It’s an exciting time to be a JavaScript programmer. Web technologies are moving forward at a faster rate, and browser vendors are no longer shy to implement new and innovative features right away. This shift in development means programmers need to continually update their skill set to stay competitive in their role.

In this article, we’ll look at six ES2020 and ES2021 features that have recently been implemented by modern browsers and see how they help JavaScript developers write less error-prone and more efficient code.

**BigInt**

When dealing with large integers in JavaScript, we often have to use third-party libraries because the Number type is incapable of safely representing integer values larger than  253.

Consider the following example:

console.log(9999999999999999); // => 10000000000000000

In this code, 9999999999999999 is rounded to 10000000000000000 because that’s larger than the largest integer supported by the Number type. If you’re not careful, such rounding can compromise your program’s security.

Here’s another example:

// notice the last digit

9800000000000007 === 9800000000000008; // => true

Fortunately, ECMAScript has recently introduced the BigInt data type that provides a straightforward way to represent integers larger than the range supported by Number.

A BigInt can be created by adding n to the of an integer.

Compare:

console.log(9800000000000007n); // => 9800000000000007n

console.log(9800000000000007); // => 9800000000000008

It’s also possible to use a constructor:

BigInt('9800000000000007'); // => 9800000000000007n

Now, you can perform arithmetic operations on large integers in standard JavaScript without having to use a workaround:

9999999999999999 \* 3; // => 30000000000000000

// with BigInt, integer overflow won’t be an issue

9999999999999999n \* 3n; // => 29999999999999997n

It’s important to understand that Number and BigInt are two distinct data types, and you cannot compare them with the strict equality operator:

5n === 5; // => false

typeof 5n; // => bigint

typeof 5; // => number

However, you can still use the equality operator because it implicitly converts the operands to the same type before comparing:

5n == 5; // => true

You can perform arithmetic operations on BigInts just like Numbers:

50n + 30n; // => 80n

50n - 30n; // => 20n

50n \* 20n; // => 1000n

50n / 5n; // => 10n

56n % 10n; // => 6n

50n \*\* 4n; // => 6250000n

Increment, decrement, and unary negation operators also work as expected. But, the unary plus (+) operator is an exception and applying it to a BigInt will cause a TypeError:

let x = 50n;

++x; // => 51n

--x; // => 50n

-50n; // => -50n

+50n; // => TypeError: Cannot convert a BigInt value to a number

**Nullish coalescing operator**

ES2020 adds another short-circuiting operator to the JavaScript language: the nullish coalescing (??) operator. This operator differs from the existing short-circuiting operators in that it checks whether its left operand is nullish (null or undefined) rather than falsy.

In other words, ?? returns its right operand only if its left operand is null or undefined:

null ?? 2; // => 2

undefined ?? 2; // => 2

0 ?? 2; // => 0

false ?? true; // => false

The logical OR (||) operator, on the other hand, returns its right operand if the left one is 0, -0, 0n, false, "" (empty string), null, undefined, or NaN. Compare:

null || 2; // => 2

undefined || 2; // => 2

0 || 2; // => 2

false || true; // => true

?? is especially handy when setting a default value for a property or variable. For example:

function Config(darkMode) {

 this.darkMode = darkMode ?? true;

 // …

}

new Config(null); // => {darkMode: true}

new Config(); // => {darkMode: true}

new Config(false); // => {darkMode: false}

The Config constructor provides a default value for the darkMode property in case the given value is nullish or no value is given.

?? is also useful when working with DOM APIs:

// querySelector() returns null when the element doesn’t exist in the document

const elem = document.querySelector('elem') ?? document.createElement('elem');

Keep in mind that when using ?? with other short-circuiting operators in an expression, you must denote the order of evaluation with parentheses, or else the code throws an error.

Parentheses also help with the readability of the code:

false || (true ?? true); // no error

false || true ?? true; // => SyntaxError

**Promise.any()**

ES2015 introduced the promise object with two methods: Promise.all() and Promise.race(). ES2021 further enhances JavaScript asynchronous capabilities by adding Promise.any(). This new method returns a promise that fulfills when one of the promises in the given iterable fulfills, or rejects if all of the promises reject.

Here’s how it works in action:

const promise = Promise.any([

 Promise.reject('Error'),

 fetch('https://www.google.com/images/branding/googlelogo/2x/googlelogo\_color\_92x30dp.png', {mode: 'no-cors'}).then(() => 'google.com'),

 fetch('https://en.wikipedia.org/static/images/project-logos/enwiki.png', {mode: 'no-cors'}).then(() => 'wikipedia.org'),

 fetch('https://s.w.org/images/home/swag\_col-1.jpg?1', {mode: 'no-cors'}).then(() => 'w.org')

]);

promise.then((fastest) => {

 // the first promise that fulfills

 console.log(fastest);

}).catch((error) => {

 console.log(error);

});

This code executes three fetch requests. As soon as one of the promises is fulfilled, it returns a promise that fulfills with the value from that promise. Promise.any() differs from Promise.race() in how it handles rejection. The promise returned by Promise.any() rejects only if all of the promises in the iterable reject:

const promise = Promise.any([

 Promise.reject('Exception1'),

 Promise.reject('Exception2'),

 Promise.reject('Exception3')

]);

promise.then((response) => {

 // ...

}).catch((e) => {

 console.log(e.errors);

});

// logs:

// => ["Exception1", "Exception2", "Exception3"]

Notice how the rejection value of all promises is passed as an array to the catch() method. Alternatively, you can use async and await to handle the result:

(async () => {

 try {

 result = await Promise.any([

 Promise.reject('Exception1'),

 Promise.reject('Exception2'),

 Promise.resolve('Success!')

 ]);

 console.log(result);

 } catch(e) {

 console.log(e);

 }

})();

// logs:

// => Success!

**Promise.allSettled()**

Another useful method that has recently been added to the promise object is Promise.allSettled(). This method, which complements the existing Promise.all() method, is designed to return the result of all promises — whether rejected or fulfilled.

Here’s an example:

const p1 = Promise.resolve('Success');

const p2 = Promise.reject('Exception');

const p3 = Promise.resolve(123);

Promise.allSettled([p1, p2, p3]).then((response) => {

 response.forEach(result => console.log(result.value || result.reason))

});

// logs:

// => Success

// => Error!

// => 123

Notice how the result of all promises is passed as an array to then(). Inside then(), a forEach() method loops over the items of the array. If the left operand of the || operator isn’t undefined, it will be logged to the console. Otherwise, the promise has been rejected, and the right operand will be logged.

By comparison, Promise.all() immediately rejects as soon as one of the promises rejects.

**Optional chaining operator**

The optional chaining operator (?.) allows you to access a nested property without validating each property in the chain.

Consider the following example:

const obj = {};

const nickname = obj?.user?.profile?.nickname;

console.log(nickname); // => undefined

This code attempts to assign the value of a nested property to a constant. But, there’s no such a property in obj. Additionally, user and profile don’t exist. But thanks to the optional chaining operator, the code returns undefined instead of throwing an error.

With the regular chaining operator, you’d get an error:

const obj = {};

const nickname = obj.user.profile.nickname;

console.log(nickname); // => TypeError

The optional chaining operator can also be used when calling an object’s method:

const obj = {};

const value = obj.myMethod?.();

console.log(value); // => undefined

Here, myMethod doesn’t exist in obj; however, since it’s called using the optional chaining operator, the return value is undefined. Again, with the regular chaining operator, you’d get an error.

But what if you want to access a property dynamically? The ?.[] token allows you to reference a variable using the bracket notation.

Here’s how it works:

const obj = {

 user: {

 id: 123

 }

};

const prop = 'nickname';

const nickname = obj?.user?.profile?.[prop];

console.log(nickname); // => undefined

**globalThis**

Although JavaScript was created with the intention of executing complex features in web browsers, it can now run in completely different environments such as servers, smartphones, and even robots. Because each environment has its own object model, you’ll need to use a different syntax to access the global object.

In the browser environment, you may use window, frames, or self. In Web Workers, you may use self. And in Node, you may use global. This discrepancy makes it harder for web developers to write portable JavaScript programs.

globalThis provides a single universal property in all environments to access the global object:

// browser environment

console.log(globalThis); // => Window {...}

// web worker environment

console.log(globalThis); // => DedicatedWorkerGlobalScope {...}

// node environment

console.log(globalThis); // => Object [global] {...}

Previously, developers had to write additional checks to ensure they refer to the correct property. With globalThis, that’s no longer required, and the code will work in both window and non-window contexts. Keep in mind that you may still need to use a polyfill for backward compatibility on older browsers.

**Conclusion**

JavaScript is quickly evolving, and interesting new features are added to the language every so often. In this article, we looked at six new JavaScript features, including BigInt, nullish coalescing operator, Promise.any(), Promise.allSettled(), the optional chaining operator, and globalThis.

BigInt allows representing large integer values. The nullish coalescing operator brings a new short-circuiting operator to JavaScript. Promise.any() and Promise.allSettled() permit further control over asynchronous operations. The optional chaining operator simplifies accessing nested properties. And globalThis provides a single universal property in all environments to access the global object.

To get updated on the latest features, check out the [list of finished proposals](https://github.com/tc39/proposals/blob/master/finished-proposals.md). If you have any questions feel free to ask in the comments, I’m also on [Twitter](https://twitter.com/FarazKelhini).