educative

What's new in ES2020

We'll cover the following

- What’s coming in ES2020
- BigInt
- Dynamic Import
- Optional Chaining
- Promise.allSettled
- Nullish Coalescing
- String.protype.matchAll
- Module Namespace Exports
- import.meta
- globalThis

What’s coming in ES2020

The latest version of ECMAScript, ES2020, includes many new interesting changes and we are going to cover them in this chapter.

Not all browsers currently support these features so, I recommend you use the latest version of Chrome or Firefox to test the code examples. Otherwise, if you want to use them in your project, be sure to install a compiler like **Babel**, which at their latest version 7.8 already supports ES2020 by default so you don't need to use any plugin.

BigInt

The support for `BigInt` means that we will be able store much larger integers in our `JavaScript`. The current max is 2^53 and you can get it by using `Number.MAX_SAFE_INTEGER`. That does not mean that you cannot store larger integer, but `JavaScript` does not handle them well, let's look at an example:

```
let num = Number.MAX_SAFE_INTEGER
console.log(num);
// 9007199254740991
console.log(num + 1);
// 9007199254740992
console.log(num + 2);
// 9007199254740992
console.log(num + 3);
// 9007199254740994
console.log(num + 4);
// 9007199254740996
```

As you can see, once we hit the max that `JavaScript` can handle, things stop working as we think they should.

In order to start using `BigInt`, we can use the constructor `BigInt` or we can simply append an `n` at the end of your long integer and everything will continue working smoothly as it should.

```
// IF YOU ARE GETTING ERRORS, RUN THIS CODE IN THE CONSOLE OF YOUR BROW

// let bigInt = BigInt(99999999999999999);
let bigInt = 99999999999999999n;
console.log(bigInt + 1n);
// 100000000000000000n
```

As you can see I did not add `1` but I added `1n`, that's because you can't add one integer to a `BigInt` if you want to do that you first need to parse that `BigInt` using `parseInt(bigInt,10)`.

Dynamic Import

This will allow you to dynamically import your modules when you need them. Look at the following example:

```
if(condition1 && condition2){
    const module = await import('./path/to/module.js');
    module.doSomething();
}
```

```
4  }
```

If you don't need a module, you don't have to import it and you can just do that when/if it's needed, using `async/await`.

Optional Chaining

Let's take these simple `Object` that represent our Users.

```
1  const user1 = {
2   name: 'Alberto',
3   age: 27,
4   work: {
5    title: 'software developer',
6    location: 'Vietnam'
7   }
8  }
9
10 const user2 = {
11  name: 'Tom',
12  age: 27
13 }
```

Let's say we want to display the job title of our user. As we can see, `work` is an optional property of our `Object` so we would have to write something like this:

```
1  const user1 = {
```

```
 2   name: 'Alberto',
 3   age: 27,
 4   work: {
 5    title: 'software developer',
 6    location: 'Vietnam'
 7   }
 8  }
 9
10  const user2 = {
11   name: 'Tom',
12   age: 27
13  }
14
15  if (user1.work){
16   console.log(user1.work.title);
17  }
```

or using a ternary operator:

```
1  const jobTitle = user.work ? user.work.title : ''
```

Before we access the property `title` of `work` we need to check that the user actually has a `work`.

When we are dealing with simple objects it's not such a big deal but when the data we are trying to access is deeply nested, it can be a problem.

This is where the Optional Chaining `?.` operator comes to the rescue. This is how we would rewrite our code with this new operator:

```
const user1 = {
 name: 'Alberto',
 age: 27,
 work: {
  title: 'software developer',
  location: 'Vietnam'
 }
}

const user2 = {
 name: 'Tom',
 age: 27
}

// IF YOU ARE GETTING ERRORS, RUN THIS CODE IN THE CONSOLE OF YOUR BROW
if (user1.work){
 console.log(user.work?.title);
}
```

Done! More concise and readable.

You can read the code above as `does the user have a work property? if yes, access the title property inside of it`

```
const user1 = {
```

```
  name: 'Alberto',
  age: 27,
  work: {
   title: 'software developer',
   location: 'Vietnam'
  }
}

const user2 = {
 name: 'Tom',
 age: 27
}

// IF YOU ARE GETTING ERRORS, RUN THIS CODE IN THE CONSOLE OF YOUR BROW
const user1JobTitle = user1.work?.title;
console.log(user1JobTitle);
// software developer

const user2JobTtile = user2.work?.title;
console.log(user2JobTitle);
// undefined
```

As soon as we hit a property that is not available on the `Object`, the operator will return `undefined`

Imagine dealing with a deeply nested object with optional properties such as these two users and their school records.

```
const elon = {
 name: 'Elon Musk',
```

```
 3   education: {
 4    primary_school: { /*  primary school stuff */ },
 5    middle_school: { /* middle school stuff */ },
 6    high_school: {/* high school stuff here */},
 7    university: {
 8     name: 'University of Pennsylvania',
 9     graduation: {
10      year: 1995
11     }
12    }
13   }
14  }
15
16  const mark = {
17   name: 'Mark Zuckerberg',
18   education: {
19    primary_school: { /*  primary school stuff */ },
20    middle_school: { /* middle school stuff */ },
21    high_school: {/* high school stuff here */},
22    university: {
23     name: 'Harvard University',
24    }
25   }
26  }
```

Not all of our Users have studied in University so that property is going to be optional and the same goes for the graduation as some have dropped out and didn't finish the study.

Now imagine wanting to access the graduation year of our two users:

```
1  let graduationYear;
2  if(
```

```
 user.education.university && user.education.university.graduation && u
  graduationYear = user.education.university.graduation.year;
}
```

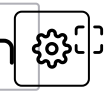
And with the Optional Chaining operator:

```
const elon = {
 name: 'Elon Musk',
 education: {
  primary_school: { /*  primary school stuff */ },
  middle_school: { /* middle school stuff */ },
  high_school: {/* high school stuff here */},
  university: {
   name: 'University of Pennsylvania',
   graduation: {
    year: 1995
   }
  }
 }
}

const mark = {
 name: 'Mark Zuckerberg',
 education: {
  primary_school: { /*  primary school stuff */ },
  middle_school: { /* middle school stuff */ },
  high_school: {/* high school stuff here */},
  university: {
   name: 'Harvard University',
  }
 }
}

// IF YOU ARE GETTING ERRORS, RUN THIS CODE IN THE CONSOLE OF YOUR BROW
```



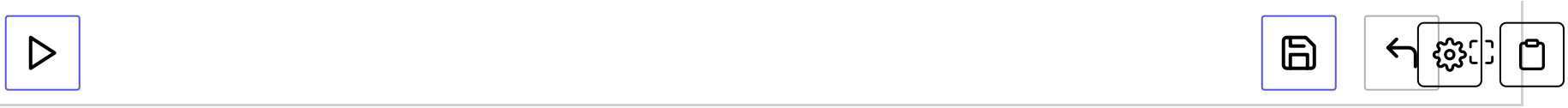




Promise.allSettled

ES6 added `Promise.all` that let us await until all the promises given to it are passed. `Promise.allSettled` goes one step further and let us await until all the promises are completed, returning us an `Array` of objects describing the outcome of each of them.

This means that we will be able to tell easily which one of our promises is failing:

```
1
2  // IF YOU ARE GETTING ERRORS, RUN THIS CODE IN THE CONSOLE OF YOUR BROW
3
4  const arrayOfPromises = [
5      new Promise((res, rej) => setTimeout(res, 1000)),
6      new Promise((res, rej) => setTimeout(rej, 1000)),
7      new Promise((res, rej) => setTimeout(res, 1000)),
8  ]
9
10 Promise.allSettled(arrayOfPromises).then(data => console.log(data));
11
12 // [
13 //   Object { status: "fulfilled", value: undefined},
14 //   Object { status: "rejected", reason: undefined},
15 //   Object { status: "fulfilled", value: undefined},
16 // ]
```



As you can clearly see, the second promise rejected and `Promise.allSettled` returned us the status of each of them.

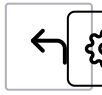
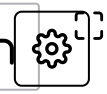
Nullish Coalescing

A falsey and a nullish value (`null` or `undefined`), may be similar sometimes, but they are two different values and this new operator allows us to check specifically for nullish values.

Look at this example for a refresher on falsey values:

```
// we use the !! to convert the value to boolean
const str = "";
console.log(!!str);
// false
const num = 0;
console.log(!!num);
// false
const n = null;
console.log(!!n);
// false
const u = undefined;
console.log(!!u);
// false
```



As you can see, all of these values are falsey. Sometimes we want to distinguish between an empty string or an undefined and this is where the Nullish Coalescing operator (`??`) will come in handy.

The Nullish Coalescing operator (`??`) returns the right-hand side operand when the left-hand side is nullish.

```
// IF YOU ARE GETTING ERRORS, RUN THIS CODE IN THE CONSOLE OF YOUR BROW

const x = '' ?? 'empty string';
console.log(x);
// ''
const num = 0 ?? 'zero';
console.log(num);
// 0
const n = null ?? "it's null";
console.log(n);
// "it's null"
const u = undefined ?? "it's undefined";
console.log(u);
// "it's undefined"
```



As you can see, in the first two examples the value was not nullish but falsey, so the operator did **not** return the value on the right-hand side.

String.protype.matchAll

The `matchAll()` method is a new string method that returns an iterator of all the results matching a string against a specified regEx.

```
// regex that matches any character in the range from 'a' to 'd'
const regEx = /[a-d]/g;
const str = "Lorem ipsum dolor sit amet"
const regExIterator = str.matchAll(regEx);

console.log(Array.from(regExIterator));
// [
//     ["d", index: 12, input: "Lorem ipsum dolor sit amet", groups: un
//     ["a", index: 22, input: "Lorem ipsum dolor sit amet", groups: un
// ]
```

As you can see, we called the `matchAll` method against our string and since our regEx matches every character in the range from ‘a’ to ‘d’, we got two results from our example string.

Module Namespace Exports

We could already do something like this:

```
1  import·*·as·stuff·from·'./test.mjs';
```

But now we can also do the same for **exports**:

```
1  export * as stuff from './test.mjs';
```

which would be the same as doing:

```
1  export { stuff }
```

It's not a game-changer feature, but it adds a better symmetry between import and export statements and their syntax.

import.meta

The `import.meta` object exposes information about a module, such as its URL.

```
1  <script type="module" src="test.js"></script>
2  console.log(import.meta); // { url: "file:///home/user/test.js" }
```

The URL contained in the object can either be the path of the document base for inline scrips or the URL where it was obtained for external scripts.

globalThis

Up until ES2020, there was no standardized global `this` in `JavaScript` meaning that it could either be the `window` when accessed in browsers, `global` for Node environments, and `self` for web workers.

You would have to manually detect the environment at runtime and bind the appropriate object to your global `this`.

Now, in ES202 you can use the `globalThis` which always refers to the `global` object. In Browsers, due to the fact, the `global` object is not directly accessible, the `globalThis` will be a reference to a `Proxy` of it.

Using the new `globalThis` you won't have to worry anymore about the environment in which your application is running in order to access this global value.

Report an Issue

Ask a Question (https://discuss.educative.io/tag/whats-new-in-es2020__from-es2016-to-es2020__the-complete-guide-to-modern-javascript)