**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 takes **exactly one parameter**, you may **omit** the wrapping parentheses.

Instead of

1. (userName) => { ... }

you could write

1. 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 return statement**, you may **omit the curly braces** and the return keyword.

Instead of

1. number => {
2. return number \* 3;
3. }

you could write

1. number => number \* 3;

The following code would be invalid:

1. number => return number \* 3; // invalid because return keyword must also be omitted!
2. 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 in 2) and you're trying to return a **JavaScript object**, you may end up with the following, **invalid** code:

1. 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:

1. 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.

Array methods

Hobbies=[ “sports”,”movies”]

1.push().

2.findIndexOf()->it takes a function as input.

Find index of(( item)=>{

Item===”sports”;

});

By removing parenthesis.

Find index of(( item)=>

Item===”sports”;

);

3.map((item )=>{

Item=item+”!”;

});

It is used to modify the array

4.map(item)=>{

({val:item});

}

function transformToObjects(numberArray) {

let numberArray11= numberArray .map(item=>

({val:item}));

Return numberArray11;

// Todo: Add your logic

// should return an array of objects

}

DE structuring for arrays

Const [username ,age]=[“rakesh”,”34”];

Console.log(username)

DE structuring for objects.

Const user={

Name:”rakesh”,

Age:”24”

};

//

Const {name : username,age}={

name:”rakesh”,

age:”24”

};

**Destructuring in Function Parameter Lists**

The destructuring syntax explained in the previous lecture can also be used in **function parameter lists**.

For example, if a function accepts a parameter that will **contain an object** it can be destructured to *"pull out"* the object properties and make them available as **locally scoped variables** (i.e., variables only available inside the function body).

Here's an example:

1. function storeOrder(order) {
2. localStorage.setItem('id', order.id);
3. localStorage.setItem('currency', order.currency);
4. }

Instead of accessing the order properties via the *"dot notation"* inside the storeOrder function body, you could use destructuring like this:

1. function storeOrder({id, currency}) { // destructuring
2. localStorage.setItem('id', id);
3. localStorage.setItem('currency', currency);
4. }

The destructuring syntax is the same as taught in the previous lecture - just without creating a constant or variable manually.

Instead, id and currency are *"pulled out"* of the incoming object (i.e., the object passed as an argument to storeOrder).

It's very important to understand, that storeOrder **still only takes one parameter** in this example! It does **not** accept two parameters. Instead, it's one single parameter - an **object** which then just is destructured internally.

The function would still be called like this:

1. storeOrder({id: 5, currency: 'USD', amount: 15.99}); // one argument / value!

**5.Spread operator**.->to combine two arrays

Const hobbies=[“sports”,”movies”]

Const cars=[“bmw”,”tata punch ev”]

Const combinedarray=[…hobbies,..cars]

It is used to objects also

Const users={  
name:”rakesh”,

age:”24”

};

Const extendeduser={

Isadmin:True,

…users

};

You will get details of user.

* **For loop in javaScript**..
* **We use of keyword in for loop**
* **Prompt () function is used to take input from the user.**

Const hobbies=[“sports”,”cooking”]

For(const hobbie of hobbies){

Console.log(hobbie);

}.

**These are bulitIn functions**

**setTimeout(()=>{**

**},2000);**

**Its take function as parameter.**

**User defined functions also take functions as parameter.**

**Ex:**

**Function grettings(greetFn){**

**greetFn();**

**};**

**Calling function grettings(()=>{**

**Console.log(“HI”);**

**});**

* **React notes**

**More Prop Syntaxes**

Beyond the various ways of setting and extracting props about which you learned in the previous lecture, there are **even more ways of dealing** with props.

But no worries, you'll see all these different features & syntaxes in action throughout the course!

**Passing a Single Prop Object**

If you got data that's already organized as a JavaScript object, you can pass that object as a single prop value instead of splitting it across multiple props.

I.e., instead of

1. <CoreConcept
2. title={CORE\_CONCEPTS[0].title}
3. description={CORE\_CONCEPTS[0].description}
4. image={CORE\_CONCEPTS[0].image} />

or

1. <CoreConcept
2. {...CORE\_CONCEPTS[0]} />

you could also pass a single concept (or any name of your choice) prop to the CoreConcept component:

1. <CoreConcept
2. concept={CORE\_CONCEPTS[0]} />

In the CoreConcept component, you would then get that one single prop:

1. export default function CoreConcept({ concept }) {
2. // Use concept.title, concept.description etc.
3. // Or destructure the concept object: const { title, description, image } = concept;
4. }

It is entirely up to you which syntax & approach you prefer.

**Grouping Received Props Into a Single Object**

You can also pass multiple props to a component and then, in the component function, group them into a single object via JavaScript's ["Rest Property"](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment#rest_property) syntax.

I.e., if a component is used like this:

1. <CoreConcept
2. title={CORE\_CONCEPTS[0].title}
3. description={CORE\_CONCEPTS[0].description}
4. image={CORE\_CONCEPTS[0].image} />

You could group the received props into a single object like this:

1. export default function CoreConcept({ ...concept }) {
2. // ...concept groups multiple values into a single object
3. // Use concept.title, concept.description etc.
4. // Or destructure the concept object: const { title, description, image } = concept;
5. }

If that syntax is a bit confusing - worry not! You'll also see concrete examples for this syntax (and for why you might want to use it in certain situations) throughout the course!

**Default Prop Values**

Sometimes, you'll build components that may receive an optional prop. For example, a custom Button component may receive a type prop.

So the Button component should be usable either with a type being set:

1. <Button type="submit" caption="My Button" />

Or without it:

1. <Button caption="My Button" />

To make this component work, you might want to set a default value for the type prop - in case it's not passed.

This can easily be achieved since JavaScript supports default values when using object destructuring:

1. export default function Button({ caption, type = "submit" }) {
2. // caption has no default value, type has a default value of "submit"
3. }

**Hooks in React**

Hooks functions are called inside the components only.

Usestate hook you should import from react .