Week 1 - JS Review

Amr A Khllaf

August 7, 2025

Contents

1 Intro and Tools

- Node.js is an Environment for running JavaScript outside the browser.
- It allows developers to use JavaScript for server-side scripting.
- Key features include non-blocking I/O, event-driven architecture, and a vast ecosystem of libraries.

1.1 What is the difference between Client-side and Server-side JavaScript?

- **Client-side JavaScript** runs in the user's browser, handling user interactions and dynamic content updates.
- **Server-side JavaScript** runs on the server, managing database interactions, server logic, and serving web pages to clients.
- Node.js is a popular platform for **server-side JavaScript**, enabling developers to build scalable network applications.
- To make the js works on the server, we need to install Node.js and use the `node` command to execute JavaScript files.

1.2 What is the difference between SSR (Server-Side Rendering) and CSR (Client-Side Rendering)?

- **Server-Side Rendering (SSR)**: The server generates the HTML content and sends it to the client. This approach is beneficial for SEO and initial page load performance.
- **Client-Side Rendering (CSR)**: The client (browser) fetches JavaScript files and renders the content dynamically. This approach allows for more interactive user experiences but may have slower initial load times.
- Node.js is primarily used for SSR, while frameworks like React and Angular are often used for CSR.

1.3 What is the Module System in Node.js?

- Node.js uses the CommonJS module system, which allows developers to create reusable modules.
- Each module has its own scope, preventing variable collisions.
- Modules can export their functionality using `module.exports` and import other modules using `require()`.
- This system promotes code organization and reusability, making it easier to manage larger applications.

1.3.1 3 Types of Modules

1. **Core Modules**: Built-in modules provided by Node.js, such as `fs` (file system), `http`, and `path`.

2. **Local Modules**: Custom modules created by developers, stored in the project directory.
3. **Third-party Modules**: Modules installed from the npm registry, which can be added to projects using the `npm install` command.

1.4 What is the IDE & What is the difference between an IDE and a Text Editor?

- An IDE (Integrated Development Environment) is a software application that provides comprehensive facilities to programmers for software development.
- A Text Editor is a simpler tool for editing code, often lacking advanced features like debugging, version control, and integrated terminal.
- **Difference**: An IDE typically includes features like code completion, debugging tools, and project management, while a text editor is more lightweight and focused on basic code editing.

1.5 What is the difference between a Text Editor and an IDE?

- A Text Editor is a basic tool for writing and editing code, often with syntax highlighting and simple features.
- An IDE (Integrated Development Environment) is a more advanced tool that includes features like debugging, code completion, version control integration, and project management.
- IDEs provide a more comprehensive environment for software development, while text editors are lightweight and focused on code editing.

---

2 Run Js from the Client & From the Server

2.1 How to run JavaScript from the Client?

- JavaScript can be run in the browser by including it in HTML files using the `<script>` tag.
- It can also be executed in the browser console for quick testing and debugging.
- Example of running JavaScript in the browser:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width,
    ↪  initial-scale=1.0" />
    <title>Client-side JS</title>
  </head>
  <body>
    <h1>Hello, World!</h1>
    <script>
      console.log("This is running from the client!");
    </script>
  </body>
</html>
```

   - Open the HTML file in a web browser to see the output in the console.

2.2 How to run JavaScript from the Server?

- To make sure that node is installed, run the command `node -v` in the terminal.
- To Know the version of npm, run the command `npm -v`.'
- JavaScript can be run on the server using Node.js by creating a `.js` file and executing it with the `node` command.
- Example of running JavaScript on the server:

  ```
  // server.js
  console.log("This is running from the server!");
  ```

- To run this file, open the terminal, navigate to the directory where the file is located, and execute:

  ```
  node server.js
  ```

- This will execute the JavaScript code in the server environment, and you should see the output in the terminal.
- **Note:** If i use any method from the browser in the server, it will not work because the server does not have access to the browser's DOM or APIs.
- For example, trying to use `document.getElementById()` in a Node.js environment will result in an error because Node.js does not have a `document` object like browsers do.

```
// server.js
console.log(document.getElementById("demo").innerText);
```

- This code will throw an error because `document` is not defined in the Node.js environment.

---

3 Variables and Primitive Data Types

- Java Script is a **Loosely Typed Language**, meaning variables can hold values of any type without strict type declarations.

3.1 What are the 2 Types of Variables in JavaScript?

- In JavaScript, there are two types of variables:
  1. **Primitive Variables**: Hold a single value and are immutable.
  2. **Reference Variables (Non-Primitive)**: Hold a reference to an object and can be mutable.
- Variable name has some rules:
  - Must start with a letter, underscore (_), or dollar sign ($).

  - Can contain letters, numbers, underscores, and dollar signs.
  - Cannot start with a number.
  - Case-sensitive (e.g., `myVar` and `myvar` are different variables).

- if the variable name is consist with a single word, it can be written in any case, but if it is consist with multiple words, it should be written in camelCase or snake_case.
  - **Example of camelCase: `myVariableName`**
  - Example of snake_case: `my_variable_name`

3.2 What are the 6 Primitive Data Types in JavaScript?

- JavaScript has 6 primitive data types:
  1. **String**: Represents a sequence of characters, enclosed in single (”), double quotes (““), or backticks (“).
     - Example: `let name = "John";`
  2. **Number**: Represents numeric values, both integers and floating-point numbers.
     - Example: `let age = 30;`
  3. **Boolean**: Represents a logical value, either `true` or `false`.
     - Example: `let isStudent = true;`
  4. **Undefined**: Represents a variable that has been declared but not assigned a value.
     - Example: `let x; // x is undefined`
  5. **Null**: Represents the intentional absence of any object value.
     - Example: `let y = null;`
  6. **Symbol**: Represents a unique and immutable value, often used as object property keys.
     - Example: `let sym = Symbol("description");`
- These primitive data types are **immutable**, meaning their values cannot be changed once created.

3.2.1 typeof Operator

- The `typeof` operator is used to determine the type of a variable or value in JavaScript.
- It returns a string indicating the type of the operand.
- Example:

```javascript
let x = 42;
console.log(typeof x); // "number"
let y = "Hello";
console.log(typeof y); // "string"
let z = true;
console.log(typeof z); // "boolean"
let a;
console.log(typeof a); // "undefined"
let b = null;
console.log(typeof b); // "object" (this is a known quirk in JavaScript)
let c = Symbol("unique");
console.log(typeof c); // "symbol"
```

3.2.2 Number Data Type and Octals

- The `Number` data type in JavaScript can represent both integers and floating-point numbers.
- JavaScript does not have a specific octal data type, but it can represent octal numbers using a specific syntax.
- To represent an octal number, prefix it with `0o` (zero followed by a lowercase 'o').
- Example of octal representation:

```
let octalNumber = 0o10; // Represents the decimal number 8
console.log(octalNumber); // Output: 8

let num = 0o125678; // This will throw a SyntaxError in strict mode
console.log(num); // Output: 125678 ==> 125678 (not a valid octal in
↪  strict mode)
```

- Note: In strict mode, using numbers with leading zeros (like `0123`) is not allowed and will throw a `SyntaxError`. Always use the `0o` prefix for octal numbers.

- To use number starting with 0, you need to put it in a string format, like this:

```
let num = "0125678"; // This is a string representation of the number
console.log(num); // Output: "0125678"
```

- This way, you can avoid the syntax error while still representing the number as a string.
- If you need to perform arithmetic operations on this string, you can convert it to a number using `Number()` or `parseInt()`:

```
let numString = "0125678";
let num = Number(numString); // Convert string to number
console.log(num); // Output: 125678 (leading zeros are ignored in number
↪  representation)
```

- We can make it by adding a `+` sign before the string:

```
let numString = "0125678";
let num = +numString; // Convert string to number using unary plus
console.log(num); // Output: 125678 (leading zeros are ignored in number
↪  representation)
```

3.2.3 When to use null and when to use undefined?

- **Undefined** is used when a variable has been declared but not assigned a value. It indicates that the variable exists but has no value.
- **Null** is used to explicitly indicate that a variable has no value or is intentionally empty. It is often used to represent the absence of an object or value.
- Use `undefined` when you want to check if a variable has not been assigned a value yet.
- Use `null` when you want to explicitly indicate that a variable should not have any value or object associated with it.
- Example:

```
let a;
console.log(a); // Output: undefined
let b = null;
console.log(b); // Output: null
```

```javascript
let c = "Hello";
console.log(c); // Output: Hello
let d = c + a; // Concatenation with undefined
console.log(d); // Output: "Helloundefined"
let e = c + b; // Concatenation with null
console.log(e); // Output: "Hellonull"
let f = c + "World"; // Concatenation with a string
console.log(f); // Output: "HelloWorld"
```

- In the above example, `a` is `undefined` because it has not been assigned a value, while `b` is `null` because it has been explicitly set to have no value. The concatenation with `undefined` results in the string “Helloundefined”, while concatenation with `null` results in “Hellonull”.

---

4 Reference (Non-Primitive) Data Types

- Reference data types in JavaScript are used to store collections of values or more complex entities.
- They are a collection of primitive data types.

4.1 What are the 3 Types of Reference Data Types in JavaScript?

1. **Object**: A collection of key-value pairs, where keys are strings (or Symbols) and values can be of any data type.
   - Example:

     ```javascript
     let person = {
       name: "John",
       age: 30,
       isStudent: false,
     };
     ```

2. **Array**: An ordered list of values, where each value can be of any data type. Arrays are a special type of object.
   - Example:

     ```javascript
     let fruits = ["apple", "banana", "cherry"];
     ```

3. **Function**: A block of code that can be called and executed. Functions are also objects in JavaScript.
   - Example:

     ```javascript
     function greet(name) {
       return "Hello, " + name + "!";
     }
     ```

4.2 Object (Big Boss of Reference Data Types)

- Objects are collections of key-value pairs, where keys are strings (or Symbols) and values can be of any data type.

- Objects can be created using object literals or the `new Object()` syntax.
- Example of creating an object using an object literal:

```
let person = {
  name: "John",
  age: 30,
  isStudent: false,
};
```

- Accessing object properties can be done using dot notation or bracket notation.

```
console.log(person.name); // Output: John
console.log(person["age"]); // Output: 30
```

- Adding or updating properties in an object can be done using either notation:

```
person.city = "New York"; // Adding a new property
console.log(person.city); // Output: New York
person.age = 31; // Updating an existing property
console.log(person.age); // Output: 31

console.log(person); // Output: { name: "John", age: 31, isStudent: false,
↪  city: "New York" }
```

4.2.1 Accessing using Dot Notation vs Bracket Notation

- **Dot Notation**: Used to access properties directly using the property name.
  - Example: `person.name`
- **Bracket Notation**: Used to access properties using a string or variable that contains the property name.
  - Example: `person["age"]`
- Bracket notation is useful when property names contain spaces or special characters, or when the property name is stored in a variable.

```
let propertyName = "isStudent";
console.log(person[propertyName]); // Output: false

let person = {
  name: "John",
  age: 30,
  isStudent: false,
  boy: {
    name: "Ali",
    age: 20,
    isStudent: true,
    dog: {
      name: "Max",
      age: 5,
      isAnimal: true,
    },
  },
};
```

```
// Accessing nested properties using dot notation
console.log(person.boy.name); // Output: Ali
console.log(person.boy.age); // Output: 20
console.log(person.boy.isStudent); // Output: true
console.log(person.boy.dog.name); // Output: Dog
console.log(person.boy.dog.age); // Output: 5
console.log(person.boy.dog.isAnimal); // Output: true

// Accessing nested properties using bracket notation
console.log(person["boy"]["name"]); // Output: Ali
console.log(person["boy"]["age"]); // Output: 20
console.log(person["boy"]["isStudent"]); // Output: true
console.log(person["boy"]["dog"]["name"]); // Output: Dog
console.log(person["boy"]["dog"]["age"]); // Output: 5
console.log(person["boy"]["dog"]["isAnimal"]); // Output: true
```

- Dot notation is more concise and easier to read, while bracket notation provides more flexibility.
- **We use it also when the property name is dynamic or not a valid identifier (e.g., starts with a number).**

```
let person = {
  "first-name": "John", // Invalid identifier, must use bracket notation
  age: 30,
};
console.log(person["first-name"]); // Output: John
console.log(person.age); // Output: 30

let userInput = window.prompt("Enter a property name:");
if (userInput in person) {
  console.log(person[userInput]); // Accessing property using user input
   ↪  by bracket notation
} else {
  console.log("Property not found.");
}

// Example of using a variable to access a property
let propertyName = "age";
console.log(person[propertyName]); // Output: 30

// Example of using a variable to access a nested property
let nestedProperty = "name";
console.log(person["boy"]["dog"][nestedProperty]); // Output: Max
```

---

5 Update & Add and Delete from Object

- Objects in JavaScript are **mutable**, meaning you can add, update, and delete properties dynamically.

5.1 What is CRUD?

CRUD stands for Create, Read, Update, and Delete. These are the four basic operations you can perform on data in a database or data structure.

- Most of work in backend is CRUD operations.
- **Create**: Adding new data.
- **Read**: Retrieving or accessing existing data.
- **Update**: Modifying existing data.
- **Delete**: Removing data.

In the context of JavaScript objects, you can perform CRUD operations as follows:

1. **Create**: Add a new property to an object.

   ```
   person.email = "john@example.com"; // Adding a new property
   console.log(person.email); // Output: john@example.com
   ```

2. **Read**: Access an existing property.

   ```
   console.log(person.name); // Output: John
   ```

3. **Update**: Modify an existing property.

   ```
   person.age = 31; // Updating an existing property
   console.log(person.age); // Output: 31
   ```

4. **Delete**: Remove a property from an object, using the `delete` operator.

   ```
   delete person.isStudent; // Deleting a property
   console.log(person.isStudent); // Output: undefined (property no
   ↪  longer exists)
   ```

- The `delete` operator removes a property from an object, making it undefined.
- Note: Don't delete anything on runtime, it will cause an error, if you are using it in another part of the code.

```
let person = {
  name: "John",
  age: 30,
  isStudent: false,
};
console.log(person); // Output: { name: "John", age: 30, isStudent: false
↪  }
delete person.isStudent; // Deleting the isStudent property
console.log(person); // Output: { name: "John", age: 30 }
console.log(person.isStudent); // Output: undefined (property no longer
↪  exists)
```

---

6 Destructuring (Destructing)

- Destructuring is a convenient way to extract values from arrays or properties from objects into distinct variables (**3mlya 3axya l tary2t el Declaration**)

```
let user = {
  name: "Amr",
  age: 25,
  salary: 5000,
  wife: {
    name: "Nada",
    age: 23,
    son: {
      name: "Omar",
      age: 13,
    },
  },
};

console.log(user.wife.son.name); // Output: Omar
console.log(user["wife"]["son"]["age"]); // Output: 13
```

- To destructure the `user` object and extract the `name` and `age` of the son, you can do the following:

```
let {
  wife: {
    name: wifeName,
    age: wifeAge,
    son: { name: sonName, age: sonAge },
  },
} = user;

console.log(sonName); // Output: Omar
console.log(sonAge); // Output: 13
console.log(wifeName); // Output: Nada
console.log(wifeAge); // Output: 23

let { name: userNName, age: userAge } = user;
console.log(userName); // Output: Amr
console.log(userAge); // Output: 25
```

- We use the nickname for every property in the object because we may have a variable with the same name in the code, so we need to avoid conflicts.

```
var name = "John"; // Global variable

let user = {
  name: "Amr",
  age: 25,
  salary: 5000,
};
let { name, age: userAge, salary: userSalary } = user;
console.log(name); // SyntaxError: Identifier 'name' has already been
↪  declared

// To Avoid This Error you need to use the nickname for the variable:
```

```
let { name: userName, age: userAge, salary: userSalary } = user;
console.log(userName); // Output: Amr

console.log(name); // Output: John (global variable)
```

7 Merge Objects (Spread Operator)

- The spread operator (...) allows you to merge objects or arrays easily.

```
let user1 = {
  name: "Amr",
  age: 25,
  salary: 5000,
};
let user2 = {
  name: "Nada",
  age: 23,
  salary: 3000,
};
let mergedUser = { ...user1, ...user2 }; // Merging two objects
console.log(mergedUser); // Output: { name: "Nada", age: 23, salary: 3000
↪  }
---

let obj1 = {
  name: "Dell',
  color: "#09c"
}
let obj2 = {
  price:3999,
  model: 2024
}

let mergedObj = {... obj1, ... obj2};
console.table(mergedObj) // Output: { name: "Dell", color: "#09c", price:
↪  3999, model: 2024 }
```

7.1 Difference between console.log(), console.table(), console.group()

- `console.log()` is used to print any value to the console, while `console.table()` is specifically designed to display tabular data in a more readable format.
- `console.table()` is particularly useful for displaying arrays of objects or objects with multiple properties, as it formats the output in a table-like structure.
- `console.group()` is used to group related log messages together, allowing you to create collapsible sections in the console for better organization.

```
let users = [
  { name: "Amr", age: 25, salary: 5000 },
  { name: "Nada", age: 23, salary: 3000 },
];
console.log(users); // Output: Array of objects
console.table(users); // Output: Table format of the array of objects
console.group("User Details");
console.log("Name: " + users[0].name); // Output: Name: Amr
console.log("Age: " + users[0].age); // Output: Age: 25
console.log("Salary: " + users[0].salary); // Output: Salary: 5000
console.groupEnd(); // End of the group

// Output Structure in the console:
// User Details
//    Name: Amr
//    Age: 25
//    Salary: 5000
```

- **Note:** If the key and value are the same, you can use the shorthand syntax to create an object:

```
let name = "Amr";
let age = 25;
let user = { name, age }; // Shorthand syntax
console.log(user); // Output: { name: "Amr", age: 25 }
```

---

8 Shallow Copy vs Deep Copy

8.1 Deference between Call by Value and Call by Reference

- **Call by Value**: When a variable is passed to a function, a copy of its value is made. Changes to the parameter inside the function do not affect the original variable.
- **Call by Reference**: When a variable is passed to a function, a reference to the original variable is made. Changes to the parameter inside the function affect the original variable.
- In JavaScript, primitive data types (like numbers, strings, booleans) are passed by value, while reference data types (like objects and arrays) are passed by reference.
- This means that when you pass an object or array to a function, you are passing a reference to the original object or array, not a copy of it.

```
let x = 10; // Primitive data type (number)
function modifyValue(a) {
  a = 20; // This change does not affect the original variable
  console.log("Inside function:", a); // Output: 20
}
modifyValue(x); // Passing by value, Output: Inside function: 20
console.log("Outside function:", x); // Output: 10 (original value remains
 ↪  unchanged)

let obj = { name: "Amr", age: 25 }; // Reference data type (object)
function modifyObject(o) {
  o.age = 30; // This change affects the original object
  console.log("Inside function:", o); // Output: { name: "Amr", age: 30 }
}
modifyObject(obj); // Passing by reference, Output: Inside function: {
↪  name: "Amr", age: 30 }
console.log("Outside function:", obj); // Output: { name: "Amr", age: 30 }
↪  (original object is modified)
```

8.2 What is Shallow Copy?

- A shallow copy creates a new object or array that contains references to the same values as the original object or array.
- If the original object contains nested objects or arrays, the shallow copy will only copy the references to those nested objects, not their actual values.
- This means that changes made to the nested objects in the shallow copy will also affect the original object.
- In JavaScript, you can create a shallow copy of an object using the spread operator (...) or the `Object.assign()` method.

```
let original = {
  name: "Amr",
  age: 25,
  hobbies: ["reading", "coding"],
};
// Using the spread operator to create a shallow copy
let shallowCopy = { ...original }; // Creating a shallow copy using spread
↪  operator
shallowCopy.age = 30; // Modifying a property in the shallow copy
shallowCopy.hobbies.push("gaming"); // Modifying a nested array in the
↪  shallow copy
console.log(original); // Output: { name: "Amr", age: 25, hobbies:
↪  ["reading", "coding", "gaming"] }

console.log(shallowCopy); // Output: { name: "Amr", age: 30, hobbies:
↪  ["reading", "coding", "gaming"] }

// Using Object.assign() to create a shallow copy
let anotherShallowCopy = Object.assign({}, original); // Creating a shallow
↪  copy using Object.assign()
anotherShallowCopy.name = "Nada"; // Modifying a property in the another
↪  shallow copy
console.log(original); // Output: { name: "Amr", age: 25, hobbies:
↪  ["reading", "coding", "gaming"] }
console.log(anotherShallowCopy); // Output: { name: "Nada", age: 25,
↪  hobbies: ["reading", "coding", "gaming"] }
```

- Shallow copy use top-level properties only, not nested objects or arrays.
- If you modify a nested object or array in the shallow copy, it will also affect the original object.

```
let original = {
  name: "Amr",
  age: 25,
  hobbies: ["reading", "coding"],
  son: {
    name: "Omar",
    age: 13,
    dog: {
      name: "Max",
      age: 5,
      isAnimal: true,
    },
  },
};
// Using the spread operator to create a shallow copy
let shallowCopy = { ...original }; // Creating a shallow copy using spread
↪   operator
shallowCopy.son.dog.name = "Jack"; // Modifying a nested object in the
↪   shallow copy
console.log(original.son.dog.name); // Output: Jack (original object is
↪   affected)
console.log(shallowCopy.son.dog.name); // Output: Jack (shallow copy is
↪   affected)
```

9 What is Deep Copy?

- A deep copy creates a new object or array that is a complete copy of the original object or array, including all nested objects and arrays.
- Changes made to the deep copy do not affect the original object, and vice versa.
- In JavaScript, you can create a deep copy using libraries like `lodash` or by using JSON methods (`JSON.stringify()` and `JSON.parse()`).

```
let original = {
  name: "Amr",
  age: 25,
  hobbies: ["reading", "coding"],
  son: {
    name: "Omar",
    age: 13,
    dog: {
      name: "Max",
      age: 5,
      isAnimal: true,
    },
  },
};
// Using JSON methods to create a deep copy
let deepCopy = JSON.parse(JSON.stringify(original)); // Creating a deep
↪   copy using JSON methods
deepCopy.son.dog.name = "Jack"; // Modifying a nested object in the deep
↪   copy
```

```
console.log(original.son.dog.name); // Output: Max (original object is not
↪ affected)
console.log(deepCopy.son.dog.name); // Output: Jack (deep copy is
↪ affected)
```

9.0.1 What is lodash?

- Lodash is a JavaScript utility library that provides helpful functions for working with arrays, objects, and other data types.
- It includes functions for deep cloning objects, manipulating arrays, and more.
- To use Lodash, you need to install it via npm or include it in your project.

```
npm install lodash
```

- Example of using Lodash for deep cloning:

```
const _ = require("lodash"); // Importing Lodash
let original = {
  name: "Amr",
  age: 25,
  hobbies: ["reading", "coding"],
  son: {
    name: "Omar",
    age: 13,
    dog: {
      name: "Max",
      age: 5,
      isAnimal: true,
    },
  },
};
// Using Lodash to create a deep copy
let deepCopy = _.cloneDeep(original); // Creating a deep copy using Lodash
deepCopy.son.dog.name = "Jack"; // Modifying a nested object in the deep
↪ copy
console.log(original.son.dog.name); // Output: Max (original object is not
↪ affected
```

9.0.2 What are the methods of Lodash?

- Lodash provides a wide range of methods for manipulating arrays, objects, and other data types. Some commonly used methods include:
  - `_.cloneDeep(value)`: Creates a deep copy of the value.
  - `_.merge(object, sources)`: Merges properties from source objects into the target object.
  - `_.map(collection, iteratee)`: Creates an array of values by running each element in the collection through the iteratee function.
  - `_.filter(collection, predicate)`: Creates an array of elements that pass the predicate function.
  - `_.find(collection, predicate)`: Returns the first element in the collection that satisfies the predicate function.

- For a complete list of Lodash methods and their usage, you can refer to the Lodash documentation.
- Lodash is a powerful library that can simplify many common programming tasks in JavaScript, especially when dealing with complex data structures.

---

10 Arrays (List)

- Arrays are ordered collections of values, where each value can be of any data type.
- Arrays are a special type of object in JavaScript, and they can hold multiple values in a single variable.
- Arrays are **zero-indexed**, meaning the first element is at index 0, the second at index 1, and so on.
- You can create an array using square brackets `[]` or the `Array` constructor.
- Type of an array is **`object`**, but it is a special type of object that has additional properties and methods for working with ordered collections.

```
let fruits = ["apple", "banana", "cherry"]; // Creating an array using
↪  square brackets
let numbers = new Array(1, 2, 3, 4, 5); // Creating an array using the
↪  Array constructor
console.log(fruits); // Output: ["apple", "banana", "cherry"]
console.log(numbers); // Output: [1, 2, 3, 4, 5]
```

- You can access array elements using their index, and you can modify them as needed.

```
fruits[0] = "orange"; // Modifying the first element
console.log(fruits[0]); // Output: "orange"
console.log(fruits.length); // Output: 3 (length of the array)
console.log(fruits[fruits.length - 1]); // Output: "cherry"
console.log(typeof fruits); // Output: "object" (arrays are a type of
↪  object)

console.log(fruits[5]); // Output: undefined (index out of bounds)
```

- You can also use the `length` property to get the number of elements in the array.

```
let fruits = ["apple", "banana", "cherry"];
console.log(fruits.length); // Output: 3 (length of the array)

//  We can empty the array by setting its length to 0:

fruits.length = 0; // Emptying the array (Better Performance)
console.log(fruits); // Output: [] (empty array)
```

- We can use the Shallow Copy to copy the array, but it will not work with nested arrays.

```
let originalArray = ["apple", "banana", "cherry"];
let shallowCopyArray = [...originalArray]; // Creating a shallow copy
↪  using spread operator
```

```
shallowCopyArray[0] = "orange"; // Modifying the first element in the
↪ shallow copy
console.log(originalArray); // Output: ["apple", "banana", "cherry"]
↪ (original array is not affected)
console.log(shallowCopyArray); // Output: ["orange", "banana", "cherry"]
```

- If the array contains nested arrays, the shallow copy will only copy the references to those nested arrays, not their actual values.

```
let originalArray = [
  ["apple", "banana"],
  ["cherry", "date"],
];
let shallowCopyArray = [...originalArray]; // Creating a shallow copy
↪ using spread operator
shallowCopyArray[0][0] = "orange"; // Modifying a nested element in the
↪ shallow copy
console.log(originalArray); // Output: [["orange", "banana"], ["cherry",
↪ "date"]] (original array is affected)
console.log(shallowCopyArray); // Output: [["orange", "banana"],
↪ ["cherry", "date"]]
```

- To create a deep copy of an array, you can use the `JSON.stringify()` and `JSON.parse()` methods or libraries like Lodash.

```
let originalArray = [
  ["apple", "banana"],
  ["cherry", "date"],
];
let deepCopyArray = JSON.parse(JSON.stringify(originalArray)); // Creating
↪ a deep copy using JSON methods
deepCopyArray[0][0] = "orange"; // Modifying a nested element in the deep
↪ copy
console.log(originalArray); // Output: [["apple", "banana"], ["cherry",
↪ "date"]] (original array is not affected)
console.log(deepCopyArray); // Output: [["orange", "banana"], ["cherry",
↪ "date"]]
```

---

11 Array Methods

- When we declare function inside the array or object, it is called a **method**.
- Arrays in JavaScript come with a variety of built-in methods that allow you to manipulate and work with the array data easily.
- Here are some commonly used array methods:

11.1 Array Methods

1. **push()**: Adds one or more elements to the end of an array and returns the new length of the array.

```
let fruits = ["apple", "banana"];
fruits.push("cherry"); // Adds "cherry" to the end
console.log(fruits); // Output: ["apple", "banana", "cherry"]
```

2. **pop()**: Removes the last element from an array and returns that element.

   ```
   let fruits = ["apple", "banana", "cherry"];
   let lastFruit = fruits.pop(); // Removes "cherry"
   console.log(fruits); // Output: ["apple", "banana"]
   console.log(lastFruit); // Output: "cherry"
   ```

3. **shift()**: Removes the first element from an array and returns that element.

   ```
   let fruits = ["apple", "banana", "cherry"];
   let firstFruit = fruits.shift(); // Removes "apple"
   console.log(fruits); // Output: ["banana", "cherry"]
   console.log(firstFruit); // Output: "apple"
   ```

4. **unshift()**: Adds one or more elements to the beginning of an array and returns the new length of the array.

   ```
   let fruits = ["banana", "cherry"];
   fruits.unshift("apple"); // Adds "apple" to the beginning
   console.log(fruits); // Output: ["apple", "banana", "cherry"]
   ```

5. **splice()**: Changes the contents of an array by removing or replacing existing elements and/or adding new elements in place.

   - Syntax: `array.splice(start, deleteCount, item1, item2, ...)`
   - Example:

   ```
   let fruits = ["apple", "banana", "cherry"];
   fruits.splice(1, 1, "orange", "grape"); // Removes "banana" and adds
   ↪  "orange" and "grape"
   console.log(fruits); // Output: ["apple", "orange", "grape",
   ↪  "cherry"]

   <!-- Addition -->
    fruits.splice(2, 0, "kiwi"); // Adds "kiwi" at index 2 without
   ↪  removing any elements
    console.log(fruits); // Output: ["apple", "orange", "kiwi", "grape",
    ↪  "cherry"]

    <!-- Deletion -->
    fruits.splice(1, 2); // Removes "orange" and "kiwi"
    console.log(fruits); // Output: ["apple", "grape", "cherry"]
    <!-- Replacing -->
    fruits.splice(1, 1, "mango"); // Replaces "grape" with "mango"
    console.log(fruits); // Output: ["apple", "mango", "cherry
   ```

6. **slice()**: Returns a shallow copy of a portion of an array into a new array object.

   - Syntax: `array.slice(start, end)`
   - Example:

```
let fruits = ["apple", "banana", "cherry", "date"];
let slicedFruits = fruits.slice(1, 3); // Returns elements from index
↪   1 to 2
console.log(slicedFruits); // Output: ["banana", "cherry"]
```

7. **concat()**: Merges two or more arrays and returns a new array.
   - Example:

```
let fruits1 = ["apple", "banana"];
let fruits2 = ["cherry", "date"];
let mergedFruits = fruits1.concat(fruits2);
console.log(mergedFruits); // Output: ["apple", "banana", "cherry",
↪   "date"]
```

8. **indexOf()**: Returns the first index at which a given element can be found in the array, or -1 if it is not present.
   - Syntax: `array.indexOf(element, fromIndex)`
   - Example:

```
let fruits = ["apple", "banana", "cherry"];
let index = fruits.indexOf("banana"); // Returns the index of
↪   "banana"
let notFoundIndex = fruits.indexOf("orange"); // Returns -1 (not
↪   found)
console.log(index); // Output: 1
console.log(notFoundIndex); // Output: -1
```

9. **includes()**: Determines whether an array includes a certain element, returning `true` or `false`.
   - Syntax: `array.includes(element, fromIndex)`
   - Example:

```
let fruits = ["apple", "banana", "cherry"];
let hasBanana = fruits.includes("banana"); // Returns true
let hasOrange = fruits.includes("orange"); // Returns false
console.log(hasBanana); // Output: true
console.log(hasOrange); // Output: false
```

10. **forEach()**: Executes a provided function once for each array element.
    - Syntax: `array.forEach(callback(currentValue, index, array))`
    - Example:

```
let fruits = ["apple", "banana", "cherry"];
fruits.forEach(function (fruit, index) {
  console.log(index + ": " + fruit);
});
// Output:
// 0: apple
// 1: banana
// 2: cherry
```

11. **map()**: Creates a new array populated with the results of calling a provided function on every element in the calling array.
    - Syntax: `array.map(callback(currentValue, index, array))`
    - Example:

    ```
    let fruits = ["apple", "banana", "cherry"];
    let fruitLengths = fruits.map(function (fruit) {
      return fruit.length;
    });
    console.log(fruitLengths); // Output: [5, 6, 6]
    ```

12. **filter()**: Creates a new array with all elements that pass the test implemented by the provided function.
    - Syntax: `array.filter(callback(currentValue, index, array))`
    - Example:

    ```
    let fruits = ["apple", "banana", "cherry"];
    let filteredFruits = fruits.filter(function (fruit) {
      return fruit.startsWith("b");
    });
    console.log(filteredFruits); // Output: ["banana"]
    ```

13. **find()**: Returns the value of the first element in the array that satisfies the provided testing function, or `undefined` if no elements satisfy the condition.
    - Syntax: `array.find(callback(currentValue, index, array))`
    - Example:

    ```
    let fruits = ["apple", "banana", "cherry"];
    let foundFruit = fruits.find(function (fruit) {
      return fruit.startsWith("b");
    });
    console.log(foundFruit); // Output: "banana"
    ```

14. **findIndex()**: Returns the index of the first element in the array that satisfies the provided testing function, or -1 if no elements satisfy the condition.
    - Syntax: `array.findIndex(callback(currentValue, index, array))`
    - Example:

    ```
    let fruits = ["apple", "banana", "cherry"];
    let foundIndex = fruits.findIndex(function (fruit) {
      return fruit.startsWith("b");
    });
    console.log(foundIndex); // Output: 1
    ```

15. **sort()**: Sorts the elements of an array in place and returns the sorted array.
    - Syntax: `array.sort(compareFunction)`
    - Example:

    ```
    let fruits = ["banana", "apple", "cherry"];
    fruits.sort();
    console.log(fruits); // Output: ["apple", "banana", "cherry"]
    ```

16. **reverse()**: Reverses the elements of an array in place and returns the reversed array.
    - Example:

    ```
    let fruits = ["apple", "banana", "cherry"];
    fruits.reverse();
    console.log(fruits); // Output: ["cherry", "banana", "apple"]
    ```

17. **reduce()**: Executes a reducer function on each element of the array, resulting in a single output value.
    - Syntax: `array.reduce(callback(accumulator, currentValue, index, array), initialValue)`
    - Example:

    ```
    let numbers = [1, 2, 3, 4, 5];
    let sum = numbers.reduce(function (accumulator, currentValue) {
      return accumulator + currentValue;
    }, 0);
    console.log(sum); // Output: 15
    ```

18. **reduceRight()**: Similar to `reduce()`, but processes the array from right to left.
    - Syntax: `array.reduceRight(callback(accumulator, currentValue, index, array), initialValue)`
    - Example:

    ```
    let numbers = [1, 2, 3, 4, 5];
    let reversedSum = numbers.reduceRight(function (accumulator,
    ↪ currentValue) {
      return accumulator + currentValue;
    }, 0);
    console.log(reversedSum); // Output: 15
    ```

19. **every()**: Tests whether all elements in the array pass the test implemented by the provided function.
    - Syntax: `array.every(callback(currentValue, index, array))`
    - Example:

    ```
    let numbers = [2, 4, 6, 8];
    let allEven = numbers.every(function (num) {
      return num % 2 === 0;
    });
    console.log(allEven); // Output: true
    ```

20. **some()**: Tests whether at least one element in the array passes the test implemented by the provided function.
    - Syntax: `array.some(callback(currentValue, index, array))`
    - Example:

    ```
    let numbers = [1, 2, 3, 4, 5];
    let hasOdd = numbers.some(function (num) {
      return num % 2 !== 0;
    ```

```
});
console.log(hasOdd); // Output: true
```

21. **fill()**: Fills all elements of an array from a start index to an end index with a static value.
    - Syntax: `array.fill(value, start, end)`
    - Example:

    ```
    let numbers = [1, 2, 3, 4, 5];
    numbers.fill(0, 2, 4);
    console.log(numbers); // Output: [1, 2, 0, 0, 5]
    ```

22. **flat()**: Creates a new array with all sub-array elements concatenated into it recursively up to the specified depth.
    - Syntax: `array.flat(depth)`
    - Example:

    ```
    let nestedArray = [1, 2, [3, 4, [5, 6]]];
    let flatArray = nestedArray.flat(2); // Flattens the array to depth 2
    console.log(flatArray); // Output: [1, 2, 3, 4, 5, 6]
    ```

23. **flatMap()**: First maps each element using a mapping function, then flattens the result into a new array.
    - Syntax: `array.flatMap(callback(currentValue, index, array))`
    - Example:

    ```
    let numbers = [1, 2, 3, 4, 5];
    let squared = numbers.flatMap(function (num) {
      return [num * num];
    });
    console.log(squared); // Output: [1, 4, 9, 16, 25]
    ```

24. **toString()**: Converts the array to a string representation.
    - Example:

    ```
    let fruits = ["apple", "banana", "cherry"];
    let fruitsString = fruits.toString();
    console.log(fruitsString); // Output: "apple,banana,cherry"
    ```

25. **toLocaleString()**: Converts the array to a string representation, using locale-specific formatting.
    - Example:

    ```
    let numbers = [1, 2, 3, 4, 5];
    let numbersLocaleString = numbers.toLocaleString();
    console.log(numbersLocaleString); // Output: "1,2,3,4,5" (format may
     ↪  vary based on locale)
    ```

26. **at()**: Returns the element at the specified index, allowing for negative indexing.
    - Syntax: `array.at(index)`
    - Example:

```
let fruits = ["apple", "banana", "cherry"];
let firstFruit = fruits.at(0);
let lastFruit = fruits.at(-1);
console.log(firstFruit); // Output: "apple"
console.log(lastFruit); // Output: "cherry"
```

27. **findLast()**: Returns the value of the last element in the array that satisfies the provided testing function, or `undefined` if no elements satisfy the condition.

    - Syntax: `array.findLast(callback(currentValue, index, array))`
    - Example:

    ```
    let fruits = ["apple", "banana", "cherry", "banana"];
    let lastBanana = fruits.findLast(function (fruit) {
      return fruit === "banana";
    });
    console.log(lastBanana); // Output: "banana"
    ```

28. **findLastIndex()**: Returns the index of the last element in the array that satisfies the provided testing function, or -1 if no elements satisfy the condition.

    - Syntax: `array.findLastIndex(callback(currentValue, index, array))`
    - Example:

    ```
    let fruits = ["apple", "banana", "cherry", "banana"];
    let lastBananaIndex = fruits.findLastIndex(function (fruit) {
      return fruit === "banana";
    });
    console.log(lastBananaIndex); // Output: 3
    ```

29. **copyWithin()**: Shallow copies part of an array to another location in the same array and returns it, without modifying its length.

    - Syntax: `array.copyWithin(target, start, end)`
    - Example:

    ```
    let numbers = [1, 2, 3, 4, 5];
    numbers.copyWithin(0, 3);
    console.log(numbers); // Output: [4, 5, 3, 4, 5]
    ```

30. **entries()**: Returns a new Array Iterator object that contains the key/value pairs for each index in the array.

    - Example:

    ```
    let fruits = ["apple", "banana", "cherry"];
    let iterator = fruits.entries();
    for (let entry of iterator) {
      console.log(entry); // Output: [0, "apple"], [1, "banana"], [2,
       ↪  "cherry"]
    }
    ```

31. **keys()**: Returns a new Array Iterator object that contains the keys for each index in the array.

    - Example:

```
let fruits = ["apple", "banana", "cherry"];
let iterator = fruits.keys();
for (let key of iterator) {
  console.log(key); // Output: 0, 1, 2
}
```

32. **values()**: Returns a new Array Iterator object that contains the values for each index in the array.

    - Example:

```
let fruits = ["apple", "banana", "cherry"];
let iterator = fruits.values();
for (let value of iterator) {
  console.log(value); // Output: "apple", "banana", "cherry"
}
```

33. **isArray()**: Determines whether the provided value is an array.

    - Syntax: `Array.isArray(value)`
    - Example:

```
let fruits = ["apple", "banana", "cherry"];
let isArray = Array.isArray(fruits); // Returns true
let isNotArray = Array.isArray("apple"); // Returns false
console.log(isArray); // Output: true
console.log(isNotArray); // Output: false
```

34. **from()**: Creates a new array from an array-like or iterable object.

    - Syntax: `Array.from(arrayLike, mapFn, thisArg)`
    - Example:

```
let str = "hello";
let charArray = Array.from(str); // Creates an array from a string
console.log(charArray); // Output: ["h", "e", "l", "l", "o"]
```

35. **join()**: Joins all elements of an array into a string, separated by a specified separator.

    - Syntax: `array.join(separator)`
    - Example:

```
let fruits = ["apple", "banana", "cherry"];
let fruitString = fruits.join(", "); // Joins elements with ", "
console.log(fruitString); // Output: "apple, banana, cherry"
```

- **From and Join are used to convert an array to a string or an array-like object to an array.**

36. **of()**: Creates a new array with a variable number of arguments, regardless of the number or type of arguments.

    - Syntax: `Array.of(element1, element2, ..., elementN)`
    - Example:

```
let numbers = Array.of(1, 2, 3, 4, 5); // Creates an array with the
↪  specified elements
```

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

37. **groupBy()**: Groups the elements of an array based on a specified criterion.
    - Syntax: `array.groupBy(callback(currentValue, index, array))`
    - Example:

```
let fruits = ["apple", "banana", "cherry", "date"];
let groupedFruits = fruits.groupBy(function (fruit) {
  return fruit[0]; // Grouping by the first letter of each fruit
});
console.log(groupedFruits);
// Output: { a: ["apple"], b: ["banana"], c: ["cherry"], d: ["date"]
↪  }
```

---

12 Arithmetic Operators

- Arithmetic operators are used to perform mathematical operations on numbers.
- The basic arithmetic operators in JavaScript are:
  - Addition (`+`)
  - Subtraction (`-`)
  - Multiplication (`*`)
  - Division (`/`)
  - Modulus (`%`)
    - Calculates the remainder of a division operation.
    - If the first number is larger than the second number, it returns the remainder of the division.
    - For example, `5 % 2` returns `1` because when `5` is divided by `2`, the remainder is `1`.
    - If the first number is smaller than the second number, it returns the first number.
    - For example, `11 % 20` returns `11` because when `11` is divided by `20`, the remainder is `11`.
    - If the first number is equal to the second number, it returns `0`.
    - For example, `10 % 10` returns `0` because when `10` is divided by `10`, the remainder is `0`.
  - Exponentiation (`**`)
- Here are some examples of using arithmetic operators:

```
let a = 10;
let b = 5;
let sum = a + b; // Addition
let difference = a - b; // Subtraction
let product = a * b; // Multiplication
let quotient = a / b; // Division
let remainder = a % b; // Modulus
let power = a ** b; // Exponentiation
console.log("Sum:", sum); // Output: Sum: 15
console.log("Difference:", difference); // Output: Difference: 5
console.log("Product:", product); // Output: Product: 50
console.log("Quotient:", quotient); // Output: Quotient: 2
console.log("Remainder:", remainder); // Output: Remainder: 0
console.log("Power:", power); // Output: Power: 100000
```

12.1 What is the difference between Pre Increment ++ and Post Increment ++?

- Pre-increment (`++variable`) increments the variable's value before using it in an expression, while post-increment (`variable++`) increments the variable's value after using it in an expression.
- Pre-increment Example:

```
let x = 5;
let preIncrement = ++x; // Pre-increment: x is incremented to 6 before
↪  assignment
console.log(preIncrement); // Output: 6
```

- Post-increment Example:

```
let y = 5;
let postIncrement = y++; // Post-increment: y is assigned to postIncrement
↪  before incrementing
console.log(postIncrement); // Output: 5
console.log(y); // Output: 6 (y is incremented after assignment)
```

- **Another Tricky Example:**

```
let x = 20;

x++; // Post-increment: x is incremented after this line

++x; // Pre-increment: x is incremented to 21 before the next line

x += 20; // x is incremented by 20, making it 41

x = 2; // x is set to 2
console.log(++x); // Output: 3 (Pre-increment: x is incremented to 3
↪  before logging)
console.log(x++); // Output: 3 (Post-increment: x is logged as 3 before
↪  incrementing)
console.log(x); // Output: 4 (x is incremented to 4 after the
```

- In summary, pre-increment increments the value before using it, while post-increment uses the current value and then increments it.

13 Concatenation and Casting

- Concatenation is the process of joining two or more strings together.
- In JavaScript, you can concatenate strings using the + operator or the `concat()` method.
- Here are some examples of string concatenation:

```
let str1 = "Hello";
let str2 = "World";
let concatenated = str1 + " " + str2; // Using the + operator
console.log(concatenated); // Output: "Hello World"
let concatenatedMethod = str1.concat(" ", str2); // Using the concat()
↪   method
console.log(concatenatedMethod); // Output: "Hello World"
```

- When concatenating strings with other data types, JavaScript will automatically convert the non-string values to strings.

```
let num = 42;
let bool = true;
let concatenatedWithNumber = str1 + " " + num; // Concatenating string with
↪   a number
console.log(concatenatedWithNumber); // Output: "Hello 42"
```

- Tricky Example:

```
let result = 10 + 20 + "30"; // Adding numbers first, then concatenating
↪   with a string
console.log(result); // Output: "3030" (10 + 20 = 30, then "30" is
↪   concatenated)

let trickyResult = "10" + 20 + 30; // Concatenating a string with numbers
console.log(trickyResult); // Output: "102030" ("10" is a string, so 20
↪   and 30 are converted to strings and concatenated)

let trickyResult2 = 10 + "20" + 30; // Concatenating a number with a
↪   string and then a number
console.log(trickyResult2); // Output: "102030" (10 is added to "20",
↪   resulting in "1020", then 30 is concatenated)

let trickyResult3 = 4 + 4 + +"4"; // Using unary plus to convert "4" to a
↪   number
console.log(trickyResult3); // Output: 12 (4 + 4 + 4 = 12)
let trickyResult4 = 4 + 4 + -"4"; // Using unary minus to convert "4" to a
↪   negative number
console.log(trickyResult4); // Output: 8 (4 + 4 - 4 = 8)
```

- What if we do this:

```
let trickyResult5 = 4 + 4 + +"Amr"; // Using unary plus on a non-numeric
↪   string
console.log(trickyResult5); // Output: NaN (Not a Number, because "Amr"
↪   cannot be converted to a number)
```

- Typeof `NaN` is `number`, but it represents an **invalid number**.

- This is a common source of confusion in JavaScript, as `NaN` is a special value that indicates an operation that does not yield a valid number.

13.1 What is Casting?

- Casting is the process of converting a value from one data type to another.
- In JavaScript, you can cast values using functions like `String()`, `Number()`, and `Boolean()`.
- Here are some examples of casting:

```
let num = 42;
let str = String(num); // Casting number to string
console.log(str); // Output: "42"
let bool = Boolean(num); // Casting number to boolean
console.log(bool); // Output: true
let str2 = "true";
let bool2 = Boolean(str2); // Casting string to boolean
console.log(bool2); // Output: true (non-empty string is truthy)
```

- Using +““ to cast a string to a number:
- This is also is a common technique to convert a string to a number in JavaScript Which is called **Casting Using Unary Plus**.

```
let str3 = "42";
let num2 = +str3; // Using unary plus to cast string to number
console.log(num2); // Output: 42
let str4 = "Hello";
let num3 = +str4; // Using unary plus on a non-numeric string
console.log(num3); // Output: NaN (Not a Number, because "Hello" cannot be
↪  converted to a number)
let str5 = "123abc";
let num4 = +str5; // Using unary plus on a string with non-numeric
↪  characters
console.log(num4); // Output: NaN (Not a Number, because "123abc" cannot
↪  be converted to a number)
```

- Using **Number()** to cast a string to a number:

```
let str6 = "42";
let num5 = Number(str6); // Using Number() to cast string to number
console.log(num5); // Output: 42
let str7 = "Hello";
let num6 = Number(str7); // Using Number() on a non-numeric string
console.log(num6); // Output: NaN (Not a Number, because "Hello" cannot be
↪  converted to a number)
let str8 = "123abc";
let num7 = Number(str8); // Using Number() on a string with non-numeric
↪  characters
console.log(num7); // Output: NaN (Not a Number, because "123abc" cannot
↪  be converted to a number)
```

13.2 What is the difference between Number() parseInt() and parseFloat()?

- `Number()` converts a value to a number, while `parseInt()` and `parseFloat()` parse a string and return an integer or float, respectively.
- `Number()` can convert various data types to numbers, while `parseInt()` and `parseFloat()` are specifically designed for parsing strings.
- `parseInt()` parses a string and returns an integer, while `parseFloat()` parses a string and returns a floating-point number.
- `parseInt()` stops parsing when it encounters a non-numeric character, while `parseFloat()` can handle decimal points and stops parsing at the first non-numeric character after the decimal point.
- Using **parseInt()** and **parseFloat()** to cast strings to numbers:

```
let str9 = "42.5";
let intNum = parseInt(str9); // Using parseInt() to cast string to integer
console.log(intNum); // Output: 42 (parses the integer part)
let floatNum = parseFloat(str9); // Using parseFloat() to cast string to
↪ float
console.log(floatNum); // Output: 42.5 (parses the float part)

let str10 = "123abc";
let intNum2 = parseInt(str10); // Using parseInt() on a string with
↪ non-numeric characters
console.log(intNum2); // Output: 123 (parses until it encounters a
↪ non-numeric character)

let num = Number(str10); // Using Number() on a string with non-numeric
↪ characters
console.log(num); // Output: NaN (Not a Number, because "123abc" cannot be
↪ converted to a number)

let floatNum2 = parseFloat(str10); // Using parseFloat() on a string with
↪ non-numeric characters
console.log(floatNum2); // Output: 123 (parses until it encounters a
↪ non-numeric character)
```

---

14 Comparison Operators

- Comparison operators are used to compare two values and return a boolean result (`true` or `false`).
- The basic comparison operators in JavaScript are:
  - Equal to (==)
  - Not equal to (!=)
  - Strict equal to (===)
  - Strict not equal to (!==)
  - Greater than (>)

  - Less than (<)
  - Greater than or equal to (>=)
  - Less than or equal to (<=)
- Here are some examples of using comparison operators:

```
let a = 10;
let b = 5;
console.log(a == b); // Output: false
console.log(a != b); // Output: true
console.log(a === b); // Output: false
console.log(a !== b); // Output: true
console.log(a > b); // Output: true
console.log(a < b); // Output: false
console.log(a >= b); // Output: true
console.log(a <= b); // Output: false
let c = "10";
console.log(a == c); // Output: true (type coercion occurs)
console.log(a === c); // Output: false (no type coercion, different types)
```

15 What is the difference between == and ===?

- == is the equality operator that checks for value equality, allowing type coercion.
- === is the strict equality operator that checks for both value and type equality, without allowing type coercion.
- Here are some examples to illustrate the difference:

```
let a = 5;
let b = "5";
console.log(a == b); // Output: true (value is equal, type coercion
↪  occurs)
console.log(a === b); // Output: false (value is equal, but types are
↪  different
let c = 10;
let d = 10;
console.log(c == d); // Output: true (both values are equal)
console.log(c === d); // Output: true (both values and types are equal)
let e = null;
let f = undefined;
console.log(e == f); // Output: true (null and undefined are loosely
↪  equal)
console.log(e === f); // Output: false (null and undefined are not
↪  strictly equal)
let g = 0;
let h = false;
console.log(g == h); // Output: true (0 is loosely equal to false)
console.log(g === h); // Output: false (0 and false are not strictly
↪  equal)
```

16 Logical Operators

- Logical operators are used to combine or negate boolean values.
- The basic logical operators in JavaScript are:
  - Logical AND (`&&`)
  - Logical OR (`||`)
  - Logical NOT (`!`)
- Here are some examples of using logical operators:

```javascript
let a = true;
let b = false;
console.log(a && b); // Output: false (both must be true for AND)
console.log(a || b); // Output: true (at least one must be true for OR)
console.log(!a); // Output: false (NOT negates the value)
console.log(!b); // Output: true (NOT negates the value)

let c = 5;
let d = 10;
console.log(c > 0 && d < 20); // Output: true (both conditions are true)
console.log(c < 0 || d > 5); // Output: true (at least one condition is
↪  true)
console.log(!(c === d)); // Output: true (NOT negates the equality check)
```

---

17 If Conditions and Ternary Operator

17.1 If Conditions

- If conditions are used to execute a block of code based on a specified condition.
- The basic syntax of an if condition is:

```javascript
if (condition) {
  // Code to execute if condition is true
} else if (anotherCondition) {
  // Code to execute if anotherCondition is true
} else {
  // Code to execute if none of the conditions are true
}
```

- Here are some examples of using if conditions With else if:

```javascript
let age = 18;
if (age < 18) {
  console.log("You are a minor.");
} else if (age >= 18 && age < 65) {
  console.log("You are an adult.");
} else {
  console.log("You are a senior citizen.");
}
```

- In this example, the code checks the value of `age` and prints a message based on the age range.

17.2 Ternary Operator

- The ternary operator is a shorthand way to write an if-else statement.
- The syntax of the ternary operator is:

```
condition ? expressionIfTrue : expressionIfFalse;
```

- Here are some examples of using the ternary operator:

```
let age = 18;
let message = age >= 18 ? "You are an adult." : "You are a minor.";
console.log(message); // Output: "You are an adult."
let isLoggedIn = true;
let loginMessage = isLoggedIn ? "Welcome back!" : "Please log in.";
console.log(loginMessage); // Output: "Welcome back!"
```

17.2.1 Handling Else if with Ternary Operator

- The ternary operator can also be used to handle multiple conditions, but it can become less readable with many conditions.

```
let score = 85;
let grade = score >= 90 ? "A" : score >= 80 ? "B" : score >= 70 ? "C" :
↪  "D";
console.log(grade); // Output: "B"
```

- In this example, the code checks the value of `score` and assigns a grade based on the score range using nested ternary operators.

- Age Example with Nested Ternary Operator :

```
let age = 18;
let ageCategory =
  age < 18
    ? "You are a minor."
    : age >= 18 && age < 65
    ? "You are an adult."
    : "You are a senior citizen.";
console.log(ageCategory); // Output: "You are an adult."
```

- In this example, the code checks the value of `age` and assigns a message based on the age range using nested ternary operators.

18 Switch Case

- The switch case statement is used to execute one block of code among multiple options based on the value of a variable or expression.
- The basic syntax of a switch case statement is:

```
switch (expression) {
  case value1:
    // Code to execute if expression matches value1
    break;
  case value2:
    // Code to execute if expression matches value2
    break;
  // Add more cases as needed
  default:
  // Code to execute if no cases match
}
```

- Here are some examples of using switch case statements:

```
let day = 3;
switch (day) {
  case 1:
    console.log("Monday");
    break;
  case 2:
    console.log("Tuesday");
    break;
  case 3:
    console.log("Wednesday");
    break;
  case 4:
    console.log("Thursday");
    break;
  case 5:
    console.log("Friday");
    break;
  case 6:
    console.log("Saturday");
    break;
  case 7:
    console.log("Sunday");
    break;
  default:
    console.log("Invalid day");
}
```

- What if we remove one break from the switch case?

```
let day = 2;
switch (day) {
  case 1:
    console.log("Monday");
    break;
  case 2:
    console.log("Tuesday");
  // No break here, so it will fall through to the next case
  case 3:
    console.log("Wednesday");
    break;
  case 4:
    console.log("Thursday");
    break;
  case 5:
    console.log("Friday");
    break;
  case 6:
    console.log("Saturday");
    break;
  case 7:
    console.log("Sunday");
    break;
  default:
    console.log("Invalid day");
}
// Output:
// "Tuesday"
// "Wednesday" (because there is no break after case 2, it falls through
 ↪  to case 3)
```

- What happen if we remove the break from the last case?

```
let day = 7;
switch (day) {
  case 1:
    console.log("Monday");
    break;
  case 2:
    console.log("Tuesday");
    break;
  case 3:
    console.log("Wednesday");
    break;
  case 4:
    console.log("Thursday");
    break;
  case 5:
    console.log("Friday");
    break;
  case 6:
    console.log("Saturday");
    break;
  case 7:
    console.log("Sunday");
  // No break here, but it's the last case, so it won't affect anything
  default:
    console.log("Invalid day");
}
// Output: "Sunday"
// "Invalid day" (because there is no break after case 7, it falls through
 ↪  to the default case)
```

19 Truthy and Falsy Values

- In JavaScript, values can be categorized as either truthy or falsy based on their boolean representation.
- **Truthy values** are values that evaluate to `true` in a boolean context, while **falsy values** are values that evaluate to `false`.
- The following values are considered **falsy** in JavaScript:
  - `false`
  - `0` (zero)
  - `""` (empty string)
  - `null`
  - `undefined`
  - `NaN` (Not a Number)
- All other values are considered **truthy**.
- Here are some examples of truthy and falsy values:

```
let truthyValue1 = "Hello"; // Non-empty string (truthy)
let truthyValue2 = 42; // Non-zero number (truthy)
let truthyValue3 = {}; // Non-empty object (truthy)
let truthyValue4 = []; // Non-empty array (truthy)
let falsyValue4 = [].length; // Falsy value (empty array length is 0)
let falsyValue1 = false; // Falsy value
let falsyValue2 = 0; // Falsy value

let falsyValue3 = ""; // Falsy value
let falsyValue4 = " "; // Non-empty string (truthy)
let falsyValue5 = null; // Falsy value
let falsyValue6 = undefined; // Falsy value
let falsyValue7 = NaN; // Falsy value

let falsyValue8 = 0n; // Falsy value (BigInt zero)
let falsyValue9 = -0; // Falsy value (negative zero)
let falsyValue10 = 0.0; // Falsy value (floating-point zero
let falsyValue11 = 0.0; // Falsy value (zero with decimal places)
let falsyValue12 = 0.0000001; // Falsy value (very small number, but still
↪   truthy)
```

---

20 Loops

- Loops are used to execute a block of code repeatedly until a specified condition is met.
- The basic types of loops in JavaScript are:
  - `for` loop
  - `while` loop
  - `do...while` loop
  - `for...in` loop (for iterating over object properties)

  - `for...of` loop (for iterating over iterable objects like arrays)
  - `forEach` method (for iterating over arrays)

20.1 For Loop

- The `for` loop is used to execute a block of code a specific number of times.
- The basic syntax of a `for` loop is:

```
for (initialization; condition; increment / decrement) {
  // Code to execute in each iteration
}
```

- Here are some examples of using a `for` loop:

```
for (let i = 0; i < 5; i++) {
  console.log("Iteration:", i);
}
// Output:
// Iteration: 0
// Iteration: 1
// Iteration: 2
// Iteration: 3
// Iteration: 4
```

- You can also use a `for` loop to iterate over an array:

```
let fruits = ["apple", "banana", "cherry"];
for (let i = 0; i < fruits.length; i++) {
  console.log("Fruit:", fruits[i]);
}
// Output:
// Fruit: apple
// Fruit: banana
// Fruit: cherry

---

let fruits = ["apple", "banana", "cherry"];
for (let i = 0; i < fruits.length; ) {
  i++; // Incrementing the index
  console.log("Fruit:", fruits[i]);
}
// Output:
// Fruit: banana
// Fruit: cherry
// Fruit: Undefined

// Note: This will skip the first fruit (apple) because the index is
↪  incremented before logging.

---
```

```
let fruits = ["apple", "banana", "cherry"];
let i = 0;
for (; i < fruits.length; i++) {
  console.log("Fruit:", fruits[i]);
}
// Output:
// Fruit: apple
// Fruit: banana
// Fruit: cherry


---

let fruits = ["apple", "banana", "cherry"];
let i = 0;
for (; i < fruits.length; ) {
  console.log("Fruit:", fruits[i]);
}
// Output: Infinite Loop

---

let fruits = ["apple", "banana", "cherry"];
let i = 0;
for (true; ;true ) {
  console.log("Fruit:", fruits[i]);
}
// Output: Infinite Loop
// By Default the Part of Condition is: True
```

20.2 While Loop

- The `while` loop is used to execute a block of code as long as a specified condition is true.
- The basic syntax of a `while` loop is:

```
while (condition) {
  // Code to execute in each iteration
}
```

- Here are some examples of using a `while` loop:

```
let count = 0;
while (count < 5) {
  console.log("Count:", count);
  count++;
}
// Output:
// Count: 0
// Count: 1
// Count: 2
// Count: 3
// Count: 4
```

- You can also use a `while` loop to iterate over an array:

```
let fruits = ["apple", "banana", "cherry"];
let index = 0;

while (index < fruits.length) {
  console.log("Fruit:", fruits[index]);
  index++;
}
// Output:
// Fruit: apple
// Fruit: banana
// Fruit: cherry
```

20.3 Do. . . While Loop

- The `do...while` loop is similar to the `while` loop, but it guarantees that the block of code will be executed at least once, even if the condition is false.
- The basic syntax of a `do...while` loop is:

```
do {
  // Code to execute in each iteration
} while (condition);
```

- Here are some examples of using a `do...while` loop:

```
let count = 0;
do {
  console.log("Count:", count);
  count++;
} while (count < 5);
// Output:
// Count: 0
// Count: 1
// Count: 2
// Count: 3
// Count: 4
```

- You can also use a `do...while` loop to iterate over an array:

```
let fruits = ["apple", "banana", "cherry"];
let index = 0;
do {
  console.log("Fruit:", fruits[index]);
  index++;
} while (index < fruits.length);
// Output:
// Fruit: apple
// Fruit: banana
// Fruit: cherry
```

20.4 For. . . In Loop

- The `for...in` loop is used to iterate over the properties of an object.
- The basic syntax of a `for...in` loop is:

```
for (let key in object) {
  // Code to execute for each property
}
```

- Here are some examples of using a `for...in` loop:

```
let person = {
  name: "John",
  age: 30,
  city: "New York",
};
for (let key in person) {
  console.log(key + ":", person[key]);
}
// Output:
// name: John
// age: 30
// city: New York
```

- You can also use a `for...in` loop to iterate over an array, but it's generally not recommended for arrays:

```
let fruits = ["apple", "banana", "cherry"];
for (let index in fruits) {
  console.log("Fruit:", fruits[index]);
}
// Output:
// Fruit: apple
// Fruit: banana
// Fruit: cherry
```

- Note: Using `for...in` for arrays can lead to unexpected results if the array has non-numeric properties or methods.

20.5 For. . . Of Loop

- The `for...of` loop is used to iterate over iterable objects like arrays, strings, and other collections.
- The basic syntax of a `for...of` loop is:

```
for (let element of iterable) {
  // Code to execute for each element
}
```

- Here are some examples of using a `for...of` loop:

```
let fruits = ["apple", "banana", "cherry"];
for (let fruit of fruits) {
  console.log("Fruit:", fruit);
}
```

```
// Output:
// Fruit: apple
// Fruit: banana
// Fruit: cherry
```

- You can also use a `for...of` loop to iterate over a string:

```
let str = "Hello";
for (let char of str) {
  console.log("Character:", char);
}
// Output:
// Character: H
// Character: e
// Character: l
// Character: l
// Character: o
```

- The `for...of` loop is generally preferred for iterating over arrays and other iterable objects because it provides a cleaner syntax and avoids issues with non-numeric properties.

20.6 For Each Method (Array Method)

- The `forEach` method is an array method that executes a provided function once for each array element.
- The basic syntax of the `forEach` method is:

```
array.forEach(function (element) {
  // Code to execute for each element
});
```

- Here are some examples of using the `forEach` method:

```
let fruits = ["apple", "banana", "cherry"];
fruits.forEach(function (fruit) {
  console.log("Fruit:", fruit);
});
// Output:
// Fruit: apple
// Fruit: banana
// Fruit: cherry
```

- You can also use an arrow function with the `forEach` method for a more concise syntax:

```
let fruits = ["apple", "banana", "cherry"];
fruits.forEach((fruit) => {
  console.log("Fruit:", fruit);
});
// Output:
// Fruit: apple
// Fruit: banana
// Fruit: cherry
```

- The `forEach` method is useful for performing operations on each element of an array without needing to manage the loop index manually.

21 Functions

- Functions are reusable blocks of code that can be called with a specific set of parameters to perform a task or calculate a value.
- Functions can be defined using the `function` keyword or as arrow functions.
- The name of function is preferred to be a verb, as it describes the action the function performs.

```
// Function declaration
function add(a, b) {
  return a + b;
}

// Arrow function
const subtract = (a, b) => {
  return a - b;
};
```

- Functions can take parameters and return values. They can also be called with different arguments to perform the same operation on different data.

```
// Calling the add function
let sum = add(5, 3); // sum will be 8
// Calling the subtract function
let difference = subtract(10, 4); // difference will be 6
console.log("Sum:", sum); // Output: Sum: 8
console.log("Difference:", difference); // Output: Difference: 6
```

- Functions can also be defined with default parameters, which are used if no argument is provided for that parameter.

```
function multiply(a, b = 1) {
  return a * b;
}
console.log(multiply(5, 2)); // Output: 10
console.log(multiply(5)); // Output: 5
```

- Functions can also be defined with default parameters and Optional Parameters, which are used if no argument is provided for that parameter.

```
function greet(name = "Guest", greeting = "Hello") {
  return `${greeting}, ${name}!`;
}
console.log(greet("Alice")); // Output: Hello, Alice!
console.log(greet()); // Output: Hello, Guest!
```

- Functions can also be defined as methods within objects, allowing for encapsulation of behavior related to that object.

```
let calculator = {
  add: function (a, b) {
    return a + b;
  },
  subtract: function (a, b) {
    return a - b;
  },
};
console.log(calculator.add(5, 3)); // Output: 8
console.log(calculator.subtract(10, 4)); // Output: 6
```

- Function can take many parameters without any limit, but it is recommended to keep the number of parameters manageable for better readability and maintainability.

```
function calculateTotalPrice(price, taxRate = 0.1, discount = 0) {
  let tax = price * taxRate;
  let totalPrice = price + tax - discount;
  return totalPrice;
}
console.log(calculateTotalPrice(100)); // Output: 110 (with default tax
↪  rate and no discount)
console.log(calculateTotalPrice(100, 0.2, 10)); // Output: 120 (with custom
↪  tax rate and discount)
```

- Function with Optional Parameters:

```
function greet(name, greeting = "Hello") {
  return `${greeting}, ${name}!`;
}
console.log(greet("Alice")); // Output: Hello, Alice! (using default
↪  greeting)
console.log(greet("Bob", "Hi")); // Output: Hi, Bob! (using custom
↪  greeting)
```

- Function With Objects and Destructuring:

```
let person = {
  name: "Alice",
  age: 30,
  city: null, // Optional parameter
};

function introduce({ name, age, city }) {
  return `My name is ${name}, I am ${age} years old}.`;
}
console.log(introduce(person)); // Output: My name is Alice, I am 30 years
↪  old.
```

---

22 Use Return Keyword With Function

- The `return` keyword is used to specify the value that a function should output when it is called.
- When a function reaches a `return` statement, it immediately stops executing and sends the specified value back to the caller.

- If no return statement is provided, the function will return undefined by default.

```
function square(number) {
  return number * number;
}
let result = square(5); // Calling the function and storing the result
console.log(result); // Output: 25 (the function returns the square of the
↪  number)
```

- You can also use the return keyword to exit a function early if a certain condition is met.

```
function checkEven(number) {
  if (number % 2 === 0) {
    return true; // Return true if the number is even
  }
  return false; // Return false if the number is odd
}
let isEven = checkEven(4); // Calling the function with an even number
console.log(isEven); // Output: true (the function returns true for even
↪  numbers)
```

- If a function does not have a return statement, it will return undefined by default.

```
function noReturn() {
  console.log("This function does not return anything.");
}
let result = noReturn(); // Calling the function
console.log(result);
// Output:
// This function does not return anything.
// undefined
```

- Any Function return value, We should use it to store in a variable or use it directly in an expression.

```
function add(a, b) {
  return a + b; // Returning the sum of a and b
}
let sum = add(3, 4); // Calling the function and storing the result
console.log(sum); // Output: 7 (the function returns the sum of the two
↪  numbers)
```

- You can also use the return keyword to return multiple values from a function by returning an object or an array.

```
function getCoordinates() {
  return { x: 10, y: 20 }; // Returning an object with x and y coordinates
}
let coordinates = getCoordinates(); // Calling the function and storing
↪  the result
console.log(coordinates); // Output: { x: 10, y: 20 }
```

- You can also use the return keyword to return an array of values.

```
function getNumbers() {
  return [1, 2, 3, 4, 5]; // Returning an array of numbers
}
let numbers = getNumbers(); // Calling the function and storing the result
console.log(numbers); // Output: [1, 2, 3, 4, 5]
console.log(numbers[1]); // Output: 2 (accessing the second element of the
↪   array)
```

---

23 Deal With Function Parameters and Arguments

- Function parameters are the variables defined in the function declaration that accept values when the function is called.
- Function arguments are the actual values passed to the function when it is called.

You can define multiple parameters in a function with rest parameters.

- Simple Example:

```
function test(...args) {
  console.log(args); // Output: Array of arguments passed to the function
}
test(1, 2, 3); // Calling the function with multiple arguments, which will
↪   be stored in the args array
// Output: [1, 2, 3]

test("Hello", "World"); // Calling the function with string arguments
// Output: ["Hello", "World"]
```

- You can also use the rest parameter syntax to accept an indefinite number of arguments in a function.
- This allows you to handle a variable number of arguments without explicitly defining each one.

```
function sum(...numbers) {
  return numbers.reduce((acc, curr) => acc + curr, 0);
}
let total = sum(1, 2, 3, 4, 5); // Calling the function with multiple
↪   arguments
console.log(total); // Output: 15 (the function returns the sum of all the
↪   numbers)
```

---

24 Hoisting Concept

- Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their containing scope during the compilation phase.
- This means that you can use variables and functions before they are declared in the code.
- However, only the declarations are hoisted, not the initializations. This can lead to unexpected behavior if you're not aware of how hoisting works.
- Here are some examples to illustrate hoisting:

```
console.log(x); // Output: undefined (variable declaration is hoisted, but
↪  not the initialization)
var x = 5; // Variable declaration and initialization
console.log(x); // Output: 5 (after initialization, x has the value 5)
```

- In the above example, the variable `x` is hoisted to the top of its scope, but its value is not assigned until the line where it is initialized. Therefore, when we log `x` before its initialization, it outputs `undefined`.

24.1 Hoisting with var and Let and Const

- Variables declared with `var` are hoisted, but variables declared with `let` and `const` are not hoisted in the same way. They are in a "temporal dead zone" until they are initialized.

```
console.log(a); // Output: undefined (variable declaration is hoisted, but
↪  not the initialization)
var a = 5;
console.log(a); // Output: 5 (after initialization, a has the value 5)

console.log(b); // Output: ReferenceError: Cannot access 'b' before
↪  initialization
let b = 10;

console.log(c); // Output: ReferenceError: Cannot access 'c' before
↪  initialization
const c = 15;
```

- In this example, the variable `a` is declared with `var`, so it is hoisted and outputs `undefined` before its initialization. However, the variables `b` and `c`, declared with `let` and `const`, respectively, throw a `ReferenceError` if accessed before their initialization.

- Tricky Example with Hoisting:

```
var x = 30;

function trickyFunction() {
  console.log(x); // Output: undefined (due to hoisting)
  var x = 20; // Local variable x is declared and initialized
  console.log(x); // Output: 20 (after initialization)
}
trickyFunction(); // Output:
// undefined
// 20
console.log(x); // Output: 30 (global variable x is still 30)
```

- In this example, the local variable `x` inside `trickyFunction` is hoisted to the top of the function scope, so when we log `x` before its initialization, it outputs `undefined`. After the initialization, it outputs `20`. The global variable `x` remains unchanged and outputs `30`.

```
var x = 30;

function trickyFunction() {
  console.log(x); // Output: 30 (global variable x is accessed)
  x = 20; // Local variable x is not declared, so it modifies the global
↪  variable
  console.log(x); // Output: 20 (after modification)
}
trickyFunction(); // Output: 30
console.log(x); // Output: 20 (global variable x is modified)
```

---

25 Declaration Functions and Expression Functions

- In JavaScript, functions can be defined in two main ways: function declarations and function expressions.

25.1 Function Declarations

- Function declarations are defined using the `function` keyword followed by the function name and parentheses.
- They are hoisted, meaning you can call them before they are defined in the code.

```
function greet() {
  console.log("Hello, World!");
}
greet(); // Output: "Hello, World!" (function can be called before its
↪  declaration)
```

25.2 Function Expressions

- Function expressions are defined by assigning a function to a variable. They can be anonymous (without a name) or named.

```
let greet = function () {
  console.log("Hello, World!");
};
greet(); // Output: "Hello, World!" (function can be called after its
↪  expression)
```

25.3 Hoisting with Function Declarations

- Function declarations are also hoisted, allowing you to call a function before it is defined in the code.

```
sayHello(); // Output: "Hello, World!" (function can be called before its
↪  declaration)

function sayHello() {
  console.log("Hello, World!");
}
```

- In this example, the function `sayHello` can be called before its declaration because function declarations are hoisted to the top of their scope.

25.4 Hoisting with Function Expressions

- Function expressions, however, are not hoisted in the same way as function declarations. If you try to call a function expression before it is defined, you will get an error.

```
console.log(sayGoodbye); // Output: undefined (variable declaration is
↪  hoisted, but not the initialization)
var sayGoodbye = function () {
  console.log("Goodbye, World!");
};
sayGoodbye(); // Output: "Goodbye, World!" (function can be called after
↪  its expression)
```

- In this example, the variable `sayGoodbye` is hoisted, but since it is a function expression, it is not initialized until the line where it is defined. If you try to call `sayGoodbye` before its definition, you will get a `TypeError`.
- Another example of function expression:

```
var testFunction = function () {
  console.log("This is a function expression.");
  return 10;
};
var result = testFunction(); // Output: "This is a function expression."
console.log(result); // Output: 10 (the function returns 10)
```

- Another Tricky Example of Function Expression:

```
var testFunction = function (param) {
  if (!param) return;
  console.log("This is a function expression.");
  return 10;
};
var result = testFunction(); // Output: "No parameter provided."
console.log(result); // Output: "No parameter provided."

let result2 = testFunction(5); // Output: "This is a function expression."
console.log(result2); // Output: 10 (the function returns 10)
```

- In this example, the function `testFunction` checks if a parameter is provided. If not, it returns early without executing the rest of the code. If a parameter is provided, it logs a message and returns `10`.

---

26 Immediately Invoked Function Expression (IIFE)

- It is also called **Self-Executing Anonymous Function.**
- It's a type of Expression Function that is executed immediately after it is defined.
- IIFE is a function that runs as soon as it is defined.
- It is often used to create a new scope and avoid polluting the global scope.

```
(function () {
  var x = 10;
  console.log(x); // Output: 10
})();
console.log(x); // Output: ReferenceError: x is not defined
```

- In this example, the function is defined and executed immediately, logging `10` to the console. The variable `x` is not accessible outside the IIFE, preventing it from polluting the global scope.
- Tricky Example of IIFE:

```
(function (a, b) {
  console.log(a + b); // Output: 15
})(5, 10);


function (a, b) {
    console.log(a + b); // Output: 15
  }(5, 10);
// Output: 15 (the function is executed immediately with arguments 5 and
↪  10)
```

- Note: If you remove the last semicolon at the end of first function, it will throw an error because it will be treated as a function declaration instead of an expression.

```
let x = 10(function (a, b) {
  console.log(a + b); // Output: 15
})(5, 10); // This will work fine
```

- This is the same issue and it will throw an error: **TypeError: 10 is not a function**, and this problem will happen with the 2 functions above if we ignored the semicolon.

---

27 Interview Questions for the Function and Hoisting

```
function foo() {
  function bar() {
    return 3;
  }
  return bar();
  function bar() {
    return 8;
  }
}
console.log(foo()); // Output: 8
```

- In this example, the function `bar` is declared twice within the function `foo`. However, due to hoisting, the second declaration of `bar` takes precedence, so when `foo` is called, it returns `8`.
- If you remove the second declaration of `bar`, it will return `3` instead.
- Another Example of Function and Hoisting:

```
function foo() {
  var bar = function () {
    return 3;
  };
  return bar();
  var bar = function () {
    // unreached code
    return 8;
  };
}
console.log(foo()); // Output: 3
```

- In this example, the variable `bar` is declared twice, but since it is a variable declaration and not a function declaration, the first declaration is hoisted, and the second declaration is ignored. Therefore, when `foo` is called, it returns `3`.
- Another Example of Function and Hoisting with IIFE:

```
function foo() {
  return bar();
  function bar() {
    return 3;
  }
  var bar = function () {
    // unreached code
    return 8;
  };
}
console.log(foo()); // Output: 3
```

- In this example, the function `bar` is declared as a function declaration and then as a variable. The function declaration is hoisted, so when `foo` is called, it returns `3`. The variable declaration is ignored because the function declaration takes precedence.
- Another Example of Function and Hoisting with IIFE:

```
console.log(foo());
var foo = function () {
  return bar();
  function bar() {
    return 3;
  }
  var bar = function () {
    return 8;
  };
};
// Output: TypeError: foo is not a function
```