Assume you are experienced in answering in many interviews and you have enough knowledge on (JavaScript Recap: Variables, Conditional Statements, Loops, Functions & Reusability) then answer the following question "What are the different ways to declare variables in JavaScript?"

Great question — this is a very common one in interviews!

In **JavaScript**, there are **three primary ways** to declare variables:

✅ 1. `var`

- Introduced in the earliest versions of JavaScript.
- **Function-scoped** or **globally scoped** if declared outside any function.
- Can be **redeclared and updated**.
- **Hoisted** to the top of its scope and initialized as `undefined`.

> ⚠️ `var` can cause issues due to its less predictable behavior, so it's not commonly recommended in modern code.

Example:

```
var name = "Alice";

var name = "Bob";  // allowed
```

✅ 2. `let`

- Introduced in ES6 (2015).
- **Block-scoped** — limited to the enclosing `{}` block.
- Can be **updated**, but cannot be **redeclared** in the same scope.
- **Hoisted** but not initialized — you can't access it before its declaration (Temporal Dead Zone).

Example:

```
let age = 25;

age = 26;     // allowed

// let age = 30; // ❌ Error: already declared
```

✅ 3. `const`

- Also introduced in ES6.
- **Block-scoped** — limited to the enclosing block.
- Cannot be **redeclared** or **updated** (must be assigned when declared).
- Especially used for values you don't intend to change.
- Works well for **constants** or **immutable references**, like arrays and objects (the content of the object can still be modified).

Example:

```
const pi = 3.1415;

// pi = 3.14;    // ❌ Error: cannot reassign
```

💡 **Summary:**

Keyword	**Scope**	**Redeclare?**	**Reassign?**	**Hoisted?**
`var`	Function	✅ Yes	✅ Yes	✅ Yes
`let`	Block	❌ No	✅ Yes	✅ (TDZ)
`const`	Block	❌ No	❌ No	✅ (TDZ)

💭 **Why this matters in interviews:**
Interviewers often want to see that you:

- Understand **variable scope**.
- Know the **differences** between `var`, `let`, and `const`.
- Can explain **hoisting** and the **temporal dead zone (TDZ)**.
- Make informed choices in real-world code.

🎯 Let me know if you'd also like a quick **example demonstrating hoisting** or a **small quiz** on this topic — happy to help!

What is the difference between `var`, `let`, and `const`?

Great question — this is one of the most common interview questions in JavaScript. Here's a clear, concise answer you can give:

In JavaScript, `var`, `let`, and `const` are used to declare variables, but they have important differences:

✅ 1. Scope:

- `var`: **Function-scoped** or global-scoped if declared outside any function.
- `let` and `const`: **Block-scoped**, i.e. limited to the nearest `{ ... }` block, `if`, or `for` loop.

✅ 2. Hoisting & Initialization:

- `var`: **Hoisted** and initialized with `undefined`. You can access a `var` variable before its declaration without error.
- `let` and `const`: **Hoisted but not initialized** — they stay in a "temporal dead zone" until the declaration is reached. Accessing them before declaration throws a `ReferenceError`.

✅ 3. Reassignment:

- `var`: **Can be updated and re-declared** in the same scope.
- `let`: **Can be updated** but cannot be re-declared in the same scope.
- `const`: **Cannot be updated or re-declared** — must be initialized at the time of declaration.

✅ 4. Typical Usage:

- `var`: Considered outdated — mostly avoided in modern code.
- `let`: Preferred for **variables that need to change**.
- `const`: Preferred for **constants** or when you want to ensure the binding stays the same.

💡 Example to illustrate the scope difference:

```
if (true) {
  var x = 10;
  let y = 20;
  const z = 30;
}
console.log(x); // 10 (function or global scoped)
console.log(y); // Error
console.log(z); // Error
```

✅ **Summary:**
Always use `let` or `const` in modern JavaScript for better scoping and to avoid bugs. `const` is most restrictive and preferred when you don’t intend to reassign a variable, while `let` is used for variables that will change.

Let me know if you’d also like a quick one-liner version for interview purposes!

How do you write a simple if-else statement in JavaScript?

Great question — this is a classic one often asked to check your basic JavaScript knowledge.

Here’s a **simple example of an `if-else` statement in JavaScript**:

```
let score = 85;

if (score >= 50) {
  console.log("You passed!");
} else {
  console.log("You failed.");
}
```

✅ **Explanation**:

- `if (score >= 50)` is the **condition** being checked.
- If the condition is `true`, it runs the first block (`You passed!`).
- Otherwise (`else`), it runs the second block (`You failed.`).

That’s all there is to a basic `if-else`! In an interview, you can also mention that `if-else` can be nested or extended into `else if` for multiple conditions. Let me know if you'd also like to see an example of that.

What is the difference between `==` and `===` in JavaScript?

That’s a great question — interviewers often use this to gauge your understanding of JavaScript’s nuances.

Here’s a clear, concise way to answer:

In JavaScript:

`==` (loose equality)
Compares two values **after converting them to a common type** (this is called *type coercion*).
Example:

```
 5 == "5"; // true — because "5" is coerced to number 5
null == undefined; // true — special loose-equality rule
```

-

`===` (strict equality)
Compares two values **without any type conversion** — they must be the same type *and* value.
Example:

```
 5 === "5"; // false — number is not the same type as string
5 === 5;   // true
null === undefined; // false
```

-

✅ **When to use which?**
In most cases, always use `===` to avoid unexpected type coercion and produce more predictable, error-free code.

💡 **Summary:**

- `==` → converts types before comparison
- `===` → checks types and values directly

Would you also like a quick demo snippet or examples of pitfalls?

How does a `for` loop work? Write an example.

Great question — this is one of the most fundamental concepts in JavaScript!

✅ How a `for` loop works:

A `for` loop is used to repeat a block of code a specific number of times. It consists of three main parts:

1. **Initialization** — executed **once** at the beginning.
2. **Condition** — checked **before each iteration**; if `true`, the loop continues; if `false`, the loop stops.
3. **Increment/Decrement** — happens **after each iteration**.

In short:

```
for (initialization; condition; increment) {
  // code block to execute repeatedly
}
```

✅ Example:

Here’s a simple `for` loop that prints numbers `1` to `5`:

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

💡 What happens here?

- **Initialization:** `let i = 1` → Start with `i` as 1.
- **Condition:** `i <= 5` → Keep looping as long as `i` is less than or equal to 5.
- **Increment:** `i++` → Increase `i` by 1 after each iteration.

✅ Summary:

A `for` loop is an efficient and compact way to repeat code a fixed number of times. It's commonly used when you already know the number of iterations you need.

Let me know if you'd also like to see examples with arrays or `break` and `continue`!

How do you write a while loop?

Great question — here's a clear and interview-style answer:

💡 Answer:

A `while` loop in JavaScript is used to repeat a block of code **as long as a specified condition is true**.

🧠 Syntax:

```
while (condition) {
  // code to execute repeatedly
}
```

🔍 Example:

```
let count = 0;      // Initialization
while (count < 5) { // Condition
  console.log(count);
  count++;          // Increment
}
```

✅ **Explanation:**

1. The `while` loop first **evaluates the condition** (`count < 5`).
2. If it’s `true`, it **executes the code block**.
3. After execution, it **checks the condition again**.
4. This process continues until the condition is `false`.

⚠️ **Key point:**

Always make sure to update the loop variable (`count++`), or you'll end up with an **infinite loop**!

💭 **In short:**
A `while` loop keeps running as long as its condition stays true. Be careful to eventually make the condition false!

Would you also like an example comparing it to a `for` loop?

What is a function in JavaScript?

A **function in JavaScript** is a **reusable block of code** that is **defined once** and can be **executed (invoked) multiple times** throughout your program. Functions help you **organize your code**, **avoid repetition**, and make your code **more modular and maintainable**.

Here are a few key points about functions in JavaScript:

✅ **Definition** – You can define a function using the `function` keyword:

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

✅ **Invocation (Calling)** – Once defined, you invoke (call) the function:

```
greet('Alice'); // Output: Hello, Alice!
```

✅ **Parameters & Arguments** – Functions can accept **parameters** (like `name` above), and you pass **arguments** when you call the function.

✅ **Return Values** – Functions can return a value using the `return` keyword:

```
function add(a, b) {
  return a + b;
}
let sum = add(2, 3); // sum = 5
```

✅ **Reusable & Modular** – By writing logic inside a function, you can use that code **as often as needed** without repeating yourself.

✅ **First-Class Citizens** – Functions in JavaScript can be:

- Assigned to variables
- Passed as arguments to other functions
- Returned as a value from other functions

💡 **In short:**
A **function** in JavaScript is a **self-contained, named set of instructions** that you can **call repeatedly** to perform a specific task, helping you write **clearer and DRY (Don't Repeat Yourself)** code.

How do you define and call a function?

Great question! Let’s break it down clearly.

✅ Defining a function in JavaScript

A **function** is a block of reusable code that can take input, process it, and optionally return an output. In JavaScript, you can define a function using the `function` keyword like this:

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

Here:

- `function` is the keyword.
- `greet` is the name of the function.
- `name` is a **parameter** — a variable you pass into the function.

✅ Calling (or invoking) a function

Once the function is defined, you can **call** it by writing its name followed by parentheses `()` and optional arguments inside:

```
// Call the function with an argument
greet('Alice');  // Output: "Hello, Alice!"
greet('Bob');    // Output: "Hello, Bob!"
```

🎯 Summary

✅ **Define** a function with `function functionName(parameters) { /* code */ }`.
✅ **Call** it with `functionName(arguments)`.

That's it — defining and calling a function allows you to reuse code whenever you need it!

Let me know if you'd also like to see examples with ES6 arrow functions or any variations.

What is a function parameter?

Great question — here's a clear and concise way to explain **function parameters** in a JavaScript interview:

💡 **A function parameter is a named variable listed in a function's definition that acts as a placeholder for a value that will be passed into the function when it is invoked.**

✅ It allows you to make your function **more dynamic and reusable**, because the behavior of the function can change depending on the argument passed.

Here’s a quick example:

```
function greet(name) {  // "name" is a parameter
  console.log("Hello, " + name + "!");
}

// Passing different arguments when calling:
greet("Alice");  // Hello, Alice!
greet("Bob");    // Hello, Bob!
```

In this example:

- `name` is the **parameter**.
- `"Alice"` and `"Bob"` are **arguments**.

💭 **Key point for interviews:**
Explain that **parameters are specified in the function definition** and **arguments are the actual values you pass in during the call**. That subtle difference is often what interviewers look for.

Would you also like me to show variations like default parameters or rest parameters? Let me know!

How do you return a value from a function?

Great question — this is a very common one in JavaScript interviews. Here’s a clear and concise way to answer:

> In JavaScript, you return a value from a function using the `return` keyword. Anything specified after the `return` keyword will be sent back to the caller when the function is invoked. Once a `return` statement is reached, the function stops executing further code.

✅ **Example:**

```
function add(a, b) {
  return a + b; // the sum is returned to the caller
}
```

let result = add(5, 3); // result will be 8

💡 **Key points to mention in an interview**:

- Once `return` is called, the function ends immediately.
- If you don't explicitly `return` a value, the function returns `undefined`.
- You can return any type of value — numbers, strings, objects, arrays, other functions, etc.

That's a straightforward and solid answer for most interviewers. Would you also like a more advanced example with conditionals or loops?

What is the scope of variables declared with var, let, and const?

That's a great fundamental question — and one that often comes up in interviews. Here's a clear, concise answer you can give:

In JavaScript, the scope of variables depends on how they're declared:

✅ `var`

- **Scope:** *Function-scoped* or *globally scoped* if declared outside any function.
- **Hoisting:** `var` declarations are *hoisted* to the top of their scope and initialized as `undefined`.
- **Block level:** `var` does **not** respect block scope (`if`, `for`, etc.), so a `var` inside a block is still accessible outside that block.

✅ `let`

- **Scope:** *Block-scoped* — only accessible inside the nearest `{ }` (e.g. inside `if`, `for`, or a function).
- **Hoisting:** `let` is also hoisted but stays in the *temporal dead zone* until the declaration is reached.
- **Redeclaration:** Cannot be re-declared in the same scope.

✅ `const`

- **Scope:** *Block-scoped*, just like `let`.
- **Hoisting:** Also hoisted but not accessible before declaration (temporal dead zone).
- **Immutable binding:** The *reference* cannot be reassigned. (If it's an object or array, its contents can still change.)

💡 Key takeaway:

- `var` → function scope, less predictable in block structures
- `let` and `const` → block scope, safer and more predictable

That's a neat summary you can use. Let me know if you'd also like a short code example to demonstrate this!

Explain the difference between function declaration and function expression.

Great question — this is a very common interview topic!

Here's a clear, concise answer you can use:

Function Declaration

A *function declaration* defines a named function using the `function` keyword. It looks like this:

```
function sayHello() {
  console.log('Hello');
}
```

Key points about function declarations:

- **Hoisted** entirely — you can call the function *before its definition* in the code.
- Always has a **name**.
- Defined at the top of its scope at compile time.

Function Expression

A *function expression* is when you create a function as part of an expression — often assigned to a variable:

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

Key points about function expressions:

- **Not hoisted** — cannot be invoked before the assignment.
- Can be **anonymous** or named.
- Allows flexibility (e.g. passing as arguments, creating closures).

Quick Comparison Table:

Feature	Function Declaration	Function Expression
Hoisted?	Yes	No

Requires a name?	Yes	Optional
Defined at	Parse time (before code runs)	Run time (when execution reaches it)
Typical use cases	Global helper functions	Callbacks, IIFE, closures

---

Summary:
Function declarations are fully hoisted and can be called before they're defined. Function expressions require the assignment to happen first before you can use them. This difference in hoisting behavior is often the most practical thing to remember in interviews. Let me know if you'd also like a short code demo for an interview whiteboard session!

What are arrow functions? How are they different from traditional functions?

Great question — this is a very common topic in JavaScript interviews! Here's a clear and structured way to answer:

Arrow functions (introduced in ES6) are a concise way to define functions in JavaScript using the `=>` (fat arrow) syntax. They provide a shorter, more readable way to write anonymous functions, especially for small utility or callback functions.

Example of an arrow function:

```
// Arrow function
const add = (a, b) => a + b;
```

Compared to a traditional function expression:

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

✅ Key differences between arrow functions and traditional functions:

1. **Syntax & Conciseness**
   Arrow functions have a much more concise syntax, especially when returning a single expression.

`this` Binding

Arrow functions **do not have their own `this`**.

They **lexically bind** to the `this` value of the enclosing scope.

→ Traditional functions get their own `this` based on how they are invoked.

```
 function Person(name) {
  this.name = name;
  this.sayHi = function() {
    setTimeout(function() {
      // 'this' is not the Person object here; it's the global/window object.
      console.log(this.name); // undefined
    }, 1000);
  };
}
```

```
function PersonArrow(name) {
  this.name = name;
  this.sayHi = function() {
    setTimeout(() => {
      // Arrow function inherits 'this' from the enclosing scope
      console.log(this.name); // Correctly logs the name
    }, 1000);
  };
}
```

2.
3. **No `arguments` Object**
   Arrow functions don’t have their own `arguments` object.
   → If you need arguments inside an arrow, use the rest parameter (`...args`).

4. **Cannot Be Used as Constructors**
   Arrow functions cannot be invoked with `new`.
   → Traditional functions can act as constructors and can be invoked with `new`.

5. **No `prototype`**
   Arrow functions don’t have a `prototype` property.

✅ When to use arrow functions?

- When you need a short, inline callback.
- When you want to preserve the `this` context of the enclosing scope.
- When working with functional patterns like `map`, `filter`, `reduce`, or event listeners.

💡 **Summary:**
Arrow functions are great for short, non-method callbacks where you want lexical `this`. Traditional functions are better if you need your own `this`, `arguments`, or plan to use them as constructors.

Would you also like some quick practice questions or code examples to test these differences?

How would you create a reusable function to add two numbers?

That’s a great question — and one you’d often get in interviews to check your basics and your understanding of **functions** and **reusability** in JavaScript.

Here’s how you can do it:

✅ **Step 1: Create a named function** that accepts two parameters.
✅ **Step 2: Return the sum of the two numbers.**
✅ **Step 3: Reuse this function wherever you need it.**

Here’s a simple implementation:

```
// Define the reusable function
function addTwoNumbers(a, b) {
  return a + b;
}

// Example usage
console.log(addTwoNumbers(5, 10)); // Output: 15
console.log(addTwoNumbers(-3, 7)); // Output: 4
```

🎯 Why this is reusable?

✅ It’s **independent** of any particular data — you pass in the numbers you want.
✅ It **returns a value**, so you can use that value later (e.g. store in a variable, pass to another function, etc.).
✅ It can be invoked **as many times as you want** with different arguments.

🧠 Variations you could also mention:

✅ **Arrow function version** (ES6 syntax):

```
const addTwoNumbers = (a, b) => a + b;
```

✅ **Input validation** if you want to be extra cautious:

```
function addTwoNumbers(a, b) {
  if (typeof a !== 'number' || typeof b !== 'number') {
    throw new Error('Both arguments must be numbers');
  }
  return a + b;
}
```

💡 **Conclusion:**
By creating a standalone function like `addTwoNumbers()`, you promote code reusability, reduce repetition, and make the code more readable and testable — which is exactly what interviewers look for.

Would you also like me to show examples with loops or conditional checks inside? Let me know!

What is the purpose of the default case in a switch statement?

The **purpose of the `default` case in a `switch` statement** is to act as a **fallback or catch-all** when none of the specified `case` values match the expression.

In other words:

- If the value you're switching on doesn't match any of the listed `case` labels, the code inside the `default` block will execute.
- It's optional but **recommended** to include a `default` so that you can handle unexpected or out-of-range values gracefully.

Example:

```
let color = "purple";

switch (color) {
  case "red":
    console.log("The color is red.");
    break;
  case "blue":
    console.log("The color is blue.");
    break;
  default:
    console.log("Color not recognized.");
}
```

Here, since `color` is `"purple"` and it doesn't match any of the cases (`"red"`, `"blue"`), the `default` case will run and print:

```
Color not recognized.
```

✅ **Key point:** The `default` case improves the robustness of your code by handling any situation that's not explicitly handled by other `case` branches.

Explain how you can exit a loop prematurely.

Great question — this is something interviewers often ask to check your control flow understanding in JavaScript.

In JavaScript, you can exit a loop prematurely using the `break` statement.

Here's a clear explanation you could give:

> "In JavaScript, the `break` statement allows you to exit a loop before it has iterated over all its elements or met its natural ending condition. When a `break` is encountered inside a `for`, `while`, or `do-while` loop, it immediately stops the loop's execution and transfers control to the first statement after the loop block.
>
> For example:

```
for (let i = 0; i < 10; i++) {
  if (i === 5) {
    break; // Exit the loop when i reaches 5
  }
  console.log(i); // Will print numbers 0 to 4
}
```

> Similarly, `break` works in `while` and `do-while` loops too.
>
> Another keyword is `return`, which will also exit a loop prematurely *and* exit the entire function containing the loop. `break` only stops the loop, while `return` stops the entire function."
>
> That would cover the key points an interviewer is looking for — you can also mention the `continue` keyword as a comparison (`continue` skips the current iteration but keeps the loop going) if you want to show even deeper knowledge.

Let me know if you'd also like me to give a quick code demo or tailor this answer for a particular interview style!

What is the difference between break and continue statements?

Great question — this is a common one in interviews when discussing loops and control flow. Here's a clear, concise way to answer:

In JavaScript (as well as many other languages):

✅ `break`

- Terminates the **entire loop** immediately.
- When a `break` is encountered inside a `for`, `while`, or `do-while` loop, the control jumps **out of the loop**, and execution continues after the loop block.

Example:

```
 for (let i = 0; i < 5; i++) {
  if (i === 3) {
    break; // stops the loop completely
  }
  console.log(i); // prints 0,1,2 then stops
}
```

-

✅ `continue`

- Skips the **current iteration** of the loop and moves to the **next iteration**.
- The loop itself is **not terminated**, only the remaining code in that iteration is skipped.

Example:

```
 for (let i = 0; i < 5; i++) {
  if (i === 3) {
    continue; // skip this iteration when i is 3
  }
  console.log(i); // prints 0,1,2,4 (skips 3)
}
```

-

💡 **In short**:

- `break` → Exit the loop entirely.
- `continue` → Skip to the next iteration.

This difference is often tested because it affects control flow and can optimize or change the behavior of loops in real-world code.

Let me know if you'd also like a flowchart or more examples for clarity!

How do you use template literals inside functions?

Great question — interviewers often want to check if you understand modern JavaScript features like **template literals** and can explain them in the context of reusable code.

Here's a clear and concise way to answer:

Answer:

Template literals allow you to embed expressions and variables directly into a string using backticks (`` ` ``). When you use them inside functions, they help you produce **dynamic strings** more cleanly and readably.

Here's a simple example:

```
function greetUser(name, age) {
  return `Hello, my name is ${name} and I am ${age} years old.`;
}

// Usage
console.log(greetUser('Alice', 25));
// Output: "Hello, my name is Alice and I am 25 years old."
```

✅ Key points to highlight in an interview:

- You use backtick (`` ` ``) syntax for template literals.
- You can insert variables or any valid JavaScript expression inside `${...}`.

- When you use template literals inside functions, it simplifies string concatenation and improves readability.
- Functions return these strings, making them **reusable** across your application.

More complex example with expressions inside a template literal:

```
function getDiscountedPrice(price, discountRate) {
  const discountedPrice = price - price * discountRate;
  return `Your discounted price is $${discountedPrice.toFixed(2)}.`;
}

console.log(getDiscountedPrice(100, 0.2));
// Output: "Your discounted price is $80.00."
```

That's the idea — template literals make your strings more flexible and easier to manage inside functions. 🎯

💡 Would you also like me to show an example using template literals with loops or conditional logic? Let me know — I'd be happy to add that as well!

Explain how recursion works with a simple example.

Great question — recursion is a fundamental programming concept that often comes up in interviews!

✅ What is Recursion?

Recursion is when a function **calls itself** directly or indirectly.
That means the problem is solved by breaking it down into **smaller subproblems of the same kind** until reaching a **base case** — which stops the recursion.

🔑 How Does Recursion Work?

Every recursive function needs two important parts:

1. 🎯 **Base case** — the condition where the recursion stops.

2. 🔄 **Recursive case** — where the function calls itself with a smaller or simpler input.

If the base case is missing, the recursion will go on forever, leading to a **“maximum call stack size exceeded” error**.

✏️ Example: Factorial Calculation

Factorial of a number `n` (`n!`) is the product of all positive integers up to `n`.
By definition:

```
n! = n * (n-1)!
1! = 1   (base case)
```

Here’s a simple example in JavaScript:

```
function factorial(n) {
  if (n <= 1) {          // Base case
    return 1;
  }
  return n * factorial(n - 1); // Recursive call
}

// Usage:
console.log(factorial(5)); // Output: 120
```

⚙️ What Happens Behind the Scenes?

When you call `factorial(5)`, the execution looks like this:

```
factorial(5)
  -> returns 5 * factorial(4)
          -> returns 4 * factorial(3)
                  -> returns 3 * factorial(2)
                          -> returns 2 * factorial(1)
                                  -> returns 1   (base case)
```

And then it “unwinds”:

```
2 * 1 = 2
3 * 2 = 6
4 * 6 = 24
5 * 24 = 120
```

🎯 Why Recursion?

✅ Makes the code **easier to read** for problems that have a natural recursive structure.
✅ Examples include tree traversal, factorials, Fibonacci numbers, and more.

❌ But recursion can use more memory due to the call stack, so be careful with very large inputs!

💡 **In summary:** Recursion is just a function calling itself until it hits a base case. The example of `factorial` is a classic one and is commonly asked in interviews.

Would you also like to see an example with other concepts like `arrays` or `strings`? Let me know — I'm here to help!

How do you pass arguments to a function and what are rest parameters?

Great question — this is a very common JavaScript topic in interviews! Here's a clear, structured way to answer:

💡 **Passing arguments to a function:**
In JavaScript, when you define a function with parameters, you pass arguments to that function when you invoke it. The arguments become the values of those parameters inside the function.

Example:

```
function greet(name, age) {
  console.log(`Hello ${name}, you are ${age} years old.`);
}

greet('Alice', 25); // 'Alice' and 25 are the arguments passed.
```

Here:

- `name` and `age` are the parameters.
- `'Alice'` and `25` are the arguments.

💡 Rest Parameters (`...args`):

Rest parameters allow a function to accept an indefinite number of arguments as an array. This is useful if you want a flexible number of arguments.

Example:

```
function sum(...numbers) {
  let total = 0;
  for (let num of numbers) {
    total += num;
  }
  return total;
}

console.log(sum(2, 5, 10)); // 17
```

Here:

- `...numbers` is the rest parameter — it gathers all arguments passed into the `sum` function into an array named `numbers`.
- Inside the function, you can treat `numbers` like a regular array.

✅ Key points to highlight in an interview:

- Normal parameters must come **before** the rest parameter.
- Rest parameters simplify handling variable numbers of arguments.
- Rest parameters produce a real array (unlike the old `arguments` object), so array methods can be used directly.