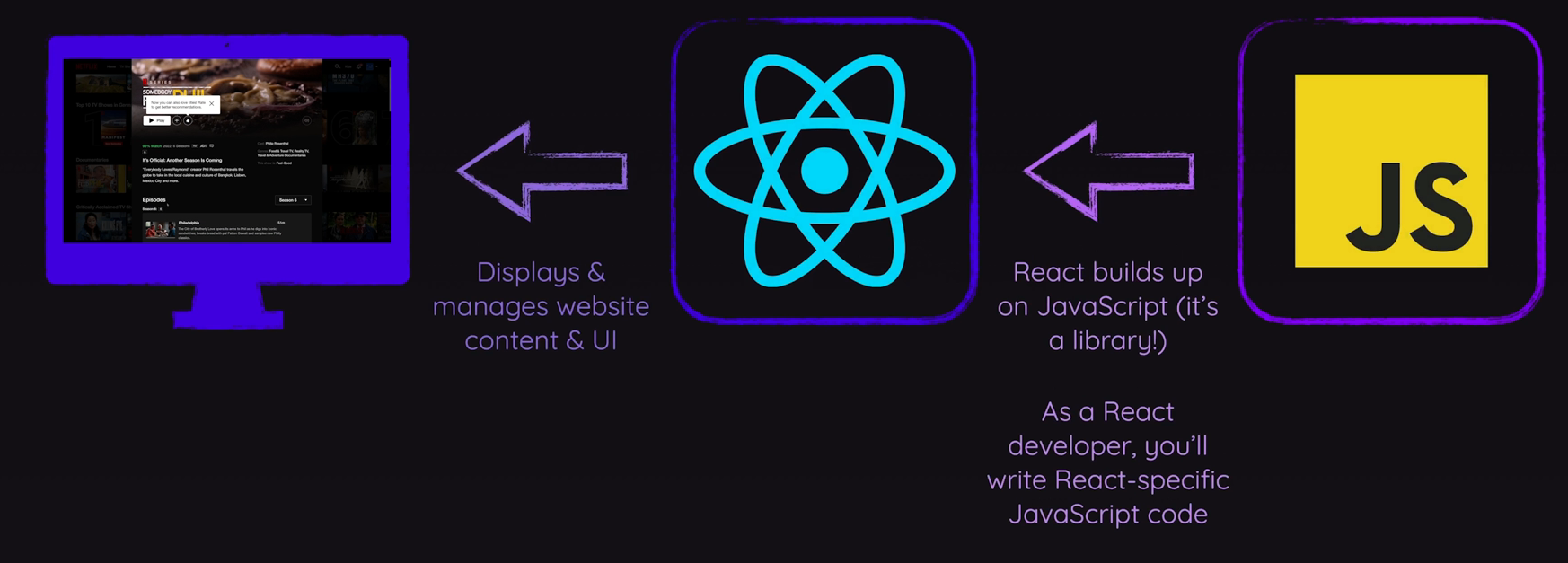
Npm install

Npm start

# React

React is the library for web and native user interfaces. React is a library for building user interfaces. A JavaScript library for building user interfaces



JavaScript controls the UI & content of this simple demo page.



Clicking different tabs triggers event listeners and kicks of a series of function executions that display different content & highlight the selected tab.

Using "Just JavaScript" Typically Isn't a Great Option

* Writing complex JavaScript code quickly becomes cumbersome
* Complex JavaScript code quickly becomes error-prone
* Complex JavaScript code often is hard to maintain or edit
* React offers a simpler mental model

# Demo App -- Javascript



## Index.html

<!DOCTYPE html>

<html>

  <head>

    <title>Vanilla JavaScript</title>

    <meta charset="UTF-8" />

    <link rel="stylesheet" href="styles.css" />

    <script src="index.js" defer></script>

  </head>

  <body>

    <header>

      <img src="js-logo-xs.png" alt="JavaScript logo" />

      <div>

        <h1>Vanilla JavaScript</h1>

        <p>i.e., JavaScript without any (UI) framework or library</p>

      </div>

    </header>

    <div id="tabs">

      <menu>

        <button id="btn-why-react" class="active">Why React?</button>

        <button id="btn-core-features">Core Features</button>

        <button id="btn-resources">Related Resources</button>

      </menu>

      <div id="tab-content"></div>

    </div>

  </body>

</html>

## Index.js

const content = [

    [

      "React is extremely popular",

      "It makes building complex, interactive UIs a breeze",

      "It's powerful & flexible",

      "It has a very active and versatile ecosystem"

    ],

    [

      "Components, JSX & Props",

      "State",

      "Hooks (e.g., useEffect())",

      "Dynamic rendering"

    ],

    [

      "Official web page (react.dev)",

      "Next.js (Fullstack framework)",

      "React Native (build native mobile apps with React)"

    ]

  ];

  const btnWhyReact = document.getElementById("btn-why-react");

  const btnCoreFeature = document.getElementById("btn-core-features");

  const btnResources = document.getElementById("btn-resources");

  const tabContent = document.getElementById("tab-content");

  function displayContent(items) {

    let listContent = "";

    for (const item of items) {

      listContent += `<li>${item}</li>`;

    }

    const list = document.createElement("ul");

    tabContent.innerHTML = ""; // clear existing content

    list.innerHTML = listContent; // insert new content

    tabContent.append(list);

  }

  function highlightButton(btn) {

    // Clear all existing styling / highlights

    btnWhyReact.className = "";

    btnCoreFeature.className = "";

    btnResources.className = "";

    btn.className = "active"; // set new style / highlight

  }

  function handleClick(event) {

    const btnId = event.target.id;

    highlightButton(event.target);

    if (btnId === "btn-why-react") {

      displayContent(content[0]);

    } else if (btnId === "btn-core-features") {

      displayContent(content[1]);

    } else {

      displayContent(content[2]);

    }

  }

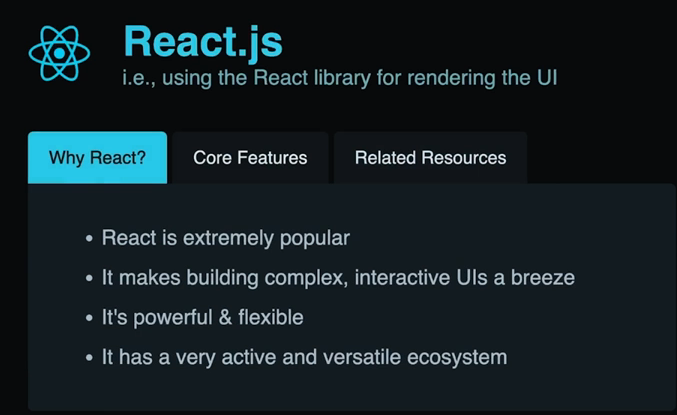
  displayContent(content[0]); // initially show this content

  btnWhyReact.addEventListener("click", handleClick);

  btnCoreFeature.addEventListener("click", handleClick);

  btnResources.addEventListener("click", handleClick);

# Demo App – React



## Index.html

<!DOCTYPE html>

<html lang="en">

<head>

  <meta charset="utf-8">

  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

  <meta name="theme-color" content="#000000">

  <link rel="manifest" href="%PUBLIC\_URL%/manifest.json">

  <link rel="shortcut icon" href="%PUBLIC\_URL%/favicon.ico">

  <title>React App</title>

</head>

<body>

  <noscript>

    You need to enable JavaScript to run this app.

  </noscript>

  <div id="root"></div>

</body>

</html>

## Index.js

import { StrictMode } from "react";

import { createRoot } from "react-dom/client";

import App from "./App";

const rootElement = document.getElementById("root");

const root = createRoot(rootElement);

root.render(

  <StrictMode>

    <App />

  </StrictMode>

);

## App.js

import { useState } from "react";

import "./styles.css";

const content = [

  [

    "React is extremely popular",

    "It makes building complex, interactive UIs a breeze",

    "It's powerful & flexible",

    "It has a very active and versatile ecosystem"

  ],

  [

    "Components, JSX & Props",

    "State",

    "Hooks (e.g., useEffect())",

    "Dynamic rendering"

  ],

  [

    "Official web page (react.dev)",

    "Next.js (Fullstack framework)",

    "React Native (build native mobile apps with React)"

  ],

  [

    "Vanilla JavaScript requires imperative programming",

    "Imperative Programming: You define all the steps needed to achieve a result",

    "React on the other hand embraces declarative programming",

    "With React, you define the goal and React figures out how to get there"

  ]

];

export default function App() {

  const [activeContentIndex, setActiveContentIndex] = useState(0);

  return (

    <div>

      <header>

        <img src="react-logo-xs.png" alt="React logo" />

        <div>

          <h1>React.js</h1>

          <p>i.e., using the React library for rendering the UI</p>

        </div>

      </header>

      <div id="tabs">

        <menu>

          <button

            className={activeContentIndex === 0 ? "active" : ""}

            onClick={() => setActiveContentIndex(0)}

          >

            Why React?

          </button>

          <button

            className={activeContentIndex === 1 ? "active" : ""}

            onClick={() => setActiveContentIndex(1)}

          >

            Core Features

          </button>

          <button

            className={activeContentIndex === 2 ? "active" : ""}

            onClick={() => setActiveContentIndex(2)}

          >

            Related Resources

          </button>

          <button

            className={activeContentIndex === 3 ? "active" : ""}

            onClick={() => setActiveContentIndex(3)}

          >

            React vs Js

          </button>

        </menu>

        <div id="tab-content">

          <ul>

            {content[activeContentIndex].map((item) => (

              <li key={item}>{item}</li>

            ))}

          </ul>

        </div>

      </div>

    </div>

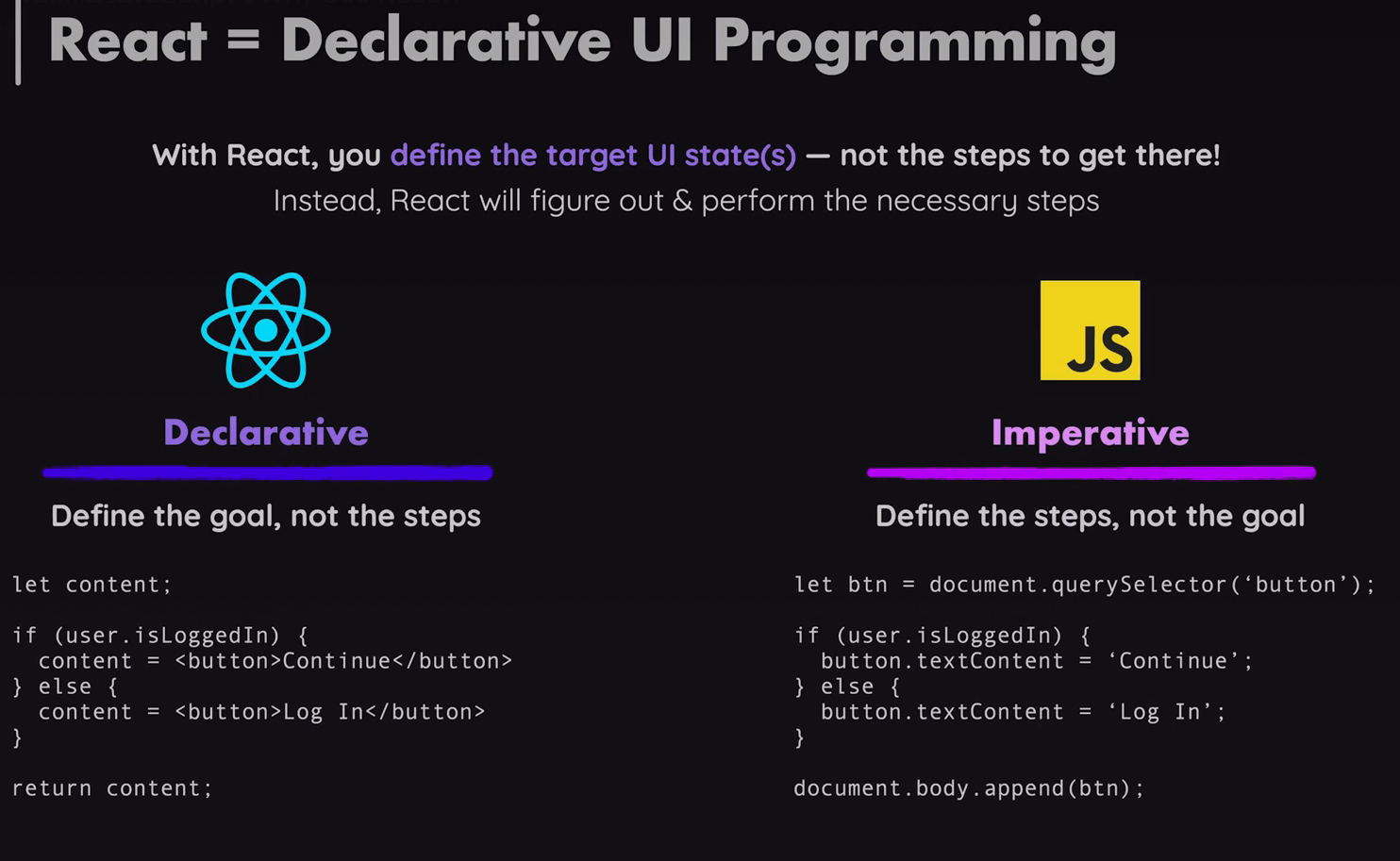
  );

}

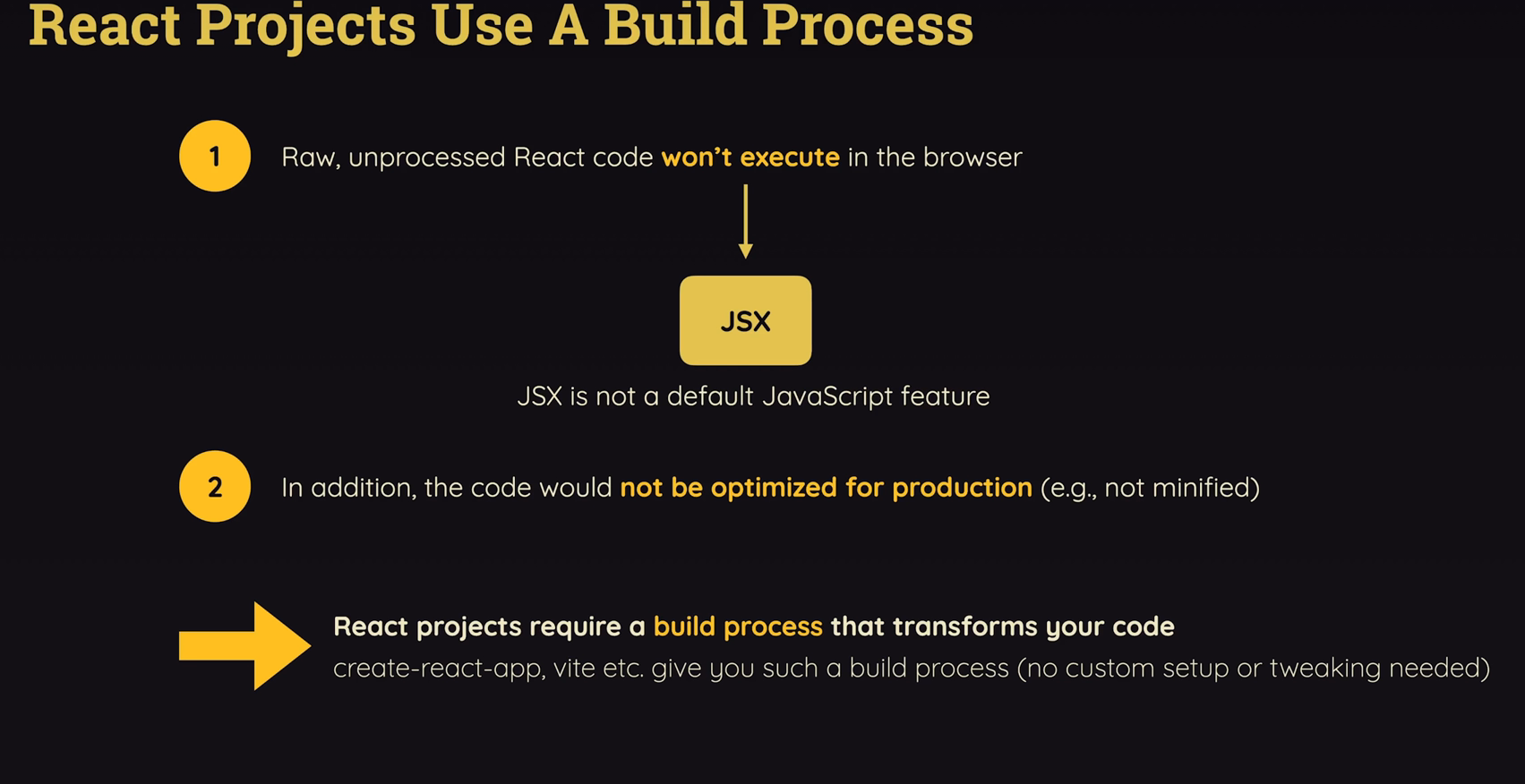
React = Declarative UI Programming

With React, you define the target Ul state(s) not the steps to get there!

Instead, React will figure out & perform the necessary steps



React projects use a build process. The code you write is not the code that gets executed (like this) in the browser. Your code is transformed before it's handed off to the browser.



The first reason is that raw unprocessed React code won't execute in the browser. Mostly because React code uses this special JSX feature.

JSX code simply is this HTML code written in JavaScript files. Out of the box, this would not work because this is not a standard JavaScript feature. React will be all about including this HTML-like code in your JavaScript files, to enable this feature, the code must be transformed so that you can use it whilst writing the code but it's transformed to something JavaScript knows before the code gets executed in the browser.

Another reason is that the code you write would not be optimized for production, it would not be minified, for example. Minification simply means that, of variables or functions are shortened to reduce the amount of JavaScript code that's served to the user.

If we take a look at the code that is served to the user here by, for example, visiting one of these imported code files i.e. from page source that is served here is pretty unreadable, not the kind of code we would like to work on as a developer, but valid JavaScript code in the end. Though it is code that's highly optimized to be as short and small as possible to reduce the amount of code that has to be downloaded by the website visitor. So that's another reason why React projects typically require a built process.

That's also the reason why you also need to install NodeJS on your system in order to work on React projects. Because NodeJS is not just used to install packages with the NPM command or to create projects with the NPX command as you already learned it, but instead it's also used behind the scenes by these tools that are used by that build process that's running behind the scenes.

So NodeJS is also needed and used behind the scenes to make sure that your React code gets transformed.

# JavaScript refreshers

## Export and Import

<script src="Assets/scripts/app.js" defer></script>

The defer attribute, to make sure that the script that will be imported, will only be executed after the rest of the HTML document has been read and parsed. This ensures that, if the script code, for example, needs to work with some HTML elements, those HTML elements are available when the JavaScript code starts executing. Without defer, it would start executing right away. And if you then would try to reach out to say an unordered list, for example, that unordered list might not be available yet, if the script is loaded before the list is output here. That's why you might want to add defer.

Now in modern JavaScript projects, it's also quite common that you have another attribute on your script tags instead of having defer.

To use export and import features in various javascript files we need to add

<script src="Assets/scripts/app.js" type="module"></script>

in html file. here, type="module" is important.

From one js file i.e. **util.js**

export let apikey = "Swapnadip";

this **apikey** can be imported to another js file by i.e. **app.js**

import { apikey } from "./util.js";

console.log(apikey);

Also, while we can export as default. In case of default variable name is not required. Can directly export a value

export default "Swapnadip";

**Note: in a page there should be only one** default **parameter**.

While importing the same we can use

import apikeyNew from "./util.js";

console.log(apikeyNew);

Here apikeyNew is a variable name , and this can be anything as we are importing a default value.

In case there are more than one export values , then there several methods to call them-

export default "Swapnadip";

export let apikey = "Swapnadip Saha";

export let abc = "abc";

The methods to import them are

import { apikey , abc} from "./util";

console.log(abc);

console.log(apikey);

or

import \* as utils from "./util.js";

console.log(utils.abc);

console.log(utils.apikey);

console.log(utils.default);

In this case all the objects will be imported into the utils variable

import { apikey , abc as content} from "./util.js";

console.log(content);

or

import { apikey, abc, default as apikeynew } from "./util.js";

console.log(abc);

console.log(apikey);

console.log(apikeynew);

### Exporting functions

export default function (userName, message) {

return userName + message;

}

For default there should not be any function name.

export default (userName, message) => {

return userName + message;

}

While using default and => notation, there should not be any the function keyword either.

## Functions

A simple function

function combine(a, b, c) {

return a \* b / c;

}

### More on the Arrow Function Syntax

When working with Arrow Functions, you have a couple of "syntax shortcuts" available. Most importantly, you should know about the following alternatives:

#### 1) Omitting parameter list parentheses

If your arrow functions **take exactly one parameter**, you may **omit the wrapping parentheses**.

Instead of

(userName) => { ... }

you could write

userName => { ... }

Please note:

* If your function takes **no parameters**, **parentheses must not be omitted** –

 () => { ... } **is the only correct form in that case.**

If your function **takes more than one parameter**, you also **must not omit parentheses** - userName, userAge => { ... } would be invalid

(userName, userAge) => { ... } is correct!

#### 2) Omitting function body curly braces

If your arrow function contains **no other logic but a**returnstatement, you **may omit the curly brace**s and the return keyword.

Instead of

number => {

return number \* 3;

}

you could write

number => number \* 3;

The following code would be invalid:

number => return number \* 3; // invalid because return keyword must also be omitted!

number => if (number === 2) { return 5 }; // invalid because if statements can't be returned

The following three are the same

export default function (userName, message) {

  return userName + message;

};

OR

(userName, message) => {

  return userName + message;

};

OR

(userName, message) => userName + message;

#### 3) Special case: Just returning an object

If you go for the shorter alternative explained and you're trying to return a JavaScript object, you may end up with the following, invalid code:

number => { age: number }; // trying to return an object

This code would be invalid because JavaScript treats the curly braces as **function body wrappers** (not as code that creates a JS object).

To "tell" JavaScript that an object should be created (and returned) instead, the code would need to be adjusted like this:

number => ({ age: number }); // wrapping the object in extra parentheses

By wrapping the object and its curly braces with an **extra pair of parentheses**, JavaScript understands that the curly braces are not there to define a function body but instead to create an object. Hence that object then gets returned.

## Values and objects

Let’s say we have two variables

const userName = "Swapnadip";

const userAge = 34;

These can be combined into an object like

const user = {

Name : "Swapnadip", //property

Age : 34 //property

};

Now we can use

console.log(user) //this will give {Name = "Swapnadip", Age = 34 }

or

console.log(user.Name) //this will give Swapnadip

Objects can also contain functions  **without the function keyword**

const user = {

  Name: "Swapnadip", //property

  Age: 34, //property

  greet() {

    console.log("hello World");

  },

};

To access the function

console.log(user.greet());

//this will give hello World

or

const user = {

  Name: "Swapnadip", //property

  Age: 34, //property

  greet(userName) {

    return userName;

  },

};

console.log(user.greet("Swapnadip Saha from User"));

//this will give Swapnadip Saha from User

To access the objects with in the object function we can use this keyword

const user = {

  Name: "Swapnadip", //property

  Age: 34, //property

  greet(userName) {

    console.log(this.Age);

    return userName;

  },

};

console.log(user.greet("Swapnadip Saha from User"));

//this will give 34

Swapnadip Saha from User

Also, we can use class in javascript

class User {

  constructor(name, age) {

    this.name = name;

    this.age = age;

  }

  greet() {

    console.log("HI");

  }

}

const user1 = new User("Swapnadip", 35);

console.log(user1);

user1.greet();

//this will give {name: 'Swapnadip', age: 35}

HI

## Array

Objects in JavaScripts are key : value pair but arrays are values in a certain order.

const hobbies = ["Sports", "Music", "Reading "];

console.log(hobbies[1]);

//this will give Music

Arrays can contain objects, values even other arrays.

const content = [

  [

    "React is extremely popular",

    "It makes building complex, interactive UIs a breeze",

    "It's powerful & flexible",

    "It has a very active and versatile ecosystem",

  ],

  [

    "Components, JSX & Props",

    "State",

    "Hooks (e.g., useEffect())",

    "Dynamic rendering",

  ],

  [

    "Official web page (react.dev)",

    "Next.js (Fullstack framework)",

    "React Native (build native mobile apps with React)",

  ],

];

console.log(content[1]);

//This will give  *(4) ['Components, JSX & Props', 'State', 'Hooks (e.g., useEffect())', 'Dynamic rendering']*

console.log(content[1][2]);

//This will give Hooks (e.g., useEffect())

The content array contains arrays of character arrays.

### Functionalities:

hobbies.push("Cooking");

console.log(hobbies);

//This will give (4) ['Sports', 'Music', 'Reading ', 'Cooking']

The push will add another item into the array at the last.

#### findIndex() method.

const index=hobbies.findIndex((item) => {

    return item === "Cooking";

  })

  console.log(index);

//This will give 3

Here the arrow function

(item) => {

return item === "Cooking";

}

is similar to

function checkItem(item) {

return item === "Cooking";

}

\*\*\*\*checkItem is the name of the function is optional in arrow function

This findIndex() method allows to find the index of a certain value. This, findIndex () actually takes a function as an input, **and that's a great use case for using such an arrow function**,

This arrow function, which to pass to findIndex () **should accept at least one input parameter**, which could be called item, because findIndex () behind the scenes will execute this function, which is passing to findIndex (), and will automatically provide a value for this input parameter.

Therefore, of course, the function also must accept the parameter in order to then use it in this function body. Now, in this function body, in case of findIndex(), should return true if the array has the item and false otherwise.

For that, we can return item === "Cooking", For example, if we were looking for the index of the item Sports. So, if we were looking for the index of this item. What this code here does is it executes this function automatically behind the scenes for every item in this array, including this ("Cooking")item, which was pushed onto the array. And then for every item, it compares that item.

So, with help of this comparison operator, if the two are equal, this function here returns true, and therefore findIndex knows that it found the item, and it will then give the index of that item. Otherwise, this will return false, and findIndex will do nothing.

So, therefore, here store that index in a const index and then console.log it.

All that's happening here is that findIndex needs a function, which it can execute on our behalf, and it will execute this function for every item in this array. It will pass that item for every execution into that function. And, therefore, of course, item will be different for every execution. It will be those items here. And then we compare the item we're getting to some value we are looking for.

const index=hobbies.findIndex((item) => {

    return item === "Cooking";

  })

  console.log(index);

Shortcut of the above snippet is

const index1 = hobbies.findIndex((item) => item === "Sports");

console.log(index1);

//This will give 0

#### Map()

map() is used to append things with the existing array element. But as this do not change the original array. This needs to be stored into another array.

map() allows to transform every item in an array to another item. For that map(), just like findIndex(),takes a function as an input, typically such **an arrow function,** And like this arrow function for findIndex this **arrow function** here also will receive every item in the array as an input because this function also will be executed automatically by map() for every item in the array, and every item off the array will then be provided as an input value to this function when it's being executed. And then here returns the value of this item should be transformed to,

const editedHobbies = hobbies.map((item) => item + "!");

console.log(editedHobbies);

//Output

//['Sports!', 'Music!', 'Reading!', 'Cooking!']

Here just mapping my items to strings with an exclamation mark at the end. And what map() will do is it'll not edit the original array. Instead, that array will stay unchanged. And instead, map will return a new array,

Now, what's important to note about map() is that use it to transform any item to any other kind of item. For example, here, we don't have to convert our strings to new strings. Instead, I could also convert them to JavaScript objects.

`For that, create an object with opening and closing curly braces, which here, however, would have to be wrapped with parentheses since, otherwise, they would be treated as the parentheses that wrapped the function body. If, instead, want to return a JavaScript object, needs to wrap the curly braces with parentheses. And this will tell JavaScript that these curly braces will not define the **function body of this arrow function**, but instead will define a new object returned by that arrow function. And then here, could define any key of own choice. And, for example, store the item, which in this case will be such a string as a value for that key. Really, you can create any kinds of values here, any objects of any shape, numbers, strings, Booleans, whatever you want .

const editedHobbies2 = hobbies.map((item) => ({ text: item }));

console.log(editedHobbies2);

//Output

//[

// { text: 'Sports' },

// { text: 'Music' },

// { text: 'Reading' },

// { text: 'Cooking' }

//]

console.log(editedHobbies2[1]);

//Output

//{text: 'Music'}

<div id="tab-content">

  <ul>

    {content[activeContentIndex].map((item) => (

      <li key={item}>{item}</li>

    ))}

  </ul>

</div>;

## Destructuring

const userName = ["Swapnadip", "Saha"];

const firstName = userName[0];

const lastName = userName[1];

console.log(firstName, lastName);

This can be destructrued as

const [firstName, lastName] = ["Swapnadip", "Saha"];

console.log(firstName, lastName);

This Destructuring is also available for objects not only for arrays

const user = {

  Name: "Swapnadip",

  Age: 34,

};

const name = user.Name;

const age = user.Age;

console.log(name, age);

This can be destructrued as

const { Name, Age } = {

  Name: "Swapnadip",

  Age: 34,

};

console.log(Name, Age);

Here the object name and the variables names should be same.

We can also user alias like

const { Name:userNameObj, Age :userAge} = {

  Name: "Swapnadip",

  Age: 39,

};

console.log(userNameObj, userAge);

In this case

console.log(Name, Age);

will be error. We need to use the alias name.

## Spread Operator

Let say there are two arrays:

const hobbies = ["Sports", "Music", "Reading"];

const newHobbies = ["Cooking"];

Now doing the following will create arrays in side array, it will create arrays of array

const mergeHobbies = [hobbies, newHobbies];

console.log(mergeHobbies);

//Output

//[ [ 'Sports', 'Music', 'Reading' ], [ 'Cooking' ] ]

console.log(mergeHobbies[1]);

//Output

['Cooking']

console.log(mergeHobbies[0]);

//Output

['Sports', 'Music', 'Reading']

Instead use the following it will merge the arrays into one

const mergeHobbies1 = [...hobbies, ...newHobbies];

console.log(mergeHobbies1);

// Output

//[ 'Sports', 'Music', 'Reading', 'Cooking' ]

**This is spread operator**.

Same can be done with objects also—

const user = {

  Name: "Swapnadip",

  Age: 34,

};

const extendeduser = {

  isAdmin: true,

  user,

};

console.log(extendeduser);

//output

*{isAdmin: true, user: Object}*

**isAdmin**: true

**user**:

**Age**: 34

**Name**: "Swapnadip"

Here one object is merged with another object with in the object.

**Using spread operator, both objects are merged into one**

const user = {

  Name: "Swapnadip",

  Age: 34,

};

const extendeduser = {

  isAdmin: true,

  ...user,

};

console.log(extendeduser);

//output

{isAdmin: true, Name: 'Swapnadip', Age: 34}

Age: 34

Name: "Swapnadip"

isAdmin: true

or

const extendeduser = {

  ...user,

  isAdmin: true,

};

console.log(extendeduser);

//output

//{ Name: 'Swapnadip', Age: 34 , isAdmin: true}

## For loop

const hobbies = ["Sports", "Music", "Reading"];

for (const hobby of hobbies) {

  console.log(hobby);

}

prompt() and setTimeOut() are some built in function.

## Passing function as value

function manageTimeOut() {

  console.log("Timed out!");

}

const newManageTimeOut = () => {

  console.log("Timed out!....Again");

};

manageTimeOut();

newManageTimeOut();

setTimeout(manageTimeOut);

here passing the manageTimeOut() without the parenthesis as value. Passing with the parenthesis will throw exception as function cannot be pass as parameter to a function

setTimeout(newManageTimeOut, 5000);

it will take milliseconds also as a parameter, when should the thing will timeout. Also, we can use an anonymous function

setTimeout(() => {

  console.log("More Timed out!....Again");

}, 3000);

Can be used for custom functions also,

function greeter(greetfn) {

  greetfn();

}

greeter(() => {

  console.log("Hi");

});

This **arrow function** here is getting executed because we're passing it as a value, for this greetfn parameter to the greeter()function, and inside of that greeter () function executing this greetFn() parameter, so, the value that's received on that parameter, which is this arrow function. So, the arrow function is passed as a value to greetFn, and then greetFn is executed inside greeter ().

Therefore, indirectly, the arrow function is getting executed there because the arrow function at the bottom is the value received on greetFn. It really is important to be aware of the fact that passing functions as values is not limited to built-in functions like setTimeout, but can be done with all functions, including own functions. Those can also accept functions as input.

## Functions within function

function init() {

  function greet() {

    console.log("Hi All");

  }

  greet();

}

init();

Here the greet()function can only be call into inside the init()function , not from outside. As here is local to init(). That is in the scope of init()

## Reference vs primitive values

let userMessage = "Hello";

console.log(userMessage);

userMessage = "Hello overwritten";

console.log(userMessage);

userMessage = userMessage.concat(" Again");

console.log(userMessage);

userMessage = [];

console.log(userMessage);

//Output

Hello

Hello overwritten

Hello overwritten Again

[]

string, Boolean, numbers these are primitive values, as they cannot be edited, but can be overwrite. Here the userMessage in second line is a brand new string. Also, the using of concat here will also provide a new string, instead editing the earlier string. These are primitive value.

But in arrays, objects etc we can edit the existing values.

const hobbies = ["Sports", "Music", "Reading"];

console.log(hobbies);

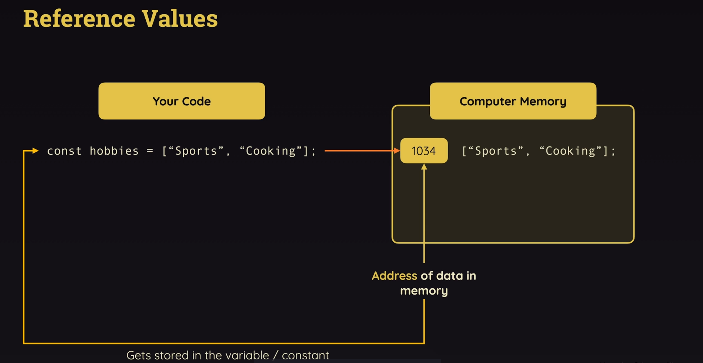
hobbies.push("Cooking");

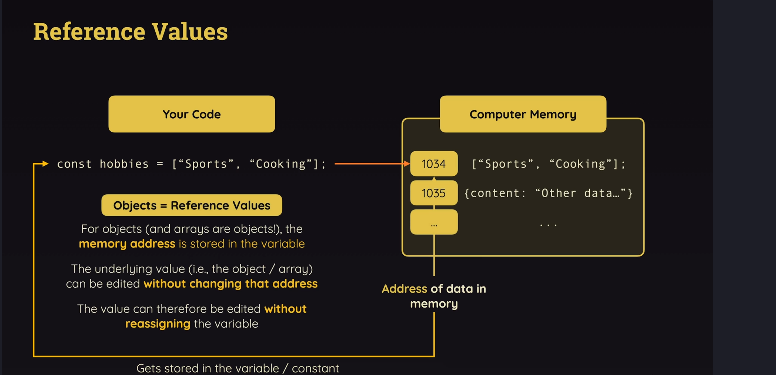
console.log(hobbies);

These are reference values. Here

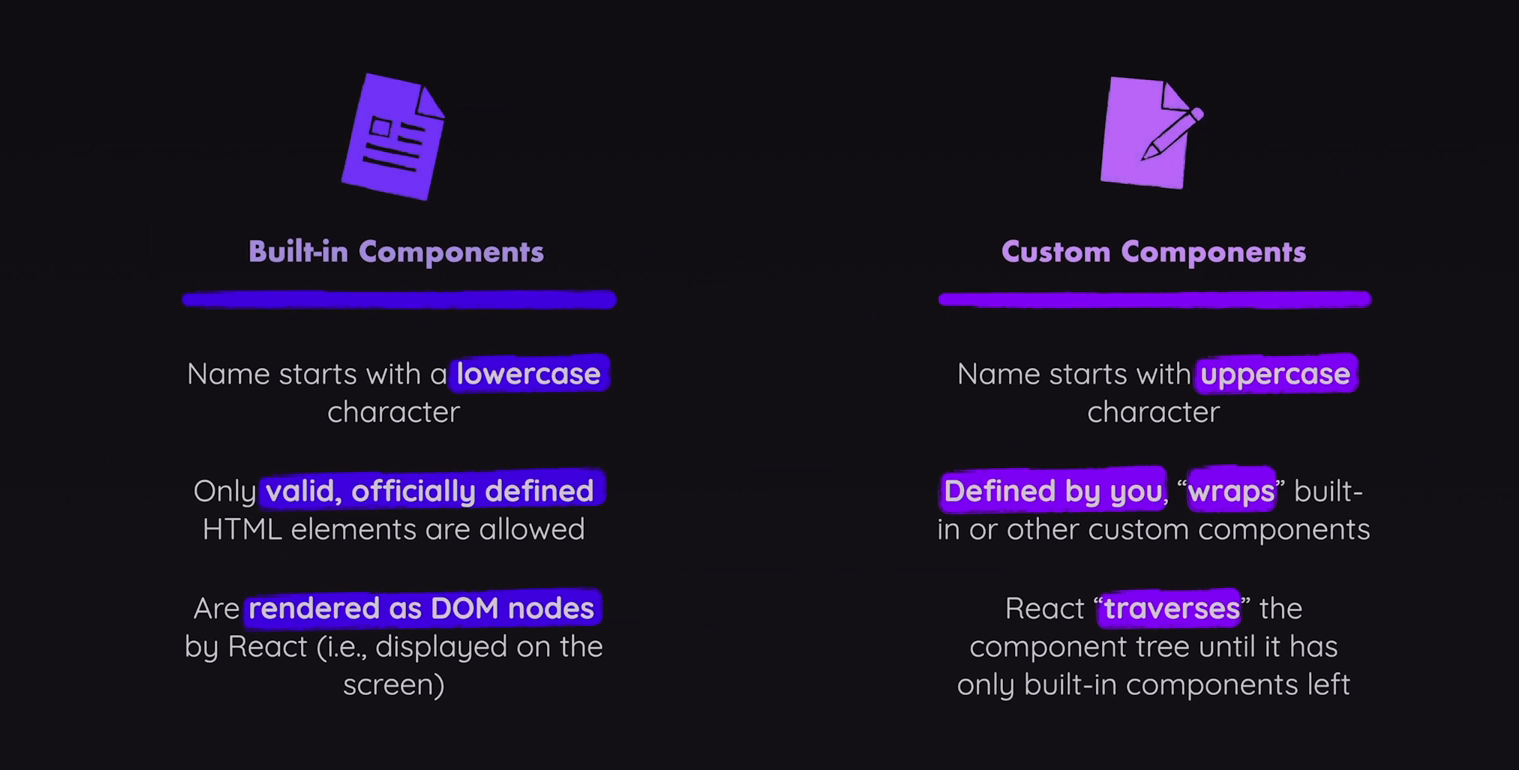
hobbies = [];

this won’t work as hobbies is const, so it cannot be edited, only can add value into it,





# Building Components in jsx



So as a first step, let's go to the website and let's inspect the source code of that website. It just contains some metadata and at least one JavaScript import, one JavaScript file that's being loaded. But this loaded JavaScript file, the index.jsx file in the end is the key because the code that is loaded and executed here is, in the end, the transformed React code you wrote.

index.jsx

import ReactDOM from "react-dom/client";

import App from "./App.jsx";

import "./index.css";

const entryPoint = document.getElementById("root");

ReactDOM.createRoot(entryPoint).render(<App />);

The code you written in the index.jsx wouldn't work like this in the browser, hence, it's transformed.

The index.jsx file being loaded here in the actual website source code because that's also what we see here in the index.html file.

index.html

<!DOCTYPE html>

<html lang="en">

  <head>

    <meta charset="UTF-8" />

    <link rel="icon" type="image/svg+xml" href="/vite.svg" />

    <meta name="viewport" content="width=device-width, initial-scale=1.0" />

    <title>React Essentials</title>

  </head>

  <body>

    <div id="root"></div>

    <script type="module" src="/src/index.jsx"></script>

  </body>

</html>

Now the index.jsx file does import something from the App.jsx file. It imports the App component to be precise because it's that App component that's being exported in that App.jsx file.

App.jsx

function Header(){

    return(

        <header>

          <img src="src/assets/react-core-concepts.png" alt="Stylized atom" />

          <h1>React Essentials</h1>

          <p>

            Fundamental React concepts you will need for almost any app you are

            going to build!

          </p>

        </header>

    )

}

function App() {

    return (

      <div>

        {/\* <header>

          <img src="src/assets/react-core-concepts.png" alt="Stylized atom" />

          <h1>React Essentials</h1>

          <p>

            Fundamental React concepts you will need for almost any app you are

            going to build!

          </p>

        </header> \*/

        <Header></Header>

        }

        <main>

          <h2>Time to get started!</h2>

        </main>

      </div>

    );

  }

  export default App;

**// the header section moved to new component or function Header().**

**// rule --> the function or component will be always initcap and the function name will be the tag here**

So, it's this App component function that's being imported into the index.jsx file. And in this file, it's being used as JSX code here.

There is no React component here though. This JSX code is not getting returned by some function. Instead, it's getting used as a value, i.e. **entryPoint** as an argument for some other method **ReactDOM.createRoot(entryPoint)** that's being called here, the **render(<App />)** method.

index.jsx

import ReactDOM from "react-dom/client";

import App from "./App.jsx";

import "./index.css";

const entryPoint = document.getElementById("root");

ReactDOM.createRoot(entryPoint).render(<App />);

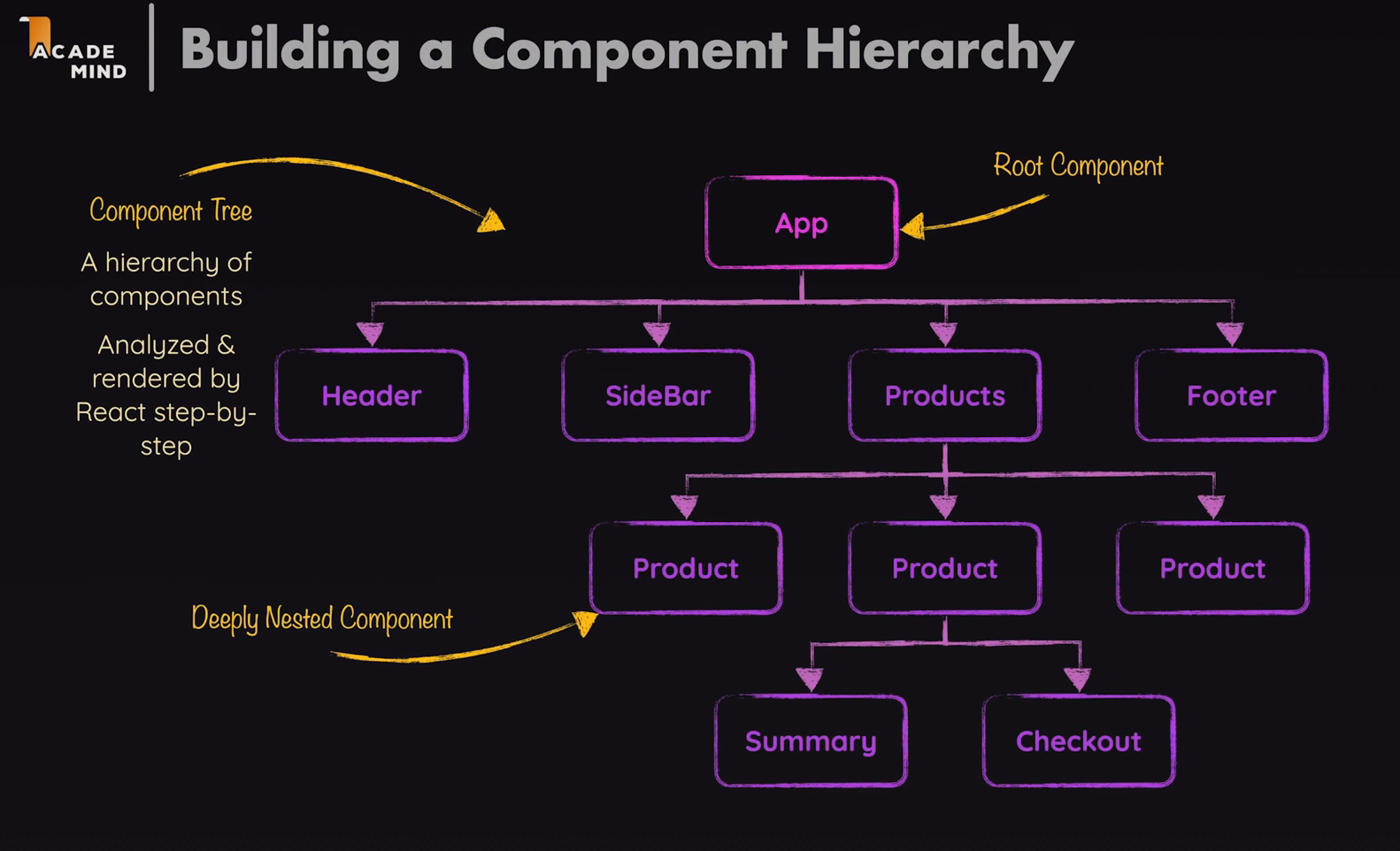
And indeed, as a React developer, you will almost exclusively use JSX code inside of component functions. The index.jsx file is the one important exception because this file, acts as the main entry point of the React app since it is the first file to be loaded by the HTML file.

And it's in this place where the React app boots up, it's this special React DOM library, which in the end belongs to the overall React library from which we're importing here which ultimately renders this App component. So, which is responsible for outputting the App component's content on the screen. And this App component is rendered by passing JSX code to this **render** method.

This **render** method, however, is being called on an object that's created with another method, the **createRoot** method. This method takes an existing HTML element as an input, that’s not being created by React but that instead is part of the index.html file already. In this case, that would be this div here with the id root. Since that's what we're selecting here with getElementById, that's what's getting passed to this **createRoot** method. And then with that element selected and set as a root for the React project.

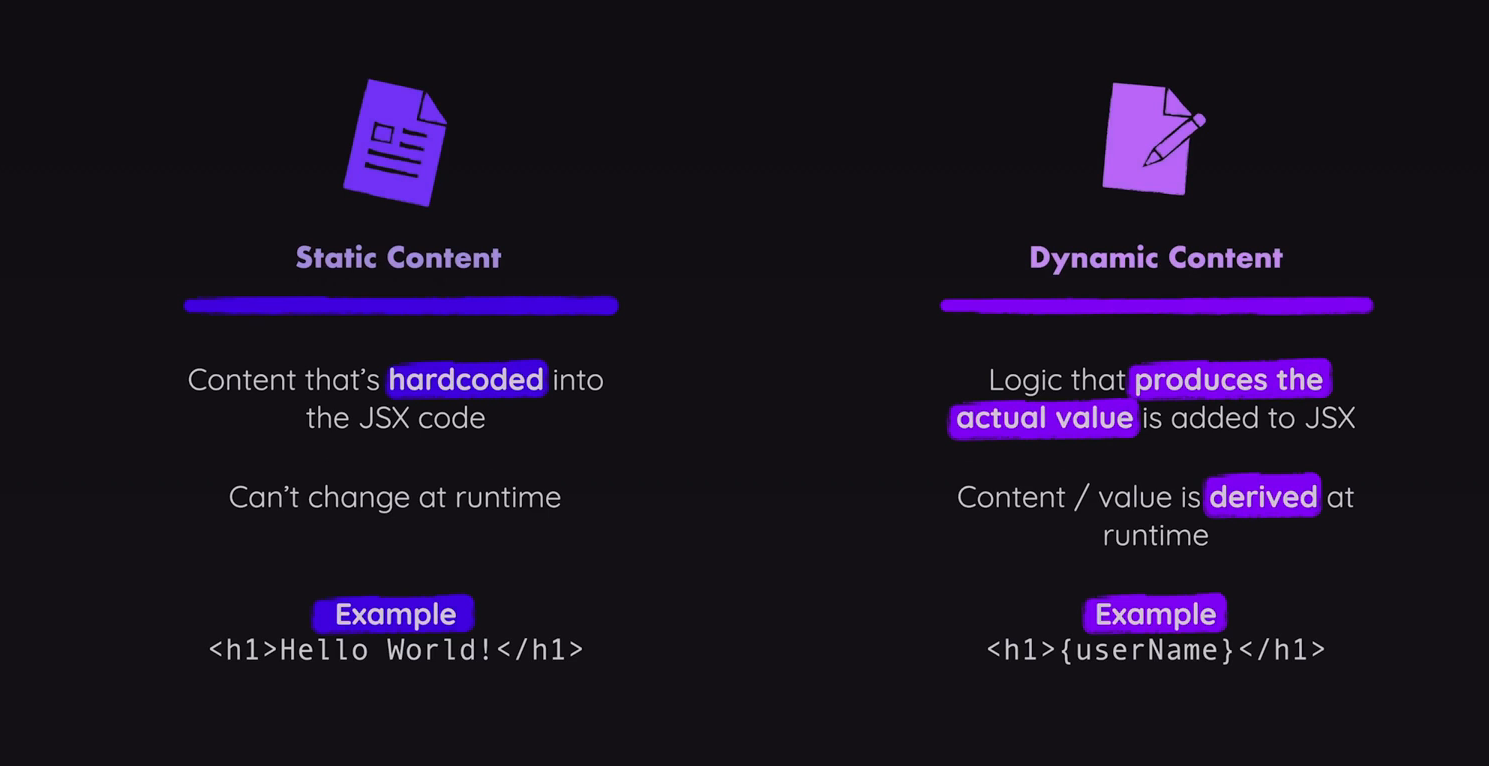
React goes ahead and injects a React component, the **(<App />)** component in this case, into this element. It renders this component and all its contents, including any nested components it may contain and their content into this div. That's what this **render** method does. And that's why if you open the developer tools and there the elements tab, you see more than just the initial HTML structure. Instead, you see all the elements that are indeed visible on the screen and you see that they are inside of that div here with the id root.

So, the **createRoot** and **render** methods are responsible for rendering a single root component, the **App** component in this case, which then in turn may contain as many nested components as needed. And those nested components, it may include like this Header component in this case, could then contain even more child components. And with that, ultimately you end up with a component hierarchy, which is often called a tree of components, a structure of components, which is then rendered to the screen via React.



But what's important to understand about this tree of components and components in general is that your custom components are not showing up in the actual rendered DOM though. There you only find default HTML elements for example, here, the header with the lowercase h, which is the built-in header element, not our custom component, which had an uppercase H. We also don't see the App component in here. So, your tree of components is, in the end, just analysed by React. And React then combines all the JSX code from all those components to generate the overall DOM, the elements that are showing up on the screen.

# Statics vs Dynamic contents in Components



App.jsx

const dynamicValue=["Fundamental","Crucial","Core"];

function genRandomInt(max){

  return Math.floor(Math.random()\*(max+1));

}

function Header(){

    return(

        <header>

          <img src="src/assets/react-core-concepts.png" alt="Stylized atom" />

          <h1>React Essentials</h1>

          <p>

            {dynamicValue[genRandomInt(2)]} React concepts you will need for almost any app you are going to build!

          </p>

        </header>

    )

}

Previously written **Fundamental** is now replaced by **{dynamicValue[genRandomInt(2)]}.**

Here **dynamicValue** is a variable which is contains some values. And every time the page loads depending on the values of **genRandomInt(2)** any thing will be displayed. In those **{}** of dynamic value syntax anything can be written, only if else statements, for loops etc cannot be written. For example, we can write

{1+1} React concepts you will need for almost any app you are going to build!

In this case always 2 will be printed. Like this... Also, in place of

**<img src="src/assets/react-core-concepts.png" alt="Stylized atom" />**

We can write

**<img src= {….} alt="Stylized atom" />**

import randomImage from "./assets/react-core-concepts.png"

const dynamicValue=["Fundamental","Crucial","Core"];

function genRandomInt(max){

  return Math.floor(Math.random()\*(max+1));

}

function Header(){

  const description=dynamicValue[genRandomInt(2)];

    return(

        <header>

          <img src={randomImage} alt="Stylized atom" />

          <h1>React Essentials</h1>

          <p>

            {description} React concepts you will need for almost any app you are

            going to build!

          </p>

        </header>

    )

}

We can use dynamic values for attributes also. Here we can use the link of the image

Alternatively, we can also do like this,

const dynamicValue=["Fundamental","Crucial","Core"];

function genRandomInt(max){

  return Math.floor(Math.random()\*(max+1));

}

function Header(){

  const description=dynamicValue[genRandomInt(2)];

    return(

        <header>

          <img src="src/assets/react-core-concepts.png" alt="Stylized atom" />

          <h1>React Essentials</h1>

          <p>

            {description} React concepts you will need for almost any app you are

            going to build!

          </p>

        </header>

    )

}

This is recommended and preferable for a nit and clean code.

Example with destructuring

  import React from 'react';

export const userData = {

  firstName: 'Maximilian', // feel free to replace the name value

  lastName: 'Schwarzmüller', // feel free to replace the name value

  title: 'Instructor', // feel free to replace the title value

};

// Edit the User component code to output the userData data

export function User() {

  // Destructure the userData object

  const { firstName, lastName, title } = userData;

  return (

    <div id="user" data-testid="user">

      <h2>

        {firstName} {lastName}

      </h2>

      <p>{title}</p>

    </div>

  );

}

// DON'T edit the App component code

function App() {

  return (

    <div id="app">

      <h1>Time to Practice</h1>

      <p>Welcome on board of this course! You got this 💪</p>

      <User />

    </div>

  );

}

export default App;

**Explanation**

1. **Destructuring**:
   * Inside the User component, destructure firstName, lastName, and title from the userData object.
   * This makes it easier to use these properties in the JSX.
2. **Interpolate values in JSX**:
   * Use {firstName} {lastName} within the <h2> element to display the full name.
   * Use {title} within the <p> element to display the user's title.

# Props

One of the main advantages of components is that they are reusable, we can, for example, use this <Header></Header>component as often as we want. And if we do that, we, of course, see multiple headers on this page.

For example, these CoreConcept items might involve building one single core concept component, which can then be reused four times for these four different key concepts, but every time it's used, it should be with different data. Just as you can define a normal JavaScript function once and then use it multiple times, thanks to working with parameters, you can build and reuse JavaScript functions with different data. Similarly, we can build and reuse certain React components with different input data. That's why React offers another crucial concept related to components called props, which is about being able to pass data into components and use that data within them. In our project, for the moment still in the app.jsx file (though that will change soon), we can add a new component function, perhaps called CoreConcept, for outputting core concept data. This component could output a list item containing an image with a source and an alt tag, an h3 tag with a title, and a paragraph with a description. The goal is to replace placeholders with actual data that's different every time this component gets used. With this component defined, we can go to the app component and add a new section in the main area with an ID of core concepts (for styling purposes, as defined in the index.css file). Add an h2 tag with "Core Concepts" and below that an unordered list, then use the newly added core concept component multiple times, passing different data each time. Thanks to the props concept, this is easy. You can add custom attributes to your components—entirely up to you, as these are your components. For example, add a title attribute with a value of "components" for the first usage, a description attribute (a prop, since configuring components is called props in React) like "the core UI building block," and an image prop set to a dynamic value by importing an image (e.g., components image from ./assets/component.png). Use the imported components image as a value for the image prop. Props can hold strings, numbers, objects, arrays, or anything you need. With data passed to the component function, you can accept and use it there. In React component functions, you typically accept one parameter called props, though you could name it anything. React sets this parameter when it executes the function under the hood, passing an object with key-value pairs where keys match custom attributes and values are the corresponding values. For example, access props.image to get the value set for the image key. Consistency in keys is critical—what's set in the component call must match in the function. Using props.image, props.title, and props.description within the component will render the content on-screen. By reusing the component with different data, such as changing the title to "props" for a second core concept item, you can render varied outputs. Similarly, you can set the image, description, and other data for additional items. This is how the props concept works.

