Quick JavaScript Revision

|  |
| --- |
| What is JavaScript? |
| JavaScript is a lightweight, cross-platform, single-threaded, and interpreted compiled programming language. It is also known as the scripting language for WebPages. It is well-known for the development of web pages, and many non-browser environments also use it.  JavaScript is a [weakly typed language](https://www.geeksforgeeks.org/type-systemsdynamic-typing-static-typing-duck-typing/) (dynamically typed). JavaScript can be used for [Client-side](https://www.geeksforgeeks.org/server-side-client-side-programming/) developments as well as [Server-side](https://www.geeksforgeeks.org/server-side-client-side-programming/) developments. JavaScript is both an imperative and declarative type of language. JavaScript contains a standard library of objects, like [Array](https://www.geeksforgeeks.org/arrays-in-javascript/), [Date](https://www.geeksforgeeks.org/javascript-date-objects/), and [Math](https://www.geeksforgeeks.org/javascript-math-object/), and a core set of language elements like [operators](https://www.geeksforgeeks.org/javascript-operators/), control structures, and [statements](https://www.geeksforgeeks.org/javascript-statements/). |
| Key Features of JavaScript:   1. **Client-Side Scripting:** JavaScript is most commonly used on the client side (i.e., in the browser) to control the behaviour of web pages. For example, it can handle user interactions like clicks, form submissions, or input validation. 2. **Interpreted Language:** JavaScript code is executed by the browser's JavaScript engine, such as V8 in Chrome, without needing to be compiled. 3. **Event-Driven:** JavaScript can respond to events like clicks, mouse movements, and keyboard inputs, making it ideal for creating interactive web applications. 4. **Object-Oriented:** Although not a fully object-oriented language, JavaScript supports object-oriented programming concepts such as objects, classes, and inheritance. 5. **Versatile and Dynamic:** JavaScript is not limited to just web browsers. It can be used on servers (using Node.js), for mobile app development (using frameworks like React Native), and even for desktop applications. 6. **Cross-Browser Compatibility:** JavaScript is supported by all modern web browsers, although differences in implementation may require developers to use polyfills or frameworks for consistent behaviour across different environments. 7. **Asynchronous Programming:** JavaScript supports asynchronous programming, allowing developers to execute code without blocking the main thread. This is achieved through features like call-backs, promises, and async/await. 8. **Rich Ecosystem:** JavaScript has a vast ecosystem of libraries, frameworks, and tools, such as React, Angular, Vue.js, and jQuery, which simplify and enhance development tasks. |
| History of JavaScript |
| It was created in 1995 by Brendan Eich while he was an engineer at Netscape. It was originally going to be named LiveScript but was renamed. Unlike most programming languages, JavaScript language has no concept of input or output. It is designed to run as a scripting language in a host environment, and it is up to the host environment to provide mechanisms for communicating with the outside world. The most common host environment is the browser. |
|  |

. = access member operator

JavaScript Versions

|  |
| --- |
| This language became an ECMA standard in the year 1997. Versions are named ECMAScript2016, 2017…2020 after 2016. |
|  |

ECMA🡪 **European Computer Manufacturers Association**

|  |
| --- |
| JavaScript was developed under the name "**Mocha**," then renamed to "**LiveScript**," and **finally, it was called "JavaScript**" when **Netscape introduced** it. However, due to trademark issues with the name "Java," the language's standardization was carried out under the name "ECMAScript" by the European Computer Manufacturers Association (ECMA).  **Key Terms:**  **JavaScript:** The popular, commonly used name.  **ECMAScript:** The standardized version of the language, which defines the core features and specifications |
|  |

|  |
| --- |
| JavaScript is used primarily in the following areas:   1. **Web Development**: To create interactive and dynamic content on websites (e.g., animations, form validation, pop-ups). 2. **Web Applications**: For building complex web applications like single-page applications (SPAs) using frameworks like React, Angular, or Vue.js. 3. **Server-Side Development**: Through environments like Node.js, JavaScript can run on servers to handle backend logic. 4. **Mobile App Development**: Using frameworks like React Native to create cross-platform mobile apps. 5. **Desktop Applications**: With tools like Electron, JavaScript can be used to build desktop applications. 6. **Game Development**: JavaScript, along with HTML5, is used to develop browser-based games. |
|  |
| In JavaScript, variables can be declared using three main keywords:  **1. var**   * **Scope**: Function-scoped or globally scoped. * **Usage**: It was the original way to declare variables in JavaScript. * **Behavior**: Allows re-declaration of the same variable within the same scope.  |  | | --- | | var name = "John"; |   **2. let**   * **Scope**: Block-scoped (i.e., within {}). * **Usage**: Introduced in ES6 (ECMAScript 2015), let is preferred for variables that may change value but should be limited to the block where they are defined. * **Behavior**: Cannot be re-declared within the same block scope, but can be reassigned.  |  | | --- | | let age = 25; |   **3. const**   * **Scope**: Block-scoped. * **Usage**: Also introduced in ES6, const is used to declare variables that should not be reassigned after their initial value is set. * **Behavior**: The value assigned to a const variable cannot be changed, making it a constant. However, if the const variable is an object or array, the properties or elements inside can still be modified.  |  | | --- | | const pi = 3.14; |   **Summary:**   * **var**: Function or globally scoped, allows re-declaration. * **let**: Block-scoped, allows reassignment but not re-declaration. * **const**: Block-scoped, cannot be reassigned or re-declared. |
|  |

|  |
| --- |
| **Rules to Declare a Variable** |
| * A variable name can consist of Capital letters A-Z, lowercase letters a-z digits 0-9, and two special characters such as \_ underscore and $ dollar sign. * **The first character must not be a digit.** * **Blank spaces cannot** be used in variable names. * **Java keywords** cannot be used as variable names. * Variable names are **case-sensitive.** * There is no limit on the length of a variable name but by convention, it should be between 4 to 15 chars. * Variable names always should exist on the **left-hand side** of assignment operators. |

|  |
| --- |
| Example  script.js |
| // Declare and initialize two variables  let number1 = 10;  let number2 = 20;  // Add the two numbers  let sum = number1 + number2;  // Display the result  console.log("The sum of " + number1 + " and " + number2 + " is: " + sum); |
| The sum of 10 and 20 is: 30 |

List of Java keywords

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| abstract | continue | for | new | switch |
| assert | package | synchronized | default | goto |
| boolean | do | if | private | this |
| break | else | import | public | throw |
| byte | enum | implements | protected | throws |
| case | double | instanceof | return | transient |
| catch | extends | int | short | try |
| char | final | interface | static | void |
| class | finally | long | strictfp | volatile |
| const | float | native | super | while |

# Javascript Data Types

|  |
| --- |
| There are two type of data type.   1. Non-primitive (reference) data type 2. Primitive data type    **Primitive Data Types**: Number, String, Boolean, Null, Undefined, Symbol, BigInt.   **Reference (Object) Data Types**: Object, Array, Function, Date, RegExp, Map, Set. |
| The predefined data types provided by JavaScript language are known as **primitive data types**. Primitive data types are also known as **in-built data types**. |

|  |  |
| --- | --- |
| Type | Description |
| Number | JavaScript numbers are always stored in double-precision 64-bit binary format IEEE 754. |
| String | JavaScript Strings are made up of a list of characters, essentially an array of characters. |
| Boolean | Represents a logical entity and can have two values: true or false. |
| Null | This type has only one value: null. |
| Undefined | A variable that has not been assigned a value is undefined. |
| Symbol | Symbols return unique identifiers that can be used as property keys in objects without colliding with other keys. |
| BigInt | BigInt is a built-in object providing a way to represent whole numbers larger than 253-1. |

**Operator in Java script**

|  |
| --- |
| Types of JavaScript Operators |

1) Arithmetic Operators

Arithmetic operators are used to perform mathematical operations on numbers. These operators are fundamental in handling numerical data and calculations in JavaScript.

|  |  |
| --- | --- |
| Operator | Description |
| + | Adds two numeric operands |
| - | Subtracts the second operand from the first |
| \* | Multiplies two numeric operands |
| / | Divides the first operand by the second |
| % | Returns the remainder of the division |
| ++ | Increments the operand by one |
| -- | Decrements the operand by one |

2) Assignment Operators

Assignment operators are used to provide values to variables. They also include shorthand operations for performing arithmetic and assignment in a single step.

|  |  |
| --- | --- |
| Operator | Description |
| = | Assigns the right-hand operand to the left-hand operand |
| += | assigns the result after adding the right-hand operand to the left-hand operand |
| -= | Subtracts the right-hand operand from the left-hand operand and assigns the result |
| \*= | Multiplies the right-hand operand with the left-hand operand and assigns the result |
| /= | Divides the left-hand operand by the right-hand operand and assigns the result |
| %= | Calculates the remainder when the left-hand operand is divided by the right-hand operand and assigns the result |

3) Comparison Operators

Comparison operators compare two values and provide a boolean result. These operators are essential for making decisions in code based on different conditions.

|  |  |
| --- | --- |
| Operator | Description |
| == | Checks if two values are equal |
| != | Checks if two values are not equal |
| === | Checks if two values are strictly equal (value and type) |
| !== | Checks if two values are strictly not equal (value and type) |
| > | Checks if the first value is greater than the second |
| < | Checks if the first value is less than the second |
| >= | Checks if the first value is greater than or equal to the second |
| <= | Checks if the first value is less than or equal to the second |

4) Logical Operators

Multiple criteria can be combined using logical operators, and a boolean result can be returned. They are crucial for controlling the flow of a program based on complex logical conditions.

|  |  |
| --- | --- |
| Operator | Description |
| && | Logical AND returns true if both operands are true |
| || | Logical OR returns true if one operand is true |
| ! | Logical NOT inverts the boolean value of the operand |

5) Bitwise Operators

Binary representations of numbers are operated on by bitwise operators. These operators are used for low-level programming tasks that involve direct manipulation of bits.

|  |  |
| --- | --- |
| Operator | Description |
| & | Bitwise AND compares each bit of its operands and returns 1 only if both bits are 1 |
| | | Bitwise OR compares each bit of its operands and returns 1 if either bit is 1 |
| ^ | Bitwise XOR compares each bit of its operands and returns 1 only if one of the bits is 1, but not both |
| ~ | Bitwise NOT, inverts all the bits of the operand |
| << | Left shift shifts the bits of the operand to the left |
| >> | Right shift shifts the bits of the operand to the right |
| >>> | Zero-fill right shift shifts the bits of the operand to the right with zero-fill |

6) String Operators

String operators are utilised to perform operations on string values. They are essential for handling and manipulating text data.

|  |  |
| --- | --- |
| Operator | Description |
| + | Concatenates two string values |

7) Typeof Operators

The Typeof operators are used to determining or convert the data types of values. They help in checking and manipulating the types of variables.

|  |  |
| --- | --- |
| Operator | Description |
| typeof | Returns the type of a variable |

8) Conditional (Ternary) Operator

The conditional (ternary) operator is a shorthand for an if-else statement. It allows for a concise syntax to assign values based on a condition.

|  |  |
| --- | --- |
| Operator | Description |
| condition ? value1 : value2 | Returns value1 if the condition is true. otherwise returns value2 |

9) Unary Operators

Unary operators operate on a single operand. They are used for various purposes like negation, type determination, and deletion of properties.

|  |  |
| --- | --- |
| Operator | Description |
| + | Unary plus converts the operand to a number |
| - | Unary negation negates the operand |
| ! | Logical NOT inverts the boolean value of the operand |
| ++ | Increments the operand by one |
| -- | Decrements the operand by one |
| void | Discards the return value of an expression |
| delete | Deletes an object property |

Example:

|  |
| --- |
| let str = "5";  let num = 5;  let bool = true;  let obj = { name: "John", age: 30 };  let count = 10;  // Unary plus  console.log("Unary plus: " + (+str));  // Unary negation  console.log("Unary negation: " + (-num));  // Logical NOT  console.log("Logical NOT: " + (!bool));  // Increment  count++;  console.log("Increment: " + count);  // Decrement  count--;  console.log("Decrement: " + count);  // Void  void function() {      console.log("This will not return anything.");  }();  // Delete  delete obj.age;  console.log("Delete property: " + JSON.stringify(obj)); |

Output: 

|  |
| --- |
| Unary plus: 5  Unary negation: -5  Logical NOT: false  Increment: 11  Decrement: 10  This will not return anything.  Delete property: {"name":"John"} |

10) Relational Operators

Relational operators test the relationship between two values. They are used to determine the presence of properties in objects and the instance relationship of objects.

|  |  |
| --- | --- |
| Operator | Description |
| in | Checks if a property is in an object |
| instanceof | Checks if an object is an instance of a constructor |

Example:

|  |
| --- |
| let obj = { name: "John", age: 30 };  let arr = [1, 2, 3];  let date = new Date();  // `in` operator  console.log("name in obj: " + ("name" in obj));  console.log("length in arr: " + ("length" in arr));  console.log("height in obj: " + ("height" in obj));  // `instanceof` operator  console.log("date instanceof Date: " + (date instanceof Date));  console.log("arr instanceof Array: " + (arr instanceof Array));  console.log("obj instanceof Object: " + (obj instanceof Object));  console.log("date instanceof Object: " + (date instanceof Object)); |

Output:

|  |
| --- |
| name in obj: true  length in arr: true  height in obj: false  date instanceof Date: true  arr instanceof Array: true  obj instanceof Object: true  date instanceof Object: true |

Conditional Statement

|  |  |
| --- | --- |
| Conditional Statement | Description |
| if statement | Executes a block of code if a specified condition is true. |
| else statement | Executes a block of code if the same condition of the preceding if statement is false. |
| else if statement | Adds more conditions to the if statement, allowing for multiple alternative conditions to be tested. |
| switch statement | Evaluates an expression, then executes the case statement that matches the expression’s value. |
| ternary operator | Provides a concise way to write if-else statements in a single line. |
| Nested if else statement | Allows for multiple conditions to be checked in a hierarchical manner. |

Loop in Java script

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1. **for Loop**: Executes a block of code a specified number of times.  |  | | --- | | for (initialization; condition; increment) {  // code to be executed  } | | for (let i = 0; i < 5; i++) {  console.log(i);  } |  1. **while Loop**: Runs a block of code as long as a condition is true.  |  | | --- | | while (condition) {  // code to be executed  } | | let i = 0;  while (i < 5) {  console.log(i);  i++;  } |  1. **do...while Loop**: Executes a block of code once, then repeats as long as a condition is true.  |  | | --- | | do {  // code to be executed  } while (condition); | | let i = 0;  do {  console.log(i);  i++;  } while (i < 5); |  1. **for...in Loop**: Iterates over the enumerable properties of an object. For print index value  |  | | --- | | for (key in object) {  // code to be executed  } | | const person = { firstName: "John", lastName: "Doe", age: 30 };  for (let key in person) {  console.log(key + ": " + person[key]);  } |  1. **for...of Loop**: Iterates over the values of iterable objects.for of use to print value of object  |  | | --- | | for (const value of iterable) {  // code to be executed  } | | const fruits = ["apple", "banana", "cherry"];  for (const fruit of fruits) {  console.log(fruit);  } |  1. **Labeled Statement**: Allows labeling of loops or blocks for use with break and continue.  |  | | --- | | label: {  // code to be executed  } | | outerLoop: for (let i = 0; i < 3; i++) {  for (let j = 0; j < 3; j++) {  if (i === 1 && j === 1) break outerLoop;  console.log(`i = ${i}, j = ${j}`);  }  } |  1. **break Statement**: Exits a loop immediately.  |  | | --- | | break; | | outerLoop: for (let i = 0; i < 3; i++) {  for (let j = 0; j < 3; j++) {  if (i === 1 && j === 1) break outerLoop;  console.log(`i = ${i}, j = ${j}`);  }  } |  1. **continue Statement**: Skips the current iteration and proceeds to the next.  |  | | --- | | continue; | | for (let i = 0; i < 5; i++) {  if (i === 3) continue;  console.log(i);  } |  1. **Infinite Loop (Loop Error)**: A loop that runs indefinitely due to a condition that always evaluates to true.  |  | | --- | | while (true) {  console.log("This will run forever");  } | |  | |

**Array in Java script**

In JavaScript, an array is a special type of object used to store a collection of values. Arrays can hold multiple values under a single variable name and provide methods for manipulating these values.

|  |
| --- |
| JavaScript array is an object that represents a collection of similar type of elements.  There are 3 ways to construct array in JavaScript   * By array literal * By creating instance of Array directly (using new keyword) * By using an Array constructor (using new keyword) |
| 1) JavaScript array literal The syntax of creating array using array literal is given below:  **var arrayname=[value1,value2.....valueN];**  As you can see, values are contained inside [ ] and separated by , (comma).   |  | | --- | | 1. **<script>** 2. var emp=["Sonoo","Vimal","Ratan"]; 3. for (i=0;i**<emp.length**;i++){ 4. document.write(emp[i] + "**<br/>**"); 5. } 6. **</script>** | | Output  Sonoo Vimal Ratan | |
| 2) JavaScript Array directly (new keyword) The syntax of creating array directly is given below:   1. var arrayname=new Array();   Here, **new keyword** is used to create instance of array.   |  | | --- | | 1. **<script>** 2. var i; 3. var emp = new Array(); 4. emp[0] = "Arun"; 5. emp[1] = "Varun"; 6. emp[2] = "John"; 8. for (i=0;i**<emp.length**;i++){ 9. document.write(emp[i] + "**<br>**"); 10. } 11. **</script>** | | Arun Varun John | |
| 3) JavaScript array constructor (new keyword) Here, you need to create instance of array by passing arguments in constructor so that we don't have to provide value explicitly. The example of creating object by array constructor is given below.   |  | | --- | | 1. **<script>** 2. var emp=new Array("Jai","Vijay","Smith"); 3. for (i=0;i**<emp.length**;i++){ 4. document.write(emp[i] + "**<br>**"); 5. } 6. **</script>** | | Jai Vijay Smith | |
|  |

## JavaScript Array Methods

|  |  |
| --- | --- |
| Methods | Description |
| [concat()](https://www.javatpoint.com/javascript-array-concat-method) | It returns a new array object that contains two or more merged arrays. |
| [copywithin()](https://www.javatpoint.com/javascript-array-copywithin-method) | It copies the part of the given array with its own elements and returns the modified array. |
| [entries()](https://www.javatpoint.com/javascript-array-entries-method) | It creates an iterator object and a loop that iterates over each key/value pair. |
| [every()](https://www.javatpoint.com/javascript-array-every-method) | It determines whether all the elements of an array are satisfying the provided function conditions. |
| [flat()](https://www.javatpoint.com/javascript-array-flat-method) | It creates a new array carrying sub-array elements concatenated recursively till the specified depth. |
| [flatMap()](https://www.javatpoint.com/javascript-array-flatmap-method) | It maps all array elements via mapping function, then flattens the result into a new array. |
| [fill()](https://www.javatpoint.com/javascript-array-fill-method) | It fills elements into an array with static values. |
| [from()](https://www.javatpoint.com/javascript-array-from-method) | It creates a new array carrying the exact copy of another array element. |
| [filter()](https://www.javatpoint.com/javascript-array-filter-method) | It returns the new array containing the elements that pass the provided function conditions. |
| [find()](https://www.javatpoint.com/javascript-array-find-method) | It returns the value of the first element in the given array that satisfies the specified condition. |
| [findIndex()](https://www.javatpoint.com/javascript-array-findindex-method) | It returns the index value of the first element in the given array that satisfies the specified condition. |
| [forEach()](https://www.javatpoint.com/javascript-array-foreach-method) | It invokes the provided function once for each element of an array. |
| [includes()](https://www.javatpoint.com/javascript-array-includes-method) | It checks whether the given array contains the specified element. |
| [indexOf()](https://www.javatpoint.com/javascript-array-indexof-method) | It searches the specified element in the given array and returns the index of the first match. |
| [isArray()](https://www.javatpoint.com/javascript-array-isarray-method) | It tests if the passed value ia an array. |
| [join()](https://www.javatpoint.com/javascript-array-join-method) | It joins the elements of an array as a string. |
| [keys()](https://www.javatpoint.com/javascript-array-keys-method) | It creates an iterator object that contains only the keys of the array, then loops through these keys. |
| [lastIndexOf()](https://www.javatpoint.com/javascript-array-lastindexof-method) | It searches the specified element in the given array and returns the index of the last match. |
| [map()](https://www.javatpoint.com/javascript-array-map-method) | It calls the specified function for every array element and returns the new array |
| [of()](https://www.javatpoint.com/javascript-array-of-method) | It creates a new array from a variable number of arguments, holding any type of argument. |
| [pop()](https://www.javatpoint.com/javascript-array-pop-method) | It removes and returns the last element of an array. |
| [push()](https://www.javatpoint.com/javascript-array-push-method) | It adds one or more elements to the end of an array.  It return length of the updated array. |
| [reverse()](https://www.javatpoint.com/javascript-array-reverse-method) | It reverses the elements of given array. |
| [reduce(function, initial)](https://www.javatpoint.com/javascript-array-reduce-method) | It executes a provided function for each value from left to right and reduces the array to a single value. |
| [reduceRight()](https://www.javatpoint.com/javascript-array-reduceright-method) | It executes a provided function for each value from right to left and reduces the array to a single value. |
| [some()](https://www.javatpoint.com/javascript-array-some-method) | It determines if any element of the array passes the test of the implemented function. |
| [shift()](https://www.javatpoint.com/javascript-array-shift-method) | It removes and returns the first element of an array. |
| [slice()](https://www.javatpoint.com/javascript-array-slice-method) | It returns a new array containing the copy of the part of the given array. |
| [sort()](https://www.javatpoint.com/javascript-array-sort-method) | It returns the element of the given array in a sorted order. |
| [splice()](https://www.javatpoint.com/javascript-array-splice-method) | It add/remove elements to/from the given array. |
| [toLocaleString()](https://www.javatpoint.com/javascript-array-tolocalestring-method) | It returns a string containing all the elements of a specified array. |
| [toString()](https://www.javatpoint.com/javascript-array-tostring-method) | It converts the elements of a specified array into string form, without affecting the original array. |
| [unshift()](https://www.javatpoint.com/javascript-array-unshift-method) | It adds one or more elements in the beginning of the given array.  It return length of the updated array. |
| [values()](https://www.javatpoint.com/javascript-array-values-method) | It creates a new iterator object carrying values for each index in the array. |
| Arrraname.at() ES6 | It return index value; |

Function in JavaScript

|  |
| --- |
| Function 🡪 A JavaScript function is a block of code designed to perform a **particular tas**k. It encapsulates a set of instructions that can be reused throughout a program.[Functions](https://www.geeksforgeeks.org/types-of-functions/) can take parameters, execute statements, and return values, enabling code organization, modularity, and reusability in JavaScript programming. |
|  |

1. Function Declaration

|  |
| --- |
| function functionName(parameters) {  // code to be executed  } |
| function greet(name) {  console.log("Hello, " + name + "!");  }  greet("Alice"); // Output: Hello, Alice! |

**2. Function Expression**

A function expression defines a function and assigns it to a variable. Function expressions can be named or anonymous.

**Syntax:**

|  |
| --- |
| const functionName = function(parameters) {  // code to be executed  }; |
| const add = function(a, b) {  return a + b;  };  console.log(add(5, 3)); // Output: 8 |

**3. Arrow Function**

Arrow functions provide a more concise syntax for writing functions. They are particularly useful for short functions.

**Syntax:**

|  |
| --- |
| const functionName = (parameters) => {  // code to be executed  }; |
| const multiply = (x, y) => x \* y;  console.log(multiply(4, 5)); // Output: 20 |

**4. IIFE (Immediately Invoked Function Expression)**

An IIFE is a function that is defined and executed immediately.

**Syntax:**

|  |
| --- |
| (function() {  // code to be executed  })(); |
| (function() {  console.log("This function runs immediately!");  })(); |

Function Parameters and Return Value

**1. Parameters**

Functions can accept parameters, which are values passed into the function.

**Example:**

|  |
| --- |
| function greet(name, age) {  console.log("Hello, " + name + ". You are " + age + " years old.");  }  greet("Bob", 25); // Output: Hello, Bob. You are 25 years old. |
|  |

**2. Return Value**

Functions can return a value using the return statement. If no return statement is specified, the function returns undefined.

**Example:**

|  |
| --- |
| function square(number) {  return number \* number;  }  let result = square(4);  console.log(result); // Output: 16 |

**Scope and Closures**

**1. Local and Global Scope**

* **Local Scope**: Variables declared inside a function are local to that function and cannot be accessed outside it.
* **Global Scope**: Variables declared outside any function are global and can be accessed from anywhere in the code.

**Example:**

|  |
| --- |
| let globalVar = "I am global";  function example() {  let localVar = "I am local";  console.log(globalVar); // Accessible  console.log(localVar); // Accessible  }  console.log(globalVar); // Accessible  console.log(localVar); // Error: localVar is not defined |

**2. Closures**

A closure is a function that retains access to its lexical scope even when the function is executed outside that scope.

**Example:**

|  |
| --- |
| function createCounter() {  let count = 0;  return function() {  count += 1;  return count;  };  }  const counter = createCounter();  console.log(counter()); // Output: 1  console.log(counter()); // Output: 2 |

### ****Function Overloading****

JavaScript does not support function overloading in the traditional sense (i.e., defining multiple functions with the same name but different parameters). However, you can handle multiple arguments in a function by checking the arguments object or using default parameters.

**Example:**

|  |
| --- |
| function greet(name, greeting = "Hello") {  console.log(greeting + ", " + name + "!");  }  greet("Charlie"); // Output: Hello, Charlie!  greet("Charlie", "Hi"); // Output: Hi, Charlie! |

 **Function Declaration**

Javascript

|  |
| --- |
| function greet() {  console.log("Hello, World!");  } |

 **Function Expression**

javascript

|  |
| --- |
| const greet = function() {  console.log("Hello, World!");  }; |

 **Arrow Function**

javascript

|  |
| --- |
| const greet = () => {  console.log("Hello, World!");  }; |

 **Immediately Invoked Function Expression (IIFE)**

javascript

|  |
| --- |
| (function() {  console.log("Hello, World!");  })(); |

 **Anonymous Function**

* These are functions without a name, often used as arguments to other functions.

javascript

|  |
| --- |
| setTimeout(function() {  console.log("Hello, World!");  }, 1000); |

 **Named Function Expression**

javascript

|  |
| --- |
| const greet = function sayHello() {  console.log("Hello, World!");  }; |

 **Constructor Function**

javascript

|  |
| --- |
| function Person(name, age) {  this.name = name;  this.age = age;  }  const person1 = new Person('John', 30); |

 **Method in an Object**

javascript

|  |
| --- |
| const person = {  name: 'John',  greet: function() {  console.log("Hello, " + this.name);  }  };  person.greet(); |
|  |

**DOM**

In JavaScript, "DOM" stands for Document Object Model. It's a programming interface for web documents. The DOM represents the structure of a web page as a tree of objects, where each object corresponds to a part of the page, such as elements, attributes, and text.

Here’s a brief overview of how the DOM is used in JavaScript:

**Key Concepts:**

1. **Document Structure**:
   * The DOM models the HTML or XML document as a tree of nodes. Each node represents an element, attribute, or piece of text.
2. **Accessing Elements**:
   * JavaScript can access and manipulate elements in the DOM using methods like getElementById, getElementsByClassName, querySelector, and querySelectorAll.

|  |
| --- |
| // Get an element by ID  const element = document.getElementById('myElement');  // Get elements by class name  const elements = document.getElementsByClassName('myClass');  // Get an element using a CSS selector  const element = document.querySelector('.myClass'); |

**Manipulating Elements**:

* JavaScript can modify elements' content, attributes, and styles.

|  |
| --- |
| // Change the text content of an element  element.textContent = 'New text';  // Change an element's attribute  element.setAttribute('href', 'https://new-url.com');  // Change an element's style  element.style.backgroundColor = 'blue'; |
|  |

**Event Handling**:

* You can attach event listeners to elements to respond to user actions like clicks, form submissions, etc.

|  |
| --- |
| // Add an event listener  element.addEventListener('click', function() {  alert('Element clicked!');  }); |

A callback is a function passed as an argument to another function

This technique allows a function to call another function

A callback function can run after another function has finished

|  |
| --- |
| JavaScript **SetTimeout and SetInterval** are the only native function in JavaScript that is used to run code asynchronously, it means allowing the function to be executed immediately, there is no need to wait for the current execution completion, it will be for further execution.  JavaScript setTimeout() Method  This method executes a function, after waiting a specified number of milliseconds. |
| window.setTimeout(function, milliseconds); |

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport">  <title>Document</title>  </head>  <body>  <button onclick="setTimeout(gfg, 2000);">  Press me  </button>  <script>  function gfg() {  alert('Welcome to GeeksforGeeks');  }  </script>  </body>  </html> |

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <meta name="viewport">  <title>Document</title>  </head>  <body>  <p>Press the stop button  before the alert is shown</p>  <button onclick="val = setTimeout(gfg, 2000);">  Press me  </button>  <button onclick="clearTimeout(val);">  Stop Execution</button>  <script>  function gfg() {  alert('Welcome to GeeksforGeeks');  }  </script>  </body>  </html> |

# Window local Storage

The localStorage object allows you to save key/value pairs in the browser.

## Note

The localStorage object stores data with no expiration date.

The data is not deleted when the browser is closed, and are available for future sessions.

## Syntax

or just:

localStorage

### Save Data to Local Storage

localStorage.setItem(*key*, *value*);

### Read Data from Local Storage

let lastname = localStorage.getItem(*key*);

### Remove Data from Local Storage

localStorage.removeItem(*key*);

### Remove All (Clear Local Storage)

localStorage.clear();

## Parameters

|  |
| --- |
|  |
| Parameter | Description |
| key | Required. The name of a key. |
| value | Required. The value of the key. |

## Return Value

|  |
| --- |
|  |
| Type | Description |
| An object | A localStorage object. |

# JSON – Introduction

* JSON stands for JavaScript Object Notation
* JSON is a text format for storing and transporting data
* JSON is "self-describing" and easy to understand
* JSON is a lightweight data-interchange format
* JSON is plain text written in JavaScript object notation
* JSON is used to send data between computers
* JSON is language independent \*

\*  
The JSON syntax is derived from JavaScript object notation, but the JSON format is text only.

Code for reading and generating JSON exists in many programming languages.

 JSON format is syntactically similar to the code for creating JavaScript objects. Because of this, a JavaScript program can easily convert JSON data into JavaScript objects.

Since the format is text only, JSON data can easily be sent between computers, and used by any programming language.

JavaScript has a built in function for converting JSON strings into JavaScript objects:

JSON.parse()

Object to string

JavaScript also has a built in function for converting an object into a JSON string:

JSON.stringify()

String to object

## JSON Syntax Rules

JSON syntax is derived from JavaScript object notation syntax:

* Data is in name/value pairs
* Data is separated by commas
* Curly braces hold objects
* Square brackets hold arrays

## Valid Data Types

In JSON, values must be one of the following data types:

* a string
* a number
* an object (JSON object)
* an array
* a boolean
* null

JSON values **cannot**be one of the following data types:

* a function
* a date
* undefined

# JSON.parse()

* A common use of JSON is to exchange data to/from a web server.
* When receiving data from a web server, the data is always a string.
* Parse the data with JSON.parse(), and the data becomes a JavaScript object.

# JSON.stringify()

* A common use of JSON is to exchange data to/from a web server.
* When sending data to a web server, the data has to be a string.
* Convert a JavaScript object into a string with JSON.stringify().