INTRODUCTION TO JAVASCRIPT

1. What is JavaScript?

- **Answer:** JavaScript is a high-level, interpreted programming language primarily used to create interactive effects within web browsers. It is used for front-end development, making websites dynamic and responsive. JavaScript can also be used on the server-side using frameworks like Node.js.

2. How do you declare a variable in JavaScript?

- **Answer:** In JavaScript, you can declare a variable using `var`, `let`, or `const`.
  - `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 constants (values that do not change).

javascript Copy code

```javascript
var x = 10;
let y = 20;
const z = 30;
```

3. What are JavaScript data types?

- **Answer:** JavaScript has several data types which can be categorized as:
  - **Primitive Data Types:**
    1. **Number:** e.g., `42`, `3.14`
    2. **String:** e.g., `"Hello"`, `'World'`
    3. **Boolean:** `true`, `false`
    4. **Undefined:** A variable declared but not assigned a value
    5. **Null:** Represents an intentional absence of value
    6. **Symbol:** A unique and immutable data type introduced in ES6
    7. **BigInt:** For very large integers beyond `Number`'s limit (introduced in ES11)
  - **Reference Data Type:**
    Object: e.g., Arrays, Functions, and Objects.

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

- **Answer:**
  - `==` (Double equals) is used for *loose equality*. It performs type coercion, meaning it converts the operands to the same type before comparing them.
  - `===` (Triple equals) is used for *strict equality*. It checks both the value and the type of the operands without converting them.

javascript Copy code

```javascript
console.log(5 == "5");  // true (type coercion occurs)
console.log(5 === "5"); // false (different types)
```

5. What is an array in JavaScript, and how do you create one?

- **Answer:** An array is a special type of object in JavaScript that allows you to store multiple values in a single variable. Arrays are used to store lists of items (e.g., numbers, strings, or other variables).

You can create an array using square brackets `[]` or the `Array` constructor.

```javascript
let numbers = [1, 2, 3, 4, 5]; // Using square brackets
let colors = new Array("red", "green", "blue"); // Using Array constructor
```

SECTION # 01

VAR

1. How is `var` used to declare a variable?

- **Answer:** You can use the `var` keyword to declare a variable. The variable can hold different types of data like numbers, strings, arrays, etc.

```javascript
var name = "John";  // Declares a variable called name
var age = 25;       // Declares a variable called age
var isStudent = true; // Declares a variable with a boolean value
```

2. Can you redeclare a `var` variable?

- **Answer:** Yes, with `var`, you can declare the same variable multiple times without any errors.

```javascript
var city = "New York";
var city = "London"; // No error, city is redeclared
console.log(city);   // Output: "London"
```

3. Does `var` have block scope?

- **Answer:** No, `var` does not have block scope. It is either **global** or **function-scoped**. Even if declared inside a block like an `if` or `for` loop, `var` is accessible outside of that block.

```javascript
if (true) {
    var message = "Hello";
}
console.log(message); // Output: "Hello"
```

In this case, `var message` is available outside the `if` block because `var` ignores block scope.

4. What happens when you use `var` without initialization?

- **Answer:** You can declare a variable with `var` without assigning it an initial value. In this case, its value will be `undefined`.

```javascript
var test; // Declared but not initialized
console.log(test); // Output: undefined
```

5. Can a `var` variable be updated?

- **Answer:** Yes, variables declared with `var` can be updated after their initial assignment.

```javascript
var count = 10;
count = 20;  // Updating the value of count
console.log(count); // Output: 20
```

1. What happens when you declare a `var` variable globally?

- **Answer:** When you declare a variable using `var` outside of any function, it becomes a global variable. This means it is attached to the `window` object in the browser and can be accessed from anywhere in the code.

```javascript
var globalVar = "I'm global!";
console.log(window.globalVar); // Output: "I'm global!"
```

6

SECTION # 02

LET

1. What is `let` in JavaScript?

- **Answer:** `let` is a keyword used to declare variables in JavaScript. Unlike `var`, `let` is **block-scoped**, meaning the variable is only available within the block in which it is defined (inside `{}`).

```javascript
let x = 10;
console.log(x); // Output: 10
```

2. What is block scope, and how does `let` behave in block scope?

- **Answer:** Block scope means that variables declared with `let` are only accessible within the block (between `{}`) where they are defined. This includes structures like `if`, `for`, and `while`.

```javascript
if (true) {
    let y = 20;
    console.log(y); // Output: 20
}
console.log(y); // Error! y is not defined outside the block
```

3. Can a `let` variable be redeclared?

- **Answer:** No, unlike `var`, variables declared with `let` cannot be redeclared within the same scope. If you try to redeclare a `let` variable, it will throw an error.

```javascript
let name = "Alice";
let name = "Bob"; // Error! Cannot redeclare 'name'
```

4. Can a `let` variable be updated?

- **Answer:** Yes, variables declared with `let` can be updated (reassigned), but they cannot be redeclared in the same scope.

```javascript
let count = 5;
count = 10; // Updating the value
console.log(count); // Output: 10
```

5. Is `let` hoisted like `var`?

- **Answer:** Yes, `let` is hoisted, but unlike `var`, it is not initialized with `undefined`. Accessing a `let` variable before its declaration results in a **ReferenceError**. This is because `let` is in a **temporal dead zone** from the start of the block until the declaration is encountered.

```javascript
console.log(a); // ReferenceError! Cannot access 'a' before initialization
let a = 5;
```

6. What happens when you declare a `let` variable in a loop?

- **Answer:** Variables declared with `let` in a loop (like `for` or `while`) are scoped to each iteration of the loop. This makes `let` very useful for loop counters or block-scoped variables.

```javascript
for (let i = 0; i < 3; i++) {
    console.log(i); // Output: 0, 1, 2
}
console.log(i); // Error! i is not defined outside the loop
```

7. What are the advantages of using `let` over `var`?

- Answer:
  1. **Block Scope**: `let` is block-scoped, which prevents accidental access to variables outside the intended block.
  2. **No Redeclaration**: `let` helps avoid bugs by not allowing redeclaration in the same scope.
  3. **Temporal Dead Zone**: `let` helps avoid bugs by ensuring variables are initialized before they are accessed.

8 Can `let` variables be declared without initialization?

- **Answer:** Yes, you can declare a `let` variable without initializing it, and it will be `undefined` until you assign a value to it later.

javascript Copy code

```javascript
let num;
console.log(num); // Output: undefined
num = 10;
console.log(num); // Output: 10
```

9 What happens if you redeclare a `let` variable in a different block?

- **Answer:** You can redeclare a `let` variable in a different block because `let` is block-scoped. Each block has its own scope.

javascript Copy code

```javascript
let color = "red";
{
    let color = "blue"; // Allowed, because it's a different block
    console.log(color); // Output: blue
}
console.log(color); // Output: red
```

10 What happens if you redeclare a `let` variable in a different block?

- **Answer:** You can redeclare a `let` variable in a different block because `let` is block-scoped. Each block has its own scope.

javascript Copy code

```javascript
let color = "red";
{
    let color = "blue"; // Allowed, because it's a different block
    console.log(color); // Output: blue
}
console.log(color); // Output: red
```

SECTION # 03

CONSTANT

1. What is `const` in JavaScript?

- **Answer:** The `const` keyword is used to declare **constants** in JavaScript. Once a variable is declared with `const`, its value **cannot be reassigned.** It is also block-scoped, similar to `let`.

```javascript
const pi = 3.14;
console.log(pi); // Output: 3.14
```

2. Can a `const` variable be updated or reassigned?

- **Answer:** No, a variable declared with `const` **cannot be reassigned** after it has been initialized. However, if the constant holds an object or an array, you can still modify the contents of the object or array, but you cannot reassign the variable to a new object or array.

```javascript
const number = 10;
number = 20; // Error! Assignment to constant variable is not allowed

const person = { name: "Alice" };
person.name = "Bob"; // Allowed, you can change the properties of the object
console.log(person.name); // Output: Bob
```

3. Is `const` block-scoped?

- **Answer:** Yes, `const`, like `let`, is block-scoped. This means that a `const` variable declared inside a block (like an `if` or `for` loop) is only accessible within that block.

```javascript
if (true) {
    const city = "New York";
    console.log(city); // Output: New York
}
console.log(city); // Error! city is not defined outside the block
```

4. Can you declare a `const` variable without initialization?

- **Answer:** No, you **must initialize** a `const` variable at the time of declaration. If you try to declare it without assigning a value, you will get an error.

```javascript
const age; // Error! Missing initializer in const declaration
```

5. Can a `const` variable be hoisted?

- **Answer:** Like `let`, `const` is hoisted, but it is not initialized. If you try to use a `const` variable before it is declared, you will get a **ReferenceError** due to the **temporal dead zone**.

```javascript
console.log(a); // ReferenceError! Cannot access 'a' before initialization
const a = 5;
```

6. What happens when you declare objects or arrays with `const`?

- **Answer:** When you declare an object or an array with `const`, you cannot reassign the entire object or array, but you **can modify** the contents (properties of the object or elements of the array).

```javascript
const arr = [1, 2, 3];
arr.push(4); // Allowed, you can modify the array
console.log(arr); // Output: [1, 2, 3, 4]

arr = [5, 6, 7]; // Error! Assignment to constant variable is not allowed
```

7. Why should you use `const` in JavaScript?

- **Answer:** Using `const` improves code safety by preventing reassignment of variables. If you know a value should not change, declaring it as a constant helps avoid accidental modifications, making your code more robust and easier to understand.

javascript Copy code

```javascript
const maxValue = 100;
maxValue = 200; // Error! This ensures the value doesn't accidentally change
```

8 What happens if you use `const` in a loop?

- **Answer:** If you use `const` in a loop, you cannot reassign the variable within that loop. However, you can declare a new `const` for each iteration of the loop (since `const` is block-scoped).

javascript Copy code

```javascript
for (const i = 0; i < 3; i++) {
    console.log(i); // Error! Reassignment not allowed for const in loops
}
```

But using `const` works fine in `for...of` loops or when no reassignment is needed inside the loop:

javascript Copy code

```javascript
const arr = [10, 20, 30];
for (const value of arr) {
    console.log(value); // Output: 10, 20, 30 (works fine with no reassignment)
}
```

9 Is it good practice to always use `const`?

- **Answer:** Yes, it is a good practice to use `const` by default when declaring variables that should not change. This makes your code more predictable and helps prevent accidental reassignment. Use `let` only when you know the variable will change over time.

```javascript
const pi = 3.14159;  // Use const for values that won't change
let count = 0;       // Use let for values that can change
```

10 What are the advantages of using `const` over `let` and `var`?

- **Answer:**

  1. **Ensures Immutability:** `const` helps prevent reassignment, making your code more predictable.
  2. **Block Scope:** Like `let`, `const` is block-scoped, ensuring that the variable doesn't leak outside its intended scope.
  3. **Clearer Intent:** Using `const` makes it clear to other developers that the variable's value should not be changed.

SECTION # 04

FINAL-SECTION

1. **What are the main differences between `var`, `let`, and `const`?**

- Answer:
  1. Scope:
     - `var` is function-scoped or globally scoped.
     - `let` and `const` are block-scoped, meaning they are confined to the block where they are declared.
  2. Reassignment:
     - `var` can be redeclared and reassigned.
     - `let` can be reassigned but cannot be redeclared in the same scope.
     - `const` cannot be reassigned or redeclared after its initial assignment.
  3. Hoisting:
     - `var` is hoisted and initialized with `undefined`.
     - `let` and `const` are hoisted but are in the **temporal dead zone** and not initialized, causing a ReferenceError if accessed before declaration.

javascript Copy code

```javascript
var x = 5;
let y = 10;
const z = 15;
```

2. When should you use `var`, `let`, or `const`?

- Answer:
  - Use `const` for values that should not change throughout the program. This helps in ensuring immutability.
  - Use `let` when you know the variable's value will change or update (e.g., loop counters, states that change over time).
  - Avoid using `var` unless you're maintaining legacy code. It's considered less safe due to its function scope, hoisting behavior, and the ability to be redeclared.

javascript Copy code

```javascript
const PI = 3.14;   // Use const for constants
let counter = 0;   // Use let for variables that change
var oldVar = "legacy"; // Avoid var, use it only when necessary
```

3. Can you redeclare and update variables declared with `var`, `let`, and `const`?

- Answer:
  - `var` can be both redeclared and updated.
  - `let` can be updated but not redeclared in the same scope.
  - `const` cannot be redeclared or updated after initialization, though the contents of objects or arrays declared with `const` can still be modified.

javascript Copy code

```javascript
var a = 10;
var a = 20; // Redeclaration allowed with var
a = 30;     // Updating allowed with var

let b = 10;
// let b = 20; // Error! Redeclaration not allowed with let
b = 20;     // Updating allowed with let

const c = 10;
// c = 20;   // Error! Reassignment not allowed with const
```

4. What happens when you declare `var`, `let`, and `const` inside a block or loop?

- Answer:
  - `var` is **not block-scoped**, so it will be accessible outside the block (except in functions).
  - `let` and `const` are block-scoped, so they exist only within the block they are declared in (like within `if`, `for`, `while` loops, etc.).

javascript Copy code

```javascript
if (true) {
    var x = 1;   // Accessible outside the block
    let y = 2;   // Block-scoped
    const z = 3; // Block-scoped
}
console.log(x); // Output: 1
console.log(y); // Error! y is not defined outside the block
console.log(z); // Error! z is not defined outside the block
```

5. How does hoisting affect `var`, `let`, and `const`?

- Answer:
  - `var` is hoisted and initialized with `undefined`, meaning you can use it before its declaration, but it will hold the value `undefined`.
  - `let` and `const` are also hoisted, but they are not initialized and exist in a **temporal dead zone** until their declaration is encountered, leading to a **ReferenceError** if accessed before initialization.

```javascript
console.log(a); // Output: undefined (var is hoisted)
var a = 10;

console.log(b); // ReferenceError! Cannot access 'b' before initialization
let b = 20;

console.log(c); // ReferenceError! Cannot access 'c' before initialization
const c = 30;
```

6. Why is `const` considered better for constants, and why should `let` be preferred over `var`?

- Answer:
  - `const` ensures that the value cannot be accidentally reassigned, making your code more predictable and less prone to bugs.
  - `let` is safer than `var` because it is block-scoped, avoids issues with redeclaration, and helps prevent errors related to hoisting.