**VARIABLES & DATA-TYPES**

**What Are Variables in JavaScript?**

In JavaScript, variables are containers for storing data values. They act like labeled boxes, where each label (the variable name) points to a specific value or piece of information. Variables allow you to reuse and manipulate data throughout your program.

For example, you can store a name, a number, or an object in a variable and use it later in your script.

**Declaring Variables in JavaScript**

JavaScript provides three keywords to declare variables: **var**, **let**, and **const**. Each has different rules and use cases.

**1. var (Function-Scoped Variables)**

* Introduced in the earliest versions of JavaScript.
* Variables declared with var are **function-scoped**, meaning they are accessible within the function they are defined in, but not outside of it.
* var variables are **hoisted**, meaning they are moved to the top of their scope during the execution phase. However, they are initialized with undefined until their declaration is encountered.

**Example:**

var name = "John";

console.log(name); // Output: John

function example() {

var age = 25;

console.log(age); // Output: 25

}

example();

// console.log(age); // Error: age is not defined (outside function scope)

**2. let (Block-Scoped Variables)**

* Introduced in **ES6** (2015).
* Variables declared with let are **block-scoped**, meaning they are only accessible within the block ({}) in which they are defined.
* Unlike var, let variables are not initialized until their declaration is encountered, which prevents unexpected behavior.

**Example:**

let city = "New York";

console.log(city); // Output: New York

if (true) {

let country = "USA";

console.log(country); // Output: USA

}

// console.log(country); // Error: country is not defined (outside block scope)

**3. const (Block-Scoped Constants)**

* Also introduced in **ES6**.
* Variables declared with const are **block-scoped**, similar to let.
* The value of a const variable **cannot be reassigned** after its initial declaration. However, for objects or arrays, you can still modify their contents (e.g., add/remove properties or elements).
* const is often used for values that should not change, like configuration settings or constants.

**Example:**

const pi = 3.14159;

console.log(pi); // Output: 3.14159

// pi = 3.14; // Error: Assignment to constant variable

const colors = ["red", "blue"];

colors.push("green"); // Modifying the array is allowed

console.log(colors); // Output: ["red", "blue", "green"]

**Key Differences Between var, let, and const**

| **Feature** | **var** | **let** | **const** |
| --- | --- | --- | --- |
| **Scope** | Function-scoped | Block-scoped | Block-scoped |
| **Re-declaration** | Allowed | Not allowed | Not allowed |
| **Re-assignment** | Allowed | Allowed | Not allowed |
| **Hoisting** | Hoisted (initialized to undefined) | Hoisted (not initialized) | Hoisted (not initialized) |

**When to Use var, let, and const**

1. Use const when you do not intend to reassign the variable. This is the preferred choice for most variables.
2. Use let when you expect the value to change, such as in loops or conditional blocks.
3. Avoid using var in modern JavaScript unless you need to support very old browsers, as it can lead to unexpected behavior due to its function-scoping and hoisting.

**Example Combining All:**

function demo() {

var globalVar = "I am function-scoped";

let blockVar = "I am block-scoped";

const constantVar = "I cannot be changed";

console.log(globalVar); // Output: I am function-scoped

console.log(blockVar); // Output: I am block-scoped

console.log(constantVar); // Output: I cannot be changed

}

demo();

**Question 2: Explain the different data types in JavaScript. Provide examples for each.**

JavaScript supports a variety of data types that can be broadly categorized into **primitive** and **non-primitive (object)** types. These data types help you store and manipulate different kinds of information.

**1. Primitive Data Types**

Primitive types are basic data types that represent a single value and are immutable.

**a. Number**

Represents numeric values, including integers and floating-point numbers. Special numeric values include:

* Infinity
* -Infinity
* NaN (Not-a-Number)

**Example:**

let age = 25; // Integer

let price = 99.99; // Floating-point number

console.log(1 / 0); // Output: Infinity

console.log("abc" \* 2); // Output: NaN

**b. String**

Represents text or sequences of characters, enclosed in single ('), double ("), or backticks (`).

**Example:**

let name = "John"; // Double quotes

let greeting = 'Hello'; // Single quotes

let message = `Hi, ${name}!`; // Template literal with variable interpolation

console.log(message); // Output: Hi, John!

**c. Boolean**

Represents logical values: true or false.

**Example:**

let isOnline = true;

let isAdmin = false;

console.log(isOnline); // Output: true

**d. Undefined**

A variable is undefined if it has been declared but not assigned a value.

**Example:**

let myVar;

console.log(myVar); // Output: undefined

**e. Null**

Represents the intentional absence of any value. It is a primitive type with a single value: null.

**Example:**

let emptyValue = null;

console.log(emptyValue); // Output: null

**f. Symbol (Introduced in ES6)**

Represents unique identifiers. Symbols are useful for creating unique property keys in objects.

**Example:**

let sym1 = Symbol("id");

let sym2 = Symbol("id");

console.log(sym1 === sym2); // Output: false (Symbols are always unique)

**g. BigInt (Introduced in ES11)**

Used to represent integers larger than the Number type can handle (greater than 253−12^{53} - 1).

**Example:**

let bigNumber = 1234567890123456789012345678901234567890n;

console.log(bigNumber); // Output: 1234567890123456789012345678901234567890n

**2. Non-Primitive (Object) Data Types**

Objects are more complex data structures that can hold collections of values and functions.

**a. Object**

An object is a collection of key-value pairs.

**Example:**

let person = {

name: "Alice",

age: 30,

isEmployed: true

};

console.log(person.name); // Output: Alice

**b. Array**

Arrays are special types of objects used to store ordered collections of items.

**Example:**

let colors = ["red", "green", "blue"];

console.log(colors[0]); // Output: red

**c. Function**

Functions are objects that can be executed. They represent reusable blocks of code.

**Example:**

function greet(name) {

return `Hello, ${name}!`;

}

console.log(greet("Bob")); // Output: Hello, Bob!

**d. Date**

A built-in object used for handling dates and times.

**Example:**

let now = new Date();

console.log(now); // Output: Current date and time

**3. Special Data Type: typeof**

The typeof operator helps determine the type of a variable.

**Example:**

console.log(typeof 42); // Output: number

console.log(typeof "hello"); // Output: string

console.log(typeof true); // Output: boolean

console.log(typeof undefined); // Output: undefined

console.log(typeof null); // Output: object (historical quirk)

console.log(typeof Symbol("id")); // Output: symbol

console.log(typeof 123n); // Output: bigint

console.log(typeof { key: "value" }); // Output: object

console.log(typeof [1, 2, 3]); // Output: object (arrays are objects)

**Summary of Data Types**

| **Category** | **Type** | **Examples** |
| --- | --- | --- |
| **Primitive** | Number | 42, 3.14, Infinity, NaN |
|  | String | "hello", 'world', `template` |
|  | Boolean | true, false |
|  | Undefined | undefined |
|  | Null | null |
|  | Symbol | Symbol("unique") |
|  | BigInt | 12345678901234567890n |
| **Non-Primitive** | Object | { key: "value" } |
|  | Array | ["red", "green", "blue"] |
|  | Function | function () {} |
|  | Date | new Date() |

Each data type serves a specific purpose, and understanding them helps you handle data more effectively in JavaScript programs.

**Question 3: What is the difference between undefined and null in JavaScript?**

**Difference Between undefined and null in JavaScript**

In JavaScript, undefined and null are both primitive types used to represent the absence of a value, but they serve different purposes and are used in distinct contexts.

**1. undefined**

* **Meaning:**  
  Indicates that a variable has been declared but has not yet been assigned a value. It also represents the default return value of functions that don’t explicitly return anything.
* **Type:**  
  undefined is its own primitive type.
* **Use Cases:**
  + When you declare a variable without initializing it:
  + let x;
  + console.log(x); // Output: undefined
  + When you try to access a property that doesn’t exist in an object:
  + let person = {};
  + console.log(person.age); // Output: undefined
  + Default return value of a function:
  + function example() {}
  + console.log(example()); // Output: undefined

**2. null**

* **Meaning:**  
  Represents the intentional absence of any value. It is explicitly assigned to indicate "no value" or "empty."
* **Type:**  
  Although null is a primitive type, the typeof operator mistakenly returns object for historical reasons:
* console.log(typeof null); // Output: object (a known quirk)
* **Use Cases:**
  + When you want to explicitly indicate that a variable has no value:
  + let y = null;
  + console.log(y); // Output: null
  + When working with APIs or databases, null is often used to indicate a "missing" or "empty" value.

**Key Differences**

| **Feature** | **undefined** | **null** |
| --- | --- | --- |
| **Definition** | A variable is declared but not assigned a value. | Explicitly assigned to indicate "no value." |
| **Type** | undefined | object (due to a historical quirk) |
| **Default Value** | Default value for uninitialized variables, function parameters, or return values. | Not assigned by default; must be set explicitly. |
| **Use Case** | Represents "value not assigned." | Represents "no value" or "empty." |
| **Intentionality** | Generally unintentional. | Explicitly assigned to indicate "nothing." |

**Examples of Usage**

**Scenario 1: Uninitialized Variable vs. Explicitly Empty Variable**

let a; // uninitialized

let b = null; // explicitly empty

console.log(a); // Output: undefined

console.log(b); // Output: null

**Scenario 2: Object Properties**

let car = { brand: "Toyota" };

console.log(car.model); // Output: undefined (property doesn't exist)

car.model = null; // Explicitly setting it to "no value"

console.log(car.model); // Output: null

**Scenario 3: Comparison**

* Loose Equality (==): undefined and null are considered loosely equal because they both represent absence of value:
* console.log(undefined == null); // Output: true
* Strict Equality (===): They are not strictly equal because they are different types:
* console.log(undefined === null); // Output: false

**Best Practices**

1. Use undefined for uninitialized variables or missing properties (this happens automatically).
2. Use null when you want to explicitly assign "no value" to a variable.

Understanding the difference between undefined and null helps you write clearer, more intentional JavaScript code.