node -v 🡪 returns the version of node installed

npm -v 🡪 returns the version of npm

npm install create-react-app 🡪 installs create-react-app module in node modules

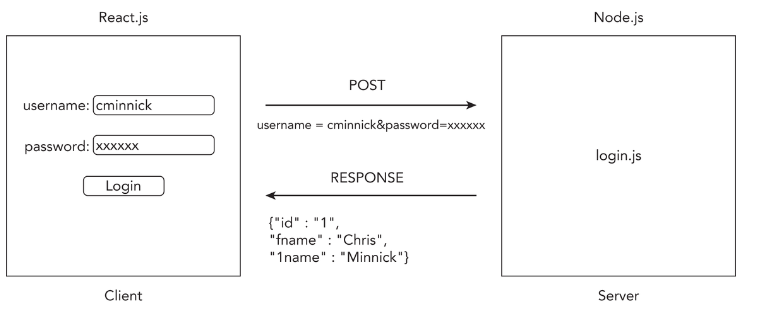
npx create-react-app my-firstapp OR 🡪 Creates a react app by name ‘my-FirstApp’

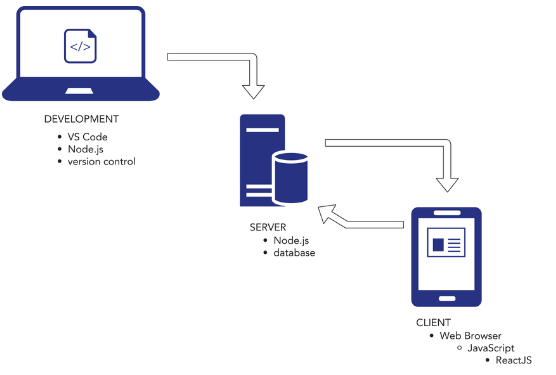
Open package.config and press Ctrl + S to resolve any build errors if you get know

npm install 🡪 It installs node modules folder from packages.json

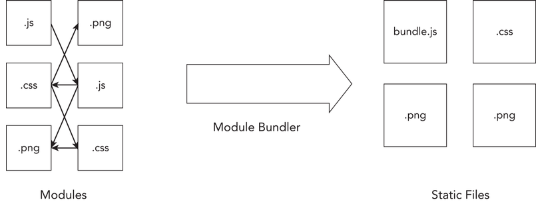
**Node JS**

* It started as a way to run Javascript on web servers so that same language can be used in both server side and client side





* Tasks done by Node.js
  + **Minification:** The process of removing white spaces, comments , line breaks making scripts, web pages and stylesheets more efficient and faster.
  + **Transpiling:** Not all browsers support the same version of javascript code. The process of converting one version of javascript code into another version is called as Transpiling. A screenshot of a computer

    Description automatically generated
  + **Module Bundling:** A typical website can make hundreds of scripts, stylesheets etc., If a browser were to download all of them it will slow down the page. Hence to overcome this we use bundlers. The main job of the bundler is to combine/bundle the javascript code so that it loads in the browser much faster. 
  + **Package Management:** As so many different programs are involved in JavaScript Development just installing, upgrading and keeping track of them can be quite complex. Hence we need a strong package manager.
  + **CSS Preprocessors:** A CSS preprocessor such as SCSS or LESS allows you to write style sheets that CSS lacks like variables, mathematical operations, functions, scope and nesting.
  + **Testing Frameworks:** Testing is an important part for any web project and the process of writing logic to test is a powerful tool.
  + **Build Automation:** If we a have complex manual process of writing code, testing, compiling, deploying then it’s better to automate where you can write a script or a program that can automate these things for us.

**NPM vs Yarn**

|  |  |  |
| --- | --- | --- |
| **Installation** | NPM comes pre-installed with Node.js  Ex: npm init 🡪 to initialize a project | Yarn needs to installed separately |
| **Speed** | Generally slower due to its sequential installation process. | Generally faster as it performs parallel installation of packages. |
| **Lock Files** | NPM uses package-lock.json for dependencies. | Yarn uses yarn.lock to lock dependencies. Yarn’s lock file ensures more consistency across different environments. |
| **Offline-Mode** | NPM doesn’t have built in offline capabilities. | Yarn supports offline-caching. Once package has been installed, yarn uses the cached-version without needing to re-download it. |
| **Deterministic Dependency Resolution:** | NPM 5 and above improved on this with package-lock.json | Provides deterministic dependency tree through yarn.lock ensuring same dependencies are installed every time. |
| **Command Syntax** | npm install , npm start | yarn add , yarn start |
| **Security** | NPM has audit command to fix vulnerabilities | Yarn also has security features and integrates with npm’s audit system. |
| **Workspaces** | Introduced workspaces in NPM 7, allowing for mono-repo support | Has workspace support |
| **Plug’n’Play** | NPM does not equivalent feature | Yarn 2 introduced Plug’n’Play feature that eliminates the need for having ‘node\_modules’ folder as it will directly reference the dependencies from cache. |
| **Community and Ecosystem** | NPM has larger user base with broader community | Yarn has huge community support but lesser than NPM. |

**React**

* Facebook designed React in response to its need to be able to efficiently update websites in response to events.
* Facebook wanted to create a way to more easily build applications that respond, or react to new data, rather than simply refreshing pages whether the underlying data has changed or not.
* This method of updating a user interface in response to data changes is called reactive programming.

*Create React App*

* It installs a tool chain for react development and configures a boilerplate react application we can use as starting point for our application.

|  |  |
| --- | --- |
| * NPM is Node package manager manages packages and dependencies in the project | * npx is Node Package Executor. It’s main job is to execute the binaries present in node\_modules/bin folder |
| * Allows developers to publish their own packages to the npm registry. | * Npx executes binaries locally or from npm registry without the need to install them globally. |
| * Handles versioning and dependency resolution. | * Directly executes binaries without needing a script in package.json |
| npm install express // installs locally  npm install -g express // installs globally  npm run start // runs a script in package.json | npx create-react-app my-app // runs a package without installing  npx eslint . // runs locally installed package |

To create an app name below are the rules :

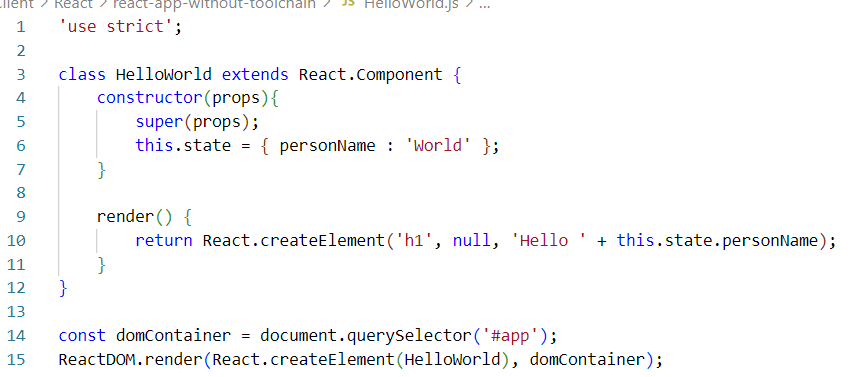
* It must be less than 214 characters long.
* The name can't start with a dot or underscore.
* The name can't have uppercase letters.
* It can't contain any characters that aren't allowed in URLs (such as ampersands and dollar signs) and that are “unsafe” in URLs (such as the percent symbol and spaces).

In addition to these rules, there are several common conventions for how Node.js packages, and therefore apps created using Create React App, are named:

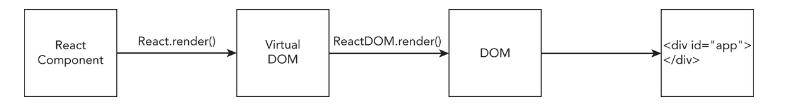
* Keep it simple and as short as possible.
* Use only lowercase letters.
* Use dashes in place of spaces.
* Don't use the same name as a common Node.js package.

**React CDN:**

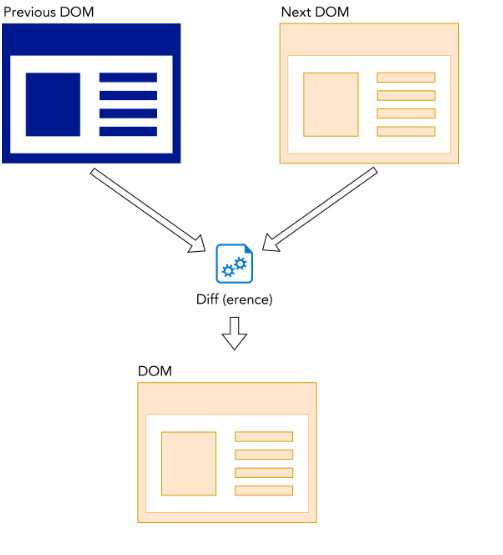
* React application can be created without any toolchain of modules.
* Simply copy the CDN links from here 🡪  <https://reactjs.org/docs/cdn-links.html>
* UMD stands for Universal Module Definition is what allows the CDN version of React to work in browsers without compile step.



* React.render() 🡪 Generates output for the component
* ReactDOM.render() 🡪 Causes the output to displayed in browser’s window.
* React just renders the components and how these components gets rendered to user is up to separate library.
* The library that handles rendering of React components in web browsers is called ReactDOM. If we want to react components to static HTML then we use ReactDOMServer.



* **DOM :** The document object model (DOM) is an internal representation of a web page. It converts HTML, styles and content into nodes that can be operated using JavaScript.
* A programmer writes React code to render a user interface, which results in a single React element being returned.
* ReactDOM's render method creates a lightweight and simplified representation of the React element in memory (this is the ***Virtual DOM***).
* ReactDOM listens for events that require changes to the web page.
* The ReactDOM.render method creates a new in-memory representation of the web page.
* The ReactDOM library compares the new Virtual DOM representation of the web page to the previous Virtual DOM representation and calculates the difference between the two. This process is called ***reconciliation***.
* ReactDOM applies just the minimal set of changes to the browser DOM in the most efficient way that it can and using the most efficient batching and timing of changes**.**



* React is a library for creating and putting together components to build user interfaces.
* Every component of React should have *Single Responsibility.*
* Composition vs Inheritance
  + In OOPs, it is common to create variations of the class that inherit properties from parent class.
  + Instead of creating multiple specific purpose components React suggests to create a generic purpose component that can be configured by passing data into it. Once we create such generic components we can create specific components by combining more generalized ones. This is called *Composition*.
  + React is Declarative meaning the user should specify what the user interface should look like and react will be render it.

**JSX:**

**JavaScript Modules:**

* JavaScript were in small size in early days but when javascript code started to increase the programmers wanted them to split into difference parts. This gave raise to modularization in JavaScript.
* First came ‘requireJS’ which used to load modules in AMD (Asynchronous Module Definition) way i.e.. all the imports in a module run prior to any of the code in those modules being executed.
* Later came ‘CommonJS’ which was built into Node.js and it became a way for modularization library. With ‘CommonJS’ we can export variables, modules, functions or objects in a file. But ‘CommonJS’ used to load the modules synchronously , parsing and executing each module as it’s loaded.

A diagram of a computer program

Description automatically generated

* **ES Modules :** ECMAScript Modules (ESA) features Asynchronous Module Definition like ‘requireJS’ and has simple syntax like ‘commonJS’.
  + export creates modules where as import statement imports modules into JavaScript Code.
  + *import { shippingMethods, calculateShippingCharges } from './modules/ecommerce-utilities.js'*

We can import individual items from the file by surrounding them with curly braces.

* Default export 🡪 We can use ‘default’ keyword in the module so that it gets exported and when someone imports it then they can directly specify without curly braces.

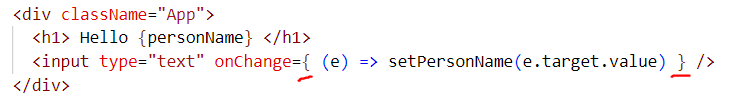
function calculateShippingCharge(weight,shippingMethod){ import calculateShippingCharge from ./calculateShippingCharge.js;

  // do something here

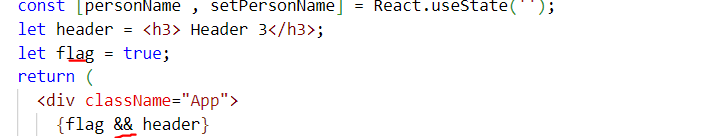
   }

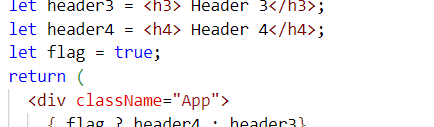
   export default calculateShippingCharge;

* Both import and export statements need to be at the top level of your JavaScript file—that is, not inside of a function or any other statement.
* Imports must be done before any other statements in a module.
* import and export can only be used inside modules (not inside of ordinary JavaScript files).
* JSX is an XML extension to JavaScript, it’s a way to write JavaScript code using XML. React uses JSX to create custom components.
* Transpilation: It’s a process of converting from one version of javaScript to another for old browsers compatibility.
* Babel: It’s a tool used to convert JSX into plain vanilla JavaScript. Babel is incorporated into *Create React App*. It’s online version is here 🡪 <https://babeljs.io/repl>
* **Rules for writing JSX**:
  + HTML elements in JSX must be in lowercase.
  + Elements with no child nodes should end with />
  + All elements must be closed.
  + JSX uses camelCase.
  + As JSX gets converted to JavaScript it may be possible that element or attribute name that we use in our JSX code can cause errors in our compiled program so to avoid that certain HTML reserved words should not be used
    - ***class*** becomes ***className***.
    - ***for*** becomes ***htmlFor***.
    - onclick becomes onClick.
    - tabindex becomes tabIndex.
* Syntax of JSX:
  + User defined elements can have custom attributes after React 16. Prior to React 16 it is a good practice to have ‘data-‘ attributes defined for DOM elements instead of custom attributes.
  + Use curly braces for include literal javascript.

Ex: 

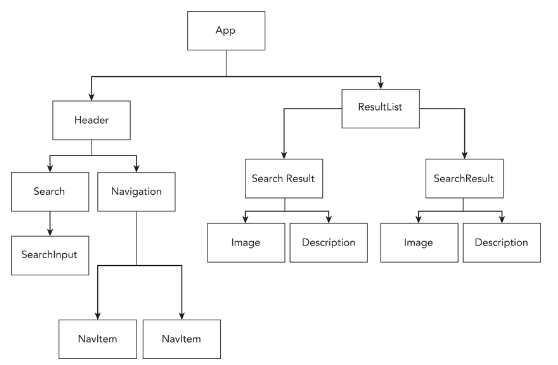
* + Use double curly braces for object literals 
  + The code that does calculations, retrieves data, combines data and controls the flow of the application should be written outside the *return* statement in the functions.
  + Conditional Operator with logical AND &&: The conditional operator says that if expression on left side of && evaluates to true then expression on right side of && will be printed. If either of expression on left side is false then value will be returned as false.



* + Conditional Operator : 
  + React.Fragment component wraps your JSX into single JSX element You can use the React.Fragment component in one of three ways:
  + *By using dot notation: <React.Fragment></React.Fragment>*
  + *By importing Fragment from the react library using curly braces*
  + *By using its short syntax, which is just a nameless element: <> </>*

**Components**:

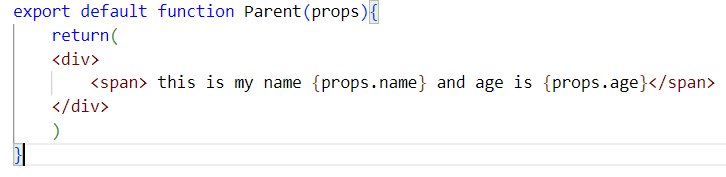
* A component is a function or javascript class that optionally accepts data that describes a piece of user interface. A React user interface is made up of hierarchy of components. The job of the component is to return an element.



* React has built in components for HTML and their attributes, it also has built in components for SVG (Scalable Vector Graphics) elements and attributes. These built-in components produce output for the DOM and serve as a base for your custom components.
* A React HTML component has same name HTML 5. Using them in a component renders the equivalent HTML component.

| **Table 4-1: HTML Elements Supported by React** | |
| --- | --- |
| **HTML ELEMENT** | **DESCRIPTION** |
| a | Creates a hyperlink. |
| abbr | Represents an abbreviation or acronym. |
| address | Indicates that the containing HTML includes contact information. |
| area | Defines a clickable area in an imagemap. |
| article | Represents a self-contained composition (such as a story or an article) in a page. |
| aside | Represents content that is indirectly related to the main content. |
| audio | Embeds sound content. |
| b | Used to draw the reader's attention to the contents. Previously, this was the “bold” element, but it's now called the “Bring to Attention” element to separate its purpose from how it's styled. |
| base | Specifies the base URL for all relative URLs in the document. |
| bdi | Bidirectional Isolate. Isolates text that may flow in a different direction from text around it. |
| bdo | Bidirectional Text Override. Changes the direction of text. |
| big | Renders text at a font size one level larger (obsolete). |
| blockquote | Indicates an extended quotation. |
| body | Represents the content of an HTML document. |
| br | Produces a line break. |
| button | Represents a clickable button. |
| canvas | Creates an area for drawing with the canvas API or WebGL. |
| caption | Specifies a caption for a table. |
| cite | Describes a reference to a cited work. |
| code | Indicates that its content should be styled as computer code. |
| col | Defines a column within a table. |
| colgroup | Defines a group of columns in a table. |
| data | Links content to a machine-readable translation. |
| datalist | Contains option elements indicating the permissible options available for a form control. |
| dd | Provides the definition for a preceding term (specified using dt). |
| del | Represents text that has been deleted from a document. |
| details | Creates a widget in which information is visible when the widget is toggled to its “open” state. |
| dfn | Indicates the term being defined within a sentence. |
| dialog | Represents a dialog box, subwindow, alert box, or other such interactive element. |
| div | A generic container with no effect on content or layout. |
| dl | Represents a description list. |
| dt | Specifies a term in a definition list. Used inside dl. |
| em | Marks text that has emphasis. |
| embed | Embeds external content in the document. |
| fieldset | Groups controls and labels within a form. |
| figcaption | Describes the contents of a parent figure element. |
| figure | Represents self-contained content, optionally with a caption. |
| footer | Represents a footer for its nearest sectioning content. |
| form | Represents a document section containing interactive controls. |
| h1 | First-level section heading. |
| h2 | Second-level section heading. |
| h3 | Third-level section heading. |
| h4 | Fourth-level section heading. |
| h5 | Fifth-level section heading. |
| h6 | Sixth-level section heading. |
| head | Contains machine-readable information about the document. |
| header | Represents introductory content. |
| hr | Represents a thematic break between sections. |
| html | Represents the root of an HTML document. |
| i | Represents idiomatic text that is set off from the normal text. |
| iframe | Represents a nested browser context. |
| img | Embeds an image into the document. |
| input | Creates interactive controls for web-based forms. |
| ins | Represents a range of text that has been added to the document. |
| kbd | Represents a span of text denoting textual user input. |
| keygen | Facilitates generation of key material and submission of the public key in an HTML form. |
| label | Represents a caption for an item in a user interface. |
| legend | Represents a caption for an element in a fieldset. |
| li | Represents an item in a list. |
| link | Specifies a relationship between the document and an external resource. Commonly used to link stylesheets. |
| main | Represents the dominant content of the body of a document. |
| map | Used with area elements to define an imagemap. |
| mark | Represents marked, or highlighted, text. |
| menu | Represents a group of commands. |
| menuitem | Represents a command in a menu. |
| meta | Represents metadata that can't be represented with other metadata elements (such as title, link, script, or style). |
| meter | Represents a fractional value or a scalar value within a known range. |
| nav | Represents a section containing navigation links. |
| noscript | Represents a section to be inserted if a script type is unsupported or if scripting is disabled in the browser. |
| object | Represents an external resource. |
| ol | Represents an ordered list. |
| optgroup | Creates a grouping of options within a select element. |
| option | Defines an item in a select or optgroup. |
| output | Creates a container for the results of a calculation or for user input. |
| p | Represents a paragraph. |
| param | Defines parameters for an object. |
| picture | Contains source elements and an img element to provide alternative versions of an image. |
| pre | Represents preformatted text which should be presented exactly as written. |
| progress | Displays an indicator showing progress towards the completion of a task, such as a progress bar. |
| q | Indicates that its content is a quotation. |
| rp | Used to provide fallback content for browsers that don't support ruby annotations using the ruby element. |
| rt | Specifies the ruby text component of a ruby annotation. |
| ruby | Represents annotations for showing the pronunciation of East Asian characters. |
| s | Represents a strikethrough. |
| samp | Encloses text that represents sample output from a computer program. |
| script | Embeds executable code or data. |
| section | Represents a standalone section in a document. |
| select | Represents a control that shows a menu of options. |
| small | Represents small print, such as copyright or legal text. |
| source | Specifies multiple media resources for picture and audio elements. |
| span | A generic inline container. |
| strong | Indicates that its contents have strong importance. |
| style | Contains style information for a document. |
| sub | Specifies inline text that should be displayed as subscript. |
| summary | Specifies a summary, legend, or caption for details content. |
| sup | Specifies inline text that should be displayed as superscript. |
| table | Represents tabular data. |
| tbody | Encapsulates table rows in a table. |
| td | Defines a cell in a table. |
| textarea | Represents a multi-line text editing control. |
| tfoot | Defines a set of rows summarizing the columns in a table. |
| th | Defines a cell as a header of a group of table cells. |
| thead | Defines a set of rows defining the head of the columns in a table. |
| time | Represents a period of time. |
| title | Defines the title that is shown in the browser's title bar and browser tab. |
| tr | Defines a row of cells in a table. |
| track | Contains timed text tracks (such as subtitles) for audio and video content. |
| u | Originally the underline element, specifies that text should be rendered in a way that indicates that it has non-textual annotation (whatever that means). |
| ul | Represents an unordered list (usually rendered as a bulleted list). |
| var | Represents the name of a variable in mathematic or programming context. |
| video | Embeds a media player that supports video playback. |
| wbr | Represents a word break opportunity, where the browser may optionally break a line. |

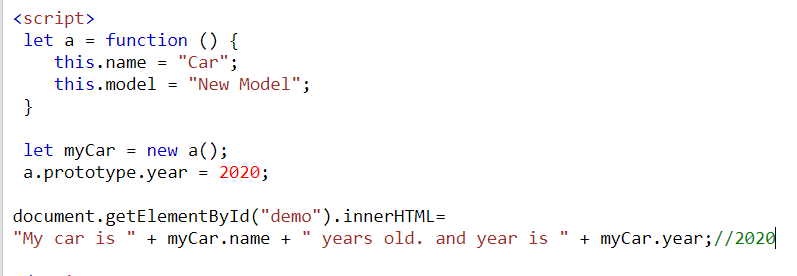
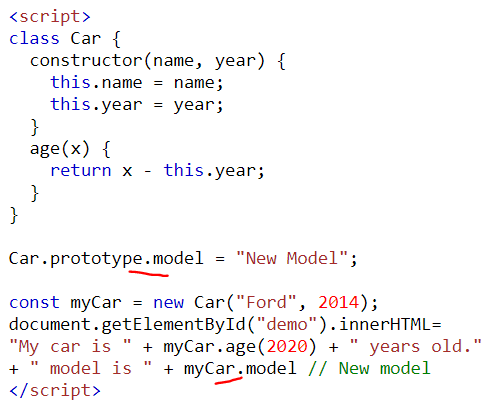
* Attributes vs Props:
  + Attributes that we defined for JSX elements are passed to the component represented by element as ***props***



* + Special attributes that exist in React but not in HTML
    - dangerouslySetInnerHTML 🡪 Not a best practice to update the inner HTML of element
    - suppressContentEditableWarning 🡪 suppresses a warning that React will give you on contentEditable attribute on a element with children
    - suppressHydrationWarning 🡪 This will suppress a warning that React will generate when content generated by Server side react and client side react are different.
  + Some attributes that behave differently :
    - checked & defaultChecked 🡪 checked is used to set/unset a radio button, defaultChecked is used to set a default value for a radio button or checkbox
    - style 🡪 It accepts ONLY javascript object containing style properties and values.
    - selected 🡪 It is used select an option value from the dropdown list.

**JavaScript Classes:**

* In traditional languages like C#, Java classes are blueprints to create objects. In JavaScript classes are themselves objects that serve as a template for objects. JavaScript has *prototypes* not true classes.
* The class syntax in JavaScript is just a new way to use *function constructors* and *prototypal inheritance*.
  + *Prototypal Inheritance* : In JavaScript we can create objects in 3 ways :
    - By using *Object Literal* notation.
    - By using *new* keyword
    - By using *Object.Create* method.

0

* Every object that you create in JavaScript is a copy of another object which is called it’s prototype.
* **React.Component 🡪** base class that every user defined component class can extend. It has number of methods, lifecycle methods, class properties and instance properties that we can make use of.
  + **Importing React.Component (Default Import) 🡪** If we use *import React from ‘react’*  then it imports entire React library and we will need to extend the class like this :
    - * Class MyComponent extends **React.Component** {

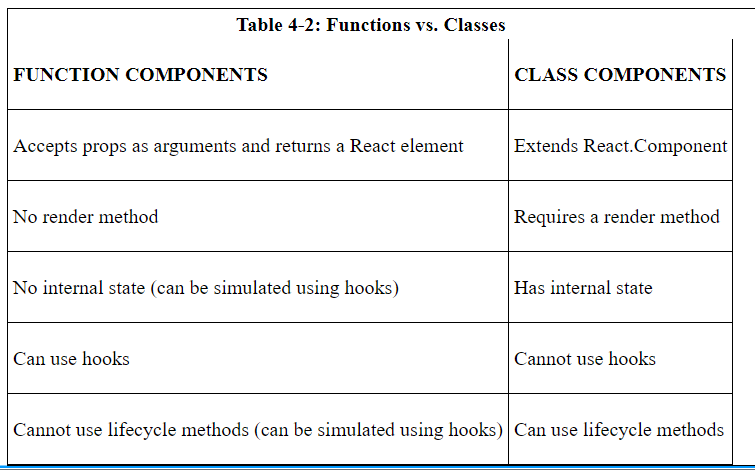
}

* **Named Import :**  *import {Component} from ‘react’*  this means to import ‘Component’ class specifically
  + - Class MyComponent extends Component {   
      }
* **Constructor Function:** This function will run exactly once when object for the class gets created. Here we will bind event handler functions to the instance of the class and set up the local state for the instance. A screen shot of a computer code

  Description automatically generated
* Every object of the component maintains it’s own local state. This state is initialized in the constructor. The state of a component is stored in an object called *‘state’* and every time the state changes React re-renders the UI. The other object in a component instance that stores the data is

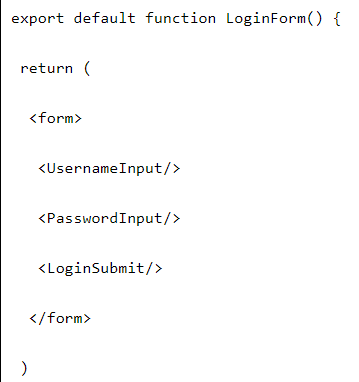
‘*props*’ 🡪 This is the data that is passed to a component from a Parent Component in component hierarchy. If we are going to use ‘props’ object in the constructor then we will need to pass this ‘props’ to superclass’s constructor by using super(props).

* **Function Components:**
  + These are javascript functions that return React Elements. They don’t have constructor, no need ‘bind’ other events and no render method.
  + As they are just javascript functions the variables declared can be initialized but functions can’t maintain state.
  + Since React 17, React has introduced **hooks** that allow function components to create and access data from one invocation of the function to next.
  + With **hooks** a function component can behave like a class component by maintaining state.
  + React has many built-in hooks and even lets us write our own hooks and the hooks that let us persist data in function components is ***useState***.

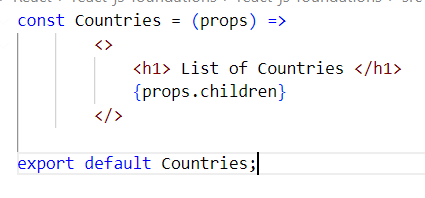


**React Component Children:**

* Components that are rendered inside other components are called *children.* Components that contain other components are called *parents.* A Child component can have children as well. Here <form> is a child of *LoginForm* and it has 3 grandchildren.



* **this.props.children 🡪** Every component in React UI has a property called *children* that stores the children of the component. By using this.props.children (or props.children in case of function component) in the return statement of the component , you can create a placeholder where child components will be placed.

A screenshot of a computer

Description automatically generated

**Component Life Cycle:** During React application, components gets created, do their job and get destroyed. At each state there are certain events that get fired and methods that invoked. These methods and events are called *component lifecycle*. The stages of component’s life are:

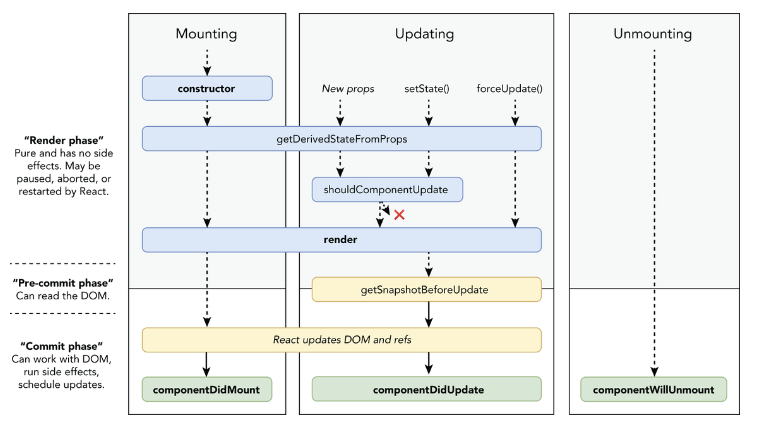
**Mounting**: Components get constructed with props passed into it and with default state with JSX being rendered.

**Updating**: If state gets updated then component is re-rendered.

**Unmounting**: Component gets removed from the react application.

**Error Handling**: These methods run when an error happens during component lifecycle.

* Initial Render 🡪 React creates a Virtual DOM tree
* State/Props Change 🡪 Triggers re-rendering of the component, producing a new Virtual DOM tree.
* Diffing 🡪 React compares the new virtual DOM to previous one by using optimized algorithm
* React efficiently updates the real DOM



**Reconciliation 🡪** React determines minimum number of changes needed to update the actual DOM to match Virtual DOM.

**Mounting:**

* constructor() 🡪 In class components we can set the state of component and bind event handlers.
* getDerivedStateFromProps 🡪 Checks If props for the component has changed and updates the new props to update the state.
* render() 🡪 After mounting, *render* runs every time the component updates and this runs in both Mounting and Updating stages. In this stage JSX will be rendered to the DOM. When a component renders React creates a Virtual DOM tree based on component’s *render* method.
* componentDidMount() 🡪 This method runs when a component has finished mounting and has been inserted into the DOM. Here it is safe to write code that depends on DOM nodes or to fetch remote data.

**Updating**:

* shouldComponentUpdate() 🡪 The default behavior of a React Component is it gets updated every time when state changes. There may be times when we want to tell React to not update the component even if the state got changed.

This method returns either true/false. If we have component that we know that it will never get updated once it’s mounted we can prevent it by using this code:

shouldComponentUpdate() { return false; }

More often the way this method is used to compare the previous props to new props and if there is a change in them only then this method returns true.

A computer code with black text

Description automatically generated

* getSnapShotBeforeUpdate() 🡪 The purpose of this is to get the relevant information about the browser or any device before making the component active in the DOM. This method happens right before the rendered output from the component is made active in the DOM. One example when one example use for it is to maintain the scroll position of an element (such as a text box) between renders. If an update to the browser DOM would affect what the user is currently viewing in the browser, getSnapshotBeforeUpdate can be used to find out the relevant information about the browser DOM so that it can be restored after the update happens.
* componentDidUpdate() 🡪 This runs immediately after the component updates. It is used to perform network requests based on new props passed to the component or for performing operations that depend on the snapshot of the DOM after getSnapShotBeforeUpdate () method.

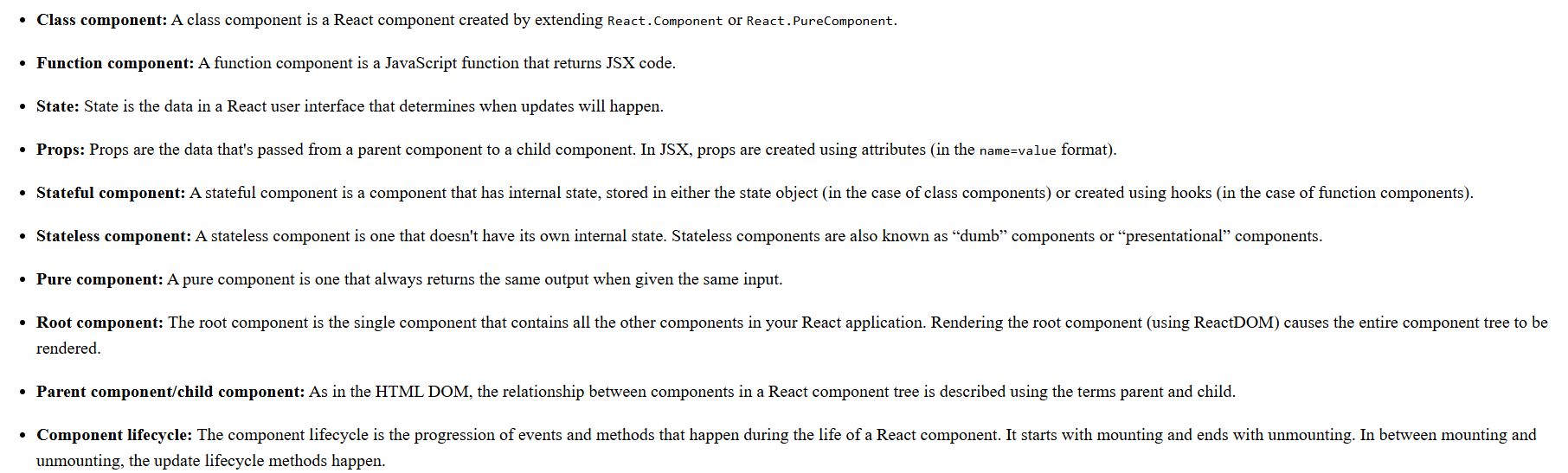
If shouldComponentUpdate() returns false then this method will not run.

**Unmounting**

* componentWillUnmount() 🡪 This method is called right before the component is removed from the DOM. If we need to any clean up related to the component here in this method we can do it like stopping any network requests that are in progress, stopping timers and removing event listeners created in *componentDidMount.* Unmounting is a phase where clean up operations should happen so as to avoid potential memory leaks. This phase happens when a component is no longer needed such as when it is conditionally rendered and the condition changes when user navigates away from the component’s view.

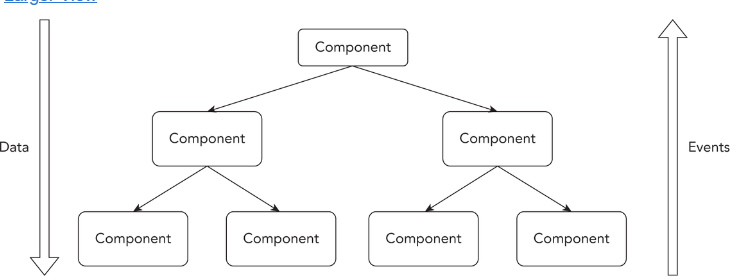
**Error Handling:**

* getDerivedStateFromError() 🡪 If any of the descendant components throw an error then the parent component will run this method as this method receives the error occurred and returns an object that will update the state.
* componentDidCatch() 🡪 This runs after a descendant component throws an error. It can useful in error logging as it doesn’t run in render phase.



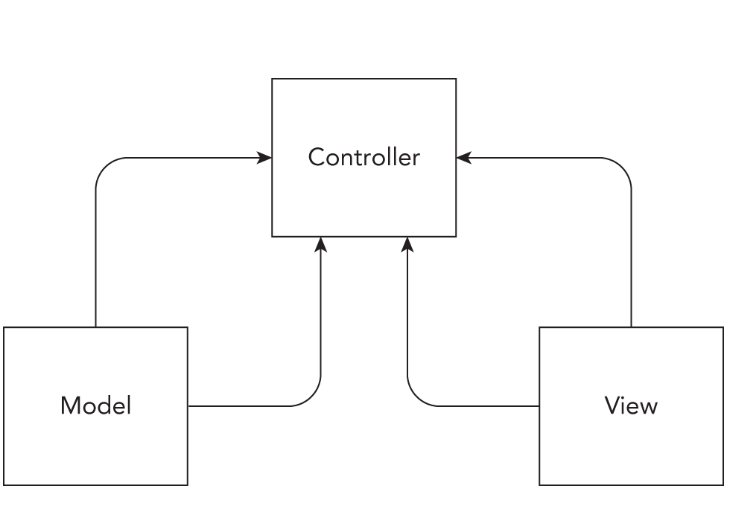
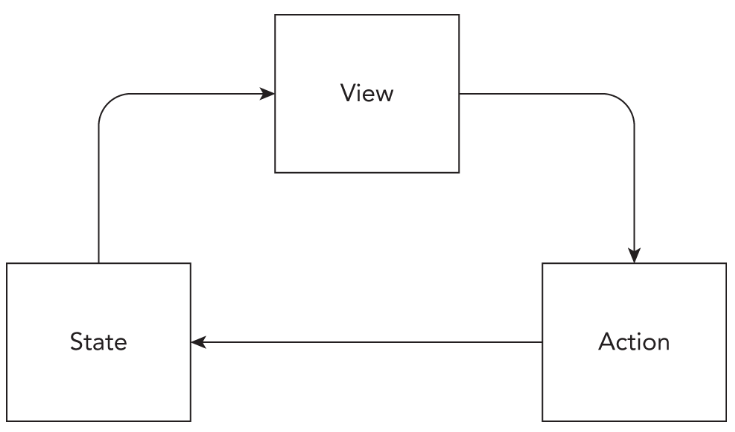
**One-Way Data Flow:**

* Also called as *UniDirectional Data Flow* means that all the data in react application flows from parent component to child component. “Data flows from parent to child component (downstream) and events flow up (upstream)”.



* One way data flow means that the way we send data from child to parent component or in between sibling components is different than the way we send from parent to child component.
* ***Why One-Way data flow* ?** Two way data flow or bidirectional data flow where data flows from parent component to child component and changes within the child component can affect data in parent component is convenient but this may be prone to errors, also increases complexity of a user interface.

**Two way data binding One way data binding**

 ****

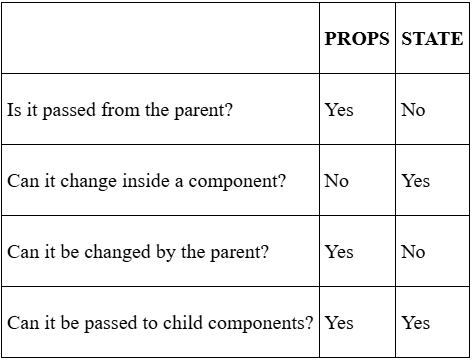
**Props:** These are the primary way where parent and child components share data. To create a prop, we will simply give an attribute to a custom element and this attribute will become property to *props* object for the custom element.

A component can have any number of props, a props value can be any javascript data type value or function. Props are read only meaning once a component receives these values it can’t change/overwrite them inside the component.

**State:** In React, a state is an object containing set of properties that can change over the lifetime of a component. The component behavior changes once state changes.

* Initialization of State :
  + Class Component : In a class component it is assigned inside a constructor as this is the place which gets called when an instance to the class gets created.
  + Function Component : By default, functions don’t store a state. By using hooks we can replicate the behavior of making functions store a state. The hook that makes possible for function components to maintain state is *useState*; The first time the function component containing *useState* is rendered then it creates a state variable and a function to set the variable. For all subsequent requests *useState* makes use of first variable created in the first render. The first time the component renders *useState* serves the same purpose of a constructor function for class component.

**Differences between State and Props:**

* These are both javascript objects and changes to either of them updates the components. The real difference is that state is maintained for the component and it is local to it where as props are being passed to the component by it’s parent component. 
* Calls to setState are always asynchronous. When you call setState, it may not immediately update the state.It actually just schedules, or enqueues, an update to the component's state. The reason for this behavior is that it reduces the number of unnecessary component re-renders. It's helpful to think of a call to setState as a request, rather than an immediate operation.
* **Passing function into setState() 🡪** When we pass a function that returns an object into a setState, the inner function receives current state and props of the component and returns the state object. This function is called *updater function*.

**Updating *state* with Function Components:**

* *useState* hook will be used to update the state in function components.

const [counter , setCounter] = useState(0);

The above line means that counter is a variable and setCounter is a function that updates(replaces the state) of this variable. The setter function replaces the counter value instead of merging it into current state. Whenever the state updates, React re-renders the function component and useState returns a new variable with the latest state value. So the setter function doesn’t actually modify the variable in the function at all and the function gets new const every time it is invoked.

If your new state depends on the old state, you'll need to make a copy of the existing array or object, modify it, and then pass the copy of the array into the setter function.

The copy you make of an object or array can't be just any copy. It needs to be a *shallow* copy. One of the easiest ways to make a *shallow* copy, which is widely used in React, is by using the spread operator (…).

To see how the spread operator works, we'll start with a very simple example. The following function accepts three numbers and returns the sum of the numbers:

   function sum(x,y,z){

    return x+y+z;

   }

If you have an array of three numbers that you want to find out the sum of, you could invoke the sum function and pass in each element of the array separately, like this:

   sum(myNumbers[0],myNumbers[1],myNumbers[2]);

Or you could just spread the array into its component parts, which accomplishes the same thing:

   sum(…myNumbers)

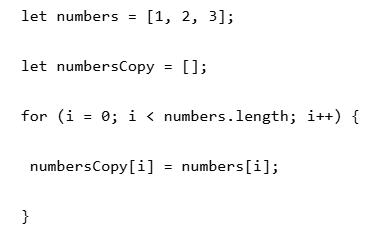
The spread operator is useful in cases where you want to include all of the elements of an array or object in a new object or array, such as when you're creating a new array or object that's partially made up of an existing one.

**Shallow Copy**:

In JavaScript, arrays are always *reference values*.



So to make sure that we don’t need to reference to original array, we need to create a shallow copy. We can do in below ways :

* Use loop to create another array 
* By using slice function,
* By using spread operator(…) 🡪 numbersCopy = […numbers];

**Lifting State Up**:

* Having lot of components that maintain their own state can very quickly increase the complexity of the app. A good rule of thumb is majority of our components should be stales pure functions. A pure function is one in which the output of the function is solely result of it’s input.
* To turn stateful components to stateless components in React we do something called as “***Lifting State Up***” 🡪

Instead of each component maintaining it’s own state we can have a component at higher level in the hierarchy of our user interface to control the state. This state can be passed down as props to the components that need it.

* For example in a search form , the words typed in a search are used by other components which use these words so hence the *state* variable for this text field should be lifted up to a common ancestor that other components inside the form can use.
* **PropTypes** and **defaultProps** 🡪 We use propTypes for validating the props and defaultProps to set initial values for the props. ***propTypes.shape*** validates the object the component receives and it checks correct data types and correct properties.
* **Key :** Any time we have a list of components each element in the list must have a prop named *key*.  The value of *key* must be unique to each item in the list. Since the *index* position of an element in an array is a unique value, this makes a convenient value for the *key* prop. The value of *key* is not available as part of the props object inside the component.

**SyntheticBaseEvent:** It’s a wrapper around native DOM events.

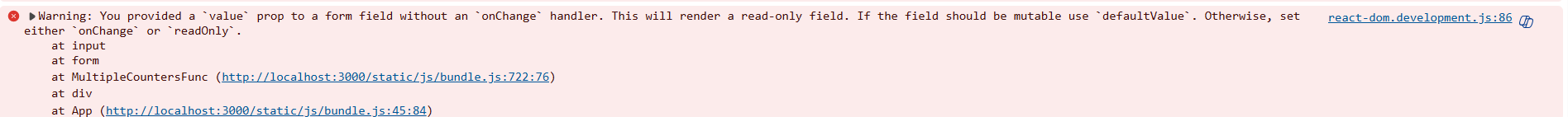
**Controlled and Uncontrolled Inputs:** The default behavior of any input element is to allow the user to change it’s value directly. If a user changes the value of any form input control then React calls it *uncontrolled input* where as if any form input element changes the value based on the change in a state then it’s called as *controlled input.*



When we provide a ‘value’ to a text box it becomes a *controlled input* and it can’t be edited by the user unless we add event listener attribute and event handler function to it.

If we don’t add any event listener React writes an error saying that ‘value’ attribute has been added without any event listener to it.





When we add *onChange* event listener then we can detect this Event object and e.target.value property to update the ‘state’.

Using controlled inputs ensures that your user interface strictly adheres to one-way binding.

To get values from *uncontrolled inputs* we can use a technique called a *ref*. The *ref* creates a reference to the underlying DOM node , which allows React to access it’s properties directly.

**Refs:**

* If we want to access a DOM node or access it’s property directly then we can use *refs.* A *ref* is a reference to a child component that allows us to modify the child component or DOM node from parent component rather than using a traditional way of using props.
* In a class component we use React.createRef() to create a ref. In function components we use useRef(null).
* If we assign a *ref* to a child component then we can access the methods or properties of the child component by using a property of the *ref* called as *current*.
* When we create a *ref* to access a DOM node then *current* can access all the properties of the DOM node. When we create a *ref* to access a custom React element then *current* receives the mounted instance of the component.
* **forwardRef** is a utility function in React that allows you to forward a ref from a parent component to a child component. It is particularly useful when you need to pass a ref to a child component that is not a DOM element, such as a custom component.

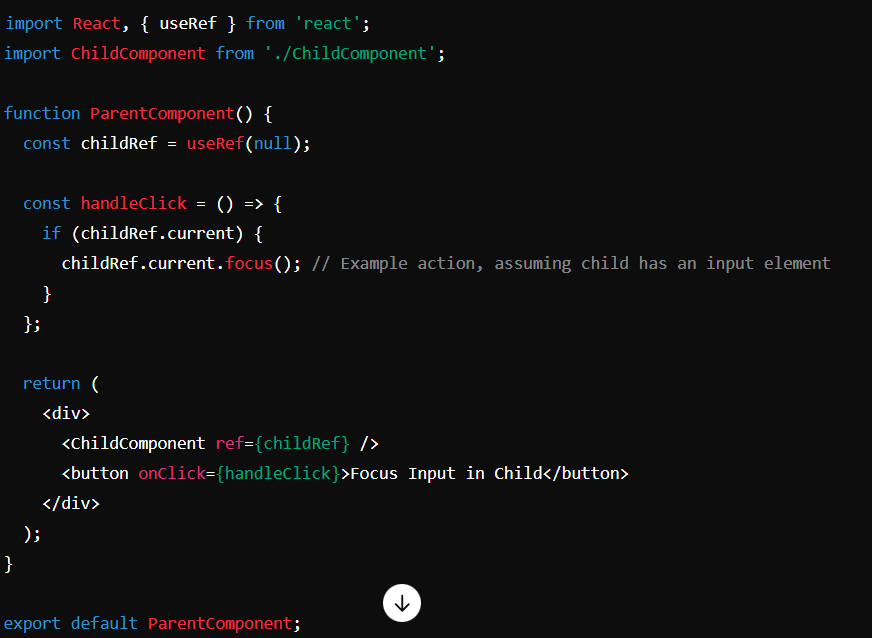
**Key Points about forwardRef:**

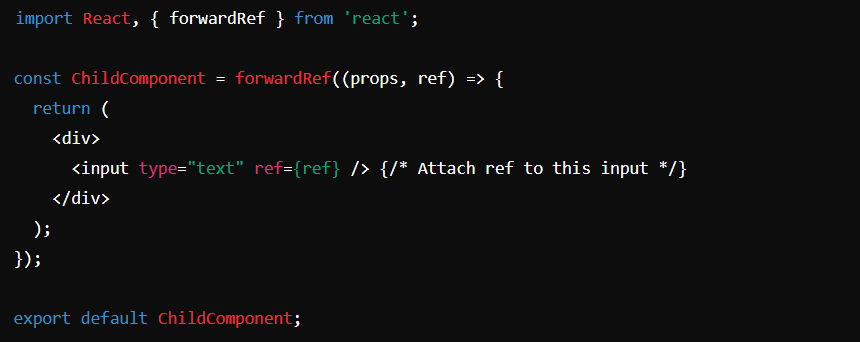
1. **Purpose**: forwardRef is used to pass a ref from a parent component to a child component. It enables the parent to access the child's instance or a DOM element inside the child component.
2. **Syntax**: forwardRef takes a function component as an argument. This function receives two arguments:
   * props: The props that are passed to the child component.
   * ref: The ref that is forwarded from the parent component.
3. **Usage**: It is commonly used when you have functional components and need to expose a ref to a parent component. For class components, direct refs are usually sufficient, but forwardRef is essential for functional components.

**Example of Using forwardRef**

**1. Parent Component**

The parent component creates a ref and passes it to the child component.





**Explanation:**

* **forwardRef Function**:
  + forwardRef wraps the ChildComponent functional component. It allows ChildComponent to receive a ref from its parent.
  + Inside the ChildComponent, the ref is attached to an input element, allowing the parent to directly interact with that input element.
* **Ref Forwarding**:
  + The parent component creates a ref (childRef) and passes it to ChildComponent.
  + When the button in the parent component is clicked, it triggers handleClick, which accesses the input element within the child component and calls its focus method.

**Use Cases for forwardRef:**

1. **Accessing DOM Elements**: When a functional component needs to expose a DOM element to the parent component.
2. **Integrating with Third-Party Libraries**: When integrating with libraries that require direct access to a DOM node.
3. **Custom Hooks**: When custom hooks need to manage refs and you want to expose those refs to parent components.

In summary, forwardRef is a powerful utility in React that helps with ref management in functional components, making it easier to forward refs and interact with DOM elements or component instances within child components.

**CallBack Ref:**

* A callback ref doesn’t use *createRef* or *useRef* hook. Instead it passes a function to the ref attribute which receives the react component instance or DOM node element as it’s argument.

When to use Refs :

* Managing focus
* Automatically selecting text within a child element
* Controlling media playback
* Setting scroll position on a child element
* Triggering imperative animations
* Integrating with third-party libraries (such as jQuery, for example)

**Hooks:**

These are part of React Library that give function components access to all the features of React that were only possible with class components. These include state, refs, lifecycle events , caching of function results.

Hooks are only for function components and they must be defined at the top level of the function instead of defining them in an inner function or inside any statement etc.,

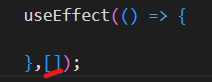
**Built-in Hooks**

➤ useState 🡪 Creates a stateful value from the argument passed to it along with function for updating it.

➤ useEffect 🡪 It accepts a function that will run every time after the function gets rendered. Any code that we think can cause side effect can be run in this hook. The side effects can happen when we make API requests, DOM manipulations, setting timers. Hence side effects are handled inside of lifecycle methods like constructor(), componentDidMount() and componentDidUpdate() for class components. The use of function inside **useEffect()** replicates as if we have passed the function inside of componentDidMount() or componentDidUpdate() events. The timing of when useEffect() runs and when these life cycle methods are bit different though. Most of the cases it will not cause any problem but in some cases it can cause problem.

A function component renders every time when state changes so stop this behaviour we can use ‘useEffect’ in below way :

The empty array indicates to render the function when one of the values in empty array has changed. Since no value exists in

 the array the function doesn’t get rendered or refreshed every time (i.e.. the function code in the useEffect). Specifying the dependencies often increases performance by eliminating unnecessary renders.

As useEffect is asynchronous and runs after the component has rendered it is better to have asynchronous tasks such as fetching data.

➤ useContext 🡪 When we want to send huge context data from parent component to child component then instead of passing them through props as they will become quite complex we can use useContext to send global data that can be used by a huge tree of components.

➤ useReducer 🡪 It’s an alternative to *useState* to store state in function components usually for complex state updates or situations where new state depends on previous state. It takes a *reducer* and initial state as it’s arguments. A *reducer* is a pure function that takes current state and an object called *action* and returns the new state.

*useReducer* returns a value (new state) and a dispatch function. A dispatch function can be used in response to events but instead of taking a value to set the stateful variable , it takes an action object.

➤ useCallback

➤ useMemo

➤ useRef

➤ useImperativeHandle

➤ useLayoutEffect

➤ useDebugValue