example {  
    background-image: url('image.jpg');  
    background-size: cover;  
}
```

```
</style>
```

```
<div class="example">
```

This is a div with a background image that covers the entire element.

```
</div>
```

Note:

The 'cover' tells the browser to cover the whole area of an element.

The 'contain' tells the browser to show the whole image without losing image.

6.background-attachment: (scroll | fixed)

=> Background image scrolls with content (default)

```
.example-scroll {  
    background-image: url('scrolling-background.jpg');  
    background-attachment: scroll;  
}
```

=> Background image remains fixed in place

```
.example-fixed {  
    background-image: url('fixed-background.jpg');  
    background-attachment: fixed;  
}
```

Ex:

```
<style>
```

```
.example {  
    background-image: url('image.jpg');  
    background-attachment: fixed;  
}
```

```
</style>
```

```
<div class="example">
```

This is a div with a fixed background image.

```
</div>
```

These are just a few examples of how you can use CSS background properties to style and enhance the backgrounds of HTML elements. You can combine these properties and adjust their values to achieve the desired visual effects in your web pages.

7. Border properties

In CSS (Cascading Style Sheets), you can use various properties to control the appearance of borders around elements such as text, images, and containers. Here are some common border properties in CSS:

1)border-width:

This property sets the width of the border. You can specify the width in pixels, ems, rems, percentages, or other units.

```
border-width: 2px;
```

2)border-color:

This property sets the color of the border. You can use color names, hexadecimal values, RGB values, or other color notations.

```
border-color: #FF0000;
```

3)border-style:

This property sets the style of the border, and it can take values like "solid," "dashed," "dotted," "double," "none,".

```
border-style: solid;
```

4)border-radius:

This property sets the radius of the corners of an element, creating rounded corners. You can specify different values for each corner or use a single value for all corners.

The border-radius property is actually a shorthand property for the

border-top-left-radius,

border-top-right-radius,

border-bottom-right-radius

border-bottom-left-radius

```
border-radius: 10px;
```

5)border-image:

This property allows you to use an image as a border instead of a solid color. It's a more advanced property and requires defining an image to use as a border.

```
border-image: url(border-image.png) 27 27 27 27 round round;
```

```
border-image-source: url('./102.PNG');
```

```
border-image-slice: 30%;
```

```
border-image-width: 20px;
```

```
border-image-repeat: repeated, stretched(default).
```

```
border-image-outset: 20px
```

border-image is a shorthand property that lets you use an image or CSS gradient as the border of an element.

The border-image property can be applied to any element, except internal table elements (e.g. tr, th, td) when border-collapse is set to collapse.

The border-image-outset property specifies the amount by which the border image area extends beyond the border box

border:

This is a shorthand property that combines border-width, border-style, and border-color into a single declaration.

border: (width) 1px (style)solid (color) #000;

8. Margin and Padding properties

margin

The CSS margin property is used to give space around an element on a web page. It defines the area outside the element's border. Margins are used to create space and separation between elements.

1)margin-top: 10px;

2)margin-right: 20px;

3)margin-bottom: 15px;

4)margin-left: 25px;

5)margin: 10px 20px 15px 25px; (top, right, bottom, left)

6)margin: 10px 20px (Top and bottom have 10px margin, right and left have 20px margin)

7)margin: 10px 20px 15px (Top has 10px margin, right and left have 20px margin, bottom has 15px margin)

8)margin: 10px (all sides)

9)margin: auto

You can set the margin property to auto to horizontally center the element within its container.

The element will then take up the specified width, and the remaining space will be split equally between the left and right margins.

Margin Collapse

Top and bottom margins of elements are sometimes collapsed into a single margin that is equal to the largest of the two margins.

This does not happen on left and right margins! Only top and bottom margins!

padding

Padding is used to create space within an element, between the element's content and its border.

1)padding-top: 10px;

2)padding-right: 20px;

3)padding-bottom: 15px;

4)padding-left: 25px;

5)padding: 10px 20px 15px 25px; /* top, right, bottom, left */

6)padding: 10px 20px

7)padding: 10px 20px 15px

8)padding: 10px (all sides)

margin vs padding

In CSS, both margin and padding are properties used to control spacing, but they affect the layout of elements in different ways:

Margin:

-Margin controls space outside an element, between the element's border and neighboring elements.

Margins create separation between elements.

-Margin values can be positive (adding space) or negative (reducing space).

-Margin do not have a background color or visible area, they only affect the spacing between elements.

-Margins can "collapse" under certain conditions, meaning that if two adjacent elements have margins, the larger of the two margins is used, and they don't stack.

Ex:

```
div {  
  margin: 10px;  
}
```

In this example, a div element will have a 10-pixel margin around it, creating space between it and neighboring elements.

Margin collapse

The concept of "margin collapsing" in CSS refers to how margins between adjacent elements are computed when they are touching or close to each other. When two or more vertical margins meet or overlap under specific conditions, they don't simply add up or stack on top of each other as you might expect. Instead, they collapse, and the larger of the two margins is used to create the space between the elements. Here's a more detailed explanation:

=> Margins Collapsing Conditions:

Margins can collapse in the following situations:

When two or more block-level elements are vertically adjacent, meaning they are stacked on top of each other in the document flow.

When there is no padding, border, or inline content separating the elements.

When the margins are of the same type (e.g., both are top margins or both are bottom margins).

=> Result of Margin Collapsing:

When margins collapse, the larger of the margins is used, and the smaller margin effectively disappears in terms of creating spacing.

This means that if one element has a margin of 10px, and the adjacent element has a margin of 20px, the effective spacing between them will be 20px, not 30px.

=> Preventing Margin Collapsing:

You can prevent margin collapsing by introducing a non-collapsible element between the two elements. For example, adding padding, a border, or inline content between the elements will stop margin collapsing.

Example of margin collapsing:

```
p {  
  margin: 10px 0;  
}  
  
div {  
  margin: 20px 0;  
}
```

In this example, if a `<p>` element and a `<div>` element are placed next to each other, the effective vertical space between them will be 20px, not 30px, because the larger of the two margins (20px) takes precedence

Padding:

- Padding controls the space inside an element, between the element's content and its border.

- Padding is used to create space within an element, affecting the content's position within the element's boundaries.

- Padding values can be positive (adding space) and are never negative.

- Padding has a background color and visible area. It extends the background of the element.

EX:

```
div {  
  padding: 10px;  
}
```

In this example, a `div` element will have a 10-pixel padding inside it, pushing the content away from the border.

Note: Inline elements vertical padding

CSS, inline elements, by default, do not respect vertical padding in the same way that block-level elements do. Vertical padding on inline elements will not push away adjacent inline elements. Instead, it may cause overlapping or wrapping of the content within the inline element.

When you apply vertical padding to an inline element, such as `` or a text within a paragraph, it increases the height of the element but doesn't create space around it like it does with

block-level elements. This means that if you have multiple inline elements with padding, they might overlap or wrap to the next line depending on the available space within their container.

If you want to create space around inline elements, you can use margin instead of padding, or you can change the display property of the elements to inline-block or inline-flex, which will make them behave more like block-level elements, respecting vertical padding and margin.

```
<p>
  <span class="inline">This is some text.</span>
  <span class="inline">This is some text with padding.</span>
</p>
```

CSS:

```
span.inline {
  padding: 10px;
  margin: 10px;
}
```

In this example, the padding will increase the height of each inline element, but the adjacent inline elements won't be pushed away. They may overlap or wrap to the next line depending on the available width of the containing element. If you want to create space between these inline elements, you would need to use margin or consider changing the display property to make them behave like block-level elements.

9. Width and height properties

Note: width and height in % => depends on parent's width and height.

- width = Width will be always fixed irrespective of screen size.
- min-width = min-width is fixed for given size and beyond min-width it depends on screen size.
- max-width = max-width is fixed for given size and below max-width it depends on screen size.
- height = Height is fixed it will not be depend on screen size.
- min-height = min-height is fixed for given size and beyond min-height depends on screen size.
- max-height = max-height is fixed for given size and below max-height it depends on screen size.

Ex:

In CSS, height, min-height, and max-height are properties used to control the height of an element, such as a container or a block-level element. They serve different purposes and have distinct effects:

height:

The height property sets the exact height of an element to a specific value in pixels, ems, rems, percentages, or other length units. This value is often fixed and doesn't change unless you explicitly modify it.

If the content inside the element exceeds the specified height, it may overflow and be hidden or affect the layout of surrounding elements unless you use additional CSS properties to manage overflow, like `overflow: hidden` or `overflow: scroll`.

```
.element {  
    height: 200px;  
}
```

`min-height`:

The `min-height` property sets the minimum height that an element should have. It ensures that the element is at least as tall as the specified value, but it can expand beyond that height if needed to accommodate content or other factors.

If the content within the element is taller than the specified `min-height`, the element's height will adjust to fit the content.

```
.element {  
    min-height: 100px;  
}
```

`max-height`:

The `max-height` property sets the maximum height that an element can have. It restricts the height to not exceed the specified value. If the content or other factors would cause the element to be taller than the `max-height`, it will be limited to that height, and scrollbars may appear to allow users to access the overflow content.

```
.element {  
    max-height: 300px;  
}
```

You can use these properties individually or in combination to control the height of elements in your web page layout. They are particularly useful when dealing with responsive web design, ensuring that elements adapt to various screen sizes and content while maintaining a defined range of height.

10. Display Property

In CSS, the `display` property is used to control how an HTML element is displayed in the web browser. It defines the type of box an element generates, which in turn affects its layout and rendering on the web page. There are various `display` property values that determine the element's behavior. Here are some common `display` property values:

1. `block`:

The element is displayed as a block-level element.

It takes up the full width available in its parent container and starts on a new line.

Examples of block-level elements include `<div>`, `<p>`, and `<h1>`.

2. `inline`:

The element is displayed as an inline-level element.

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

Examples of inline-level elements include ``, `<a>`, and ``.

3.inline-block:

Combines characteristics of both inline and block.

The element is inline, but it can have its own width and height, as well as margin and padding.

Elements with this display type can be positioned inline, but they can have block-level styling.

Note:

=> inline The element doesn't start on a new line and only occupy just the width it requires. You can't set the width or height.

=> inline-block It's formatted just like the inline element, where it doesn't start on a new line. BUT, you can set width and height values.

=> block element will start on a new line and occupy the full width available. And you can set width and height values.

4.none:

The element is not displayed at all, effectively making it invisible.

It doesn't occupy any space in the layout.

5.table:

The element is displayed as a table element.

It can have child elements with table-row, table-cell, and other table-related display values.

Float and Clear

float: left | right | none

It is used to float an element either to the left or to the right in its parent container.

float: left

float an element to the left in its parent container.

=> Block level element will be converted into inline block element.

=> Remaining childs will be pushed up in container.

=> Height of parent container will be reduced.

float: right

float an element to the right in its parent container.

float: none

It is used to remove float effect from an already floated element.

`clear: left | right | both`

It is used to clear the effect of floating on the parent container.

`clear: left`

It clears the effect of floating left on its parent container.

`clear: right`

It clears the effect of floating left on its parent container.

`clear: both`

It clears the effect of floating left and right on its parent container.

parent to be normal

```
elementselector::after{
```

```
content: " ",
```

```
display: block,
```

```
clear: left
```

```
}
```

6.flex:

The element becomes a flex container, and its children become flex items.

It is used to create flexible and responsive layouts, allowing elements to adjust their sizes based on available space.

7.grid:

The element becomes a grid container, and its children become grid items.

It is used to create grid-based layouts, providing fine control over the arrangement of items.

`display: none` vs `visibility: hidden` vs `opacity: 0`

In CSS, `display: none`, `visibility: hidden`, and `opacity: 0` are three different ways to hide elements on a web page, and they have distinct behaviors and use cases.

`display: none`

When you apply `display: none` to an element, the element is completely removed from the layout of the page. It takes up no space, and the document flows as if the element does not exist.

Child elements of the hidden element are also hidden, and they do not affect the layout.

Events and interactions (e.g., clicking, hovering) cannot target hidden elements.

Good for situations where you want to completely remove an element from view and layout.

```
element {
```

```
display: none;
```

```
}
```

visibility: hidden

When you apply `visibility: hidden` to an element, the element is still part of the layout, but it is not visible. It still takes up space on the page, as if it were still visible. Child elements of the hidden element are also hidden, but they still affect the layout. Events and interactions can still target hidden elements. Useful when you want to hide an element but maintain its space in the layout.

```
element {
```

```
  visibility: hidden;
```

```
}
```

opacity: 0

When you set `opacity: 0`, the element becomes fully transparent, but it still occupies the same space in the layout.

Child elements of the transparent element remain visible and affect the layout.

Events and interactions can target the transparent elements.

Useful when you want to hide an element while maintaining its layout position and possibly animate the element's visibility.

```
element {
```

```
  opacity: 0;
```

```
}
```

In summary, the choice of which property to use (`display`, `visibility`, or `opacity`) depends on your specific requirements. If you want to completely remove the element and its space from the layout, `display: none` is suitable. If you want to hide the element but maintain its space in the layout, `visibility: hidden` is appropriate. If you want to hide the element but keep its space and potentially use transitions or animations, `opacity: 0` is a good choice.

11) Block level vs Inline elements default behaviour

In HTML, content placement within block-level and inline elements follows specific rules that determine how elements are displayed and how they interact with the surrounding content.

Block-Level Elements:

Block-level elements create distinct blocks or containers within the layout. They naturally start on a new line and take up the full available width of their parent container. Content within block-level elements is displayed in a "block" format.

Here's how content is typically placed within block-level elements:

On New Lines:

Block-level elements start on a new line, and subsequent block-level elements are displayed below the previous one. This means each block-level element creates a new "block" in the layout.

Full Width:

By default, block-level elements span the full width of their parent container, unless specified otherwise using CSS properties like width.

Vertical Stacking:

Block-level elements stack vertically, creating a clear separation between different sections or blocks of content. Common examples of block-level elements include paragraphs (<p>), headings (<h1>, <h2>, etc.), lists (,), and div containers (<div>).

```
<div>This is a block-level container.</div>
```

```
<div>Another block-level container.</div>
```

- Block-level elements naturally create distinct "blocks" within the layout. Content placed within block-level elements is organized into separate, vertically stacked sections.

- Block-level elements extend the full width of their parent container by default. This means that they create new lines for each block-level element, and each block-level element takes up the entire width available.

- Block-level elements are ideal for structuring and organizing the content into different sections or divisions of a web page.

Inline Elements:

Inline elements, flow within the content of their parent block-level elements. They don't create new lines or block-level structures themselves.

Here's how content is typically placed within inline elements:

Within Text Flow:

Inline elements flow within the text content of their parent block-level element. They do not create new lines, and their width adjusts to fit the content they contain.

No Line Breaks:

Inline elements do not introduce line breaks, and they wrap to the next line only if there is not enough horizontal space within their parent block-level element.

Horizontal Flow:

Content within inline elements flows from left to right (in left-to-right writing systems) or from right to left (in right-to-left writing systems).

```
<p>This is a <span>inline element</span> within a paragraph.</p>
```

In this example, the element is an inline element and flows within the text of the paragraph. It doesn't introduce new lines or separate blocks of content.

- Inline elements are designed to flow within the content of a block-level element. They do not create distinct blocks or new lines by default.

- Inline elements are often used for applying formatting and styling to specific parts of text or for embedding elements within a paragraph of text.

-By default, inline elements do not create a new block formatting context and flow within the content of their parent block-level element. They only take up as much horizontal space as necessary to contain their content.

-Inline elements do not accept width or height properties. The margin-top and margin-bottom properties do not create vertical spacing between inline elements in the same way they do for block-level elements.

To control the placement of content within both block-level and inline elements, you can use various HTML and CSS properties, such as display, position, and others, to modify their default behaviors and achieve the desired layout and structure in your web pages.

Content vs width vs height vs margin vs padding vs border

The behavior of content, width, height, margin, padding, and border for inline and block-level elements in HTML and CSS differs due to their default display characteristics.

Content:

Block-Level Elements:

Block-level elements create separate blocks or containers within the layout. They naturally start on new lines and contain their content within distinct blocks. Block-level elements can hold other elements and text content.

Inline Elements:

Inline elements flow within the text content of their parent block-level element. They do not create new lines or separate blocks and are typically used for styling portions of text or inline content.

Width and Height:

Block-Level Elements:

Block-level elements, by default, extend across the full width of their parent container. You can explicitly set their width using CSS to control the element's horizontal size, and you can set their height to control their vertical size.

Inline Elements:

Inline elements take up only as much horizontal space as necessary to contain their content. Applying a width or height property to inline elements doesn't typically have the same effect as it does for block-level elements. Inline elements are usually self-sizing based on their content.

Margin:

Block-Level Elements:

Block-level elements use the margin property to create spacing around them. Margins apply both vertically and horizontally, creating space outside the element's border. You can set margins for all four sides individually.

Inline Elements:

Inline elements can have horizontal margins (left and right), but their vertical margins (top and bottom) may not create the same spacing effects as they do for block-level elements. Margins often

affect the horizontal layout and the positioning of the inline elements in relation to surrounding content.

Padding:

Block-Level Elements:

Block-level elements use the padding property to create spacing within the element, affecting the space between the element's content and its border. Padding properties apply both vertically and horizontally and can be set for all four sides individually.

Inline Elements: Inline elements can have horizontal padding (left and right), but vertical padding (top and bottom) may not have the same impact. Padding for inline elements often affects the horizontal spacing and the positioning of the content within the element.

Border:

Block-Level Elements:

Block-level elements can have a border defined using the border property. The border surrounds the entire block and is often used for creating visual boundaries around content.

Inline Elements:

Inline elements can also have borders, but the effect might differ. Borders may not consistently create a clear block-like boundary around inline elements due to their inline flow. The border can be applied to individual sides (e.g., border-left, border-right) to control how it appears in relation to the inline content.

In summary, block-level elements are typically used to create distinct blocks of content with more control over their width, height, margins, padding, and borders. Inline elements, on the other hand, are used for styling inline content within a block-level element and may not have the same level of control over these properties due to their inline flow. However, you can modify the default behavior of elements using CSS to achieve the desired layout and styling.

Inline elements

Horizontal behaviour:

width: no effect

padding: work correctly

border: work correctly

margin: work correctly

Vertical behaviour:

height: no effect

margin: no effect

padding: go into block level element

border: go into block level element

12. Box model

The CSS box model is a container that contains multiple properties including borders, margins, padding, and the content itself. It is used to create the design and layout of web pages. It can be used as a toolkit for customizing the layout of different elements. The web browser renders every element as a rectangular box according to the CSS box model.

The CSS box-model and box-sizing are related concepts, but they serve different purposes when it comes to controlling how the dimensions of elements are calculated. Let me explain the difference between them:

The CSS box model, defines how an element's dimensions are calculated, taking into account the content, padding, border, and margin.

By default, when you set the width and height properties of an element, you are specifying the dimensions of the content area. The padding, border, and margin are added to these dimensions.

```
.box {  
  width: 200px;  
  height: 100px;  
  padding: 20px;  
  border: 2px solid #000;  
  margin: 10px;  
}
```

The total space the .box element occupies on the page is calculated by summing the content width, padding, border, and margin, as described in the previous answer.

Box-Sizing Property:

The box-sizing property allows you to control how an element's dimensions are calculated, specifically how width and height are interpreted.

The box-sizing property has two possible values:

content-box (default):

This is the default behavior, where width and height represent the dimensions of the content area. Padding, border, and margin are added to these dimensions.

border-box:

With this value, width and height include the content area, padding, and border. The margin is still outside of these dimensions. If you set an element's box-sizing property to border-box, the specified width and height values will include the padding and border. This can be helpful for creating more predictable layouts, as you can work with the total space an element occupies.

Here's an example of how to use the box-sizing property:

```
.box {  
  width: 200px;  
  height: 100px;
```

```
padding: 20px;
border: 2px solid #000;
margin: 10px;
box-sizing: border-box; /* Include padding and border in width and height */
}
```

In this case, the specified width and height values will include the padding and border, making it easier to control the overall dimensions of the element while ensuring that the content area remains a consistent size.

In summary, the CSS box-model defines the standard model for calculating element dimensions, and the box-sizing property allows you to change how width and height are interpreted, either as content dimensions or including padding and border.

13) Positions:

In CSS position property is used to position HTML element in HTML document.

While using position property, we take help of left, right, top, bottom and z-index properties to position HTML element.

1) static :

It is default position for all html elements we can not move element's position using top, left, right, bottom.

2) relative :

Element will be displayed as per normal flow of document.

It considers its own position as a reference point to move.

3) absolute :

Element will not be displayed as per normal flow of document.

It considers its positioned relative parent element as a reference point to move.

If no positioned parent then it takes top most parent(html/viewport) as reference point.

4) fixed :

Element will not be displayed as per normal flow of document.

It always considers html element(viewport) as a reference point to move,

So it stays (fixed) in the same position even when the page is scrolled.

5) sticky :

It is used to position and stick an HTML element to top edge of viewport.

Sticky positioning is a hybrid between relative and fixed positioning.

It will work initially as a relative (it will move certain extent) after that it will be fixed.

z-index:

The z-index property is typically used with positioned elements, such as relative, absolute, fixed. It does not have any effect on non-positioned (static) elements.

- Controls the stacking order of elements.
- Elements with higher z-index values appear in front of elements with lower values.
- If multiple elements have the same z-index, their stacking order is determined by their order in the HTML document. The element that appears later in the document will be on top.
- The z-index property only works on elements with a position value other than static. So, you need to set element's position property to relative, absolute, fixed for z-index to have effect.
- Negative values for z-index are allowed, and elements with negative z-index values will be placed behind elements with positive values.

15. Responsive Design

Media queries

-Media queries are essential for creating responsive web designs that adapt to various devices and screen sizes, providing a better user experience.

Media query is a technic of applying different styles to an element based on certain condition.

-They are commonly used to provide different styles for different devices by their features like type, width, height, orientation.

Syntax:

1) @media

Media queries use the @media rule and have a specific syntax.

```
@media screen and (max-width: 600px) {  
    /* CSS rules for screens with a width of 600px or less */  
}
```

In this example, the media query targets screens with a width of 600 pixels or less.

2) Media Types:

You can specify different media types (screen, print, speech) to apply styles in different contexts. The most commonly used media type is screen.

-screen

-print

-speech

3) Media Features:

Media features like max-width, min-width, min-height, max-height, orientation, and many others are used to specify the conditions under which the CSS rules should be applied.

-min-width

-max-width

-min-height

-max-height

-orientation (landscape, portrait)

Here are some common examples of media queries:

Max-width:

To apply styles when the viewport width is at most a certain size.

```
@media screen and (max-width: 768px) {  
  /* CSS rules for screens with a width of 768px or less */  
}
```

Min-width:

To apply styles when the viewport width is at least a certain size.

```
@media screen and (min-width: 1024px) {  
  /* CSS rules for screens with a width of 1024px or more */  
}
```

Orientation:

To apply styles based on the device's orientation (portrait or landscape).

```
@media screen and (orientation: landscape) {  
  /* CSS rules for landscape orientation */  
}
```

4) Operators (Combining Conditions)

You can combine multiple conditions using and, or, not, only to create more complex media queries.

```
@media screen and (min-width: 768px) and (max-width: 1024px) {  
  /* CSS rules for screens between 768px and 1024px wide */  
}
```

Media queries are essential for creating responsive web designs that adapt to various devices and screen sizes, providing a better user experience.

16. CSS Animations

Animation is a process of changing elements style in a given duration.

we can apply multiple css styles to an element during given duration by using @keyframe.

syntax

```
@keyframe animation-name{  
  keyFrame-selector{  
  }  
}
```

- keyFrame-selectors

"from" and "to" and we can give % also.

- You should configure the animation name and animation duration on an element.

properties on an element

1)animation-name: (compulsory)

2)animation-duration:(compulsory) (total animation time)

3)animation-delay: starts animation after some delay of time. (animation starts time)

4)animation-iteration-count: how many times animation need.

5)animation-direction: normal|reverse|alternate|alternate-reverse

6)animation-fill-mode: backwards|forwards|none|both (before & after animation styles)

7)animation-play-state: paused | running

8)animation-timing-function: linear|ease-in|ease-out|ease-in-out (speed of animation)

=> ease: Default value. The animation has a slow start, then fast, before it ends slowly.

=> linear: Play an animation with the same speed from beginning to end.

=> ease-in: The animation has a slow start.

=> ease-out: The animation has a slow end.

=> ease-in-out: The animation has both a slow start and a slow end.