Objects in JavaScript

Wednesday, 30 August 2023 11:56 PM

→ Objects: An object is a collection of properties. And a property is an association between a key & a value.
A property's value can be a f^n & in that case the property is known as a method.

→ Example of an object, accessing an object & changing its property:

```
2
3  const user = {
4    name: "Roadside Coder",
5    age: 24,
6  };
7
8  user.name = "Piyush Agarwal";
9
10 console.log(user.name);
11
```

→ So, now the o/p will be Piyush Agarwal.

→ To delete any property we can write: delete user.age
→ We can access an object's properties using '.' dot operator or 'bracket notation'. Eg: user.name or user[name].

→ 'delete' does not work with local variables, it only works with object's properties. So here o/p will be '5'.

```
2
3  const func = (function (a) {
4    delete a;
5    return a;
6  })(5);
7
8  console.log(func);
9
```

→ Creating a key with spaces & then accessing its value.

```
3  const user = {
4    name: "Roadside Coder",
5    age: 24,
6    "like this video": true,
7  };
8
9  console.log(user["like this video"]);
10
```

→ We can do so by putting the key inside double quotes & access it using bracket notation.

→ Similarly we can delete this property using 'delete user["like this video"]'.

→ Adding dynamic key:value pair inside objects:

```
3  const property = "firstName";
4  const name = "Piyush Agarwal";
5
```

→ To add a dynamic key, we can simply

```
const user = {
  [property]: name,
};

console.log(user.firstName);
```

put that key inside square brackets.

→ To access that property, we must give the key's name after dot operator. The o/p here will be "Piyush Agarwal".

→ Looping through objects:

To loop through an object we use for-in loop.

```
const user = {
  name: "Roadside Coder",
  age: 24,
  isTotallyAwesome: true,
};

for (key in user) {
  console.log(key);
}
```

(This will print the keys)

```
const user = {
  name: "Roadside Coder",
  age: 24,
  isTotallyAwesome: true,
};

for (key in user) {
  console.log(user[key]);
}
```

(This will print the values)

→ Guess the o/p:

```
const obj = {
  a: "one",
  b: "two",
  a: "three",
};

console.log(obj);
```

```
script.js:10
{a: 'three', b: 'two'}
  a: "three"
  b: "two"
  [[Prototype]]: Object
```

→ So, when keys are of same names, the first key's value is replaced by the second key's value but the position of the key remains the same.

→ Create a fn that multiplies all the numeric properties of an object by 2.

```
let nums = {
  a: 100,
  b: 200,
  title: "My nums",
};

multiplyByTwo(nums);

function multiplyByTwo(obj) {
  for (key in obj) {
    if (typeof obj[key] === "number") {
      obj[key] *= 2;
    }
  }
}

console.log(nums);
```

→ Guess the output:

```
const a = {};
const b = { key: "b" };
const c = { key: "c" };
```

The o/p is 456.

```
a[b] = 123;
a[c] = 456;

console.log(a[b]);

```

→ Why is the o/p 456?,

→ this is because if we consDle `a' we'll get:

```
                                script.js:12
▼ {[object Object]: 456} i
    [object Object]: 456
  ▶ [[Prototype]]: Object
>
```

→ So, when we give the key as an object it is stored as ["object object"] & since both `a[b]' & `a[c]' are stored as ["object object"], therefore the last value remains & the o/p is 456.

```

const a = {};
const b = { key: "b" };
const c = { key: "c" };

a["[object Object]"] = 123;
a["[object Object]"] = 456;

// console.log(a[b]);
console.log(a);
```

→ Keys which are object are basically stored like this.

→ JSON.stringify VS JSON.parse:

JSON is JavaScript object notation.

```
const user = {
  name: "Piyush",
  age: 24,
};

console.log(JSON.stringify(user));

```

```
{"name":"Piyush","age" script.js:9
:24}
>
```

(o/p of above)

```
const user = {
  name: "Piyush",
  age: 24,
};

const strObj = JSON.stringify(user);

console.log(JSON.parse(strObj));

```

→ To convert a JS object to a string, we use JSON.stringify

→ To convert a string back to a JS object, we use JSON.parse.

→ The most common use case of JSON.stringify & JSON.parse is for storing objects in localStorage. Because we can't store an object directly into localStorage. So we store it as a string & then whenever we need to use it, we parse it back to an object

→ To store an object in localStorage:

localStorage.setItem("key", JSON.stringify(obj));

→ To retrieve an object from local storage & use it:

const objr = JSON.parse(localStorage.getItem("key")).

→ To remove something from localStorage:

localStorage.removeItem("key").

→ Spread operator:

Spread operator spreads the array or the properties of an object.

```
4  console.log([..."lydia"]);
5
```

→ But what will the o/p of above?

→ Here spread operator will spread all of the string's characters inside an array.

→ So o/p is:

```
▶ (5) ['L', 'y', 'd', 'i', 'a']                    script.js:4
>
```

→ When we use spread operator like in the code below, it adds the properties of the object to another object →

```
4  const user = { name: "Lydia", age: 21 };
5  const admin = { admin: true, ...user };
6
7  console.log(admin);
8
```

```
▼ {admin: true, name: 'Lydia', age: 21}          script.js:7
    admin: true
    age: 21
    name: "Lydia"
  ▶ [[Prototype]]: Object
>
```

→ o/p based question:

```
4  const settings = {
5    username: "Piyush",
6    level: 19,
7    health: 90,
8  };
9
10 const data = JSON.stringify(settings, ["level", "health"]);
11 console.log(data);
12
```

```
{"level":19,"health":90}                         script.js:11
>
```

(o/p for the code)

→ So, here, what happens is that JS just stringifies the properties with the keys mentioned in the array & ignores the other properties.

→ Output based question:

```
4  const shape = {
5    radius: 10,
6    diameter() {
7      return this.radius * 2;
8    (property) perimeter: () => number
9    perimeter: () => 2 * Math.PI * this.radius,
10 };
11
12 console.log(shape.diameter());
13 console.log(shape.perimeter());
14
```

```
20                                               script.js:12
NaN                                              script.js:13
>
```

(o/p for the code)

→ Here, diameter() is a normal fn in which 'this' points to the local object, but

perimeter() is an arrow fn where `this` points to the global window object.
Thus o/p is : 20 NaN (NaN is not a number).

→ Object destructuring :

This means taking out the specific properties of an object.

Eg:

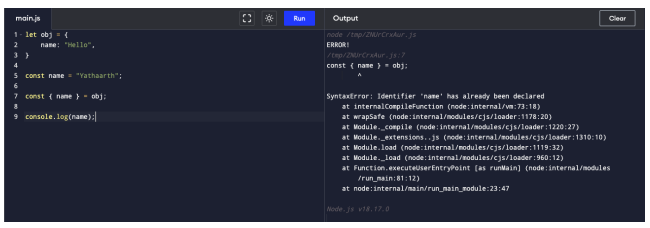
```
let user = {
  name: "Piyush",
  age: 24,
};

const { name } = user;

console.log(name);
```

→ So o/p will be "Piyush".

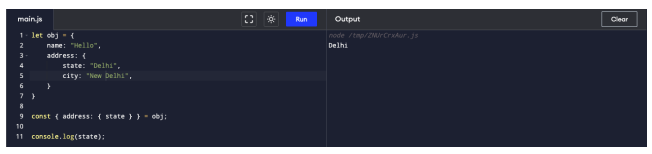
Note: If we destructure `name` but if we have another variable named `name` then it will throw us an error if try to console.log(name);



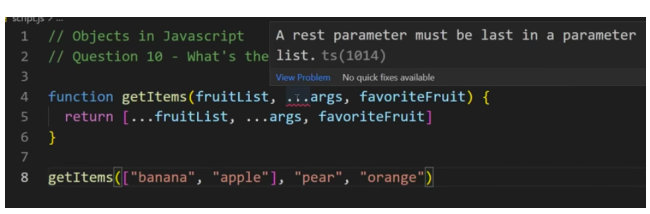
→ So what we can do is we can assign another name to the destructured property.

→ So, now `myName` has access to the destructured value of name property & we can easily access the other `name` variable without getting an error.

→ Nested object destructuring can be done by :



→ Output :



→ Throws an error, since a rest parameter must be the last parameter.

→ But spread operators can be used in between.

```
function getItems(fruitList, favoriteFruit, ...args) {
  return [...fruitList, ...args, favoriteFruit];
}

console.log(getItems(["banana", "apple"], "pear", "orange"));
```

→ So, it outputs this

because [banana, apple] is stored in fruitlist and pear is stored in "favoriteFruit" & rest of the arguments (here, "orange") in `...args`.

→ Now, while returning the array we spread `fruitlist` & `args` into individual items & concatenate them in an array (according to ES6 syntax).

→ Object Referencing:

```
4  let c = { greeting: "Hey!" };
5  let d;
6
7  d = c;
8  c.greeting = "Hello";
9  console.log(d.greeting);
10
```

```
Hello                                    script.js:9
>
```

o/p:

→ So, here we are basically putting a reference (copy of) of `c` inside `d`. So if `d` or `c` change their properties then it affects both of them.

→ So, when we assign one object to another, we are not copying the values of one object to another, instead we provide a reference to that object.

→ Output:

```
4  console.log({ a: 1 } == { a: 1 });
5  console.log({ a: 1 } === { a: 1 });
6
```

```
false                                    script.js:4
false                                    script.js:5
>
```

↳ o/p

→ Here o/p is false because each of the object occupies different memory space. No matter we check strictly or not o/p will be same.

→ Output:

```
4  let person = { name: "Lydia" };
5  const members = [person];
6  person = null;
7
8  console.log(members);
9
```

```
▼ [{…}]                                  script.js:8
  ▼ 0:
      name: "Lydia"
    ▶ [[Prototype]]: Object
    length: 1
  ▶ [[Prototype]]: Array(0)
>
```

o/p:

→ Here, `members` should have been null, due to object referencing but that's not the case because, since we've done `members=[person]`, so now `person` is stored in `members[0]`.

→ We've would've gotten this:
if we would've done "person.name=null"

```
▼ [{…}]                                  script.js:8
  ▶ 0: {name: null}
    length: 1
  ▶ [[Prototype]]: Array(0)
>
```

→ Here what has happened is that we've copied the reference of the object to zeroth index of `members` and since previously we just changed the value of the variable & not the first element of array so it didn't work.

→ Output:

```
4  const value = { number: 10 };
5
6  const multiply = (x = { ...value }) => {
7    console.log((x.number *= 2));
8  };
9
10 multiply(); // 20
11 multiply(); // 20
12 multiply(value); // 20
13 multiply(value); // 40
14
```

→ Here for the first call, value of 'value' is taken from its lexical environment i.e its parent which is global scope,
→ the spread operator will clone the object (which will take its default value from the global scope) and will print 20. For the second invokation also the same thing will happen.
→ Now in the 3rd invokation, since we are passing 'value' as argument, so 'x' will have a reference of 'value'.
→ so in 4th invokation, we get 40, as previous invokation has modified the number since 'x' had a reference to that object.

→ Output:

```
4  function changeAgeAndReference(person) {
5    person.age = 25;
6    person = {
7      name: "John",
8      age: 50,
9    };
10
11   return person;
12 }
13
14 const personObj1 = {
15   name: "Alex",
16   age: 30,
17 };
18
19 const personObj2 = changeAgeAndReference(personObj1);
20
21 console.log(personObj1); // -> ?
22 console.log(personObj2); // -> ?
```

```
▶ {name: 'Alex', age: 25}        script.js:21
▶ {name: 'John', age: 50}        script.js:22
>
```

o/p:

→ Since we are passing 'personObj1' as argument, so 'person' gets a reference to that object & in line 5, changes 'age' property's reference to '25', thus, also changing personObj1.
→ In lines 6 - 9, we are assigning an object to a variable, so that doesn't change personObj1, and therefore we return this variable & hence get this object as o/p in line 22.

→ Shallow copy & Deep copy:

Shallow copy basically means when we copy an object to another object but that particular object has got reference to some of the properties of the original object.

So when 1 object holds reference to another object's some of the properties, it is shallow copy.

Deep copy is when we clone an object completely to another object.

So, in deep copy we don't have any references to the original object.

→ How to deep copy / clone an object ?

ans.

```
4  let user = {
5    name: "Roadside Coder",
6    age: 24,
7  };
8
9  const objClone = Object.assign({}, user);
10 objClone.name = "Piyush";
11 console.log(user, objClone);
12
```

→ Using Object.assign ( )

```
                                        script.js:12
▶ {name: 'Roadside Coder', age: 24}
▶ {name: 'Piyush', age: 24}
>
```

o/p for all way of deep copying / cloning an object.

```
4  let user = {
5    name: "Roadside Coder",
6    age: 24,
7  };
8
9  // const objClone = Object.assign({}, user);
10 const objClone = JSON.parse(JSON.stringify(user));
11 objClone.name = "Piyush";
12
13 console.log(user, objClone);
14
```

→ Using JSON.stringify & JSON.parse together

```
4  let user = {
5    name: "Roadside Coder",
6    age: 24,
7  };
8
9  // const objClone = Object.assign({}, user);
10 // const objClone = JSON.parse(JSON.stringify(user));
11 const objClone = { ...user };
12 objClone.name = "Piyush";
13
14 console.log(user, objClone);
15
```

→ Using spread operator.

Note: Above 3 methods will still store nested properties by creating a copy of their reference.

Note: shallow copy is simple: just assign original object to new object.
eg: let newObj = obj;
Now, 'newObj' will store copy of reference of all properties of obj.