# JavaScript Fundamentals

# Engage and Think

Imagine you are designing a dynamic e-commerce website where users can browse products, add items to their cart, and complete purchases seamlessly. To make the website interactive and responsive, you need to ensure that actions like showing product details on hover, updating the cart instantly, and validating user inputs in forms happen smoothly without refreshing the page.

As you plan the development, you realize that the website needs a way to respond to user actions in real time and provide a smooth user experience.

How do you think a website can dynamically update content, handle user interactions, and improve responsiveness without requiring constant page reloads?

# Learning Objectives

By the end of this lesson, you will be able to:

- Apply JavaScript fundamentals by using core concepts such as variables, data types, and operators to build a strong programming foundation
- Utilize control flow and loops by applying conditional statements and iterations to create dynamic and interactive programs
- Apply data types and variables by declaring, assigning, and handling different types of data in JavaScript
- Make use of arithmetic and logical operations to perform calculations and support decision-making processes in a JavaScript program
- Implement basic debugging techniques to identify and fix common errors in simple JavaScript code

# Introduction to JavaScript

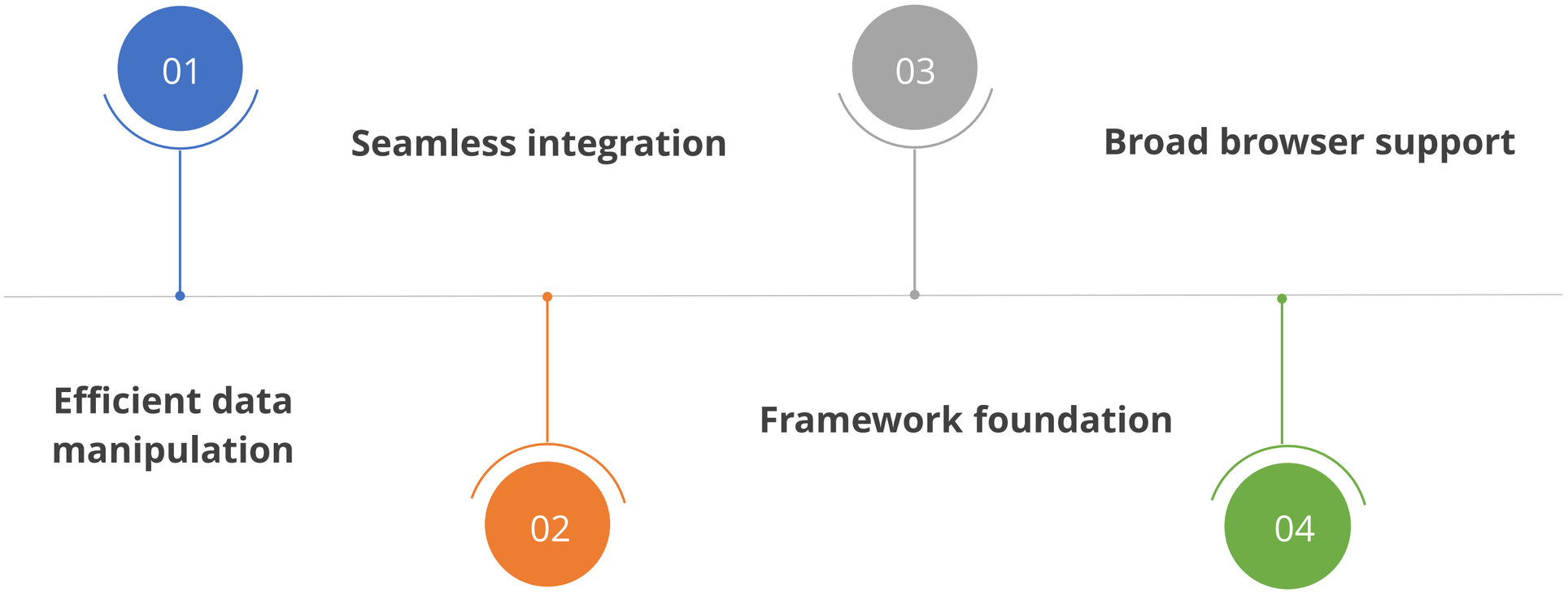
# What Is JavaScript?

JavaScript is a lightweight scripting programming language used to make webpages interactive. It can also calculate, validate, and manipulate the data.


- It can insert dynamic text into HTML and CSS and make the webpage interactive.
- It can be used in front-end and back-end web development.

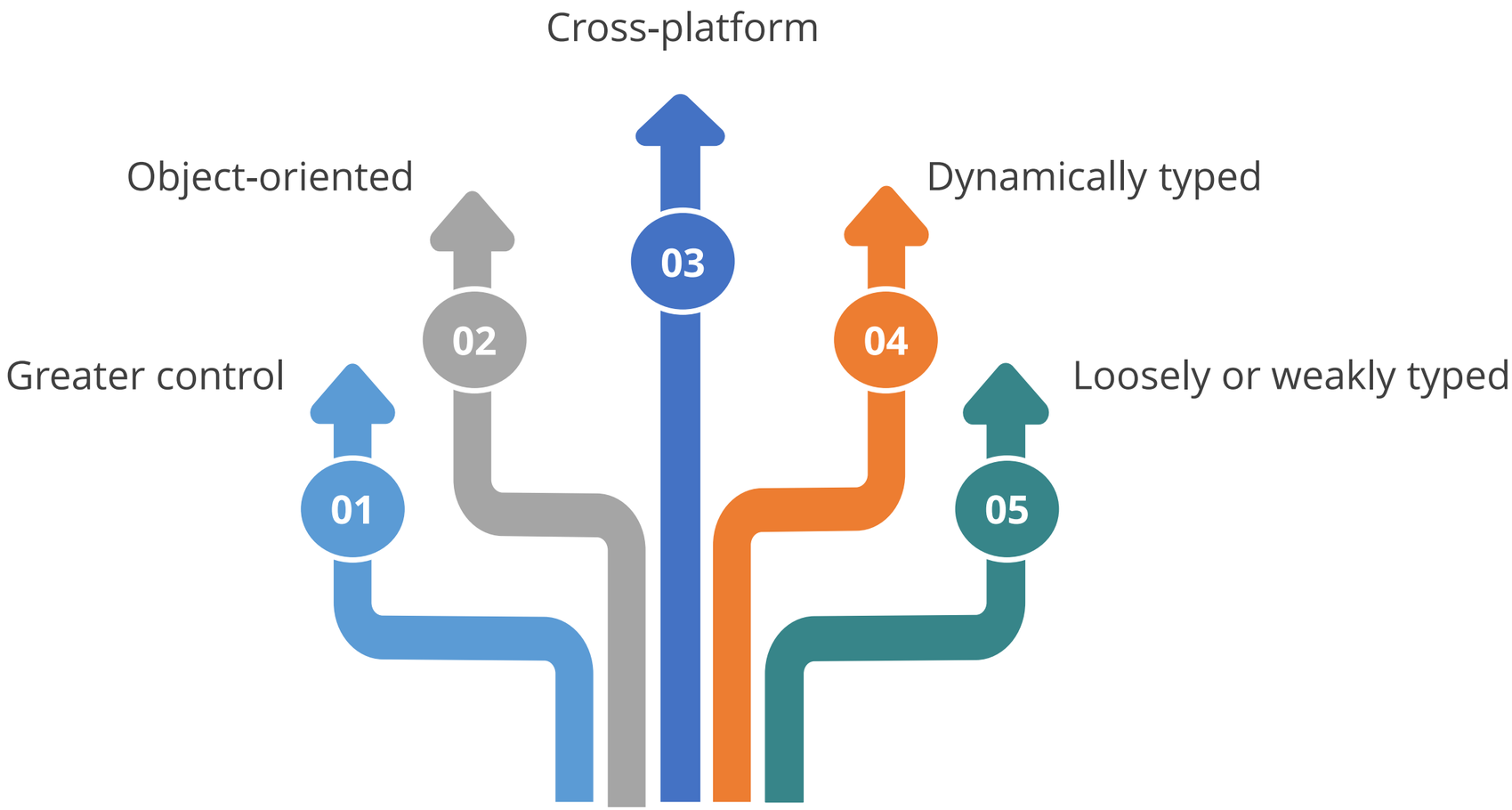
# JavaScript: Features

The most important features of JavaScript are:



# Benefits of JavaScript

It offers the following advantages:



# Application of JavaScript

01 Mobile app creation

02 Game development

03 Web development

04 Smart watch application


05 Server-side development

# Quick Check


Imagine you are the product manager for a music streaming application aiming to improve user satisfaction. In this context, which type of personalization would be crucial for adjusting the application's operational behavior and functionality to align with individual user habits and preferences?

A. Content personalization

B. Aesthetic personalization

C. Interaction personalization

D. Visual personalization

# JavaScript and Document Object Model (DOM)

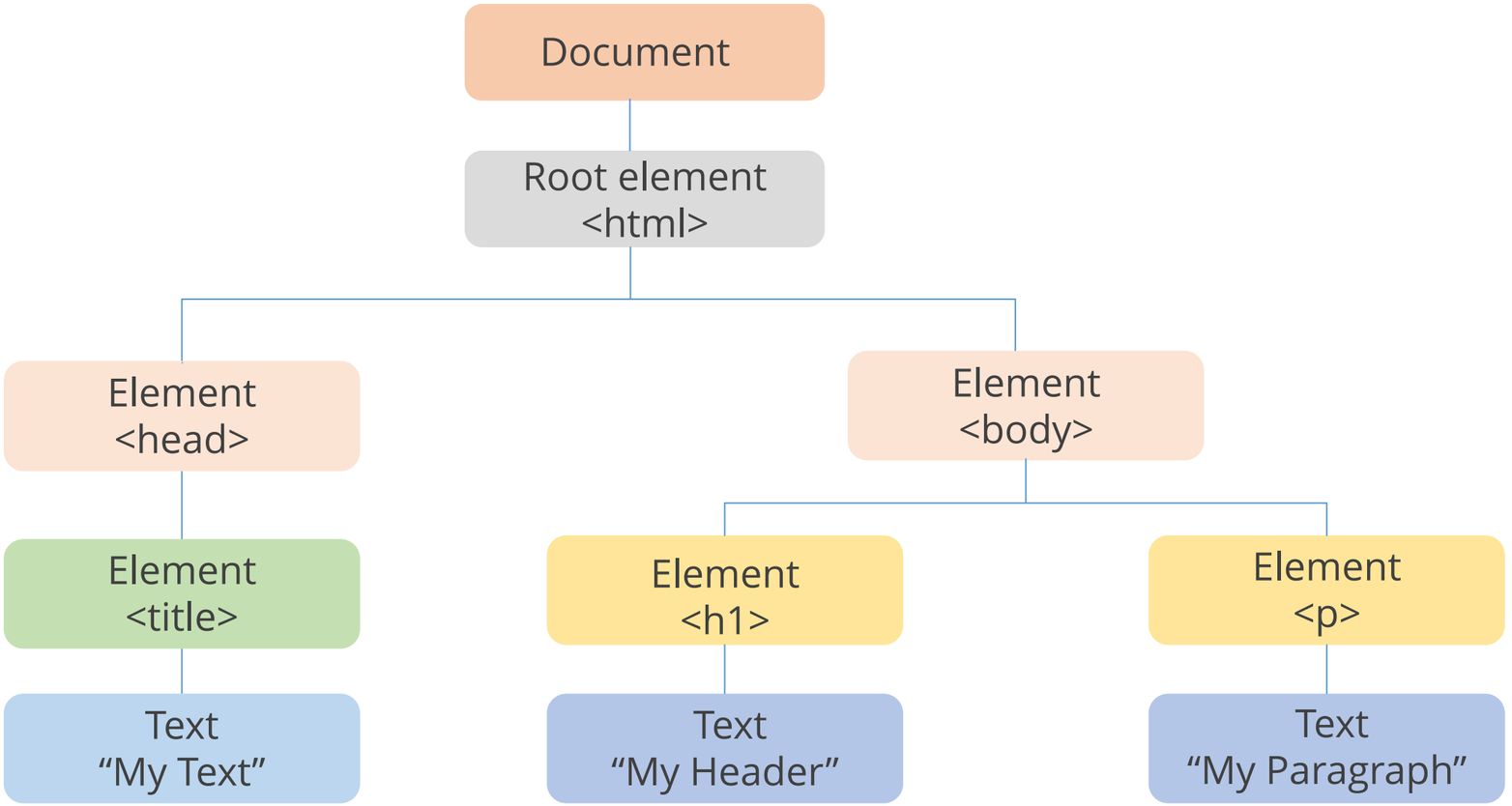
# What Is DOM?

The Document Object Model (DOM) is a web document interface that structures HTML or XML for dynamic access and manipulation using JavaScript.

Nodes and objects are used to represent the document in the DOM.

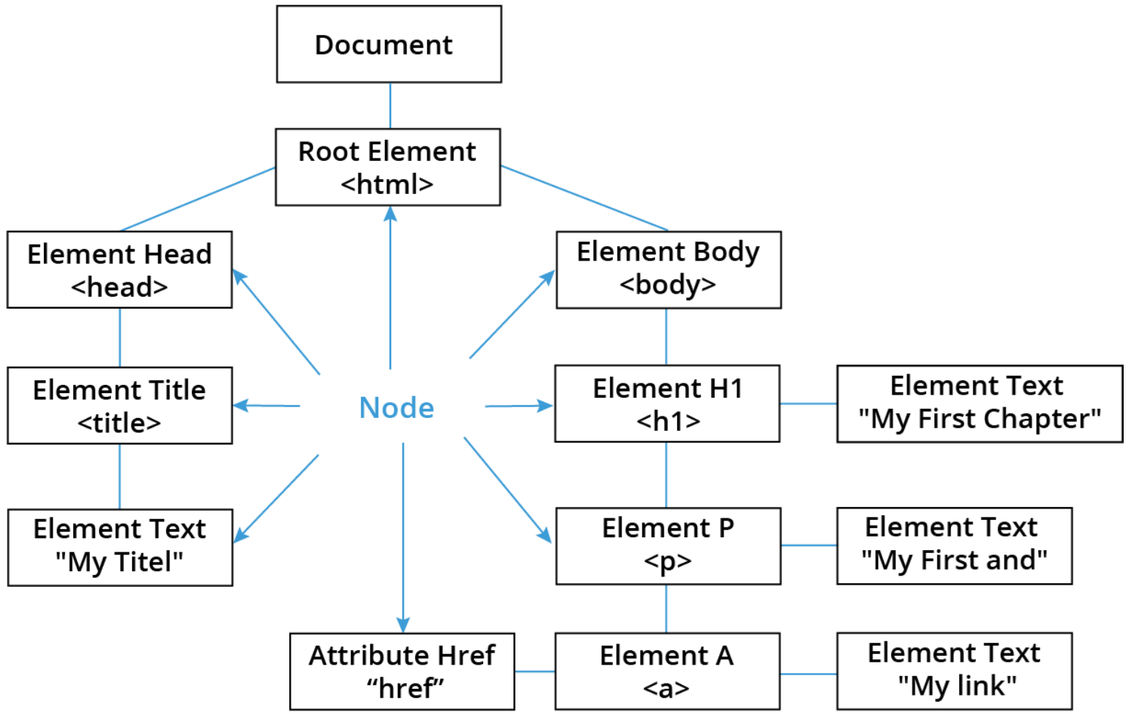
# What Is a DOM Model?

DOM model defines the structural representation of the document, as shown below:



# DOM Nodes

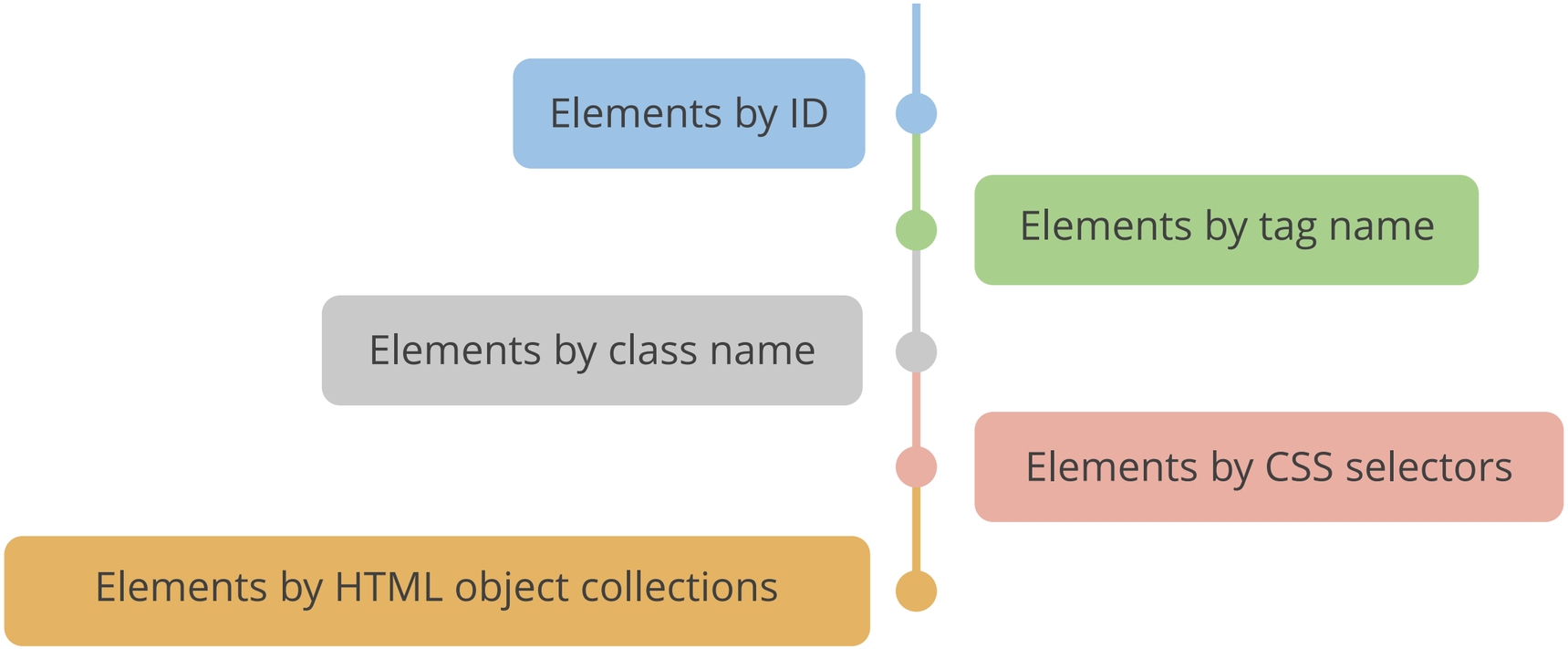
All elements, attributes, text, and other aspects of the page are grouped in a hierarchical tree-like structure in the DOM.




Nodes are the individual components of a document.

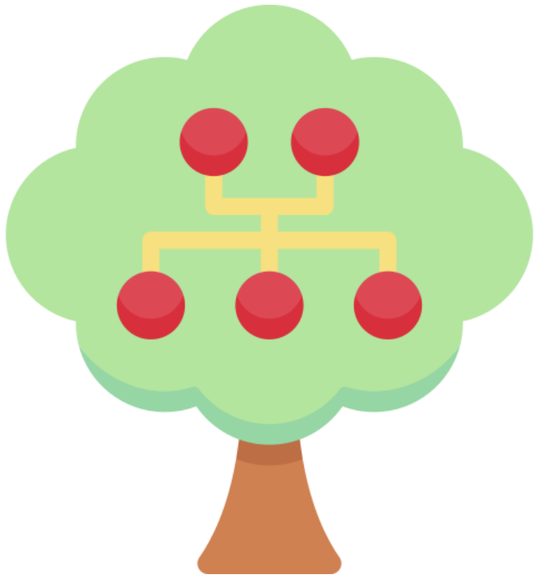
# DOM Elements

It includes DIV, HTML, and BODY elements. Users can access and manipulate DOM elements using any of the following methods:



# DOM and JavaScript

The DOM is not a programming language, but JavaScript uses it to interact with and manipulate web pages (HTML documents) and Scalable vector graphics (SVG) documents.




JavaScript and the DOM work together to create dynamic, interactive, and user-friendly web experiences.

# What Are Methods in DOM?

DOM methods are functions that allow JavaScript to manipulate and interact with HTML and XML documents. These methods help in selecting, modifying, adding, and removing elements dynamically.

These methods provide powerful ways to interact with web pages, making them dynamic and interactive. Mastering these methods enables developers to manipulate HTML and CSS efficiently.

# Methods of DOM

Some important methods of document object are:

| Method | Description |
| --- | --- |
| **write("string")** | Writes the string to the document |
| **writeln("string")** | Writes the string to the document and adds a newline at the end |
| **getElementById("string")** | Returns the value of the element with the specified ID |

# Methods of DOM

Some important methods of document object are:

| Method | Description |
| --- | --- |
| **getElementsByName()** | Returns all elements with the specified name value |
| **getElementsByTagName()** | Returns all elements with the specified tag name |
| **RegExp** | Represents a regular expression |

# DOM Programming Interface

The HTML Document Object Model (DOM) controls elements within a web page. Below are the key aspects of how JavaScript interacts with the DOM to manipulate and modify web content dynamically:

**01** JavaScript interacts with the HTML Document Object Model (DOM) to manipulate web content.

**02** In the DOM, all HTML elements are represented as objects that can be modified using JavaScript.

**03** An object's properties and methods define its programming interface in the DOM.

# DOM Programming Interface

Below are the key aspects of how JavaScript interacts with the DOM to manipulate and modify web content dynamically:

**04** A property holds a value that can be retrieved or modified, such as an HTML element's content.

**05** A method performs actions on elements, such as adding or deleting them.

Now, let us explore how JavaScript interacts with HTML elements using the innerHTML property to modify content dynamically.

# JavaScript and HTML

The innerHTML property allows JavaScript to modify an element's content. While it's simple and effective, avoid using it with user-generated content to prevent security risks.

**Example:**

```
<html>
<body>
<p id="p1">Hello World!</p>
<script>
document.getElementById("p1").innerHTML = "New text!";
</script>
</body>
</html>
```

# JavaScript and HTML

JavaScript can generate dynamic HTML content by modifying an element's properties.

In this example, we display the current date dynamically:

**Example:**

```
<!DOCTYPE html>
<html>
<body>
<script>
document.getElementById("demo").innerHTML = "Date : " + Date();
</script>
</body>
</html>
```

# Quick Check


You are developing a web page that allows users to update their profile information dynamically. When a user submits the updated details, the new information should instantly appear on the screen without reloading the page. Which approach best utilizes the DOM Programming Interface to achieve this?

A. Update HTML elements using JavaScript

B. Reload the page to show updates

C. Redirect to a new HTML file

D. Change CSS to display new data

# Exploring Variables and Data Types

# What Are Variables?

Variables are the names users give to the memory locations in a computer program where values are stored.

There are two types of variables in JavaScript:

1. **Local variable**
2. **Global variable**

# Types of Variables

### Local variable

A local variable is declared inside a block or function. It is accessible within the function or block only.


### Global variable

A global variable is accessible from any function. A variable declared outside the function or declared with a window object.

Variables are declared using the **var** or **let** keyword, and they can be reassigned new values.

# Variables: Example

**Example:**

```
// Variable declared with 'var'
var varVariable = 42;
console.log("Original varVariable:", varVariable); // Output: 42

// 'var' allows reassignment
varVariable = 20;
console.log("Reassigned varVariable:", varVariable); // Output: 20

// Variable declared with 'let'
let letVariable = "Hello";
console.log("Original letVariable:", letVariable); // Output: Hello

// 'let' allows reassignment
letVariable = "World";
console.log("Reassigned letVariable:", letVariable); // Output: World
```

The above code demonstrates how variables declared with *var* and *let* can be reassigned in JavaScript, showing their mutability and behavior.

# What Are Constants?

A constant holds an unchanging value throughout a program's execution, improving code clarity and preventing unintended reassignment.

**Example:**

```
const constantExample = 10;

// Uncommenting the line below will result in an error

// constantExample = 20;
// TypeError: Assignment to constant variable.
```

Constants are declared using the *const* keyword.

Constants cannot be reassigned after their initial assignment.

# let vs. const

The difference between *let* and *const* keywords is listed below:

## let

- *let* is used when you need to reassign a variable.
- It declares a local variable in a block scope and can be used for loops or mathematical operations.

## const

- *const* means that the identifier cannot be reassigned.
- It declares a local variable in a block scope, like let, and is used for constants that should not change, such as configuration values or fixed references.

# What Are Blocks?

A block statement is a group of zero or more statements enclosed in {}, where variables declared with let and const have block scope.

let

**Example:**

```
let x=1;
{
let x=5;
}

console.log(x); //answer will be 1
```

const

**Example:**

```
const y=20;
{
const y=40;
}

console.log(y); //answer will be 20
```

# Blocks: Example

Block scope restricts access to variables declared inside a block, preventing them from being accessed outside of it.

**Example:**

```
function display(){
    if(true){
        var item1 = 'apple';         //exist in function scope
        const item2 = 'ball';         //exist in block scope
        let item3 = 'cloud';      //exist in block scope
    }
    console.log(item1);
    console.log(item2);
    console.log(item3);
}

display();
//result:
//apple
//error: item2 is not defined
//error: item3 is not defined
```

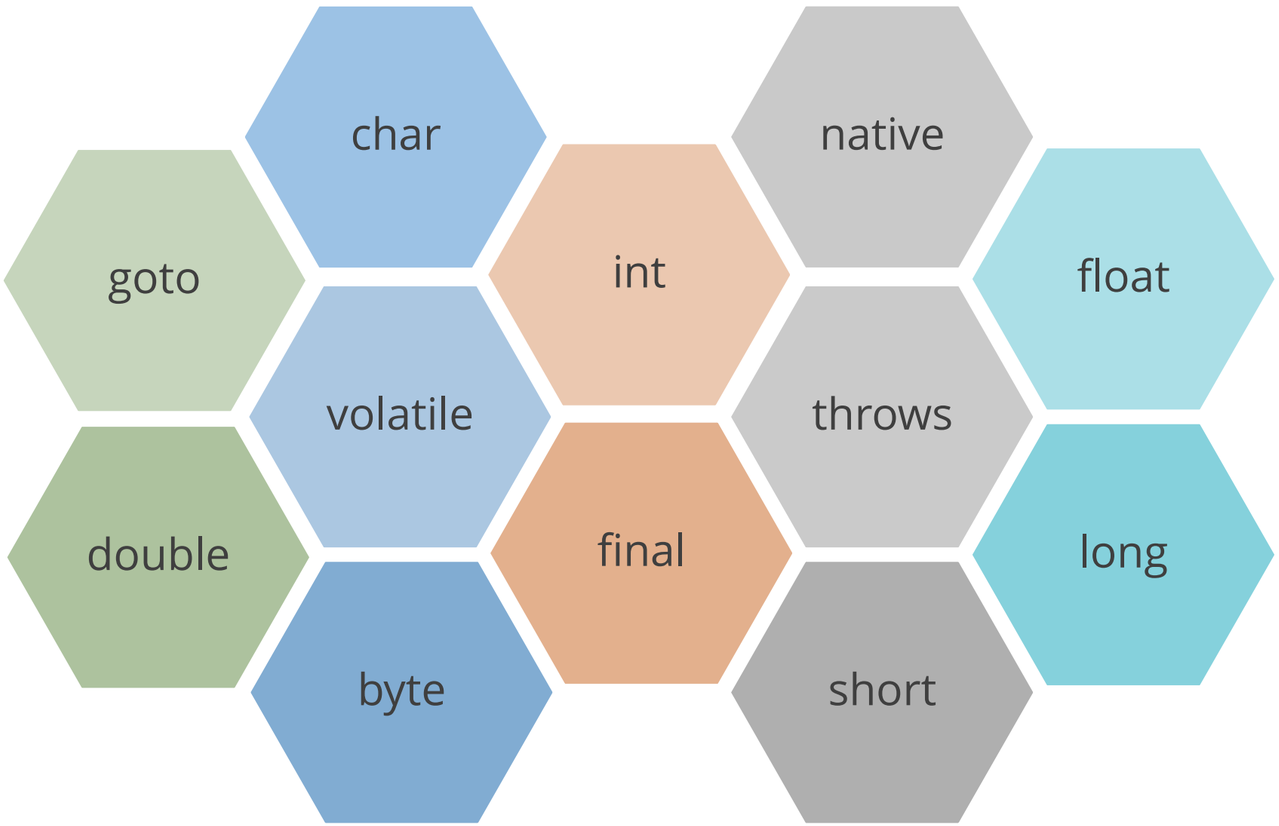
# What Are Keywords?

Keywords are reserved words that have a specific meaning for the compiler.


These words are reserved for language features and functionalities, so they cannot be used as identifiers like variable or function names.

# Reserved Keywords in Programming

Below is a list of standard reserved keywords that cannot be used as identifiers:

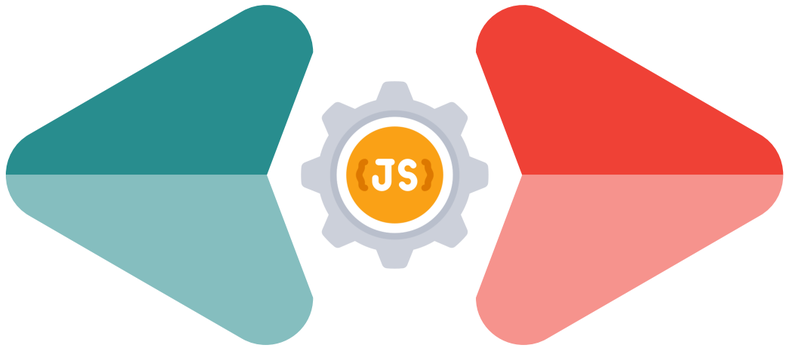


# Data Types in JavaScript

A data type defines the kind of value a variable can hold and determines the operations that can be performed on it without errors.

There are two types of data types in JavaScript:

Primitive data type




Non-primitive data type

# Primitive Data Type

A primitive data type is not an object and has no methods and properties. The types of primitive data types are:

1. Numbers

**0 1 2 3 4 5 6**

2. Strings

**“Hello World”**

3. Boolean

**1010101**

4. Null

**{ }**

5. Undefined

**?**

# Non-Primitive Data Type

Non-primitive data types (Objects) are not defined by the programming language but can be created by the programmer. The types of non-primitive data types are:

1. Objects

2. Arrays


3. Functions

4. Date

5. Regex

{}*?@[^]\.

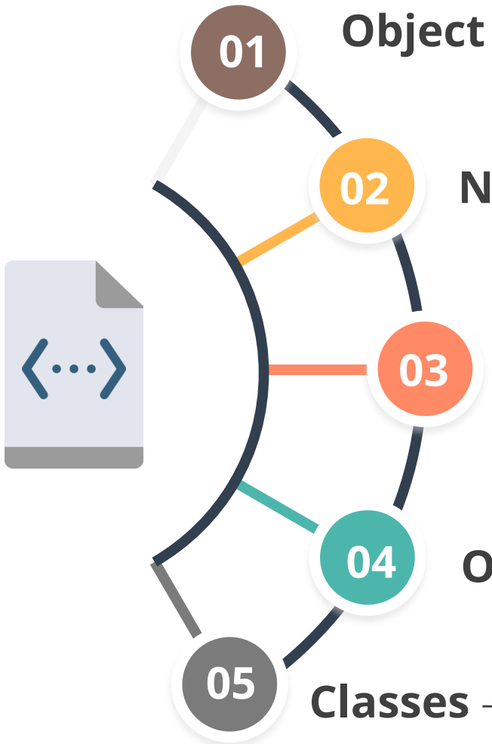
# Primitive vs. Non-Primitive Data Types

The differences between primitive and non-primitives are:

| Feature | Primitive data types | Non-primitive data types |
| --- | --- | --- |
| **Mutability** | Immutable; values cannot be changed after creation | Mutable; values can be changed after creation |
| **Definition** | Predefined in JavaScript | User-defined and created dynamically |
| **Storage mechanism** | Stored directly in memory | Stored as references in memory |
| **Comparison** | Compared by value | Compared by reference |
| **Examples** | Number, String, Boolean, Undefined, Null, Symbol, BigInt | Object, Array, Function, Date, RegExp |

# Non-Primitive Data Types (Objects)

The different methods to create objects are:




**Object literal** → Best for single objects with known properties

**New operator with Constructor Function** → Used for creating multiple instances

**Factory functions** → Provide flexibility by returning customized objects

**Object.create()** → Useful for prototype-based object inheritance

**Classes** → Modern ES6+ way to create objects with constructors and inheritance

# Objects Creation: Object Literal

The object literal can be created by using key-value pairs.

**Example:**

```
var greeting = {
Fullname: “Ben Shapiro",
greet: (message, name) => {
console.log(message + " " + name + "!!");
}
};
```

The object literal is an array of key-value pairs with commas (,) following each key-value pair and colons (:) separating the keys and values.

# Objects Creation: New Operator

The new operator allows the programmer to create an instance of a user-defined or built-in object type.

Below are some examples:

*new* operator with object constructor

**Example:**

```
let student = new Object();

student.Name = 'testName';
student.Rollno = 'testRollno';
```

*new* operator with a constructor function

**Example:**

```
function student(Name, Rollno)
{
  this.FullName = Name;
  this.Rollno = rollno;
}

let student1 = new
Student('Fullname', 'Rollno');
```

# Objects Creation: Factory Functions

A factory function is a function that returns an object, allowing dynamic object creation.

**Example:**

```
function createStudent(name, rollno) {
    return {
        Name: name,
        Rollno: rollno,
        details() {
            console.log(`Student: ${this.Name}, Roll No:
${this.Rollno}`);
        }
    };
}

let student1 = createStudent("Alice", 101);
student1.details(); // Output: Student: Alice, Roll No: 101
```

Factory functions provide a reusable way to create multiple object instances without using classes or constructor functions.

# Objects Creation: Object.create()

The **Object.create()** method generates a new object using an existing object as a prototype.

**Example:**

```
let org = { company: 'Simplilearn' };

let student = Object.create(org, { name: { value: 'student1' } });

console.log(student);
console.log(student.name);
```

The code demonstrates how to create a new object (student) using Object.create() with an existing object (org) as its prototype while also defining a new property (name) with a specified value.

# Objects Creation: Class

The class method is like using new operator with a user-defined constructor function.

**Example:**

```
class student {

  constructor(Name, Rollno) {
    this.Name = fullname;
    this.Rollno = rollno;
  }

}
```

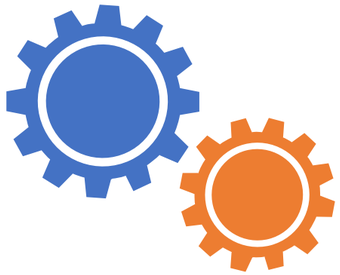
The code defines a student class with a constructor to initialize Name and Rollno, but it has errors—fullname and rollno should be Name and Rollno for correct parameter assignment.

# What Is Data Type Conversion?

It is a fundamental aspect of JavaScript that allows developers to convert data from one type to another.

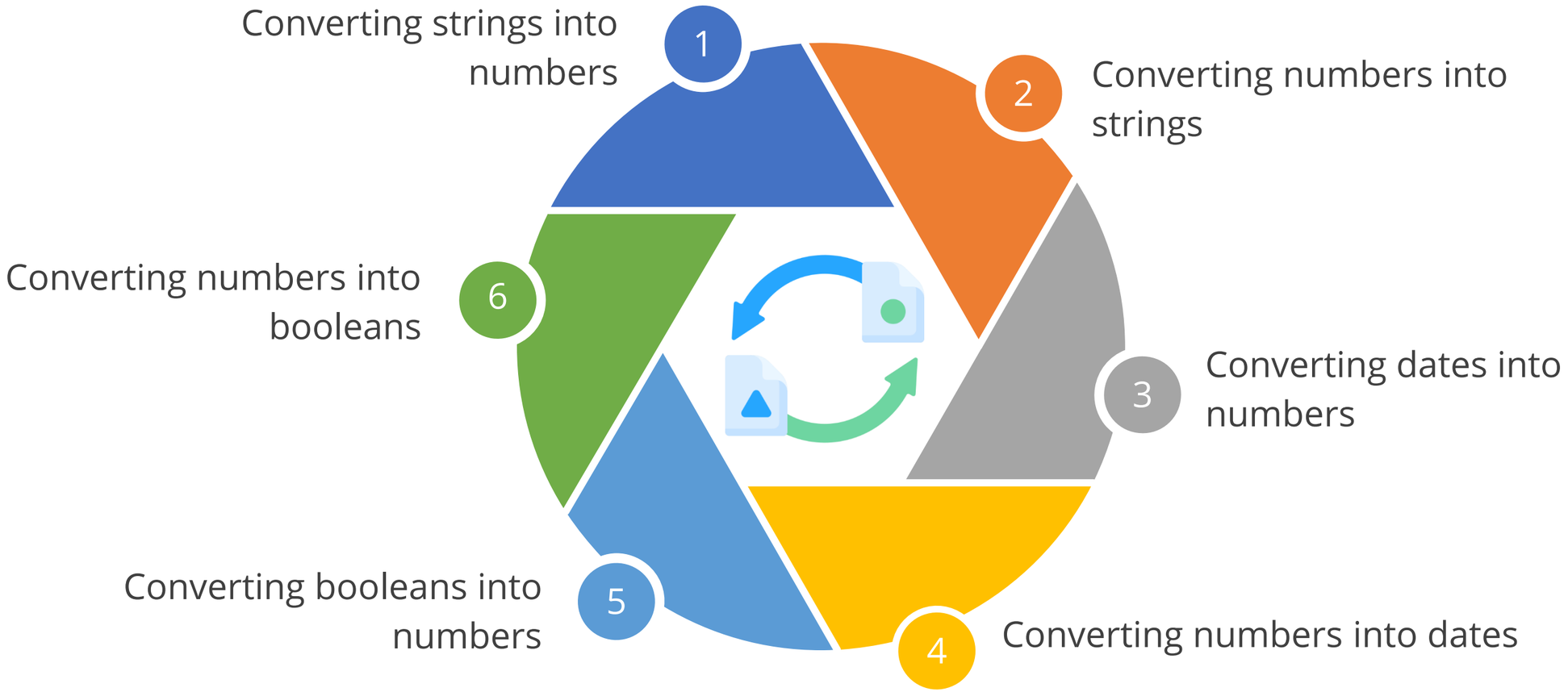
Below are its types:

**Implicit conversion:**
JavaScript automatically converts data types when needed, such as converting a number to a string during concatenation ("5" + 2 → "52").



**Explicit conversion:**
Explicit conversion requires manual type conversion using functions like Number(), String(), or Boolean() to ensure the correct data type.

# Data Type Conversion: Common Conversion Functions



# Quick Check


You are developing a JavaScript program that stores a user's name and a list of their favorite movies. Which data types should you use for each?

A. String for name, Array for movies

B. Number for name, String for movies

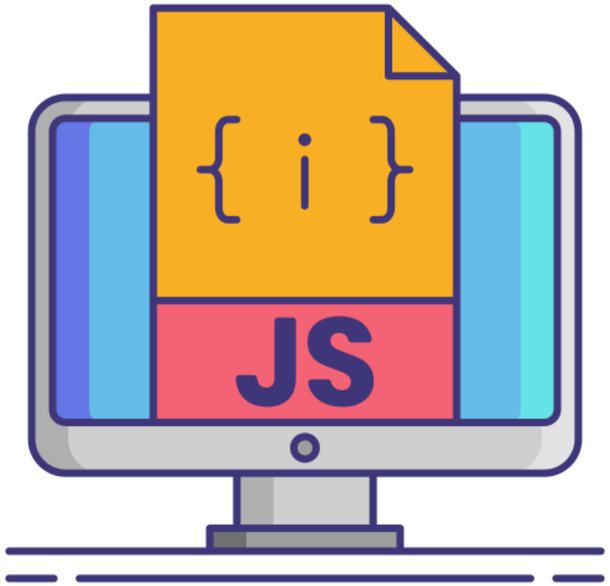
C. Boolean for both

D. Object for name, Number for movies

# Implementing Operators and Expressions in JavaScript

# What Are Operators?

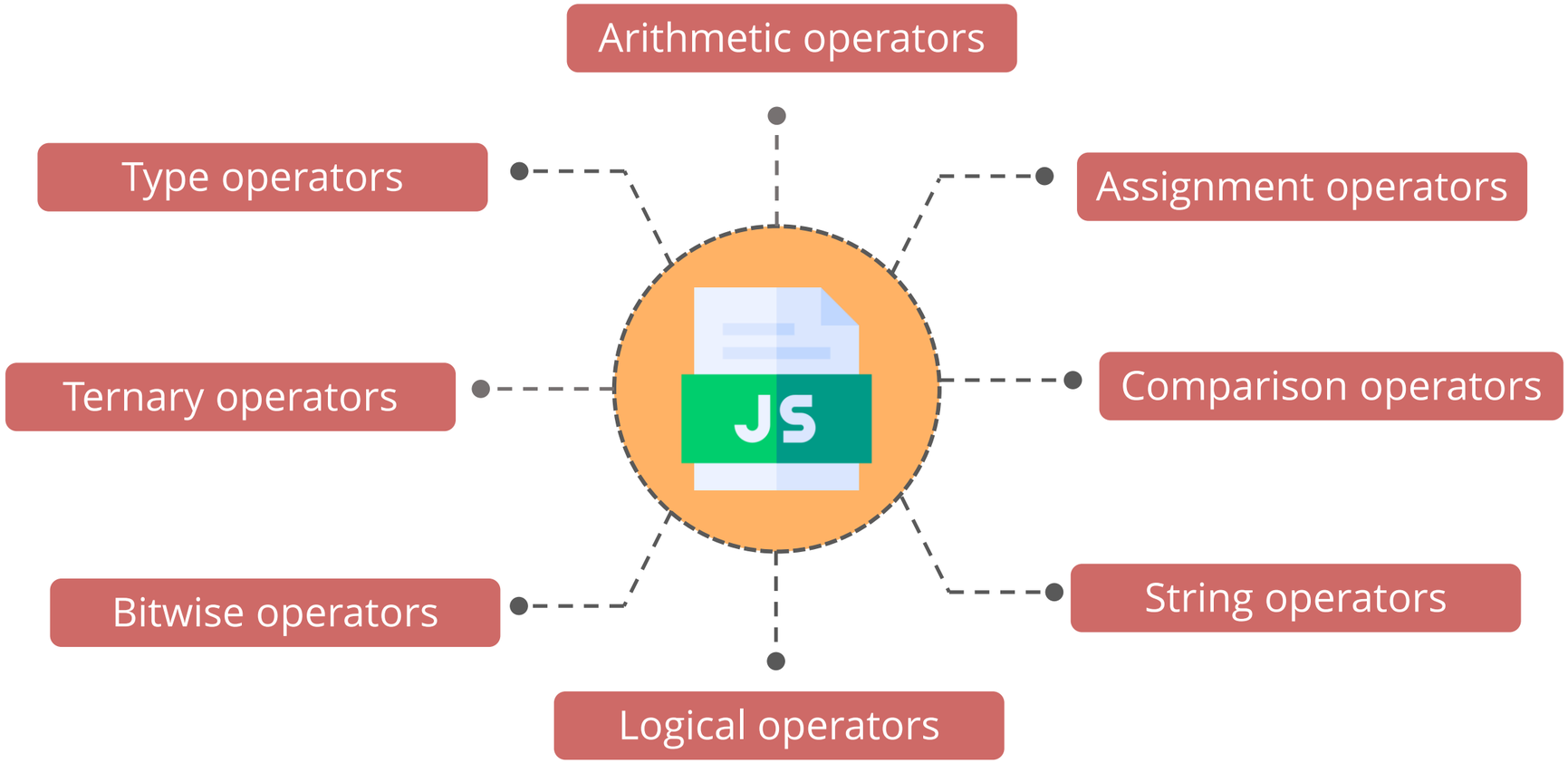
Operators in JavaScript are symbols that perform operations on values and variables. They are used to manipulate data, perform calculations, compare values, and control program logic.




JavaScript provides different types of operators, each serving a specific purpose.

# Types of Operators

The following are the different types of JavaScript operators:



# Arithmetic Operators

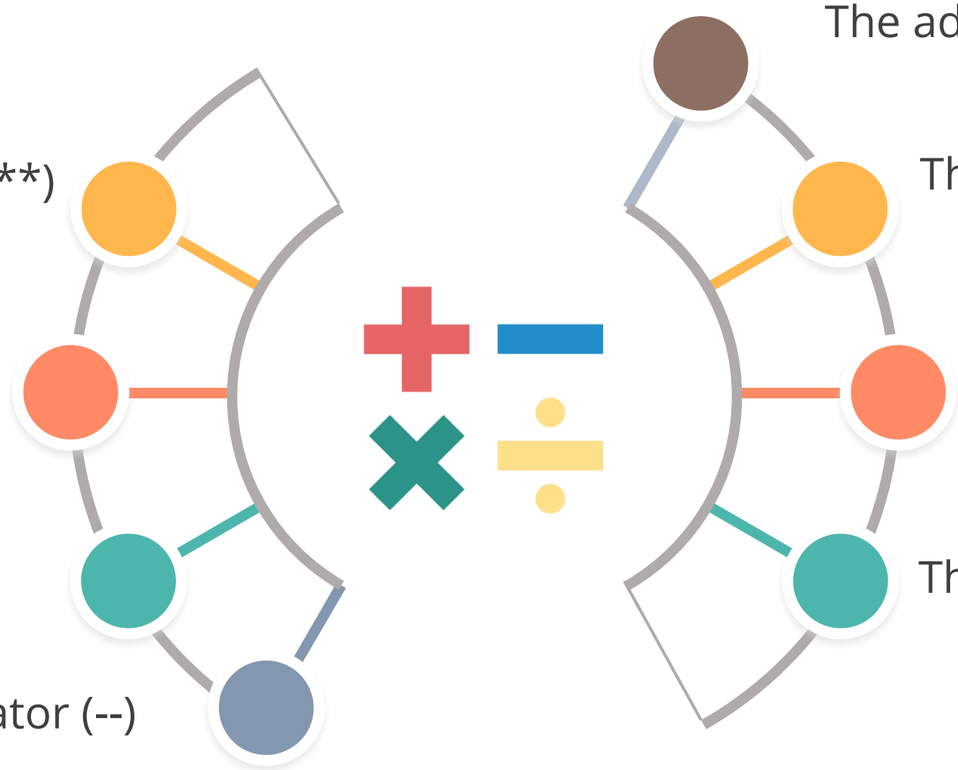
They are used to perform mathematical operations like addition, subtraction, multiplication, division, and modulus on the given operands.

The exponentiation operator (**)

The modulus operator (%)

The increment operator (++)

The decrement operator (--)



The addition operator (+)

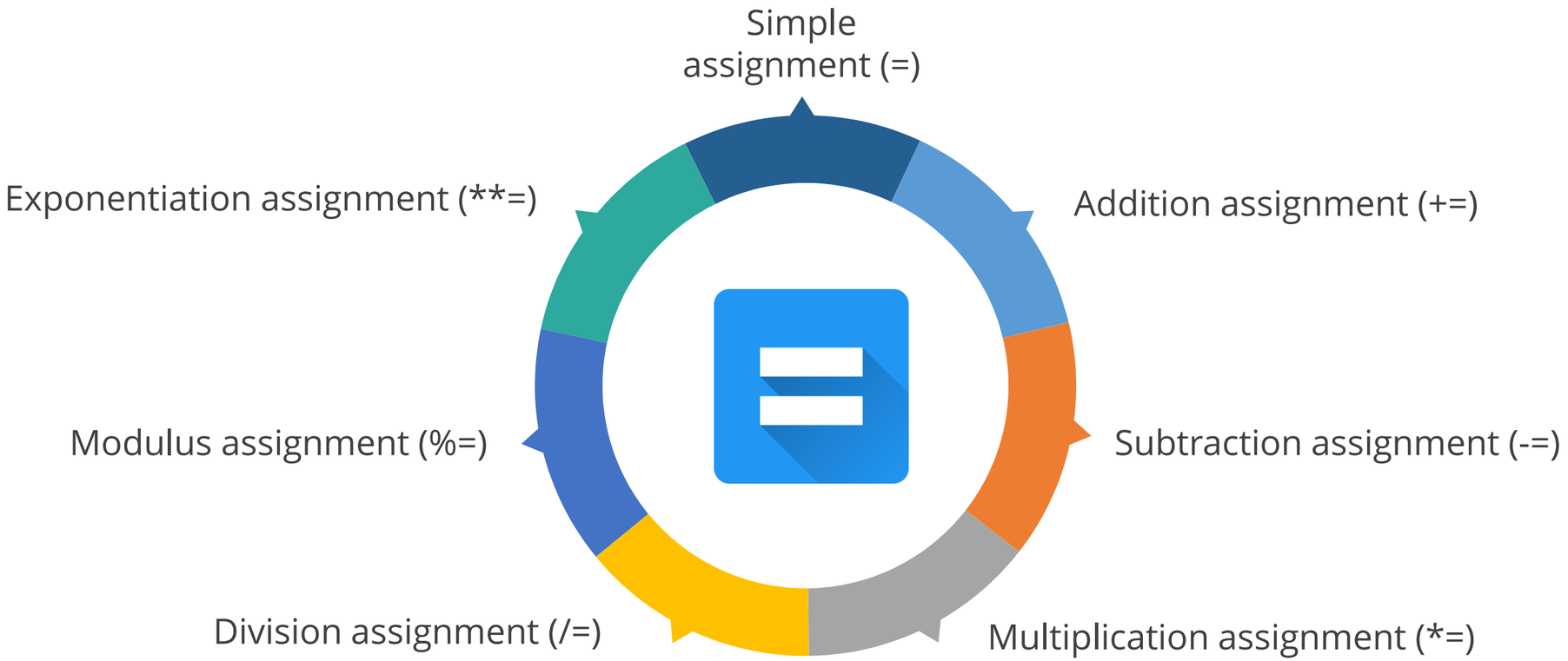
The subtraction operator (-)

The asterisk operator (*)

The slash operator (/)

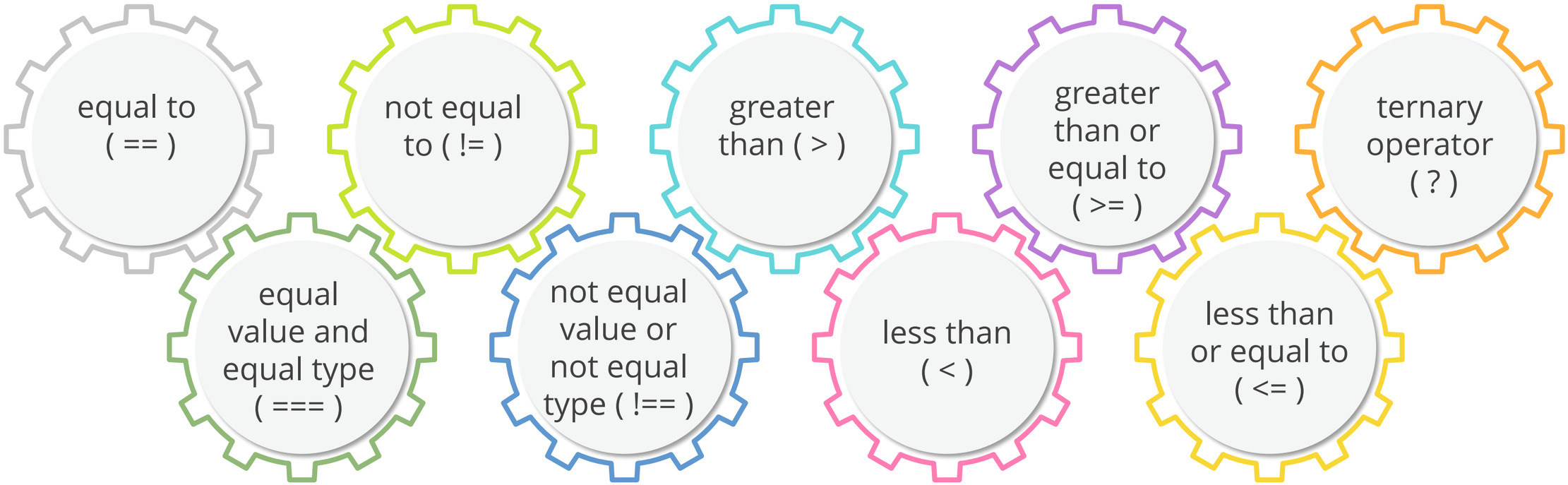
# Assignment Operators

They assign values to variables. Following are the operators used for assigning values in JavaScript:



# Comparison Operators

They compare values and return a boolean result. Following are the operators used to compare the values:




All the comparison operators (listed above) can be used on strings.

# String Operators

These are used to manipulate and concatenate strings in JavaScript. The most commonly used string operators include the concatenation operator (+) and the concatenation assignment operator (+=).

**Concatenation operator (+) –**

It joins two or more strings together.

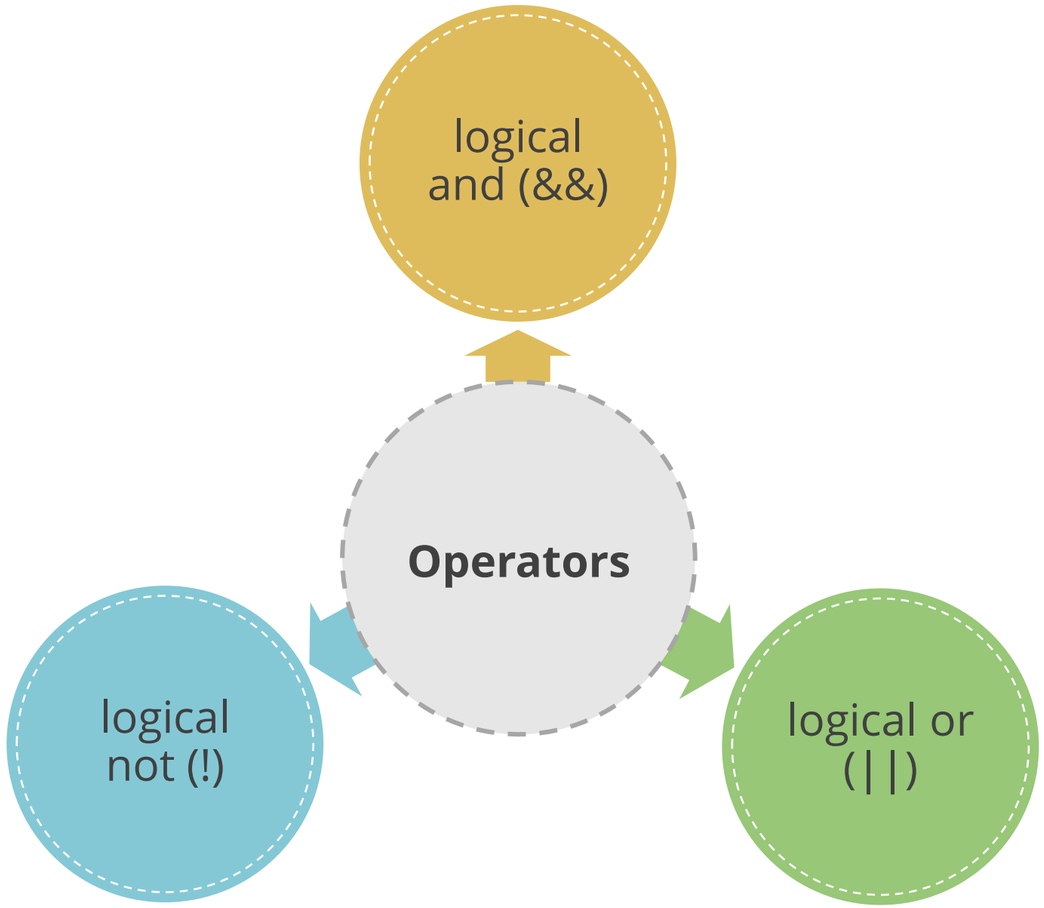
**Concatenation assignment operator (+=) –**

It appends a string to an existing variable.

These string operators help in dynamically constructing text-based content, making them essential for handling and displaying strings in JavaScript applications.

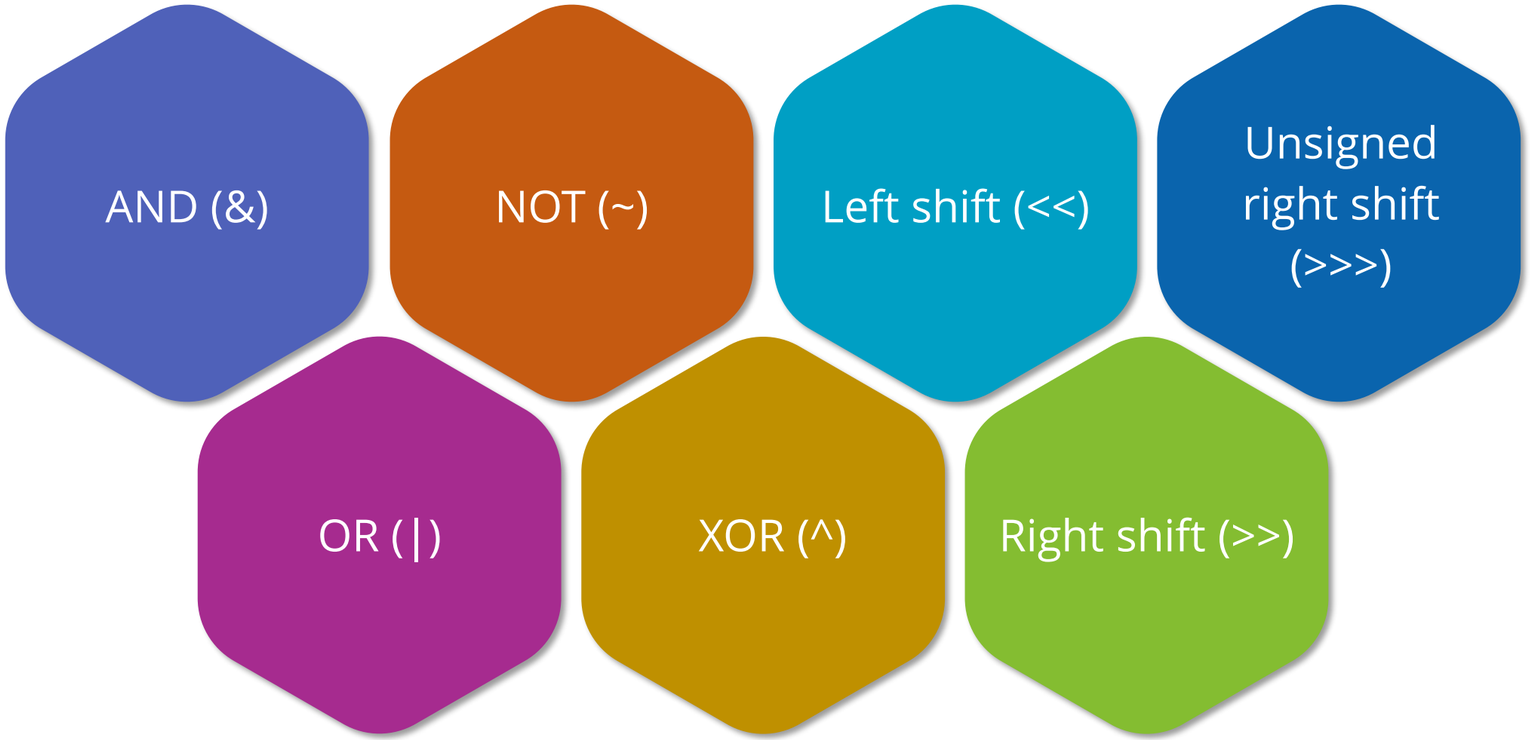
# Logical Operators

They perform logical operations and return a boolean result. Following are the operators used for logical operations:



# Bitwise Operators

They perform operations on binary representations of numbers. Following are the operators used for bitwise operations:



# Ternary Operators

These are used to evaluate conditions in JavaScript using a compact, single-line expression. The ternary operator (? :) is a shorthand for if-else statements, making code more readable and efficient.

**? :**

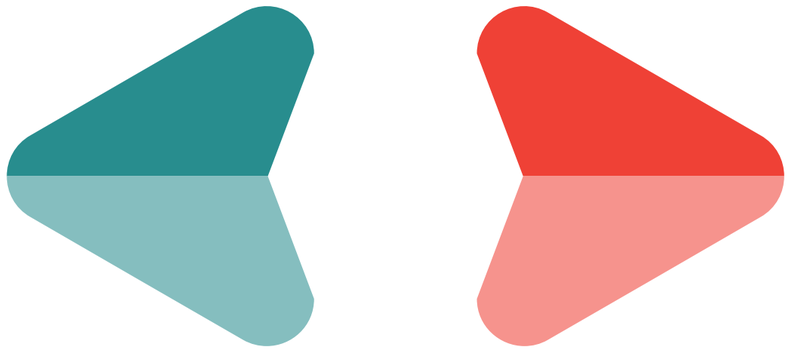
It evaluates a condition and returns one of two values based on whether the condition is true or false.

# Type Operators

These operators are used to determine the type or structure of a variable in JavaScript. The two main type operators in JavaScript are as follows:

**typeof**

Returns a string indicating the type of a variable



**instanceof**

Checks if an object is an instance of a specific class or constructor

# Understanding Operator Precedence and Associativity

## Operator precedence

- It defines the sequence in which operators are evaluated in an expression
- Higher precedence operators are executed first
- Parentheses allow explicit control over evaluation order

## Operator associativity

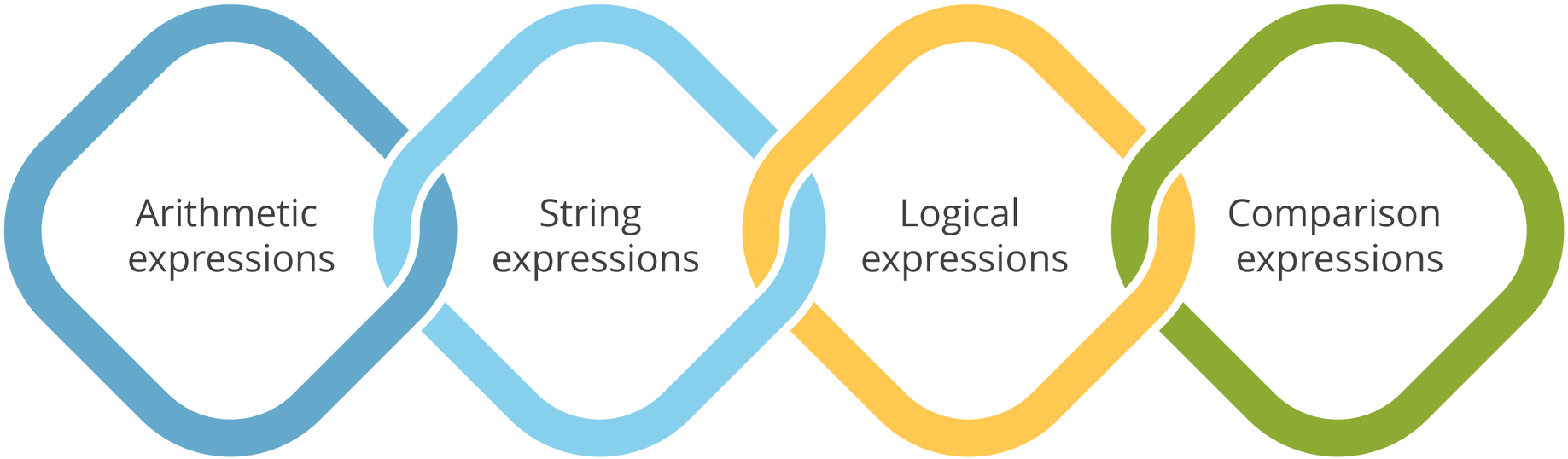
- It determines the direction in which operators of the same precedence are evaluated
- Left-to-right associativity means the leftmost operation is executed first
- Right-to-left associativity means the rightmost operation is executed first

Understanding operator precedence and associativity ensures correct expression evaluation, helping to avoid unexpected results and improving code efficiency in JavaScript.

# What Is Expression?

An expression is a combination of values, variables, and operators that produces a result.

Expressions can be broadly categorized into:




Expressions are the foundation of JavaScript computations and decision-making, allowing developers to effectively manipulate data, perform operations, and control program flow.

# Arithmetic Expressions

Arithmetic expressions perform mathematical operations using numeric values and arithmetic operators, producing a numerical result.

**Example:**

```
let a = 10;
let b = 5;

let sum = a + b;        // Addition: 10 + 5 = 15
let difference = a - b; // Subtraction: 10 - 5 = 5
let product = a * b;    // Multiplication: 10 * 5 = 50
let quotient = a / b;   // Division: 10 / 5 = 2
let remainder = a % b; // Modulus: 10 % 5 = 0
```

They are essential for calculations in JavaScript, enabling mathematical computations that are fundamental to programming logic and data manipulation.

# String Expressions

String expressions involve manipulating and combining text values using string operators, producing a textual output.

**Example:**

```
let firstName = "John";
let lastName = "Doe";

let fullName = firstName + " " + lastName; //
Concatenation: "John Doe"
let greeting = `Hello, ${firstName}!`;     //
Template Literal: "Hello, John!"

let repeatedText = "Hi! ".repeat(3);       // Repeats
string: "Hi! Hi! Hi! "
let messageLength = greeting.length;       // String
length: 12
```

They are essential for handling and displaying text-based data, making them crucial for dynamic content generation in JavaScript applications.

# Logical Expressions

Logical expressions evaluate conditions using boolean logic, determining true or false outcomes based on logical operators.

**Example:**

```
let x = 10;
let y = 5;
let isGreater = x > y && y > 0;  // Logical AND: true
(both conditions must be true)
let isEitherPositive = x > 0 || y < 0;  // Logical
OR: true (at least one condition is true)
let isNotEqual = !(x === y);  // Logical NOT: true
(negates the condition)
```

They are fundamental for decision-making in JavaScript, allowing efficient control flow and condition handling in programs.

# Comparison Expressions

Comparison expressions evaluate the relationship between values and return a boolean result (true or false), helping in decision-making.

**Example:**

```
let a = 10;
let b = 5;

let isEqual = a == b;        // Equal to: false
let isStrictEqual = a === b; // Strict equal to
(checks type too): false
let isNotEqual = a != b;     // Not equal to: true
let isGreater = a > b;       // Greater than: true
let isLessOrEqual = a <= b;  // Less than or equal
to: false
```

These expressions play a crucial role in conditional statements and loops, enabling logical decision-making in JavaScript.

# What Is a Statement?

It is a complete line of code that performs a specific action. The following are the types of statements:

1. **Variable declaration:** Declares a variable using let, const, or var
2. **Assignment statement:** Assigns a value using =
3. **Conditional statements:** Control flow using if-else and switch
4. **Loop statements:** Execute repetition using for, while, and do-while
5. **Function declaration:** Defines a function using the function keyword

JavaScript statements are fundamental building blocks of programs, allowing developers to define, control, and execute code logic effectively.

# Assisted Practice

**Using Variables and Data Types** **Duration: 15 Min.**

**Problem statement:**

You have been asked to create and execute a JavaScript file to demonstrate the usage of variables, primitive data types, and data type conversion. This includes declaring and updating variables, performing implicit and explicit conversions, and validating data through console output.

**Outcome:**

By the end of this task, you will have successfully declared and modified variables, worked with primitive data types, converted data types, and validated JavaScript operations, ensuring correct usage and systematic execution.

**Note:** Refer to the demo document for detailed steps:
01_Using_Variables_and_Data_Types

# Assisted Practice: Guidelines

Steps to be followed:

1. Create Demo1 folder
2. Execute the JavaScript file

# Assisted Practice

**Using Operators and Expressions** **Duration: 15 Min.**

**Problem statement:**

You have been asked to create and execute a JavaScript file to demonstrate the use of arithmetic, comparison, logical, bitwise, and ternary operators. The task also involves understanding operator precedence, expressions, and new JavaScript features like numeric separators and Object.groupBy().

**Outcome:**

By the end of this task, you will have successfully applied various operators, evaluated expressions, and validated their execution using systematic testing, ensuring a clear understanding of operator functionality in JavaScript.

**Note:** Refer to the demo document for detailed steps:
02_Using_Operators_and_Expressions

# Assisted Practice: Guidelines

Steps to be followed:

1. Create an OperatorsAndExpressions.js folder
2. Execute the JavaScript file

# Quick Check


A JavaScript program calculates the final price of a product after applying a discount. Which type of expression is used to compute the final price?

A. Arithmetic expression

B. String expression

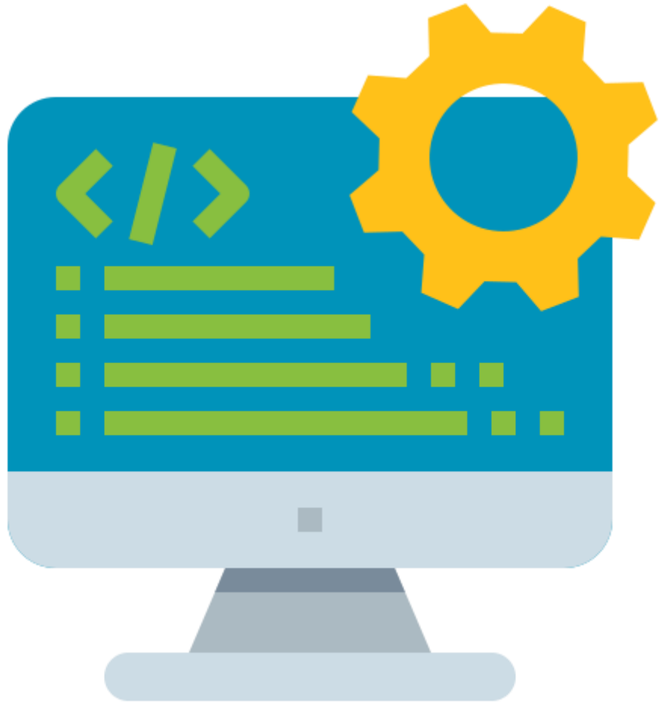
C. Logical expression

D. Comparison expression

# Exploring the Control Statements

# What Are Control Statements?

Control statements determine the flow of execution in a program, enabling conditions, loops, and structured flow control.

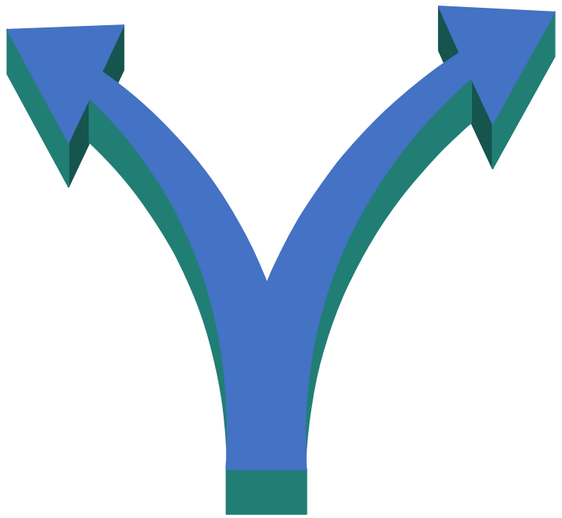


These statements play a crucial role in decision-making and looping constructs, allowing developers to manage the execution flow based on conditions and repetitions efficiently.

# Control Statements: Types

Control statements are divided into two categories:

Conditional statements

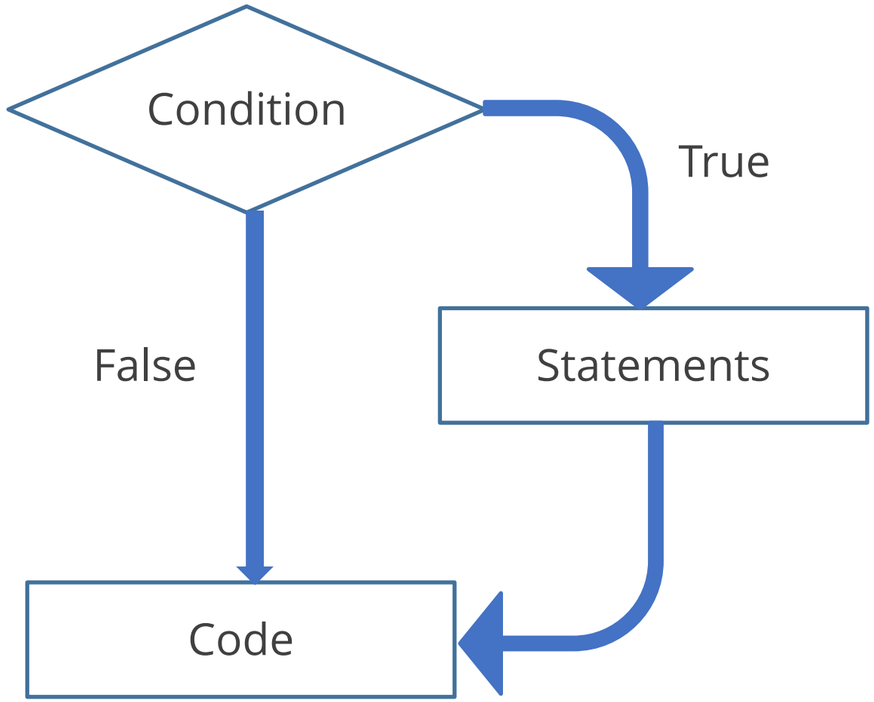


Iterative statements

They manage the flow of execution in JavaScript, enabling decision-making through conditional statements and repetition using loops, ensuring structured and efficient program execution.

# What Are Conditional Statements?

They evaluate expressions and determine the program's execution path based on whether the condition is true or false.




- When the condition evaluates to true, the program executes the specified statements and proceeds to the next step.
- If the condition evaluates to false, an alternative code block (if provided) executes, or the program skips the conditional statements.

# Conditional Statements: If Statement

The *if* statement executes the code block only when the given condition evaluates to true.

**Example:**

```
<script>
  var a = 20;
  if (a > 10) {
    document.write("The value of a is greater than
10");
  }
</script>
```

The if statement is essential for decision-making in programming, allowing conditional execution of specific code blocks based on given conditions.

# Conditional Statements: If else

The *if-else* statement executes one block of code if the given condition evaluates to true and another block if the condition evaluates to false.

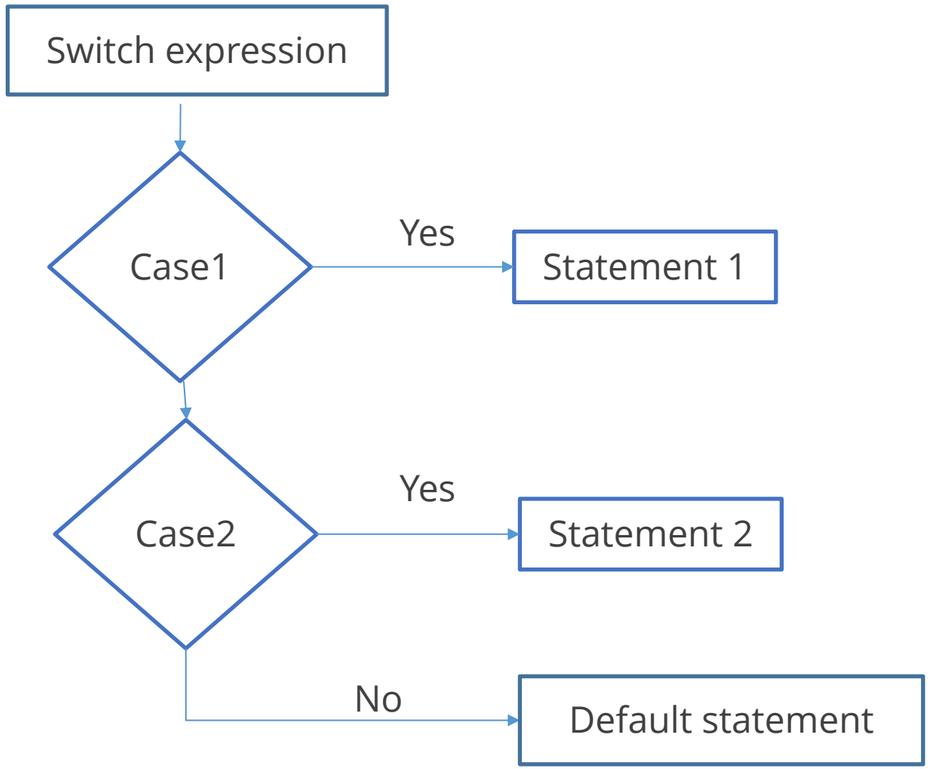
**Example:**

```
<script>
  var a = 20;
  if (a % 2 == 0) {
    document.write("a is an even number");
  } else {
    document.write("a is an odd number");
  }
</script>
```

They extend the if statement by providing an alternative execution path when the condition is false, ensuring better decision-making in program flow.

# Conditional Statements: Switch

The JavaScript switch statement evaluates an expression and executes the corresponding code block based on matching cases.




- The switch statement evaluates an expression and compares it against multiple case clauses before executing the matching block.
- If no case matches, the default statement executes (if provided).
- Without a break statement, execution continues to the next case, leading to fall-through behavior.

# Conditional Statements: Switch

The code below demonstrates the usage of a switch statement in JavaScript:

**Example:**

```
<script>
var grade = 'B';
var result;

switch (grade) {
    case 'A':
        result = "A Grade";
        break;
    case 'B':
        result = "B Grade";
        break;
    default:
        result = "No Grade";
}

document.write(result);
</script>
```

If no case matches the provided value, the default case executes. In this example, if the grades were neither 'A' nor 'B,' it would output 'No Grade.'

# What Are Iterative Statements?

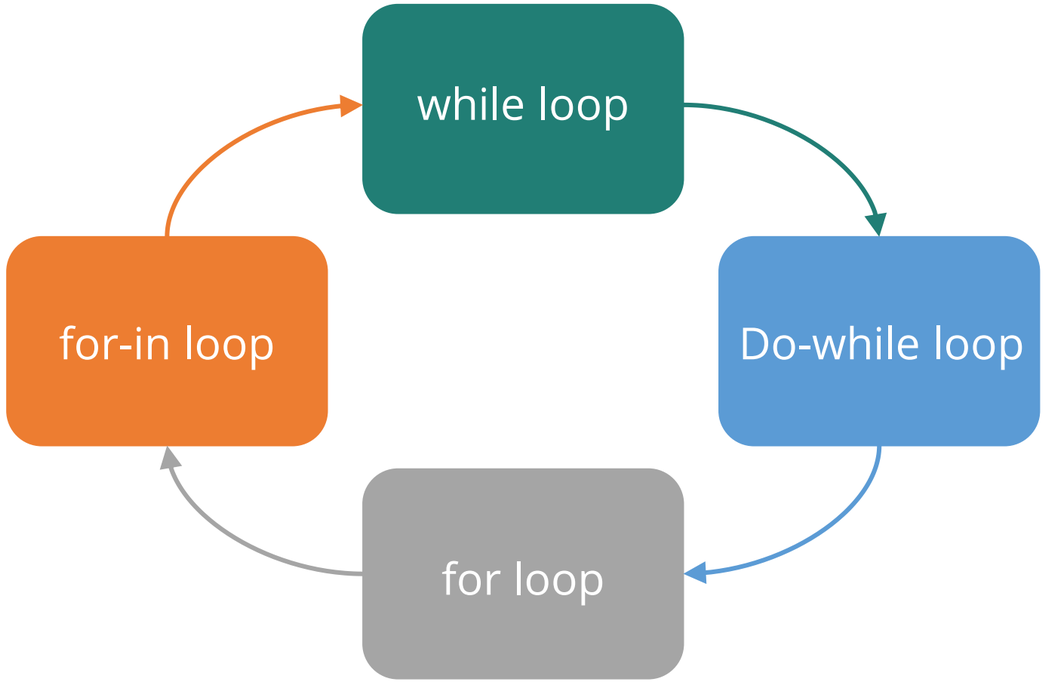
Iterative statements, also known as loop statements, execute a block of code repeatedly based on a specified condition. These loops continue execution until the defined termination condition is met.

The main types of iterative statements in JavaScript are:

1. While statement
2. Do-While statement
3. For statement

# What Is a Loop?

A loop is a control structure that repeatedly executes a block of code as long as a specified condition holds true. It helps automate repetitive tasks and ensures efficient execution of iterative processes.




Loops simplify repetitive tasks, making the code efficient and readable. They are commonly used to iterate over arrays, objects, and other iterable structures.

# While Loop

A *while* loop repeatedly executes a block of code as long as the specified condition remains true. It is useful when the number of iterations is unknown beforehand.

**Example:**

```
<script>
var i = 11;
while (i <= 15) {
   document.write(i + "<br/>");
   i++;
}
</script>
```

The key difference between if and while statements is that *if* executes a block of code once when the condition is true, whereas *while* continuously executes a block of code as long as the condition remains true.

# Do-While Loop

A *do-while* loop executes the loop body at least once before evaluating the condition. It is useful when an operation must be performed at least once, even if the condition is false.

**Example:**

```
<script>
var i = 21;
do {
   document.write(i + "<br/>");
   i++;
} while (i <= 25);
</script>
```

*Do-while* loops ensure that the code executes at least once, making them useful for input validation, menu-driven programs, and scenarios requiring at least one iteration before condition evaluation.

# For Loop

A *for* loop is used to execute a block of code a specific number of times. It consists of three components initialization, condition, and update that control the execution flow.

**Example:**

```
<script>
for (let i = 1; i <= 5; i++) {
    document.write(i + "<br/>");
}
</script>
```

*For* loops provide a structured way to perform repeated actions efficiently, making them essential for iterating over data structures and handling repetitive tasks in JavaScript.

# For-In Loop

The *for-in* loop iterates over an object's enumerable properties, including inherited ones. It is primarily used for iterating over object keys rather than array elements.

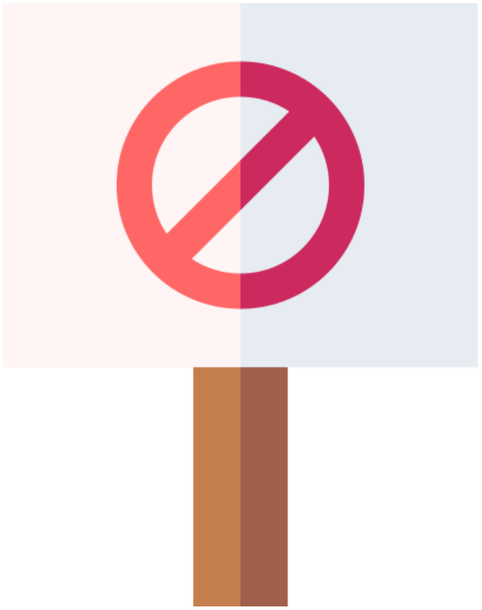
**Example:**

```
<script>
let person = {name: "John", age: 30, city:
"New York"};

let text = "";
for (let key in person) {
    text += key + ": " + person[key] +
"<br>";
}

document.write(text);
</script>
```

The *for-in* loop executes once for each enumerable property of an object, making it useful for iterating over object keys but not ideal for arrays.

# Break and Continue Statements

The break and continue statements control the flow of loops by either stopping execution early (break) or skipping a specific iteration (continue).



Break stops execution entirely, while continue skips only the current iteration. Both are essential for controlling loop flow efficiently.

# Break Statement

The break statement immediately exits the loop, even if the loop condition remains true.

**Example:**

```
for (let i = 0; i < 5; i++) {
    if (i === 3) {
        console.log("Loop terminated at i
=", i);
        break;
    }
    console.log(i);
}
```

It is commonly used in *for*, *while*, and *do-while* loops to stop execution when a certain condition is met.

# Continue Statement

The continue statement skips the current iteration of the loop but does not stop the loop.

**Example:**

```
for (let i = 0; i < 5; i++) {
    if (i === 2) {
        console.log("Skipping iteration
for i =", i);
        continue;
    }
    console.log(i);
}
```

It is useful when certain conditions should be ignored without exiting the loop.

# Assisted Practice

**Using Control Flow Statements** **Duration: 15 Min.**

**Problem statement:**

You have been asked to create and execute a JavaScript file demonstrating control flow statements, including conditional statements (if-else, switch), loops (for, while, do-while), and branching statements (break, continue).

**Outcome:**

By the end of this task, you will have successfully implemented control flow statements, loops, and branching mechanisms in JavaScript. You will also validate program execution through structured decision-making and iteration techniques.

**Note:** Refer to the demo document for detailed steps:
03_Using_Control_Flow_Statements

# Assisted Practice: Guidelines

Steps to be followed:

1. Create a ControlFlowStatements.js folder
2. Execute the JavaScript file

ASSISTED PRACTICE

# Assisted Practice

**Implementing Form Validation** **Duration: 15 Min.**

**Problem statement:**

You have been asked to create and execute a JavaScript-based login form validation. The task involves verifying user input fields, checking credentials, and displaying appropriate alerts based on validation results.

**Outcome:**

By the end of this task, you will have successfully implemented form validation using JavaScript, ensuring that login credentials are checked before submission and providing user feedback for valid or incorrect inputs.

**Note:** Refer to the demo document for detailed steps:
04_Implementing_Form_Validation

ASSISTED PRACTICE

# Assisted Practice: Guidelines

Steps to be followed:

1. Create a folder
2. Execute the JavaScript file

# Quick Check


You are building a JavaScript program that processes a list of user orders. The program should keep checking for new orders and process them until there are no more left. Which loop structure is the best choice?

A. for loop

B. while loop

C. switch statement

D. if statement

# Exploring Arrays

# Introduction to Arrays

An array is a fundamental data structure in JavaScript that stores a collection of items in an ordered manner, allowing efficient data management and manipulation.

**Syntax:**

```
//syntax for creating an Array
const arr = new Array ();
//alternative way
const arr = [];
```

They can contain various data types, including strings, numbers, objects, and other arrays, while providing efficient methods for manipulating and accessing elements.

# Arrays: Example

The following is an example of creating an array in JavaScript to represent a grocery shopping list:

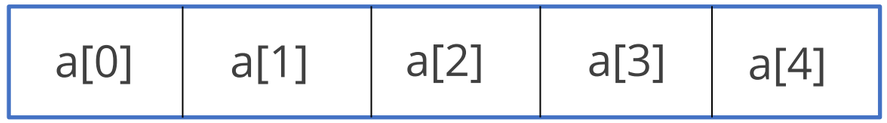
**Example:**

```
<script>
const groceryList = ["apples", "bananas",
"oranges"];
</script>
```

This JavaScript code creates an array to store a grocery list, making item management easier.

# Arrays: Indexing

Arrays are indexed starting from 0, the first element has an index of 0, the second has an index of 1, and so on. The following diagram shows indexing in an array:



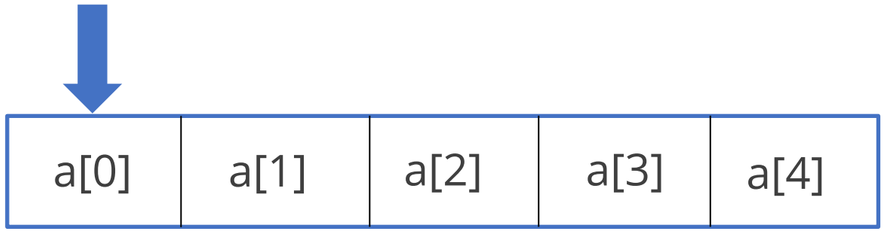

**Example:**

```
let a = [10, 20, 30, 40, 50];
console.log(a[2]); // Output: 30
```

Elements within an array are accessed using their respective indices.

# Array Positioning: Calculating Memory Location

The below diagram depicts the position and memory location of an element in an array:




- Arrays store elements in contiguous memory, starting at index 0.
- Each element’s memory address is calculated as Base Address + (Index × Element Size).

The array's name typically represents the memory location of the first element. The offset of an element is determined by its index multiplied by the element size, allowing efficient indexing for fast element access.

# Properties of Arrays

The following are the built-in properties for effective data manipulation:

1. **length:** Sets or returns the number of elements in an array
2. **constructor**: Returns the function that created the array's prototype
3. **prototype:** Facilitates adding properties and methods to array objects

# Example: Length

Below is an example of the length property of arrays:

**Example:**

```
const fruits = ['apple',
'banana', 'orange’];
console.log(fruits.length);
// Output: 3
```

The length property of an array returns the total number of elements in the array. In this example, the fruits array contains three elements, so fruits.length outputs 3.

# Example: Constructor

Below is an example of the constructor property of arrays:

**Example:**

```
const arr = [1, 2, 3];
 console.log(arr.constructor);
// Outputs: function Array() {
[native code] }
```

The constructor property of an array returns the function that created the array. In this example, arr.constructor output's function Array() { [native code] }, indicating that arr is an instance of the Array object in JavaScript.

# Example: Prototype

Below is an example of the prototype property of arrays:

**Example:**

```
Array.prototype.customMethod =
function() {
// Your custom method logic here
};
```

The prototype property allows adding custom methods to all array instances. In this example, Array.prototype.customMethod defines a new method that can be accessed by any array, extending its functionality.

# What Are Methods in Arrays?

Array methods in JavaScript provide built-in functions that allow efficient manipulation, transformation, and management of array elements.

JavaScript provides a variety of methods to modify, access, and iterate through arrays, enabling efficient data processing and transformation.

# Categories of Array Methods in JavaScript

The methods in arrays can be categorized into different functionalities as listed below:

## Mutator methods:

Modify the original array (Example: **push()**, **pop()**, **shift()**, **unshift()**, **splice()**).

## Accessor methods:

Return a new value without changing the original array (Example: **concat()**, **slice()**, **includes()**, **indexOf()**).

## Iteration methods:

Execute a function for each array element (Example: **map()**, **filter()**, **forEach()**, **reduce()**).

# Arrays: Methods

The following are some of the methods in an array:

## push()

Adds one or more elements to the end of an array

**Example:**

```
const numbers = [1, 2, 3];
numbers.push(4, 5);
console.log(numbers);
//Output: [1, 2, 3, 4, 5]
```

## pop()

Removes the last element from an array

**Example:**

```
const numbers = [1, 2, 3];
const removedElement = numbers.pop();
console.log(removedElement);
 // Output: 3
console.log(numbers);
// Output: [1, 2]
```

# Arrays: Methods

## shift()

Removes the first element from an array

**Example:**

```
const numbers = [1, 2, 3];
const removedElement = numbers.shift();
console.log(removedElement);  // Output: 1
console.log(numbers);  // Output: [2, 3]
```

## unshift()

Adds one or more elements to the beginning of an array

**Example:**

```
const numbers = [2, 3];
numbers.unshift(0, 1);
console.log(numbers);
// Output: [0, 1, 2, 3]
```

# Arrays: Methods

## concat()

Merges two or more arrays, creating a new array

**Example:**

```
const array1 = [1, 2];
const array2 = [3, 4];
const newArray = array1.concat(array2);
console.log(newArray);
// Output: [1, 2, 3, 4]
```

## join()

Joins all elements of an array
Into a string

**Example:**

```
const fruits = ['apple', 'banana', 'orange'];
const joinedString = fruits.join(', ');
console.log(joinedString);
// Output: 'apple, banana, orange'
```

# Arrays: Methods

## slice()

Selects a part of an array and returns a new array

**Example:**

```
const numbers = [1, 2, 3, 4, 5];
const slicedArray = numbers.slice(1, 4);
console.log(slicedArray);
// Output: [2, 3, 4]
```

## splice()

Adds or removes elements from an array at a specified index

**Example:**

```
const numbers = [1, 2, 3, 4, 5];
numbers.splice(2, 1, 6);
console.log(numbers);
// Output: [1, 2, 6, 4, 5]
```

# Arrays: Methods

## forEach()

Executes a provided function once for each array element

**Example:**

```
const numbers = [1, 2, 3];
numbers.forEach(num => {
  // Process each element using num
  console.log(num);
});
// Output:
// 1
// 2
// 3
```

## filter()

Creates a new array with elements that satisfy a condition

**Example:**

```
const numbersFilter = [1, 2, 3, 4, 5];
const evenNumbers =
numbersFilter.filter(num => num % 2 === 0);
console.log(evenNumbers);
// Output:
// [2, 4]
```

# Arrays: Methods

## map()

Creates a new array by applying a function to each element

**Example:**

```
const numbersFilter = [1, 2, 3, 4, 5];
const evenNumbers =
numbersFilter.filter(num => num % 2 ===
0);
console.log(evenNumbers);
// Output:
// [2, 4]
```

## reduce()

Reduces an array to a single value through a provided function

**Example:**

```
const numbersReduce = [1, 2, 3, 4, 5];
const sum = numbersReduce.reduce((acc,
num) => acc + num, 0);
console.log(sum);
// Output:
// 15
```

# Assisted Practice

**Implementing Arrays Methods** **Duration: 15 Min.**

**Problem statement:**

You have been asked to create and execute a JavaScript file to demonstrate array properties, manipulation methods, and iteration techniques. This includes performing operations such as adding, removing, merging, slicing, and iterating over array elements.

**Outcome:**

By the end of this task, you will have successfully applied array methods like push(), pop(), splice(), map(), filter(), and reduce(), validating their execution and enhancing your understanding of JavaScript array operations.

**Note:** Refer to the demo document for detailed steps:
05_Implementing_Arrays_Methods

# Assisted Practice: Guidelines

Steps to be followed:

1. Create a folder named ArraysOne
2. Execute the JavaScript file

# What Is a Multidimensional Array?

A multidimensional array is an array containing other arrays, allowing structured data representation in rows and columns.

**Example:**

```
const gameBoard = [
  [0, 1, 2],  // First row
  [3, 4, 5],  // Second row
  [6, 7, 8]   // Third row
];
```

This code creates a game board using a multidimensional array, where each nested array represents a row, and each element represents a square on the board.

# Arrays: Sorting

It refers to the process of arranging the elements of an array in a specified order, either ascending or descending.


Sorting enhances data organization, enabling faster searching, structured representation, and efficient comparisons in JavaScript programming.

# Arrays: Sorting

The syntax of the **sort()** method in JavaScript is as follows:

**Syntax:**

```
arr.sort(compareFunction)
```

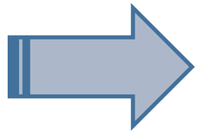
The optional **compareFunction** parameter allows defining a custom sorting order.

# Arrays: Sorting

The sort() method arranges array elements in lexicographic (alphabetical) order by default. For numbers, a compare function is required to ensure numerical sorting.

**Example: sort() method**

```
let cities = ['New York', 'Los Angeles',
'Chicago', 'Houston', 'Phoenix'];

// Sorts the city array in lexicographic
(alphabetical) order
let sortedCities = cities.sort();
console.log(sortedCities);
```



**Output**

```
['Chicago', 'Houston', 'Los
Angeles', 'New York', 'Phoenix']
```

Sorting is case-sensitive unless explicitly modified using localeCompare() for case-insensitive sorting.

# Arrays: Searching

It is a process of finding the position or existence of specific elements within an array.

It helps to locate and retrieve elements based on specific criteria in an array.

# Common Methods for Searching Elements in JavaScript Arrays

JavaScript offers various methods to search arrays, locate values, check existence, or filter elements based on conditions. Below are common array searching methods:

**filter()**: Returns all elements that match a condition

**find()**: Returns the first matching element

**includes()**: Checks if an element exists in the array

**indexOf()**: Finds the index of an element

# Filter Method

The **filter()** method returns a new array containing all elements that pass a test provided by a function.

**Filter: Syntax**

```
//syntax:
array.filter(callback (currentValue, index, arr), thisValue)
```

It does not execute the function for empty elements or change the original array.

# Filter Method: Parameters

The parameters of the filter() method define how each array element is processed, allowing customization based on value, index, and context. They are:

**callback()**: A function executed on each element of the array (required)

**currentValue**: The current element being processed (required)

**index**: The index of the current element within the array (optional)

**arr**: The array on which the filter() method is applied (optional)

**thisValue**: A value to use as this when executing the callback function (optional)

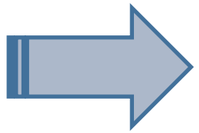
By understanding its parameters, you can customize the filter() method to efficiently extract specific elements from an array based on given criteria.

# Filter Method: Example

It is useful for retrieving a subset of an array that meets a specific condition and returns an array instead of a single element.

**filter() method**

```
const array = [7, 2, 13, 10, 5];

const greaterThanFive = array.filter(element =>
element > 5);

console.log(greaterThanFive)
```



**Output**

```
[ 7, 13, 10 ]
```

The filter() method creates a new array containing elements greater than 5 from the original array, returning [7, 13, 10].

# Find Method

The **find()** method searches an array and returns the first element that matches a given condition. The key features include:

- This function returns the first element that satisfies a condition.
- It does not execute the callback function for empty slots in sparse arrays.
- find() returns undefined if no element matches the condition.

**Find: Syntax**

```
//syntax:
array.find(callback
(currentValue,index,arr), thisValue)
```

The find() method is useful for searching objects in an array, making it ideal for retrieving elements based on complex conditions, such as finding a user by ID in a list of objects.

# Find Method Parameters

The parameters of the find() method determine how the search is conducted, allowing customization based on value, index, and context. They are:


**callback() –** A function executed for each array element until a match is found (required).

**currentValue** – The current element being processed (required).

**index** – The index of the current element (optional).

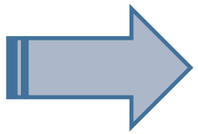
**arr** – The array on which find() is applied (optional).

**thisValue** – A value to use as this inside the callback function (optional).

# Find Method: Example

**find() method**

```
const array = [9, 12, 13, 20, 7];

const greaterThanNum = array.find(element
=> element > 9);

console.log(greaterThanNum)
```



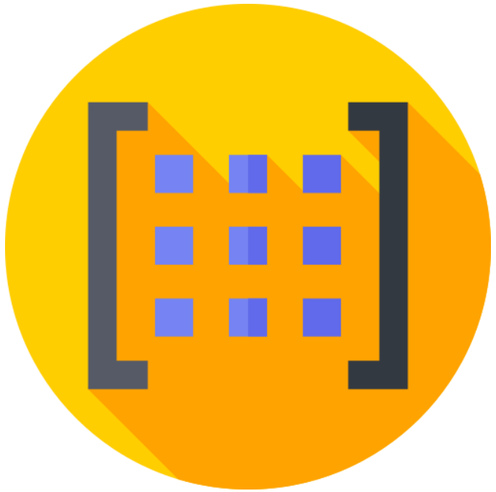
**Output**

```
12
```

The find() method returns only the first matching element, unlike filter(), which returns all matching elements in an array.

# Includes Method

This method provides a concise way to check if an array contains a particular element.



**Find: Syntax**

```
//syntax:
array.includes(searchElement, startIndex)
```

It returns a boolean value (true or false) indicating whether the element is present in the array.

# Includes Method Parameters

The parameters of the includes() method define how the search is performed, specifying the target value and the starting index for the search. They are:

**searchElement:**
The element to search for within the array

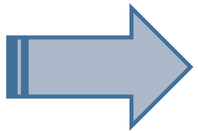
**startIndex:**
The index to start the search from (optional)

includes() checks if an array contains a specific value and returns true or false, unlike methods that use callback functions for conditional checks.

# Includes Method: Example

**includes() method**

```
const array = [9, 12, 13, 20, 7];

const includesNum = array.includes(13);

console.log(includesNum)
```



**Output**

```
true
```

The includes() method checks if the number 13 exists in the array and returns true, indicating its presence.

# IndexOf Method

It is a simple and straightforward method for finding the index of a specific element in an array.

**IndexOf: Syntax**

```
//syntax:
array.indexOf(searchElement, startIndex)
```

It returns the first occurrence of the element, or -1 if not found. The optional startIndex parameter lets you begin searching from a specific position in the array.

# IndexOf Method Parameters

The parameters of the indexOf() method define how the search is performed, specifying the target value and the starting index for the search. They are:

**searchElement:**
The element to search for within the array

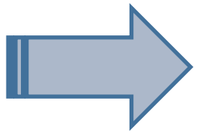
**startIndex:**
The index to start the search from (optional)

indexOf() returns the first occurrence of the specified value and returns -1 if the value is not found. It performs a strict equality check (===), meaning values with different types, such as '4' and 4, will not be considered equal.

# IndexOf Method: Example

**indexOf() method**

```
const array = [9, 12, 13, 20, 7];

const includesNum = array.indexOf(20);

console.log(includesNum)
```



**Output**

```
3
```

The indexOf() method searches for the number 20 in the given array and returns its first occurrence index, which is 3. If the element is not found, it returns -1.

# Searching Methods: Summary

The summary of when to use each method is give below:

Use filter() to return all elements in an array that satisfy a condition.

Use find() to retrieve the first element that satisfies a condition.

Use includes() to determine if an array contains a specific value.

Use indexOf() to return the index of a specific element in an array.

Beyond searching methods, JavaScript provides powerful tools for working with arrays and objects efficiently. One such tool is the spread syntax (...), which simplifies data manipulation.

# Spread Syntax (...)

The spread syntax (...) in ES6 allows iterable elements (arrays, objects, or strings) to be expanded into individual values.

**Spread Syntax (...)**

```
// Syntax:
const newArray = [...iterable];
```

The spread syntax simplifies array merging, shallow copying, and passing multiple arguments to functions efficiently.

# Spread Syntax: Use Cases

Common use cases of spread syntax include:

01 Creates shallow copies of arrays or objects, maintaining references for nested structures.

02 Facilitates merging arrays
**Example**: const merged = [...arr1, ...arr2];

03 Expands an array into function arguments
**Example**: Math.max(...numbers);

04 Merges objects efficiently
**Example**: const newObj = {...obj1, ...obj2};

05 Expands a string into characters
**Example**: const chars = [...'hello’];

# Assisted Practice

**Implementing Advance Array Operations** **Duration: 15 Min.**

**Problem statement:**

You have been asked to create and execute a JavaScript file to demonstrate advanced array operations, including sorting, searching, and using the spread syntax.

**Outcome:**

By the end of this task, you will have successfully implemented sorting algorithms, performed array searches, utilized the spread syntax, and validated operations using console.assert(), ensuring accuracy in handling arrays.

**Note:** Refer to the demo document for detailed steps:
06_Implementing_Advance_Array_Operations

ASSISTED PRACTICE

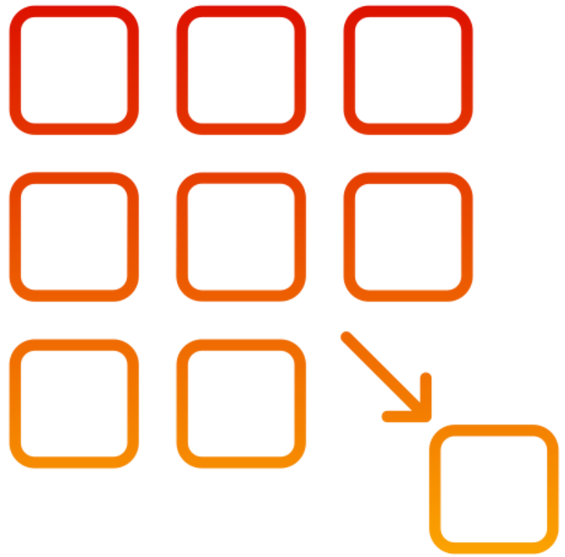
# Assisted Practice: Guidelines

Steps to be followed:

1. Create and execute the JS file

# What Is Mutability in Arrays?

JavaScript arrays support mutability, allowing elements to be modified, added, or removed dynamically.

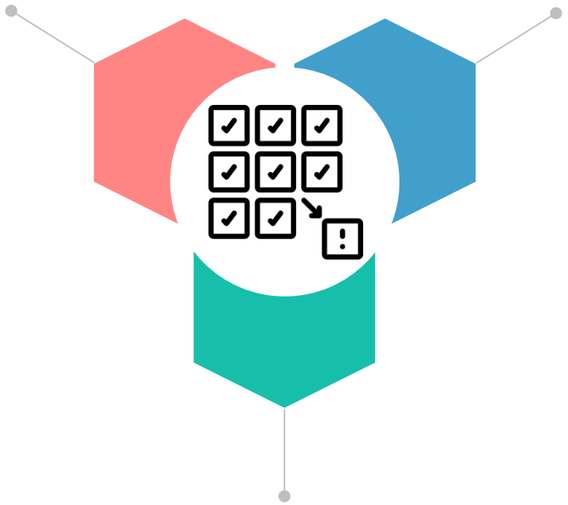


This mutability allows for dynamic modifications to the array's content, such as adding or removing elements, changing values at specific indices, and altering the array's length.

# Mutability in JavaScript Arrays

Key aspects of array mutability include:

Updating existing elements



Adding, removing, and inserting elements

Resizing an array

# Updating Existing Elements

Array elements can be modified directly by referencing their index.

**Example:**

```
let numbers = [1, 2, 3];
numbers[1] = 10;

// Modifying the second element

console.log(numbers);

// Output: [1, 10, 3]
```

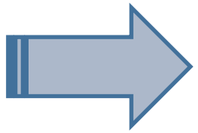
The code demonstrates how to modify an array element using its index. It updates the second element of the numbers array from 2 to 10, showcasing JavaScript's ability to mutate arrays directly.

# Adding and Removing the Elements

The methods push(), pop(), shift(), and unshift() modify an array by adding or removing elements, dynamically adjusting its length.

**Example:**

```
let fruits = ['apple', 'banana'];

// Adding an element at the end
fruits.push('orange');

// Removing the last element
fruits.pop();

// Adding an element at the beginning
fruits.unshift('grape');

// Removing the first element
fruits.shift();

console.log(fruits);
```



**Output:**

```
['apple', 'banana']
```

# Resizing an Array

The length property of an array can be modified to increase or decrease its size. Reducing the length removes elements while increasing it creates empty slots without assigning values.

**Example:**

```
let colors = ['red', 'green', 'blue'];

// Reducing array length
colors.length = 2;
console.log(colors);
// Output: ['red', 'green']

// Increasing array length
colors.length = 5;
console.log(colors);
// Output: ['red', 'green', empty × 3]
```

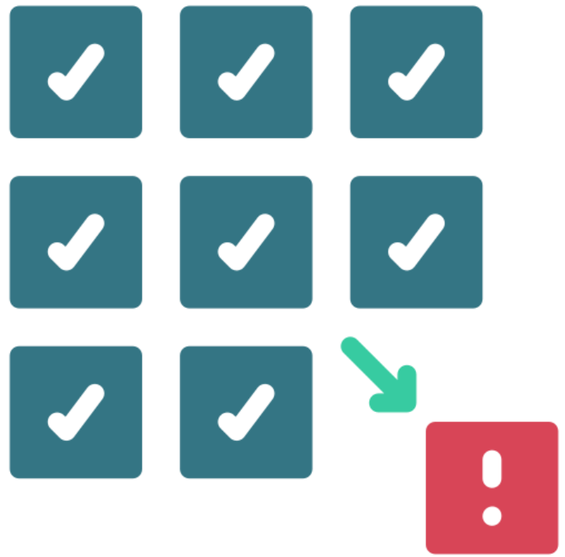
Resizing an array helps manage memory by trimming elements or preparing space for future data without initializing new values.

# Array Mutability: Benefits

The following are key benefits of mutability in JavaScript arrays:

# What Is Immutability in Arrays?

Immutability refers to the inability to modify an object after it is created. In JavaScript, arrays are mutable by default, but immutability can be achieved using specific techniques.



JavaScript arrays allow modifications, but immutability can be achieved by creating new arrays using methods like map(), filter(), or the spread operator (...), ensuring the original array remains unchanged.

# Arrays Immutability

Key aspects of array immutability include:


Ensuring array immutability in JavaScript helps maintain predictable state management, prevent unintended modifications, and improve performance in functional programming.

# Creating New Arrays

In JavaScript, ensuring immutability involves creating a new array instead of modifying the existing one.

**Example:**

```
let users = ['Alice', 'Bob', 'Charlie'];
let updatedUsers = [...users, 'David'];
console.log(updatedUsers);

// Output: ['Alice', 'Bob', 'Charlie', 'David']
```

This approach helps maintain predictable state management, supports functional programming, and avoids unintended side effects.

# Using Non-Mutating Array Methods

Methods like map(), filter(), and concat() return new arrays instead of modifying the original one, ensuring immutability.

**Example:**

```
let numbers = [1, 2, 3, 4, 5];

let evenNumbers = numbers.filter(num => num % 2 === 0);
console.log(evenNumbers);
// Output: [2, 4]

let extendedArray = numbers.concat([6, 7]);
console.log(extendedArray);
// Output: [1, 2, 3, 4, 5, 6, 7]
```

This helps maintain predictable state management, especially in frameworks like React.

# Preventing In-Place Modifications

Using the push() method on an array reference modifies both the reference and the original array. Assigning a new array prevents unintentional modifications and ensures immutability.

**Example:**

```
// Original array
const originalArray = [1, 2, 3, 4, 5];

// Direct mutation (avoided for immutability)
const mutatedArray = originalArray;
mutatedArray.push(6);

console.log(originalArray);
// Output: [1, 2, 3, 4, 5, 6]

console.log(mutatedArray);
// Output: [1, 2, 3, 4, 5, 6] (originalArray is also
modified)
```

push() modifies the original array through references. Use [...] or concat() to prevent unintended changes.

# Preventing In-Place Modifications

Users can achieve immutability by creating a new array.

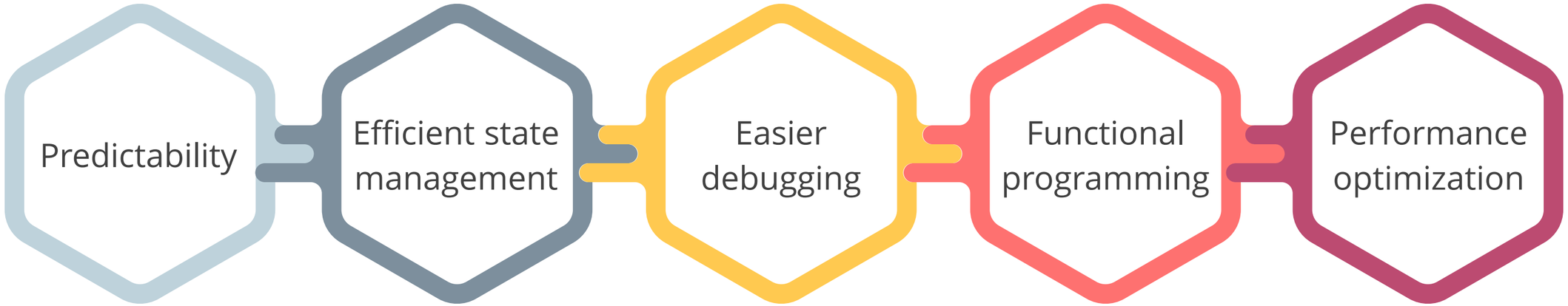
**Example:**

```
// Original array
const originalArray = [1, 2, 3, 4, 5];

// Creating a new array without direct mutation
const newArray = [...originalArray, 6];

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

console.log(newArray);
// Output: [1, 2, 3, 4, 5, 6] (originalArray remains
unchanged)
```

The spread operator (...) creates a new array (newArray) that includes all elements from the originalArray along with the new element (6). This ensures immutability, as the originalArray is not modified directly.

# Array Immutability: Benefits

The following are some benefits of immutability in JavaScript arrays:




Array immutability ensures stable state management, while array destructuring simplifies value extraction for cleaner, more readable code.

# What Is Destructuring of an Array?

It is a powerful feature introduced in ECMAScript 6 (ES6) that allows users to extract values from arrays and assign them to variables in a concise and expressive manner.


This feature simplifies the process of working with arrays and enhances code readability.

# Array Destructuring

The ways of destructuring an array include:

- Using basic syntax
- Skipping elements
- Assigning default values
- Swapping variables

# Array Destructuring

The ways of destructuring an array include:

- Using the rest operator
- Performing nested destructuring
- Ignoring remaining elements
- Destructuring function parameters

# Assisted Practice

**Implementing Array Mutability, Immutability, and Advanced Destructuring** **Duration: 15 Min.**

**Problem statement:**

You have been asked to create and execute a JavaScript file to demonstrate array mutability, immutability, and advanced destructuring techniques.

**Outcome:**

By the end of this task, you will have successfully applied mutable and immutable array operations, utilized advanced destructuring techniques, and validated array transformations to ensure efficient and structured programming.

**Note:** Refer to the demo document for detailed steps:
07_Implementing_Array_Mutability,_Immutability,_and_Advanced_Destructuring

# Assisted Practice: Guidelines

Steps to be followed:

1. Create and execute the JS file

# Quick Check


You are developing a JavaScript application where an array of usernames needs to be updated by adding a new name without modifying the original array. Which method should you use?

A. push()

B. pop()

C. concat()

D. splice()

# Key Takeaways

- JavaScript enables dynamic web interactivity by manipulating HTML and CSS, handling user inputs, and controlling program flow.
- Conditional statements guide decision-making in JavaScript by executing code based on true or false conditions.
- Loops automate repetitive tasks by executing a block of code multiple times until a condition is met.
- JavaScript data types include primitive and non-primitive types, with primitives being immutable and non-primitives being mutable and reference-based.
- Arrays and objects store and manipulate data efficiently, allowing structured data handling in JavaScript applications.

# Thank You