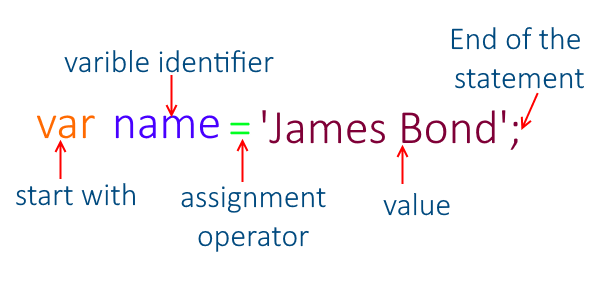
**COMMON FEATURES IN PROGRAMMING LANGUAGES**

**VARIABLES:**



In JavaScript, **variables** are containers for storing data. They allow you to hold values like numbers, strings, booleans, objects, and more. Let’s explore how to declare variables in JavaScript with some examples:

1. **Automatically Declared Variables:**
   * When you use a variable without explicitly declaring it, JavaScript automatically creates it. For instance:

**x = 5;**

**y = 6;**

**z = x + y; // z will be 11**

* + However, it’s good practice to **always declare variables** before using them.

1. **Using var:**
   * The var keyword was commonly used in JavaScript from 1995 to 2015:

**var x = 5;**

**var y = 6;**

**var z = x + y; // z will be 11**

1. **Using let:**
   * Introduced in 2015, let allows you to declare variables with block scope:

**let x = 5;**

**let y = 6;**

**let z = x + y; // z will be 11**

1. **Using const:**
   * Declare constants with const. These values cannot be changed:

**const x = 5;**

**const y = 6;**

**const z = x + y; // z will be 11**

1. **Mixed Example:**
   * You can mix const and let:

**const price1 = 5;**

**const price2 = 6;**

**let total = price1 + price2; // total can be changed**

Remember these guidelines:

* Always declare variables.
* Use const for unchanging values.
* Use let only when you can’t use const.
* Use var only for older browsers. Just like in algebra, variables hold values, and they’re used in expressions to calculate results!
* **Variable Declaration:**

In JavaScript, you can declare a variable using the var, let, or const keyword.

**Example:**

var myVar; // Declaration using var

let myLet; // Declaration using let

const myConst = 10; // Declaration and initialization using const

* **Variable Initialization:**

Variables can be initialized with an initial value at the time of declaration or later in the code.

**Example:**

var myVar = 5; // Declaration and initialization using var

let myLet; // Declaration using let

myLet = 10; // Initialization later in the code

* **Dynamic Typing:**

JavaScript is dynamically typed, meaning you don't have to specify the data type of a variable explicitly.

**Example:**

var myVar = 5; // myVar is a number

myVar = "Hello"; // myVar is now a string

* **Variable Scope:**

JavaScript variables have function-level scope with var and block-level scope with let and const.

**Example:**

var x = 10;

if (true) {

var y = 20; // y is accessible outside the block

let z = 30; // z is only accessible within the block

}

console.log(x); // 10

console.log(y); // 20

console.log(z); // ReferenceError: z is not defined

* **Naming Convention:**

Variable names can consist of letters, digits, underscores, and dollar signs. They must start with a letter, underscore, or dollar sign.

**Example:**

var myVariable;

var my\_var;

var $myVar;

var \_myVar;

Variables in JavaScript are fundamental for storing and manipulating data, and they play a crucial role in writing dynamic and interactive web applications.

**Variables in other programming languages**

**JavaScript**

* **Declaration**: **var**, **let**, **const**
* **Example**:

javascript

let name = "Alice";

const age = 25;

var isStudent = true;

**Python**

* **Declaration**: No explicit keyword; variables are created upon assignment
* **Example**:

python

name = "Alice"

age = 25

is\_student = True

**Java**

* **Declaration**: Explicit type declaration required
* **Example**:

java

String name = "Alice";

int age = 25;

boolean isStudent = true;

**C++**

* **Declaration**: Explicit type declaration required
* **Example**:

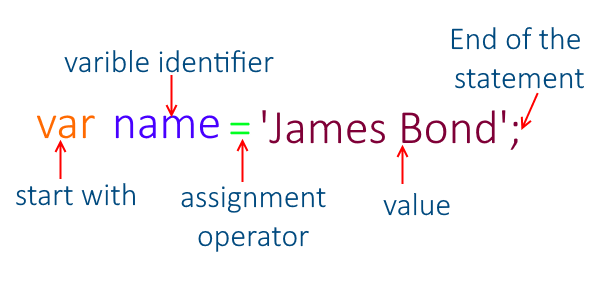
C++

std::string name = "Alice";

int age = 25;

bool isStudent = true;

**Variables**



**variables are like containers which are used to store the data and manipulate data**

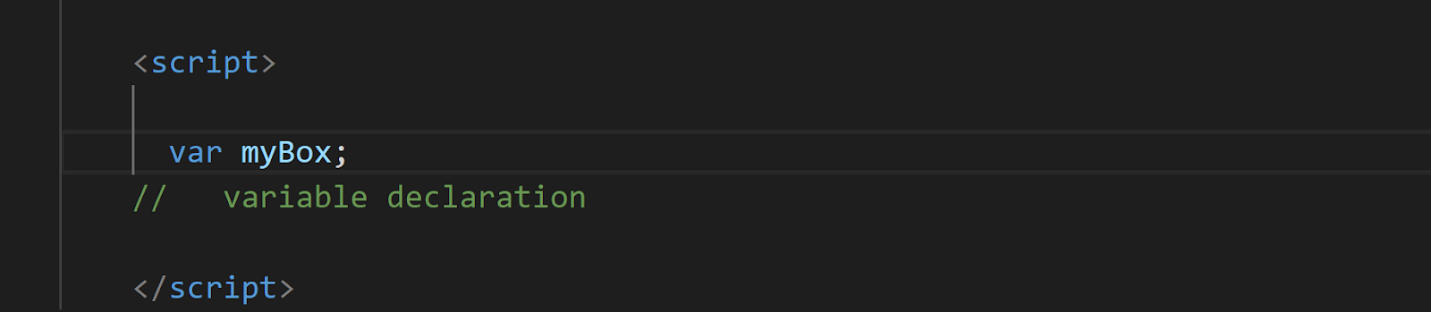
That's a perfect way to describe variables. Just like containers, variables hold values that you can use and manipulate in your code. Whether it's numbers, text, lists, or more complex data, variables provide a way to store and access information in your programs. And just like how you can label containers to know what's inside, you give variables names so you can refer to them later in your code. So, think of variables as the containers of information in your program!

Imagine you have a box. This box is like a container where you can keep different things. You can give this box a name, like "my-Box". Inside this box, you can put anything you want, like toys, books, or snacks.

**Naming the Box:** You give your box a name, like "myBox". This name helps you remember what's inside. (Variable Declaration)

 Declaration:

- Definition: Declaration is the process of introducing a variable to the programming environment. It tells the compiler or interpreter the name and data type of the variable.

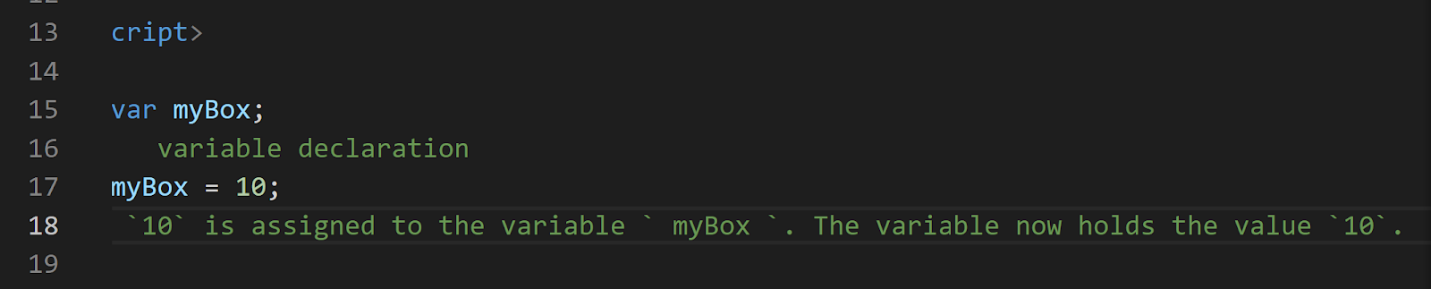


var is used to declare a variable in JavaScript. It tells the JavaScript engine to allocate memory for the variable and associate it with a particular name.

**Adding Items**: You can put things into your box, like toys, books, or snacks. (Variable assignment)

 Assignment:

   - Definition: Assignment is the process of giving a value to a variable that has already been declared.



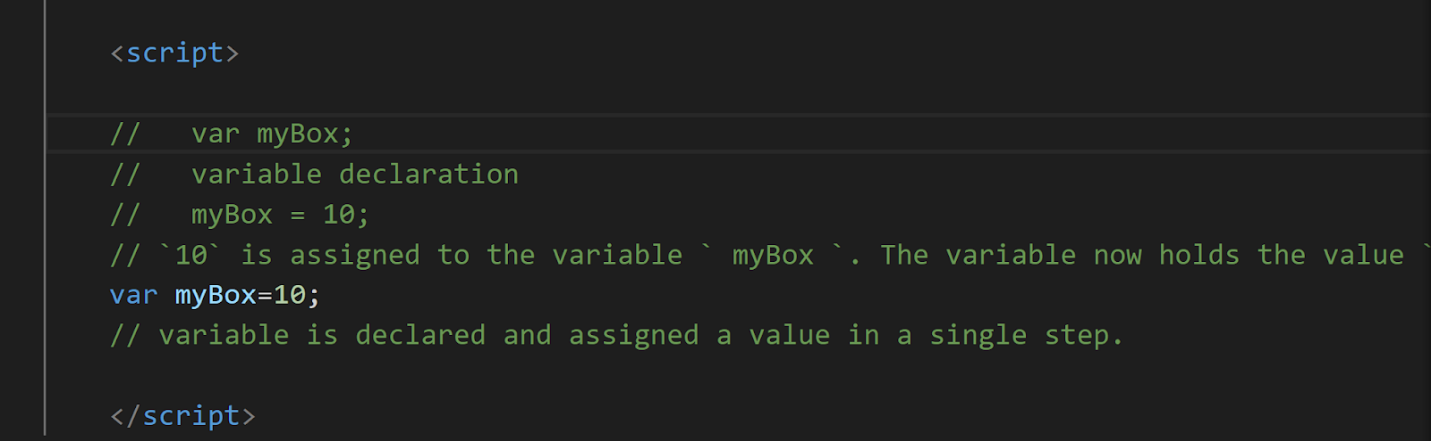
**Changing Items:** If you want, you can take something out of the box and put something else in its place. For example, you can take out a toy and put in a book instead. (Variable Reassignment)

**Reusing the Box**: You can use the same box to hold different things at different times. Today it might hold toys, tomorrow it might hold snacks. (Variable Reuse)

So, a variable in programming is like this box. It's a named container where you can store different types of information, change what's inside, and reuse it as needed in your code.

Initialization:

- Definition: Initialization is a specific type of assignment that occurs when a variable is declared and assigned a value in a single step.



To summarize:

- Declaration introduces a variable to the environment.

- Assignment gives a value to a variable that has already been declared.

- Initialization combines declaration and assignment in a single step, assigning a value to a variable at the time of declaration.

Let’s dive into the concept:

Declaration:

JavaScript Variables can be declared in 4 ways:

* Automatically
* Using var
* Using let
* Using const

Here we will learn about

Automatically, var

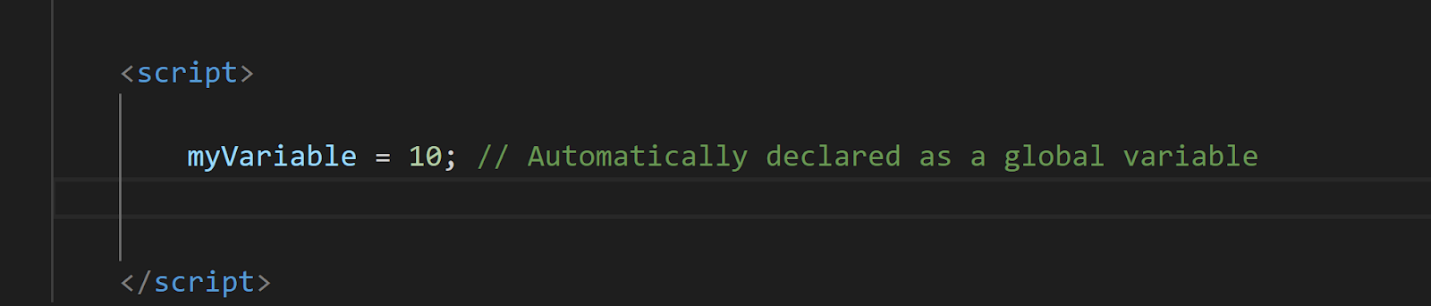
In the further classes we will learn about let const and var in deeper, because these need some basic understanding of blocks, functions etc…

Automatically:

Variables in JavaScript can be automatically declared when they are first assigned a value without explicitly using a declaration keyword (var, let, const).

When you assign a value to an identifier that hasn't been declared previously, JavaScript automatically creates a global variable (if not in strict mode) or throws a ReferenceError (in strict mode).

This behavior is generally discouraged as it can lead to accidental creation of global variables and make code harder to understand and maintain



Using var:

The var keyword is traditionally used to declare variables in JavaScript.

Variables declared with var have function scope, meaning they are accessible within the function in which they are declared or globally if declared outside of any function.

Variables declared with var can be redeclared and reassigned.



JavaScript Identifiers

All JavaScript variables must be identified with unique names.

These unique names are called identifiers.

Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

The general rules for constructing names for variables (unique identifiers) are:

**Names can contain letters, digits, underscores, and dollar signs.**

**Names must begin with a letter.**

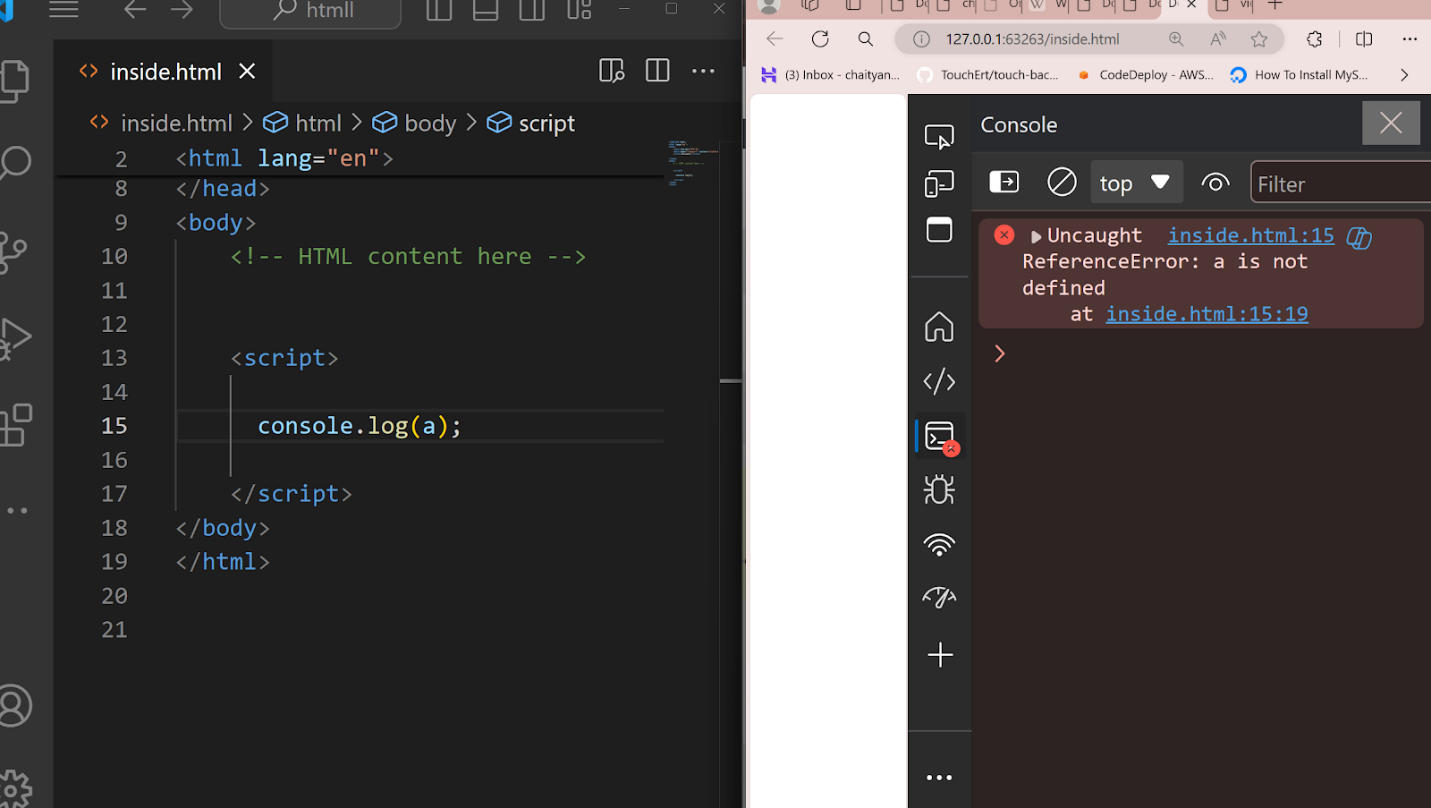
**Names can also begin with $ and \_ (but we will not use it in this tutorial).**

**Names are case sensitive (y and Y are different variables).**

**Reserved words (like JavaScript keywords) cannot be used as names.**

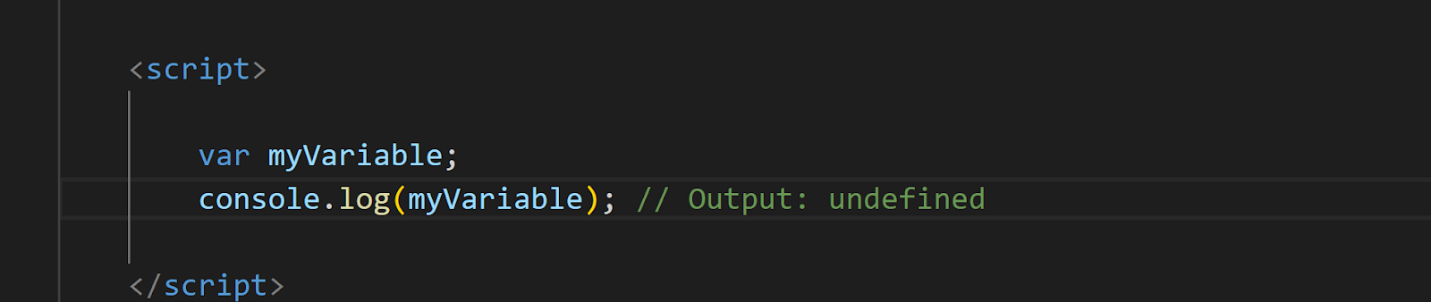
Without declaring a variable

In JavaScript, if you attempt to use a variable without declaring it first, you'll encounter a "not defined" error. This is known as a ReferenceError.



In this example, `a ` is being used without being declared first. JavaScript doesn't automatically create variables if they're referenced before declaration, unlike some other programming languages. This behavior helps catch potential errors in your code and encourages good programming practices like declaring variables before using them.

In JavaScript, if you declare a variable but don't assign it a value, its default value will be **undefined**



You will learn more about undefined in datatypes chapter.

Console.log ---let’s try to understand this:

In JavaScript, `console.log()` is a function used to output information to the console. The console is a built-in tool available in most web browsers' developer tools, which allows developers to debug their JavaScript code by inspecting values, logging messages, and viewing errors.

Debugging with `console.log()` is a common practice in JavaScript development. Here's how you can use it effectively for debugging purposes:

**1. Inspecting Variable Values:**

   - Use `console.log()` to print the current value of variables at different points in your code.

   - This helps you verify whether variables hold the expected values and track how they change during program execution.

**2. Logging Control Flow:**

   - Insert `console.log()` statements within loops, conditionals, or function calls to trace the flow of execution through your code.

   - This allows you to understand which parts of your code are being executed and in what order.

**3. Identifying Errors:**

   - Use `console.log()` to print error messages or diagnostic information when unexpected behaviour occurs.

   - This helps you identify the source of errors and understand what went wrong in your code.

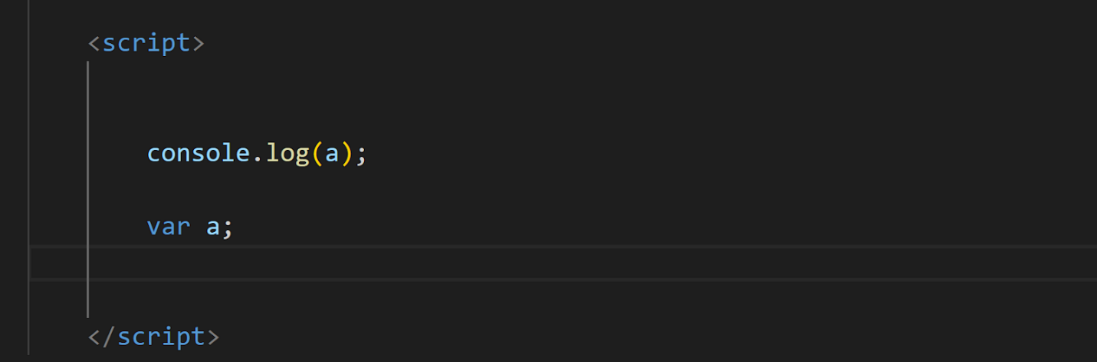
**5. Performance Profiling:**

   - Use `console.time()` and `console.timeEnd()` to measure the execution time of specific code blocks.

   - This helps you identify performance bottlenecks and optimize critical parts of your code.

By strategically placing `console.log()` statements throughout your code, you can gain insights into its behaviour, diagnose issues, and improve the overall quality of your JavaScript applications.

Let’ s try the snippet:



We will think that the answer for the above snippet is

Error – a is not defined

If we think like this then its good, we are in the right path of the learning journey

But javascript working has some other concept like

when you reference a variable before it's declared, you won't get a "not defined" error. Instead, due to a behaviour called "hoisting," the variable declaration is moved to the top of its containing scope during the compilation phase. This means that the variable is effectively declared before any code is executed, even if its initialization happens later in the code.

In this case:

1. `var a;` declares the variable `a`, but it doesn't assign any value to it.

2. When `console.log(a)` is encountered, `a` is declared due to hoisting, but it's not yet initialized, so its value is `undefined`.

3. Therefore, when `console.log(a)` is executed, it outputs `undefined` to the console.

So we will get the answer as – undefined

**JavaScript hoisting** is a mechanism where variable and function declarations are moved to the top of their containing scope during the compilation phase. JavaScript hoisting is a important concept for understanding how variables and functions are processed during code execution. This guide will delve into the features, nuances, and implications of hoisting in JavaScript, covering key topics such as variable declaration, function scoping, and the differences between var, let, and const declarations

**DATA TYPES:**

In JavaScript, data types represent the different kinds of values you can use. Let’s explore the **primitive data types** along with examples:

1. **String**: Represents textual data. You can use single quotes ('), double quotes ("), or backticks (`) to define strings.

let fruit = 'apple';

let country = "USA";

let result = `fail`;

1. **Number**: Represents both integers and floating-point numbers.

let integer\_number = -3; // Integer

let float\_number = 3.15; // Floating-point number

1. **BigInt**: Handles very large or very small integers beyond the regular number data type’s range.

let bigIntValue = 900719925124740999n;

1. **Boolean**: Represents true or false.

let isSunny = true;

let isRainy = false;

1. **Undefined**: Denotes a variable that is not initialized.

let x;

console.log(x); // Outputs: undefined

1. **Null**: Represents a null value.

let emptyObject = null;

console.log(emptyObject); // Outputs: null

1. **Symbol**: Instances are unique and immutable.

let uniqueSymbol = Symbol('hello');

let sym1 = Symbol("mySymbol");

let sym2 = Symbol("mySymbol"); // These will be different symbols even with the same name

console.log(sym1 === sym2); // Outputs: false

Remember, these are **primitive data types**. There are also **non-primitive data types** like objects, which can hold multiple values. Objects consist of key-value pairs, representing collections of data:

**Composite/ non-primitive Data Types:**

**8. Objects:** Unordered collections of key-value pairs. Keys are typically strings, and values can be any data type. Objects are used to store and organize complex data.

* **Example:**

let student = { name: "John" };

let person = {

name: "Bob",

age: 25,

hobbies: ["coding", "reading"]

};

console.log(person.name); // Outputs: "Bob" (accessing by key)

**9. Arrays:** Ordered collections of items, similar to lists in other languages. Arrays can hold elements of any data type, including other arrays.

* **Example:**

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

console.log(colors[1]); // Outputs: "green" (accessing by index)

**Special Data Types:**

* + These data types are used for special purposes in JavaScript.
  + **Function:** Represents a block of reusable code.

function greet(name) {

return "Hello, " + name + "!";

}

* + **Date:** Represents date and time values.

var now = new Date();

**Checking Data Types:**

You can use the typeof operator to check the data type of a variable at runtime.

let num = 10;

console.log(typeof num); // Outputs: "number"

let str = "hello";

console.log (typeof str); // Outputs: "string"

data types is essential for writing efficient and maintainable JavaScript code. By using the appropriate data type for your variables, you can avoid errors and make your code more readable.

**Data Types in other programming languages**

**1. C**

* **int**: Integer type

int age = 30;

* **float**: Floating-point type

float salary = 12345.67;

* **double**: Double precision floating-point type

double pi = 3.141592653589793;

* **char**: Character type

char grade = 'A';

* **void**: Represents the absence of type (commonly used in functions)

void sayHello() { printf("Hello, World!"); }

* **Derived Data Types**:
  + **Pointer**:

int\* ptr; int var = 10; ptr = &var;

* + **Array**:

int numbers[5] = {1, 2, 3, 4, 5};

* + **Structure (struct)**:

struct Person { char name[50]; int age; }; struct Person person1;

union Data { int i; float f; char str[20]; }; union Data data;

* + **Enumeration (enum)**:

enum Weekday {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday}; enum Weekday today = Monday;

**2. Java**

* **int**: Integer type

int age = 30;

* **float**: Single precision floating-point type

float salary = 12345.67f;

* **double**: Double precision floating-point type

double pi = 3.141592653589793;

* **char**: Single 16-bit Unicode character

char grade = 'A';

* **boolean**: Boolean type

boolean isJavaFun = true;

* **Reference Data Types**:
  + **Class**:

public class Person { String name; int age; Person(String name, int age) { this.name = name; this.age = age; } } Person person1 = new Person("Alice", 30);

* + **Array**:

int[] numbers = {1, 2, 3, 4, 5};

* + **Interface**:

interface Animal { void makeSound(); } class Dog implements Animal { public void makeSound() { System.out.println("Woof"); } }

* + **Enum**:

enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY } Day today = Day.MONDAY;

**3. Python**

* **int**: Integer type

age = 30

* **float**: Floating-point type

salary = 12345.67

* **complex**: Complex number type

z = 1 + 2j

* **bool**: Boolean type

is\_python\_fun = True

* **str**: String type

name = "Alice"

* **list**: Ordered, mutable collection

numbers = [1, 2, 3, 4, 5]

* **tuple**: Ordered, immutable collection

coordinates = (10, 20)

* **set**: Unordered collection of unique items

unique\_numbers = {1, 2, 3, 4, 5}

* **dict**: Unordered collection of key-value pairs

person = {"name": "Alice", "age": 30}

**4. JavaScript**

* **Number**: Represents both integer and floating-point numbers

let age = 30; let salary = 12345.67;

* **String**: Represents a sequence of characters

let name = "Alice";

* **Boolean**: Represents true or false

let isJavaScriptFun = true;

* **Undefined**: Indicates a variable that has not been assigned a value

let x;

* **Null**: Represents the intentional absence of any object value

let y = null;

* **Symbol**: Represents a unique identifier

let sym = Symbol('sym');

* **Object**:

let person = { name: "Alice", age: 30 };

**5. C++**

* **int**: Integer type

int age = 30;

* **char**: Character type

char grade = 'A';

* **float**: Floating-point type

float salary = 12345.67f;

* **double**: Double precision floating-point type

double pi = 3.141592653589793;

* **bool**: Boolean type

bool isCppFun = true;

* **Derived Data Types**:
  + **Array**:

int numbers[5] = {1, 2, 3, 4, 5};

* + **Pointer**:

int\* ptr; int var = 10; ptr = &var;

* + **Reference**:

int var = 10; int& ref = var;

* + **Function**:

void sayHello() { std::cout << "Hello, World!"; }

* **User-Defined Data Types**:
  + **Class**:

class Person { public: std::string name; int age; Person(std::string name, int age) : name(name), age(age) {} }; Person person1("Alice", 30);

* + **Structure (struct)**:

struct Person { std::string name; int age; }; Person person1 = {"Alice", 30};

* + **Union**:

union Data { int i; float f; char str[20]; }; Data data;

* + **Enumeration (enum)**:

enum Weekday {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday}; Weekday today = Monday;

**6. Ruby**

* **Integer**: Whole numbers

age = 30

* **Float**: Floating-point numbers

salary = 12345.67

* **Rational**: Rational numbers

r = Rational(2, 3)

* **Complex**: Complex numbers

c = Complex(1, 2)

* **String**: Sequence of characters

name = "Alice"

* **Symbol**: Immutable, interned string

status = :active

* **Array**: Ordered, integer-indexed collection of objects

numbers = [1, 2, 3, 4, 5]

* **Hash**: Collection of key-value pairs

person = {name: "Alice", age: 30}

* **Range**: Represents an interval

range = (1..10)

* **Boolean**: Represents true or false values (TrueClass and FalseClass)

is\_ruby\_fun = true

* **NilClass**: Singleton object used to represent an "empty" or "default" value

nothing = nil

Each of these examples illustrates how data types are defined and used in different programming languages. Understanding these data types and their nuances is crucial for effective programming in each language.

Datatypes

data types define the type of values that can be used and manipulated in a program.

Here are the main data types in JavaScript:

1. Primitive Data Types
2. Composite Data Types / non- Primitive Data Types
3. Special Data Types

**Primitive Data Types:**

    - These are basic data types that store single values and stored in stack.

   - They are immutable, meaning their values cannot be changed.

   - Primitive data types are stored directly in memory.

   - Examples:

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

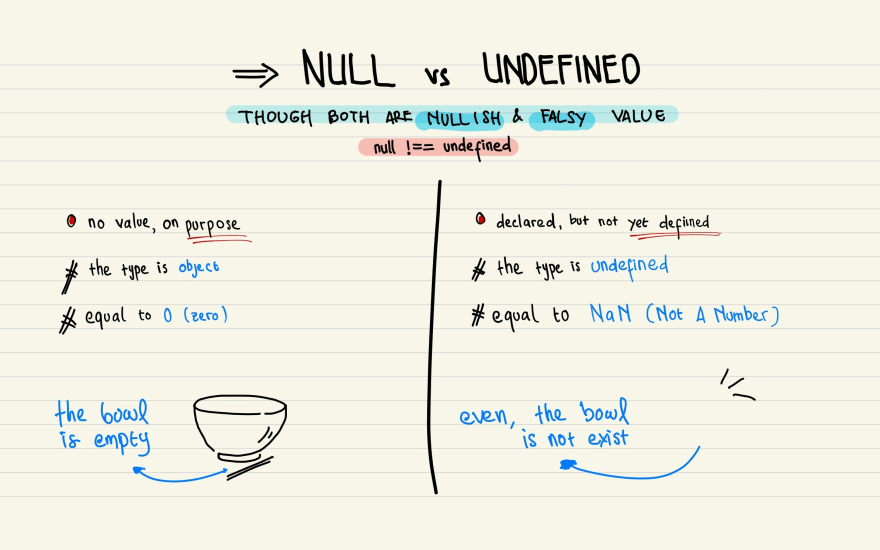
   - String: Represents sequences of characters, enclosed in single (`'`) or double (`"`) quotes.

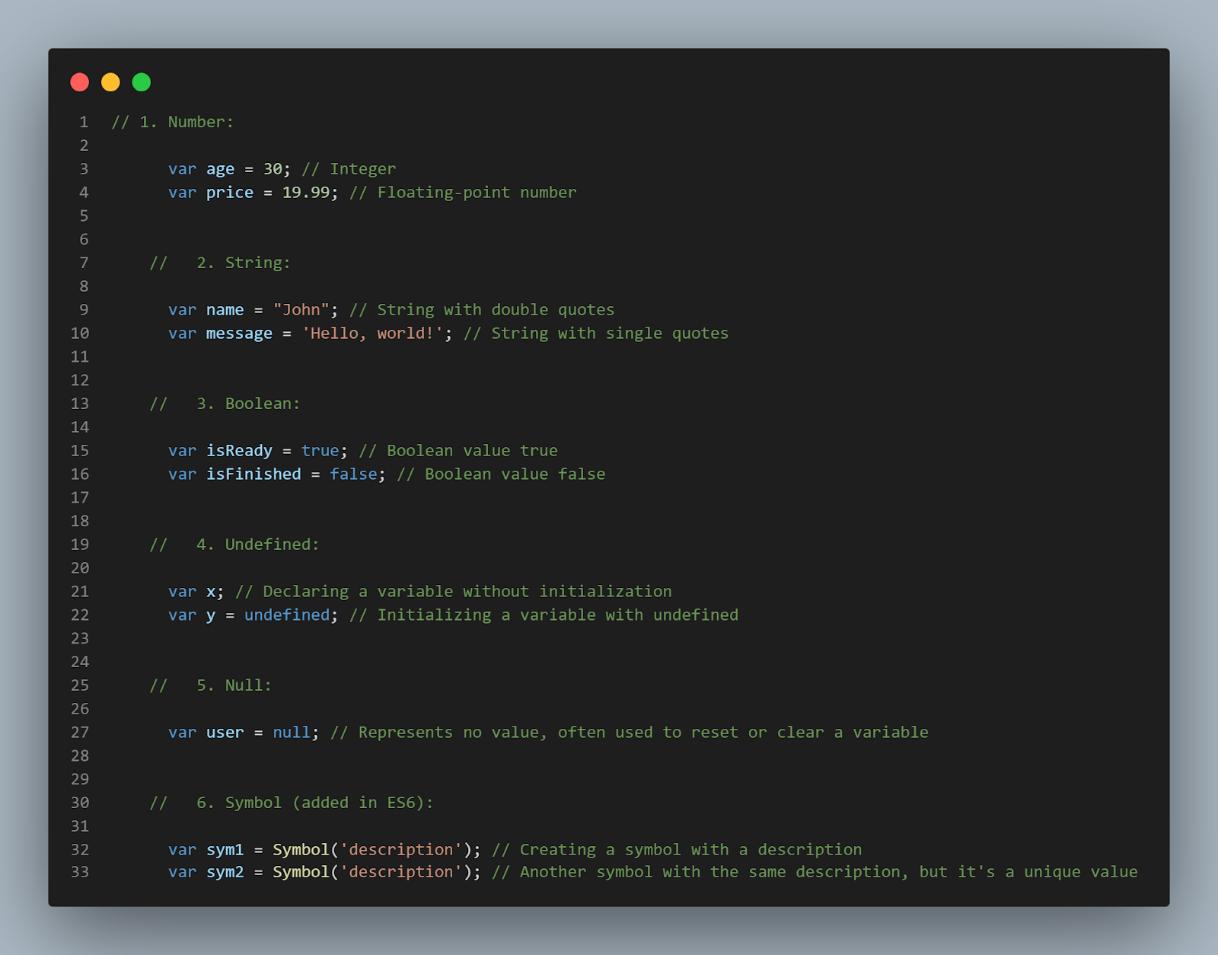
   - Boolean: Represents logical values `true` and `false`.

   - Undefined: Represents a variable that has been declared but not initialized, or a non-existent property of an object.

   - Null: Represents an intentional absence of any value.

   - Symbol (added in ES6): Represents unique identifiers.





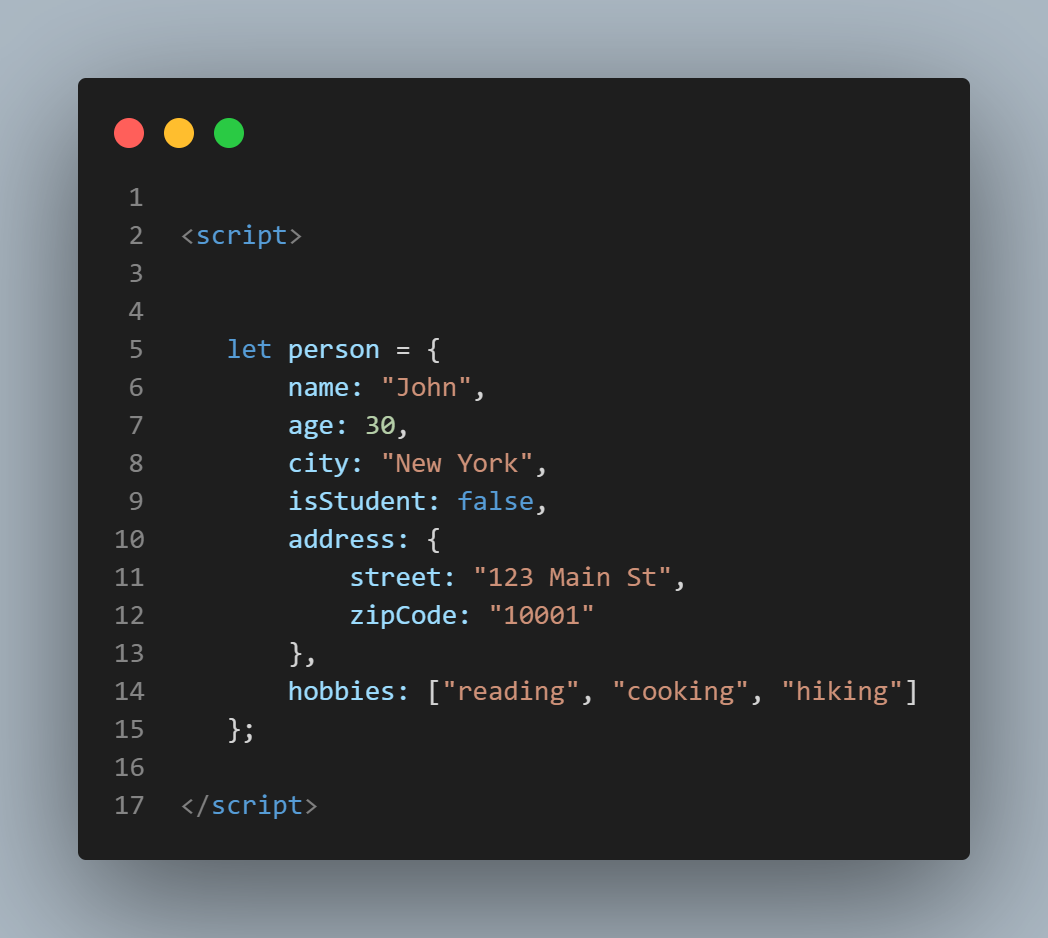
Composite Data Types / non- Primitive Data Types

- These are more complex data types that can store collections of values or references to other values and are stored in heap.

   - They are mutable, meaning their values can be changed.

   - Non-primitive data types are stored as references to memory locations.

Objects: Objects are like containers that hold many different pieces of information, each identified by a name. They can hold numbers, strings, other objects, and more.



Arrays: Arrays are like lists in JavaScript. They can hold multiple values, and each value has an index number that identifies its position in the list



Special Data Types:

   - Function: Functions in JavaScript are a type of object that can be invoked.

   - Date: Represents dates and times.

   - RegExp: Represents regular expressions for pattern matching.

Function:

   - Functions in JavaScript are objects that can be invoked or called.

   - They can be defined using function declarations, function expressions, arrow functions, or methods on objects.

   - Functions can accept parameters and return values.

Date:

   - The `Date` object in JavaScript represents dates and times.

   - It can be used to create new Date objects that represent specific dates or times.

   - It provides methods for manipulating dates, such as getting the current date and time, setting specific date components, and calculating differences between dates.

RegExp:

   - The `RegExp` object in JavaScript represents regular expressions, which are patterns used for pattern matching within strings.

   - Regular expressions can be used to search, match, and replace substrings within strings based on specific patterns.

   - They provide a powerful tool for text manipulation and validation.

We will deal deep in functions, date ,regexp in their particular chapter .

JavaScript also has some additional types and concepts related to handling data, such as:

- Primitive Wrapper Objects: JavaScript provides objects that wrap primitive data types (`Number`, `String`, `Boolean`, `Symbol`) to provide additional functionality.

- Type Conversion: JavaScript automatically converts values between different data types as needed, known as type coercion.

- NaN (Not-a-Number): Represents a value that is not a valid number.

- Infinity and -Infinity: Represent positive and negative infinity, respectively.

- Typeof Operator: Used to determine the data type of a variable or expression.

**0PERATORS:**

Operators in JavaScript are symbols that perform operations on values or variables. They help you manipulate data, control the flow of your program, and create complex expressions. Here's a breakdown of some common JavaScript operators with examples:

**1. Arithmetic Operators:**

These operators perform mathematical calculations on numbers.

* **+ (Addition):** Adds two numbers.

let sum = 10 + 5;

console.log(sum);

// Outputs: 15

* **- (Subtraction):** Subtracts one number from another.

let difference = 20 - 7;

console.log(difference);

// Outputs: 13

* **\* (Multiplication):** Multiplies two numbers.

let product = 3 \* 4;

console.log(product);

// Outputs: 12

* **/ (Division):** Divides one number by another.

let quotient = 30 / 6;

console.log(quotient);

// Outputs: 5

* **% (Modulo)/Remainder:** Returns the remainder after division.

let remainder = 11 % 3;

console.log(remainder);

// Outputs: 2 (11 divided by 3 leaves a remainder of 2)

**2. Comparison Operators:**

These operators compare values and return a boolean (true or false) based on the comparison.

* **== (Equal to):** Checks if two values are equal.

let x = 10;

let y = 5;

console.log (x == y);

// Outputs: false

* **!= (Not equal to):** Checks if two values are not equal.

console.log (x != y);

// Outputs: true

* **> (Greater than):** Checks if one value is greater than another.

console.log (x > y);

// Outputs: true

* **< (Less than):** Checks if one value is less than another.

console.log (x < y);

// Outputs: false

* **>= (Greater than or equal to):** Checks if one value is greater than or equal to another.

let z = 10;

console.log (x >= z);

// Outputs: true

* **<= (Less than or equal to):** Checks if one value is less than or equal to another.

console.log (x <= z);

// Outputs: true

**3. Logical Operators:**

These operators combine boolean expressions and control the flow of your program based on true/false conditions.

* **&& (Logical AND):** Returns true if both operands are true, otherwise false.

let a = true;

let b = false;

console.log (a && b);

// Outputs: false (because b is false)

* **|| (Logical OR):** Returns true if at least one operand is true, otherwise false.

console.log (a || b);

// Outputs: true (because a is true)

* **! (Logical NOT):** Inverts the boolean value of an operand.

console.log(!a);

// Outputs: false (inverts true to false)

**4. Assignment Operators:**

These operators assign values to variables and can also combine assignment with operations.

* **= (Assignment):** Assigns a value to a variable.

let num = 10;

* **+= (Add and Assign):** Adds a value to a variable and then assigns the result back to the variable.

num += 5;

console.log(num);

// Outputs: 15 (num was 10, now it's 15)

* **-= (Subtract and Assign):** Similar to += but subtracts.
* **\*= (Multiply and Assign):** Similar to += but multiplies.
* **/= (Divide and Assign):** Similar to += but divides.

**5.Bitwise Operators**

Perform bitwise operations on binary representations of numbers.

* **AND (&)**: Performs a bitwise AND on two operands.  
  Example: 5 & 1;

// 1

**6.Unary Operators:**

* Operate on a single operand.

var a = 5;

console.log(-a); // Unary minus (-5)

console.log(++a); // Increment (prefix) (6)

console.log (--a); // Decrement (prefix) (5)

**7.Ternary Operator (Conditional Operator):**

* Used for conditional expressions.

var age = 20;

var status = (age >= 18) ? "Adult" : "Minor";

console.log(status);

// Adult

**8.String Concatenation Operators**

* Concatenate strings.
* **Concatenation (+)**: Concatenates two string values.  
  Example:

let greeting = 'Hello, ' + 'world!'; // greeting is 'Hello, world!'

var firstName = "John";

var lastName = "Doe";

var fullName = firstName + " " + lastName;

console.log(fullName); // John Doe

**9. Other Operators:**

JavaScript has several other operators for various tasks:

* **typeof:** Returns the data type of a variable.

let str = "hello";

console.log (typeof str);

// Outputs: "string"

**Operates in other programming languages**

**Operators**

**JavaScript**

* **Arithmetic Operators**: **+**, **-**, **\***, **/**, **%**
* **Comparison Operators**: **==**, **===**, **!=**, **!==**, **<**, **>**, **<=**, **>=**
* **Logical Operators**: **&&**, **||**, **!**
* **Assignment Operators**: **=**, **+=**, **-=**, **\*=**, **/=**
* **Example**:

javascript

let sum = 5 + 10;

let isEqual = (5 === 5);

let isGreater = (10 > 5);

let andResult = (true && false);

**Python**

* **Arithmetic Operators**: **+**, **-**, **\***, **/**, **%**, **//**, **\*\***
* **Comparison Operators**: **==**, **!=**, **<**, **>**, **<=**, **>=**
* **Logical Operators**: **and**, **or**, **not**
* **Assignment Operators**: **=**, **+=**, **-=**, **\*=**, **/=**
* **Example**:

python

sum = 5 + 10

is\_equal = (5 == 5)

is\_greater = (10 > 5)

and\_result = (True and False)

**Java**

* **Arithmetic Operators**: **+**, **-**, **\***, **/**, **%**
* **Comparison Operators**: **==**, **!=**, **<**, **>**, **<=**, **>=**
* **Logical Operators**: **&&**, **||**, **!**
* **Assignment Operators**: **=**, **+=**, **-=**, **\*=**, **/=**
* **Example**:

java

int sum = 5 + 10;

boolean isEqual = (5 == 5);

boolean isGreater = (10 > 5);

boolean andResult = (true && false);

**C++**

* **Arithmetic Operators**: **+**, **-**, **\***, **/**, **%**
* **Comparison Operators**: **==**, **!=**, **<**, **>**, **<=**, **>=**
* **Logical Operators**: **&&**, **||**, **!**
* **Assignment Operators**: **=**, **+=**, **-=**, **\*=**, **/=**
* **Example**:

C++

Copy codeint sum = 5 + 10;

bool isEqual = (5 == 5);

bool isGreater = (10 > 5);

bool andResult = (true && false);

Operators

Javascript operators are used to perform different types of mathematical and logical computations.

1. Arithmetic Operators
2. Assignment Operators
3. Comparison Operators
4. String Operators
5. Logical Operators
6. Bitwise Operators
7. Ternary Operators
8. Type Operators

Arithmetic Operators:

Used to perform arithmetic operations on numbers.

• Addition (+)

• Subtraction (-)

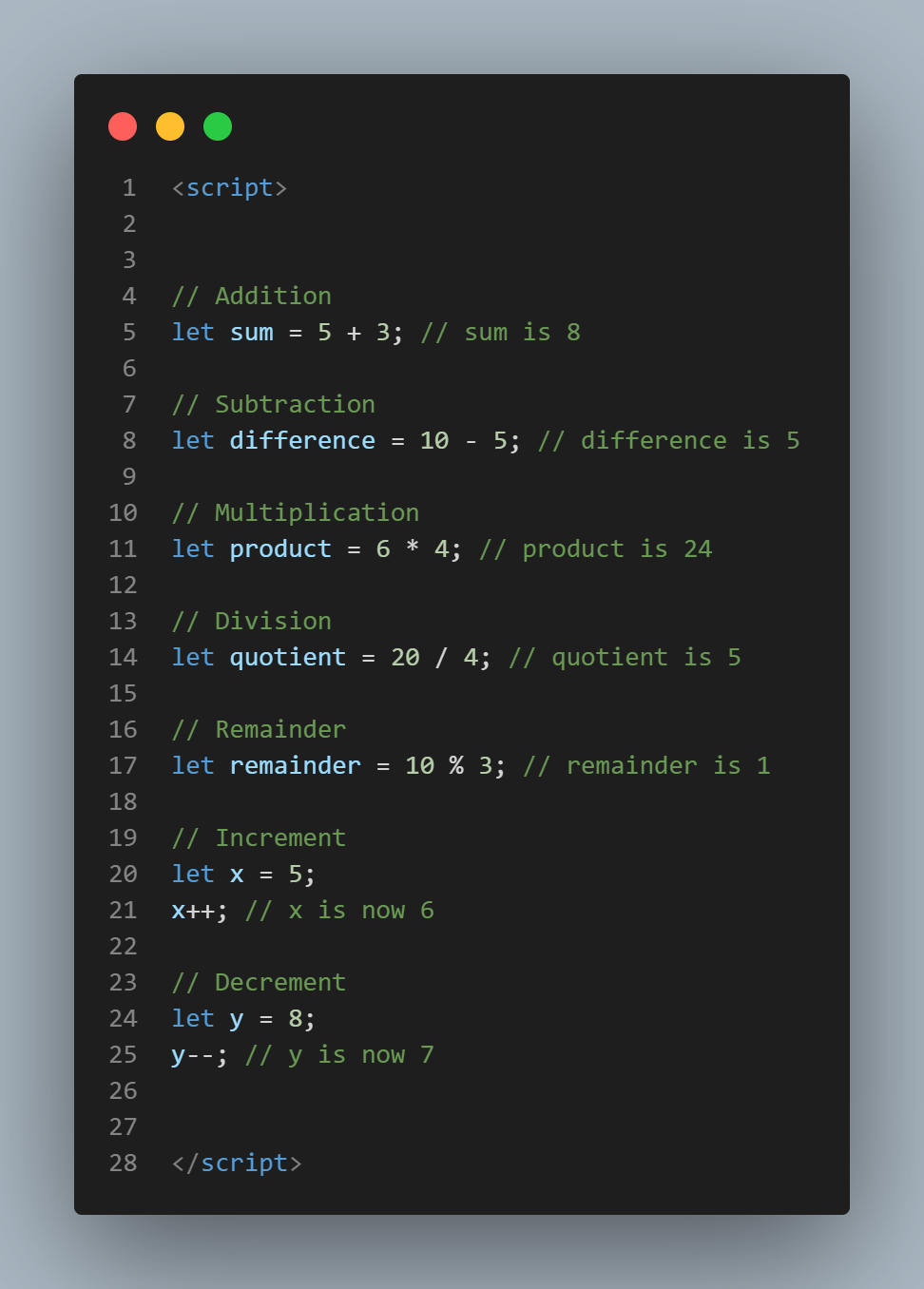
• Multiplication (\*)

• Division (/)

• Remainder (%)

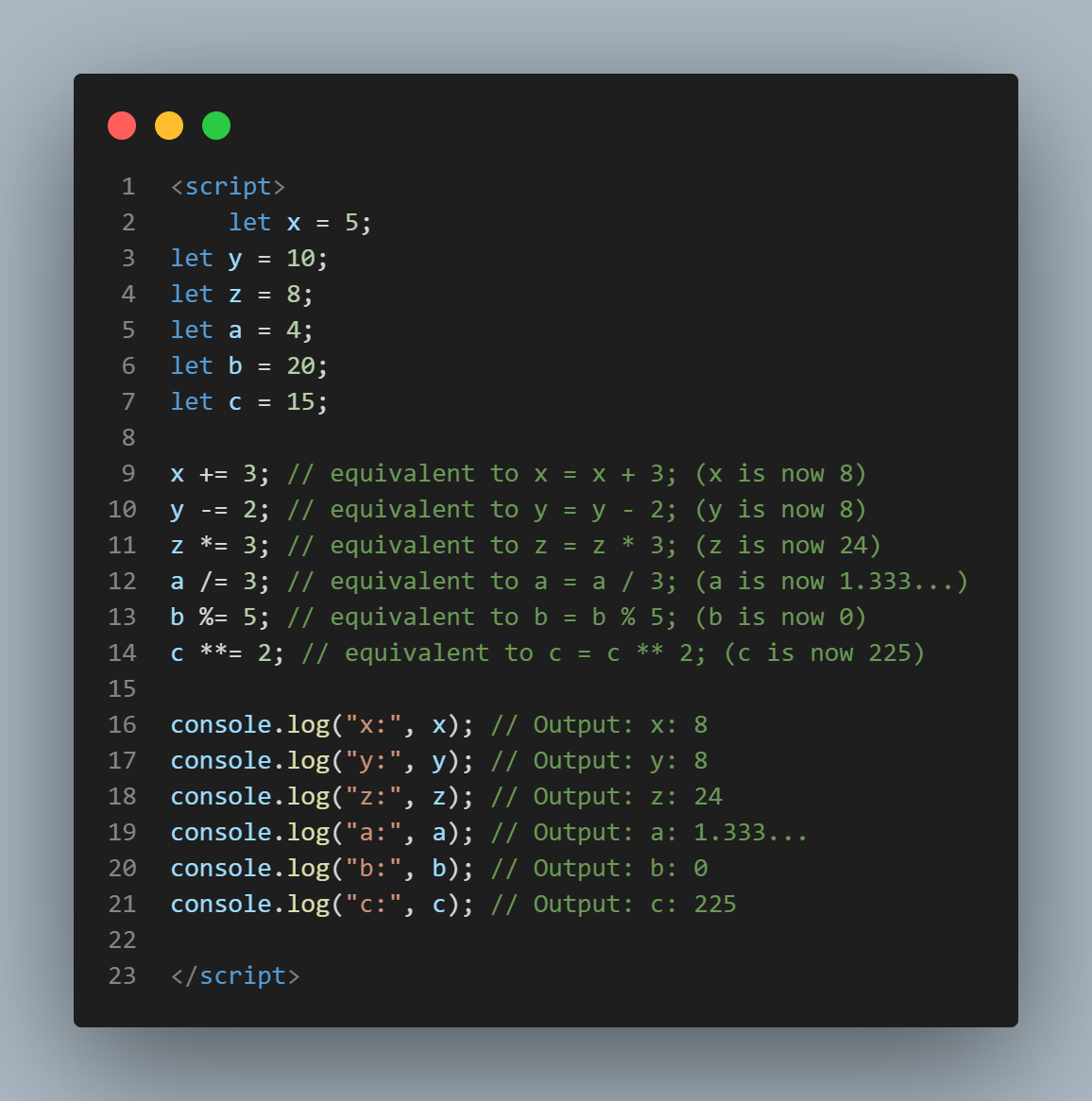
• Increment (++)

• Decrement (--)



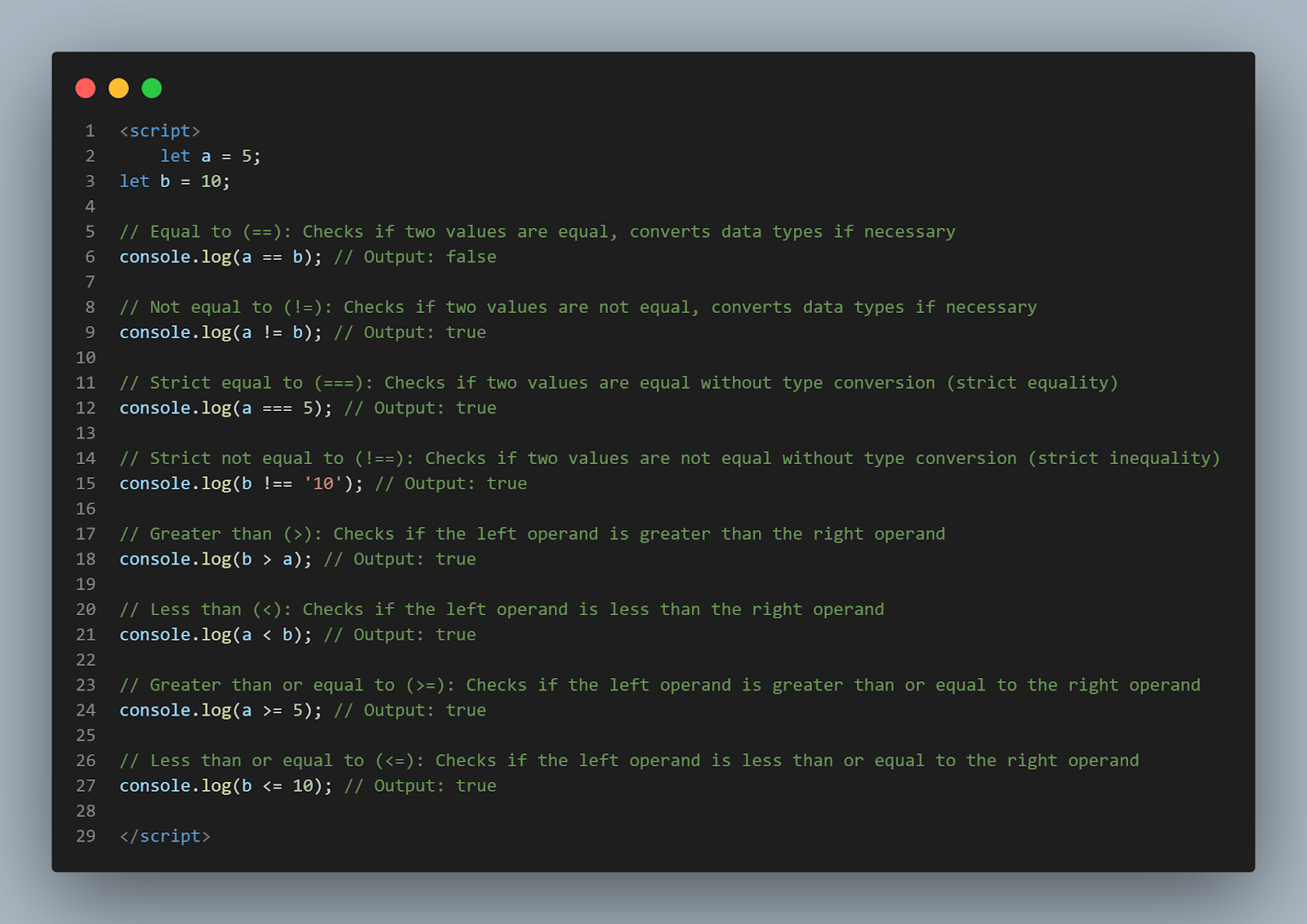


Assignment Operators:

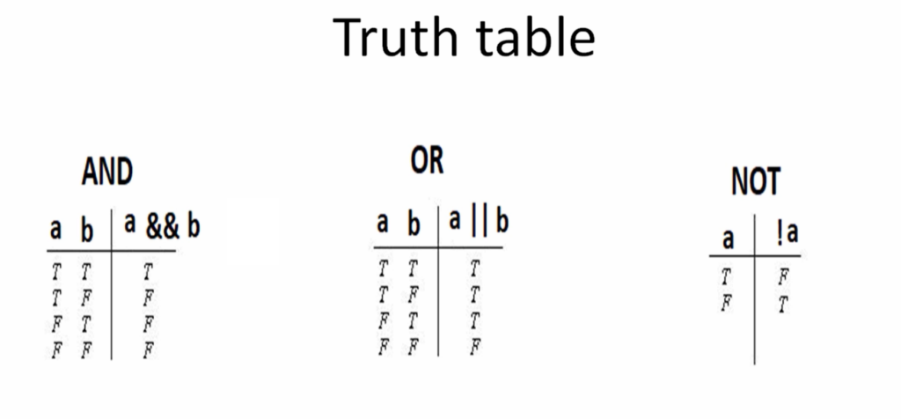
Assignment operators in JavaScript are used to assign values to variables. They combine the operation of assigning a value to a variable with some other operation, such as addition, subtraction, multiplication, division, or modulus.

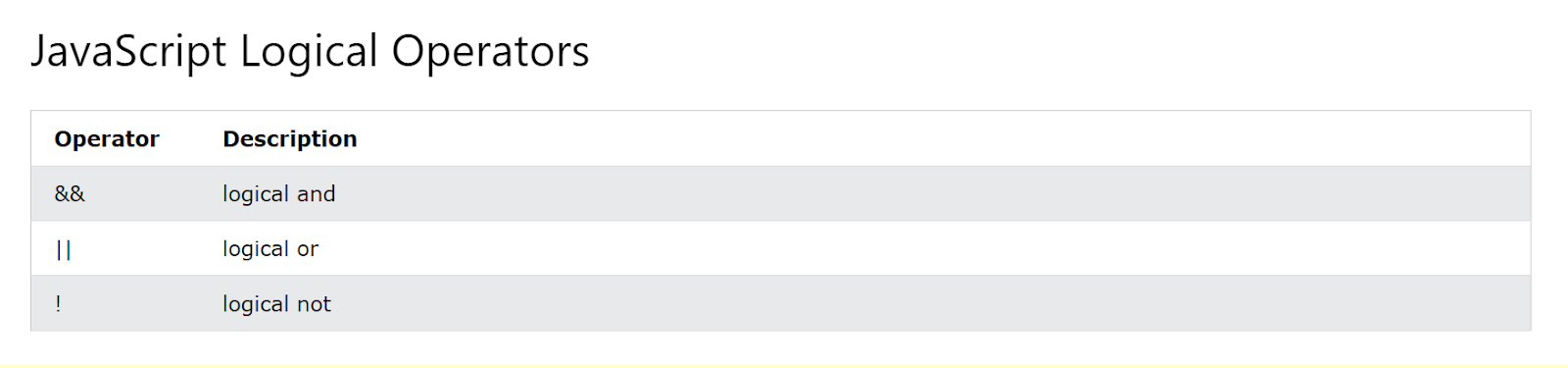
Comparison Operators

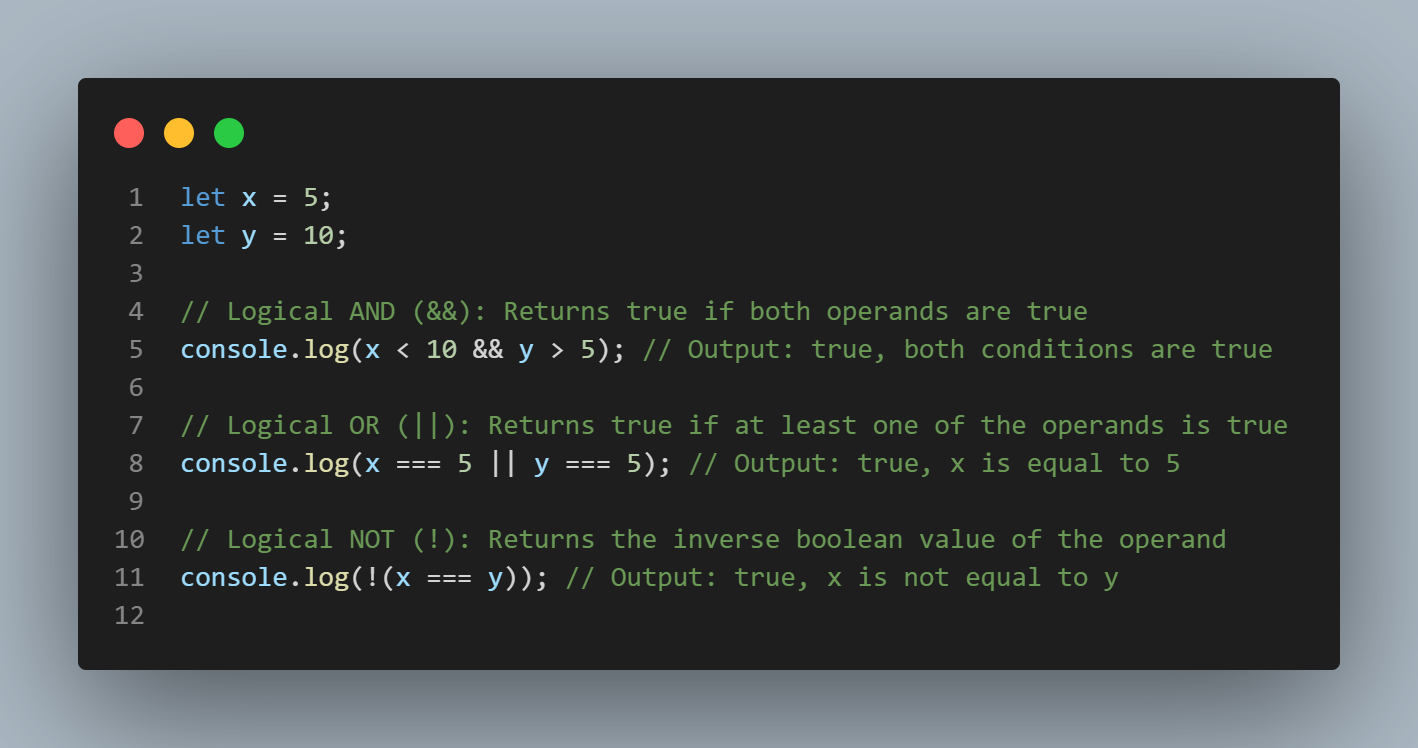
Comparison operators in JavaScript are used to compare two values and return a Boolean result



Logical operators:

  
Logical operators in JavaScript are used to combine or modify logical statements





In this snippet:

- `&&` (Logical AND) returns `true` if both conditions are true.

- `||` (Logical OR) returns `true` if at least one of the conditions is true.

- `!` (Logical NOT) returns `true` if the operand is false, and vice versa. It negates the boolean value of the operand.

string operators

In JavaScript, string operators are used to manipulate strings. The main string operator is the concatenation operator (`+`), which is used to concatenate or join two strings together. Here's an explanation of the string operator along with an example in one snippet:



In this snippet:

- The concatenation operator (`+`) is used to concatenate the `firstName` variable, a space, and the `lastName` variable, resulting in the full name "John Doe".

String operators are essential for manipulating and combining strings in JavaScript, enabling dynamic generation of textual content in web applications and scripts.

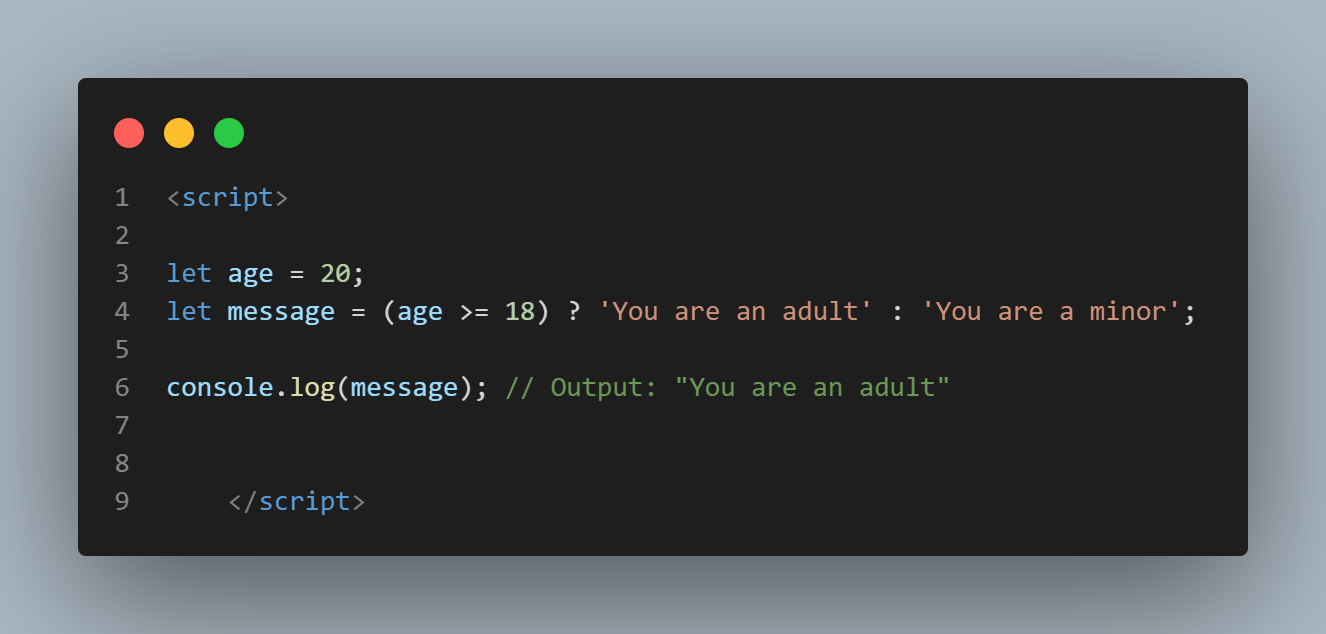
The ternary operator

also known as the conditional operator, provides a concise way to write conditional statements in JavaScript. Its syntax is:

condition ? expression1 : expression2

If the condition evaluates to true, the ternary operator returns `expression1`; otherwise, it returns `expression2`.

Here's an example demonstrating the ternary operator in action:



In this example:

- If the `age` variable is greater than or equal to 18, the condition `(age >= 18)` evaluates to true.

- Therefore, the ternary operator returns the value `'You are an adult'`, and this value is assigned to the `message` variable.

- If the condition is false, the ternary operator returns the value `'You are a minor'`.

The ternary operator is often used for assigning values to variables based on conditions, resulting in more concise and readable code compared to traditional if-else statements for simple conditions.

Type Operator:

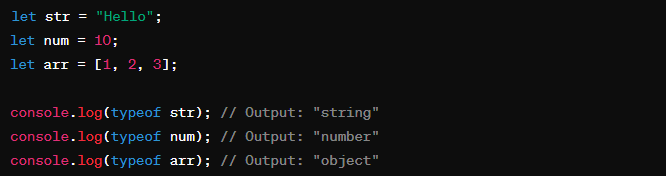
In JavaScript, there are two main type operators: `typeof` and `instanceof`. Let's explore each of them:

**1. typeof Operator:**

   - The `typeof` operator returns the data type of its operand.

   - It is often used to determine the type of a variable or expression.

   - The return value is a string indicating the type of the operand.

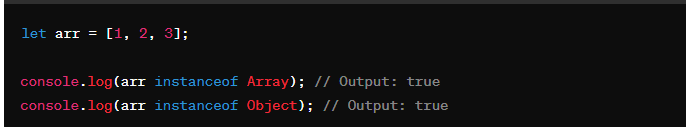


**2. instanceof Operator:**

   - The `instanceof` operator checks if an object is an instance of a specific object type.

   - It returns `true` if the object is an instance of the specified object type; otherwise, it returns `false`.

   - It is commonly used to check if an object is an instance of a particular class or constructor function.



   In this example, `arr` is an instance of both `Array` and `Object` because arrays are objects in JavaScript.

These type operators are useful for type checking and determining the nature of variables or objects in JavaScript code.

**CONTROL STRUCTURES:**

Control structures in JavaScript are fundamental building blocks that allow you to control the flow of your program's execution based on conditions or loops. Here's a breakdown of some common control structures with examples:

**1. Conditional Statements:**

These statements allow your code to make decisions and execute different blocks of code based on certain conditions.

* **if statement:** Executes code only if a specified condition is true.

if (temperature > 30) {

console.log ("It's hot outside!");

}

let age = 20;

if (age >= 18) {

console.log ("You are eligible to vote.");

} else {

console.log ("You are not eligible to vote.");

}

* **If-Else Statement**: Executes a block of code if a condition is true; otherwise, it executes another block of code.

if (temperature > 30) {

console.log ("It's hot outside!");

} else {

console.log ("It's not so hot outside.");

}

* **if-else if statement:** Allows checking multiple conditions and executing the corresponding block if a condition is true.

let grade = 85;

if (grade >= 90) {

console.log("Excellent!");

} else if (grade >= 80) {

console.log ("Very good!");

} else {

console.log ("Good effort, keep practicing!");

}

**Ternary Operator**

* A shorthand for the if-else statement.

let result = (age >= 18) ? "Adult" : "Minor";

console.log(result);

**switch statement:**

* Provides a multi-way branching mechanism based on the value of an expression.
* Selects one of many code blocks to be executed.

let day = "Tuesday";

switch (day) {

case "Monday":

console.log("Start of the week!");

break;

case "Tuesday":

case "Wednesday":

case "Thursday":

console.log("Midweek!");

break;

case "Friday":

console.log("TGIF!");

break;

default:

console.log("Weekend!");

}

**2. Loops:**

These statements allow you to repeat a block of code multiple times, either for a predetermined number of iterations or until a certain condition is met.

* **for loop:** Executes a code block repeatedly for a specific number of times, defined by a start, stop, and increment/decrement step.

for (let i = 1; i <= 5; i++) {

console.log("Iteration", i);

}

* **while loop:** Executes a code block repeatedly as long as a specified condition is true.

let count = 0;

while (count < 3) {

console.log("Count:", count);

count++;

}

* **do-while loop:** Similar to while, but the code block is guaranteed to execute at least once, even if the condition is initially false.
* Executes a block of code once, and then repeats the execution as long as a specified condition is true.

let i = 0;

do {

console.log("Number " + i);

i++;

} while (i < 5);

let message = "";

do {

message = prompt("Enter a message (or 'quit' to exit): ");

console.log("You entered:", message);

} while (message !== "quit");

**3. Break and Continue Statements:**

These statements are used to control the flow within loops.

* **break:** Exits a loop completely, even if the loop condition is still true.
* **continue:** Skips the current iteration of the loop and moves on to the next.

**Break Statement**

The break statement is used to exit a loop prematurely. It immediately terminates the loop and transfers control to the statement following the loop.

for (let i = 0; i < 10; i++) {

if (i === 3) {

break; // Exits the loop when i is 3

}

console.log ("The number is " + i);

}

// Output: The number is 0

// The number is 1

// The number is 2

**Continue Statement**

The continue statement is used to skip the current iteration of the loop and continue with the next iteration.

for (let i = 0; i < 10; i++) {

if (i === 3) {

continue; // Skips the current iteration when i is 3

}

console.log ("The number is " + i);

}

// Output: The number is 0

// The number is 1

// The number is 2

// The number is 4

// The number is 5

// ... up to 9

In the break example, the loop stops running entirely when i is 3. In the continue example, the loop skips the iteration where i is 3 but continues with the subsequent iterations. These statements are particularly useful for managing complex conditions within loops.

**Control structures in other programming languages**

**Control Structures**

**JavaScript**

* **Conditional Statements**: **if**, **else if**, **else**, **switch**
* **Loops**: **for**, **while**, **do while**, **for...of**, **for...in**

**Example**:

javascript

if (age > 18) {

console.log("Adult");

} else {

console.log("Minor");

}

for (let i = 0; i < 5; i++) {

console.log(i);

}

**Python**

* **Conditional Statements**: **if**, **else if**, **else**
* **Loops**: **for**, **while**

**Example**:

python

if age > 18:

print("Adult")

else:

print("Minor")

for i in range(5):

print(i)

**Java**

* **Conditional Statements**: **if**, **else if**, **else**, **switch**
* **Loops**: **for**, **while**, **do while**, **for-each**

**Example**:

java

if (age > 18) {

System.out.println("Adult");

} else {

System.out.println("Minor");

}

for (int i = 0; i < 5; i++) {

System.out.println(i);

}

**C++**

* **Conditional Statements**: **if**, **else if**, **else**, **switch**
* **Loops**: **for**, **while**, **do while**

**Example**:

C++

if (age > 18) {

std::cout << "Adult" << std::endl;

} else {

std::cout << "Minor" << std::endl;

}

for (int i = 0; i < 5; i++) {

std::cout << i << std::endl;

}

control structures

control structures are used to control the flow of execution in a program. Here are the main types of control structures

Control structures are fundamental elements in languages that enable the execution of specific code blocks based on certain conditions or criteria. They allow programmers to dictate the flow of execution within a program. Control structures can broadly be categorized into three types:

1. Sequential: Sequential control structures execute instructions in a sequence, one after the other, without any branching or looping. This is the default behavior of most programming languages unless explicitly modified.

2. Selection: Selection control structures, also known as conditional statements, execute specific blocks of code based on certain conditions. The most common types of selection structures are:

   - If-else statements: These allow the program to make decisions based on whether a given condition is true or false. If the condition is true, one block of code is executed; otherwise, another block is executed.

   - Switch statements (or case statements): These provide a way to execute different blocks of code based on the value of a variable or expression.

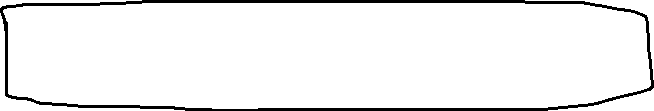
3. Iteration (or Repetition): Iteration control structures, also known as loops, allow a certain block of code to be executed repeatedly until a specific condition is met. The most common types of iteration structures are:

   - For loops: These execute a block of code a fixed number of times, iterating over a sequence of values (e.g., numbers from 1 to 10).

   - While loops: These execute a block of code as long as a specified condition is true. The condition is evaluated before each iteration, and if it's true, the loop continues; otherwise, it terminates.

   - Do-while loops: These are similar to while loops but with one crucial difference: the condition is evaluated after the execution of the loop's block of code. This ensures that the block of code is executed at least once, even if the condition is initially false.

Control structures are essential for creating programs that can perform complex tasks, respond dynamically to user input, and handle various scenarios efficiently. They provide the flexibility and logic necessary for writing functional and adaptable code.

a block is a set of statements enclosed within curly braces **{}**. Blocks are used to group zero or more statements together, forming a single compound statement. Blocks are commonly used in various contexts, including function bodies, loop bodies, conditional statements, and more

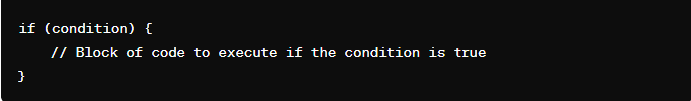
Selection control structures/ conditional statements:

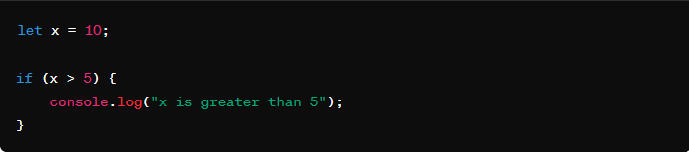
Conditional statements, also known as control structures, are fundamental constructs in programming languages that allow the execution of different blocks of code based on certain conditions or criteria. They enable the program to make decisions and perform different actions depending on the values of variables, user input, or the result of expressions.

Conditional statements provide the ability to create dynamic and flexible programs that can adapt their behavior at runtime. They are essential for implementing logic and decision-making processes within software applications.

There are several types of conditional statements commonly used in programming languages, including:

1. if statement: Executes a block of code if a specified condition is true.

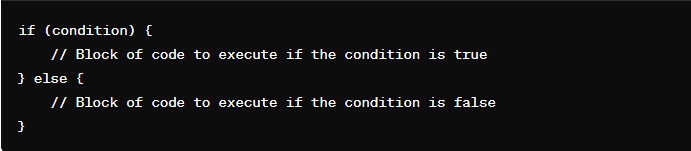




In this example, the condition x > 5 evaluates to true because the value of x is 10, which is indeed greater than 5. Therefore, the block of code inside the if statement (console.log) is executed, and it will output "x is greater than 5" to the console.

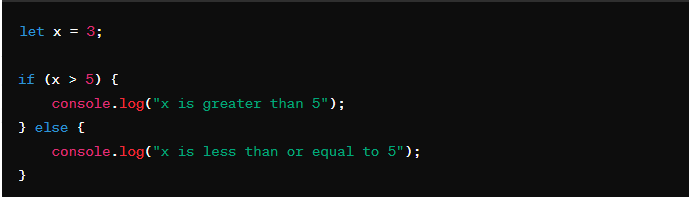
1. else statement: Executes a block of code if the condition of the preceding if statement is false.

else statement: The else statement is used to execute a block of code if the condition of the preceding if statement (or else if statement) is false. It is optional and provides an alternative block of code to execute when the condition of the preceding if statement evaluates to false



If the condition in the if statement evaluates to true, the block of code inside the first set of curly braces {} is executed, and the block of code inside the else statement is skipped.

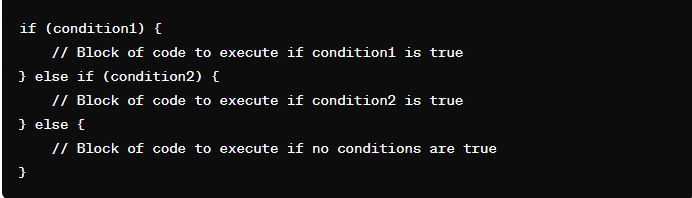
If the condition in the if statement evaluates to false, the block of code inside the else statement is executed, and the block of code inside the first set of curly braces {} is skipped.

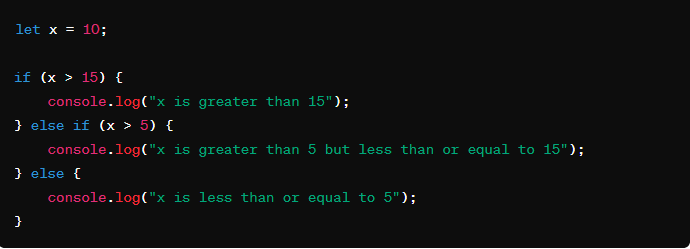


In this example, the condition x > 5 evaluates to false because the value of x is 3, which is less than 5. Therefore, the block of code inside the else statement is executed, and it will output "x is less than or equal to 5" to the console

1. else if statement: Provides an alternative condition to test if the preceding if condition is false. It can be used multiple times after an initial if statement.

else if statement: The else if statement provides an alternative condition to test if the preceding if condition is false. It allows you to check for additional conditions sequentially after the initial if statement. Each else if statement is evaluated only if the preceding if statement (or else if statement) evaluated to false.

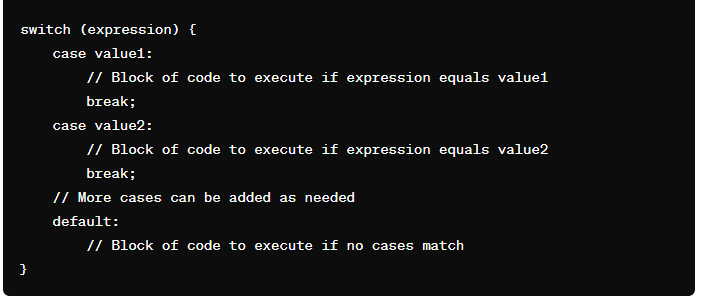




In this example, the value of x is 10. The first condition (x > 15) evaluates to false, so the program moves to the next else if statement. The second condition (x > 5) evaluates to true, so the corresponding block of code is executed, and it will output "x is greater than 5 but less than or equal to 15" to the console.

1. switch statement: Evaluates an expression and executes different blocks of code based on matching cases. It provides an alternative to multiple else if statements.

switch statement: The switch statement evaluates an expression and executes different blocks of code based on matching cases. It provides an efficient way to handle multiple possible outcomes of a single expression.



expression is the expression or variable whose value is to be compared with the cases.

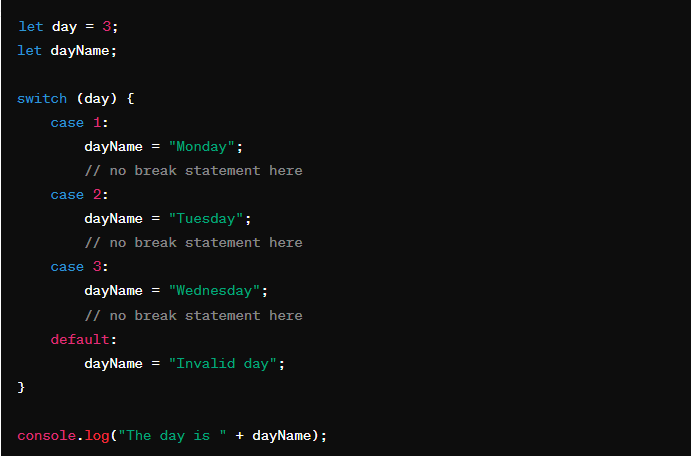
case value1, case value2, etc., represent the different possible values of the expression.

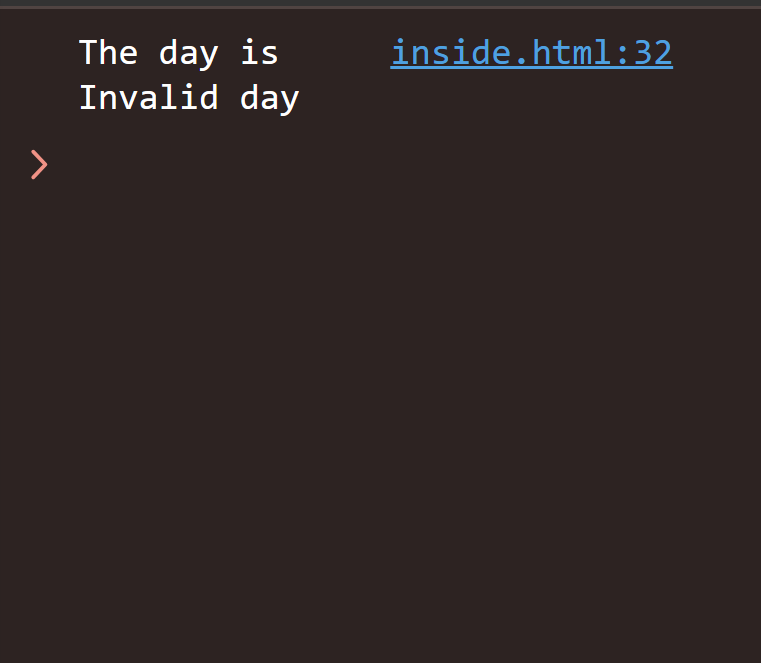
When the switch statement is executed, the expression is evaluated, and control jumps to the case that matches the value of the expression.

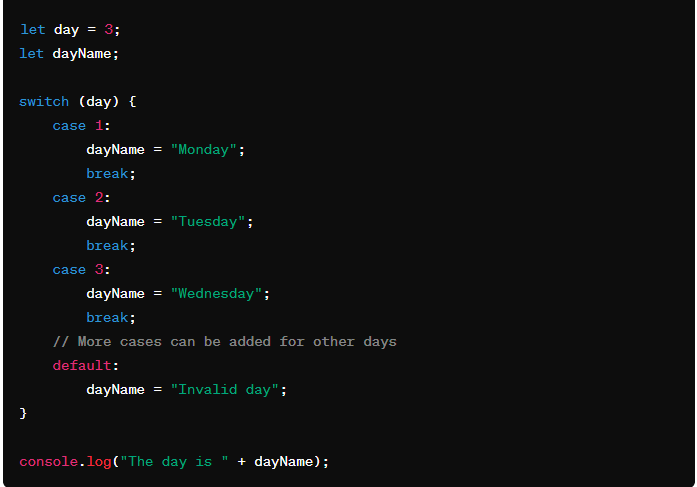
If no matching case is found, the default block (if provided) is executed. The default block is optional.

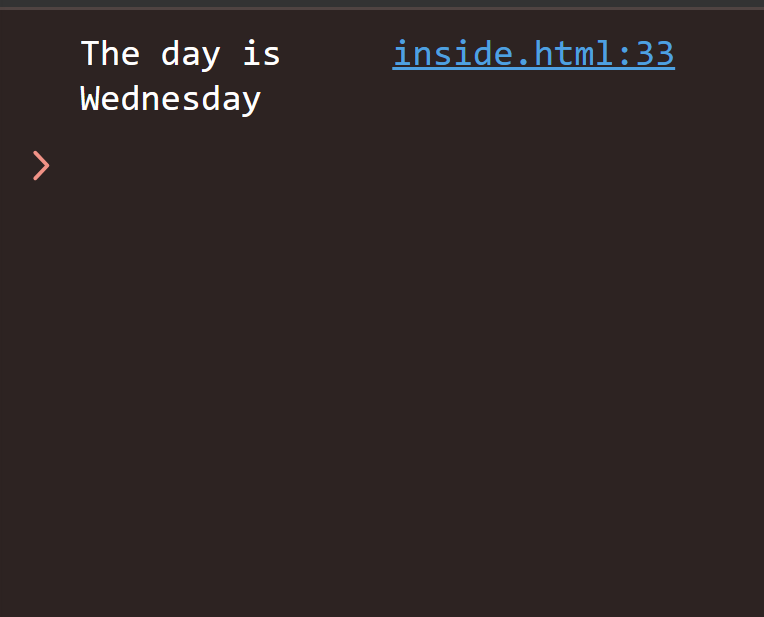
When break is encountered inside a switch statement, it exits the switch block.

Without the break statement, control would fall through to the next case, executing the code for that case regardless of whether its condition is met.









In this example, the value of day is 3. The switch statement evaluates the value of day and executes the corresponding block of code based on the matching case. Since day is 3, it matches the case 3, and the dayName is set to "Wednesday". Finally, it logs "The day is Wednesday" to the console.

5. Ternary operator: A concise way to write conditional statements. It takes three operands and evaluates a Boolean expression, returning one of two expressions depending on whether the expression is true or false.

These conditional statements form the backbone of decision-making in programming, allowing developers to create programs that respond dynamically to various scenarios and conditions. They are essential for creating robust and versatile software applications.

loops

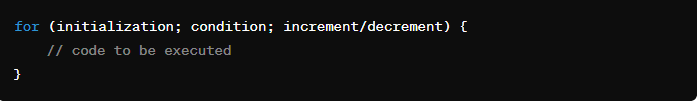
In JavaScript, loops are control structures that allow you to execute a block of code repeatedly as long as a specified condition is true. There are several types of loops available in JavaScript:

In JavaScript, loops are control structures that allow you to execute a block of code repeatedly as long as a specified condition is true. There are several types of loops available in JavaScript:

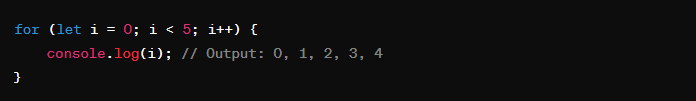
1. for Loop:

   - The for loop repeats a block of code a specified number of times.

   - Syntax:



   - Example:



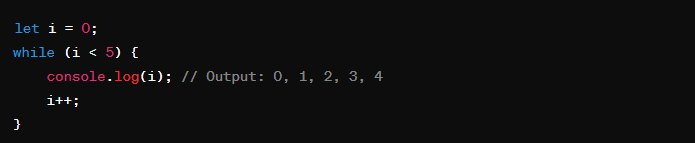
2. while Loop:

   - The while loop repeats a block of code as long as a specified condition is true.

   - Syntax:



   - Example:



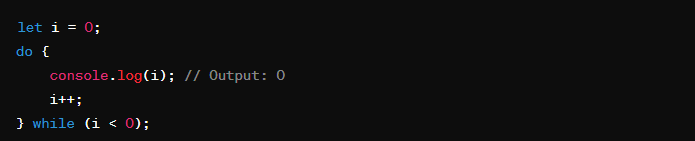
3. do...while Loop:

   - The do...while loop is similar to the while loop, but it always executes the block of code at least once before checking the condition.

   - Syntax:



   - Example:



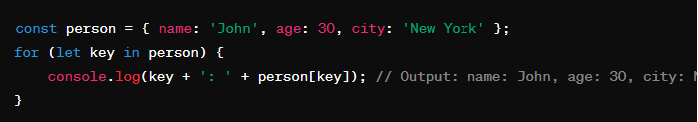
4. for...in Loop:

   - The for...in loop iterates over the enumerable properties of an object.

   - Syntax:



       - Example:



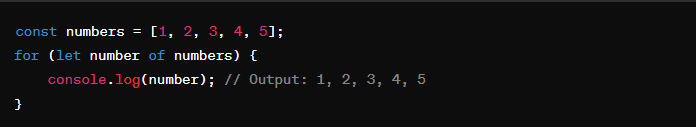
5. for...of Loop:

   - The for...of loop iterates over iterable objects such as arrays, strings, maps, sets, etc.

   - Syntax:



   - Example:



These are the main types of loops available in JavaScript, each serving different purposes depending on the requirements of your code.

**Functions:**

In JavaScript, **functions** are blocks of code designed to perform a particular task, and they are executed when something invokes (calls) them. A **method** is essentially a function associated with an object.

Here are some examples of functions and methods in JavaScript:

### Function Declaration

A function can be declared using the function keyword followed by a name, followed by parentheses ().

// Function declaration

function functionName(){

// Code to be executed

return value; // Optional return statement

}

function greet() {

console.log("Hello World!");

}

// Calling the function

greet();

// Output: Hello World!

### Function Expression

A function can also be defined using a function expression by storing a function in a variable.

// Function expression

let displayPI = function() {

console.log("PI = 3.14");

};

// Calling the function

displayPI();

// Output: PI = 3.14

function greet(name) {

console.log("Hello, " + name + "!");

return "Welcome!"; // Function can return a value

}

let greeting = greet("Alice"); // Call the function and store the return value

console.log(greeting);

// Outputs: "Welcome!"

**Function Calls/invocation:**

* To execute a function, you call it by its name followed by parentheses ().
* Any arguments (values) you provide within the parentheses are passed to the function's parameters.
* The function's code is then executed, and the returned value (if any) is assigned to a variable or used directly.

**Parameters and Arguments:**

* Parameters are placeholders defined within the function's parentheses that act as variables to receive the values passed during the function call. These values are called arguments.
* You can have zero or more parameters in a function.
* The number of arguments provided during the function call must match the number of parameters defined in the function.

**Example with Parameters:**

function functionName(parameter1, parameter2, ...) {

// Code to be executed

return value; // Optional return statement

}

function calculateArea(length, width) {

let area = length \* width;

return area;

}

let rectangleArea = calculateArea(5, 3); // Pass arguments (actual values)

console.log(rectangleArea);

// Outputs: 15 (calculated area)

**Scope:**

* Variables declared within a function (local variables) are only accessible within that function's scope. They are not accessible from outside the function.
* Function parameters also have local scope, meaning they are only accessible within the function.

**Example:**

function printMessage() {

let message = "This message is local"; // Local variable

console.log(message);

}

printMessage(); // Outputs: "This message is local"

console.log(message); // ReferenceError: message is not defined (outside function scope)

### Arrow Function

ES6 introduced arrow functions, which provide a shorter syntax for writing functions.

// Arrow function

// Arrow function

var greet = (name) => {

console.log("Hello, " + name + "!");

};

const add = (a, b) => a + b;

// Calling the function

console.log(add(3, 2));

// Output: 5

**Function Invocation:**

Once a function is declared, you can invoke (call) it by using its name followed by parentheses (). You can pass arguments to the function inside the parentheses.

// Function invocation

greet("John");

Return Statement:

Functions can optionally return a value using the return statement. This value can then be used by the calling code.

function add(a, b) {

return a + b;

}

var result = add(5, 3);

console.log(result);

// Output: 8

**Function Parameters:**

Functions can accept parameters (inputs) that are specified when the function is called. These parameters act as variables within the function.

function greet(name) {

console.log("Hello, " + name + "!");

}

greet("Alice"); // Output: Hello, Alice!

Default Parameters (ES6 and later):

You can specify default values for function parameters.

function greet(name = "World") {

console.log("Hello, " + name + "!");

}

greet(); // Output: Hello, World!

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

**Anonymous Functions:**

Functions without a name are called anonymous functions. They are typically used as callbacks or immediately invoked function expressions (IIFE).

// Anonymous function

var greet = function(name) {

console.log("Hello, " + name + "!");

};

**Functions in other programming languages**

**Functions**

**JavaScript**

* **Function Declaration**:

javascript

function add(a, b) {

return a + b; }

console.log(add(2, 3)); // Outputs: 5

* **Function Expression**:

javascript

const subtract = function(a, b) {

return a - b;

};

console.log(subtract(5, 2)); // Outputs: 3

* **Arrow Function**:

javascript

const multiply = (a, b) => a \* b;

console.log(multiply(2, 4)); // Outputs: 8

**Python**

* **Function Definition**:

python

def add(a, b):

return a + b

print(add(2, 3)) # Outputs: 5

**Java**

* **Function Definition**:

java

public int add(int a, int b) {

return a + b; }

// Usage:

int result = add(2, 3);

System.out.println(result); // Outputs: 5

**C++**

* **Function Definition**:

C++

int add(int a, int b) { return a + b; } // Usage: int result = add(2, 3); std::cout << result << std::endl; // Outputs: 5

**Higher-Order Functions**

**JavaScript**

**Example**:

javascript

const numbers = [1, 2, 3, 4, 5];

const squared = numbers.map(x => x \* x);

console.log(squared); // Outputs: [1, 4, 9, 16, 25]

**Python**

**Example**:

python

numbers = [1, 2, 3, 4, 5]

squared = list(map(lambda x: x\*x, numbers))

print(squared) # Outputs: [1, 4, 9, 16, 25]

**Java**

**Example (using streams)**:

java

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

List<Integer> squared = numbers.stream().map(x -> x \* x).collect(Collectors.toList());

System.out.println(squared); // Outputs: [1, 4, 9, 16, 25]

**C++**

**Example (using standard algorithms)**:

C++

std::vector<int> numbers = {1, 2, 3, 4, 5};

std::vector<int> squared;

std::transform(numbers.begin(), numbers.end(), std::back\_inserter(squared), [](int x) { return x \* x; });

for (int n : squared) {

std::cout << n << " "; }

// Outputs: 1 4 9 16 25

**Recursive Functions**

**JavaScript**

**Example**:

javascript

function factorial(n) {

if (n <= 1) {

return 1;

} else {

return n \* factorial(n - 1);

} }

console.log(factorial(5)); // Outputs: 120

**Python**

**Example**:

python

def factorial(n):

if n <= 1:

return 1

else:

return n \* factorial(n - 1)

print(factorial(5)) # Outputs: 120

**Java**

**Example**:

java

public int factorial(int n) {

if (n <= 1) {

return 1;

} else {

return n \* factorial(n - 1);

}

}

// Usage:

int result = factorial(5);

System.out.println(result); // Outputs: 120

**C++**

**Example**:

C++

int factorial(int n) {

if (n <= 1) {

return 1;

} else {

return n \* factorial(n - 1);

}

}

// Usage:

int result = factorial(5);

std::cout << result << std::endl; // Outputs: 120

**Immediately Invoked Function Expressions (IIFE)**

**JavaScript**

**Example**:

javascript

(function() {

console.log('IIFE executed');

})();

**Python**

* Python does not have a direct equivalent to IIFE, but you can achieve similar behavior with functions:

python

(lambda: print('IIFE executed'))()

**Java & C++**

* Java and C++ do not support IIFE directly, but you can use static blocks or inline code execution within scopes for similar effects.

**Generator Functions (Introduced in ES6 for JavaScript)**

**JavaScript**

**Example**:

javascript

function\* generateSequence(start, end) {

for (let i = start; i <= end; i++) {

yield i;

}

const generator = generateSequence(1, 5);

console.log([...generator]); // Outputs: [1, 2, 3, 4, 5]

**Python**

**Example**:

python

def generate\_sequence(start, end):

for i in range(start, end + 1):

yield i

generator = generate\_sequence(1, 5)

print(list(generator)) # Outputs: [1, 2, 3, 4, 5]

**Java & C++**

* Java and C++ do not have built-in support for generator functions, but similar behavior can be achieved using iterators and custom iterable classes.

### Methods

Methods are functions that are stored as object properties.

// Object with a method

let person = {

name: 'Alice',

greet: function() {

console.log("Hello, " + this.name);

}

};

// Calling the method

person.greet(); // Output: Hello, Alice

### Built-in Methods:

JavaScript has many built-in methods, such as toUpperCase() for strings.

// Using a built-in method

let message = 'hello';

console.log(message.toUpperCase()); // Output: HELLO

Methods are similar to functions, but they are associated with objects. They are used to perform operations on that object's data.

Methods are defined within the object definition using a key-value pair structure, where the key is the method name and the value is the function definition.

Syntax:

let objectName = {

methodName: function(parameter1, parameter2, ...) {

// Code to be executed with access to object properties

return value;

},

// Other properties and methods

};

Functions and methods are fundamental concepts in JavaScript, allowing you to encapsulate code for reuse, which leads to more modular and maintainable code.

**1. Function Definition:**

**a. Function Declaration:**

A function declaration defines a named function. This type of function is hoisted, meaning it can be called before it is defined in the code.

function add(a, b) {

return a + b; }

console.log(add(2, 3));

// Outputs: 5

**b. Function Expression:**

A function expression defines a function and assigns it to a variable. This function is not hoisted.

const subtract = function(a, b) {

return a - b; };

console.log(subtract(5, 2)); // Outputs: 3

**c. Arrow Function (Introduced in ES6):**

Arrow functions offer a shorter syntax and lexically bind **this** value.

const multiply = (a, b) => a \* b;

console.log(multiply(2, 4)); // Outputs: 8

**2. Types of Functions:**

**a. Named Functions:**

Named functions are useful for recursion and when you want to reference the function by name.

function greet(name) {

return `Hello, ${name}!`; }

console.log(greet('Alice')); // Outputs: Hello, Alice!

**b. Anonymous Functions:**

Anonymous functions are often used as callbacks or in IIFEs.

const sayHello = function(name) {

return `Hello, ${name}!`; };

console.log(sayHello('Bob')); // Outputs: Hello, Bob!

**c. Arrow Functions:**

Arrow functions are best for concise code and do not have their own **this** context.

const double = (x) => x \* 2;

console.log(double(3)); // Outputs: 6

**d. Higher-Order Functions:**

Higher-order functions take other functions as arguments or return functions.

const numbers = [1, 2, 3, 4, 5];

const squared = numbers.map(x => x \* x);

console.log(squared); // Outputs: [1, 4, 9, 16, 25]

**e. Recursive Functions:**

Recursive functions call themselves to solve problems that can be divided into similar subproblems.

function factorial(n) { if (n <= 1) {

return 1; } else { return n \* factorial(n - 1); } }

console.log(factorial(5)); // Outputs: 120

**f. Immediately Invoked Function Expressions (IIFE):**

IIFEs run as soon as they are defined and are used to create private scopes.

(function() {

console.log('IIFE executed'); })();

**g. Generator Functions (Introduced in ES6):**

Generator functions can pause and resume execution, useful for handling asynchronous flows.

function\* generateSequence(start, end) {

for (let i = start; i <= end; i++) { yield i; } }

const generator = generateSequence(1, 5);

console.log([...generator]); // Outputs: [1, 2, 3, 4, 5]

**h. Method Functions:**

Method functions are functions defined inside an object and can access the object's properties using **this**.

const person = { name: 'Alice', greet: function() { return `Hello, ${this.name}!`; } }; console.log(person.greet()); // Outputs: Hello, Alice!

Understanding these different types of functions in JavaScript allows developers to choose the appropriate function type for their specific needs, making their code more readable, maintainable, and efficient.

**Method Functions**

**JavaScript**

**Example**:

javascript

const person = {

name: 'Alice',

greet: function() {

return `Hello, ${this.name}!`;

}

};

console.log(person.greet()); // Outputs: Hello, Alice!

**Python**

**Example**:

python

class Person:

def \_\_init\_\_(self, name):

self.name = name

def greet(self):

return f"Hello, {self.name}!"

person = Person("Alice")

print(person.greet()) # Outputs: Hello, Alice!

**Java**

**Example**:

java

public class Person {

private String name;

public Person(String name) {

this.name = name;

}

public String greet() {

return "Hello, " + this.name + "!";

}

}

// Usage:

Person person = new Person("Alice");

System.out.println(person.greet()); // Outputs: Hello, Alice!

**C++**

**Example**:

C++

class Person {

private:

std::string name;

public:

Person(std::string name) : name(name) {}

std::string greet() {

return "Hello, " + name + "!";

}

};

// Usage:

Person person("Alice");

std::cout << person.greet() << std::endl; // Outputs: Hello, Alice!