**11 Amazing New JavaScript Features in ES13 -** [Coding Beauty](https://codingbeauty.medium.com/?source=post_page-----eed7ed2f1497--------------------------------) Jul 23, 2022

Like a lot of other programming languages, JavaScript is constantly evolving. Every year, the language is made more powerful with new capabilities that let developers write more expressive and concise code.

Let’s explore the most recent features added in ECMAScript 2022 (ES13), and see examples of their usage to understand them better.

**1. Class Field Declarations**

Before ES13, class fields could only be declared in the constructor. Unlike in many other languages, we could not declare or define them in the outermost scope of the class.

class Car {  
 constructor() {  
 this.color = 'blue';  
 this.age = 2;  
 }  
}const car = new Car();  
console.log(car.color); // blue  
console.log(car.age); // 2

ES13 removes this limitation. Now we can write code like this:

class Car {  
 color = 'blue';  
 age = 2;  
}const car = new Car();  
console.log(car.color); // blue  
console.log(car.age); // 2

**2. Private Methods and Fields**

Previously, it was not possible to declare private members in a class. A member was traditionally prefixed with an underscore (\_) to indicate that it was meant to be private, but it could still be accessed and modified from outside the class.

class Person {  
 \_firstName = 'Joseph';  
 \_lastName = 'Stevens'; get name() {  
 return `${this.\_firstName} ${this.\_lastName}`;  
 }  
}const person = new Person();  
console.log(person.name); // Joseph Stevens// Members intended to be private can still be accessed  
// from outside the class  
console.log(person.\_firstName); // Joseph  
console.log(person.\_lastName); // Stevens// They can also be modified  
person.\_firstName = 'Robert';  
person.\_lastName = 'Becker';console.log(person.name); // Robert Becker

With ES13, we can now add private fields and members to a class, by prefixing it with a hashtag (#). Trying to access them from outside the class will cause an error:

class Person {  
 #firstName = 'Joseph';  
 #lastName = 'Stevens'; get name() {  
 return `${this.#firstName} ${this.#lastName}`;  
 }  
}const person = new Person();  
console.log(person.name);// SyntaxError: Private field '#firstName' must be  
// declared in an enclosing class  
console.log(person.#firstName);  
console.log(person.#lastName);

Note that the error thrown here is a syntax error, which happens at compile time, so no part of the code runs. The compiler doesn’t expect you to even try to access private fields from outside a class, so it assumes you’re trying to declare one.

**3. await Operator at the Top Level**

In JavaScript, the await operator is used to pause execution until a Promise is settled (fulfilled or rejected).

Previously, we could only use this operator in an async function - a function declared with the async keyword. We could not do so in the global scope.

function setTimeoutAsync(timeout) {  
 return new Promise((resolve) => {  
 setTimeout(() => {  
 resolve();  
 }, timeout);  
 });  
}// SyntaxError: await is only valid in async functions  
await setTimeoutAsync(3000);

With ES13, now we can:

function setTimeoutAsync(timeout) {  
 return new Promise((resolve) => {  
 setTimeout(() => {  
 resolve();  
 }, timeout);  
 });  
}// Waits for timeout - no error thrown  
await setTimeoutAsync(3000);

**4. Static Class Fields and Static Private Methods**

We can now declare static fields and static private methods for a class in ES13. Static methods can access other private/public static members in the class using the this keyword, and instance methods can access them using this.constructor.

class Person {  
 static #count = 0; static getCount() {  
 return this.#count;  
 } constructor() {  
 this.constructor.#incrementCount();  
 } static #incrementCount() {  
 this.#count++;  
 }  
}const person1 = new Person();  
const person2 = new Person();console.log(Person.getCount()); // 2

**5. Class static Block**

ES13 allows the definition of static blocks that will be executed only once, at the creation of the **class**. This is similar to static constructors in other languages with support for object-oriented programming, like C# and Java.

A class can have any number of static {} initialization blocks in its class body. They will be executed, along with any interleaved static field initializers, in the order they are declared. We can use the super property in a static block to access properties of the super class.

class Vehicle {  
 static defaultColor = 'blue';  
}class Car extends Vehicle {  
 static colors = []; static {  
 this.colors.push(super.defaultColor, 'red');  
 } static {  
 this.colors.push('green');  
 }  
}console.log(Car.colors); // [ 'blue', 'red', 'green' ]

**6. Ergonomic Brand Checks for Private Fields**

We can use this new feature to check if an object has a particular private field in it, using the in operator.

class Car {  
 #color; hasColor() {  
 return #color in this;  
 }  
}const car = new Car();  
console.log(car.hasColor()); // true;

The in operator can correctly distinguish private fields with the same names from different classes:

class Car {  
 #color; hasColor() {  
 return #color in this;  
 }  
}class House {  
 #color; hasColor() {  
 return #color in this;  
 }  
}const car = new Car();  
const house = new House();console.log(car.hasColor()); // true;  
console.log(car.hasColor.call(house)); // false  
console.log(house.hasColor()); // true  
console.log(house.hasColor.call(car)); // false

**7. at() Method for Indexing**

We typically use square brackets ([]) in JavaScript to access the Nth element of an array, which is usually a simple process. We just access the N - 1 property of the array.

const arr = ['a', 'b', 'c', 'd'];  
console.log(arr[1]); // b

However, we have to use an index of arr.length - N if we want to access the Nth item from the end of the array with square brackets.

const arr = ['a', 'b', 'c', 'd'];// 1st element from the end  
console.log(arr[arr.length - 1]); // d// 2nd element from the end  
console.log(arr[arr.length - 2]); // c

The new at() method lets us do this more concisely and expressively. To access the Nth element from the end of the array, we simply pass a negative value of -N to at().

const arr = ['a', 'b', 'c', 'd'];// 1st element from the end  
console.log(arr.at(-1)); // d// 2nd element from the end  
console.log(arr.at(-2)); // c

Apart from arrays, strings and TypedArray objects also now have at() methods.

const str = 'Coding Beauty';  
console.log(str.at(-1)); // y  
console.log(str.at(-2)); // tconst typedArray = new Uint8Array([16, 32, 48, 64]);  
console.log(typedArray.at(-1)); // 64  
console.log(typedArray.at(-2)); // 48

**8. RegExp Match Indices**

This new feature allows us to specify that we want the get both the starting and ending indices of the matches of a RegExp object in a given string.

Previously, we could only get the starting index of a regex match in a string.

const str = 'sun and moon';const regex = /and/;const matchObj = regex.exec(str);// [ 'and', index: 4, input: 'sun and moon', groups: undefined ]  
console.log(matchObj);

We can now specify a d regex flag to get the two indices where the match starts and ends.

const str = 'sun and moon';const regex = /and/d;const matchObj = regex.exec(str);/\*\*  
[  
 'and',  
 index: 4,  
 input: 'sun and moon',  
 groups: undefined,  
 indices: [ [ 4, 7 ], groups: undefined ]  
]  
 \*/  
console.log(matchObj);

With the d flag set, the object returned will have an indices property that contains the starting and ending indices.

**9. Object.hasOwn() Method**

In JavaScript, we can use the Object.prototype.hasOwnProperty() method to check if an object has a given property.

class Car {  
 color = 'green';  
 age = 2;  
}const car = new Car();console.log(car.hasOwnProperty('age')); // true  
console.log(car.hasOwnProperty('name')); // false

But there are certain problems with this approach. For one, the Object.prototype.hasOwnProperty() method is not protected - it can be overridden by defining a custom hasOwnProperty() method for a class, which could have completely different behavior from Object.prototype.hasOwnProperty().

class Car {  
 color = 'green';  
 age = 2; // This method does not tell us whether an object of  
 // this class has a given property.  
 hasOwnProperty() {  
 return false;  
 }  
}const car = new Car();console.log(car.hasOwnProperty('age')); // false  
console.log(car.hasOwnProperty('name')); // false

Another issue is that for objects created with a null prototype (using Object.create(null)), trying to call this method on them will cause an error.

const obj = Object.create(null);  
obj.color = 'green';  
obj.age = 2;// TypeError: obj.hasOwnProperty is not a function  
console.log(obj.hasOwnProperty('color'));

One way to solve these issues is to use to call the call() method on the Object.prototype.hasOwnProperty Function property, like this:

const obj = Object.create(null);  
obj.color = 'green';  
obj.age = 2;  
obj.hasOwnProperty = () => false;console.log(Object.prototype.hasOwnProperty.call(obj, 'color')); // true  
console.log(Object.prototype.hasOwnProperty.call(obj, 'name')); // false

This isn’t very convenient. We can write a reusable function to avoid repeating ourselves:

function objHasOwnProp(obj, propertyKey) {  
 return Object.prototype.hasOwnProperty.call(obj, propertyKey);  
}const obj = Object.create(null);  
obj.color = 'green';  
obj.age = 2;  
obj.hasOwnProperty = () => false;console.log(objHasOwnProp(obj, 'color')); // true  
console.log(objHasOwnProp(obj, 'name')); // false

No need for that though, as we can use the new built-in Object.hasOwn() method. Like our reusable function, it takes an object and property as arguments and returns true if the specified property is a direct property of the object. Otherwise, it returns false.

const obj = Object.create(null);  
obj.color = 'green';  
obj.age = 2;  
obj.hasOwnProperty = () => false;console.log(Object.hasOwn(obj, 'color')); // true  
console.log(Object.hasOwn(obj, 'name')); // false

**10. Error Cause**

Error objects now have a cause property for specifying the original error that caused the error about to be thrown. This helps to add additional contextual information to the error and assist the diagnosis of unexpected behavior. We can specify the cause of an error by setting a cause property on an object passed as the second argument to the Error() constructor.

function userAction() {  
 try {  
 apiCallThatCanThrow();  
 } catch (err) {  
 throw new Error('New error message', { cause: err });  
 }  
}try {  
 userAction();  
} catch (err) {  
 console.log(err);  
 console.log(`Cause by: ${err.cause}`);  
}

**11. Array Find from Last**

In JavaScript, we can already use the Array find() method to find an element in an array that passes a specified test condition. Similarly, we can use findIndex() to find the index of such an element. While find() and findIndex() both start searching from the first element of the array, there are instances where it would be preferable to start the search from the last element instead.

There are scenarios where we know that finding from the last element might achieve better performance. For example, here we’re trying to get the item in the array with the value prop equal to y. With find() and findIndex():

const letters = [  
 { value: 'v' },  
 { value: 'w' },  
 { value: 'x' },  
 { value: 'y' },  
 { value: 'z' },  
];const found = letters.find((item) => item.value === 'y');  
const foundIndex = letters.findIndex((item) => item.value === 'y');console.log(found); // { value: 'y' }  
console.log(foundIndex); // 3

This works, but as the target object is closer to the tail of the array, we might be able to make this program run faster if we use the findLast() and findLastIndex() methods to search the array from the end.

const letters = [  
 { value: 'v' },  
 { value: 'w' },  
 { value: 'x' },  
 { value: 'y' },  
 { value: 'z' },  
];const found = letters.findLast((item) => item.value === 'y');  
const foundIndex = letters.findLastIndex((item) => item.value === 'y');console.log(found); // { value: 'y' }  
console.log(foundIndex); // 3

Another use case might require that we specifically search the array from the end to get the correct item. For example, if we want to find the last even number in a list of numbers, find() and findIndex() would produce a wrong result:

const nums = [7, 14, 3, 8, 10, 9];// gives 14, instead of 10  
const lastEven = nums.find((value) => value % 2 === 0);// gives 1, instead of 4  
const lastEvenIndex = nums.findIndex((value) => value % 2 === 0);console.log(lastEven); // 14  
console.log(lastEvenIndex); // 1

We could call the reverse() method on the array to reverse the order of the elements before calling find() and findIndex(). But this approach would cause unnecessary mutation of the array, as reverse() reverses the elements of an array in place. The only way to avoid this mutation would be to make a new copy of the entire array, which could cause performance problems for large arrays.

Also, findIndex() would still not work on the reversed array, as reversing the elements would also mean changing the indexes they had in the original array. To get the original index, we would need to perform an additional calculation, which means writing more code.

const nums = [7, 14, 3, 8, 10, 9];// Copying the entire array with the spread syntax before  
// calling reverse()  
const reversed = [...nums].reverse();// correctly gives 10  
const lastEven = reversed.find((value) => value % 2 === 0);// gives 1, instead of 4  
const reversedIndex = reversed.findIndex((value) => value % 2 === 0);// Need to re-calculate to get original index  
const lastEvenIndex = reversed.length - 1 - reversedIndex;console.log(lastEven); // 10  
console.log(reversedIndex); // 1  
console.log(lastEvenIndex); // 4

It’s in cases like where the findLast() and findLastIndex() methods come in handy.

const nums = [7, 14, 3, 8, 10, 9];const lastEven = nums.findLast((num) => num % 2 === 0);  
const lastEvenIndex = nums.findLastIndex((num) => num % 2 === 0);console.log(lastEven); // 10  
console.log(lastEvenIndex); // 4

This code is shorter and more readable. Most importantly, it produces the correct result.

**Conclusion**

So we’ve seen the newest features ES13 brings to JavaScript. Use them to boost your productivity as a developer and write cleaner code with greater conciseness and clarity.