JavaScript Fundamentals

Created by: **Muhammad Mahfuj**
Student: Programming Hero - Batch 10

1 Basics of JavaScript

1.0 Introduction

JavaScript is a scripting or programming language that allows you to implement complex features on web pages. With Javascript one can Change/update elements in web pages, and bring functionality to the website.

JavaScript Mainly Runs on browsers. Where it uses v8 engine to run the actual code. But through node.js Javascript can now also be run outside the browser.
Node.js is a program written in c++ which has a v8 engine that allows it to run Javascript code outside of a browser.

Generally Javascript program is written in a separate .js file, which is then linked to the html code. Generally rule is to put the link at the very bottom of the html body

1.1 Variables

1.1.2 Introduction

Variables are used to store data. We have three ways to declare variables in JavaScript: *var*, *let*, and *const*.

```
// 'var' is function-scoped and Allows re-declaration within the same scope.
var name = "John";

var z = 1;
var z = 2; // No error

// 'let' is block-scoped and Does not allow re-declaration within the same scope.
let age = 25;
```

```
let a = 1;
let a = 2; // SyntaxError: Identifier 'a' has already been declared

// 'const' is also block-scoped, but its value cannot be changed
(immutable).
const pi = 3.14;
```

1.1.3 Variables - Key differences

In JavaScript, var, let, and const are used to declare variables, but they have different behaviors and use cases. Here's a breakdown of the differences:

1. Scope

Definition and Meaning

Scope in programming refers to the **context** or **environment** in which variables, functions, and objects are *accessible* or *visible*. It determines which parts of a program can access or modify a particular variable or function. Essentially, *scope defines the **boundaries** within which variables and functions are declared and where they can be used.*

- **var:** Has **function scope**. It is scoped to the function in which it is declared or globally if declared outside of any function.
- **let:** Has **block scope**. It is scoped to the block ({}) in which it is declared.
- **const:** Also has **block scope**, just like let.

2. Hoisting

Definition and Meaning

Hoisting is a JavaScript mechanism where variable and function declarations are moved ("hoisted") to the top of their containing scope (either function or global scope) during the compile phase, before the code is actually executed. This

means that you can refer to variables or functions before you have actually declared them in your code.

- **var:** Is **hoisted** to the top of its scope (function or global). It is initialized with undefined during the hoisting, so you can reference it before its declaration without getting an error, but the value will be undefined.

```
console.log(x); // undefined
var x = 5;
```

- **let and const:** Are also **hoisted** to the top of their scope, but they are not initialized. They remain in the **Temporal Dead Zone (TDZ)** from the start of the block until the declaration is encountered. Accessing them before their declaration results in a ReferenceError.

```
console.log(y); // ReferenceError: Cannot access 'y' before
initialization
let y = 10;
```

3. Re-declaration

- **var:** Allows **re-declaration** within the same scope.

```
var z = 1;
var z = 2; // No error
```

- **let and const:** Do **not** allow re-declaration within the same scope.

```
let a = 1;
let a = 2; // SyntaxError: Identifier 'a' has already been declare
```

4. Re-assignment

var and let: Both allow **re-assignment** of their values. **const:** Does **not** allow re-assignment. A const variable must be initialized at the time of declaration, and its

value cannot be changed afterward.

```
var b = 3;
b = 4; // Allowed

let c = 5;
c = 6; // Allowed

const d = 7;
d = 8; // TypeError: Assignment to constant variable
```

- **const:** Does **not** allow re-assignment. A const variable must be initialized at the time of declaration, and its value cannot be changed afterward.

```
const d = 7; d = 8; // TypeError: Assignment to constant variable
```

5. Mutability of const

- While const does not allow re-assignment, **objects and arrays** declared with const can still have their **contents mutated** (properties or elements can be changed, added, or removed).

```
const obj = { name: 'John' };
obj.name = 'Doe';
// Allowed - You are changing a property of the object, not the
object reference itself.
obj = {}; // TypeError - You are trying to reassign `obj` to a new
object, which is not allowed for a `const` variable.

const arr = [1, 2, 3];
arr.push(4); // Allowed - You are modifying the array's contents,
not its reference.
arr = [5, 6]; // TypeError - You are trying to reassign `arr` to a
new array, which is not allowed for a `const` variable.
```

6. Global Object Property

- **var:** When declared globally (outside any function), it becomes a property of the global object **(window in browsers | *will not work outside browsers*)**.

```
var foo = 'bar';
console.log(window.foo); // 'bar'
```

- **let and const:** When declared globally, they do not become properties of the global object.

```
let foo = 'bar';
console.log(window.foo); // undefined
```

Summary

- **Use let** for variables that need to be block-scoped and can be reassigned.
- **Use const** for variables that should not be reassigned. This is particularly useful for constants or to prevent accidental reassignments.
- **Use var** sparingly, mostly in legacy codebases, due to its function-scoping and potential for causing unintended side effects. let and const are generally preferred in modern JavaScript development.

1.2 Data Types

JavaScript has *Primitive/Value* and *non-primitive/Reference* data types.
Primitive: In JavaScript, a 'primitive' (or 'primitive value') is a data type that is not an object and has no methods or properties. Primitive types represent the most basic, fundamental types of data that JavaScript can handle.

Non-Primitive: In JavaScript, **non-primitive** types refer to **objects**, which are more complex data structures compared to primitives. Non-primitives are **mutable**, meaning they can be changed after being created, and are **stored by reference**, not by value.

Examples:

```
// Primitive data types

```

```
let string = "Hello, World!";  // String Literal
let number = 42;               // Number Literal
let boolean = true;            // Boolean Literal
let firstName;                 // Undefined
// Undefined is a data type, as well as a value
let firstName = undefined;

let nullVar = null;            // Null
// null is used in situations where we want to explicitly clear the
value of a variable

let symbolVar = Symbol('id');  // Symbol
let bigIntVar = 9007199254740991n; // BigInt

---

// Non-primitive/Reference data type (Object)
Object | Array | Function

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

// Array
let studyTopics = ['Math', 'Arabic', 'Seerah', 'Chemistry',
'Logic']

//Function
function greet(name){
        console.log('Hello' + name);
}
greet('Mahfuj');
```

null use-case:

```
// We may want to present the user with a list of colors. If the
user has no selection, you want to set the
let selectedColor = null;

// In the future, if the user selects a color, then we'll reassign
```

```
the variable to the corresponding color.
let selectedColor = 'red';

// and when they select a color, we could set the variable to null
again. to reset the process
```

1.2.1 Literal Meaning

In general, the term **literal** refers to something that represents exactly what it is—without interpretation or transformation. In the context of programming, a **literal** is a fixed, constant value that you write directly into your code. It represents the actual value itself, not a variable or an expression that needs to be evaluated.

For example, when you write:

```
let number = 42;  // 42 is a number literal
let text = "Hello";  // "Hello" is a string literal`
```

- **Literal** means you're writing the actual value directly: `42` for a number, `"Hello"` for a string. These are not variables or expressions but specific values written exactly as they are.

In short, a **literal** is the exact, raw value written into your code that the computer understands directly.

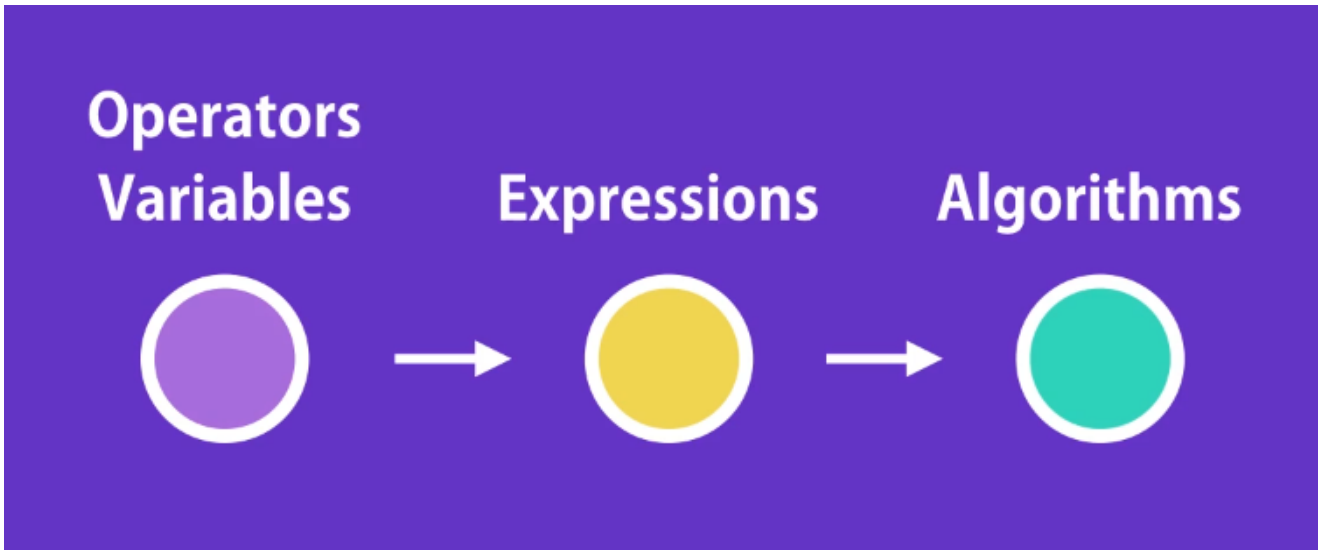
1.3 Operators

In JavaScript, an operator is a symbol or a keyword that performs a specific operation on one or more operands (values or variables). Operators allow you to manipulate and evaluate data.

For example, the addition operator `+` is used to add two or more operands together:

```
let sum = 5 + 3; // sum will be 8
```

Operators are used along with variables and constants to create **expressions**. And with these expressions we can create logic and algorithms.



An **Expression** is any combination of variables, values, and operators that results in a single value. With these expressions, you can build **logic** (rules for decision-making) and **algorithms** (step-by-step instructions to solve a problem).

এক কথায় বলতে গেলে, সূত্র, তার উপাদান এবং তাদের মাঝে সম্পর্ক স্থাপনকারি তথা "সমীকরণ"

Example:

```
let x = 5; // A variable
let y = 10; // Another variable

// An expression using the addition operator (+)
let sum = x + y; // sum is now 15

// An expression using the comparison operator (>)
if (sum > 10) {
  console.log("The sum is greater than 10.");
} else {
  console.log("The sum is 10 or less.");
}
```

In Javascript we have different kinds of operators. Some are given below,

```
Arithmetic | Assignment | Comparison | Logical

// Arithmetic Operators: perform mathematical calculations
let sum = 5 + 3;        // "=" Assigns value | Addition: 8
let difference = 5 - 3; // Subtraction: 2
```

```
// Assignment Operators: assign values
let x = 10;
x += 5;  // x is now 15

// Comparison Operators: compare values
console.log(5 == '5');    // Loose Equality | Value same but
Different type
console.log(5 === 5);   // Strict Equality | Type and Value same
console.log(5 !== 5);   // Strict Inequality; not equal to

// Logical Operators: combine conditions
console.log(true && false); // AND: false
console.log(true || false); // OR: true
```

1.3.1 Operators in detail

1. Arithmetic Operators

- Perform basic mathematical operations.
- Examples:

```
+ (addition),
- (subtraction),
* (multiplication),
/ (division),
% (modulus),
** (exponentiation),
++ (increment),
-- (decrement)
```

2. Assignment Operators

- Assign values to variables.
- Examples:

```
= (assign),
+= (add and assign) | value = value + anotherValue,
-= (subtract and assign) | value = value - anotherValue, ,
```

```
*= (multiply and assign) | value = value * anotherValue,
/= (divide and assign) | value = value / anotherValue,
%= (modulus and assign) | value = value % anotherValue
```

3. Comparison Operators

- Compare two values and return a Boolean (`true` or `false`).
- Examples:

```
== (equal to) | Loose Equality | Value same but could be Different
type,
=== (strict equal to) | Strict Equality | Type and Value same,
!= (not equal to) | compares the values but does not check the
type,
!== (strict not equal to) | Strict Inequality; both type and value,
> (greater than),
< (less than),
>= (greater than or equal to),
<= (less than or equal to)
```

4. Logical Operators

- Perform logical operations.
- Examples:

```
&& (AND) | this and that both,
|| (OR) | either this or that,
! (NOT)
```

5. Bitwise Operators

- Operate on binary representations of numbers. not commonly used or needed.
- Examples:

```
& (AND),
| (OR),
^ (XOR),
```

```
~ (NOT),
<< (left shift),
>> (right shift),
>>> (unsigned right shift)
```

```
// Read, Write, Execute
// 00000100
// 00000010
// 00000001

const readPermission = 4;
const writePermission = 2;
const executePermission = 1;

let myPermission = 0;
myPermission = myPermission | readPermission;

let message = (myPermission & readPermission) ? 'yes' : 'no';
console.log(message);
```

6. String Operators

- Perform operations on strings.
- Examples:

```
+ (concatenation) to join two strings
```

7. Type Operators

- Work with data types.
- Examples:

```
typeof (returns the type of a variable),
instanceof (checks if an object is an instance of a class or
constructor)
```

8. Ternary (Conditional) Operator

- A shorthand for `if...else` statements.
- Example:

```
condition ? expression1 : expression2
// If a customer has more than 100 points
// they are 'gold' customer, otherwise
// they are 'silver' customer

let points = 110;

if (points > 100) {
  return 'gold';
}
else {
  return 'silver';
}

// shorthand for above code
let points = 110;
let customerType = points > 100 ? 'gold' : 'silver';

console.log(customerType); // gold | as there are 110 points.
```

9. Comma Operator

- Allows multiple expressions to be evaluated in a single statement, and returns the value of the last expression.
- Example:

```
(a = 1, b = 2) // Sets both `a` and `b` and returns `2`
```

10. Unary Operators

- Operate on a single operand.
- Examples:

```
typeof (returns the type of a variable),
! (logical NOT),
++ (increment),
-- (decrement),
delete (deletes a property from an object),
void (evaluates an expression without returning a value)
```

11. Relational Operators

- Determine the relationship between two values.
- Examples:

```
in (checks if a property exists in an object),
instanceof (checks if an object is an instance of a particular
constructor)
```

12. Nullish Coalescing Operator (`??`)

- Returns the right-hand operand when the left-hand operand is `null` or `undefined`; otherwise, it returns the left-hand operand.
- Example:

```
let name = null;
let greeting = name ?? "Guest"; // greeting will be "Guest"
```

13. Optional Chaining Operator (`?.`)

- Allows safe access to deeply nested properties without having to check if each reference in the chain is `null` or `undefined`.
- Example:

```
let user = {};
console.log(user?.profile?.name); // undefined (doesn't throw an
```

```
error)
```

> **Summary**
>
> JavaScript has a wide range of operators that serve different purposes, from basic arithmetic to more advanced operations like type checking, object manipulation, and logical control flow.

---

2. Control Structures

2.1 Conditional Statements

Conditional statements are used in JavaScript to perform different actions based on different conditions. **They allow the code to make *decisions* and *execute specific blocks of code* based on whether a *condition is* `true` or `false`.**

=> There are mainly two types of conditional statements in Javascript
- *if-else statement*
- *switch statement*

```
// if-else statement
let num = 10;

if (num > 0) {
  console.log("Positive number");
}
else {
  console.log("Zero"); // Positive number
}

// switch statement: an alternative to multiple 'if' conditions
let color = "red";
switch (color) {
  case "red":
    console.log("Stop");
    break;
  case "yellow":
    console.log("Ready");
```

```
    break;
  case "green":
    console.log("Go");
    break;
  default:
    console.log("Invalid color");
}
```

2.1.2 Conditionals Statements *Detailed*

1. `if` Statement

The `if` statement is used to specify a block of code that will be executed if a specified condition is `true`.

Example:

```
let age = 18;
if (age >= 18) {
    console.log("You are eligible."); // You are eligible.
}
```

2. `else` Statement

The `else` statement specifies a block of code that will be executed if the same condition is `false`.

Example:

```
let age = 16;
if (age >= 18) {
    console.log("You are eligible.");
} else {
    console.log("You are not eligible."); // You are not eligible.
}
```

3. `else if` Statement

The `else if` statement is used to specify a new condition to test if the previous condition(s) were `false`.

Example:

```
let score = 75;
if (score >= 90) {
    console.log("Grade: A");
} else if (score >= 75) {
    console.log("Grade: B");
} else {
    console.log("Grade: C");
}
```

another example: `else-if` statement checks multiple conditions and executes the first block of code that evaluates to true.

```
let score = 85;
if (score >= 90){
        console.log("Grade: A");
}
else if (score >= 80){
        console.log("Grade: B");
}
else if (score >= 70){
        console.log("Grade: C");
}
else{
        console.log("Grade: D");
}
```

Note: All the `else if` will go inside `if-else`

4. `switch` Statement

The `switch` statement is used to perform different actions based on different conditions. It evaluates an expression and matches its value against several `case` clauses.

Example:

```
let fruit = "apple";
switch (fruit) {
    case "banana":
```

```
        console.log("Banana is $1.");
        break;
        /* if break is not used then it will just execute all other
        cases/code below, regardless they fulfill the condition or
not */
    case "apple":
        console.log("Apple is $2.");
        break;
    case "orange":
        console.log("Orange is $1.5.");
        break;
    default:
        console.log("Sorry, we are out of that fruit.");
}
```

Another example:

```
let testScore = 92;
let letterGrade;

switch(true){
        case testScore >= 90:
                letterGrade = 'A';
                break;
}
console.log(letterGrade); // Outputs: "A"
```

`switch(true)` construct is used to evaluate expressions that result in `true` or `false`. This allows the `switch` statement to behave similarly to a series of `if-else` statements.

Why `switch(true)`?

- `switch (true)`: The `switch` statement is initialized with the `true` value. This means each `case` condition is checked to see if it evaluates to `true`.
- `switch(true)`: We are using `true` as the constant reference value to check each condition against. We do this when we want to evaluate multiple conditions that are not simple value matches but logical comparisons or boolean expressions.

- `case testScore >= 90:` : This checks if `testScore` is greater than or equal to `90`. If it is, this expression will evaluate to `true`, and the corresponding block (`letterGrade = 'A';`) will execute.
- **Standard `switch`** : No `true` or condition is needed because it compares a single value (like `"apple"`) to several possible values (`"banana"`, `"apple"`, etc.).

When to Use `switch(true)`

Use `switch(true)` when:

- You need to evaluate multiple conditions or expressions that may not be simple value matches.
- You want to use a `switch` statement in a way that resembles a series of `if-else` statements for readability or organization.

How It Works

- The `switch` statement compares `true` (from `switch(true)`) with the result of each `case` expression.
- If `case testScore >= 90` evaluates to `true`, it matches the `true` in `switch(true)`, so the code inside that `case` block runs.

A More Complete Example

```
let testScore = 92;
let letterGrade;

switch (true) {
    case testScore >= 90:
        letterGrade = 'A';
        break;
    case testScore >= 80:
        letterGrade = 'B';
        break;
    case testScore >= 70:
        letterGrade = 'C';
        break;
    case testScore >= 60:
        letterGrade = 'D';
```

```
        break;
    default:
        letterGrade = 'F';
}
console.log(letterGrade); // Outputs: "A"
```

Summary

- `switch (true)` is a way to use the `switch` statement for conditional logic, similar to using multiple `if-else` statements.
- This approach is useful when you want to check multiple conditions that may not all be mutually exclusive.

Difference between if-else and switch

SWITCH = can be an efficient replacement to many else if statements.
Cleaner code but a bit complicated.

```
let day = 2;

if(day == 1){
    console.log(`Monday`);
}
else if(day == 2){
    console.log(`Tuesday`);
}
else if(day == 3){
    console.log(`Wednesday`);
}
else if(day == 4){
    console.log(`Thursday`);
}
else if(day == 5){
    console.log(`Friday`);
}
else if(day == 6){
    console.log(`Saturday`);
}
else if(day == 7){
    console.log(`Sunday`);
```

```
}
else {
    console.log(`Not a weekday`);
}
// It is Tuesday
```

```
let day = 2;
switch (day) {
    case 1:
        console.log(`Monday`);
        break;
    case 2:
        console.log(`Tuesday`);
        break;
    case 3:
        console.log(`Wednesday`);
        break;
    case 4:
        console.log(`Thursday`);
        break;
    case 5:
        console.log(`Friday`);
        break;
    case 6:
        console.log(`Saturday`);
        break;
    case 7:
        console.log(`Sunday`);
        break;

    default:
    console.log(`Not a weekday`);
}
```

2.2 Loops

Loops in JavaScript allow us to execute a block of code repeatedly based on a condition. They help automate repetitive tasks.

In Javascript There are Five types of loops. They essentially do the same thing but have subtle differences on how they start and end.

- for
- while
- do...while
- for...of
- for...in

```
// for loop
for (let i = 0; i < 5; i++) {
  console.log(i);  // Output: 0 1 2 3 4
}

// while loop
let i = 0;
while (i < 5) {
  console.log(i);  // Output: 0 1 2 3 4
  i++;
}

// do...while loop
let j = 0;
do {
  console.log(j);  // Output: 0 1 2 3 4
  j++;
} while (j < 5);

// for...of loop
let array = ['apple', 'banana', 'cherry'];
for (let fruit of array) {
        console.log(fruit); // Outputs: 'apple', 'banana', 'cherry'
}

// for...in
let person = { name: 'John', age: 30, city: 'New York' };

for (let key in person) {
  console.log(key + ": " + person[key]);
}
// Outputs:
// name: John
// age: 30
// city: New York
```

`for` loop

The `for` loop repeats a block of code a specific number of times. It consists of three parts: **initialization**, **condition**, and **increment/decrement**.

Syntax:

```
for (initialization; condition; increment/decrement) {
  // code to execute
}
```

Example:

```
for (let i = 0; i < 5; i++) {
  console.log(i);  // Outputs: 0, 1, 2, 3, 4
}

for (let i = 0; i <= 5; i++){
        if (i % 2 !== 0) console.log(i); // Outputs: 1, 3, 5
}
```

`while` Loop

The `while` loop continues to execute a block of code as long as the specified condition is `true`.

Syntax:

```
while (condition) {
  // code to execute
}
```

Example:

```
let i = 0;
while (i < 5) {
```

```
  console.log(i);  // Outputs: 0, 1, 2, 3, 4
  i++;
}

let i = 0;
while (i <= 5){
        if (i % 2 !== 0) console.log(i); // Outputs: 1, 3, 5
        i++
}
```

`do...while` Loop

The `do...while` loop is similar to the `while` loop, but it guarantees that the code block is executed **at least once**, as the condition is checked after the code runs.

Syntax:

```
do {
  // code to execute
} while (condition);
```

Example:

```
let i = 0;
do {
  console.log(i);  // Outputs: 0, 1, 2, 3, 4
  i++;
} while (i < 5);


let i = 0;
do {
        if (i % 2 !== 0) console.log(i); // Outputs: 1, 3, 5
        i++
} while (i <=5);

let i = 9;
do {
        if (i % 2 !== 0) console.log(i); // Outputs: 9
        i++
// Code is executed once with the set value before terminating at
the condition
```

```
} while (i <=5);
```

`for...of` Loop

The `for...of` loop iterates over the values of an iterable object, such as arrays, strings, or other collections.

Syntax:

```
for (let element of iterable) {
        // code to execute
}
```

Example:

```
let array = ['apple', 'banana', 'cherry'];

for (let fruit of array) {
  console.log(fruit);  // Outputs: 'apple', 'banana', 'cherry'
}
```

`for...in` Loop

The `for...in` loop iterates over the **properties** of an object (or the indexes in an array).

Syntax:

```
for (let key in object) {
  // code to execute
}
```

```
let person = { name: 'John', age: 30, city: 'New York' };

for (let key in person) { // here 'key' is also a variable
  console.log(key + ": " + person[key]);
}
// Outputs:
```

```
// name: John
// age: 30
// city: New York

// person[key] is bracket notation
// it is used when we don't know the name of the property
beforehand

// person.key is dot notation
// it is used when we know the name of the property beforehand
```

Infinite Loop

A loop which continues infinitely. It could crash your system or browser.

```
let i = 0;
while (i < 5) {
        console.log(i);
        // i++;
}

/* Suppose you forgot to put the increment after the console.log
now in every instance i < 5 as i = 0; and it won't have a way to
break the loop.
So it goes on inifintely */

let i =0;
while (i > 0){
        console.log(i);
        i++;
}
// here it is a logic error

// this can happen in other loops as well.
// So always be careful before executing a loop
```

`break` and `continue`

- `break` : Exits the loop completely, regardless of the loop's condition.

- `continue` : Skips the current iteration and continues with the next one. (it is not used that much in practical work)

Example:

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

// break
for (let i = 0; i < array.length; i++) {
  if (array[i] === 3) {
    break;  // Loop stops when 3 is found
  }
  console.log(array[i]);  // Outputs: 1, 2
}

// continue
for (let i = 0; i < array.length; i++) {
  if (array[i] === 3) {
    continue;  // Skip the iteration when 3 is found
  }
  console.log(array[i]);  // Outputs: 1, 2, 4, 5
}
```

3. Functions

A **function** in JavaScript is a reusable block of code that is designed to perform a specific task. Functions take in inputs (called **parameters**) and can return outputs (**results**). Functions help in organizing code, making it more readable, modular, and **reusable**.

There are different ways to define and use functions in JavaScript. Here are the main types:

3.1 Function Declaration

A **function declaration** defines a named function that can be invoked later in the code. It is "hoisted," meaning it can be called even before its declaration in the code.

```
// Function Declaration
function greet(name) {
```

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

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

- The function has a name (`greet`) and can be **invoked** (*called*) anywhere after or before its declaration.

3.2 Function Expression

A **function expression** defines a function as part of an expression (usually assigned to a variable). Function expressions are not hoisted, so they cannot be called before they are defined.

```
const greetExpression = function(name) {
  return Hello, ${name}!;
};

console.log(greetExpression("Mahfuj"));  // Output: Hello, Mahfuj!
```

- The function is **anonymous** (*does not have a name*) and is assigned to a variable (`greetExpression`).

3.3 Arrow Functions

Arrow functions are a shorthand for writing functions.

```
// Arrow Function
const greetArrow = (name) => Hello, ${name}!;

console.log(greetArrow("Charlie"));  // Output: Hello, Charlie!
```

Note: Arrow functions do not have their own `this` value, which makes them behave differently in certain contexts (especially when used with objects).

Example:

```
const person = {
    name: "Alice",
    regularFunction: function() {
        console.log(this.name); // 'this' refers to the person
object
    },
    arrowFunction: () => {
        console.log(this.name); // 'this' is inherited from the
surrounding scope
    }
};

person.regularFunction(); // Output: Alice
person.arrowFunction();   // Output: undefined (or an error)
```

Difference Between Arrow Functions and Regular Functions

- **Arrow Functions**: Typically need to be assigned to variables or used directly in expressions (e.g., as callbacks).
- **Regular Functions**: Can be declared using the `function` keyword and exist independently. They are "hoisted," meaning you can call them before their declaration in the code.

Exception:
You can also use an arrow function directly as a callback without assigning it to a variable:

```
setTimeout(() => {
    console.log("This runs after 2 seconds!");
}, 2000);
```

Summary: arrow functions are expressions and need to be assigned or used directly in an expression, while regular functions can be declared independently using the `function` keyword.

3.4 Immediately Invoked Function Expression (IIFE)

An **IIFE** is a function that is defined and immediately executed. It is useful when you need to run some code right away without cluttering the global namespace.

```
(function() {
    console.log("This function runs immediately!");
})();
```

- **Main Characteristics**: Executed immediately after being defined. Helps avoid polluting the global scope.

3.5 Anonymous Function

An **anonymous function** is a function without a name. It is usually used when a function is passed as an argument to another function or event handler.

```
document.getElementById('btn').addEventListener('click', function()
{
    console.log("Button clicked!");
});
```

- **Main Characteristics**: The function does not have a name and is typically used as an inline function.

3.6 Constructor Function

A **constructor function** is used to create objects. When called with the `new` keyword, it creates a new instance of the object.

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}

const person1 = new Person("David", 25);
console.log(person1.name); // Output: David
```

- **Main Characteristics**: Used to create new objects with properties and methods.

3.7 Generator Function

A **generator function** is a special type of function that can pause execution (`yield`) and resume later. It is declared using the `function*` syntax.

```
function* numberGenerator() {
    yield 1;
    yield 2;
    yield 3;
}

const gen = numberGenerator();
console.log(gen.next().value); // Output: 1
console.log(gen.next().value); // Output: 2
```

- **Main Characteristics**: Can pause and resume execution. Useful for working with asynchronous operations.

3.8 Callback Function

A **callback function** is passed as an argument to another function and is executed after the completion of that function.

```
function greet(name, callback) {
    console.log("Hello, " + name);
    callback();
}

function sayGoodbye() {
    console.log("Goodbye!");
}

greet("Emma", sayGoodbye);
// Output:
// Hello, Emma
// Goodbye!
```

- **Main Characteristics**: Used in asynchronous operations such as event handling or server requests.

3.9 Higher-Order Function

A **higher-order function** is a function that takes another function as an argument, returns a function, or both.

```
function multiplier(factor) {
    return function(number) {
        return number * factor;
    };
}

const double = multiplier(2);
console.log(double(5)); // Output: 10
```

- **Main Characteristics**: Takes a function as an argument or returns a function. Used for functional programming patterns.

3.10 Async Function

An **async function** is a function that always returns a `Promise`. It allows asynchronous, promise-based behavior to be written in a cleaner, more readable way using `async` and `await` keywords.

```
async function fetchData() {
    let response = await fetch('https://api.example.com/data');
    let data = await response.json();
    console.log(data);
}

fetchData();
```

- **Main Characteristics**: Simplifies working with asynchronous code by allowing the use of `await` for promise resolutions.

Summary:

- **Function Declaration**: Defines a named function.
- **Function Expression**: Defines a function as part of an expression.
- **Arrow Function**: Concise syntax for functions.
- **IIFE**: A function that is invoked immediately after it's defined.
- **Anonymous Function**: Function without a name, usually used as a callback.
- **Constructor Function**: Used to create objects.
- **Generator Function**: Allows pausing and resuming execution.
- **Callback Function**: Function passed as an argument to another function.
- **Higher-Order Function**: Function that takes or returns another function.
- **Async Function**: Function that handles asynchronous code with promises.

These types of functions give you the flexibility to structure your code in a modular, reusable, and efficient way.

---

5 Arrays

An **array** in JavaScript is a special type of object that is used to store multiple values in a single variable. Arrays can hold any type of data—numbers, strings, objects, or even other arrays. They are ordered collections, and each element in an array has a numbered index, starting from `0`.

```
// Creating an array
let fruits = ["apple", "banana", "cherry"];

// Accessing array elements
console.log(fruits[1]);  // Output: banana

// Adding an element to the array
fruits.push("date");
console.log(fruits);  // Output: ["apple", "banana", "cherry",
"date"]
```

5.1 Single-Dimensional Arrays (Basic Arrays)

- The most common type of array in JavaScript, consisting of a single list of elements.
- Elements are accessed using a numerical index starting from `0`

```
const fruits = ["apple", "banana", "cherry"];
console.log(fruits[0]); // Output: apple
```

Explanation: This is a simple array with three strings. The element at index `0` is `"apple"`.

5.2 Multidimensional Arrays (Nested Arrays)

- Arrays that contain other arrays as elements. This is useful when you want to represent a matrix or grid.
- Elements are accessed using multiple indices.

```
const matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];
console.log(matrix[1][2]); // Output: 6
```

Explanation: This is a 2D array (or matrix), where the element at the second row and third column is `6`.

5.3 Sparse Arrays

- Arrays that have gaps between their indices. In a sparse array, some elements are missing.
- JavaScript automatically assigns `undefined` to the missing elements.

```
const sparseArray = [1, , 3]; // Missing element at index 1
console.log(sparseArray[1]);  // Output: undefined
```

Explanation: The second element in the array is missing, and JavaScript treats it as `undefined`.

5.4 Array-Like Objects

- These are objects that look like arrays but are not arrays. They have indexed elements and a `length` property but lack array methods like `push()`.
- You can convert them to real arrays using `Array.from()`.

```
function myFunction() {
    console.log(arguments); // Array-like object
    const args = Array.from(arguments); // Convert to array
    console.log(args);
}
myFunction(1, 2, 3); // Output: [1, 2, 3]
```

Explanation: `arguments` is an array-like object that can be converted into an array using `Array.from()`.

Common Array Methods

push(): Adds an element to the end of the array.

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

pop(): Removes the last element from the array.

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

shift(): Removes the first element from the array.

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

unshift(): Adds an element to the beginning of the array.

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

forEach(): Calls a function for each array element.

```
const arr = [1, 2, 3];
arr.forEach(element => console.log(element));
// Output: 1 2 3
```

map(): Creates a new array with the results of calling a function on every array element.

```
const arr = [1, 2, 3];
const doubled = arr.map(num => num * 2);
console.log(doubled); // Output: [2, 4, 6]
```

Summary of Array Types and Methods

- **Single-Dimensional Arrays**: Basic arrays with a single list of elements.
- **Multidimensional Arrays**: Arrays that contain other arrays.
- **Sparse Arrays**: Arrays with gaps between elements, treated as `undefined`.
- **Array-Like Objects**: Objects with a `length` property and indexed elements but without array methods.

Array methods like `push()`, `pop()`, `shift()`, and `map()` allow you to manipulate arrays in JavaScript easily.

---

4 Objects

In JavaScript, an **object** is a collection of properties and methods. Each property is a key-value pair, and methods are functions that can be associated with an object. Objects are one of the core concepts in JavaScript, as everything in JavaScript (except for primitive types) is an object, including arrays, functions, and even other objects.

4.1 Plain Object (Object Literals)

- The most common type of object in JavaScript.
- Created using curly braces `{}`.
- Properties are key-value pairs, and methods are functions defined within the object.

```
// Creating an object
let person = {
  name: "Alice",
  age: 30,
  greet() {  // Method inside an object
    console.log(Hello, my name is ${this.name});
  }
};

console.log(person.name);  // Accessing object property: Output - Alice
person.greet();  // Calling object method: Output - Hello, my name is Alice
```

4.2 Constructor Functions

- A blueprint for creating multiple objects of the same type.
- Typically used with the `new` keyword to create instances.

```
function Car(make, model, year) {
    this.make = make;
    this.model = model;
    this.year = year;
}
```

```
const myCar = new Car("Toyota", "Corolla", 2021);
console.log(myCar.make);  // Output: Toyota
```

Explanation: `Car` is a constructor function used to create new `Car` objects. `myCar` is an instance of `Car` with specific values.

4.3 Built-In Objects

- JavaScript comes with several built-in objects like `Array`, `Date`, `Math`, `RegExp`, etc.
- These objects have predefined methods and properties

```
const today = new Date();
console.log(today.toDateString());  // Output: Wed Sep 13 2024
```

Explanation: `Date` is a built-in object that allows you to work with dates and times.

4.4 Objects Created Using `Object.create()`

`Object.create()` allows you to create a new object with a specified prototype.

```
const animal = {
    speak() {
        console.log(`${this.name} makes a noise.`);
    }
};

const dog = Object.create(animal);
dog.name = "Rex";
dog.speak();  // Output: Rex makes a noise.
```

Explanation: `dog` is created with `animal` as its prototype, meaning it inherits the `speak` method from `animal`.

4.5 Singleton Objects

- A single instance of an object that can be reused.
- Created using object literals, they represent one instance of an object that you may need in your code.

```
const appConfig = {
    appName: "MyApp",
    version: "1.0.0"
};
console.log(appConfig.appName);  // Output: MyApp
```

Explanation: `appConfig` is a singleton object that stores the configuration of an application.

4.6 Prototype-Based Objects

- Every JavaScript object has a prototype, and objects can inherit properties and methods from their prototype.
- You can add methods or properties to an object's prototype.

```
function Person(name) {
    this.name = name;
}

Person.prototype.sayHello = function() {
    console.log(`Hello, my name is ${this.name}`);
};

const john = new Person("John");
john.sayHello();  // Output: Hello, my name is John
```

Explanation: `sayHello` is added to the prototype of `Person`, so all instances of `Person` can use this method.

4.7 Class-Based Objects

- JavaScript classes provide a cleaner syntax for creating objects.
- Classes are a template for creating objects with shared properties and methods.

```
class Animal {
    constructor(name, sound) {
        this.name = name;
        this.sound = sound;
    }
    speak() {
        console.log(`${this.name} says ${this.sound}`);
    }
}

const dog = new Animal("Dog", "Woof");
dog.speak();  // Output: Dog says Woof
```

Explanation: `Animal` is a class that defines properties and methods. The `dog` object is created using the `Animal` class.

Summary of Object Types

- **Plain Objects (Object Literals)**: Objects created with curly braces `{}` and key-value pairs.
- **Constructor Functions**: Functions that act as blueprints for creating multiple objects.
- **Built-In Objects**: Predefined objects like `Array`, `Date`, `Math`, etc.
- `Object.create()` : Method to create new objects with a specified prototype.
- **Singleton Objects**: A single instance of an object used across the application.
- **Prototype-Based Objects**: Objects inheriting properties and methods through prototypes.
- **Class-Based Objects (ES6)**: Newer syntax for creating objects using classes.

Each of these object types has its specific use cases and benefits depending on what you're trying to achieve.

DOM and Events

DOM

DOM is an interface that browsers implement to represent the structure of a web document. It allows programming languages like JavaScript to access and manipulate the content, structure, and styles of web pages dynamically.

Key Points:

- The DOM represents the document as a tree of nodes, where each node corresponds to a part of the document (elements, attributes, text, etc.).
- JavaScript can interact with the DOM to change the document structure, update content, and respond to user actions.

```
// Accessing an element by ID
const heading = document.getElementById('myHeading');

// Changing the text content
heading.textContent = 'New Heading Text';

// Adding a class to the element
heading.classList.add('highlight');
```

Common DOM Methods:

```
getElementById(): // Retrieves an element by its ID.
getElementsByClassName(): // Retrieves a collection of elements by
their class name.
getElementsByTagName(): // Retrieves a collection of elements by
their tag name.
querySelector(): // Retrieves an element by its CSS selector.
querySelectorAll(): // Retrieves a collection of elements by their
CSS selector.
```

Events

Events are actions or occurrences that happen in the browser window, which can be detected and responded to using JavaScript. Events allow developers to create interactive web applications by responding to user actions.

Key Points:

- Common events include clicks, mouse movements, keyboard input, form submissions, and more.
- JavaScript uses event listeners to execute code in response to events.

```
// Adding a click event listener to a button
const button = document.getElementById('myButton');
button.addEventListener('click', function() {
    alert('Button was clicked!');
});
```

Summary

- The **DOM** allows JavaScript to interact with HTML and CSS, making web pages dynamic and interactive.
- **Events** enable developers to respond to user actions and other occurrences in the browser, enhancing user experience.