Session 24

Mohamed Emary

June 27, 2024

1 "use strict"

When JavaScript was first introduced, it was a very forgiving language. It would try to make sense of whatever code you gave it, even if it was poorly written. This could lead to bugs that were hard to track down.

Examples of common coding problems that JavaScript would allow when it was first introduced include:

- Using a variable and assigning a value to it without declaring it first. `x = 5;`
- Duplicating a parameter name. `function sum(x, x) { /* function body */ }`
- Using a reserved word as a variable or function name. `var let = 5;`

In 2009, ECMAScript 5 (ES5) introduced a new feature called "strict mode" that would help developers catch these bugs earlier. Strict mode is a way to use to a restricted variant of JavaScript that would catch common coding problems and throw exceptions.

Using strict mode in cases like the ones above would throw an error, which would help you catch the bugs earlier in the development process.

To enable strict mode, you can add the following line to the top of your JS code:

```
'use strict'; // be sure to include the "quotes"
```

You can also enable strict mode for just a single function by adding the same line at the top of the function.

```
function doSomething() {
  'use strict';
  // This code is in strict mode
}
```

Strict mode is supported in all modern browsers, and it's a good practice to use it in your code.

```
'use strict';

function doSomething() {
```

```
  // This code is in strict mode
}

function doSomethingElse() {
  // This code is also in strict mode
}
```

Classes and modules which have strict mode enabled by default.

2 let and const

ES6 introduced two new ways to declare variables: `let` and `const`.

2.1 let

`let` is similar to `var`, but it has a few key differences:

- Variables declared with `let` are **block-scoped**, while variables declared with `var` are **function-scoped**.
  - This will save memory because the variable will only be available within the block where it was declared, and after the block ends, the variable will be removed from memory, this will free up memory from unnecessary variables.
- Variables declared with `let` are not **hoisted to the top of the block**, while variables declared with `var` are **hoisted**.
- Variables declared with `let` **cannot be redeclared** in the same scope, while variables declared with `var` can be.

2.2 const

`const` is similar to `let`, but it has one key difference:

- Variables declared with `const` **cannot be reassigned** to a new value.

Example of using `const` is to store a value that you know will not change, like the value of $\pi = 3.14159$.

It can also be used in DOM manipulation to store references to elements that you know will not change

```
const p = document.getElementById('myParagraph');

// Even if you change any property of the element, no problem
// the reference to the element is still the same
p.textContent = "Hello, World"
```

Here are some examples of using `let` and `const`:

```
// Block scope
{
  let x = 5;
  var y = 10;
  console.log(x); // 5
  console.log(y); // 10
```

```
}
// console.log(x); // ReferenceError: x is not defined, because `let` is block-scoped
console.log(y);    // 10

// reassignment
let x = 15;
const z = 20;
x = 25;
// z = 30; // TypeError: Assignment to constant variable, because `const` does not allow reassignment

// redeclaration
var y = 30;
var y = 35;

let a = 40;
// let a = 45; // SyntaxError: Identifier 'a' has already been declared
```

Temporal Dead Zone (TDZ) with `let`

Variables declared with `let` get hoisted to the top of the block, but they are not initialized until the line where they are declared is reached. This is called the Temporal Dead Zone (TDZ).

Example:

```
console.log(x); // ReferenceError: Cannot access 'x' before initialization
let x = 5;
```

2.2.1 `for of` & `const`

When using `const` with `for of`, it will not throw an error because in a `for...of` loop, each iteration creates a new block scope, allowing `const` to be safely used without reassignment errors.

This means it doesn't reassign the variable, but it creates a new variable in a new scope in each iteration.

```
const arr = [1, 2, 3, 4, 5];

for (const item of arr) {
  console.log(item);
}
```

Note: *Now you should always use either* ***`let`*** *or* ***`const`****, never use* ***`var`*** *again.*

3 Default Parameter Value

ES6 introduced a new feature called default parameter values. This allows you to specify a default value for a parameter in a function if no argument is provided.

```
function greet(name = 'World') {
  console.log(`Hello, ${name}!`);
}

greet(); // Hello, World!
greet('Mohamed'); // Hello, Mohamed!
```

This feature is useful when you want to provide a default value for a parameter if no argument is provided.

The old way to do this was to use the `||` operator or an `if` statement:

```
// Using the || operator
function greet(name) {
  name = name || 'World';
  console.log(`Hello, ${name}!`);
}

greet(); // Hello, World!
greet('Mohamed'); // Hello, Mohamed!

// Using an if statement
function greet(name) {
  if (name === undefined) {
    name = 'World';
  }
  console.log(`Hello, ${name}!`);
}

greet(); // Hello, World!
greet('Mohamed'); // Hello, Mohamed!
```

4 Template Literal `String`

ES6 introduced a new way to create strings called template literals. Template literals are enclosed by backticks (`) instead of single quotes (') or double quotes (").

Template literals can contain placeholders, which are indicated by the dollar sign and curly braces (`${expression}`). The expression inside the curly braces is evaluated and the result is inserted into the string.

```
let name = 'Mohamed';
let age = 30;

// Old way
let message = 'Hello, ' + name + '! You are ' + age + ' years old.';
console.log(message); // Hello, Mohamed! You are 30 years old.

// New way
let message = `Hello, ${name}! You are ${age} years old.`;
console.log(message); // Hello, Mohamed! You are 30 years old.
```

Template literals can span multiple lines without the need for escape characters:

```
let message = `This is a
multi-line
string.`;
console.log(message); // This is a
                      // multi-line
                      // string.
```

5 Destruction Assignment

Destructuring assignment is a feature introduced in ES6 that allows you to extract values from arrays or objects and assign them to variables in a more concise way.

5.1 Array Destructuring

Array destructuring allows you to extract values from an array and assign them to variables in a single statement.

```
let numbers = [1, 2, 3, 4, 5];

[a, b, c, d, e] = numbers;
console.log(a, b, c, d, e); // 1 2 3 4 5

[f, g, h] = numbers;
console.log(f, g, h); // 1 2 3
```

You can also skip elements in the array by leaving empty spaces:

```
let numbers = [1, 2, 3, 4, 5];

let [a, , c, , e] = numbers;

console.log(a, c, e); // 1 3 5
```

You can also use the rest operator `...` to capture the remaining elements of an array:

```
let numbers = [1, 2, 3, 4, 5];

let [a, b, ...rest] = numbers;

console.log(a, b); // 1 2
console.log(rest); // [3, 4, 5]
```

5.2 Object Destructuring

Object destructuring allows you to extract values from an object and assign them to variables in a single statement.

```
let person = { name: 'Mohamed', age: 30 };

let { name, age } = person;

console.log(name, age); // Mohamed 30
```

You can also use different variable names for the extracted values:

```
let person = { name: 'Mohamed', age: 30 };

let { name: personName, age: personAge } = person;

console.log(personName, personAge); // Mohamed 30
```

You can also provide default values for the variables:

```
let person = { name: 'Mohamed' };

let { name, age = 30 } = person;

console.log(name, age); // Mohamed 30
```

Lets try a more complex example:

```
let person = {
  name: 'Mohamed',
  age: 30,
  address: {
    country: 'USA',
    city: {
      name: 'New York',
      zip: 10001,
    }
  }
};

let {
  name,
  age,
  address: {
    country,
    city: { name: cityName, zip },
  },
} = person;

console.log(name, age, country, cityName, zip); // Mohamed 30 USA New York
↪  10001
```

You can also combine both dot notation and object destructuring:

```
let person = {
  name: 'Mohamed',
  age: 30,
  address: { country: 'USA', city: { name: 'New York', zip: 10001 } },
};

let { zip } = person.address.city;

console.log(zip); // 10001
```

6 this Keyword

The `this` keyword in JavaScript refers to the object it belongs to. It has different values depending on where it is used:

- In a method, `this` refers to the owner object.
- Alone, `this` refers to the global object. In a browser, it refers to the `window` object.
- In a function, `this` refers to the global object too.
- In a function, in strict mode, `this` is `undefined`.
- In an event, `this` refers to the element that received the event. For example, `e.target` is equivalent to `this.target`.
- In an object, `this` refers to the object itself.

In JavaScript, `this` always refers to the "owner" of the function we're executing, or rather, to the object that a function is a method of.

```
let person = {
  firstName: 'Mohamed',
  lastName: 'Ahmed',
  fullName: function() {
    return this.firstName + ' ' + this.lastName;
  }
};

console.log(person.fullName()); // Mohamed Ahmed
```

In the example above, `this` refers to the `person` object because the `fullName` function is a method of the `person` object.

If you were to call the `fullName` function without the `person` object:

```
let person = {
  firstName: 'Mohamed',
  lastName: 'Ahmed',
  fullName: function() {
    return this.firstName + ' ' + this.lastName;
  }
};

let fullName = person.fullName;
console.log(fullName()); // TypeError: Cannot read properties of undefined
↪  (reading 'firstName')
```

In this case, `this` refers to the global object because the `fullName` function is not a method of the `person` object. Since the global object does not have `firstName` and `lastName` properties, it throws an error.

6.1 this In A Function Inside An Object Method

When strict mode is not used, if we use this in a function inside an object method, it will refer to the global object.

```
let obj = {
  getThis: function () {
    let innerFunc = function () {
      console.log(this);
    };
    innerFunc();
  },
};

obj.getThis(); // window
```

And if we `"use strict"`, `this` will be `undefined`.

People used to solve this problem by using a variable to store the value of `this` before entering the function.

```
let obj = {
  that: this,
  getThis: function () {
    let that = this;
    let innerFunc = function () {
      console.log(that);
    };
    innerFunc();
  },
};

obj.getThis(); // The object itself
```

7 Arrow Functions

Arrow functions are a new way to write functions introduced in ES6. They provide a more concise syntax for writing functions compared to traditional function expressions.

Arrow functions have the following syntax:

```
let add = (a, b) => a + b;
```

This is equivalent to the following traditional function expression:

```
let add = function(a, b) {
  return a + b;
};
```

Arrow functions have the following features:

- They have a more concise syntax compared to traditional function expressions.
- They do not have their own `this`. They inherit these from the surrounding code.

Here are some examples of arrow functions:

```
// Single parameter
let square = x => x * x;

// Multiple parameters
```

```
let add = (a, b) => a + b;

// No parameters
let greet = () => 'Hello, World!';

// Multiple statements
let sum = (a, b) => {
  let result = a + b;
  return result;
};
```

Some Notes:

- With single parameter you can ignore the parentheses of the parameter.
- With one statement you can ignore the curly braces and the `return` keyword.
- With no parameters you can use empty parentheses.
- With multiple statements you need to use curly braces and the `return` keyword.
- With multiple parameters you need to use parentheses.

7.1 `this` & Arrow Functions

We mentioned earlier that arrow functions do not have their own `this`, they inherit `this` from the surrounding code.

This will help us solve the problem mentioned earlier in the section about `this` in a function inside an object method.

```
let obj = {
  getThis: function () {
    let innerFunc = () => {
      console.log(this);
    };
    innerFunc();
  },
};

obj.getThis(); // The object itself
```

Now we don't need to use a variable to store the value of `this` before entering the function.

8 Set

`Set` was introduced in ES6. A `Set` is a collection of **unique values**. It is similar to an array, but it **does not allow duplicate elements**.

You can create a `Set` by passing an array of values to the `Set` constructor:

```
let set = new Set([1, 2, 3, 4, 5, 1, 2, 3]); // Duplicate values are
↪  removed

console.log(set); // Set(5) { 1, 2, 3, 4, 5 }
```

8.1 Set & Array

You can convert a `Set` to an array using the `Array.from` method:

```
let set = new Set([1, 2, 3]);

let arr = Array.from(set);

console.log(arr); // [1, 2, 3]
```

You can also convert an array to a `Set` using the `Set` constructor:

```
let arr = [1, 2, 3, 4, 5, 1, 2, 3];

let set = new Set(arr);

console.log(set); // Set(5) { 1, 2, 3, 4, 5 }
```

8.2 Set Methods

8.2.1 add

You can add values to a `Set` using the `add` method:

```
let set = new Set();

set.add(1);
set.add(2);
set.add(3);

// Or you can chain the add method
set.add(1).add(2).add(3);

console.log(set); // Set(3) { 1, 2, 3 }
```

8.2.2 size

You can get the number of elements in a `Set` using the `size` property:

```
let set = new Set([1, 2, 3]);

console.log(set.size); // 3
```

8.2.3 has

You can check if a `Set` contains a value using the `has` method:

```
let set = new Set([1, 2, 3]);

console.log(set.has(1)); // true
console.log(set.has(4)); // false
```

8.2.4 delete

You can remove values from a `Set` using the `delete` method:

```
let set = new Set([1, 2, 3]);

set.delete(2);

console.log(set); // Set(2) { 1, 3 }
```

9 Map

`Map` was introduced in ES6. A `Map` is a collection of **key-value pairs**. It is similar to an object, but it has some key differences:

- The keys in a `Map` can be of any type, while the keys in an object are always strings.
- The keys in a `Map` preserve the order in which they were inserted, while the keys in an object do not.
- The size of a `Map` can be easily determined using the `size` property.
- You can easily iterate over the keys and values in a `Map`.
- You can remove an entry from a `Map` using the `delete` method.

Similar to `Set`, you can create a `Map` by passing an array of key-value pairs to the `Map` constructor:

```
let map = new Map([
  ['name', 'Mohamed'],
  ['age', 30],
]);

console.log(map); // Map(2) { 'name' => 'Mohamed', 'age' => 30 }
```

9.1 Map & Object

You can convert an object to a `Map` using the `Map` constructor and the `Object.entries` method:

```
let obj = { name: 'Mohamed', age: 30 };

let map = new Map(Object.entries(obj));

console.log(map); // Map(2) { 'name' => 'Mohamed', 'age' => 30 }
```

You can also convert a `Map` to an object using the `Object.fromEntries` method:

```
let map = new Map([
  ['name', 'Mohamed'],
  ['age', 30],
]);

let obj = Object.fromEntries(map);

console.log(obj); // { name: 'Mohamed', age: 30 }
```

9.2 Map Methods

9.2.1 set

You can add key-value pairs to a Map using the set method:

```
let map = new Map();

map.set('name', 'Mohamed');
map.set('age', 30);

// Or you can chain the set method
map.set('name', 'Mohamed').set('age', 30);

console.log(map); // Map(2) { 'name' => 'Mohamed', 'age' => 30 }
```

9.2.2 size

You can get the number of key-value pairs in a Map using the size property:

```
let map = new Map([
  ['name', 'Mohamed'],
  ['age', 30],
]);

console.log(map.size); // 2
```

9.2.3 keys & values

You can get the keys and values of a Map using the keys and values methods:

```
let map = new Map([
  ['name', 'Mohamed'],
  ['age', 30],
]);

console.log(map.keys()); // MapIterator { 'name', 'age' }
console.log(map.values()); // MapIterator { 'Mohamed', 30 }
```

9.2.4 has

You can check if a Map contains a key using the has method:

```
let map = new Map([
  ['name', 'Mohamed'],
  ['age', 30],
]);

console.log(map.has('name')); // true
console.log(map.has('gender')); // false
```

9.2.5 delete

You can remove key-value pairs from a Map using the delete method:

```
let map = new Map([
  ['name', 'Mohamed'],
  ['age', 30],
]);

map.delete('age');

console.log(map); // Map(1) { 'name' => 'Mohamed' }
```

9.2.6 clear

You can remove all key-value pairs from a `Map` using the `clear` method:

```
let map = new Map([
  ['name', 'Mohamed'],
  ['age', 30],
]);

map.clear();

console.log(map); // Map(0) {size: 0}
```

9.2.7 entries

You can get the key-value pairs of a `Map` using the `entries` method:

```
let map = new Map([
  ['name', 'Mohamed'],
  ['age', 30],
]);


console.log(map.entries()); // MapIterator {'name' => 'Mohamed', 'age' =>
↪  30}
```

9.3 Map Iteration

You can iterate over the key-value pairs of a `Map` using the `for...of` method:

9.3.1 Iterating Over Entries

```
let map = new Map([
  ['name', 'Mohamed'],
  ['age', 30],
]);


for (const entry of map) {
  console.log(entry);
}
```

9.3.2 Iterating Over Keys

```
let map = new Map([
  ['name', 'Mohamed'],
  ['age', 30],
]);

for (const key of map.keys()) {
  console.log(key);
}
```

9.3.3 Iterating Over Values

```
let map = new Map([
  ['name', 'Mohamed'],
  ['age', 30],
]);

for (const value of map.values()) {
  console.log(value);
}
```

9.3.4 Iterating With Destructuring

```
let map = new Map([
  ['name', 'Mohamed'],
  ['age', 30],
]);

for (const [key, value] of map) {
  console.log(key, value);
}
```

10 Summary

In this session, we covered the following topics:

- `"use strict"` which is a way to use a restricted variant of JavaScript that would catch common coding problems and throw exceptions.
- `let` and `const` which are new ways to declare variables in ES6.
- Default parameter values which allow you to specify a default value for a parameter in a function if no argument is provided.
- Template literals which are a new way to create strings in ES6.
- Destructuring assignment which allows you to extract values from arrays or objects and assign them to variables in a more concise way.
- Arrow functions which are a new way to write functions in ES6.
- `this` keyword which refers to the object it belongs to.
- `Set` which is a collection of unique values.
- `Map` which is a collection of key-value pairs.