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

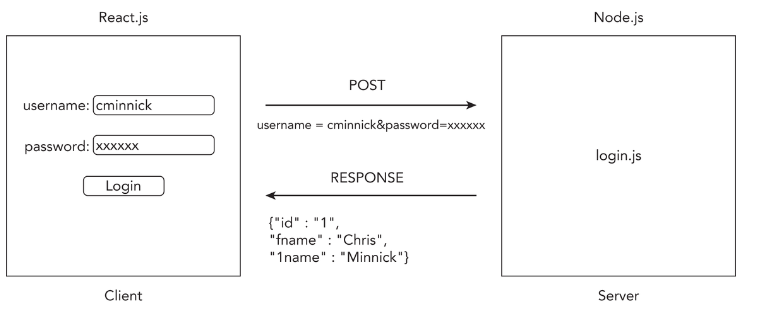
npx create-react-app my-firstapp OR 🡪 Creates a react app by name ‘my-FirstApp’

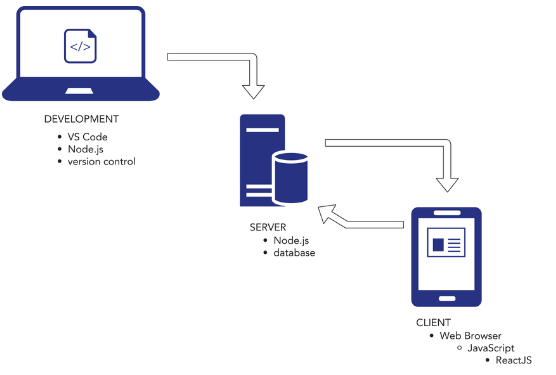
Open package.config and press Ctrl + S to resolve any build errors if you get know

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

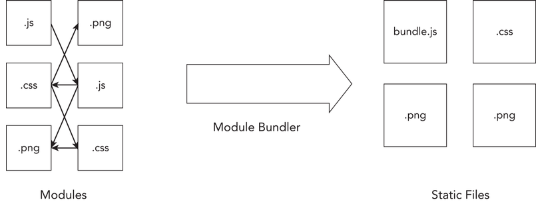
**Node JS**

* 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 :

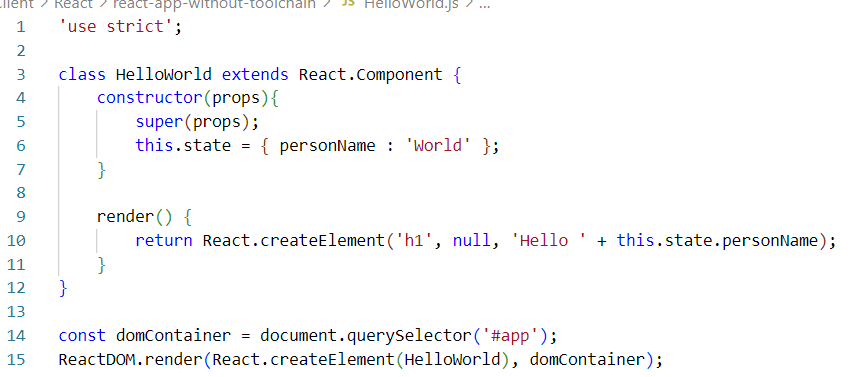
* It must be less than 214 characters long.
* The name can't start with a dot or underscore.
* The name can't have uppercase letters.
* It can't contain any characters that aren't allowed in URLs (such as ampersands and dollar signs) and that are “unsafe” in URLs (such as the percent symbol and spaces).

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:

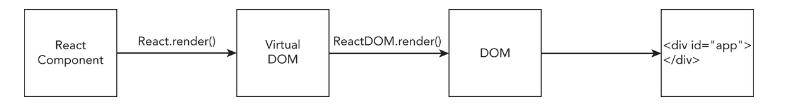
* Keep it simple and as short as possible.
* Use only lowercase letters.
* Use dashes in place of spaces.
* Don't use the same name as a common Node.js package.

**React CDN:**

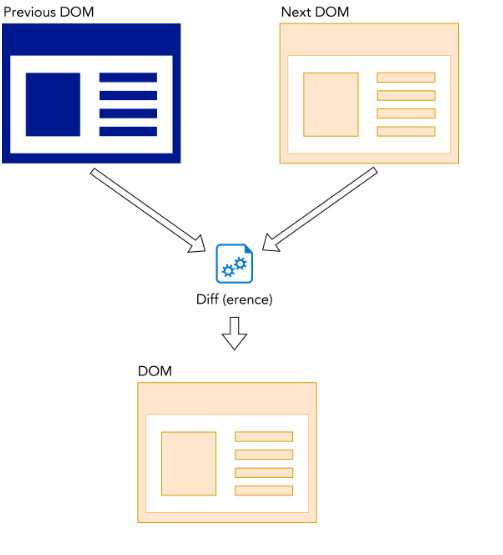
* React application can be created without any toolchain of modules.
* Simply copy the CDN links from here 🡪  <https://reactjs.org/docs/cdn-links.html>
* UMD stands for Universal Module Definition is what allows the CDN version of React to work in browsers without compile step.



* 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.

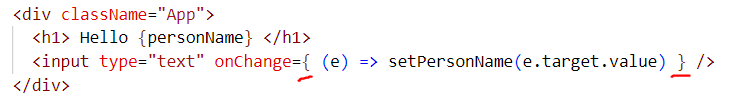
function calculateShippingCharge(weight,shippingMethod){ import calculateShippingCharge from ./calculateShippingCharge.js;

  // do something here

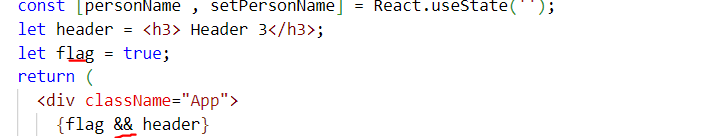
   }

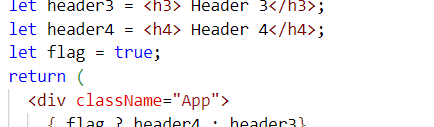
   export default calculateShippingCharge;

* 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: 

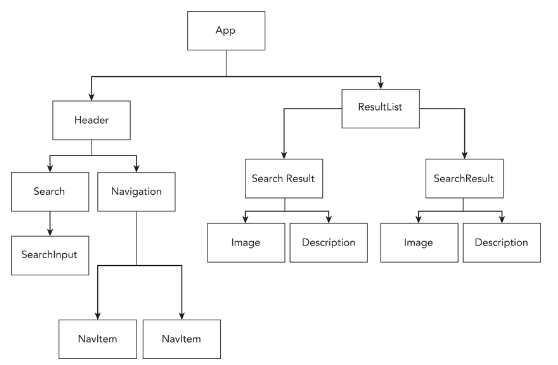
* + Use double curly braces for object literals 
  + The code that does calculations, retrieves data, combines data and controls the flow of the application should be written outside the *return* statement in the functions.
  + Conditional Operator with logical AND &&: The conditional operator says that 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**:

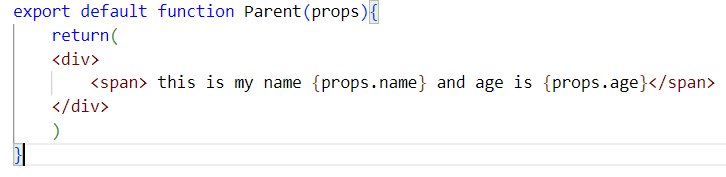
* A component is a function or javascript class that optionally accepts data that describes a piece of user interface. A React user interface is made up of hierarchy of components. The job of the component is to return an element.



* 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. |

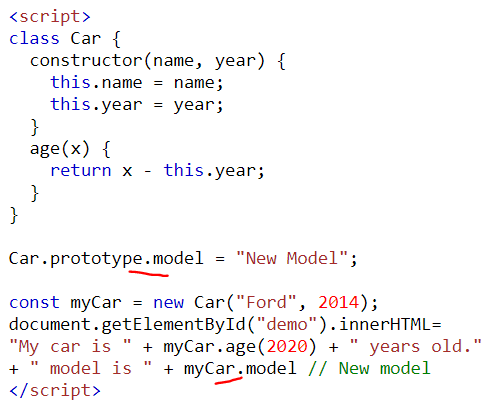
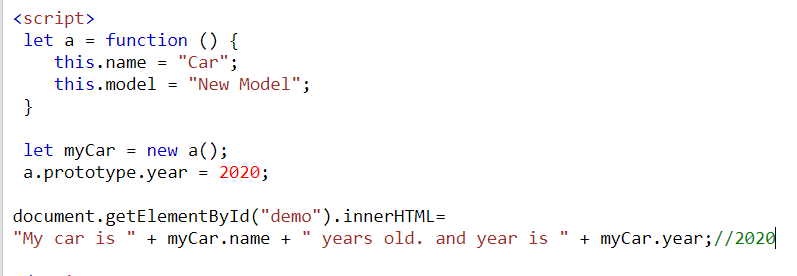
* Attributes vs Props:
  + Attributes that we defined for JSX elements are passed to the component represented by element as ***props***



* + 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.



* 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** {

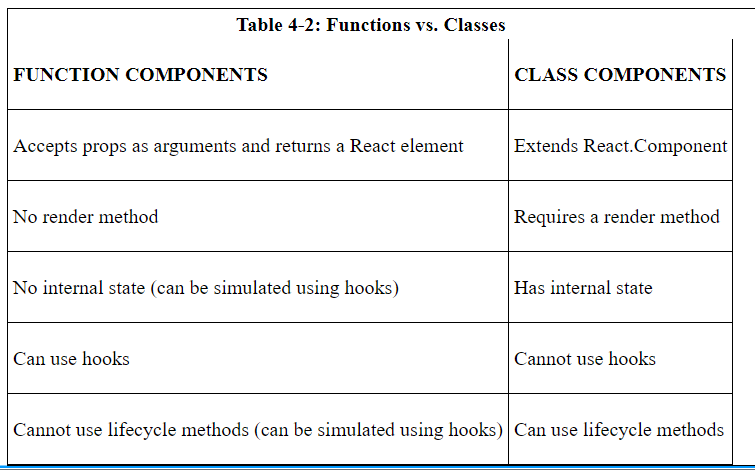
}

* **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

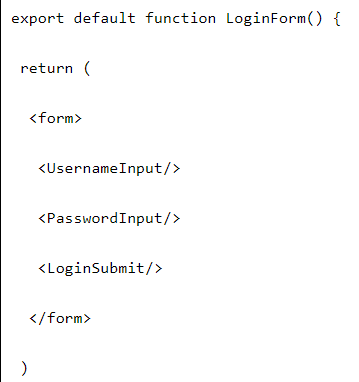
‘*props*’ 🡪 This is the data that is passed to a component from a Parent Component in component hierarchy. If we are going to use ‘props’ object in the constructor then we will need to pass this ‘props’ to superclass’s constructor by using super(props).

* **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***.

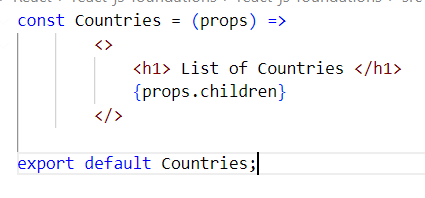


**React Component Children:**

* 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:

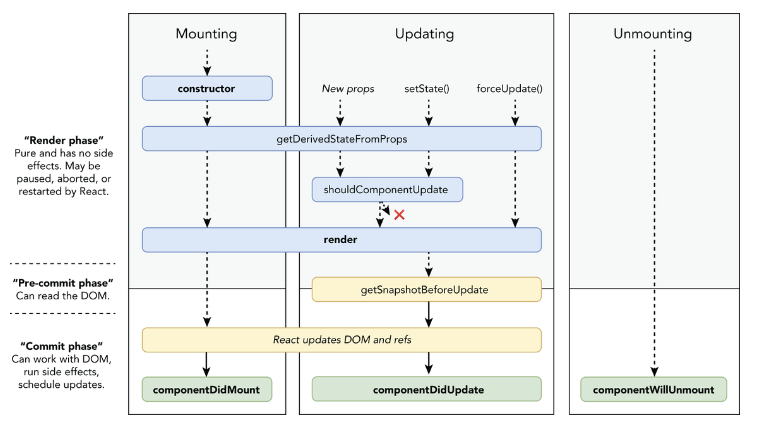
**Mounting**: Components get constructed with props passed into it and with default state with JSX being rendered.

**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 updated 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. Once 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.