ES6

The difference among const,let and var

The const one can not be redifined, so we have to give it a value when it is defined, and the let is a true variable, because it can be changed, so we can choose not give it a value once it is difined.
Lastly,there is a var left, as gor var, it is special, because you could redifined it in it's scope. So what's the meaning of it? When we defined a var as a global variable like so:

```
let a = 12;
let a = 'Hello'; //It will be a fatal error
var b = 12;
var b = 'Hello'; //It can be passed
```

Template String

Format:

```
const templateStr = `String ${object / expression} String`;
```

And also we could add a function reference before the template string and the this function will have the access to the template string, then we could manipulate this template string:

```
function person() {
  this.name = 'griffin';
  this.age = 22;
  this.sex = 'boy';
}
var person = new person();
const templateStr = template`Hello there! My name is ${person.name}, I am
${person.age} and I am a ${person.sex}`;
// console.log(templateStr);
function template(text, ...vars) {
  const textGet =
    text /
    vars
      .map((item) => {
        return item;
      })
      .join('/');
  console.log(textGet);
}
```

As we saw, afer we put the reference before the template string, this function would get content of the template string, the first parameter point to the text content of the template string, and it will return a array of all the text content of the template string if you use text, then you will get this result:

```
Hello there! My name is /, I am / and I am a /
```

if you use var you will get this result:

```
griffin/22/boy
```

Arrow fuction

The arrow function is a shorthand for the annoymous function, with this function we could build a function more easier. The basic construct:

```
const arrowFunction=([parameter1],parameter2)=>[{function Body}]
```

When there is only one parameter in it, the parentheses can be ignored, and when there is only one statement the curly braces can be ignored too. And not only we could build a function, we could define an object to , but at this time we should have the parentheses:

```
const object=({object body})
```

Keyword "this" in arrow function: In the regular function, "this" refers the parent, so it can be considered as the left of the dot,but the arrow function will always refers to current surrounding scope,what's the meaning of it, let input this example:

```
const object1 = {
  firstName: 'Rongxin',
  lastName: 'Yang',
  sayName: function () {
    console.log(this);
    console.log(`Hello there! My name is ${this.firstName} ${this.lastName}`);
  },
};

const object2 = {
  firstName: 'Rongxin',
  lastName: 'Yang',
  sayName: () => {
    console.log(this);
    console.log(`Hello there! My name is ${this.firstName} ${this.lastName}`);
```

```
  },
};
```

The the sayName in object1, will take output the name correctly, but for the object2 the answer will be undefined. Because in object2 the current surrounding scope to sayName is window, so "this" refers to the window, if we just get a little change like this:

```
const object1 = {
  firstName: 'Rongxin',
  lastName: 'Yang',
  sayName: function () {
    console.log(this);
    setTimeout(() => {
      console.log(this);
    });
  },
};

object1.sayName();
```

We found at this time the second this also refers to the object1, cause now the surrounding scope is regular function which is the method of object1, but when we change the regular also to a arrow function the surrounding scope will be the global scope again, so we will get the result that two "this" refer to the window:

```
const object1 = {
  firstName: 'Rongxin',
  lastName: 'Yang',
  sayName: () => {
    console.log(this);
    setTimeout(() => {
      console.log(this);
    });
  },
};
```

Both of them refer to the window

```
Note:The hoisted is not permited in arrow function, so before you invoke it you
have to initialize it
```

Destructuring

For destructuring we have an example:

```
const frineds = ['john', 'peter', 'bob', 'anna', 'kelly'];
const [p1, p2, p3] = friends;
console.log(p1);
```

We well get :john;
And also we could change the values in destructure way, like this:

```
let first = 'john';
let second = 'tom';
[second, first] = [first, second];
console.log(second);
```

We will also get the john
We could destructure the arrray, and in fact we could also destructure the object, but it just has a little differences:

```
const bob = {
  first: 'bob',
  last: 'sanders',
  city: 'chicago',
  siblings: {
    sister: 'jahe',
  },
};
const {
  first:firstName;
  last,
  city,
  siblings:{sister:favoriateSbling}
}=bob;
console.log(firstName,last,city,favoriateSbling);
```

We will get the result:

```
bob,sanders,chicago,jane
```

So we found that when we are destructuring an object we should give have the same values to meet the elements in the object, not like the random values in array, and when we are destructuring the object in this object we should use the curly braces;

New String methods(Be careful, they are all case sensitive)

startsWith(String,[from]):

Use this method we could determine if a string starts with specific string, and if we set the second parameter we could change the start point.

endsWith(String,[to]):

Use this method we could determine if a string ends with specific string, and if we set the second parameter we could change the end point.

includes(String):

With this method we could determine if a string contains a specific string.

repeat(String,times):

With this method we could make a string repeat the specific times and it will return this new string.

for of:

Use this function we could also iterate through the array:

```
const fruits = ['apple', 'orange', 'banana', 'peach'];
for (const fruit of fruits) {
  if (fruit === 'orange') {
    continue;
  }
}
```

So unlike the forEach method we could use break or continue to end and skip this iteration.

Spread operator ...

This operator allows us to iterate through an array/String/object and spread/expend them individually inside reciver. And it is just split it into single items and copy them, and there is an example:

```
//for String:
const udemy = 'udemy';
const letters = [...udemy];
//result:['u','d','e','m','y']
const fruits = ['apple', 'orange', 'banana', 'peach'];
const people = ['tom', 'john', 'lucy'];

const sum = [...fruits, ...people];
//result:['apple', 'orange', 'banana', 'peach','tom', 'john', 'lucy'];
```

And there is a difference between spread operator and just assigh directly:

```
const people = ['tom', 'john', 'lucy'];
const friends = people;
/*At this time, if we change the value in friends , the people will also be
changed, caus now they get the same reference*/
const friends = [...people];
//Now the friends is just a copy of people, so its change won't affect the people
```

If we use the spread operator to operate the destructuring, it will be an another result:

```
//for array:
const fruit = ['apple', 'orange', 'banana', 'peach'];
const [first, second, ...fruits] = fruit;
console.log(frist, fruit);
//result:apple ['banana', 'peach']

//for object:
const person = { name: 'rongxin', lastName: 'yang', job: 'developer' };
const { job } = person;
console.log(job);
//result:developer
const { name, ...rest } = person;
console.log(name, rest);
//result:rongxin {lastName: 'yang', job: 'developer'}
//Note:The rest only available at the end of the object
```

Because the spread could gather all the elements, so we could put it in the parameter to gather the arguments:

```
const getAverage = (...scores) => {
  let total = 0;
  for (const score of scores) {
    total += score;
  }
  console.log(`Average score is ${total / scores.length}`);
};
getAverage(78, 90, 56, 43);
//result:66.75
```

Array.of and Array.from

Array.of:

From the above we know the way how to build a dynamic method with spread operator And the array could also be built dynamically with Array.of:

```
const friends = Array.of('rongxin', 22, true);
console.log(friends);
//result:['rongxin',22,true]
```

So all the parameters will be sent to the array when use Array.of.

Array.from:

Array.from: could be used to transfer the array-like into an array, include the string,nodelist and set.
It's very useful, because we know so many methods can only be used by arrray like forEach,redce,map etc, but querySelector will return an nodeList, so it can not use these methods directly, but with Array.from the situation will change.

```
function countTotal() {
  let total = Array.from(arguments).reduce(
    (total, currentNum) => (total += currentNum),
    0
  );
  console.log(total);
}
countTotal(67, 89, 54, 100);
//result:310
```

And an the Array.from could also have a parameter and a annoymous function like this:

```
const text = Array.from(document.querySelectorAll('p'), (item) => {
  return `<span>${item.textContent}</span>`;
}).join(' ');
```

find,findIndex,every and some

find:

This method is uesd to find the object in an array that matches condition:

```
const people = [
  { name: 'anna', age: 22 },
  { name: 'rongxin', age: 22 },
  { name: 'TXT', age: 21 },
];
const person = people.find((person) => person.name === 'TXT');
console.log(person);
//result:{name: 'TXT', age: 21}
```

findIndex:

Differ from the above method, this method is uesd to find the the index of the object in an array that matches condition:

```
const people = [
  { name: 'anna', age: 22 },
  { name: 'rongxin', age: 22 },
  { name: 'TXT', age: 21 },
];
const person = people.findIndex((person) => person.age === 21);
console.log(person);
//result:2
```

every and some:

Both of these methods are returning a boolean value, they are all used to determine if this array contains the specific object meeet the condition. The difference is every need all the values meet the condition, but some is finding even one exists.

for in:

We mentioned that the for of is used to iterate over the array, and there is an for in to iterate all the properties of the object

Object.keys(object),Object.values(object) and Object.entries(object)

These three methods are all covert the object to an array, the keys just convert the object's property names to an array, so th values convert the property values and the entries conver the both, and all the objects will be an single array.

The set

Stores a collection of unique values of any type.

Methods:

new set([array]):create a new set
add(value):add a value to the set
delete(value):delete a value from the set
clear():clear the set
has(value):return true if the set contains the specific object

Modules

Use module we could seperate our data and functions into separate positions so that we could easily to manage the code.
To use the module, we should add a type value 'module' to script

```
<script type="module" src="test.js"></script>
```

Next step is to create a module:
1.create a folder
2.create a js file in that folder
3.use export keyword to export the specific data you want

```
export const name = 'Rongxin Yang';
```

Then we need use keyword import and from to import it

```
import { name } from './data/data.js';
console.log(name);
```

And you would get the result:

```
Rongxin Yang
```

The above has described how to export the variables, and we could also export functions, but that has a little differences:
We still need a js file and in this file we could "use export default functionName"(after this function):

```
const newFunction = (parameter) => {
  callback;
};
export default newFunction;
//Or you could do in this way:
export default [function](parameter) => {
  callback;
};
```

Then we could import that:

```
import functionName from '../path';
/*Note:Unlike the variables above, this place, the function name can be anything
we want, we could set a name does not match initial function name*/
```