History of JavaScript

JavaScript, the most widely used scripting language for web development, has an interesting history. Here's a detailed and simplified timeline of **JavaScript's evolution**:

---

1. The Birth of JavaScript (1995)

- **Who:** Created by **Brendan Eich**, a developer at **Netscape Communications**.
- **Why:** Netscape wanted to make websites **interactive** and **dynamic**. The goal was to add scripting capabilities to browsers to complement static HTML.
- **When:** It was created in **just 10 days** in **May 1995**.
- **What:** Initially called **Mocha** → renamed to **LiveScript** → finally rebranded as **JavaScript** to ride on the popularity of **Java** at the time.

---

2. JavaScript in Netscape Navigator (1996)

- Netscape integrated JavaScript into its web browser, **Netscape Navigator 2.0**.
- Microsoft, seeing Netscape's success, launched its own browser (**Internet Explorer**) and implemented a similar scripting language called **JScript**.

---

3. Standardization (1997)

- As browser wars between Netscape and Microsoft heated up, inconsistencies in JavaScript emerged.
- JavaScript was submitted to **ECMA International** for standardization.
- The **ECMAScript** standard (ES1) was released in **1997**.

---

4. ECMAScript Versions

Over time, the **ECMAScript standard** has evolved, with new features added to improve JavaScript's capabilities:

1. **ES1 (1997):** The first version of ECMAScript.
2. **ES3 (1999):** Added better **error handling** and **regular expressions**.
3. **ES5 (2009):** Major improvements:
    - `JSON` support.

    - Array methods like `forEach`, `map`, `filter`, `reduce`.
4. **ES6 (2015):** A game-changer for JavaScript. Key features include:
    - `let` and `const` for variable declarations.
    - Arrow functions (`=>`).
    - Template literals.
    - Classes and Modules.
    - Promises for asynchronous programming.
    - Destructuring assignment.
5. **Post ES6 (2016 - Present):** Annual updates with features like:
    - `async/await` for easier asynchronous code (ES2017).
    - Optional chaining (`?.`).
    - Nullish coalescing operator (`??`).
    - `flat()` and `flatMap()` methods for arrays.

---

5. Rise of JavaScript Frameworks and Libraries (2006 - 2015)

As JavaScript became central to web development, several frameworks and libraries emerged to make development easier:

- **jQuery (2006):** Simplified DOM manipulation and cross-browser issues.
- **Node.js (2009):** Brought JavaScript to the **server-side**.
- **AngularJS (2010):** A framework by Google for building dynamic web applications.
- **React.js (2013):** A library by Facebook for building user interfaces.
- **Vue.js (2014):** Lightweight and easy-to-use framework for modern applications.

---

6. JavaScript Today

- JavaScript is now a **full-stack language**, enabling development on both the client-side (browsers) and server-side (Node.js).
- It powers modern web technologies, mobile apps, desktop apps, and even IoT.
- Popular frameworks and libraries like **React**, **Vue**, and **Angular** dominate front-end development, while **Express.js** and **NestJS** dominate server-side development.

---

7. The Importance of JavaScript

- **Browsers:** JavaScript is the **only programming language** natively supported by all browsers.
- **Modern Trends:** JavaScript has expanded into:

    - **Progressive Web Apps (PWAs)**
    - **Single Page Applications (SPAs)**
    - **Serverless Computing**
    - **AI and ML Libraries**
- **Community:** With a large open-source community and active contributions, JavaScript continues to grow and improve.

Summary of Key Milestones

Year	Milestone
1995	Brendan Eich creates JavaScript (Mocha/LiveScript).
1997	ECMAScript (ES1) is standardized.
2009	Node.js introduces JavaScript on the server-side.
2015	ES6 introduces modern features like `let`, `const`, classes, and arrow functions.
2017+	JavaScript gets annual updates (e.g., `async/await`).

JavaScript has grown from a simple browser scripting language to the **most popular programming language in the world**. It continues to power the web, enabling developers to create complex, dynamic, and interactive applications.

Let me know if you want to dive deeper into any particular milestone or concept! 🚀

Advantages of JavaScript

1. **Client-Side Execution**
    - JavaScript runs directly in the browser, reducing server load and improving performance.
    - This enables faster execution of code without round trips to the server.
2. **Easy to Learn and Use**
    - JavaScript is beginner-friendly and has a straightforward syntax.
    - It's a great entry point for new developers in web development.
3. **Versatility**
    - JavaScript is used for both **front-end** and **back-end** development (via Node.js).

    - It can build web applications, mobile apps, server-side APIs, and even IoT solutions.
4. **Rich User Interfaces**
    - JavaScript enables interactive features like:
        - **Sliders**
        - **Form validations**
        - **Dynamic content updates**
        - Animations (e.g., CSS transitions and libraries like GSAP).
5. **Wide Browser Support**
    - JavaScript is natively supported by all modern browsers (Chrome, Firefox, Safari, Edge, etc.).
6. **Huge Ecosystem and Community**
    - Massive community support with countless libraries and frameworks:
        - React, Angular, Vue for front-end.
        - Node.js, Express for back-end.
    - A vast number of open-source tools and packages are available on **npm**.
7. **Asynchronous Programming**
    - Features like **Promises** and **async/await** make handling asynchronous tasks (e.g., API calls) easier.
8. **Cross-Platform**
    - JavaScript can run on different platforms (Windows, macOS, Linux).
    - Frameworks like **React Native** and **Electron** allow building mobile and desktop apps.
9. **Event-Driven Language**
    - JavaScript handles user events (like clicks, form submissions) seamlessly, making it perfect for interactive web apps.
10. **Continuous Improvement**

- JavaScript evolves rapidly with annual updates (ECMAScript), adding new features and improvements.

Disadvantages of JavaScript

1. **Security Risks**
    - JavaScript code is **visible to everyone** in the browser, making it vulnerable to:
        - Code injection attacks (e.g., Cross-Site Scripting - XSS).
        - Malicious tampering with scripts.
    - Secure coding practices are essential to mitigate these risks.
2. **Browser Compatibility**
    - Older browsers may not support modern JavaScript features.
    - Developers often need **polyfills** or **transpilers** like **Babel** to ensure compatibility.
3. **Lack of Strong Typing**
    - JavaScript is **loosely typed**, which can cause:

        - Bugs that are hard to trace.
        - Type coercion issues (e.g., `5 + "5"` results in `"55"`).
    - Tools like **TypeScript** help solve this issue.
4. **Single-Threaded Execution**
    - JavaScript runs on a **single thread**, making it difficult to handle heavy tasks or CPU-intensive operations.
    - Long-running scripts can block the UI.
5. **Debugging Challenges**
    - Errors in JavaScript are not always easy to debug because of its dynamic nature.
    - A small mistake, like a missing semicolon, can sometimes cause unexpected results.
6. **Performance Limitations**
    - While JavaScript is fast, it's not ideal for resource-intensive tasks like large-scale computations.
    - **WebAssembly** can address this by running faster, compiled code alongside JavaScript.
7. **Code Readability Issues**
    - JavaScript code, when written poorly or without structure, becomes **difficult to maintain**.
    - Anonymous functions and callbacks can lead to **callback hell**.
8. **Excessive Libraries and Frameworks**
    - The JavaScript ecosystem has many libraries and frameworks.
    - Choosing the "right" tool for a project can be overwhelming for beginners.
9. **Client-Side Dependency**
    - Users can disable JavaScript in their browser, which can break the functionality of JavaScript-heavy websites.
10. **Inconsistent Behavior**

- Different browsers may interpret JavaScript code slightly differently, requiring testing and compatibility fixes.

Summary Table

Advantages	Disadvantages
Fast client-side execution	Security vulnerabilities (XSS, code tampering)
Easy to learn and use	Browser compatibility issues
Versatile for full-stack development	Loosely typed, leading to hidden bugs
Interactive UI and dynamic behavior	Single-threaded execution limitations

Cross-platform applications	Poorly written code can become unreadable
Massive community and libraries	Difficult to handle resource-heavy tasks
Supports asynchronous programming	Users can disable JavaScript

---

Conclusion

JavaScript remains the backbone of web development due to its versatility, ease of use, and robust ecosystem. However, developers must be mindful of its limitations, such as security risks and debugging challenges, and adopt tools like **TypeScript** or **linters** to improve code quality.

If you want a detailed explanation of any specific advantage or disadvantage, let me know! 🚀

Variables and Data Types in JavaScript

In JavaScript, variables are used to store and manipulate data. JavaScript provides three keywords to declare variables: **`var`**, **`let`**, and **`const`**, each with specific rules and use cases. Additionally, JavaScript has dynamic and versatile data types.

---

1. Variable Declarations

`var`

- **Scope:** Function-scoped (accessible throughout a function where it is declared).
- **Hoisting:** Variables declared with `var` are hoisted to the top of their scope but initialized as `undefined`.
- **Re-declaration:** Allows re-declaring the same variable within the same scope.
- **Use:** Considered outdated; avoid using it in modern JavaScript.

javascript
Copy code

```
var x = 10;
console.log(x); // 10
```

`let`

- **Scope:** Block-scoped (accessible only within the block `{}` in which it is defined).
- **Hoisting:** Hoisted but not initialized (ReferenceError if accessed before declaration).

- **Re-declaration:** Cannot re-declare a variable in the same scope.
- **Use:** Use `let` when the variable value will change later.

javascript
Copy code
```
let y = 20;
y = 30; // Allowed
console.log(y); // 30
```

`const`

- **Scope:** Block-scoped (like `let`).
- **Hoisting:** Hoisted but not initialized.
- **Re-declaration:** Cannot be re-declared or reassigned.
- **Use:** Use `const` for values that will not change.

javascript
Copy code
```
const z = 50;
// z = 60; // Error: Assignment to constant variable
console.log(z); // 50
```

---

2. Data Types in JavaScript

JavaScript is dynamically typed, meaning the type of a variable is determined at runtime, and you don't need to explicitly specify it.

Primitive Data Types
Number: Represents integers and floating-point numbers.
javascript
Copy code
```
let num = 42; // Integer
let pi = 3.14; // Float
```

1.

String: Represents text enclosed in quotes (`' '`, `" "`, or `` ` ` ``).
javascript
Copy code
```
let name = "John Doe";
```

2.

Boolean: Represents `true` or `false`.
javascript
Copy code
```
let isActive = true;
```

3.

Null: Represents an intentional absence of value.
javascript
Copy code
```
let emptyValue = null;
```

4.

Undefined: A variable declared but not assigned a value.
javascript
Copy code
```
let undefinedValue; // Default is undefined
```

5.

Symbol: Represents a unique value (introduced in ES6).
javascript
Copy code
```
let uniqueId = Symbol("id");
```

6.

BigInt: Represents integers larger than `2^53 - 1` (introduced in ES2020).
javascript
Copy code
```
let bigNum = 1234567890123456789012345678 90n;
```

7.

Non-Primitive (Reference) Data Types
Object: A collection of key-value pairs.
javascript
Copy code
```
let obj = { name: "John", age: 30 };
```

1.

Array: A list-like structure.
javascript
Copy code
```
let arr = [1, 2, 3, 4];
```

2.

Function: A reusable block of code.
javascript
Copy code
```
function greet() {
  console.log("Hello");
}
```

3.
4. **Date, RegExp, and others:** Special built-in objects for specific purposes.

---

3. Differences Between `var`, `let`, and `const`

Feature	var	let	const
Scope	Function-scoped	Block-scoped	Block-scoped
Hoisting	Yes, initialized as `undefined`	Yes, but no initialization	Yes, but no initialization
Re-declaration	Allowed	Not allowed	Not allowed
Re-assignment	Allowed	Allowed	Not allowed
Use Case	Avoid using	For variables that change	For constants

---

4. Example Code

javascript
Copy code
```
// var example
var a = 10;
if (true) {
```

```
  var a = 20; // Re-declares and modifies the same variable
}
console.log(a); // 20

// let example
let b = 10;
if (true) {
  let b = 20; // Block-scoped
  console.log(b); // 20
}
console.log(b); // 10

// const example
const c = 10;
// c = 20; // Error: Assignment to constant variable
console.log(c); // 10
```

Key Points

- **`let` and `const`** should be preferred in modern JavaScript because they prevent unexpected issues caused by `var`.
- Use `const` by default unless you need to reassign the value.

Let me know if you'd like more examples or explanations! 😊

---

1. `var` in Global Scope

- When you declare a variable using `var` **outside of any function**, it becomes a **global variable**.
- In a browser environment, a global `var` variable is added as a property of the global object (`window`).

Example:

javascript
Copy code

```
var globalVar = "I am global";
```

```
console.log(window.globalVar); // "I am global"
```

This makes `var` behave like it's "globally scoped," but technically it's **function-scoped**, not block-scoped.

---

2. `var` Inside Functions (Function Scope)

- Variables declared with `var` inside a function are **limited to that function** (function-scoped).
- You cannot access them outside the function.

Example:

javascript
Copy code

```
function testVar() {
  var localVar = "I am local to this function";
  console.log(localVar); // "I am local to this function"
}

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

---

3. `var` Inside Blocks (No Block Scope)

- `var` does **not respect block scope**, meaning it is accessible outside the block in which it was declared.

Example:

javascript
Copy code

```
if (true) {
  var blockVar = "I am declared in a block";
}
console.log(blockVar); // "I am declared in a block"
```

This behavior differs from `let` and `const`, which are **block-scoped**.

---

4. Why `var` is Function-Scoped but Can Feel "Global"?

- If `var` is declared **outside any function**, it behaves like a global variable.
- If it's declared **inside a function**, it's scoped to that function only.
- Variables declared with `var` inside **blocks like `if` or `for`** are NOT restricted to the block. Instead, they "leak" into the surrounding function or global scope.

Example with For Loop:

javascript
Copy code

```
for (var i = 0; i < 5; i++) {
  console.log(i); // Outputs 0, 1, 2, 3, 4
}
console.log(i); // `i` is still accessible here: 5
```

Summary

`var` Behavior	Explanation
Global Scope	Declared outside a function; becomes a property of `window`.
Function Scope	Scoped to the function where it is declared.
No Block Scope	Does not respect block boundaries (e.g., `if`, `for`, etc.).
Attached to `window`	Global `var` variables are added as properties to the `window` object.

Why Prefer `let` and `const`?

- **`let` and `const`** respect block scope, which prevents unintended variable access or modification outside the intended scope.
- `var` can lead to bugs due to its lack of block scope, especially in loops or nested conditions.

Primitive vs. Object in JavaScript

1. **Primitive** values are immutable and have no methods or properties by default.
2. **Objects** are reference types and can have properties and methods.

Strings as Primitive Values

- A **string** in JavaScript is a **primitive data type** because it holds simple, immutable textual data.
- When you declare a string using quotes (`' '`, `" "`, or `` ` ` ``), it is a primitive.

Example:

javascript
Copy code

```
let str = "Hello, World!";
console.log(typeof str); // "string"
```

Strings Behave Like Objects (Wrapper Objects)

When you try to access a property or method of a string (like `.length` or `.toUpperCase()`), JavaScript **temporarily wraps the string primitive** in a **String object**. This is done automatically by the JavaScript engine.

- The **`String` object** is a **wrapper object** that provides methods and properties for primitive strings.

Example:

javascript
Copy code

```
let str = "Hello";
console.log(str.length); // 5
console.log(str.toUpperCase()); // "HELLO"
```

- **What happens behind the scenes:**
    1. JavaScript converts the primitive `str` to a temporary **`String` object**.
    2. The method `.toUpperCase()` is called on the temporary object.

3. After the method execution, the temporary object is discarded, and `str` remains a primitive.

Proof:

javascript
Copy code
```
let str = "Hello";
console.log(str instanceof String); // false -> str is NOT an object
console.log(typeof str); // "string" -> It's still a primitive
```

Explicit String Objects

If you use the **`String` constructor** to create a string, it explicitly creates a **String object**, not a primitive.

Example:

javascript
Copy code
```
let strPrimitive = "Hello"; // Primitive string
let strObject = new String("Hello"); // String object

console.log(typeof strPrimitive); // "string"
console.log(typeof strObject); // "object"

console.log(strPrimitive == strObject); // true (value comparison)
console.log(strPrimitive === strObject); // false (type and value
comparison)
```

Key Differences:

- Primitive strings are simple **string values**.
- String objects are complex **object types**.

Summary

- **Strings are primitive data types** in JavaScript.

- However, when you use methods or properties on a string, JavaScript temporarily converts the string to a **`String` object** (a wrapper object).
- Explicitly using `new String()` creates a **String object**, which is different from a primitive string.

Best Practice

Always use string literals (primitive) instead of `new String()`. Creating String objects unnecessarily adds complexity and can cause unexpected results

---

What Does "Immutable" Mean?

- **Immutable** means **unchangeable**.
- When you perform any operation on a primitive value (like a string, number, or boolean), a **new value** is created, and the original value stays untouched.

How Primitive Data Types Behave

1. **Strings (Immutable Example)**
   Strings in JavaScript are immutable. Operations like concatenation, slicing, or changing characters do not alter the original string—they create a new string.

Example:

javascript
Copy code

```
let str = "Hello";
str[0] = "h"; // Attempt to modify the first character
console.log(str); // "Hello" (unchanged)

let newStr = str + " World";
console.log(newStr); // "Hello World"
console.log(str);    // "Hello" (original remains unchanged)
```

Here:

- Attempting to modify `str[0]` does not change the string.
- Concatenation produces a **new string**, and the original string remains unmodified.

---

2. **Numbers (Immutable Example)**
   Operations on numbers create new values rather than modifying the existing number.

Example:

javascript
Copy code
```
let num = 10;
let newNum = num + 5;

console.log(num);    // 10 (original remains unchanged)
console.log(newNum); // 15 (new value created)
```

---

3. **Booleans (Immutable Example)**
   Boolean values (`true` or `false`) cannot be changed.

Example:

javascript
Copy code
```
let isTrue = true;
let isFalse = !isTrue; // Creates a new boolean value

console.log(isTrue);  // true (original remains unchanged)
console.log(isFalse); // false (new value created)
```

---

4. **Other Primitives**

- `null`, `undefined`, and `Symbol` are also immutable because they are single, unique values and cannot be altered.

Key Point: Reassigning Variables vs. Immutability

The immutability of primitives refers to their **values** not being changeable.
However, the **variable** holding a primitive can be reassigned to another value.

Example:

javascript
Copy code

```
let x = 5;
x = 10; // The variable x now refers to a new value

console.log(x); // 10
```

Here, we reassign `x`, but the value `5` itself did not change—it simply got replaced by a new value `10`.

Why is Immutability Important?

- Immutability makes primitive values **predictable** and ensures they are **safe** to use without side effects.
- Operations on primitive values never alter the original value, which avoids unexpected behavior in programs.

Summary

- **Primitive data types (string, number, boolean, null, undefined, Symbol, BigInt)** are immutable.
- Any operation on a primitive value creates a **new value** without modifying the original.
- Variables can be reassigned, but the values they hold remain unchangeable.

Immutability of Primitive Values vs. Variable Reassignment

- The **value** `5` itself is **immutable**—it cannot be changed to become something else.
- When you write `x = 10`, you are not changing the value `5`. Instead, you are reassigning the variable `x` to point to a **new value** (`10`).

The **variable** `x` is mutable (it can hold a new value), but the **primitive values** like `5` and `10` are immutable.

Understanding with an Analogy

Think of **primitive values** as **numbers written on pieces of paper**:

- `5` is written on one piece of paper.
- `10` is written on another piece of paper.

When you say:

javascript
Copy code
```
let x = 5;
```

The variable `x` is like a **label** pointing to the piece of paper with `5`.

When you reassign:

javascript
Copy code
```
x = 10;
```

The label `x` is now pointing to the **piece of paper with `10`**.
The piece of paper with `5` remains unchanged—it still says `5`. You just stopped referring to it.

Why Is This Immutability?

The **value `5` itself** cannot be modified. You cannot turn the value `5` into `10`. What you can do is:

1. Reassign the variable `x` to a new value.
2. Perform operations that produce new values.

Example:

javascript
Copy code
```
let x = 5;
let y = x; // y also refers to the value 5

x = 10; // Reassign x to a new value
```

```
console.log(y); // 5 -> y still holds the original value
console.log(x); // 10 -> x now refers to a new value
```

Here:

- `5` remains untouched (immutable).
- `y` still holds `5` because primitives are **copied** when assigned to new variables.

Mutability of Objects vs. Immutability of Primitives

To further clarify:

- **Primitives**: Values like `5`, `"hello"`, or `true` are immutable. You cannot change their internal content.
- **Objects**: Objects are mutable. Their properties or values can be modified.

Example with an Object:

javascript
Copy code

```
let obj = { value: 5 };
obj.value = 10; // The object is modified

console.log(obj); // { value: 10 }
```

Here, the object is mutable—you can change its content.

Key Point

When you reassign a variable holding a primitive value, you are **not modifying the value itself**. You are simply making the variable reference a new value.

The value `5` is still `5`. It remains immutable. Reassigning the variable does not violate this immutability principle.

Summary

- Primitive values like `5` are immutable: you cannot change the actual value `5` to something else.
- Reassigning a variable (like `x = 10`) **does not change the original value**—it simply points the variable to a new value.
- Objects, in contrast, are mutable because their properties can be changed.

When you reassign a **primitive value** to a variable in JavaScript, the **previous value** stays in memory (for a short time), and the **new value** is assigned to a new memory location. Let me break it down for clarity:

Memory Behavior for Primitives

1. **Primitives (like numbers, strings, booleans)** are stored in the stack memory.
2. When you reassign a primitive, the variable gets updated to point to a **new memory location** where the new value is stored.
3. The **previous value** remains in memory until the **garbage collector** clears it (if nothing is referencing it).

Step-by-Step Example

javascript
Copy code

```
let x = 5; // "5" is stored in memory, and x points to it
x = 10;    // A new memory location is created for "10", and x now
points to it
```

- **Step 1:** `5` is stored in one memory location, and the variable `x` refers to it.
- **Step 2:** When you assign `x = 10`, a new memory location is allocated for `10`.
- **Step 3:** The variable `x` is updated to point to the new memory location for `10`.
- **Step 4:** The value `5` becomes **unreferenced** and will eventually be cleared by the **garbage collector**.

Visualization of Memory

Memory Location	Value	Variable
`0x001`	`5`	`x`

`0x002`	`10`	`x` (updated)

Initially, `x` points to `0x001`. After reassignment, it points to `0x002`. The value `5` (at `0x001`) remains in memory until the garbage collector removes it.

Why is This Important?

- Primitives are **immutable**, so any operation that seems to "change" a primitive actually creates a **new value** in memory.
- Variables are just **pointers** or **labels** to memory locations.
- This behavior makes primitives lightweight and predictable.

Objects (Different Behavior)

For comparison, objects are stored in the **heap memory** and behave differently because they are mutable:

javascript
Copy code

```
let obj = { value: 5 };
obj.value = 10; // The same memory location is modified
```

Here, the memory location holding the object does not change. Instead, the **content of the object** is updated.

Summary

- For **primitives**, reassignment points the variable to a new memory location with the new value.
- The **old value** remains in memory temporarily until garbage collection.
- Primitives are immutable because their values cannot be altered in place—they are always replaced with a new value.

How Non-Primitive Data Types Work in Memory

1. **Non-primitive data types** (e.g., objects, arrays, and functions) are stored in the **heap memory**.
2. When you assign a non-primitive value to a variable, the variable stores a **reference (or pointer)** to the memory location where the value (object/array) is stored, **not the actual value itself**.
3. When you modify the object or array, the changes are made directly to the value stored in the **heap memory**, because both variables (or references) point to the same location.

Example: Non-Primitive Data Types (Objects)

javascript
Copy code

```
let obj1 = { name: "Ali" }; // An object is created in heap memory
let obj2 = obj1; // obj2 gets a reference (pointer) to the same object

obj2.name = "Azhar"; // Modify the object through obj2

console.log(obj1.name); // "Azhar"
console.log(obj2.name); // "Azhar"
```

What Happens Here?

1. `obj1` stores a **reference** to the memory location where the object `{ name: "Ali" }` is stored.
2. `obj2` is assigned the **same reference** as `obj1`.
3. When `obj2.name` is modified, it updates the object **in the heap memory**, so both `obj1` and `obj2` reflect the change.

Key Difference: Primitives vs. Non-Primitives

Behavior	Primitives	Non-Primitives
Stored in	Stack memory	Heap memory
Value type	Actual value	Reference (pointer to memory)
Immutability	Immutable	Mutable

Copy behavior	Creates a new copy of the value	Creates a copy of the reference
Example (modification)	Doesn't affect the original value	Affects the original object/array

---

Example: Arrays (Non-Primitive)

javascript
Copy code

```
let arr1 = [1, 2, 3]; // Array created in heap memory
let arr2 = arr1; // arr2 gets a reference to the same array

arr2.push(4); // Modify the array through arr2

console.log(arr1); // [1, 2, 3, 4] (modified)
console.log(arr2); // [1, 2, 3, 4] (modified)
```

- Here, `arr1` and `arr2` point to the **same memory location**.
- Modifying the array via `arr2` affects the array that `arr1` also references.

Copying Non-Primitive Types (Avoiding Shared References)

If you want to avoid modifying the original object/array, you need to create a **shallow copy** or **deep copy**.

Shallow Copy Example

Using the **spread operator** or `Object.assign()`:

javascript
Copy code

```
let arr1 = [1, 2, 3];
let arr2 = [...arr1]; // Creates a shallow copy
arr2.push(4);

console.log(arr1); // [1, 2, 3] (unchanged)
console.log(arr2); // [1, 2, 3, 4] (new copy modified)
```

Deep Copy Example

For deeply nested objects or arrays, you can use `JSON.parse(JSON.stringify(obj))` or libraries like Lodash.

Summary

1. **Non-primitive data types** (objects, arrays, functions) are stored in **heap memory**.
2. Variables hold a **reference** to the memory location of the object.
3. Modifying the object or array via one reference affects all other references pointing to the same memory.
4. To avoid shared references, create a **shallow** or **deep copy**.

What is an Object in Computer Science?

- An **object** is a value in memory that can be identified and referenced by an **identifier** (a variable or a constant).
- It typically represents a collection of properties (key-value pairs), and those properties may themselves be primitive values, references to other objects, or functions.
- Objects can also be used to model real-world entities by grouping related data and behavior.

Objects in JavaScript

In JavaScript:

- Objects are the **only mutable values**, meaning you can change their properties or contents after creation.
- **Primitive values** (like numbers, strings, booleans) are immutable, meaning they cannot be altered once created.

Example of Mutable Objects:
javascript
Copy code

```
let person = { name: "Ali", age: 25 };
person.age = 26; // Modifies the object
console.log(person); // { name: "Ali", age: 26 }
```

Functions as Objects

Functions in JavaScript are **special types of objects** with additional capabilities:

1. **Callable**: Functions can be invoked using `()` syntax.
2. **Properties**: Functions can have properties, just like any other object.

Example of Functions as Objects:
javascript
Copy code

```
function greet() {
  console.log("Hello!");
}

greet.language = "English"; // Adding a property to the function
object
console.log(greet.language); // "English"
```

Here, the function `greet` behaves like an object because you can add properties to it, but it also has the added capability of being **callable**.

What Makes Objects Mutable in JavaScript?

- An object's properties can be added, updated, or deleted at any time.
- This mutability is what differentiates objects from primitives.

Example of Adding/Deleting Properties:
javascript
Copy code

```
let car = { brand: "Toyota" };
car.model = "Corolla"; // Adding a new property
delete car.brand; // Deleting a property
console.log(car); // { model: "Corolla" }
```

Immutability of Functions

Even though functions are objects, the function's **code** (behavior when invoked) is immutable. For example:

javascript
Copy code

```
function sayHi() {
  console.log("Hi");
}

// You cannot change the function's code directly
sayHi = () => console.log("Hello!"); // This creates a new function, not mutating the old one
```

Key Points to Remember

1. **Objects are mutable**: Their properties can be altered.
2. **Functions are objects**:
    - They can store additional properties.
    - They have the special capability of being callable.
3. **Primitives are immutable**: They cannot be changed after creation.

Let me know if you'd like further clarification on any of these points! 😊

JavaScript Object Properties in Simple Terms

In JavaScript, objects are collections of **key-value pairs**, and these key-value pairs are called **properties**. Let's break it down step by step:

---

1. What are Properties?

- A **property** is a key-value pair.
    - **Key**: The name of the property (must be a string or symbol).
    - **Value**: The data stored in the property (can be of any type, even another object).
- You can add, modify, or remove properties after creating an object.

Example:

javascript
Copy code

```
let car = { brand: "Toyota", model: "Corolla" };

// Adding a property
car.year = 2020;

// Modifying a property
car.brand = "Honda";

// Removing a property
delete car.model;

console.log(car); // { brand: "Honda", year: 2020 }
```

---

2. Two Types of Properties

In JavaScript, there are **two main types of properties**:

A. Data Properties

These are the most common type of properties and simply hold a value.

- **Attributes of Data Properties**:
    1. `value`: The value of the property.
    2. `writable`: If `true`, the value can be changed.
    3. `enumerable`: If `true`, the property appears in loops like `for...in`.
    4. `configurable`: If `true`, the property can be deleted or changed to an accessor property.

Example:

javascript
Copy code

```
let obj = {};
Object.defineProperty(obj, "name", {
  value: "Ali",
  writable: true,       // Can modify the value
  enumerable: true,     // Will show up in for...in loop
  configurable: true    // Can delete or modify the property
```

```
});

console.log(obj.name); // "Ali"
```

B. Accessor Properties

These don't directly store a value but instead define functions to **get** or **set** the value.

- **Attributes of Accessor Properties**:
    1. `get`: A function that returns the property value.
    2. `set`: A function that sets a new value to the property.
    3. `enumerable` and `configurable`: Same as data properties.

Example:
javascript
Copy code

```
let user = {
  firstName: "Ali",
  lastName: "Khan",

  // Accessor property
  get fullName() {
    return this.firstName + " " + this.lastName;
  },
  set fullName(name) {
    [this.firstName, this.lastName] = name.split(" ");
  }
};

// Using the getter
console.log(user.fullName); // "Ali Khan"

// Using the setter
user.fullName = "Azhar Ali";
console.log(user.firstName); // "Azhar"
console.log(user.lastName);  // "Ali"
```

---

3. The Prototype

- Every object in JavaScript has a hidden property called `[[Prototype]]`, which points to another object or `null`.
- Properties on the prototype can be accessed as if they were properties of the object itself.

Example:
javascript
Copy code

```
let animal = { eats: true };
let dog = Object.create(animal); // dog inherits from animal

dog.barks = true;

console.log(dog.eats);  // true (from prototype)
console.log(dog.barks); // true (own property)
```

---

4. Objects vs Maps

- **Objects**: Best for structured data where keys are strings or symbols.
- **Maps**: Better for cases where you need arbitrary key-value pairs with more performance and flexibility.

When to Use Maps:

- Keys are not limited to strings or symbols (e.g., numbers or objects as keys).
- Easier to iterate and get size with `map.size`.

Example:
javascript
Copy code

```
let map = new Map();
map.set("name", "Ali");
map.set(1, "Number Key");

console.log(map.get("name")); // "Ali"
console.log(map.get(1));      // "Number Key"
```

Key Differences Between Data and Accessor Properties

Feature	Data Property	Accessor Property
Stores a value	Directly holds a value	Does not hold a value directly
Attributes	`value`, `writable`, `enumerable`, `configurable`	`get`, `set`, `enumerable`, `configurable`
Example Use	`{ key: "value" }`	`{ get key() { return value; } }`

Summary

1. **Properties** are key-value pairs.
2. **Data properties** store values directly, while **accessor properties** use `get` and `set` functions to retrieve or modify values.
3. Use `Object.defineProperty` for fine control over property attributes.
4. Objects have prototypes that allow inheritance.
5. For key-value storage with arbitrary keys, consider using `Map`.

Understanding JavaScript Memory Areas and Concepts for Different Data Types

JavaScript handles memory allocation and management differently for **primitive data types** and **non-primitive data types (objects)**. Here's an explanation:

1. Memory Areas in JavaScript

A. Stack Memory

- Used for storing **primitive data types** and function execution contexts.
- **Primitive types** (like numbers, strings, booleans, etc.) are stored directly in the stack because they are small and have fixed sizes.
- Stack memory is **faster** to access and is automatically managed when the function or scope ends.

B. Heap Memory

- Used for storing **non-primitive data types** (objects, arrays, functions).
- Objects are stored in the heap because they can grow dynamically in size.
- References (pointers) to these objects are stored in the stack.
- Heap memory is managed by **garbage collection**.

---

2. Memory Handling for Different Data Types

Primitive Data Types (Immutable)

- Stored **directly in the stack**.
- When you assign a value to a primitive type, the value itself is copied.

Example:
javascript
Copy code

```
let x = 5;
let y = x; // Copy the value
y = 10;

console.log(x); // 5 (x is unaffected)
console.log(y); // 10
```

Here, `x` and `y` are independent because their values are stored in the stack as separate copies.

Non-Primitive Data Types (Mutable)

- Stored in **heap memory**.
- The variable in the stack contains a **reference** (or pointer) to the location in the heap.

Example:
javascript
Copy code

```
let obj1 = { name: "Ali" };
let obj2 = obj1; // Copy the reference
obj2.name = "Azhar";

console.log(obj1.name); // "Azhar" (obj1 is affected)
```

```
console.log(obj2.name); // "Azhar"
```

Here:

- `obj1` and `obj2` both reference the same memory location in the heap.
- Changing the value through `obj2` affects `obj1` because they share the same reference.

---

3. How Memory Works with Different Operations

A. Assignment

- **Primitive**: Creates a copy of the value.
- **Object**: Creates a copy of the reference.

B. Functions

- **Primitive arguments**: Passed by value.
- **Object arguments**: Passed by reference.

Example:
javascript
Copy code

```
function modifyPrimitive(val) {
  val = 10;
}
let num = 5;
modifyPrimitive(num);
console.log(num); // 5 (unchanged)

function modifyObject(obj) {
  obj.name = "Changed";
}
let person = { name: "Original" };
modifyObject(person);
console.log(person.name); // "Changed" (modified)
```

---

4. Garbage Collection

JavaScript automatically manages memory with **garbage collection**.

- **Unreachable memory** (not referenced anywhere) is cleaned up to free memory space.
- Objects in the heap remain in memory until no references point to them.

Example:
javascript
Copy code

```
let obj = { name: "Ali" };
obj = null; // Previous object becomes eligible for garbage collection
```

---

5. Important Concepts

Immutability of Primitives

- When a new value is assigned to a primitive, the old value stays in memory temporarily but becomes eligible for garbage collection.

Mutability of Objects

- Object properties can be changed in place because the reference remains the same.

Shallow Copy vs Deep Copy

- **Shallow Copy**: Copies references for nested objects (e.g., using `Object.assign()` or spread operator).
- **Deep Copy**: Copies the entire object structure, ensuring no shared references (e.g., using `JSON.parse(JSON.stringify(obj))`).

---

6. Summary Table

Feature	Primitive	Object (Non-Primitive)
Memory Area	Stack	Heap
Stored As	Value	Reference
Size	Fixed	Dynamic
Mutability	Immutable	Mutable

Assignment	Creates a new copy	Shares the same reference
Garbage Collection	Automatically when out of scope	When no references exist

Understanding these concepts is critical for optimizing memory usage and avoiding bugs like unintended mutations. Let me know if you'd like examples or explanations on specific points! 😊

How `undefined` Works

A variable declared but not assigned a value automatically has the value `undefined`:
javascript
Copy code
```
let x;
console.log(x); // undefined
```

1.

You can manually assign `undefined` to a variable:
javascript
Copy code
```
let y = 10;
y = undefined;
console.log(y); // undefined
```

2.
3. Explicitly assigning `undefined` is different from assigning `null`:
    - `undefined`: Indicates a variable has been declared but not assigned a value.
    - `null`: Explicitly represents "no value" or "empty."

Why Assigning `undefined` is Discouraged

- **Confusion**: Assigning `undefined` makes it harder to distinguish between a variable intentionally set to `undefined` and one that is uninitialized.

- **Best Practice**: Use `null` when you want to explicitly clear a variable or indicate an absence of value.

Clearing a Variable

Instead of assigning `undefined`, you should use `null` to indicate the variable is empty:

javascript
Copy code
```
let value = 42;
value = null; // Explicitly cleared
console.log(value); // null
```

Summary

While you **can** assign `undefined` to a variable, it's better to use `null` for clarity and to follow JavaScript best practices. The use of `undefined` is typically reserved for JavaScript itself to indicate uninitialized variables or missing function arguments.

Here are some **commonly used and interview-relevant methods** from the `Math` object:

1. Rounding Numbers

`Math.round()`

Rounds a number to the nearest integer.

javascript
Copy code
```
console.log(Math.round(4.7)); // 5
console.log(Math.round(4.4)); // 4
```

`Math.ceil()`

Rounds a number **up** to the next largest integer.

javascript
Copy code
```
console.log(Math.ceil(4.1)); // 5
console.log(Math.ceil(-4.1)); // -4
```

`Math.floor()`

Rounds a number **down** to the largest integer less than or equal to it.

javascript
Copy code
```
console.log(Math.floor(4.9)); // 4
console.log(Math.floor(-4.9)); // -5
```

`Math.trunc()`

Removes the fractional part of a number, keeping only the integer part.

javascript
Copy code
```
console.log(Math.trunc(4.9)); // 4
console.log(Math.trunc(-4.9)); // -4
```

---

2. Generating Random Numbers

`Math.random()`

Generates a random decimal number between `0` (inclusive) and `1` (exclusive).

javascript
Copy code
```
console.log(Math.random()); // e.g., 0.123456789
```

To generate a random integer within a range:

javascript

Copy code

```
let min = 1, max = 10;
let randomInt = Math.floor(Math.random() * (max - min + 1)) + min;
console.log(randomInt); // Random integer between 1 and 10
```

---

3. Power and Roots

`Math.pow(base, exponent)`

Raises a number to the power of another.

javascript
Copy code

```
console.log(Math.pow(2, 3)); // 8 (2^3)
console.log(Math.pow(5, 2)); // 25
```

`Math.sqrt()`

Calculates the square root of a number.

javascript
Copy code

```
console.log(Math.sqrt(16)); // 4
console.log(Math.sqrt(2)); // ~1.414
```

`Math.cbrt()`

Calculates the cube root of a number.

javascript
Copy code

```
console.log(Math.cbrt(27)); // 3
console.log(Math.cbrt(8)); // 2
```

---

4. Absolute Value

`Math.abs()`

Returns the absolute value of a number.

javascript
Copy code
```
console.log(Math.abs(-7)); // 7
console.log(Math.abs(7));  // 7
```

---

5. Maximum and Minimum

`Math.max()`

Returns the largest number from a list of numbers.

javascript
Copy code
```
console.log(Math.max(1, 5, 10, 20)); // 20
```

`Math.min()`

Returns the smallest number from a list of numbers.

javascript
Copy code
```
console.log(Math.min(1, 5, 10, 20)); // 1
```

---

6. Trigonometric Functions

`Math.sin(x)`

Returns the sine of an angle (in radians).

javascript
Copy code
```
console.log(Math.sin(Math.PI / 2)); // 1
```

`Math.cos(x)`

Returns the cosine of an angle (in radians).

javascript
Copy code
```
console.log(Math.cos(0)); // 1
```

`Math.tan(x)`

Returns the tangent of an angle (in radians).

javascript
Copy code
```
console.log(Math.tan(Math.PI / 4)); // 1
```

---

7. Logarithmic Functions

`Math.log(x)`

Returns the natural logarithm (base `e`) of a number.

javascript
Copy code
```
console.log(Math.log(Math.E)); // 1
console.log(Math.log(1)); // 0
```

`Math.log10(x)`

Returns the base-10 logarithm of a number.

javascript
Copy code
```
console.log(Math.log10(100)); // 2
```

---

8. Constants

JavaScript provides some useful mathematical constants through the `Math` object:

`Math.PI`: Ratio of a circle's circumference to its diameter (~3.14159).
javascript

Copy code
```
console.log(Math.PI); // 3.141592653589793
```

-

`Math.E`: Euler's constant (~2.718).
javascript
Copy code
```
console.log(Math.E); // 2.718281828459045
```

-

---

9. Miscellaneous

`Math.sign(x)`

Determines the sign of a number:

- Returns `1` for positive numbers, `-1` for negative numbers, and `0` for `0`.

javascript
Copy code
```
console.log(Math.sign(-10)); // -1
console.log(Math.sign(10));  // 1
console.log(Math.sign(0));   // 0
```

`Math.hypot(x, y, ...)`

Returns the square root of the sum of squares of its arguments.

javascript
Copy code
```
console.log(Math.hypot(3, 4)); // 5 (Pythagoras theorem)
```

Common Interview Questions Using Math Methods

1. **Generate random numbers within a range.**
2. **Round a floating-point number to `n` decimal places.**
3. **Find the largest or smallest number in an array.**

4. **Calculate the hypotenuse of a triangle using `Math.hypot()`.**
5. **Check if a number is a power of 2 using `Math.log2()` or `Math.pow()`.**

---

most commonly used and asked string methods:

1. Accessing String Characters

`charAt(index)`

Returns the character at the specified index.

javascript
Copy code
```
let str = "JavaScript";
console.log(str.charAt(0)); // "J"
console.log(str.charAt(4)); // "S"
```

`charCodeAt(index)`

Returns the Unicode of the character at the specified index.

javascript
Copy code
```
console.log("A".charCodeAt(0)); // 65
```

---

2. String Length

`length` (property)

Returns the number of characters in a string.

javascript

Copy code
```
let str = "Hello";
console.log(str.length); // 5
```

---

3. Searching in Strings

`indexOf(substring)`

Finds the first occurrence of a substring. Returns `-1` if not found.

javascript
Copy code
```
let str = "hello world";
console.log(str.indexOf("world")); // 6
console.log(str.indexOf("JavaScript")); // -1
```

`lastIndexOf(substring)`

Finds the last occurrence of a substring.

javascript
Copy code
```
let str = "hello world, hello again";
console.log(str.lastIndexOf("hello")); // 13
```

`includes(substring)`

Checks if the string contains the given substring. Returns `true` or `false`.

javascript
Copy code
```
console.log("JavaScript".includes("Script")); // true
console.log("JavaScript".includes("Python")); // false
```

`startsWith(substring)`

Checks if the string starts with the given substring.

javascript

Copy code
```
console.log("JavaScript".startsWith("Java")); // true
```

`endsWith(substring)`

Checks if the string ends with the given substring.

javascript
Copy code
```
console.log("JavaScript".endsWith("Script")); // true
```

---

4. Modifying Strings

`toUpperCase()`

Converts the string to uppercase.

javascript
Copy code
```
console.log("hello".toUpperCase()); // "HELLO"
```

`toLowerCase()`

Converts the string to lowercase.

javascript
Copy code
```
console.log("HELLO".toLowerCase()); // "hello"
```

`trim()`

Removes whitespace from both ends of the string.

javascript
Copy code
```
console.log("  hello  ".trim()); // "hello"
```

`padStart(targetLength, padString)`

Pads the string at the start with the specified character.

javascript
Copy code
```
console.log("5".padStart(3, "0")); // "005"
```

`padEnd(targetLength, padString)`

Pads the string at the end with the specified character.

javascript
Copy code
```
console.log("5".padEnd(3, "0")); // "500"
```

---

5. Extracting Substrings

`slice(start, end)`

Extracts a section of a string. `end` is exclusive.

javascript
Copy code
```
let str = "JavaScript";
console.log(str.slice(0, 4)); // "Java"
console.log(str.slice(4));    // "Script"
```

`substring(start, end)`

Similar to `slice`, but does not support negative indices.

javascript
Copy code
```
console.log("JavaScript".substring(0, 4)); // "Java"
```

`substr(start, length)`

Extracts a substring, given a starting index and a length.

javascript
Copy code

```
console.log("JavaScript".substr(4, 6)); // "Script"
```

---

6. Splitting and Joining Strings

`split(separator)`

Splits a string into an array of substrings.

javascript
Copy code
```
let str = "a,b,c";
console.log(str.split(",")); // ["a", "b", "c"]
```

`concat()`

Concatenates two or more strings.

javascript
Copy code
```
let str1 = "Hello";
let str2 = "World";
console.log(str1.concat(" ", str2)); // "Hello World"
```

Template Literals (Modern Alternative to `concat`)

Use backticks and `${}` for string interpolation.

javascript
Copy code
```
let name = "John";
console.log(`Hello, ${name}!`); // "Hello, John!"
```

---

7. Replacing Parts of Strings

`replace(substring, newSubstring)`

Replaces the first occurrence of a substring with another.

javascript
Copy code
```
let str = "hello world";
console.log(str.replace("world", "JavaScript")); // "hello JavaScript"
```

`replaceAll(substring, newSubstring)`

Replaces all occurrences of a substring with another (introduced in ES2021).

javascript
Copy code
```
let str = "hello world, hello again";
console.log(str.replaceAll("hello", "hi")); // "hi world, hi again"
```

---

8. Checking and Comparing Strings

`localeCompare()`

Compares two strings in the current locale.

javascript
Copy code
```
console.log("a".localeCompare("b")); // -1 (a comes before b)
console.log("b".localeCompare("a")); // 1
console.log("a".localeCompare("a")); // 0
```

---

9. Advanced Methods

`repeat(count)`

Repeats the string `count` times.

javascript
Copy code
```
console.log("ha".repeat(3)); // "hahaha"
```

`match()`

Matches a string against a regular expression.

javascript
Copy code
```
let str = "JavaScript is awesome";
console.log(str.match(/is/)); // ["is", index: 11, input: "JavaScript is awesome"]
```

search()

Searches for a match against a regular expression.

javascript
Copy code
```
console.log("hello world".search(/world/)); // 6
```

valueOf()

Returns the primitive value of a string object.

javascript
Copy code
```
let str = new String("Hello");
console.log(str.valueOf()); // "Hello"
```

Common Interview Questions

How do you reverse a string?
javascript
Copy code
```
let str = "hello";
let reversed = str.split("").reverse().join("");
console.log(reversed); // "olleh"
```

1.

How do you count the occurrences of a character in a string?
javascript
Copy code
```
let str = "hello world";
```

```
let count = str.split("o").length - 1;
console.log(count); // 2
```

2.

How do you check if a string is a palindrome?
javascript
Copy code

```
let str = "madam";
let isPalindrome = str === str.split("").reverse().join("");
console.log(isPalindrome); // true
```

3.

How do you find the longest word in a sentence?
javascript
Copy code

```
let sentence = "JavaScript is awesome";
let longestWord = sentence.split(" ").reduce((a, b) => (a.length >
b.length ? a : b));
console.log(longestWord); // "JavaScript"
```

4.

Mastering these string methods will help you solve real-world problems efficiently and excel in JavaScript interviews. Let me know if you need more examples or explanations! 😊

The `reduce()` method in JavaScript is a powerful array method used to reduce an array to a single value by applying a callback function to each element in the array.

Here's how it works step by step:

Syntax

javascript
Copy code

```
array.reduce(callback(accumulator, currentValue, currentIndex, array),
initialValue)
```

- **`callback`**: A function executed on each element of the array, taking the following parameters:
    1. **`accumulator`**: The accumulated result from the previous callback execution.
    2. **`currentValue`**: The current element being processed.
    3. **`currentIndex`** (optional): The index of the current element.
    4. **`array`** (optional): The array being reduced.
- **`initialValue`** (optional): The initial value of the accumulator. If not provided, the first element of the array is used as the initial value, and the iteration starts from the second element.

How `reduce()` Works

1. The `reduce()` method starts with the **`initialValue`** (if provided) or the first element of the array as the **`accumulator`**.
2. For each element in the array, the **callback function** is executed.
3. The return value of the callback becomes the new **`accumulator`** for the next iteration.
4. At the end of the array, the final value of the **`accumulator`** is returned as the result of the `reduce()` method.

Examples

1. Summing Numbers

javascript
Copy code

```
let numbers = [1, 2, 3, 4];
let sum = numbers.reduce((accumulator, currentValue) => accumulator + currentValue, 0);
console.log(sum); // Output: 10
```

Explanation:

- Initial `accumulator` = 0.
- Iteration 1: `accumulator` = 0 + 1 = 1.
- Iteration 2: `accumulator` = 1 + 2 = 3.
- Iteration 3: `accumulator` = 3 + 3 = 6.
- Iteration 4: `accumulator` = 6 + 4 = 10.

---

2. Finding the Maximum Value

javascript
Copy code

```
let numbers = [5, 3, 8, 2];
let max = numbers.reduce((acc, curr) => (acc > curr ? acc : curr),
-Infinity);
console.log(max); // Output: 8
```

Explanation:

- Initial `accumulator` = -Infinity.
- Iteration 1: Compare -Infinity and 5 → `accumulator` = 5.
- Iteration 2: Compare 5 and 3 → `accumulator` = 5.
- Iteration 3: Compare 5 and 8 → `accumulator` = 8.
- Iteration 4: Compare 8 and 2 → `accumulator` = 8.

---

3. Flattening an Array

javascript
Copy code

```
let nestedArray = [[1, 2], [3, 4], [5]];
let flatArray = nestedArray.reduce((acc, curr) => acc.concat(curr),
[]);
console.log(flatArray); // Output: [1, 2, 3, 4, 5]
```

Explanation:

- Initial `accumulator` = `[]` (empty array).
- Iteration 1: Concatenate `[]` and `[1, 2]` → `accumulator` = `[1, 2]`.
- Iteration 2: Concatenate `[1, 2]` and `[3, 4]` → `accumulator` = `[1, 2, 3, 4]`.
- Iteration 3: Concatenate `[1, 2, 3, 4]` and `[5]` → `accumulator` = `[1, 2, 3, 4, 5]`.

---

4. Counting Occurrences

javascript
Copy code

```
let items = ['apple', 'banana', 'apple', 'orange', 'banana', 'apple'];
let count = items.reduce((acc, curr) => {
    acc[curr] = (acc[curr] || 0) + 1;
    return acc;
}, {});
console.log(count);
// Output: { apple: 3, banana: 2, orange: 1 }
```

Explanation:

- The `accumulator` starts as an empty object `{}`.
- Each item's occurrence is tracked by incrementing its count in the `accumulator`.

---

5. Chaining with Other Methods

javascript
Copy code

```
let numbers = [1, 2, 3, 4];
let result = numbers
  .filter(num => num > 2) // Filter numbers greater than 2 → [3, 4]
  .reduce((acc, curr) => acc + curr, 0); // Sum the filtered numbers → 7
console.log(result); // Output: 7
```

Interview Tip

1. Understand the flow of how `accumulator` updates.
2. Be comfortable using both an **initial value** and starting with the **array's first element**.
3. Practice real-world examples like:
    - Summing or averaging numbers.
    - Grouping or counting elements.
    - Building data structures (e.g., converting arrays to objects).

Let me know if you need further clarification or more examples! 😊

---

1. Function Declaration

A function declared with the `function` keyword.

Syntax:

javascript
Copy code
```
function functionName(parameters) {
    // function body
    return value;
}
```

Example:

javascript
Copy code
```
function add(a, b) {
    return a + b;
}
console.log(add(3, 5)); // Output: 8
```

---

2. Function Expression

A function assigned to a variable. These can be anonymous or named.

Syntax:

javascript
Copy code
```
const variableName = function(parameters) {
    // function body
};
```

Example:

javascript
Copy code
```
const subtract = function(a, b) {
    return a - b;
};
console.log(subtract(10, 5)); // Output: 5
```

---

3. Arrow Functions

Introduced in ES6, they provide a shorter syntax for writing functions. They are always anonymous.

Syntax:

javascript
Copy code
```
const functionName = (parameters) => {
    // function body
    return value;
};
```

Example:

javascript
Copy code
```
const multiply = (a, b) => a * b;
console.log(multiply(2, 3)); // Output: 6
```

Key Differences from Regular Functions:

- Arrow functions do not bind their own `this`. They inherit `this` from their enclosing context.

---

4. Immediately Invoked Function Expressions (IIFE)

Functions executed immediately after they are defined.

Syntax:

javascript
Copy code
```
(function() {
    // function body
})();
```

Example:

javascript
Copy code
```
(function() {
    console.log("This function runs immediately!");
})();
```

---

5. Anonymous Functions

Functions without a name. Commonly used in callbacks.

Example:

javascript
Copy code
```
setTimeout(function() {
    console.log("Anonymous function example.");
}, 1000);
```

---

6. Named Function Expressions

A function expression with a name. Useful for debugging.

Example:

javascript
Copy code
```
const greet = function sayHello() {
    console.log("Hello!");
```

```
};
greet(); // Output: Hello!
```

---

7. Higher-Order Functions

Functions that accept other functions as arguments or return functions.

Example:

javascript
Copy code

```
function calculate(operation, a, b) {
    return operation(a, b);
}
const sum = (a, b) => a + b;
console.log(calculate(sum, 4, 5)); // Output: 9
```

---

8. Callback Functions

A function passed as an argument to another function and executed later.

Example:

javascript
Copy code

```
function processUserInput(callback) {
    const name = "Azhar";
    callback(name);
}
processUserInput(name => console.log(`Hello, ${name}!`)); // Output:
Hello, Azhar!
```

---

9. Generator Functions

Functions that can pause execution and return multiple values using the `yield` keyword.

Syntax:

javascript
Copy code
```
function* generatorFunction() {
    yield value1;
    yield value2;
    // ...
}
```

Example:

javascript
Copy code
```
function* count() {
    yield 1;
    yield 2;
    yield 3;
}
const counter = count();
console.log(counter.next().value); // Output: 1
console.log(counter.next().value); // Output: 2
```

---

10. Async Functions

Functions that handle asynchronous operations using `async` and `await`.

Syntax:

javascript
Copy code
```
async function functionName() {
    const result = await somePromise;
    return result;
}
```

Example:

javascript
Copy code
```
async function fetchData() {
    const response = await
fetch('https://jsonplaceholder.typicode.com/posts/1');
    const data = await response.json();
    console.log(data);
}
fetchData();
```

---

11. Constructor Functions

Functions used to create objects. Typically capitalized.

Example:

javascript
Copy code
```
function Person(name, age) {
    this.name = name;
    this.age = age;
}
const person1 = new Person("Ali", 25);
console.log(person1); // Output: Person { name: 'Ali', age: 25 }
```

---

12. Methods

Functions that are properties of an object.

Example:

javascript
Copy code
```
const person = {
    name: "Azhar",
    greet() {
        console.log(`Hello, ${this.name}!`);
```

```
    }
};
person.greet(); // Output: Hello, Azhar!
```

---

13. Default Parameters

Functions with default values for parameters.

Example:

javascript
Copy code

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

---

14. Rest Parameters

Functions that accept an indefinite number of arguments.

Example:

javascript
Copy code

```
function sum(...numbers) {
    return numbers.reduce((acc, curr) => acc + curr, 0);
}
console.log(sum(1, 2, 3, 4)); // Output: 10
```

---

15. Function Overloading (Not Directly Supported)

JavaScript doesn't directly support overloading, but it can be mimicked.

Example:

javascript
Copy code

```
function greet(name, age) {
    if (age) {
        console.log(`Hello ${name}, you are ${age} years old.`);
    } else {
        console.log(`Hello ${name}!`);
    }
}
greet("Ali", 25); // Output: Hello Ali, you are 25 years old.
greet("Azhar"); // Output: Hello Azhar!
```

Common Interview Topics

- Arrow functions and `this` binding.
- Differences between `function` declarations and expressions.
- Callback and higher-order functions.
- Asynchronous programming using `async/await`.
- Rest parameters and default values.

Let me know if you'd like further details or examples!

In JavaScript, **functions are objects** and are considered **first-class citizens**. Here's what this means and why it is significant:

Functions Are Objects

1. **Functions in JavaScript are treated as objects.**
   They have properties and methods just like any other object.
   - For instance, every function has a `name` property (function's name) and a `length` property (number of arguments it expects).

Example:
javascript
Copy code

```
function greet(name) {
    return `Hello, ${name}!`;
}

console.log(greet.name); // Output: "greet" (function name)
console.log(greet.length); // Output: 1 (number of parameters)
```

2.

Functions can also have their own custom properties:
javascript
Copy code

```
function greet(name) {
    return `Hello, ${name}!`;
}

greet.customProperty = "This is a custom property";
console.log(greet.customProperty); // Output: "This is a custom
property"
```

3.
4. **Function objects** have additional capabilities like being callable (`()`) or constructible (`new`).

First-Class Citizens

JavaScript treats functions as **first-class citizens**, meaning they can be treated like any other variable. Here's what you can do with functions:

1. Assign Functions to Variables

Functions can be assigned to variables, passed around, and stored just like any other value.

javascript
Copy code

```
const sayHello = function() {
    console.log("Hello!");
};
```

```
sayHello(); // Output: Hello!
```

2. Pass Functions as Arguments

Functions can be passed as arguments to other functions (commonly called **callbacks**).

javascript
Copy code
```
function executeCallback(callback) {
    callback();
}

executeCallback(() => console.log("Callback executed!"));
// Output: Callback executed!
```

3. Return Functions from Other Functions

Functions can return other functions.

javascript
Copy code
```
function createMultiplier(multiplier) {
    return function (number) {
        return number * multiplier;
    };
}

const double = createMultiplier(2);
console.log(double(5)); // Output: 10
```

4. Store Functions in Data Structures

Functions can be stored in arrays or objects.

javascript
Copy code
```
const operations = [
    (x) => x + 1,
    (x) => x * 2,
    (x) => x / 2
```

```
];

console.log(operations Output: 8 (second function doubles the input)
```

Why Functions Are First-Class Citizens in JavaScript

- **Flexibility:** Functions can be passed, returned, and stored, enabling powerful design patterns like higher-order functions.
- **Asynchronous Programming:** JavaScript's event-driven model (e.g., `setTimeout`, `fetch`) relies heavily on passing functions as callbacks.
- **Functional Programming:** JavaScript supports functional programming paradigms, thanks to functions being first-class citizens.

Practical Applications

Callbacks: Enable asynchronous programming.
javascript
Copy code

```
setTimeout(() => console.log("Executed after delay"), 1000);
```

1.

Event Handling: Functions are used as handlers.
javascript
Copy code

```
document.getElementById("btn").addEventListener("click", () => {
    console.log("Button clicked!");
});
```

2.

Functional Utilities: Functions like `map`, `filter`, and `reduce` take other functions as arguments.
javascript
Copy code

```
const numbers = [1, 2, 3];
const doubled = numbers.map((num) => num * 2);
console.log(doubled); // Output: [2, 4, 6]
```

3.

Summary

- Functions in JavaScript are **objects** with additional callable behavior.
- Functions are **first-class citizens**, allowing them to be treated like any other value.
- This capability makes JavaScript highly flexible and powerful for functional and asynchronous programming.

Let me know if you'd like further examples or clarifications!

A **function** in JavaScript is a reusable block of code designed to perform a specific task, which is also a callable object, making it capable of accepting inputs (parameters), producing outputs (return values), and serving as a first-class citizen in the language—allowing it to be assigned to variables, passed as arguments, returned by other functions, and enriched with properties and methods, while supporting a variety of forms like anonymous, named, arrow, and constructor functions.

---

functions are first-class citizens in JavaScript.

What It Means:

1. **Functions can be treated like any other value.**
   They can be assigned to variables, passed as arguments, returned from other functions, and stored in data structures like arrays or objects.
2. **Practical Examples:**

Assigned to Variables:
javascript
Copy code

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

-

Passed as Arguments (Callbacks):
javascript
Copy code

```
function executeCallback(callback) {
    callback();
}
executeCallback(() => console.log("Callback executed!"));
// Output: Callback executed!
```

-

Returned from Other Functions:
javascript
Copy code

```
function multiplier(factor) {
    return function (number) {
        return number * factor;
    };
}
const double = multiplier(2);
console.log(double(5)); // Output: 10
```

-

Stored in Data Structures:
javascript
Copy code

```
const operations = {
    add: (a, b) => a + b,
    multiply: (a, b) => a * b
};
console.log(operations.add(2, 3)); // Output: 5
console.log(operations.multiply(2, 3)); // Output: 6
```

-

Why Are They First-Class Citizens?

- They can be dynamically created, assigned, and executed at runtime.
- They enable powerful programming paradigms such as **functional programming** and **event-driven programming**.

- JavaScript's flexibility for **higher-order functions** (functions that take other functions as arguments or return them) relies on functions being first-class citizens.

This property of functions is a core strength of JavaScript, enabling developers to write concise, modular, and flexible code.

Yes, we can use **anonymous functions** multiple times in JavaScript due to their ability to be assigned to variables, passed as arguments, or returned by other functions. This is possible because of **function expressions**, which allow creating unnamed (anonymous) functions on the fly.

Explanation

1. **Anonymous Function:**
    - An anonymous function is a function without a name.
    - It is typically used when the function does not need to be reused directly or referenced by its name.

javascript
Copy code

```
const greet = function () {
    console.log("Hello!");
};
greet(); // Output: Hello!
```

2.
3. **Function Expression:**
    - A function expression allows you to assign an anonymous function to a variable.
    - This enables its reuse and allows calling the function by the variable's name.

javascript
Copy code

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

console.log(add(2, 3)); // Output: 5
```

4.
5. **Reusing Anonymous Functions via Function Expression:**

Anonymous functions assigned to variables or passed as arguments can be used repeatedly. For example:
javascript
Copy code

```
// Assigning an anonymous function to a variable
const multiply = function (x, y) {
    return x * y;
};

console.log(multiply(3, 4)); // Output: 12
console.log(multiply(5, 6)); // Output: 30
```

6. **As Callbacks (Used Multiple Times):**

Anonymous functions are widely used as callbacks, allowing them to be executed multiple times by the higher-order function.
javascript
Copy code

```
const numbers = [1, 2, 3, 4];
const squaredNumbers = numbers.map(function (num) {
    return num * num;
});

console.log(squaredNumbers); // Output: [1, 4, 9, 16]
```

7. **Returning Anonymous Functions from Other Functions:**

Anonymous functions can be returned and reused via closures.
javascript
Copy code

```
function createMultiplier(multiplier) {
    return function (number) {
        return number * multiplier;
    };
}

const double = createMultiplier(2);
console.log(double(5)); // Output: 10

const triple = createMultiplier(3);
```

```
console.log(triple(5)); // Output: 15
```

-

Why Use Anonymous Functions?

- **Conciseness:** When a function is used only once or inline.
- **Flexibility:** They are versatile and can be passed around easily.
- **Readability:** Useful in situations like callbacks or functional programming where naming isn't required.

Summary

We can use anonymous functions multiple times in JavaScript due to **function expressions**, allowing them to be assigned to variables, passed as arguments, or returned from other functions. This flexibility makes them a key feature of JavaScript's functional programming paradigm.

In JavaScript, **`bind`**, **`apply`**, and **`call`** are methods available on functions to explicitly set the `this` value and invoke the function in different ways. Here's a detailed explanation:

1. `call` Method

- **What it does:**
  Invokes a function immediately, allowing you to explicitly set the value of `this` and pass arguments individually.

Syntax:
javascript
Copy code
```
functionName.call(thisArg, arg1, arg2, ...)
```

-

Example:
javascript
Copy code
```
const person = {
```

```
    firstName: "Ali",
    lastName: "Khan",
    fullName: function () {
        return `${this.firstName} ${this.lastName}`;
    }
};

const anotherPerson = { firstName: "John", lastName: "Doe" };

console.log(person.fullName.call(anotherPerson)); // Output: "John
Doe"
```

-
- **When to use:**
  Use `call` when you need to invoke a function immediately with a specific `this` context and arguments.

---

2. `apply` Method

- **What it does:**
  Similar to `call`, but arguments are passed as an array instead of individually.

Syntax:
javascript
Copy code
```
functionName.apply(thisArg, [arg1, arg2, ...])
```

-

Example:
javascript
Copy code
```
const numbers = [5, 6, 2, 3, 7];
const max = Math.max.apply(null, numbers);
console.log(max); // Output: 7
```

-
- **When to use:**
  Use `apply` when you have an array of arguments to pass to the function.

---

3. `bind` Method

- **What it does:**
  Returns a new function with the specified `this` value and optional arguments, but does not invoke it immediately.

Syntax:
javascript
Copy code
```
const boundFunction = functionName.bind(thisArg, arg1, arg2, ...)
```

-

Example:
javascript
Copy code
```
const person = {
    name: "Ali",
    greet: function (greeting) {
        console.log(`${greeting}, my name is ${this.name}`);
    }
};

const anotherPerson = { name: "John" };

const boundGreet = person.greet.bind(anotherPerson, "Hello");
boundGreet(); // Output: "Hello, my name is John"
```

-
- **When to use:**
  Use `bind` when you want to create a new function with a specific `this` value and optionally preset arguments for later invocation.

Comparison Table

Feature	`call`	`apply`	`bind`
Execution	Invokes the function immediately.	Invokes the function immediately.	Returns a new function, does not invoke immediately.

Arguments	Passed individually.	Passed as an array.	Presets arguments for later.
Output	Return value of the function.	Return value of the function.	New function reference.

Key Takeaways

1. Use **`call`** or **`apply`** when you need to invoke the function immediately.
    - Use `call` for individual arguments.
    - Use `apply` for an array of arguments.
2. Use **`bind`** when you need to create a reusable function with a specific `this` context and optional preset arguments.

These methods are powerful for controlling `this`, especially when working with objects,

What is `this` in JavaScript?

- The value of `this` refers to the object that is currently executing the function.
- Sometimes, you want to control what `this` points to when calling a function. That's where `call`, `apply`, and `bind` come in.

---

1. `call()`

- **What it does:**
  The `call()` method allows you to:
    - Set the value of `this`.
    - Pass arguments one by one to the function.
    - Execute the function immediately.

How it works:
Syntax:
javascript
Copy code
```
functionName.call(thisValue, arg1, arg2, ...);
```

-
    - `functionName`: The function you want to call.
    - `thisValue`: The object you want `this` to refer to.

- `arg1, arg2, ...`: Arguments to pass to the function.

Example:
javascript
Copy code

```
const person = {
    fullName: function () {
        return `${this.firstName} ${this.lastName}`;
    }
};

const person1 = { firstName: "Ali", lastName: "Khan" };
const person2 = { firstName: "John", lastName: "Doe" };

console.log(person.fullName.call(person1)); // Output: "Ali Khan"
console.log(person.fullName.call(person2)); // Output: "John Doe"
```

-
- **When to use:**
  Use `call` when you need to immediately call a function with a specific `this` value and pass arguments one at a time.

---

2. `apply()`

- **What it does:**
  Similar to `call()`, but instead of passing arguments one by one, you pass them as an array.

How it works:
Syntax:
javascript
Copy code

```
functionName.apply(thisValue, [arg1, arg2, ...]);
```

-
  - `functionName`: The function you want to call.
  - `thisValue`: The object you want `this` to refer to.
  - `[arg1, arg2, ...]`: Arguments passed in an array.

Example:
javascript
Copy code

```
const numbers = [1, 2, 3, 4, 5];
const maxNumber = Math.max.apply(null, numbers);
console.log(maxNumber); // Output: 5
```

Another example:
javascript
Copy code

```
const person = {
    fullName: function (city, country) {
        return `${this.firstName} ${this.lastName} from ${city},
${country}`;
    }
};

const person1 = { firstName: "Ali", lastName: "Khan" };

console.log(person.fullName.apply(person1, ["Lahore", "Pakistan"]));
// Output: "Ali Khan from Lahore, Pakistan"
```

-
- **When to use:**
  Use `apply` when you have arguments in the form of an array.

---

3. `bind()`

- **What it does:**
  The `bind()` method creates a new function with:
  - A specific `this` value.
  - Optional preset arguments. Unlike `call` and `apply`, **`bind` does not call the function immediately**; instead, it returns a new function that can be called later.

How it works:
Syntax:
javascript
Copy code

```
const newFunction = functionName.bind(thisValue, arg1, arg2, ...);
```

-

- `functionName`: The function you want to bind.
- `thisValue`: The object you want `this` to refer to.
- `arg1, arg2, ...`: Arguments to preset for the new function.

Example:
javascript
Copy code
```
const person = {
    fullName: function () {
        return `${this.firstName} ${this.lastName}`;
    }
};

const person1 = { firstName: "Ali", lastName: "Khan" };

const boundFunction = person.fullName.bind(person1);
console.log(boundFunction()); // Output: "Ali Khan"
```
With preset arguments:
javascript
Copy code
```
function greet(greeting, name) {
    console.log(`${greeting}, ${name}!`);
}

const sayHelloToAli = greet.bind(null, "Hello", "Ali");
sayHelloToAli(); // Output: "Hello, Ali!"
```

-
- **When to use:**
  Use `bind` when you need to:
  - Save a function for later use with a specific `this` value.
  - Create a reusable function with preset arguments.

Key Differences

Feature	`call()`	`apply()`	`bind()`
Execution	Calls the function immediately.	Calls the function immediately.	Returns a new function, does not execute immediately.

Arguments	Passed individually.	Passed as an array.	Preset arguments for later use.
Output	Function's return value.	Function's return value.	A new function reference.

Summary

- Use `call()` to call a function immediately with specific arguments.
- Use `apply()` to call a function immediately, but pass arguments as an array.
- Use `bind()` to create a new function with a fixed `this` and optional arguments, and call it later.

Introduction to Promises in JavaScript

A **Promise** in JavaScript is a special object used to handle asynchronous operations. It represents a **future value** or **result** of an operation that may complete successfully or fail. Promises help manage code execution when dealing with tasks like data fetching, file reading, or user interactions, where results might not be available immediately.

Why Use Promises?

- **Synchronous vs. Asynchronous Code:**
  - **Synchronous:** Executes line by line, blocking the next task until the current one completes.
  - **Asynchronous:** Non-blocking, allowing tasks to run in the background while continuing other operations.
- Promises help avoid **callback hell** (nested callbacks) and make the code cleaner and easier to maintain.

Promise States:

A Promise has three states:

1. **Pending:** Initial state, the result is not ready yet.
2. **Fulfilled:** The operation completed successfully.
3. **Rejected:** The operation failed.

Once a Promise is resolved or rejected, it becomes **settled** and cannot change state.

Creating a Promise:

```
const myPromise = new Promise((resolve, reject) => {
    let success = true; // Simulating a condition
    if (success) {
        resolve("Operation Successful!");
    } else {
        reject("Operation Failed!");
    }
});
```

Using `.then()`, `.catch()` and `.finally()`

- `.then()`: Executes when the promise is fulfilled.
- `.catch()`: Executes when the promise is rejected.
- `.finally()`: Executes after promise settles (either resolved or rejected).

```
myPromise
    .then(result => {
        console.log(result); // Runs if resolved
    })
    .catch(error => {
        console.log(error); // Runs if rejected
    })
    .finally(() => {
        console.log("Process Completed."); // Always runs
    });
```

Chaining Promises:

You can chain multiple `.then()` calls for a sequence of asynchronous tasks.

```
const fetchData = new Promise((resolve) => {
```

```
    setTimeout(() => resolve("Data Fetched"), 1000);
});

fetchData
    .then(data => {
        console.log(data);
        return "Processing Data";
    })
    .then(processedData => {
        console.log(processedData);
        return "Data Processed";
    })
    .catch(error => console.error("Error:", error));
```

Promise Methods:

- **`Promise.all()`**: Waits for all promises to resolve or any one to reject.

```
const p1 = Promise.resolve(10);
const p2 = Promise.resolve(20);
Promise.all([p1, p2]).then(values => console.log(values)); // [10, 20]
```

- **`Promise.race()`**: Returns the result of the first resolved or rejected promise.

```
const p3 = new Promise(resolve => setTimeout(() => resolve("Fast"), 100));
const p4 = new Promise(resolve => setTimeout(() => resolve("Slow"), 500));
Promise.race([p3, p4]).then(value => console.log(value)); // "Fast"
```

- **`Promise.allSettled()`**: Waits for all promises to settle (fulfilled or rejected).

```
const p5 = Promise.resolve("Success");
const p6 = Promise.reject("Error");
Promise.allSettled([p5, p6]).then(results => console.log(results));
```

Key Benefits of Promises:

- **Avoids callback hell**.
- **Easier error handling** with `.catch()`.
- **More readable asynchronous code**.

JavaScript Prototypes and Inheritance

JavaScript uses **prototypes** for inheritance, allowing objects to inherit properties and methods from other objects.

What is a Prototype?

A **prototype** is a special object attached to every JavaScript object. It serves as a blueprint for inheritance, where an object can access properties and methods defined on its prototype.

- Every object in JavaScript has an internal link (`[[Prototype]]`) to another object, called its prototype.
- You can access this using the `__proto__` property (not recommended for modern usage).

Prototype Example:

```
const person = {
   greet: function() {
      console.log("Hello!");
   }
};

const student = Object.create(person); // student inherits from person
student.greet(); // Output: Hello!
```

Here, the `student` object inherits the `greet` method from the `person` object through its prototype.

Prototype Chain:

The **prototype chain** is the mechanism where objects inherit from their prototypes. If a property is not found in an object, JavaScript will look up the chain to find it.

```
const animal = {
   type: "Mammal"
};

const dog = Object.create(animal);
dog.breed = "Labrador";

console.log(dog.breed); // Labrador
console.log(dog.type);  // Mammal (inherited)
console.log(dog.toString()); // Inherited from Object.prototype
```

Key Points About Prototype Chain:

- If a property is not found in the object, it looks in its prototype.
- The chain continues until it reaches `Object.prototype`, the top-level prototype.
- If the property isn't found there, `undefined` is returned.

How to Set Prototypes:

Using `Object.create()` (Recommended):

```
 const car = {
   wheels: 4,
   drive: function() {
      console.log("Driving...");
   }
};

const sportsCar = Object.create(car);
console.log(sportsCar.wheels); // 4 (inherited)
sportsCar.drive();             // Driving...
```

1.

Using Constructor Functions (Older Way):

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

Person.prototype.greet = function() {
   console.log(`Hello, ${this.name}`);
```

```
};

const john = new Person("John");
john.greet();  // Hello, John
```

2.

Using `class` Syntax (Modern Way):

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

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

const jane = new Person("Jane");
jane.greet();  // Hello, Jane
```

3.

Inheritance in JavaScript:

Inheritance allows one object (child) to inherit properties and methods from another object (parent).

```
class Animal {
    constructor(name) {
        this.name = name;
    }

    speak() {
        console.log(`${this.name} makes a noise.`);
    }
}

// Child class inheriting from Animal
class Dog extends Animal {
    speak() {
        console.log(`${this.name} barks.`);
```

```
  }
}

const myDog = new Dog("Buddy");
myDog.speak();  // Buddy barks.
```

Key Concepts of Inheritance:

- **Base Class:** The parent class (`Animal` in the example above).
- **Derived Class:** The child class (`Dog`).
- **`extends`:** Keyword used for inheritance.
- **`super`:** Used to call the parent class constructor or methods.

Summary:

- **Prototypes** allow objects to inherit properties from other objects.
- **Prototype chain** helps look up properties up the chain if not found directly on the object.
- **Inheritance** is simplified using `class` and `extends`.
- **Object.create()** and **Constructor functions** are older ways to work with prototypes.

Let me know if you need more practice examples!

JavaScript Modules (import/export)

JavaScript **modules** allow you to break your code into reusable files, making it easier to maintain, debug, and organize. They let you export functions, objects, or variables from one file and import them into another.

Why Use Modules?

- **Code Organization:** Break code into smaller, manageable parts.
- **Reusability:** Use the same code across multiple files.
- **Encapsulation:** Keep variables and functions private unless explicitly exported.

Exporting in JavaScript Modules:

Named Export:

You can export multiple values from a file using `export`.

```
// file: mathUtils.js
export const PI = 3.14159;

export function add(a, b) {
   return a + b;
}

export function multiply(a, b) {
   return a * b;
}
```

Default Export:

You can export a single default value per file using `export default`.

```
// file: greeting.js
export default function greet(name) {
   console.log(`Hello, ${name}!`);
}
```

Importing in JavaScript Modules:

Importing Named Exports:

You need to use **curly braces** to import specific items.

```
// file: main.js
import { PI, add, multiply } from './mathUtils.js';

console.log(PI);              // 3.14159
console.log(add(2, 3));       // 5
console.log(multiply(4, 5));  // 20
```

Importing Default Exports:

No curly braces are required when importing a default export.

```
// file: main.js
import greet from './greeting.js';

greet('Ali');  // Hello, Ali!
```

Renaming Imports and Exports:

Renaming Exports:

```
// file: circle.js
export const radius = 5;
export function calculateArea(r) {
    return Math.PI * r * r;
}

export { calculateArea as area };
```

Renaming Imports:

```
// file: main.js
import { area as circleArea, radius } from './circle.js';

console.log(circleArea(radius));  // Area of the circle
```

Combining Named and Default Exports:

```
// file: shapes.js
export const square = (side) => side * side;
export default function rectangle(length, width) {
    return length * width;
}

// file: main.js
import rectangle, { square } from './shapes.js';

console.log(rectangle(4, 5));  // 20
console.log(square(4));        // 16
```

Import All with `*`:

You can import everything from a module using `*`.

```
// file: main.js
import * as MathUtils from './mathUtils.js';

console.log(MathUtils.PI);          // 3.14159
console.log(MathUtils.add(2, 3));   // 5
```

Key Differences Between Named and Default Exports:

Feature	Named Export	Default Export
Exporting	Multiple items can be exported.	Only one default export per file.
Syntax	`export const a = 5;`	`export default a = 5;`
Importing	Must use `{}` for importing.	No `{}` required for importing.
Renaming	Can rename during import/export.	Can rename during import.

Browser vs Node.js Usage:

Browser: Use `<script type="module">` in HTML files to use modules.

```
<script type="module" src="main.js"></script>
```

-
- **Node.js:** Use `.mjs` extension or set `"type": "module"` in `package.json`.

Summary:

- **Named Exports:** Export multiple values from a module.
- **Default Exports:** Export a single main value.
- **Import:** Use `import` to bring values from other modules.

- **Best Practice:** Use modules to keep code modular, reusable, and clean.

Would you like practice problems on modules? 😊