1 `let` and `const` (Block Scope Variables)

- `let` → Can be reassigned, block-scoped.
- `const` → Cannot be reassigned, block-scoped.
- `var` (old) → Function-scoped, can cause unintended bugs.

2 Template Literals (String Interpolation)

- Use **backticks** `` ` `` instead of quotes.
- Insert variables using `${}`.
- Supports **multiline strings** without `¥n`.

3 Arrow Functions (`=>`)

- Shorter syntax for functions.
- No `this` binding (inherits from surrounding scope).

```
const arr1 = [1, 2, 3];
const arr2 = [...arr1, 4, 5]; // [1, 2, 3, 4, 5]
const nums = [1, 2, 3];
const [first, second] = nums;
console.log(first, second); // 1 2
```

`this` in Regular Functions

- `this` refers to the object that called the function.
- If called inside an **object**, it refers to that object.
- If called inside a **global function**, it refers to `window` (in browsers) or `global` (in Node.js).

```
const obj = {
  name: "Alice",
  greet: function() {
    console.log(this.name); // 'this' refers to obj
  }
};
obj.greet(); // Alice
```

□ `this` in Arrow Functions

- **Arrow functions do not have their own `this`.**
- They **inherit `this` from their surrounding (lexical) scope.**
- Useful for keeping `this` consistent inside callbacks.

Here, `this` refers to the surrounding scope (`window` or `global`), **not the object**.

```
const obj = {
   name: "Alice",
   greet: () => {
      console.log(this.name); // ✕Undefined, because arrow function doesn't bind its own `this`
   }
};
obj.greet();
```

****USES

Arrow Functions in Callbacks

- Arrow functions are **useful in callbacks** where `this` might change.

```
const user = {
   name: "Bob",
   greet: function() {
      setTimeout(function() {
         console.log(this.name); // ✕Undefined (this refers to window)
      }, 1000);
   }
};
user.greet();
```

`this` inside `setTimeout` refers to `window`, not `user`.

Arrow:

```
const user = {
   name: "Bob",
   greet: function() {
```

```
    setTimeout(() => {
        console.log(this.name); // ✅Bob (inherits `this` from greet function)
    }, 1000);
  }
};
user.greet();
```

Here, the arrow function **inherits** `this` **from** `greet()`, which refers to `user`.

`this` and Lexical Scoping in Arrow Functions

Arrow functions are special because they **inherit** `this` **from their surrounding scope** — this is tied to lexical scoping.

So, **arrow functions** inherit `this` from where they are **defined**, not where they are called. This is what makes them different from regular functions.

When an arrow function is not inside another function, it still retains `this` from its **lexical** scope, which means it will refer to the value of `this` from the surrounding environment where it was defined.

```
const name = "Global";

const arrowFunction = () => {
  console.log(this.name); // 'this' refers to the lexical scope, which is the global scope
};

arrowFunction(); // undefined (because `this.name` in global scope is undefined)
```

In the example above, since `arrowFunction` is **not inside any object or function**, it will inherit `this` from the **global scope**.

- In a browser environment, the global `this` is the `window` object. But since there is no `name` property on the `window` object (unless explicitly added), you will get `undefined`.

```
const obj = {
  name: "Object Name",
  arrowFunction: () => {
    console.log(this.name); // 'this' is inherited from the global scope, not the `obj` object
  }
};
```

obj.arrowFunction(); // undefined

Here, `this` inside the arrow function will **not refer to the** `obj` **object** but to the **global context**, which is why `this.name` is `undefined` since `name` isn't defined in the global scope.

- **For a regular function,** `this` **refers to the object it is called on** (the **calling object**). If the function is called as a method of an object, `this` will be that object.
- **If a regular function is called in the global context**, `this` refers to the **global object** (`window` in browsers).
- **In strict mode**, `this` is `undefined` when a regular function is called in the global context.

1. Arrays

- Arrays are **ordered collections** of values.
- You can store different types of data in an array (e.g., numbers, strings, objects).

Syntax:

```
javascript
CopyEdit
let fruits = ["Apple", "Banana", "Cherry"];
```

Common Array Methods:

- `push()` – Adds an element to the end of the array.

```
javascript
CopyEdit
fruits.push("Mango"); // ["Apple", "Banana", "Cherry", "Mango"]
```

- `pop()` – Removes the last element from the array.

```
javascript
CopyEdit
fruits.pop(); // Removes "Mango"
```

- `shift()` – Removes the first element.

```
javascript
CopyEdit
fruits.shift(); // Removes "Apple"
```

- `unshift()` – Adds an element to the beginning.

```
javascript
CopyEdit
fruits.unshift("Orange"); // ["Orange", "Banana", "Cherry"]
```

- `forEach()` – Iterates over all elements in the array.

```
javascript
CopyEdit
fruits.forEach(fruit => console.log(fruit));
// Output: "Banana", "Cherry"
```

- `map()` – Creates a new array by applying a function to each element.

```
javascript
CopyEdit
let lengths = fruits.map(fruit => fruit.length); // [6, 6, 6]
```

- `filter()` – Creates a new array with elements that pass a test.

```
javascript
CopyEdit
let longFruits = fruits.filter(fruit => fruit.length > 5); // ["Banana", "Cherry"]
```

2. Loops

- `for` **loop** – Repeats a block of code a set number of times.

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

- `while` **loop** – Repeats while a condition is true.

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

- `for...of` **loop** – Iterates over **values** in an array or iterable object.

```
javascript
CopyEdit
for (let fruit of fruits) {
    console.log(fruit); // Output: "Banana", "Cherry"
}
```

- `for...in` **loop** – Iterates over **keys** in an object.

```
javascript
CopyEdit
let person = { name: "John", age: 30 };
for (let key in person) {
```

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

3. String Functions

- `split()` – Splits a string into an array based on a delimiter.

```
javascript
CopyEdit
let text = "apple,banana,orange";
let fruitsArray = text.split(","); // ["apple", "banana", "orange"]
```

- `trim()` – Removes whitespace from both ends of a string.

```
javascript
CopyEdit
let greeting = "   Hello, World!   ";
let trimmed = greeting.trim(); // "Hello, World!"
```

- `toUpperCase()` – Converts a string to uppercase.

```
javascript
CopyEdit
let lowerCase = "hello";
let upperCase = lowerCase.toUpperCase(); // "HELLO"
```

- `toLowerCase()` – Converts a string to lowercase.

```
javascript
CopyEdit
let mixedCase = "HeLLo WoRLd";
let lowerCase = mixedCase.toLowerCase(); // "hello world"
```

- `indexOf()` – Returns the index of the first occurrence of a substring, or `-1` if not found.

```
javascript
CopyEdit
let sentence = "I love JavaScript";
let index = sentence.indexOf("JavaScript"); // 7
```

- `includes()` – Checks if a string contains a specified substring.

```
javascript
CopyEdit
let sentence = "I love JavaScript";
let containsJS = sentence.includes("JavaScript"); // true
```

- `replace()` – Replaces a substring with another substring.

```
javascript
CopyEdit
let sentence = "Hello, John!";
```

```
let newSentence = sentence.replace("John", "Jane"); // "Hello, Jane!"
```

- `substring()` – Extracts a part of a string between two indexes.

```
javascript
CopyEdit
let word = "Hello, World!";
let sub = word.substring(0, 5); // "Hello"
```

4. Object Basics

- Objects are collections of key-value pairs (properties).

Syntax:

```
javascript
CopyEdit
let person = {
    name: "John",
    age: 30,
    greet: function() {
        console.log("Hello");
    }
};
```

Accessing Object Properties:

- **Dot notation**: `person.name`
- **Bracket notation**: `person["name"]`

5. Arrow Functions

- Arrow functions allow you to write shorter function expressions. They inherit the `this` value from the surrounding scope.

Syntax:

```
javascript
CopyEdit
const add = (a, b) => a + b;
console.log(add(5, 3)); // Output: 8
```

6. Template Literals

- Template literals allow you to embed expressions within strings using `${}`.

Syntax:

```
javascript
CopyEdit
let name = "Alice";
```

```
let greeting = `Hello, ${name}!`; // Output: "Hello, Alice!"
```

7. Destructuring

- Destructuring allows you to extract values from arrays or objects into variables.

For Arrays:

```
javascript
CopyEdit
let colors = ["red", "green", "blue"];
let [first, second] = colors;
console.log(first); // "red"
console.log(second); // "green"
```

For Objects:

```
javascript
CopyEdit
let person = { name: "John", age: 30 };
let { name, age } = person;
console.log(name); // "John"
console.log(age); // 30
```

8. Default Parameters

- You can assign default values to function parameters.

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

9. Spread and Rest Operators

- **Spread operator (**`...`**)**: Copies elements from one array or object to another.

```
javascript
CopyEdit
let arr1 = [1, 2, 3];
let arr2 = [...arr1, 4, 5]; // [1, 2, 3, 4, 5]
```

- **Rest operator (**`...`**)**: Collects arguments into an array.

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

```
}

console.log(sum(1, 2, 3)); // Output: 6

====================
```

cmp;x

Key Points to Remember:

1. **Single-threaded**: JavaScript runs in a single thread, meaning it can only execute one task at a time.
2. **Asynchronous with Event Loop**: Even though JavaScript is single-threaded, it can handle async operations without blocking the execution of the rest of the code.
3. **Event Queue**: Asynchronous operations (like `setTimeout`, Promises, etc.) are placed in the event queue after their delay or async operation is complete.
4. **Event Loop**: The event loop checks if the call stack is empty and, if so, pushes tasks from the event queue onto the stack.

How Callback Functions Work:

A callback function is just a function that is passed as an argument to another function.

```
function greet(name, callback) {
   console.log("Hello " + name);
   callback();  // Call the callback function passed as argument
}

// Define the callback function inline inside the greet function call
greet("John", function() {
   console.log("Goodbye!");
});
Hello John
Goodbye!
```

CHAINING

```
function task1(callback) {
   console.log("Task 1 completed");
   callback();
}

function task2(callback) {
   console.log("Task 2 completed");
   callback();
}

function task3(callback) {
   console.log("Task 3 completed");
   callback();
}
```

```
task1(function() {
  task2(function() {
    task3(function() {
      console.log("All tasks completed");
    });
  });
});
```

The `task1`, `task2`, and `task3` functions are asynchronous tasks. Each task accepts a callback function that's executed after the task completes.

- After `task1` is done, it calls the callback, which triggers `task2`. After `task2` is done, it calls the callback to trigger `task3`, and so on.
- This forms a **chain of callbacks**.
- This style can get difficult to manage as the number of tasks grows, leading to "callback hell."

Task 1 completed

Task 2 completed

Task 3 completed

All tasks completed

Promises in JavaScript

A **Promise** is a JavaScript object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. Promises help you manage asynchronous operations in a much more readable and structured way, avoiding the "callback hell" that happens when you nest callbacks inside each other.

What Is a Promise?

A Promise has three possible states:

1. **Pending:** The operation is still in progress.
2. **Resolved (Fulfilled):** The operation completed successfully, and a result is available.
3. **Rejected:** The operation failed, and an error is returned.

A **Promise** allows you to chain `.then()` and `.catch()` methods to handle the resolved value or rejection.

Creating a Promise

You can create a Promise using the `new Promise()` constructor:

```
let myPromise = new Promise(function(resolve, reject) {
  let condition = true; // Simulating an operation
  if (condition) {
```

```
    resolve("Success!"); // Operation was successful
  } else {
    reject("Failed!"); // Operation failed
  }
});
```

Using Promises with `.then()` and `.catch()`

After a Promise is created, you can use `.then()` to handle success (when the Promise is resolved) and `.catch()` to handle errors (when the Promise is rejected).

```
myPromise
  .then(function(result) {
    console.log(result);  // If Promise is resolved, log success message
  })
  .catch(function(error) {
    console.log(error);   // If Promise is rejected, log error message
  });
```

- A new Promise is created that simulates an operation with the variable `success`.
- If `success` is true, it calls `resolve("Task was successful!")`.
- If `success` is false, it calls `reject("Task failed!")`.
- `.then()` is used to handle the successful result, and `.catch()` is used to handle errors.

PROMISE CHAINING

```
let task1 = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("Task 1 completed"), 1000);
});

let task2 = function(previousResult) {
  return new Promise(function(resolve, reject) {
    setTimeout(() => resolve(previousResult + " -> Task 2 completed"), 1000);
  });
};
```

```
let task3 = function(previousResult) {
    return new Promise(function(resolve, reject) {
        setTimeout(() => resolve(previousResult + " -> Task 3 completed"), 1000);
    });
};

task1
    .then(function(result) {
        console.log(result);  // "Task 1 completed"
        return task2(result); // Call next task and return the result
    })
    .then(function(result) {
        console.log(result);  // "Task 1 completed -> Task 2 completed"
        return task3(result); // Call next task and return the result
    })
    .then(function(result) {
        console.log(result);  // "Task 1 completed -> Task 2 completed -> Task 3 completed"
    })
    .catch(function(error) {
        console.log(error);
    });
```

`response.json()`: Parses JSON response from the server to a JavaScript object (often used with `fetch()`).

- `JSON.stringify()`: Converts a JavaScript object into a JSON-formatted string.
- `JSON.parse()`: Converts a JSON-formatted string back into a JavaScript object.

What is `async/await`?

- `async`: This is a keyword that you add before a function to make it asynchronous. When a function is marked as `async`, it automatically returns a **Promise**, and inside this function, you can use `await`.

- `await`: This keyword is used to pause the execution of an async function until the Promise resolves (or rejects). It can only be used inside functions marked as `async`.

```
// Define the async function
async function fetchData() {
    try {
        // Fetch data from an API
        let response = await fetch('https://jsonplaceholder.typicode.com/posts');

        // Convert the response to JSON format
        let data = await response.json();

        // Log the fetched data
        console.log(data);
    } catch (error) {
        // If there's an error, log it
        console.log('Error:', error);
    }
}

// Call the async function
fetchData();
```