17)understanding Rest & spread operator

In modern js you should be aware of 2 operators which are used heavily- rest and spread. Specifically spread is used quite a bit. Lets say we want to implement a pattern whenever we add new element to array we dnt want to add it to original array but we want to add it to copy of it. This is very common pattern called immutabiity, where we never edit existing existing values but we always replace them with copies plus changes and that is pattern that I will use a lot in this course.

We got couple of techniques for it. One of them is to use slice operator.

const hobbies = ['Sports', 'Cooking'];

const copiedArray = hobbies.slice();

we can pass argumnts to it , to narrow down rage of elements that we want to copy.

It also works for array of objects.

const hobbies = [{name: 'sumeet', age: 26}, {name: 'nitesh', age: '27'}];

const copiedArray = hobbies.slice();

copiedArray.push({name: 'shikha', age: 26});

console.log(hobbies);

If we do this-

const hobbies = ['Sports', 'Cooking'];

const copiedArray = [hobies];

now in copiedArray first element is itself array. So it is not a copy its new array where first element is old array and with that I mean exact same object not copy of that. That is not what we want to do. We can use spread operator-

spread operator is 3 dots that I can add in front of array of object. these … are operator like +,- and they do one thing, **they pull up all elements of array or properties of object and put it in what is around spread operator**. in this case we got [] around … so all elements that are pulled out from existing array are placed in []

const hobbies = ['Sports', 'Cooking'];

const copiedArray = [...hobbies];

console.log(copiedArray);

we will use this a lot to copy existing arrays or objects. In case of objects it will also work in same way.

let person = {

name: 'Sumeet',

hobbies: ['coding', 'movies']

};

let person1 = { ...person, id: 10 };

console.log(person1);

note that it will not deep clone your object or array.

Now lets see rest operator-

const toArray = (arg1,arg2,arg3) => {

return [arg1,arg2,arg3];

};

console.log(toArray(1,2,3));

lets say we want to this function to be flexible, i.e we can apss any number of arguments and it retruns array containg those argument. We can do it like this-

const toArray = (...args) => {

return args;

};

console.log(toArray(1,2,3,4));

rest operator will take all arguments, no matter how many we specify and will bundle them in array for us.so here args is going to be array.

**Note:** Rest parameters have to be at the **last argument**. This is because it collects all remaining/ excess arguments into an array. So having a function definition like this does not make sense and it errors out. :

function abc(a, ...b, c) {

...

return;

}

So rest operator is just like spread operator i.e with **…** dots. It’s the place where you use it that defines how you call it. Are you using to pull properties or elements out of arrays, then it would be spread operator. are you using it to merge multiple arguments into an array and you use it in the arguments list of function. then it’s rest operator. it’s the same operator from syntax perspective, name differs on place where you use it.

In this course we will use spread operator a lot.

18)Destructuring

Now I want to dive into important feature and that is destructuring.

const person = {

name: 'Max',

age:29,

greet() {

console.log('Hi i am '+ this.name);

}

};

const printName = (personData) => {

console.log(personData.name);

};

printName(person);

here we are only interested in name property of incoming object. we can do it like this as shown above.

In function we are getting whole object and lets say we cnt change that. We can then use a syntax or a feature called destructuring where we use curly braces here in argument and then we specifiy the property of incoming object we are interested in. it will be pulled out from incoming object, other properties will be dropped and this incoming property will be used in variable name, which can use. Code-

const person = {

name: 'Max',

age:29,

greet() {

console.log('Hi i am '+ this.name);

}

};

const printName = ({name}) => {

console.log(name);

};

printName(per

we can also pull out more property-

const person = {

name: 'Max',

age:29,

greet() {

console.log('Hi i am '+ this.name);

}

};

const printName = ({name, age}) => {

console.log(name);

console.log(age);

};

printName(person);

so that is syntax that allow us to write a bit more understandable code. Where we are very clear what do we need from incoming object and that then gets stored in local variable. You can also use this syntax outside functions.

const person = {

name: 'Max',

age:29,

greet() {

console.log('Hi i am '+ this.name);

}

};

const {name ,age} = perosn;

this will create 2 new constants which will hold value person.name and person.age. so names that you give in {} have to match with property names of person.

We can also destructure arrays.

const hobbies = ['Sports', 'Cooking'];

const [hobby1, hobby2] = hobbies;

console.log(hobby1);

console.log(hobby2);

in array destruring you can choose any names you want. Because in array elements have no names. They are pulled out by index or position. In objects you pull them out by property name ,so there names need to match.

This will only work when we want to clone a object shallowly. For deep clone then-

let person = {

name: 'Sumeet',

hobbies: ['coding', 'movies']

};

let person1 = { ...person, hobbies: [...person.hobbies] };

in answer to question jost also shared some ways-

<https://www.udemy.com/nodejs-the-complete-guide/learn/v4/questions/5447440>