Javascript

History of JavaScript

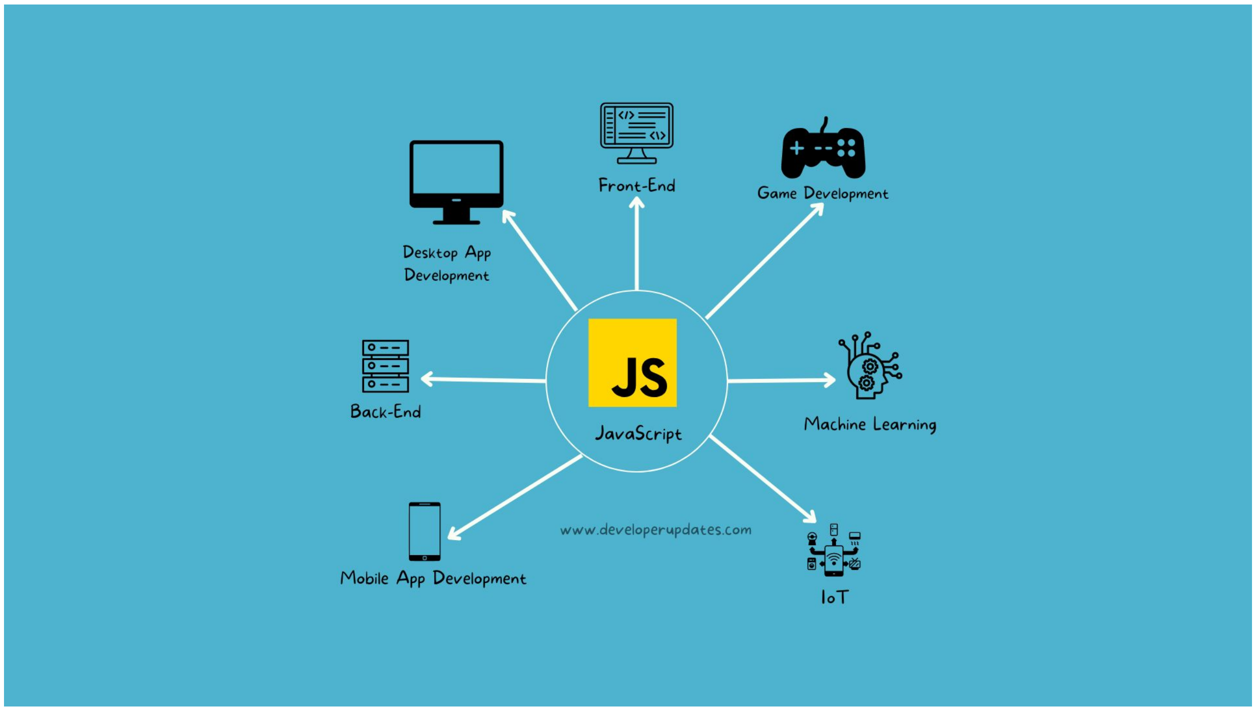
Origins of JavaScript:

JavaScript's roots lie in the early days of the World Wide Web. It was developed by Brendan Eich at Netscape Communications in 1995, originally named Mocha and then LiveScript. It was quickly renamed to JavaScript (inspired by Java's popularity at the time) and became a key part of Netscape Navigator, one of the leading web browsers at the time. JavaScript's initial goal was to add interactivity and dynamic behavior to web pages, making them more engaging for users.

Evolution:

- ECMAScript 2.0: Released in 1998, this version made minor modifications to comply with ISO standards.
- ECMAScript 3: Released in 1999, this version introduced regular expressions and exception handling.
- ECMAScript 5: Released in 2009, this version included new features, such as the ability to pair with JSON files.
- ECMAScript 6: Released in 2015, this version was renamed ECMAScript 2015.
- ECMAScript 2022: The latest version of JavaScript, released later in 2022.

JS role in modern world:



JavaScript has become indispensable for modern web development. It's used for a wide range of purposes,

Front-end development: Creating interactive user interfaces, handling events, animations, and manipulating the Document Object Model (DOM).

Back-end development: Node.js allows running JavaScript on servers, enabling server-side logic, APIs, and backend applications.

Mobile app development: Frameworks like React Native and Ionic allow building cross-platform mobile apps using JavaScript.

Game development: Engines like Phaser and Pixi.js provide tools for creating interactive web games.

JavaScript Features

Dynamic and Loosely-Typed

Dynamic Typing: JavaScript doesn't require you to explicitly declare the data type of a variable. The interpreter infers the type at runtime based on the assigned value. This allows for flexible and concise code, but it can also lead to potential errors if you're not careful.

```
let message = "Hello"; // String
message = 42;          // Now an integer
```

Loose Typing: JavaScript is loosely typed, meaning it allows variables to hold different data types without explicit type coercion. This flexibility can be convenient, but it's important to understand how type conversions happen implicitly.

```
let age = "30"; // String
let total = age + 10; // Implicit type conversion, total become
```

Event-driven, Non-Blocking I/O

- **Event-driven:** JavaScript relies on events to trigger code execution. This model allows for asynchronous execution, where different parts of the code can run independently without blocking each other.
- **Non-blocking I/O:** JavaScript doesn't wait for I/O operations (like fetching data from a server) to complete before moving on. It registers callbacks or promises to be notified when the I/O operation finishes. This significantly improves performance and allows for responsiveness in user interfaces.

```
document.addEventListener('click', function(event) {
console.log("A click happened!");
});
```

Benefits:

- **Responsiveness:** UI remains interactive even during lengthy operations.
- **Efficiency:** Multiple tasks can run concurrently without blocking each other.

- **Scalability:** Handles large volumes of requests efficiently.

Asynchronous Programming (Promises, async/await)

- **Promises:** Promises are objects that represent the eventual completion (or failure) of an asynchronous operation. They provide a standardized way to handle asynchronous results and manage potential errors.
- **async/await:** This syntactic sugar simplifies working with promises. async functions allow you to write asynchronous code that looks like synchronous code, making it easier to read and manage.
  The await keyword pauses the execution of an async function until a promise resolves.

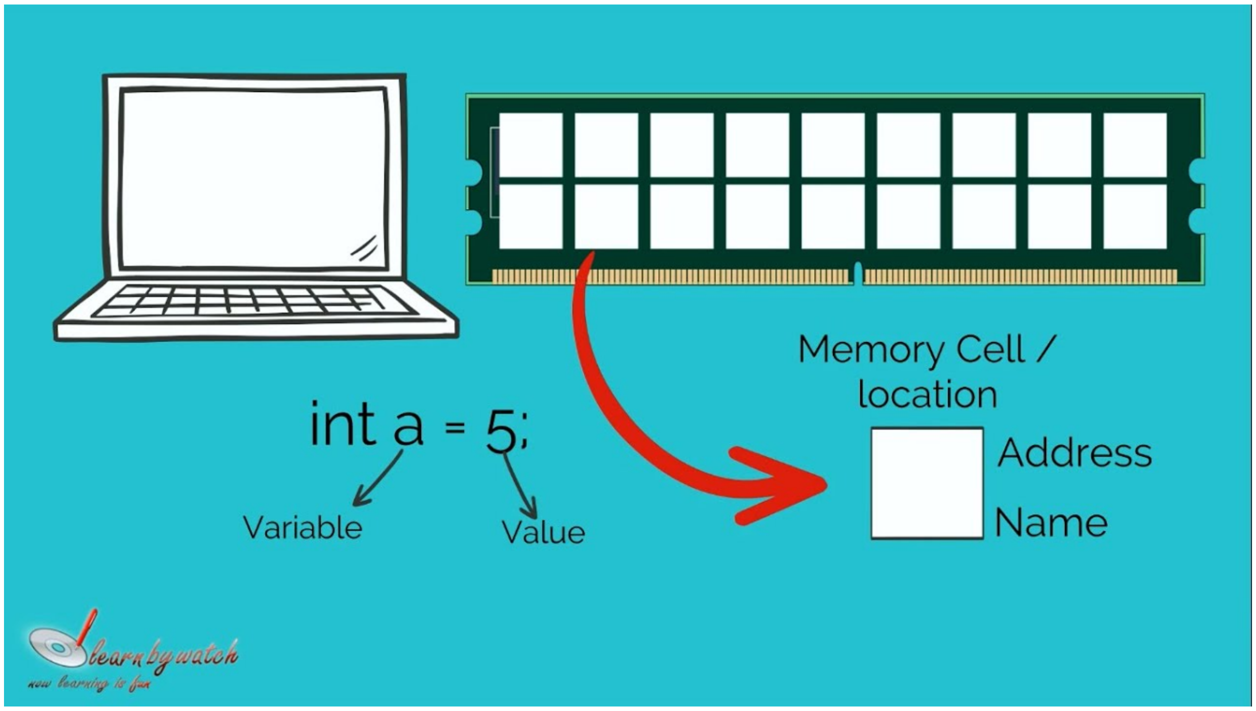
```
async function fetchData() {
    try {
        const response = await fetch('https://api.example.com/da
        const data = await response.json();
        console.log(data);
    } catch (error) {
        console.error('Error fetching data:', error);
    }
}

fetchData();
```

JavaScript's **dynamic, loosely-typed, event-driven,** and asynchronous nature make it a powerful and versatile language for building modern web applications. Understanding these core features is crucial for writing effective and efficient JavaScript code.

Variables

What is variable ?



Variables are the fundamental building blocks of any programming language, allowing you to store and manipulate data. In JavaScript, we use three keywords to declare variables: var, let, and const. Each has its unique characteristics and implications for how the variable behaves within your code.

Declaring Variables: var, let, and const

Feature	var	let	const
Scope	Function scope	Block scope	Block scope
Redeclaration	Allowed	Not allowed	Not allowed
Reassignment	Allowed	Allowed	Not allowed
Default Value	undefined	undefined	undefined

Hoisting and Scope

- Hoisting is a JavaScript behavior where variable declarations (using var) are moved to the top of their scope **before** the code executes.
- However, only the declaration is hoisted, not the assignment.

- This means you can use a var variable before its declaration in the code, but its value will be undefined.

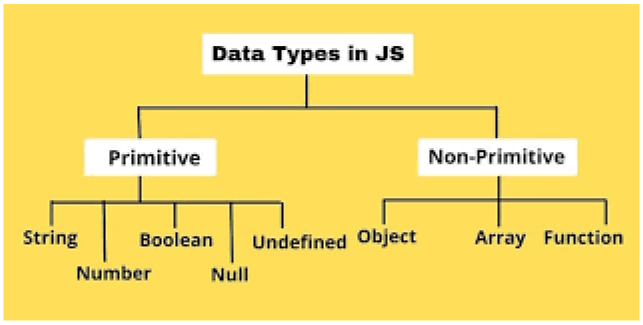
```
console.log(myVariable); // Output: undefined
var myVariable = "Hello";
```

Block Scope vs. Function Scope:

```
function myFunction() {
    var myVar = "Hello"; // Function scope
    if (true) {
        let myLet = "World"; // Block scope
        console.log(myLet); // Output: World
    }
    // console.log(myLet); // Error: myLet is not defined outsi
    console.log(myVar); // Output: Hello
}
```

Note: Using let and const is generally preferred as they promote better code structure and prevent common errors associated with var.

Data Types



1. Primitive Types

These are the fundamental building blocks of data in JavaScript. They represent single values.

- **string:** A sequence of characters enclosed in single ('...') or double ("...") quotes.

  ```
  const name = "Alice";
  const message = 'Hello, world!';
  ```

- **number:** Represents numerical values, including integers and decimals.

  ```
  const age = 30;
  const price = 19.99;
  ```

- **boolean:** Represents truth values: true or false.

  ```
  const isLoggedIn = true;
  const isAvailable = false;
  ```

- **null:** Represents the intentional absence of a value.

  `let user = null; // Initially no user data`

- **undefined:** Represents a variable that has been declared but not assigned a value.

  ```
  let age; // age is undefined
  console.log(age); // Output: undefined
  ```

- **bigint:** A special type for representing extremely large integers.

  `const hugeNumber = 1234567890123456789 0n; // 'n' indicates a BigInt`

- **symbol:** A unique and immutable value, often used for special identifiers.
  `const mySymbol = Symbol("unique identifier");`

2. Non-Primitive Types (Objects)

Objects are collections of key-value pairs. They offer more complex data structures.

- **Object:** A general-purpose container for storing data.

```
const person = {
  name: "Bob",
  age: 42,
  occupation: "Developer"
};
```

- **Array:** An ordered list of values.
  `const colors = ["red", "green", "blue"];`

- **Function:** A block of reusable code that can be executed.

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

3. Type Checking

- **typeof:** This operator returns the type of a value as a string.

```
console.log(typeof "Hello");      // Output: "string"
console.log(typeof 10);         // Output: "number"
console.log(typeof true);        // Output: "boolean"
console.log(typeof null);        // Output: "object" (a histo
console.log(typeof undefined); // Output: "undefined"
```

- **instanceof:** This operator checks if an object is an instance of a specific constructor function.

  ```
  const myArray = [1, 2, 3];
  console.log(myArray instanceof Array); // Output: true
  ```

Key Points:

- **Immutability:** Primitive values are immutable, meaning their value cannot be changed directly.
- **Mutability:** Objects (and their properties) are mutable, meaning their values can be changed.
- **Type Coercion:** JavaScript is often forgiving and will automatically convert values between types (e.g., "10" + 5 becomes the string "105"), but this can lead to subtle bugs if you're not careful.
- **Strict Equality (===):** Use strict equality (===) for comparing values without type coercion to avoid unexpected behavior.

Common String Methods

1. trim()

This method removes whitespace from both ends of a string. It doesn't change the original string but returns a new one.Useful for cleaning up user input or formatting.

Examples:

```
let str = "  Hello, World!  ";
console.log(str.trim()); // "Hello, World!"
```

```javascript
let email = "\\n   user@example.com \\t";
console.log(email.trim()); // "user@example.com"
```

2. charAt()

Returns the character at a specified index in a string. Index is zero-based, like arrays.
If the index is out of range, it returns an empty string.

Examples:

```javascript
let str = "JavaScript";
console.log(str.charAt(0)); // "J"
console.log(str.charAt(4)); // "S"
console.log(str.charAt(10)); // "" (empty string)
```

3. at()

Similar to charAt(), but allows negative indexing.Returns undefined for out-of-range indices.
Useful for accessing characters from the end of the string.

Examples:

```javascript
let str = "Hello";
console.log(str.at(1));  // "e"
console.log(str.at(-1)); // "o" (last character)
console.log(str.at(-2)); // "l" (second to last character)
```

4. slice()

Extracts a portion of a string and returns it as a new string.Takes start and end indices (end is optional).Supports negative indices, counting from the end of the string.

Examples:

```
let str = "The quick brown fox";
console.log(str.slice(4, 9));  // "quick"
console.log(str.slice(4));     // "quick brown fox"
console.log(str.slice(-3));    // "fox"
```

5. substring()

Similar to slice(), but doesn't support negative indices.If start > end, it swaps the two arguments.Returns a substring between two indices.

Examples:

```
let str = "JavaScript";
console.log(str.substring(4, 10)); // "Script"
console.log(str.substring(4));     // "Script"
console.log(str.substring(5, 2));  // "vaS" (same as str.subs
tring(2, 5))
```

6. split()

Splits a string into an array of substrings.Takes a separator and an optional limit as arguments.Useful for parsing strings into manageable chunks.

Examples:

```
let str = "apple,banana,cherry";
console.log(str.split(",")); // ["apple", "banana", "cherry"]

let sentence = "The quick brown fox";
console.log(sentence.split(" ", 3)); // ["The", "quick", "bro
wn"]

let chars = "Hello";
console.log(chars.split("")); // ["H", "e", "l", "l", "o"]
```

7. toUpperCase()

Converts all characters in a string to uppercase.Returns a new string; doesn't modify the original.
Useful for standardizing text or creating visual emphasis.

Examples:

```
let str = "Hello, World!";
console.log(str.toUpperCase()); // "HELLO, WORLD!"

let mixedCase = "javaScript";
console.log(mixedCase.toUpperCase()); // "JAVASCRIPT"
```

8. toLowerCase()

Converts all characters in a string to lowercase.Returns a new string; doesn't modify the original.
Often used for case-insensitive comparisons or formatting.

Examples:

```
let str = "HELLO, World!";
console.log(str.toLowerCase()); // "hello, world!"

let email = "User@Example.com";
console.log(email.toLowerCase()); // "user@example.com"
```

Type Conversion and Coercion

Explicit Conversion

Explicit conversion, also known as type casting, is when we manually convert a value from one type to another using built-in functions or methods.

Examples:

```
// String to Number
let strNum = "123";
let num = Number(strNum);
console.log(num, typeof num); // 123 'number'

// Number to String
let price = 49.99;
let strPrice = String(price);
console.log(strPrice, typeof strPrice); // "49.99" 'string'

// Boolean to String
let isAvailable = true;
let strAvailable = String(isAvailable);
console.log(strAvailable, typeof strAvailable); // "true" 'stri

// String to Boolean
let strBool = "false";
let bool = Boolean(strBool);
console.log(bool, typeof bool); // true 'boolean' (non-empty st
```

Implicit Type Coercion

Implicit type coercion is when JavaScript automatically converts one data type to another in certain situations, often when using operators with different types.

```
// String and Number
console.log("5" + 3);  // "53" (Number coerced to String)
console.log("5" - 3);  // 2 (String coerced to Number)

// Boolean and Number
console.log(true + 1);  // 2 (true coerced to 1)
console.log(false + 1); // 1 (false coerced to 0)

// Equality comparisons
console.log("5" == 5);  // true (loose equality, type coercion
console.log("5" === 5); // false (strict equality, no type coer

// Logical operators
```

```
console.log("hello" && 123);  // 123 (both truthy, returns last
console.log("" || "default"); // "default" ("" is falsy, so it
```

Falsy Values and Their Behavior in JavaScript

In JavaScript, the following values are considered falsy:

- `false`
- `0` (zero)
- `''` or `""` (empty string)
- `null`
- `undefined`
- `NaN` (Not a Number)

```
// Using falsy values in conditional statements
if (false) {
    console.log("This won't run");
}

if (0) {
    console.log("This won't run either");
}

if (null) {
    console.log("Null is falsy, so this won't run");
}

// Truthy vs Falsy in logical operations
console.log(false || "default");  // "default"
console.log(0 || 42);             // 42
console.log("" && "Hello");       // "" (empty string is falsy)
console.log(undefined ?? "fallback"); // "fallback"

// Coercion to boolean
```

```
console.log(Boolean(false));      // false
console.log(Boolean(0));          // false
console.log(Boolean(""));         // false
console.log(Boolean(null));       // false
console.log(Boolean(undefined)); // false
console.log(Boolean(NaN));        // false

// Non-falsy (truthy) examples for comparison
console.log(Boolean(true));       // true
console.log(Boolean(1));          // true
console.log(Boolean("hello"));    // true
console.log(Boolean([]));         // true
console.log(Boolean({}));         // true
```

Operators

Arithmetic Operators:

These operators perform mathematical operations on numbers.

- Addition (+): Adds two numbers.
- Subtraction (-): Subtracts the right operand from the left operand.
- Multiplication (*): Multiplies two numbers.
- Division (/): Divides the left operand by the right operand.
- Modulus (%): Returns the remainder of a division operation.
- Exponentiation (**): Raises the left operand to the power of the right operand.

Examples:

```
let a = 10, b = 3;

console.log(a + b);  // 13
console.log(a - b);  // 7
console.log(a * b);  // 30
console.log(a / b);  // 3.3333...
console.log(a % b);  // 1
console.log(a ** b); // 1000
```

Relational Operators:

These operators compare two values and return a boolean result.

- Less than (<): Returns true if the left operand is less than the right operand.
- Less than or equal to (<=): Returns true if the left operand is less than or equal to the right operand.
- Greater than (>): Returns true if the left operand is greater than the right operand.
- Greater than or equal to (>=): Returns true if the left operand is greater than or equal to the right operand.

Examples:

```
let x = 5, y = 10;

console.log(x < y);   // true
console.log(x <= 5);  // true
console.log(x > y);   // false
console.log(y >= 10); // true
```

Assignment Operators:

These operators assign values to variables, often combining assignment with an arithmetic operation.

- Add and assign (+=): Adds the right operand to the left operand and assigns the result to the left operand.
- Subtract and assign (-=): Subtracts the right operand from the left operand and assigns the result to the left operand.
- Multiply and assign (*=): Multiplies the left operand by the right operand and assigns the result to the left operand.

- Divide and assign (/=): Divides the left operand by the right operand and assigns the result to the left operand.
- Modulus and assign (%=): Performs the modulus operation and assigns the result to the left operand.
- Exponentiate and assign (**=): Raises the left operand to the power of the right operand and assigns the result to the left operand.

Examples:

```
let num = 5;

num += 3;  // num is now 8
console.log(num);

num -= 2;  // num is now 6
console.log(num);

num *= 4;  // num is now 24
console.log(num);

num /= 3;  // num is now 8
console.log(num);

num %= 3;  // num is now 2
console.log(num);

num **= 3; // num is now 8
console.log(num);
```

Logical Operators:

These operators are used to determine the logic between variables or values.

- AND (&&): Returns true if both operands are true.
- OR (||): Returns true if at least one of the operands is true.

- NOT (!): Returns the opposite boolean value of the operand.

Examples:

```
let isAdult = true;
let hasLicense = false;

console.log(isAdult && hasLicense); // false
console.log(isAdult || hasLicense); // true
console.log(!isAdult);              // false

let age = 25;
let hasPermission = true;
console.log((age > 18) && hasPermission); // true
```

Ternary Operator:

This is a shorthand way to write an if...else statement in a single line.

Syntax: condition ? expressionIfTrue : expressionIfFalse

Examples:

```
let age = 20;
let canVote = age >= 18 ? "Yes" : "No";
console.log(canVote); // "Yes"

let temperature = 25;
let weather = temperature > 30 ? "Hot" : temperature < 10 ?
"Cold" : "Moderate";
console.log(weather); // "Moderate"
```

Conditional Statements

1. if...else statement

The `if` statement executes a block of code if a specified condition is true. It's the most basic form of conditional statement.

Basic syntax:

```
if (condition) {
    // code to be executed if the condition is true
}
else {
        // code to be executed if the condition is false
}
```

Example:

```
let age = 18;

if (age >= 18) {
    console.log("You are eligible to vote.");
}
else{
        console.log("Sorry!!!You are not eligible to vote.");
}
```

In this example, the message will be logged because the condition `age >= 18` is true.

2. if...else ladder

The `if...else` ladder (also known as `if...else if...else`) allows you to check multiple conditions and execute different code blocks based on which condition is true.

Syntax:

```
if (condition1) {
    // executed if condition1 is true
} else if (condition2) {
    // executed if condition2 is true
} else if (condition3) {
    // executed if condition3 is true
} else {
    // executed if none of the above conditions are true
}
```

Example:

```
let score = 75;

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: F");
```

This will output "Grade: C" because the score is 75.

3. Nested if...else

Nested `if...else` statements are used when you need to check for multiple conditions within other conditions. This creates a hierarchical decision structure.

Example:

```
let age = 25;
let hasLicense = true;

if (age >= 18) {
    if (hasLicense) {
        console.log("You can drive a car.");
    } else {
        console.log("You are old enough, but need a license t
o drive.");
```

```
    }
} else {
    console.log("You are too young to drive.");
}
```

This will output "You can drive a car." because both conditions are met.

4. switch statement

The `switch` statement is used to perform different actions based on different conditions. It's especially useful when you have many potential conditions to check against a single variable.

Syntax:

```
switch (expression) {
    case value1:
        // code to be executed if expression === value1
        break;
    case value2:
        // code to be executed if expression === value2
        break;
    // more cases...
    default:
        // code to be executed if expression doesn't match an
y cases
}
```

Example:

```
let day = 3;
let dayName;

switch (day) {
    case 1:
        dayName = "Monday";
        break;
    case 2:
        dayName = "Tuesday";
        break;
    case 3:
        dayName = "Wednesday";
        break;
    case 4:
        dayName = "Thursday";
        break;
    case 5:
        dayName = "Friday";
        break;
    case 6:
        dayName = "Saturday";
        break;
    case 7:
        dayName = "Sunday";
        break;
    default:
        dayName = "Invalid day";
}

console.log("The day is " + dayName);
```

This will output "The day is Wednesday" because `day` is 3.

Key points about `switch`:

- The `break` statement is important to prevent fall-through to other cases.
- The `default` case is optional and handles any value that doesn't match a case.
- You can group cases if they should execute the same code.

Loops

1. for loop

The `for` loop is used when you know in advance how many times you want to execute a block of code.

Structure:

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

Examples:

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

// Example 3: Nested for loops (creating a multiplication tab
le)
for (let i = 1; i <= 3; i++) {
    for (let j = 1; j <= 3; j++) {
        console.log(`${i} x ${j} = ${i * j}`);
    }
}
```

2. while loop

The `while` loop executes a block of code as long as a specified condition is true. It's useful when you don't know in advance how many times the loop should run.

Structure:

```
while (condition) {
    // code to be executed
}
```

Examples:

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

3. do...while loop

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

Structure:

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

Examples:

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

// Example 2: Menu selection
let choice;
do {
    console.log("\\nMenu:");
    console.log("1. Option 1");
    console.log("2. Option 2");
    console.log("3. Exit");
```

```javascript
    choice = prompt("Enter your choice (1-3):");
} while (choice !== '3');
```

Key differences and use cases:

1. `for` loop: Best when you know the number of iterations in advance.
2. `while` loop: Useful when you don't know how many iterations you need, but you know the condition to stop.
3. `do...while` loop: Similar to while, but guarantees that the code block is executed at least once.

Object Literals

Defining objects using literal notation

Explanation: Object literals in JavaScript allow you to create objects by directly specifying their properties and values within curly braces {}. This syntax provides a quick and readable way to define objects without using a constructor function.

Examples:

```javascript
let person = { name: "John", age: 30, city: "New York" };
let book = { title: "JavaScript: The Good Parts", author: "Do
uglas Crockford", year: 2008 };
```

Accessing properties using dot notation and bracket notation

Explanation: JavaScript offers two ways to access object properties: dot notation and bracket notation. Dot notation is more concise and commonly used for known, valid identifiers. Bracket notation is useful when property names are dynamic or contain special characters.

Examples:

```javascript
let car = { make: "Toyota", model: "Corolla", year: 2022 };
console.log(car.make); // Dot notation
console.log(car["model"]); // Bracket notation
```

```
let propertyName = "year";
console.log(car[propertyName]); // Using a variable with brac
ket notation
```

Modifying and adding properties

Explanation: JavaScript objects are mutable, allowing you to modify existing properties or add new ones after the object has been created. You can use either dot notation or bracket notation to modify or add properties.

Examples:

```
let phone = { brand: "Apple", model: "iPhone 12" };
phone.color = "Black"; // Adding a new property
phone["storage"] = "128GB"; // Adding a property using bracke
t notation
phone.model = "iPhone 13"; // Modifying an existing property
```

Using methods inside objects

Explanation: Objects can contain functions as properties, which are then called methods. These methods can access and manipulate the object's data, providing a way to associate behavior with the object's properties.

Examples:

```
let calculator = {
  add: function(a, b) { return a + b; },
  multiply: function(a, b) { return a * b; }
};
console.log(calculator.add(5, 3)); // Calling a method
console.log(calculator.multiply(4, 2)); // Calling another me
thod
```

Math object

1. Properties of the Math object:

The Math object has several constant properties representing important mathematical values. These are read-only and can't be changed.

```
console.log(Math.PI);       // 3.141592653589793 (π)
console.log(Math.E);        // 2.718281828459045 (Euler's number)
console.log(Math.SQRT2);    // 1.4142135623730951 (√2)
```

2. Rounding Methods:

The Math object provides several methods for rounding numbers:

```
console.log(Math.round(4.7));    // 5 (rounds to nearest integer)
console.log(Math.round(4.4));    // 4
console.log(Math.ceil(4.1));     // 5 (rounds up to nearest integer)
console.log(Math.floor(4.9));    // 4 (rounds down to nearest integer)
console.log(Math.trunc(4.9));    // 4 (removes decimal part)
```

3. Minimum and Maximum:

You can find the minimum or maximum of a set of numbers:

```
console.log(Math.min(5, 10, 15));    // 5
console.log(Math.max(5, 10, 15));    // 15

// Using spread operator with an array
let numbers = [5, 10, 15, 20, 25];
console.log(Math.min(...numbers));   // 5
console.log(Math.max(...numbers));   // 25
```

4. Power and Square Root:

```
console.log(Math.pow(2, 3));      // 8 (2^3)
console.log(Math.sqrt(16));       // 4 (square root of 16)
console.log(Math.cbrt(27));       // 3 (cube root of 27)
```

5. Absolute Value and Sign:

```
console.log(Math.abs(-5));      // 5 (absolute value)
console.log(Math.sign(-3));     // -1 (sign of number: 1 for p
ositive, -1 for negative, 0 for zero)
```

The Math object is immutable, meaning you can't change its properties or methods. It's also important to note that all Math object methods return a new value; they don't modify any existing values.

Optional Chaining

- Allows safe navigation of deeply nested object properties.
- If any property in the chain is undefined or null, it returns undefined instead of throwing an error.
- Useful for handling potentially missing data in complex objects.

```
// Example 1: Accessing nested properties
const user = {
    name: 'Alice',
    address: {
        street: '123 Main St',
        city: 'Wonderland'
    }
};
console.log(user.address?.street); // Output: '123 Main St'
console.log(user.job?.title); // Output: undefined (no error)
```

```javascript
// Example 2: Method calls
const data = {
    fetchUser: () => ({ name: 'Bob' })
};
console.log(data.fetchUser?.().name); // Output: 'Bob'
console.log(data.getAdmin?.().name); // Output: undefined (no e
```

```javascript
// Example 3: Array access
const arr = [1, 2, 3];
console.log(arr?.[0]); // Output: 1
console.log(arr?.[10]); // Output: undefined
```

Functions

1. Function declaration vs function expression:

Function declarations are hoisted and can be called before they're defined in the code. Function expressions are not hoisted and must be defined before use. Function expressions can be anonymous or named.

Examples:

```javascript
// Function declaration
function greet() {
  return `Hello, Welcome to javascript world`;
}

// Function expression (anonymous)
const greet = function(name) {
  return `Hello, Welcome to javascript world`;
};

// Function expression (named)
const greet = function sayHello(name) {
  return `Hello, Welcome to javascript world`;
};
```

2. Parameters and default parameters:

Parameters are variables listed in a function's definition. Default parameters allow you to specify default values for parameters if no argument is provided or if the argument is undefined.

Examples:

```
// Regular parameters
function add(a, b) {
  return a + b;
}

// Default parameters
function greet(name = "Guest") {
  return `Hello, ${name}!`;
}

// Multiple parameters with defaults
function createUser(username, age = 18, isAdmin = false) {
  return { username, age, isAdmin };
}
```

3. Arrow Functions:

Arrow functions provide a concise syntax for writing function expressions. They have a shorter syntax. Arrow functions are often used for short, simple functions.

Examples:

```
// Basic arrow function
const add = (a, b) => a + b;

// Arrow function with block body
```

```
const greet = name => {
  const greeting = `Hello, ${name}!`;
  return greeting;
};

// Arrow function with no parameters
const getRandomNumber = () => Math.random();
```

Arrays

Arrays in JavaScript are ordered collections of elements. They can be defined using square brackets [] and can contain various data types. Array methods are built-in functions that perform operations on arrays.

Examples:

```
let fruits = ['apple', 'banana', 'orange'];
let mixed = [1, 'two', { name: 'John' }, [4, 5]];
let numbers = new Array(1, 2, 3, 4, 5);
```

forEach() method:

The forEach() method executes a provided function once for each array element. It doesn't create a new array and always returns undefined. This method is useful for performing operations on each element without modifying the original array.

Examples:

```
let numbers = [1, 2, 3, 4, 5];
numbers.forEach(num => console.log(num * 2));

let fruits = ['apple', 'banana', 'orange'];
fruits.forEach((fruit, index) => console.log(`Fruit ${index +
1}: ${fruit}`));
```

filter() method:

The filter() method creates a new array with all elements that pass the test implemented by the provided function. It's useful for selecting specific elements from an array based on a condition.

Examples:

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
let evenNumbers = numbers.filter(num => num % 2 === 0);

let fruits = ['apple', 'banana', 'cherry', 'date', 'elderberr
y'];
let longFruits = fruits.filter(fruit => fruit.length > 5);
```

map() method:

The map() method creates a new array with the results of calling a provided function on every element in the array. It's useful for transforming each element of an array in the same way.

Examples:

```
let numbers = [1, 2, 3, 4, 5];
let squared = numbers.map(num => num ** 2);

let words = ['hello', 'world', 'javascript'];
let uppercased = words.map(word => word.toUpperCase());
```

reduce() method:

The reduce() method executes a reducer function on each element of the array, resulting in a single output value. It's useful for performing calculations on an array and reducing it to a single value.

Examples:

```
let numbers = [1, 2, 3, 4, 5];
let sum = numbers.reduce((acc, curr) => acc + curr, 0);
let words = ['Hello', ' ', 'World', '!'];
let sentence = words.reduce((acc, curr) => acc + curr);
```

In the below i covered only mostly used methods.if you want to know more visit → https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

find()

The `find()` method returns the first element in an array that satisfies a provided testing function.

Explanation:

- It executes the callback function for each array element until it finds one where the callback returns true.
- If such an element is found, `find()` immediately returns that element.
- If no elements satisfy the testing function, `undefined` is returned.

Example:

```
const numbers = [5, 12, 8, 130, 44];
const found = numbers.find(element => element > 10);
console.log(found); // Output: 12
```

every()

The `every()` method tests whether all elements in the array pass the test implemented by the provided function.

Explanation:

- It executes the callback function for each array element.
- If the callback returns false for any element, `every()` immediately returns false.
- If the callback returns true for all elements, `every()` returns true.

Example:

```
const ages = [32, 33, 16, 40];
const allAdults = ages.every(age => age >= 18);
console.log(allAdults); // Output: false
```

some()

The `some()` method tests whether at least one element in the array passes the test implemented by the provided function.

Explanation:

- It executes the callback function for each array element.
- If the callback returns true for any element, `some()` immediately returns true.
- If the callback returns false for all elements, `some()` returns false.

Example:

```
const fruits = ['apple', 'banana', 'mango', 'guava'];
const hasLongName = fruits.some(fruit => fruit.length > 5);
console.log(hasLongName); // Output: true (because 'banana' h
as more than 5 characters)
```

includes()

The `includes()` method determines whether an array includes a certain value among its entries.

Explanation:

- It checks if a specific element exists in the array.
- Returns true if the element is found, false otherwise.
- It can take an optional second argument to specify the position to start searching from.

Example:

```
const pets = ['cat', 'dog', 'bat'];
console.log(pets.includes('cat')); // Output: true
console.log(pets.includes('at')); // Output: false

// Using the optional fromIndex parameter
console.log(pets.includes('cat', 1)); // Output: false (start
s searching from index 1)
```

For-of vs For-in

for...of loop:

The `for...of` loop is used to **iterate over iterable objects** like arrays, strings, and other array-like objects. It provides a clean and concise way to loop through elements, giving you direct access to the values without needing to use an index. This loop is ideal when you want to perform operations on each element of an iterable.

Examples:

```
// Iterating over an array
const fruits = ['apple', 'banana', 'orange'];
for (const fruit of fruits) {
  console.log(fruit);
}

// Iterating over a string
const greeting = 'Hello';
for (const char of greeting) {
  console.log(char);
}

// Iterating over a Set
const uniqueNumbers = new Set([1, 2, 3, 4, 5]);
for (const number of uniqueNumbers) {
  console.log(number);
}
```

for...in loop:

The `for...in` loop is used to **iterate over the enumerable properties of an object**. It's particularly useful when you want to examine or manipulate the properties of an object. However, it's important to note that this loop also includes properties from the prototype chain, so you might need to use `hasOwnProperty()` to filter out inherited properties if needed.

Examples:

```
// Iterating over object properties
const person = { name: 'John', age: 30, job: 'developer' };
for (const key in person) {
  console.log(`${key}: ${person[key]}`);
}

// Iterating over array indices (not recommended for arrays)
const colors = ['red', 'green', 'blue'];
for (const index in colors) {
  console.log(`Index ${index}: ${colors[index]}`);
}

// Using hasOwnProperty to filter out inherited properties
const car = Object.create({ brand: 'Generic' });
car.model = 'Sedan';
car.year = 2023;
for (const prop in car) {
  if (car.hasOwnProperty(prop)) {
    console.log(`${prop}: ${car[prop]}`);
  }
}
```

Destructuring

Array Destructuring:

Array destructuring allows you to extract values from arrays and assign them to variables in a more concise way.

Explanation:

- It uses a syntax that mimics the structure of an array literal.
- You can assign multiple variables at once, matching their positions in the array.
- You can skip elements, use rest parameters, and provide default values.

Examples:

a) Basic array destructuring:

```
const colors = ['red', 'green', 'blue'];
const [firstColor, secondColor, thirdColor] = colors;

console.log(firstColor);  // Output: 'red'
console.log(secondColor); // Output: 'green'
console.log(thirdColor);  // Output: 'blue'
```

b) Skipping elements and using rest parameter:

```
const numbers = [1, 2, 3, 4, 5];
const [first, , third, ...rest] = numbers;

console.log(first);  // Output: 1
console.log(third);  // Output: 3
console.log(rest);   // Output: [4, 5]
```

c) Using default values:

```
const [a, b, c = 3] = [1, 2];
```

```
console.log(a); // Output: 1
console.log(b); // Output: 2
console.log(c); // Output: 3 (default value used)
```

Object Destructuring:

Object destructuring allows you to extract values from objects and assign them to variables using the object's property names.

Explanation:

- It uses a syntax that mimics the structure of an object literal.
- You can assign variables with names that match the object's properties.
- You can rename variables, provide default values, and use nested destructuring.

Examples:

a) Basic object destructuring:

```
const person = { name: 'Alice', age: 30, city: 'New York' };
const { name, age, city } = person;

console.log(name); // Output: 'Alice'
console.log(age);  // Output: 30
console.log(city); // Output: 'New York'
```

b) Renaming variables and using default values:

```
const product = { id: 1, title: 'Laptop' };
const { id: productId, title: productName, price = 999 } = pr
oduct;

console.log(productId);   // Output: 1
console.log(productName); // Output: 'Laptop'
console.log(price);       // Output: 999 (default value used)
```

c) Nested object destructuring:

```
const user = {
  id: 42,
  details: {
    firstName: 'John',
    lastName: 'Doe',
    address: {
      street: '123 Main St',
      city: 'Anytown'
    }
  }
};

const { id, details: { firstName, lastName, address: { city }
} } = user;

console.log(id);        // Output: 42
console.log(firstName); // Output: 'John'
console.log(lastName);  // Output: 'Doe'
console.log(city);      // Output: 'Anytown'
```

These destructuring techniques can make your code more readable and concise, especially when working with complex data structures or when you need to extract multiple values from arrays or objects.

Set and Map

Set:

A Set is a collection that stores unique values. This means each element can only appear once in a Set.

Example 1:

```javascript
const fruits = new Set(['apple', 'banana', 'orange', 'appl
e']);
console.log(fruits); // Set(3) { 'apple', 'banana', 'orange'
}
```

Notice that 'apple' is only included once, even though we added it twice.

Example 2:

```javascript
const numbers = new Set();
numbers.add(1);
numbers.add(2);
numbers.add(3);
numbers.add(2); // This won't be added as 2 is already in the
Set
console.log(numbers); // Set(3) { 1, 2, 3 }
console.log(numbers.has(2)); // true
console.log(numbers.size); // 3
```

Map:

A Map is a collection of key-value pairs where each key can only appear once.

Example 1:

```javascript
const userInfo = new Map();
userInfo.set('name', 'John');
userInfo.set('age', 30);
userInfo.set('city', 'New York');

console.log(userInfo.get('name')); // 'John'
console.log(userInfo.has('age')); // true
console.log(userInfo.size); // 3
```

Example 2:

```
const scores = new Map([
  ['Alice', 95],
  ['Bob', 80],
  ['Charlie', 90]
]);

console.log(scores.get('Bob')); // 80
scores.set('Bob', 85); // Updating Bob's score
console.log(scores.get('Bob')); // 85
```

Iterating over Sets and Maps:

For Set:

```
const colors = new Set(['red', 'green', 'blue']);

// Using forEach
colors.forEach(color => {
  console.log(color);
});

// Using for...of
for (let color of colors) {
  console.log(color);
}
```

For Map:

```
const fruits = new Map([
  ['apple', 5],
  ['banana', 3],
  ['orange', 2]
]);

// Using forEach
```

```
fruits.forEach((value, key) => {
  console.log(`${key}: ${value}`);
});

// Using for...of
for (let [fruit, quantity] of fruits) {
  console.log(`${fruit}: ${quantity}`);
}
```

Difference between object and Map:

Feature	Object	Map
Key Types	Only strings and symbols	Any value (including functions, objects, and primitives)
Key Order	Not guaranteed to maintain insertion order	Maintains insertion order
Size	No built-in way to get size	Has a `size` property
Iteration	Requires additional methods for direct iteration	Directly iterable
Performance	Better for small collections	Better for large collections with frequent additions/removals
Accessing Properties	Dot notation or bracket notation	`get()` method
Setting Properties	Direct assignment	`set()` method
Deleting Properties	`delete` operator	`delete()` method
Use Cases	Best for simple key-value storage	Best for frequent additions/removals and when keys are unknown until runtime

try, catch, and finally

Imagine you're building a complex JavaScript application. There are many scenarios where things can go wrong:

- **Division by zero:** 10 / 0 throws an error.

- **Accessing a non-existent property:** obj.nonExistingProperty throws an error.
- **Network issues:** Fetching data from a server might fail.

The try...catch...finally block is your safety net for these situations.

1. try Block: The Code to Protect

This is where you put the code that could potentially throw an error.

```
  try {
    // Code that might throw an error
    let result = 10 / 0; // This will cause a division by zero e
    console.log(result);
    }
```

2. catch Block: Handling the Error

If an error occurs within the try block, the execution jumps to the catch block. Here, you can:

- **Identify the error:** Access the error object to understand the specific error that happened.
- **Take corrective action:** Log the error, display an error message to the user, or implement a fallback solution.

```
try {
    let result = 10 / 0;
} catch (error) {
  console.error("An error occurred:", error);
  // Example: Display an error message to the user
  alert("Oops! Something went wrong.");
}
```

3. finally Block: Always Execute

The finally block runs **regardless** of whether an error occurred. Use this block for tasks that **must** happen, such as:

- **Closing connections:** Releasing resources (e.g., closing a file, ending a database connection).
- **Cleanup tasks:** Cleaning up temporary files or resetting variables.

```
try {
    let result = 10 / 0;
} catch (error) {
  console.error("An error occurred:", error);
} finally {
  console.log("This message will always be displayed.");
}
```

Throwing Custom Errors

Sometimes, you want to raise an error explicitly in your code to signal a problem. You can do this with the throw keyword:

```
function checkAge(age) {
   if (age < 18) {
       throw new Error("You must be at least 18 years old.");
   } else {
       console.log("Welcome!");
   }
}

try {
    checkAge(15);
} catch (error) {
    console.error("Error:", error.message);
}
```

Explanation:

1. **throw new Error(...):** This creates a new Error object with a custom message and throws it.
2. **catch block:** The error is caught here.
3. **error.message:** The error message is accessed to provide information about the issue.

Key Benefits of Error Handling

- **Robust Applications:** Prevent your programs from crashing unexpectedly.
- **User-Friendly Experiences:** Gracefully handle errors to provide informative messages to users.
- **Debugging:** Error objects help you identify the source of problems.

Practical Example:

```
function fetchData(url) {
    try {
        const response = await fetch(url);
        if (!response.ok) {
            throw new Error(`Network error: ${response.status}`
        }
        const data = await response.json();
        return data;
    } catch (error) {
        console.error("Error fetching data:", error);
        return null;
    } finally {
        console.log("Data fetch complete.");
    }
}
```

This code fetches data from a URL. It handles potential network errors, uses try...catch for error handling, and uses the finally block to log the completion of the data fetch.