Day 01

Contents

Topic: ES5 vs ES6

ES5 (ECMAScript 5) and ES6 (ECMAScript 2015) are two different versions of the ECMAScript standard, which is the specification that JavaScript is based on. ES6 introduced several new features and syntax enhancements that make JavaScript code more concise, readable, and maintainable.

ES5 was widely used before ES6 and is still supported by all modern browsers. It includes features such as functions, objects, arrays, and prototypes. Here's an example of ES5 code:

```
// ES5 function declaration
function greet(name) {
  console.log('Hello, ' + name + '!');
}


var person = {
  name: 'John',
  age: 30,
  greet: function() {
    console.log('Hello, my name is ' + this.name + '.');
  }
};
```

ES6 introduced several new features, including arrow functions, block-scoped variables, template literals, and classes. These features enhance code readability and provide more efficient ways to write JavaScript applications. Here's an example of ES6 code:

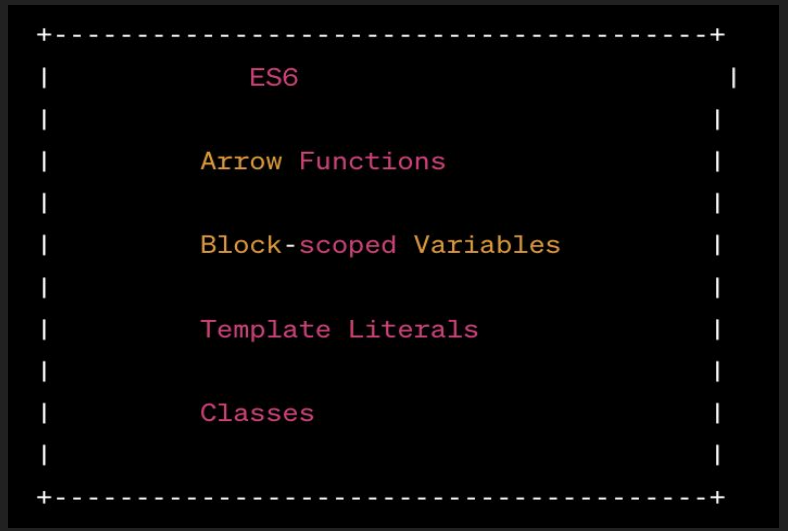
```
// ES6 arrow function
const greet = (name) => {
  console.log(`Hello, ${name}!`);
};

// ES6 object literal shorthand
const name = 'John';
const age = 30;
const person = {
  name,
  age,
  greet() {
    console.log(`Hello, my name is ${this.name}.`);
  }
};
```

ES6 provides a more modern and concise syntax for writing JavaScript code. It introduced several improvements to make code easier to read, write, and maintain. To summarize, ES6 introduced significant enhancements to JavaScript, including arrow functions, block-scoped variables, template literals, and improved object syntax. These features greatly improve developer productivity and code quality.

ES 5 - ES 6


ES5
Function Declaration
Object Literal
Prototypes


ES6
Arrow Functions
Block-scoped Variables
Template Literals
Classes

1. Variable Declarations: var vs let vs const:

In ES5, the primary way to declare variables was using the `var` keyword. However, `var` has function-level scoping and can lead to unexpected behavior. ES6 introduced two new variable declaration keywords: `let` and `const`. `let` and `const` have block-level scoping, which makes code more predictable and prevents unintended variable hoisting.

```
// ES5
var x = 5;
// ES6
let y = 10;
const z = 15;
```

2. Arrow Functions:

Arrow functions are a concise syntax for writing functions in ES6. They have a more compact syntax and inherit the `this` value from the surrounding context.

```
// ES5
var add = function(x, y) {
  return x + y;
};

// ES6
const add = (x, y) => x + y;
```

3. Template Literals:

Template literals provide an improved way to concatenate strings in ES6 by using backticks (`) instead of quotes. They also support string interpolation and multiline strings.

```
// ES5
var name = 'John';
var message = 'Hello, ' + name + '!';

// ES6
const name = 'John';
const message = `Hello, ${name}!`;
```

4.Object and Array Destructuring:

ES6 introduced destructuring, which allows you to extract values from objects or arrays into individual variables.

```
//  ES5
var person = { name: 'John', age: 30 };
var name = person.name;
var age = person.age;

// ES6
const person = { name: 'John', age: 30 };
const { name, age } = person;
```

5. Classes:

ES6 introduced a more familiar syntax for creating classes in JavaScript, providing syntactic sugar over prototype-based inheritance.

```
// ES5
function Person(name) {
  this.name = name;
}

Person.prototype.greet = function() {
  console.log('Hello, ' + this.name + '!');
};

// ES6
class Person {
  constructor(name) {
    this.name = name;
  }

  greet() {
    console.log(`Hello, ${this.name}!`);
  }
}
```

Variables and Scoping in JS

Scoping refers to the visibility and accessibility of variables within different parts of a program. In JavaScript, there are three main ways to declare variables: var, let, and const. Each of these has different scoping rules and behaviors.

It's important to note that `var` variables are function-scoped, while `let` and `const` variables are block-scoped. Block-scoping allows for better control and prevents variable hoisting and unintended variable sharing. It's generally recommended to use `let` and `const` instead of `var` to avoid potential issues and make code more predictable and maintainable.

1. var:

In ES5 and earlier versions, the var keyword was used to declare variables. Variables declared with var have function-level scope, meaning they are accessible throughout the entire function in which they are declared. If var is used outside of any function, the variable becomes globally scoped.

In the example, both x and y are accessible within the example function, even though y is declared inside an if block. The variable x is not accessible outside the function.

```
function example() {
  var x = 5;

  if (true) {
    var y = 10;
    console.log(x); // Output: 5
  }

  console.log(y); // Output: 10
}

console.log(x); // Error: x is not defined
```

2. let:

Introduced in ES6, the let keyword allows block-level scoping. Variables declared with let are only accessible within the block in which they are defined, including any nested blocks.

```
function example() {
  let x = 5;

  if (true) {
    let y = 10;
    console.log(x); // Output: 5
  }

  console.log(y); // Error: y is not defined
}

console.log(x); // Error: x is not defined
```

In this example, x is accessible within the example function, but y is only accessible within the if block. Trying to access y outside the block results in an error.

3. const:

The const keyword is also introduced in ES6 and is used to declare constants. Variables declared with const are block-scoped and have a read-only value. Once assigned, the value of a const variable cannot be changed.

```
function example() {
  const x = 5;

  if (true) {
    const y = 10;
    console.log(x); // Output: 5
  }

  y = 20; // Error: Assignment to constant variable
}

console.log(x); // Error: x is not defined
```

In this example, `x` is a constant variable accessible within the `example` function. However, attempting to assign a new value to `y` results in an error.

The this keyword in JavaScript is a special keyword that refers to the context in which a function is called. It allows access to the object that owns the executing code. The behavior of this can be a bit complex and depends on how the function is invoked.
In JavaScript, this is determined by the execution context, which can be:

1. Global Scope:

When this is used in the global scope (outside any function), it refers to the global object. In a web browser, the global object is the window object.
Javascript

```
console.log(this); // Output: window (in a web browser)
```

2. Object Method:

When this is used inside a method of an object, it refers to the object itself. The value of this is dynamically determined at the time of method invocation.
Javascript

In the example, this inside the sayHello method refers to the person object.

```
const person = {
  name: 'John',
  sayHello: function() {
    console.log('Hello, ' + this.name + '!');
  }
};

person.sayHello(); // Output: Hello, John!
```

3. **Event Handlers:**

When this is used in an event handler, it typically refers to the element that triggered the event.

html

```
<button onclick="console.log(this)">Click me</button>
```

In this case, this refers to the button element that was clicked.

4. Function Context:

When this is used in a regular function (not an arrow function), its value depends on how the function is called. If the function is called with a context object using dot notation, this refers to that object.
javascript

In the example, the call method is used to invoke the greet function with the person object as the context.

It's important to note that arrow functions do not have their own this value. Instead, they inherit this from the surrounding lexical scope.
Understanding the context of this is crucial for proper function invocation and accessing the correct object properties. It allows for dynamic behavior based on the execution context and enables code reuse through object methods.

```
function greet() {
  console.log('Hello, ' + this.name + '!');
}

const person = {
  name: 'John'
};

greet.call(person); // Output: Hello, John!
```

Arrow Function

Arrow functions were introduced in ES6 as a more concise syntax for writing JavaScript functions. They have a shorter syntax compared to regular functions and offer some unique features.

Key features of arrow functions:

- Arrow functions do not have their own this value. They inherit the this value from the surrounding lexical scope, which eliminates the need to use the bind, call, or apply methods to preserve this context.
- Arrow functions do not have their own arguments object. However, you can still access the arguments passed to the enclosing function using the rest parameter syntax (...args).

```
// ES5 function
function add(x, y) {
  return x + y;
}

// ES6 arrow function
const add = (x, y) => x + y;
```

2. Template Literals:

Template literals provide an improved way to work with strings in JavaScript. They use backticks (`) instead of single or double quotes and allow for easy string interpolation and multiline strings.

javascript

Key features of template literals:

- Template literals support string interpolation, allowing variables and expressions to be embedded directly within the string using ${} syntax.
- Multiline strings can be easily created without the need for escape characters or concatenation.

```
const name = 'John';
const age = 30;

// ES5 string concatenation
const message = 'My name is ' + name + ' and I am ' + age + ' years old.';

// ES6 template literal
const message = `My name is ${name} and I am ${age} years old.`;
```

3. Spread & Rest Operator:

The spread and rest operators were introduced in ES6 and provide a convenient way to work with arrays and objects.

- Spread Operator (...): The spread operator allows an array or object to be expanded into individual elements. It is used to make copies, merge arrays, or pass multiple arguments to a function.

javascript

```
const numbers = [1, 2, 3];
const moreNumbers = [4, 5, 6];

// Copying an array
const copy = [...numbers];

// Merging arrays
const merged = [...numbers, ...moreNumbers];

// Passing multiple arguments to a function
const max = Math.max(...numbers);
```

Rest Parameter (...): The rest parameter allows us to represent an indefinite number of arguments as an array. It is used in function definitions to collect multiple arguments into a single array.
Javascript

In the example, the rest parameter ...numbers collects all the arguments passed to the sum function into an array.
The spread and rest operators provide powerful capabilities for working with arrays and objects, making code more concise and expressive. They are widely used in modern JavaScript development to simplify common tasks and improve code readability.

```
function sum(...numbers) {
  return numbers.reduce((total, num) => total + num, 0);
}

console.log(sum(1, 2, 3, 4)); // Output: 10
```

Array and object destructuring is a feature introduced in ES6 that provides a concise way to extract values from arrays and objects into individual variables. It allows you to unpack values from complex data structures, making code more readable and eliminating the need for repetitive variable assignments.

1. Array Destructuring:

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

javascript

In the example , a and b are assigned the first two values from the numbers array, while the rest of the values are collected in the rest array using the rest parameter syntax (...).

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

// Destructuring assignment
const [a, b, ...rest] = numbers;

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

2. Object Destructuring:

Object destructuring allows you to extract values from an object and assign them to variables with the same names as the object's properties.

javascript

In the example, the variables name, age, and city are assigned the corresponding values from the person object. The nested property address.city is accessed using the syntax address: { city }.

```javascript
const person = {
  name: 'John',
  age: 30,
  address: {
    city: 'New York',
    country: 'USA'
  }
};

// Destructuring assignment
const { name, age, address: { city } } = person;

console.log(name);    // Output: John
console.log(age);     // Output: 30
console.log(city);    // Output: New York
```

3. Default Values:

Destructuring also allows you to provide default values for variables in case the corresponding value is undefined.

javascript

In the example, the variable city is assigned the default value 'Unknown' since the person object does not have a city property.

Array and object destructuring provides a concise and convenient way to extract values from arrays and objects. It simplifies code by reducing the need for manual value assignment and improves code readability by clearly expressing the intended structure of the data being manipulated.

```
const person = {
  name: 'John',
  age: 30
};

const { name, age, city = 'Unknown' } = person;

console.log(name);   // Output: John
console.log(age);    // Output: 30
console.log(city);   // Output: Unknown
```

Module Import and Export:

Module import and export is a feature introduced in ES6 that allows JavaScript code to be organized into modular pieces, making it easier to manage dependencies and reuse code across different files.

1. Exporting from a Module:

To export a value or a set of values from a module, you can use the export keyword. There are different ways to export:

- Named Exports:

```
// Exporting individual values
export const name = 'John';
export const age = 30;

// Exporting functions
export function greet() {
  console.log('Hello!');
}

// Exporting objects
export const person = {
  name: 'John',
  age: 30
};
```

- Default Export:

```
// Exporting a default value
export default function sayHello() {
  console.log('Hello!');
}
```

2. Importing in a Module:

To use values exported from another module, you need to import them using the import keyword.

- Named Imports:

```
// Importing specific named values
import { name, age } from './person.js';

// Importing all named values
import * as person from './person.js';

// Using the imported values
console.log(name);
console.log(age);
```

- Default Import:

```
// Importing a default value
import sayHello from './greeting.js';

// Using the imported default value
sayHello();
```

3. Combining Named and Default Exports:

A module can have both named and default exports. Module import and export enable code organization and encapsulation, preventing global scope pollution and allowing for better code maintainability and reusability.

```
// Exporting named values
export const name = 'John';
export const age = 30;

// Exporting a default value
export default function sayHello() {
  console.log('Hello!');
}
```

```
// Importing named values and default value
import { name, age, default as sayHello } from './person.js';

console.log(name);
console.log(age);
sayHello();
```

Class in JavaScript:
Classes in JavaScript are a way to create objects based on a blueprint, known as a class. They provide a more structured and object-oriented approach to defining and creating objects.

1. Class Declaration:

In the example, the Rectangle class is declared using the class keyword. It has a constructor method that is called when an instance of the class is created. The class also has a getArea method that calculates and returns the area of the rectangle.

```
class Rectangle {
  constructor(width, height) {
    this.width = width;
    this.height = height;
  }

  getArea() {
    return this.width * this.height;
  }
}
```

2. Creating Instances:

Instances of a class are created using the new keyword followed by the class name. In the example above, a new Rectangle object is created with

```
const rectangle = new Rectangle(5, 10);
console.log(rectangle.getArea()); // Output: 50
```