with statement

- The with statement in JavaScript extends the scope chain for a statement block. It allows developers to simplify code by avoiding repetitive references to an object's properties.
- with statement is considered bad practice and is not allowed in strict mode because it can lead to unpredictable behavior, making code harder to optimize and debug. Its usage is generally discouraged due to these issues.

Syntax

```
with (object) {
  // Code block where object's properties are used as local variables
}
```

Example

```
let person = {
        name: "Neha",
        age: 25,
        city: "New Delhi"
};
// Without "with" statement
console.log(person.name);
console.log(person.age);
console.log(person.city);

// Using "with" statement
with (person) {
        console.log(name);
        console.log(age);
        console.log(city);
}
```

In the example, the properties of the person object (name, age, and city) can be accessed directly inside the with block without needing to reference person repeatedly.

Use Cases

- **Reducing code repetition:** It was useful when dealing with large objects to avoid repeating the object's name.
- **Working with nested objects:** Accessing deeply nested object properties becomes easier, as you don't need to reference the full path every time.

Example

```
let car = {
  engine: {
    horsepower: 150,
    type: "bs6"
  }
};

with (car.engine) {
  console.log(horsepower);
  console.log(type);
}
```

Why you need to avoid using with?

- **Ambiguity:** It can make code unpredictable. If a variable or function is used inside the with block, JavaScript may have difficulty determining if it refers to a property of the object or a globally defined variable.
- **Performance:** It hampers JavaScript engines' ability to optimize code efficiently, since the exact scope is not clear at runtime.
- **Not Allowed in Strict Mode:** If you're using strict mode ("use strict"), which is a best practice for writing secure and efficient code, with will throw an error.

Best Practice: Instead of using with, you can use object destructuring or simply assign the object to a variable.

```
let { name, age, city } = person;
console.log(name);
console.log(age);
console.log(city);
```

This achieves a similar result without the issues that come with the with statement.

Object Destructuring

- Object destructuring is a powerful and concise way to extract data from objects.
- Object destructuring is a syntax feature in JavaScript that allows you to extract properties from objects and assign them to variables.
- It helps reduce redundancy and makes the code cleaner, especially when dealing with large or deeply nested objects.
- This is useful when you want to access object properties directly without using dot notation repeatedly.
- Destructuring makes the code cleaner, more readable, and easier to manage.
- Its flexibility, such as renaming variables, assigning default values, and using it in function parameters, makes it a highly practical feature for modern JavaScript development.

Basic Syntax

```
const object = { key1: value1, key2: value2 };
const { key1, key2 } = object;
```

where,

- key1 and key2 are the keys (property names) of the object.
- The variables key1 and key2 will hold the values of value1 and value2 respectively.

Example: Simple Object Destructuring

```
const person = {
                    name: "Neha Mittal",
                    age: 21,
                    city: "Agra"
};
const { name, age, city } = person;
console.log(name);
console.log(age);
console.log(city);
```

Example: Renaming Variables

You can rename the variables while destructuring by using a colon (:).

```
const user = {
                username: "Neha Mittal",
                email: "neha@learn2earnlabs.com"
};
const { username: name, email: userEmail } = user;
console.log(name);
console.log(userEmail);
```

Example: Providing Default Values

If a property doesn't exist in the object, you can assign a default value.

```
const person = {
                              name: "Neha Mittal",
                              age: 21,
};
const { name, age, gender = "female" } = person;
console.log(name);
console.log(age);
console.log(gender);
```

Example: Nested Object Destructuring

You can also destructure nested objects.

```
const student = {
                id: 1,
                name: "Neha Mittal",
                details: {
                                age: 21,
                                city: "Agra"
                       }
};
const { name, details: { age, city } } = student;
console.log(name);
console.log(age);
console.log(city);
```

Example: Destructuring in Function Parameters

Object destructuring can be used in function parameters for better clarity and avoiding repetitive property access.

```
const displayStudent = ({ name, email }) => {
                                        console.log(`Name: ${name}`);
                                        console.log(`Email: ${email}`);
};
const student = {
              name: "Neha Mittal",
              email: " neha@learn2earnlabs.com"
};
displayStudent(student);
```

Example: Destructuring with Rest Parameters

You can collect the remaining properties of an object into a new object using the rest (...) operator.

```
const person = {
                    name: "Neha Mittal",
                    age: 21,
                    city: "Agra",
                    state: "Uttar Pradesh",
                    country: "India"
};

const { name, age, ...address } = person;
console.log(name);
console.log(age);
console.log(address);
```

Strict Mode

- Strict mode is a powerful tool for writing cleaner, more robust JavaScript code.
- By enforcing stricter parsing and error handling, it helps developers avoid common pitfalls and write safer, more predictable code.
- Using strict mode is generally considered good practice, especially in larger applications or when working in teams.
- Strict mode in JavaScript is a way to opt in to a restricted variant of JavaScript, allowing you to write safer and more optimized code.
- It helps catch common coding errors and "unsafe" actions such as defining global variables unintentionally.
- You can enable strict mode by adding the directive "use strict"; at the beginning of a script or a function.

How to Enable Strict Mode

a) **Global Scope:**

```
"use strict";
// Code in strict mode
```

b) **Function Scope:**

```
function myFunction() {
        "use strict";
        // Code in strict mode
}
```

c) **ES6 Modules:** By default, all code in ES6 modules is in strict mode.

Benefits of Strict Mode

- **Eliminates Silent Errors:** In non-strict mode, some actions fail silently. For example, assigning a value to an undeclared variable does not throw an error. In strict mode, it will throw a ReferenceError.

```
"use strict";
x = 3.14; // ReferenceError: x is not defined
```

- **Prevents Accidental Globals:** Strict mode disallows the use of undeclared variables, preventing accidental global variable creation.

```
"use strict";
function myFunction() {
   y = 3.14; // ReferenceError: y is not defined
}
```

- **Disallows Duplicate Parameter Names:** In strict mode, you cannot have duplicate parameters in a function declaration.

```
"use strict";
function myFunction(a, a, b) {
// SyntaxError: Duplicate parameter name not allowed in this context
   // ...
}
```

- **Restricts this Keyword:** In strict mode, this in functions that are not called as methods of an object (e.g., standalone functions) will be undefined instead of the global object (window in browsers). This helps avoid unintended behavior.

```
"use strict";
function myFunction() {
                        console.log(this); // undefined
}
myFunction();
```

- **Prohibits with Statement:** The with statement is not allowed in strict mode. This helps improve code clarity and avoid ambiguity in variable lookup.

```
"use strict";
with (Math) { // SyntaxError: Strict mode does not allow with statements
   // ...
}
```

- **Improves Performance:** Some JavaScript engines can optimize strict mode code better than non-strict mode code, potentially leading to performance benefits.
- **Secures eval:** In strict mode, variables and functions declared inside an eval block are not accessible outside of it. This prevents unexpected behaviour and variable leakage.

```
"use strict";
eval("var x = 10;");
console.log(x); // ReferenceError: x is not defined
```

- **Avoids delete on Non-Configurable Properties:** In strict mode, trying to delete a variable, function, or function parameter will throw an error.

```
"use strict";
var obj = {};
Object.defineProperty(obj, "prop", { value: 42, configurable: false });
delete obj.prop; // TypeError: Cannot delete property 'prop' of #<Object>
```

Object.defineProperty()

- Object.defineProperty() is a powerful method in JavaScript that allows you to define or modify properties of an object with fine-grained control over their behavior.
- Unlike simply assigning properties via dot notation, Object.defineProperty() provides control over a property's enumerability, configurability, and writability.

Syntax

```
Object.defineProperty(obj, propertyName, descriptor);
```

where,

- obj: The object on which to define the property.
- propertyName: The name of the property being defined or modified.
- descriptor: An object that defines the behaviour of the property.

Property Descriptors

There are two kinds of property descriptors:

- **Data descriptors:** describe properties that have a value.
- **Accessor descriptors:** describe properties that have getter and setter functions.

Below are the key attributes in a descriptor:

- value: The value associated with the property (for data properties).
- writable: If true, the value of the property can be changed; if false, it's read-only.
- configurable: If true, the property can be deleted or modified (its attributes changed).
- enumerable: If true, the property appears during iteration over the object's properties.
- get: A function that serves as a getter for the property.
- set: A function that serves as a setter for the property.

Example: Basic Property Definition

```
const obj = {};
Object.defineProperty(obj, 'name', {
                                    value: 'Neha Mittal',
                                    writable: true,
                                    enumerable: true,
                                    configurable: true
                                  });
console.log(obj.name);
```

The object property is both writable (its value can be changed), enumerable (it will show up during property iteration), and configurable (it can be deleted or reconfigured).

Example: Non-Configurable Property

Once a property is marked as non-configurable, it cannot be deleted or have its descriptor attributes (except writable) changed.

```
const person = {};
Object.defineProperty(person, 'age', {
                    value: 21,
                    configurable: false // Not allowed to delete or reconfigure
});
console.log(person.age);
delete person.age;     // fail (in strict mode, it throws an error)
console.log(person.age);
```

Example: Non-Enumerable Property

A non-enumerable property will not show up in loops or Object.keys().

```
const person = {};
Object.defineProperty(person, 'name', {
                          value: 'Neha Mittal',
                          enumerable: false // Not included in enumeration
});
console.log(Object.keys(person));
console.log(person.name);
```

Example: Read-Only Property (Non-Writable)

A non-writable property cannot be modified after its definition.

```
const person = {};
                Object.defineProperty(person, 'age', {
                 value: 21,
                 writable: false  // Read-only property
});
console.log(person.age);
person.age = 19;        // fail (in strict mode, it throws an error)
console.log(person.age);
```

Example: Getters and Setters

Object.defineProperty() can define getters and setters to control access to object properties.

Getter

```
const person = {
                 firstName: "Neha",
                 lastName: "Mittal"
};
Object.defineProperty(person, 'name', {
                      get() {
                       return `${this.firstName} ${this.lastName}`;
                      }
});
console.log(person.name);
```

Setter

```
const person = {
                    firstName: 'Neha',
                    lastName: 'Mittal'
};
Object.defineProperty(person, 'fullName', {
                          get() {
                               return `${this.firstName} ${this.lastName}`;
                          },

                         set(value) {
                           [this.firstName, this.lastName] = value.split(' ');
                          }
});
console.log(person.fullName);
person.fullName = 'Saurabh Upadhyay';
console.log(person.firstName);
console.log(person.lastName);
```

Example: Multiple Properties at Once (Object.defineProperties())

You can define multiple properties at once using Object.defineProperties().

```
const person = {};
Object.defineProperties(person, {
                         name: {   value: 'Neha Mittal',  writable: true  },
                         age: {     value: 21,      writable: false   },
                         description: {    get() {
                               return `${this.name} is ${this.age} years old.`;
                                     }
                                     }
});
console.log(person.name);
console.log(person.age);
console.log(person.description);
```

Example: Default Descriptor Behavior

When using Object.defineProperty(), by default, properties are non-writable, non-enumerable, and non-configurable unless explicitly stated otherwise.

```
const person = {};
Object.defineProperty(person, 'intro', {
                                        value: "Hi, I am Neha Mittal"
});

console.log(person.intro);
person.intro="I am learning";     // won't change the value (not writable)
console.log(person.intro);

console.log(Object.keys(person)); // [] (not enumerable)
delete person.intro;      // won't delete the property (not configurable)
console.log(person.intro);
```

Use Cases of Object.defineProperty()

- **Creating read-only properties:** Ideal when you need properties that should not be changed after creation.
- **Defining computed properties:** Use getters to define properties whose values are computed dynamically.
- **Hiding implementation details:** You can define properties that are not enumerable, which hides them during object iteration.
- **Precise control over property behavior:** Helps to ensure that properties follow strict rules for being writable, configurable, or enumerable.
- **Creating immutable objects:** By combining writable: false and configurable: false, you can effectively create immutable properties.