Npm install

Npm start

# JavaScript refreshers

## Export and Import

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";

console.log(apikey);

Also, while we can export as default

export default "Swapnadip";

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

While importing the same we can use

import apikey from "./util";

console.log(apikey);

Here apikey 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

1. import { apikey , abc} from "./util";
2. import \* as utils from "./util";

console.log(utils.abc);

console.log(utils.apikey);

console.log(utils.default);

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

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

console.log(content);

### 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

#### 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",

Age : 34,

greet(userName) {

return userName;

}

};

To access the function

user.greet("Swapnadip")

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

const user = {

Name : "Swapnadip",

Age : 34,

greet(userName) {

console.log(this.Age);

return userName;

}

};

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();

## 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]);

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)"

]

];

The content array contains arrays of character arrays.

### Functionalities:

hobbies.push("Cooking");

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

#### findIndex() method.

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

return item === "Sports";

})

console.log(index);

This is a method that allows you to find the index of a certain value. For 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 === "Sports", 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 === "Sports";

})

console.log(index);

Shortcut of the above snippet is

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

console.log(index1);

#### 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' }

//]

<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

const mergeHobbies = [hobbies, newHobbies];

console.log(mergeHobbies);

//Output

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

Instead use the following it will merge the arrays

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, Name: 'Swapnadip', Age: 34 }

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");

}

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, 2000);

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");

}

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);

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);

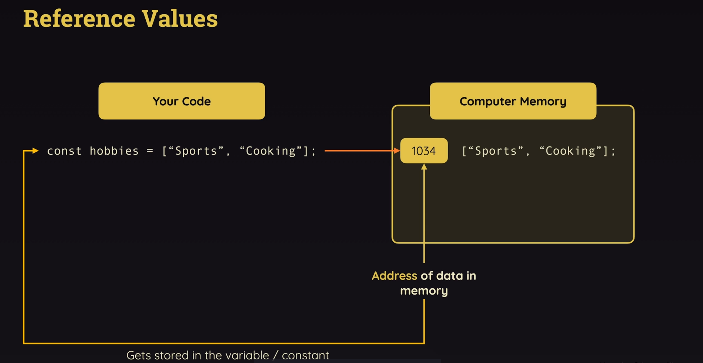
hobbies.push("Cooking");

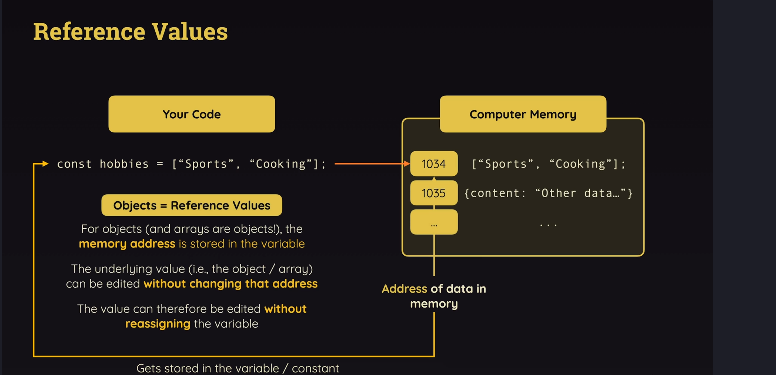
console.log(hobbies);

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,





## Summary

let  and const  basically replace var . You use let  instead of var  and const  instead of var  if you plan on never re-assigning this "variable" (effectively turning it into a constant therefore).

let : <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let>

const : <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const>

Arrow Functions: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions>

Arrow functions are a different way of creating functions in JavaScript. Besides a shorter syntax, they offer advantages when it comes to keeping the scope of the this  keyword (see [here](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions#No_binding_of_this)).

Arrow function syntax may look strange but it's actually simple.

1. function callMe(name) {
2. console.log(name);
3. }

which you could also write as:

1. const callMe = function(name) {
2. console.log(name);
3. }

becomes:

1. const callMe = (name) => {
2. console.log(name);
3. }

**Important:**

When having **no arguments**, you have to use empty parentheses in the function declaration:

1. const callMe = () => {
2. console.log('Max!');
3. }

When having**exactly one argument**, you may omit the parentheses:

1. const callMe = name => {
2. console.log(name);
3. }

When **just returning a value**, you can use the following shortcut:

1. const returnMe = name => name

That's equal to:

1. const returnMe = name => {
2. return name;
3. }

### **Exports & Imports**

In React projects (and actually in all modern JavaScript projects), you split your code across multiple JavaScript files - so-called modules. You do this, to keep each file/ module focused and manageable.

To still access functionality in another file, you need export  (to make it available) and import  (to get access) statements.

You got two different types of exports: **default** (unnamed) and **named** exports:

**default** => export default ...;

**named** => export const someData = ...;

You can import **default exports** like this:

import someNameOfYourChoice from './path/to/file.js';

Surprisingly, someNameOfYourChoice  is totally up to you.

**Named exports** have to be imported by their name:

import { someData } from './path/to/file.js';

A file can only contain one default and an unlimited amount of named exports. You can also mix the one default with any amount of named exports in one and the same file.

When importing **named exports**, you can also import all named exports at once with the following syntax:

import \* as upToYou from './path/to/file.js';

upToYou  is - well - up to you and simply bundles all exported variables/functions in one JavaScript object. For example, if you export const someData = ...  (/path/to/file.js ) you can access it on upToYou  like this: upToYou.someData .

### **Classes**

Classes are a feature which basically replace constructor functions and prototypes. You can define blueprints for JavaScript objects with them.

Like this:

1. class Person {
2. constructor () {
3. this.name = 'Max';
4. }
5. }
7. const person = new Person();
8. console.log(person.name); // prints 'Max'

In the above example, not only the class but also a property of that class (=> name ) is defined. The syntax you see there, is the "old" syntax for defining properties. In modern JavaScript projects (as the one used in this course), you can use the following, more convenient way of defining class properties:

1. class Person {
2. name = 'Max';
3. }
5. const person = new Person();
6. console.log(person.name); // prints 'Max'

You can also define methods. Either like this:

1. class Person {
2. name = 'Max';
3. printMyName () {
4. console.log(this.name); // this is required to refer to the class!
5. }
6. }
8. const person = new Person();
9. person.printMyName();

Or like this:

1. class Person {
2. name = 'Max';
3. printMyName = () => {
4. console.log(this.name);
5. }
6. }
8. const person = new Person();
9. person.printMyName();

The second approach has the same advantage as all arrow functions have: The this keyword doesn't change its reference.

You can also use **inheritance** when using classes:

1. class Human {
2. species = 'human';
3. }
5. class Person extends Human {
6. name = 'Max';
7. printMyName = () => {
8. console.log(this.name);
9. }
10. }
12. const person = new Person();
13. person.printMyName();
14. console.log(person.species); // prints 'human'

### **Spread & Rest Operator**

The spread and rest operators actually use the same syntax: ...

Yes, that is the operator - just three dots. Its usage determines whether you're using it as the spread or rest operator.

**Using the Spread Operator:**

The spread operator allows you to pull elements out of an array (=> split the array into a list of its elements) or pull the properties out of an object. Here are two examples:

1. const oldArray = [1, 2, 3];
2. const newArray = [...oldArray, 4, 5]; // This now is [1, 2, 3, 4, 5];

Here's the spread operator used on an object:

1. const oldObject = {
2. name: 'Max'
3. };
4. const newObject = {
5. ...oldObject,
6. age: 28
7. };

newObject  would then be

1. {
2. name: 'Max',
3. age: 28
4. }

The spread operator is extremely useful for cloning arrays and objects. Since both are [reference types (and not primitives)](https://youtu.be/9ooYYRLdg_g), copying them safely (i.e. preventing future mutation of the copied original) can be tricky. With the spread operator you have an easy way of creating a (shallow!) clone of the object or array.

### **Destructuring**

Destructuring allows you to easily access the values of arrays or objects and assign them to variables.

Here's an example for an array:

1. const array = [1, 2, 3];
2. const [a, b] = array;
3. console.log(a); // prints 1
4. console.log(b); // prints 2
5. console.log(array); // prints [1, 2, 3]

And here for an object:

1. const myObj = {
2. name: 'Max',
3. age: 28
4. }
5. const {name} = myObj;
6. console.log(name); // prints 'Max'
7. console.log(age); // prints undefined
8. console.log(myObj); // prints {name: 'Max', age: 28}

Destructuring is very useful when working with function arguments. Consider this example:

1. const printName = (personObj) => {
2. console.log(personObj.name);
3. }
4. printName({name: 'Max', age: 28}); // prints 'Max'

Here, we only want to print the name in the function but we pass a complete person object to the function. Of course, this is no issue but it forces us to call personObj.name inside of our function. We can condense this code with destructuring:

1. const printName = ({name}) => {
2. console.log(name);
3. }
4. printName({name: 'Max', age: 28}); // prints 'Max')

We get the same result as above but we save some code. By destructuring, we simply pull out the name  property and store it in a variable/ argument named name  which we then can use in the function body.

### JS Array Functions

Not really next-gen JavaScript, but also important: JavaScript array functions like map() , filter() , reduce()  etc.

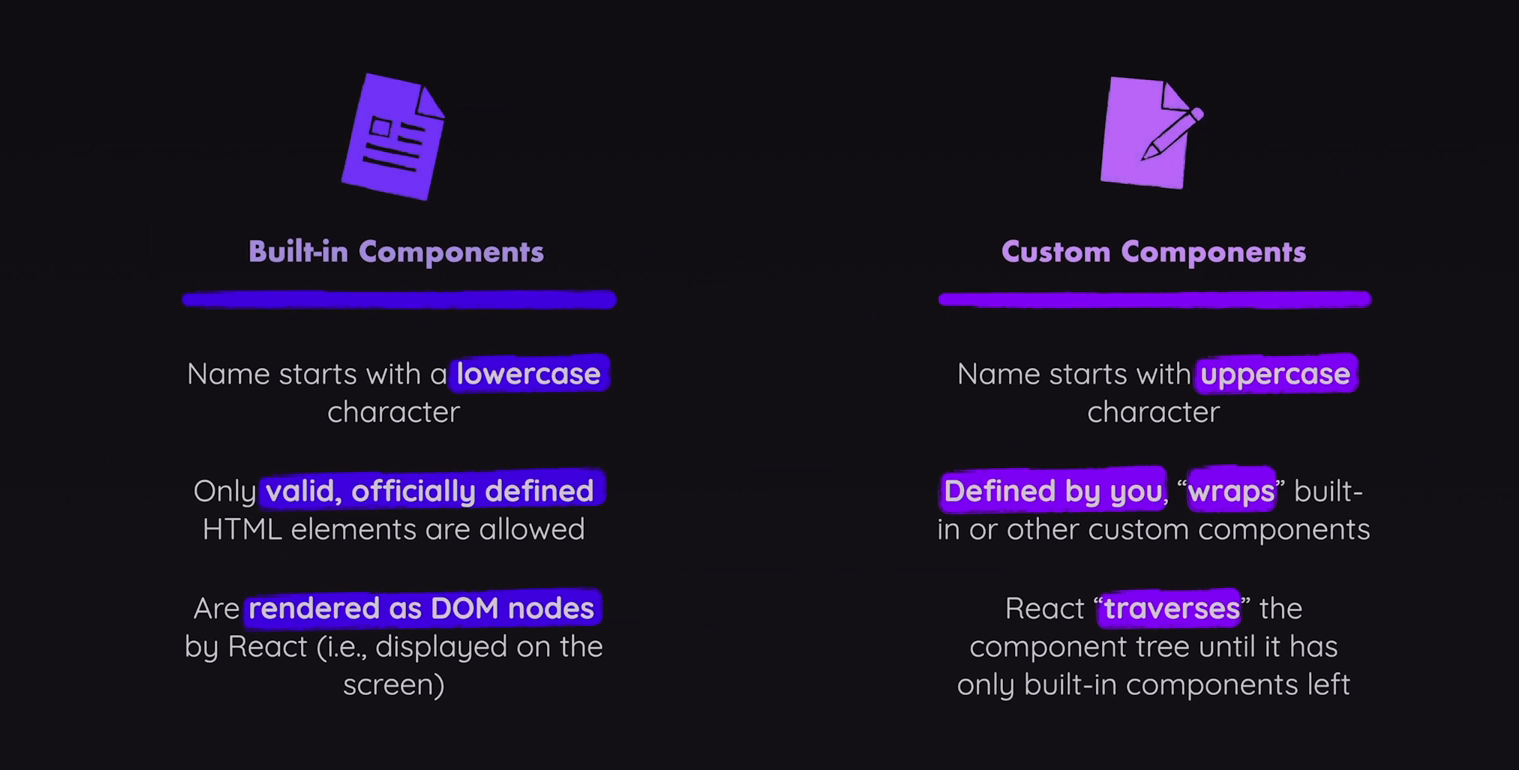
You'll see me use them quite a bit since a lot of React concepts rely on working with arrays (in immutable ways).

The following page gives a good overview over the various methods you can use on the array prototype - feel free to click through them and refresh your knowledge as required: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array>

Particularly important in this course are:

* map()  => <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map>
* find()  => <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/find>
* findIndex()  => <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/findIndex>
* filter()  => <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter>
* reduce()  => <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce?v=b>
* concat()  => <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/concat?v=b>
* slice()  => <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/slice>
* splice() => <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice>

# 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 />);

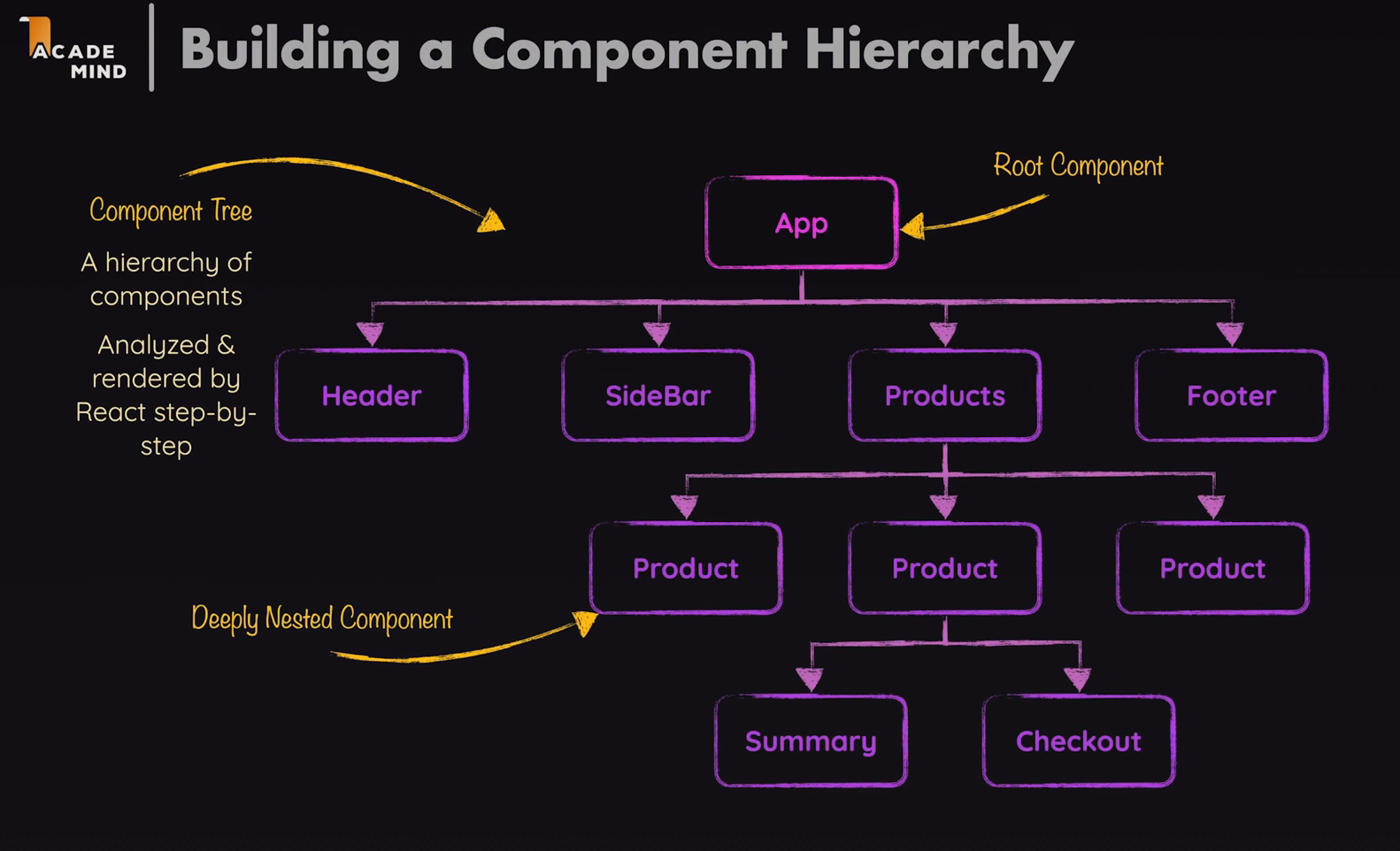
And indeed, as a React developer, you will almost exclusively use JSX code inside of component functions. The index.jsx file is the one important exception because this file, acts as the main entry point of the React app since it is the first file to be loaded by the HTML file.

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.

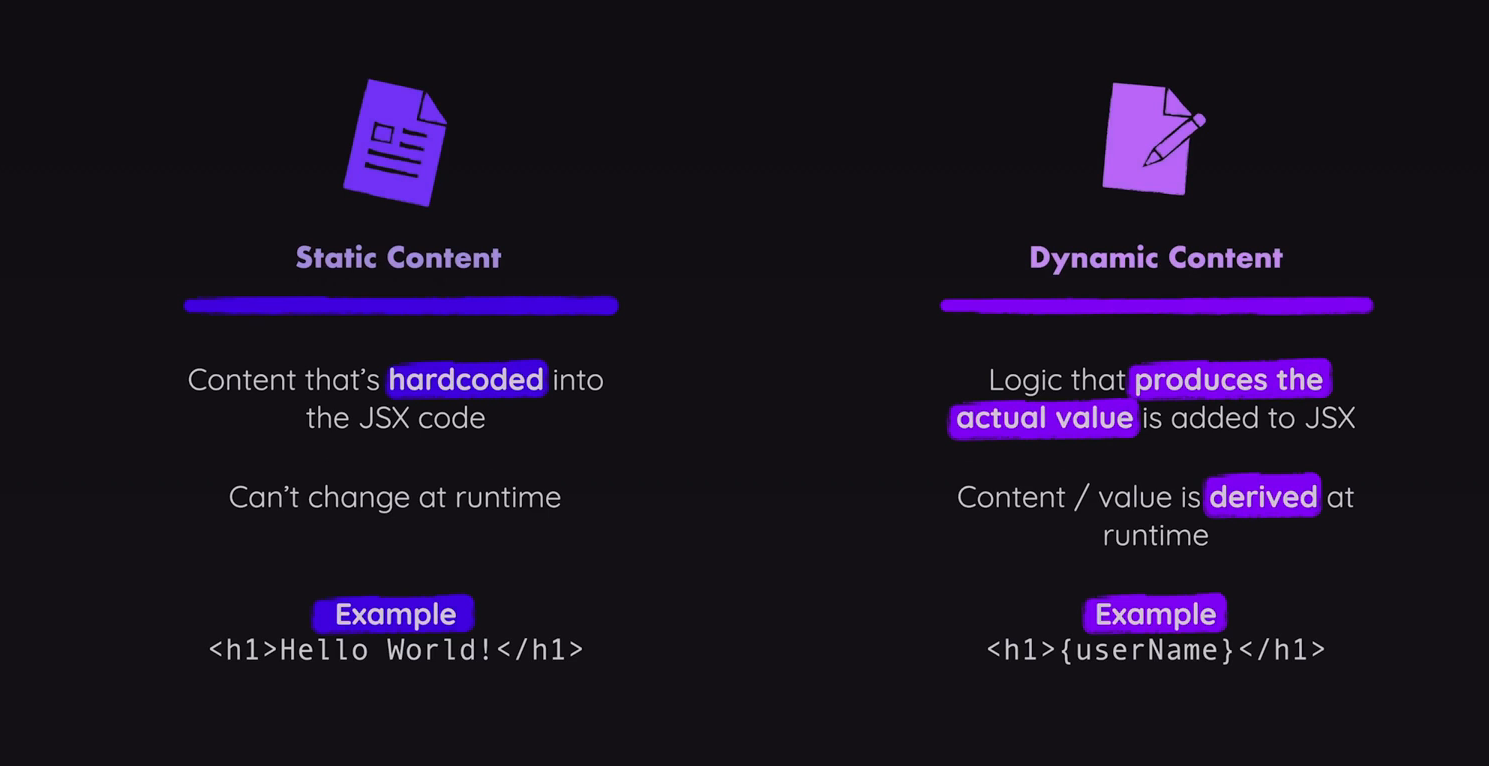
React goes ahead and injects a React component, the **(<App />)** component in this case, into this element. It renders this component and all its contents, including any nested components it may contain and their content into this div. That's what this **render** method does. And that's why if you open 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.

