Javascript

Interview Questions

Q: What are the main features introduced in JavaScript ES6?

A: JavaScript ES6, also known as ECMAScript 2015, introduced several key features including arrow functions, classes, template literals, destructuring assignment, promises, `let` and `const` keywords, modules, and default parameters. These features enhance code readability, maintainability, and enable more advanced programming patterns.

Q: What is an arrow `function` , and how does it differ from regular functions?

A: An arrow `function` is a concise way to write `function` expressions in JavaScript. It uses the '=>' syntax. One of the main differences is that arrow functions do not have their own '`this`' context; they inherit '`this`' from the surrounding lexical scope, which can help avoid common pitfalls related to '`this`' in regular functions.

Q: Explain the concept of '`let`' and '`const`' in ES6.

A: '`let`' and '`const`' are block-scoped variable declarations introduced in ES6. '`let`' allows you to declare variables that can be reassigned, whereas '`const`' is used to declare variables that cannot be reassigned after their initial assignment. Both are limited to the block in which they are defined, preventing issues related to variable `hoisting` and scope.

Q: What are template literals and how do you use them?

A: Template literals are `string` literals that allow embedded expressions, which can be multi-line and include placeholders. They are enclosed by backticks (``) instead of single or double quotes. You can embed expressions using the ${expression} syntax. For example: `Hello, ${name}` allows for easier `string` interpolation and multi-line strings.

Q: What is destructuring assignment in JavaScript ES6?

A: Destructuring assignment is a syntax that allows unpacking values from arrays or properties from objects into distinct variables. For example, with an `array` : `const` arr = [1, 2]; `const` [a, b] = arr; here, 'a' will be 1 and 'b' will be 2. For objects: `const` obj = { x: 1, y: 2 }; `const` { x, y } = obj; `this` will assign 'x' and 'y' to their corresponding values.

Q: Explain how promises work in ES6 and their significance.

A: Promises are objects that represent the eventual completion (or failure) of an asynchronous operation and its resulting value. A `promise` can be in one of three states: pending, fulfilled, or rejected. They provide a cleaner alternative to `callback` functions, allowing for better error handling and chaining of asynchronous operations using '.then()' and '.catch()' methods.

Q: What are modules in ES6, and how do you `import` / `export` them?

A: Modules in ES6 allow developers to encapsulate code in separate files, making it easier to manage and reuse. You can `export` functions, objects, or primitives using the '`export`' keyword, and `import` them in another file using the '`import`' keyword. For example, '`export` `const` `myFunction` = () => {};' and '`import` { `myFunction` } from './`myModule`';'.

Q: What is the difference between '==' and '===' in JavaScript?

A: '==' is the equality operator that performs `type` coercion, meaning it converts operands to the same `type` before making the comparison. '===' is the strict equality operator that checks for equality without `type` conversion, meaning both the value and `type` must match. It is generally recommended to use '===' to avoid unexpected results.

Q: How does the 'spread operator' work in ES6?

A: The spread operator, represented by '...', allows an iterable (like an `array`) to be expanded in places where zero or more arguments or elements are expected. It can be used to copy arrays, concatenate them, or to pass elements as arguments to functions. For example: `const` arr1 = [1, 2]; `const` arr2 = [...arr1, 3, 4]; results in arr2 being [1, 2, 3, 4].

Q: What are default parameters in ES6?

A: Default parameters allow named parameters to be initialized with default values if no value or '`undefined`' is passed. `This` enhances `function` definitions by providing a way to set default behavior. For example: `function` multiply(a, b = 1) { `return` a * b; } If 'b' is not provided when calling the `function`, it will default to 1.

Q: What are the major features introduced in ES6?

A: ES6, also known as ECMAScript 2015, introduced several major features including: arrow functions, classes, template literals, destructuring, default parameters, rest and spread operators, `let` and `const` for variable declarations, promises for asynchronous programming, modules for better code organization, and the `Map` and Set data structures.

Q: What is an arrow `function` and how does it differ from a regular `function` ?

A: An arrow `function` is a shorter syntax for writing `function` expressions in JavaScript. It does not have its own '`this`', which means it inherits '`this`' from the parent scope, allowing for easier access to the context in which the `function` was defined. In contrast, regular functions have their own '`this`' context.

Q: Can you explain destructuring in ES6?

A: Destructuring is a syntax that allows unpacking values from arrays or properties from objects into distinct variables. For example, with arrays: `const` arr = [1, 2, 3]; `const` [a, b] = arr; // a is 1, b is 2. For objects: `const` obj = {x: 1, y: 2}; `const` {x, y} = obj; // x is 1, y is 2.

Q: What are template literals and how do they work?

A: Template literals are `string` literals that allow embedded expressions, multi-line strings, and interpolation. They are enclosed by backticks (`) instead of single or double quotes. For example: `const` name = 'World'; `const` greeting = `Hello, ${name}!`; // greeting is 'Hello, World!'.

Q: What is the purpose of the '`let`' and '`const`' keywords?

A: '`let`' and '`const`' are block-scoped variable declarations introduced in ES6. '`let`' allows you to declare variables that can be reassigned, `while` '`const`' declares variables that cannot be reassigned after their initial assignment. `This` helps prevent issues with variable `hoisting` and scope that were common with '`var`'.

Q: Explain the concept of promises in ES6.

A: Promises are objects that represent the eventual completion (or failure) of an asynchronous operation and its resulting value. A `promise` can be in one of three states: pending, fulfilled, or rejected. They provide a cleaner alternative to `callback` functions for handling asynchronous code, allowing for chaining with '.then()' and '.catch()' methods.

Q: What are modules in ES6, and how do they improve code organization?

A: Modules in ES6 allow developers to break their code into smaller, reusable pieces. Each module can `export` variables, functions, or classes, which can then be imported into other modules. `This` promotes encapsulation and reusability, making code easier to maintain and organize. Modules are defined using the '`export` ' and ' `import` ' keywords.

Q: What is the difference between the rest and spread operators?

A: The rest operator (denoted by '...') is used to collect multiple elements into a single `array` , typically used in `function` parameters. For example: `function` sum(...numbers) { `return` numbers.reduce((a, b) => a + b, 0); }. The spread operator, also denoted by '...', is used to spread elements of an iterable (like an `array`) into individual elements. For example: `const` arr1 = [1, 2]; `const` arr2 = [3, 4]; `const` combined = [...arr1, ...arr2]; // combined is [1, 2, 3, 4].

Q: How do you create a `class` in ES6?

A: In ES6, you can create a `class` using the '`class`' keyword. A `class` can include a `constructor` method and other methods. For example: `class` Person { `constructor` (name) { this.name = name; } greet () { console.log(`Hello, my name is ${this.name}`); } }. You can create an instance of the `class` using '`new` Person('Alice')'.

Q: What is a `Map` in ES6, and how does it differ from an `Object` ?

A: A `Map` is a collection of key-value pairs where keys can be of `any` `type` (including objects). Maps maintain the insertion order of their elements and have a size property to keep track of the `number` of entries. In contrast, an `Object` can only have `string` or symbol keys, and the order of its properties is not guaranteed. Maps also provide methods like 'set', 'get', and 'delete' for easier manipulation.

Q: What are the main features introduced in ES6?

A: ES6 introduced several key features, including: Arrow Functions, Classes, Template Literals, Destructuring Assignment, Default Parameters, Promises, Modules, and `let` / `const` for block-scoped variables. These features enhance the syntax and functionality of JavaScript, making it more powerful and easier to write.

Q: Can you explain the difference between '`let`', '`const`', and '`var`'?

A: '`var`' is function-scoped and can be redeclared; '`let`' is block-scoped and cannot be redeclared within the same block; '`const`' is also block-scoped but is used for variables that are meant to be constant, meaning they cannot be reassigned after their initial assignment. However, note that objects declared with '`const`' can still have their properties mutated.

Q: What are arrow functions and how do they differ from regular functions?

A: Arrow functions are a more concise syntax for writing `function` expressions. They do not have their own '`this`' context; instead, they inherit '`this`' from the surrounding lexical context. `This` makes them particularly useful in callbacks and methods where maintaining the context is important.

Q: What is destructuring assignment and how is it used?

A: Destructuring assignment allows unpacking values from arrays or properties from objects into distinct variables. For example, for an `array` : '`const` arr = [1, 2, 3]; `const` [a, b] = arr;' assigns 1 to 'a' and 2 to 'b'. For an `object` : '`const` obj = { x: 1, y: 2 }; `const` { x, y } = obj;' assigns 1 to 'x' and 2 to 'y'. `This` improves code readability and conciseness.

25%

You're 25% through! Keep going! Success is built one step at a time.

Q: What are template literals and how are they used?

A: Template literals are `string` literals that allow embedded expressions, which are enclosed by backticks (`). They can contain placeholders indicated by the dollar sign and curly braces (${expression}). For example: ' `const` name = "John"; `const` greeting = `Hello, ${name}!`;'. `This` feature allows for multi-line strings and easier `string` interpolation.

Q: Explain Promises in JavaScript ES6.

A: Promises are objects that represent the eventual completion (or failure) of an asynchronous operation and its resulting value. A `Promise` can be in one of three states: pending, fulfilled, or rejected. They provide a cleaner alternative to `callback` functions for handling asynchronous operations, using '.then()' for success and '.catch()' for errors.

Q: What are modules in ES6, and how do you `export` and `import` them?

A: Modules in ES6 allow the separation of code into reusable pieces, improving code organization. You can `export` a module using '`export`' keyword and `import` it using '`import`'. For example, to `export`: '`export` `const` `myFunction` = () => {};'. To `import`: '`import` { `myFunction` } from './myModule.js';'. `This` promotes better dependency management and encapsulation.

Q: What is the spread operator and how is it different from the rest operator?

A: The spread operator ('...') allows iterable elements like arrays or objects to be expanded in places where multiple elements or key-value pairs are expected. For instance, combining arrays: '`const` arr1 = [1, 2]; `const` arr2 = [3, 4]; `const` `newArr` = [...arr1, ...arr2];'. The rest operator does the opposite, collecting multiple elements into a single `array` or `object` , used in `function` parameters like '`function` `myFunc` (...args) {}'.

Q: What are default parameters in ES6?

A: Default parameters allow named parameters to be initialized with default values if no value or '`undefined`' is passed. For example: '`function` multiply(a, b = 1) { `return` a * b; }'. If 'multiply(5)' is called, 'b' will default to 1, resulting in 5. `This` feature simplifies `function` definitions and improves code clarity.

Q: How does the '`this`' keyword behave in ES6 arrow functions?

A: In ES6 arrow functions, '`this`' retains the value of '`this`' from the enclosing lexical context, unlike regular functions which have their own '`this`' value based on how they are called. `This` is particularly useful in methods or callbacks where you want to retain the context of the surrounding scope.

Q: What are the major features introduced in ES6?

A: ES6, also known as ECMAScript 2015, introduced several major features including: '`let`' and '`const`' for variable declaration, arrow functions for concise `function` expressions, template literals for `string` interpolation, default parameters in functions, destructuring assignment for extracting values from arrays or objects, promises for better asynchronous programming, classes for object-oriented programming, and modules for better code organization.

Q: What is the difference between '`let`' and '`var`'?

A: '`let`' and '`var`' are both used to declare variables in JavaScript, but there are key differences. '`var`' is function-scoped, meaning it is accessible within the `function` it was declared in, `while` '`let`' is block-scoped, which means it is only accessible within the nearest enclosing block (e.g., within a loop or an if statement). Additionally, variables declared with '`var`' can be re-declared and updated, `while` '`let`' allows updates but not re-declarations in the same block.

Q: What are arrow functions and how do they differ from regular functions?

A: Arrow functions provide a more concise syntax for writing `function` expressions. They use the '=>' syntax and do not have their own '`this`' context, meaning they inherit '`this`' from the enclosing scope. `This` makes them particularly useful for callbacks where you want to maintain the context of '`this`'. In contrast, regular functions have their own '`this`' context, which can lead to confusion if not bound properly.

Q: Explain destructuring assignment in ES6.

A: Destructuring assignment is a syntax that allows unpacking values from arrays or properties from objects into distinct variables. For example, you can extract values from an `array` like `this` : '`const` arr = [1, 2, 3]; `const` [a, b] = arr;' which assigns 1 to 'a' and 2 to 'b'. For objects, you can do: '`const` obj = { x: 1, y: 2 }; `const` { x, y } = obj;' which assigns 1 to 'x' and 2 to 'y'. `This` feature simplifies the extraction of values and improves code readability.

Q: What are template literals and how are they used?

A: Template literals are `string` literals that allow embedded expressions, making it easier to create complex strings. They are enclosed by backticks (``) instead of single or double quotes. You can include variables and expressions inside template literals using the '${expression}' syntax. For example: '`const` name = 'John'; `const` greeting = `Hello, ${name}!`;'. `This` will result in 'Hello, John!'. Template literals also support multi-line strings without the need for escape characters.

Q: What is the purpose of promises in ES6?

A: Promises are objects that represent the eventual completion (or failure) of an asynchronous operation and its resulting value. They provide a cleaner way to handle asynchronous code compared to traditional `callback` functions, helping to avoid '`callback` hell'. A `promise` can be in one of three states: pending, fulfilled, or rejected. You can use '.then()' to handle fulfilled promises and '.catch()' for rejected promises, allowing for better error handling and chaining of asynchronous operations.

Q: What is the difference between '`const`' and '`let`'?

A: '`const`' and '`let`' are both used for block-scoped variable declarations, but '`const`' is used for declaring variables that should not be reassigned. Once a variable is declared with '`const`', its reference cannot be changed, though mutable objects can still be modified. For example: '`const` `myArray` = [1, 2]; myArray.push(3);' is valid, but '`myArray` = [4, 5];' would throw an error. '`let`' allows you to reassign the variable.

Q: How do modules work in ES6?

A: ES6 introduced a module system that allows for better organization of code. Modules can `export` variables, functions, classes, or objects using the '`export` ' keyword. Other modules can then `import` these exports using the '`import` ' keyword. For example, to `export` a `function` : '`export` `function` `myFunction` () {}', and to `import` it: '`import` { `myFunction` } from './myModule.js';'. `This` encourages encapsulation and reusability of code.

Q: Explain the concept of `class` syntax in ES6.

A: ES6 introduced a `class` syntax to simplify the creation of objects and inheritance. Classes are defined using the '`class`' keyword followed by the `class` name and a `constructor` method for initializing objects. For example: '`class` Animal { `constructor` (name) { this.name = name; } }'. You can also define methods within the `class` and use the '`extends`' keyword for inheritance. `This` syntax provides a clearer and more structured way to implement object-oriented programming in JavaScript.

Q: What are default parameters in ES6 functions?

A: Default parameters allow you to set default values for `function` parameters if no value or '`undefined`' is passed. `This` feature helps to simplify `function` calls. For example: '`function` multiply(a, b = 1) { `return` a * b; }' means that if 'b' is not provided, it defaults to 1. `This` eliminates the need to check if a parameter has been provided and makes the `function` more robust.

Q: What are the main features introduced in JavaScript ES6?

A: JavaScript ES6 introduced several important features including: arrow functions, classes, template literals, destructuring assignments, default parameters, rest and spread operators, modules, promises, and `let` / `const` declarations.

Q: Can you explain what arrow functions are and how they differ from traditional functions?

A: Arrow functions are a more concise way to write `function` expressions. They do not have their own '`this`' context, which means they inherit '`this`' from the parent scope. `This` is different from traditional functions, which define their own '`this`' context based on how they are called.

Q: What is the purpose of '`let`' and '`const`' in ES6?

A: '`let`' and '`const`' are used to declare variables. '`let`' allows you to declare block-scoped variables, which means they are limited to the block, statement, or expression where they are defined. '`const`' is used to declare block-scoped constants that cannot be reassigned once defined.

Q: How do template literals work in ES6?

A: Template literals are enclosed by backticks (``) and allow for multi-line strings and `string` interpolation. You can embed expressions within the `string` using the syntax ${expression}, which makes it easier to construct strings dynamically.

Q: What is destructuring assignment and how is it used?

A: Destructuring assignment is a syntax that allows unpacking values from arrays or properties from objects into distinct variables. For example, for an `array` : `const` arr = [1, 2]; `const` [a, b] = arr; a would be 1 and b would be 2. For an `object` : `const` obj = {x: 1, y: 2}; `const` {x, y} = obj; x would be 1 and y would be 2.

Q: What are the rest and spread operators in ES6?

A: The rest operator (...rest) allows you to represent an indefinite `number` of arguments as an array. It is often used in `function` parameters. The spread operator (...spread) allows you to expand an iterable (like an `array`) into its elements. It can be used in `function` calls or to create `new` arrays/objects.

Q: Explain how modules work in ES6.

A: ES6 introduced a module system that allows you to `export` and `import` code between different files. You can `export` variables, functions, or classes using the '`export`' keyword and `import` them in another file using the '`import`' keyword. `This` helps in organizing code and managing dependencies.

Q: What are Promises in JavaScript and how have they improved asynchronous programming?

A: Promises are objects that represent the eventual completion (or failure) of an asynchronous operation and its resulting value. They provide a simpler alternative to callbacks, allowing you to chain operations and handle errors more gracefully using '.then()' for success and '.catch()' for error handling.

Q: How does the '`class` ' syntax in ES6 improve object-oriented programming in JavaScript?

A: The '`class` ' syntax in ES6 provides a clearer and more concise way to create objects and handle inheritance. It allows you to define constructors, methods, and static methods more intuitively, making the code more readable and resembling traditional OOP languages.

50%

Halfway there! Every expert was once a beginner.

Q: What is the difference between '== ' and '===' in JavaScript?

A: '==' is the equality operator that performs `type` coercion, meaning it converts the operands to the same `type` before comparing them. '===' is the strict equality operator that checks both the value and `type` without coercion. It's generally recommended to use '===' to avoid unexpected results.

Q: What are the main features introduced in ES6?

A: ES6, also known as ECMAScript 2015, introduced several key features including `let` and `const` for variable declarations, arrow functions, template literals, destructuring assignment, default parameters, rest/spread operators, promises, and classes. These features enhance code readability, maintainability, and efficiency.

Q: What is the difference between '`let`' and '`var`'?

A: '`let`' is block-scoped, meaning it is only accessible within the block it is defined in, `while` '`var`' is function-scoped or globally scoped, which can lead to unexpected behaviors. Additionally, '`let`' cannot be redeclared in the same scope, whereas '`var`' can be redeclared.

Q: Explain arrow functions and their advantages.

A: Arrow functions are a syntactically compact way to write `function` expressions in ES6. They do not have their own '`this`', which means they inherit '`this`' from the parent scope. `This` makes them particularly useful for methods that need to maintain the context of '`this`', such as callbacks in `array` methods.

Q: What are template literals and how do you use them?

A: Template literals are `string` literals that allow embedded expressions, denoted by backticks (``). They can include placeholders indicated by the dollar sign and curly braces (${expression}). `This` allows for multi-line strings and easier `string` interpolation, improving code readability.

Q: What is destructuring assignment in ES6?

A: Destructuring assignment is a syntax that allows unpacking values from arrays or properties from objects into distinct variables. For example, for an `array` , you can do '`const` [a, b] = [1, 2];' and for an `object` , '`const` {x, y} = {x: 1, y: 2};' which makes it easier to extract data from complex structures.

Q: Can you explain the concept of 'Promises' in JavaScript?

A: Promises are a native JavaScript feature in ES6 that represent the eventual completion (or failure) of an asynchronous operation and its resulting value. A `promise` can be in one of three states: pending, fulfilled, or rejected. They allow for cleaner handling of asynchronous code compared to traditional `callback` methods.

Q: What are default parameters in ES6?

A: Default parameters allow named parameters to be initialized with default values if no value or `undefined` is passed. `This` feature simplifies `function` definitions and improves code readability. For example, ' `function` multiply(a, b = 1) { `return` a * b; }' means 'b' defaults to 1 if not provided.

Q: What are the rest and spread operators in ES6?

A: The rest operator, represented by '...', allows us to represent an indefinite `number` of arguments as an array. For example, '`function` sum(...numbers) { `return` numbers.reduce((acc, num) => acc + num, 0); }' collects all arguments into an array. The spread operator, also '...', allows an iterable such as an `array` to be expanded in places where zero or more arguments or elements are expected, like '`const` arr = [1, 2, 3]; `const` `newArr` = [...arr, 4, 5];'.

Q: What is the purpose of classes in ES6?

A: Classes in ES6 provide a clearer and more concise syntax for creating objects and handling inheritance compared to traditional prototype-based inheritance. They encapsulate data and behaviors together, making it easier to structure code. A `class` can include a `constructor` , methods, and static methods, enhancing the object-oriented programming capabilities in JavaScript.

Q: How does the '`this`' keyword behave in arrow functions?

A: In arrow functions, '`this`' is lexically bound, meaning it retains the '`this`' value from the enclosing execution context. `This` contrasts with regular functions, where '`this`' can change depending on how the `function` is called. `This` feature makes arrow functions particularly useful for callbacks where maintaining the context is important.

Q: What are the major features introduced in ES6?

A: ES6 introduced several major features including arrow functions, classes, template literals, destructuring, default parameters, `let` and `const` declarations, promises, and modules. These features enhance the syntax and capabilities of JavaScript, making it more powerful and easier to write.

Q: What is an arrow `function` and how does it differ from a regular `function` ?

A: An arrow `function` is a shorter syntax for writing `function` expressions. It does not bind its own '`this`', meaning it inherits '`this`' from the parent scope. `This` is different from regular functions, which have their own '`this`' context. For example, instead of writing '`function` () { }', you can write '() => { }'.

Q: Can you explain the concept of template literals?

A: Template literals are enclosed by backticks (``) instead of single or double quotes. They allow for multi-line strings and expression interpolation, which means you can embed expressions inside `string` literals using the ${expression} syntax. For example: `Hello, ${name}!` creates a `string` that includes the value of the variable 'name'.

Q: What is destructuring and how is it used?

A: Destructuring is a convenient way of extracting values from arrays or properties from objects into distinct variables. For example, for an `array` : '`const` arr = [1, 2, 3]; `const` [a, b] = arr;' will assign '1' to 'a' and '2' to 'b'. For objects: '`const` obj = {x: 1, y: 2}; `const` {x, y} = obj;' will assign '1' to 'x' and '2' to 'y'.

Q: What are `let` and `const`, and how do they differ from `var`?

A: Both '`let`' and '`const`' are block-scoped variable declarations introduced in ES6. '`let`' allows you to declare variables that can be reassigned, `while` '`const`' is used for variables that cannot be reassigned after their initial assignment. In contrast, '`var`' is function-scoped and can lead to issues with hoisting.

Q: What is a `promise` in JavaScript?

A: A `promise` is an `object` that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. Promises have three states: pending, fulfilled, or rejected. They allow for cleaner asynchronous code management using '.then()' for success and '.catch()' for error handling, improving readability compared to `callback` functions.

Q: How do modules work in ES6?

A: ES6 introduces a module system that allows you to `export` and `import` modules. You can use '`export`' to expose variables or functions from a module, and '`import`' to include those exports in another module. `This` helps in organizing code better and avoiding global scope pollution. For example, '`export` `const` `myVar` = 1;' and then '`import` { `myVar` } from './myModule.js';'.

Q: What is the spread operator and how is it used in ES6?

A: The spread operator, represented by '...', allows an iterable (like an `array`) to be expanded in places where zero or more arguments or elements are expected. It can be used to copy arrays, concatenate arrays, or spread elements in `function` calls. For example, '`const` arr1 = [1, 2]; `const` arr2 = [...arr1, 3, 4];' results in 'arr2' being [1, 2, 3, 4].

Q: What are default parameters in ES6?

A: Default parameters allow named parameters to be initialized with default values if no value or '`undefined`' is passed. For example, '`function` multiply(a, b = 1) { `return` a * b; }' will use '1' as the default value for 'b' if it's not provided, making `function` calls more flexible.

Q: Can you explain how `async` /`await` works in ES6?

A: `Async` /`await` is syntactic sugar built on top of promises that allows you to write asynchronous code that looks synchronous. An '`async` ' `function` always returns a promise. Inside an `async` `function` , you can use '`await` ' to pause execution until the `promise` is resolved or rejected. `This` makes handling asynchronous operations easier and more readable compared to using .then() chains.

Q: What are the main features introduced in ES6?

A: ES6, also known as ECMAScript 2015, introduced several key features, including arrow functions, classes, template literals, destructuring assignment, default parameters, rest and spread operators, promises, and `let` / `const` declarations. These features improve the readability and functionality of JavaScript code.

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

A: The '`var` ' keyword declares a variable that is function-scoped or globally-scoped. '`let` ' declares a block-scoped variable, meaning it is only accessible within the block it is defined. '`const` ' also declares a block-scoped variable, but the variable cannot be reassigned after its initial assignment, making it ideal for constants.

Q: Explain arrow functions and how they differ from regular functions.

A: Arrow functions provide a shorter syntax for writing `function` expressions. They do not have their own '`this`' context, which means they inherit '`this`' from the surrounding lexical context. `This` is different from regular functions, where '`this`' is defined based on how the `function` is called. For example: '`const` add = (a, b) => a + b;' versus '`function` add(a, b) { `return` a + b; }'.

Q: What are template literals and how are they used?

A: Template literals are enclosed by backticks (``) and allow for multi-line strings and `string` interpolation. You can embed expressions within a template literal using the syntax ${expression}. For example: '`const` name = ' John'; `const` greeting = `Hello, ${name}!`;'. `This` allows for easier `string` concatenation and formatting.

75%

You're at 75%! Almost done, push through and finish strong!

Q: What is destructuring assignment and provide an example?

A: Destructuring assignment is a syntax that allows unpacking values from arrays or properties from objects into distinct variables. For example, for an `array` : '`const` arr = [1, 2, 3]; `const` [a, b] = arr;' assigns 1 to 'a' and 2 to 'b'. For an `object` : '`const` obj = { x: 1, y: 2 }; `const` { x, y } = obj;' assigns 1 to 'x' and 2 to 'y' .

Q: What are promises and how do they work?

A: Promises are objects that represent the eventual completion (or failure) of an asynchronous operation and its resulting value. A `promise` can be in one of three states: pending, fulfilled, or rejected. You can create a `promise` using '`new` `Promise` ((resolve, reject) => {...})' and handle it using '.then()' for fulfillment and '.catch()' for rejection.

Q: What are the rest and spread operators in ES6?

A: The spread operator, denoted by '...', allows an iterable (like an `array`) to be expanded into elements. For example, '`const` `array` = [1, 2]; `const` `newArray` = [...array, 3];' results in '[1, 2, 3]'. The rest operator also uses '...' and allows us to collect multiple elements into an array. For example, '`function` sum(...args) { `return` args.reduce((acc, curr) => acc + curr, 0); }' collects all arguments into an `array` 'args'.

Q: How do you create a `class` in ES6 and what are its advantages?

A: In ES6, you can create a `class` using the '`class`' keyword. For example: '`class` Person { `constructor` (name) { this.name = name; } greet() { console.log(`Hello, ${this.name}`); } }'. Classes provide a clear and concise way to create objects and handle inheritance, making the code more organized and easier to maintain compared to traditional `constructor` functions.

Q: What is the purpose of 'default parameters' in ES6?

A: Default parameters allow named parameters to be initialized with default values if no value or '`undefined`' is passed. `This` simplifies `function` calls and reduces the need for manual checks inside the function. For example: '`function` multiply(a, b = 1) { `return` a * b; }'. If 'multiply(5)' is called, 'b' defaults to 1.

Q: What is the '`import`' and '`export`' syntax in ES6?

A: The '`import`' and '`export`' syntax allows for modular programming in JavaScript. '`export`' is used to `export` functions, objects, or primitives from a module, `while` '`import`' is used to bring them into another module. For example: '`export` `const` `myFunction` = () => {...};' and in another file, '`import` { `myFunction` } from './`myModule`';'. `This` promotes better organization and reusability of code.

Q: What are the main features introduced in ES6?

A: ES6 introduced several significant features including `let` and `const` for variable declarations, arrow functions, template literals, destructuring assignments, default parameters, rest and spread operators, classes, modules, and promises.

Q: Can you explain the difference between `var` , `let` , and `const` ?

A: The '`var` ' keyword declares a variable that is function-scoped or globally-scoped, which can lead to `hoisting` issues. '`let` ' and '`const` ', introduced in ES6, are block-scoped. '`let` ' allows you to reassign values, whereas '`const` ' creates a read-only reference, meaning the variable cannot be reassigned after its initial assignment.

Q: What are arrow functions and how do they differ from traditional functions?

A: Arrow functions provide a more concise syntax for writing functions. They do not have their own '`this`' context; instead, they inherit '`this`' from the parent scope, which is useful for avoiding common issues with '`this`' in callbacks. They also cannot be used as constructors and do not have access to the 'arguments' object.

Q: What is destructuring assignment in ES6?

A: Destructuring assignment is a syntax that allows unpacking values from arrays or properties from objects into distinct variables. For example, for an `array` : '`const` [a, b] = [1, 2];' assigns 'a' to 1 and 'b' to 2. For objects: '`const` {x, y} = {x: 1, y: 2};' assigns 'x' to 1 and 'y' to 2.

Q: What are template literals and how do you use them?

A: Template literals are `string` literals that allow embedded expressions, which can be multi-line and are enclosed by backticks (``). You can use placeholders indicated by the dollar sign and curly braces: '${expression}'. For example, '`const` greeting = `Hello, ${name}!`;'.

Q: Explain the concept of modules in ES6.

A: Modules in ES6 allow for splitting code into separate files, which helps in organizing and maintaining code. You can `export` functions, objects, or primitives from a module using '`export`' and `import` them in another module using '`import`'. `This` promotes encapsulation and reusability.

Q: What are promises and how do they work?

A: Promises are objects that represent the eventual completion (or failure) of an asynchronous operation and its resulting value. A `promise` can be in one of three states: pending, fulfilled, or rejected. You can use '.then()' to handle a fulfilled `promise` and '.catch()' to handle a rejected one.

Q: What are the rest and spread operators?

A: The rest operator (three dots '...') is used to collect all remaining elements in an `array` or `object` into a single variable, `while` the spread operator (also three dots '...') is used to expand an `array` or `object` into individual elements. For example, '`const` sum = (...args) => args.reduce((a, b) => a + b, 0);' uses the rest operator, and '[...array1, ...array2]' uses the spread operator.

Q: How does the '`class`' syntax in ES6 differ from `constructor` functions?

A: ES6 introduced a more clear and concise way to create classes using the '`class`' keyword. Unlike `constructor` functions, classes provide a clear structure for inheritance and use the '`extends`' keyword for subclassing. Methods defined in a `class` do not need the '`function`' keyword and are automatically added to the prototype.

Q: What are default parameters in ES6?

A: Default parameters allow you to initialize `function` parameters with default values if no value or '`undefined`' is passed. For example: '`function` multiply(a, b = 1) { `return` a * b; }' means if 'b' is not provided, it defaults to 1.

Q: What are the main features introduced in JavaScript ES6?

A: JavaScript ES6, also known as ECMAScript 2015, introduced several significant features including: `let` and `const` declarations for block-scoped variables, arrow functions for shorter `function` syntax, template literals for `string` interpolation, default parameters in functions, destructuring assignment for unpacking values from arrays or objects, modules for better code organization, promises for handling asynchronous operations, and the spread/rest operators for arrays and `function` parameters.

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

A: The '`var` ' keyword declares a variable that is function-scoped or globally-scoped, and it can be redeclared and updated. '`let` ' is block-scoped, meaning it is only accessible within the block it is defined in, and it cannot be redeclared in the same scope but can be updated. '`const` ' is also block-scoped and must be initialized at the time of declaration; it cannot be updated or redeclared, making it useful for constants.

Q: Can you explain arrow functions and their advantages?

A: Arrow functions are a more concise way to write `function` expressions in JavaScript. They are defined using the '=> ' syntax. One of their main advantages is that they do not have their own '`this` ' context; instead, they inherit '`this` ' from the surrounding lexical scope, which makes them particularly useful for methods that need to access the '`this` ' context from the enclosing scope. `This` reduces the need for using 'bind()' or storing '`this` ' in a variable.

Q: What are template literals and how do you use them?

A: Template literals are `string` literals that allow embedded expressions, which can be multi-line and include placeholders indicated by the dollar sign and curly braces (${expression}). They are enclosed by backticks (``) instead of single or double quotes. For example, `Hello, ${name}!` can dynamically include the variable 'name' in the string.

Q: What is destructuring assignment in ES6?

A: Destructuring assignment is a syntax that allows unpacking values from arrays or properties from objects into distinct variables. For example, for arrays: '`const` arr = [1, 2, 3]; `const` [a, b] = arr;' assigns 1 to 'a' and 2 to 'b'. For objects: '`const` obj = { x: 1, y: 2 }; `const` { x, y } = obj;' assigns 1 to 'x' and 2 to 'y'. `This` feature makes code cleaner and more readable.

Q: What are promises and how do they work?

A: Promises are a way to handle asynchronous operations in JavaScript. A `promise` represents a value that may be available now, or in the future, or never. It has three states: pending, fulfilled, and rejected. When a `promise` is fulfilled, it can be resolved with a value, and when rejected, it can be resolved with an error. Promises can be chained using '.then()' for handling success and '.catch()' for handling errors.

Q: What is the spread operator and how is it used?

A: The spread operator, represented by '...', allows an iterable (such as an `array`) to be expanded in places where zero or more arguments are expected. It can be used to merge arrays, to create shallow copies of arrays/ objects, or to pass elements of an `array` as arguments to a function. For example, '`const` arr1 = [1, 2]; `const` arr2 = [3, 4]; `const` combined = [...arr1, ...arr2];' results in 'combined' being [1, 2, 3, 4].

Q: What are modules in ES6, and how do they differ from previous implementations?

A: Modules in ES6 allow for better organization of code by encapsulating functionality in separate files. They use '`import`' and '`export`' keywords to share variables and functions between files. Unlike previous implementations (like IIFE or CommonJS), ES6 modules are statically analyzed, which improves `optimization` and loading times. An example of exporting a `function`: '`export` `function` `myFunc` () {}' and importing it: '`import` { `myFunc` } from './myModule.js';'.

Q: What is the rest parameter syntax and how is it different from the arguments `object` ?

A: The rest parameter syntax, represented by '...', allows a `function` to accept an indefinite `number` of arguments as an array. It is defined in the `function` parameters, for example: '`function` sum(...numbers) { `return` numbers.reduce((a, b) => a + b, 0); }'. `This` differs from the 'arguments' `object` , which is an array-like `object` available in all functions, but it does not work with arrow functions and does not have `array` methods.

Q: What are the key features introduced in JavaScript ES6?

A: JavaScript ES6, also known as ECMAScript 2015, introduced several significant features that enhanced the language's capabilities. Some of the key features include: 1. `Let` and `Const` : Unlike `var` , `let` allows block-scoped variable declarations, and `const` provides a way to define constants that cannot be reassigned. 2. Arrow Functions: A more concise syntax for writing `function` expressions, which also lexically binds the ' `this` ' value. 3. Template Literals: A `new` syntax for strings that allows for multi-line strings and embedded expressions using backticks (``) instead of quotes. 4. Destructuring Assignment: A syntax that allows unpacking values from arrays or properties from objects into distinct variables. 5. Default Parameters: `Function` parameters can have default values if no value or `undefined` is passed. 6. Promises: A native way to handle asynchronous operations, allowing for cleaner and more manageable code compared to traditional `callback` methods. 7. Modules: The introduction of `import` and `export` statements for modular programming, facilitating better organization of code. 8. Classes: A syntactical sugar over JavaScript's existing prototype-based inheritance, providing a clearer and more elegant way to create objects and handle inheritance.

Thank You!

You've completed all the questions.

""Learning never exhausts the mind." — Leonardo da Vinci"