Javascript

1. Introduction of JavaScript

JavaScript is a high-level, interpreted programming language that is one of the core technologies of the World Wide Web, alongside HTML and CSS. It was created by Brendan Eich in 1995 during his time at Netscape Communications.

JavaScript enables interactive web pages and is an essential part of web applications. The vast majority of websites use it, and all major web browsers have a dedicated JavaScript engine to execute it.

2. Key Features of JavaScript

a. **Interpreted Language:** Unlike compiled languages such as C or Java, JavaScript is interpreted, meaning it is executed line by line, which allows for immediate feedback and faster development cycles.

b. **Dynamic Typing:** JavaScript is dynamically typed, which means that types are determined at runtime, and variables can hold different types of values at different times.

c. **Prototypal Inheritance:** Unlike classical inheritance used in languages like Java or C++, JavaScript uses prototypal inheritance. Objects can directly inherit properties and methods from other objects.

d. **First-Class Functions**: Functions in JavaScript are first-class citizens. This means functions can be assigned to variables, passed as arguments to other functions, and returned from other functions.

e. **Event-Driven Programming**: JavaScript supports event-driven programming, which is essential for interactive web applications. Events such as clicks, mouse movements, and keystrokes can trigger JavaScript code execution.

f. **Non-blocking Asynchronous Programming:** JavaScript supports asynchronous programming, primarily through callbacks, promises, and the async/await syntax, which allows for non-blocking code execution.

3. **Uses of JavaScript**

a. **Web Development:** JavaScript is primarily used for client-side scripting to create interactive and dynamic web pages. It manipulates HTML and CSS to update the content and style of web pages in response to user actions.

b. **Server-Side Development:** With the introduction of Node.js, JavaScript can also be used for server-side development. Node.js allows developers to use JavaScript to write server-side code that can handle backend operations.

c. **Mobile Applications:** Frameworks like React Native allow developers to build mobile applications using JavaScript, which can run on both iOS and Android platforms.

d. **Game Development:** JavaScript, along with HTML5, is used for game development. Libraries such as Phaser.js provide tools to create browser-based games.

e. **Desktop Applications:** Technologies like Electron allow developers to create cross-platform desktop applications using JavaScript, HTML, and CSS.

4. **Evolution and Ecosystem**

JavaScript has evolved significantly since its creation. ECMAScript (often abbreviated ES) is the standard specification that JavaScript follows, and updates to the language are released as ECMAScript versions. Some notable versions include ES5 (2009), which brought significant improvements, and ES6 (2015), which introduced classes, modules, arrow functions, and more.

The JavaScript ecosystem is vast, with numerous libraries and frameworks enhancing its capabilities. Some of the popular ones include:

- **React:** A library for building user interfaces, particularly single-page applications.
- **Angular:** A platform and framework for building single-page client applications using HTML and TypeScript.
- **Vue.js:** A progressive framework for building user interfaces.
- **jQuery:** A fast, small, and feature-rich JavaScript library that simplifies HTML document traversal, event handling, and animation.

JavaScript

a. **Lexical structures**

1. Character Set

JavaScript uses Unicode, which means it can include characters from different languages.

2. Tokens

- **Keywords**

Keywords have specific meanings and cannot be used as identifiers. For example:

```
var if = 5; // SyntaxError: Unexpected token if
```

- **Identifiers**

Identifiers are names given to variables, functions, etc.

```
var myVariable = 10;
function myFunction() {
   return myVariable;
}
console.log(myFunction()); // Outputs: 10
```

- **Literals**

Numeric Literals

```
var num = 42;     // Integer literal
var pi = 3.14;    // Floating-point literal
```

String Literals

```
var singleQuote = 'Hello';
var doubleQuote = "World";
console.log(singleQuote + " " + doubleQuote); // Outputs: Hello World
```

Boolean Literals

```
var isTrue = true;
var isFalse = false;
console.log(isTrue); // Outputs: true
```

Null Literal

```
var empty = null;
console.log(empty); // Outputs: null
```

3. Whitespace and Line Breaks

Whitespace is generally ignored, but helps in making the code readable

```
var a = 1;
var b = 2;
var c = a + b; // No syntax error due to extra spaces
console.log(c); // Outputs: 3
```

4. Comments

Single-line Comment

Single-line comments start with //. Everything following // on that line will be treated as a comment and ignored by the JavaScript engine.

```
// This is a single-line comment
var x = 5;
```

Multi-line Comment

Multi-line comments start with /* and end with */. Everything between these markers is treated as a comment and ignored by the JavaScript engine.

```
/*
 This is a multi-line comment
 It can span multiple lines
*/
var y = 10;
```

5. Semicolons

Semicolons are used to terminate statements.

```
var a = 5;
var b = 10;
console.log(a + b); // Outputs: 15
```

6. Operators

Arithmetic Operators

```
var sum = 1 + 2;         // 3
var difference = 5 - 3;  // 2
var product = 2 * 3;     // 6
var quotient = 10 / 2;   // 5
var remainder = 10 % 3;  // 1
```

Assignment Operators

Those operator which assign the value in variable is known as assignment operator.

```
var x = 10;
x += 5; // x = x + 5
console.log(x); // Outputs: 15
```

Comparison Operator

Those operator which compare the values is called Comparison operator.

```
console.log(5 == '5');  // true (loose equality)
```

```
console.log(5 === '5'); // false (strict equality)
console.log(5 != '5');  // false (loose inequality)
console.log(5 !== '5'); // true (strict inequality)
```

Logical Operators

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

7. Keywords and Reserved Words

Keywords cannot be used as identifiers.

```
var for = 10; // SyntaxError: Unexpected token for
```

8. Literals

Object Literals

In JavaScript, an object literal is a way to create an object using a simple and concise syntax. Object literals are useful for creating a single object or for initializing an object with properties and methods

```
var person = {
    name: "John",
    age: 30
};
console.log(person.name); // Outputs: John
```

Array Literals

an array literal is a way to create an array using a straightforward and concise syntax.

```
const fruits = ["apple", "banana", "cherry"];

console.log(fruits[0]); // Output: apple
console.log(fruits[1]); // Output: banana
console.log(fruits[2]); // Output: cherry
```

9. Template Literals

Template literals allow embedding expressions and multi-line strings.

```
var name = "World";
var greeting = `Hello, ${name}!`;
console.log(greeting); // Outputs: Hello, World!

var multiLine = `This is a
multi-line string.`;
console.log(multiLine);
// Outputs:
// This is a
// multi-line string.
```

10. Automatic Semicolon Insertion (ASI)

JavaScript can insert semicolons automatically, but it's good practice to include them explicitly.

```
var x = 10
var y = 20
console.log(x + y) // Outputs: 30

// Better with semicolons
var a = 10;
var b = 20;
console.log(a + b); // Outputs: 30
```

b. Variables

Variables are used to store data that can be used and manipulated throughout a program. In JavaScript, variables can be declared using the var, let, or const keywords.

var:

Scope: var is function-scoped, meaning it is limited to the function within which it is declared. If declared outside a function, it becomes globally scoped.

Hoisting: Variables declared with var are hoisted to the top of their scope and initialized with undefined.

Re-declaration: Variables declared with var can be re-declared within the same scope.

Usage: var is generally avoided in modern JavaScript due to its function-scoping and potential for bugs.

```
function example() {
    console.log(x); // undefined due to hoisting
    var x = 10;
    console.log(x); // 10

    if (true) {
        var x = 20; // Same variable as above, re-declared and re-assigned
        console.log(x); // 20
    }

    console.log(x); // 20, because `var` is function-scoped
}
example();
```

let:

Scope: let is block-scoped, meaning it is limited to the block ({}) within which it is declared.

Hoisting: Variables declared with let are hoisted to the top of their block but are not initialized, leading to a "temporal dead zone" until the declaration is encountered.

Re-declaration: Variables declared with let cannot be re-declared within the same scope.

Usage: let is preferred for variables that need to change their value and are limited to a specific block of code.

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

    if (true) {
        let y = 20; // Different variable, block-scoped
        console.log(y); // 20
    }

    console.log(y); // 10, because `let` is block-scoped
}
example();
```

const:

Scope: const is block-scoped, similar to let.

Hoisting: Variables declared with const are hoisted to the top of their block but are not initialized, leading to a "temporal dead zone" until the declaration is encountered.

Re-declaration: Variables declared with const cannot be re-declared within the same scope.

Immutability: const variables must be initialized at the time of declaration and cannot be re-assigned a new value. However, if the variable is an object or array, its properties or elements can be modified.

Usage: const is preferred for variables that should not change their value after being initialized.

```
function example() {
    // console.log(z); // ReferenceError: Cannot access 'z' before initialization
    const z = 10;
    console.log(z); // 10

    if (true) {
        const z = 20; // Different variable, block-scoped
        console.log(z); // 20
    }

    console.log(z); // 10, because `const` is block-scoped

    // z = 30; // TypeError: Assignment to constant variable
}
example();
```

c. Identifiers in JavaScript

Identifiers in JavaScript are used to name variables, functions, classes, and other entities within the code. They play a crucial role in making the code readable and maintainable. The rules and conventions for

creating identifiers are essential to understand to avoid syntax errors and to write clean code.

Rules for Creating Identifiers

Starting Character:

An identifier must start with a letter (A-Z or a-z), an underscore (_), or a dollar sign ($).

Example: var name;, let _id;, const $element;.

Subsequent Characters:

After the first character, identifiers can include letters, digits (0-9), underscores (_), and dollar signs ($).

Example: let userName1;, const _temp_var;, var $priceValue;.

Case Sensitivity:

Identifiers are case-sensitive, which means myVariable and myvariable would be considered two different identifiers.

Example: let myVariable = 5;, let myvariable = 10;.

Reserved Keywords:

Identifiers cannot be any of the reserved keywords in JavaScript (e.g., break, case, class, catch, etc.).

Example: You cannot name a variable let, if, for, etc.

Incorrect: var for = 5; (This will throw an error)

Naming Conventions

While not strictly enforced by the JavaScript engine, following these conventions can make your code more readable and maintainable:

Camel Case:

Commonly used for variables and function names. The first letter is lowercase, and each subsequent word starts with an uppercase letter.

Example: let userName;, function getUserInfo() {}.

Pascal Case:

Often used for class names and constructor functions. Each word starts with an uppercase letter.

Example: class UserProfile {}, function Person(name, age) {}.

Snake Case:

Sometimes used for constants. Words are separated by underscores and are usually uppercase.

Example: const MAX_USERS = 100;.

d. Reserved Words in JavaScript

Reserved words in JavaScript are keywords that are reserved by the language for its syntactic use. These keywords have predefined meanings and cannot be used as identifiers (variable names, function names, etc.). Using them inappropriately can lead to syntax errors.

- **Categories of Reserved Words**

Keywords: These are predefined words that have special meanings in the language.

Literals: These are reserved for special values.

- **Keywords in Reserve Words**

These are the core reserved words that cannot be used as identifiers:

await break case catch class const
continue debugger default delete do else
enum export extends false finally for
function if import in instanceof let
new null return super switch this
throw true try typeof var void
while with yield

- **Literals Reserve words**

true false null

4.2.3 Expression and operators

Expressions

An expression is any valid unit of code that resolves to a value. JavaScript has several types of expressions, including:

Arithmetic Expressions: Perform arithmetic operations and resolve to numbers.
String Expressions: Combine strings and resolve to a new string.
Logical Expressions: Evaluate conditions and resolve to Boolean values (true or false).
Assignment Expressions: Assign values to variables.

Operators

Operators are special symbols or keywords used to perform operations on operands (values or variables). They can be classified into various categories:

- **Arithmetic Operators**

  Used to perform mathematical operations.

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

console.log(a + b); // Addition: 13
console.log(a - b); // Subtraction: 7
console.log(a * b); // Multiplication: 30
console.log(a / b); // Division: 3.3333...
console.log(a % b); // Modulus (Remainder): 1
console.log(a ** b); // Exponentiation: 1000
```

- **Comparison Operators**

  Used to compare two values and return a Boolean.

```
let x = 10;
let y = 20;

console.log(x == y); // Equal to: false
console.log(x != y); // Not equal to: true
console.log(x === y); // Strict equal to: false
console.log(x !== y); // Strict not equal to: true
console.log(x > y); // Greater than: false
console.log(x < y); // Less than: true
console.log(x >= y); // Greater than or equal to: false
console.log(x <= y); // Less than or equal to: true
```

- **Logical Operators**

Used to combine multiple Boolean expressions or values.

```
let p = true;
let q = false;

console.log(p && q); // Logical AND: false
console.log(p || q); // Logical OR: true
console.log(!p); // Logical NOT: false
```

- **Assignment Operators**

Used to assign values to variables.

```
let value = 5;

value += 2; // Equivalent to value = value + 2
console.log(value); // 7

value -= 1; // Equivalent to value = value - 1
console.log(value); // 6

value *= 3; // Equivalent to value = value * 3
console.log(value); // 18

value /= 2; // Equivalent to value = value / 2
console.log(value); // 9

value %= 4; // Equivalent to value = value % 4
```

```
console.log(value); // 1

value **= 2; // Equivalent to value = value ** 2
console.log(value); // 1
```

- **String Operators**

Used to concatenate strings.

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

let fullName = firstName + " " + lastName;
console.log(fullName); // "John Doe"
```

- **Conditional (Ternary) Operator**

A shorthand for if-else statements.

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

- **Type Operators**

Used to determine the type of a variable or convert values from one type to another.

```
let num = 42;
console.log(typeof num); // "number"
let str = String(num);
console.log(typeof str); // "string"
```

- **Bitwise Operators**

Perform operations on the binary representations of numbers.

```
let m = 5;  // 0101 in binary
let n = 3;  // 0011 in binary

console.log(m & n); // AND: 0001 (1)
console.log(m | n); // OR: 0111 (7)
console.log(m ^ n); // XOR: 0110 (6)
console.log(~m); // NOT: 1010 (-6)
console.log(m << 1); // Left shift: 1010 (10)
console.log(m >> 1); // Right shift: 0010 (2)
console.log(m >>> 1); // Zero-fill right shift: 0010 (2)
```

- **Unary Operators**

Operate on a single operand.

```
let num = 5;
console.log(++num); // Increment: 6
console.log(--num); // Decrement: 5
console.log(typeof num); // Typeof: "number"
```

- **Relational (in, instanceof) Operators**

Used to check if a property exists in an object or if an object is an instance of a specific class.

```
let car = { make: "Toyota", model: "Corolla" };
```

```
console.log("make" in car); // true
console.log("year" in car); // false
```

Statements

In JavaScript, statements are instructions that the JavaScript engine executes. They are the building blocks of a JavaScript program, each performing a specific task.

Different type of javascript operations are:

1. Declaration Statements

Declaration statements are used to declare variables, functions, and classes.

- **Variable Declarations:**

1. var, let, and const are used to declare variables.

Example:

```
var x;          // Declares a variable 'x' that can be globally or function scoped.
let y;          // Declares a variable 'y' that is block-scoped.
const z = 10; // Declares a block-scoped constant 'z' and initializes it with 10.
```

- **Function Declarations:**

1. Used to declare named functions.

Example:

```
function greet(message) {
   console.log(message)
}
```

2. **Expression Statements**

Expression statements perform operations and evaluate to a value.

Assigns a value to a variable.

Example:

```
x = 5;
y = x + 10;
```

3. **Conditional Statements**

JavaScript conditional statements allow you to execute specific blocks of code based on conditions. If the condition is met, a particular block of code will run; otherwise, another block of code will execute based on the condition.

There are several methods that can be used to perform Conditional Statements in JavaScript.

a. **Using if Statement**

The if statement is used to evaluate a particular condition. If the condition holds true, the associated code block is executed.

Syntax:

```
if ( condition ) {
   // If the condition is met,
   //code  will get executed.
}
```

Example:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <h1>Conditional Statement</h1>
    <script>
      let num = 20;
      if (num % 2 === 0) {
        alert("Given number is even number.");
      }
      if (num % 2 !== 0) {
        alert("Given number is odd number.");
      }
    </script>
  </body>
</html>
```

b. Using if-else Statement

The if-else statement will perform some action for a specific condition. Here we are using the else statement in which the else statement is written after the if statement and it has no condition in their code block.

Syntax:

```
if (condition1) {
    // Executes when condition1 is true
 else {
        // Executes when condition2 is true
```

```
    }
}
```

Example:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <h1>Conditional Statement</h1>
    <script>
      let age = 25;
      if (age >= 18) {
        alert("You are eligible of driving licence");
      } else {
        alert("You are not eligible for driving licence");
      }
    </script>
  </body>
</html>
```

C . else if Statement

The else if statement in JavaScript allows handling multiple possible conditions and outputs, evaluating more than two options based on whether the conditions are true or false.

Syntax:

```
if (1st condition) {
   // Code for 1st condition
} else if (2nd condition) {
   // ode for 2nd condition
} else if (3rd condition) {
   // Code for 3rd condition
} else {
   //  code that will execute if all
   // above conditions are false
}
```

Example:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <h1>Conditional Statement</h1>
```

```
    <script>
      const num = 0;
      if (num > 0) {
        alert("Given number is positive.");
      } else if (num < 0) {
        alert("Given number is negative.");
      } else {
        alert("Given number is zero.");
      }
    </script>
  </body>
</html>
```

D. Using Switch Statement (JavaScript Switch Case)

As the number of conditions increases, you can use multiple else-if statements in JavaScript. but when we dealing with many conditions, the switch statement may be a more preferred option.

Syntax:

```
switch (expression) {
    case value1:
        statement1;
        break;
    case value2:
        statement2;
        break;
```

```
    . . .
    case valueN:
        statementN;
        break;
    default:
        statementDefault;
};
```

Example:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <h1>Conditional Statement</h1>
    <script>
      const marks = 85;
      let Branch;
      switch (true) {
        case marks >= 90:
          Branch = "Computer science engineering";
          break;
```

```
        case marks >= 80:
          Branch = "Mechanical engineering";
          break;
        case marks >= 70:
          Branch = "Chemical engineering";
          break;
        case marks >= 60:
          Branch = "Electronics and communication";
          break;
        case marks >= 50:
          Branch = "Civil engineering";
          break;
        default:
          Branch = "Bio technology";
          break;
      }

      alert(`Student Branch name is : ${Branch}`);
    </script>
  </body>
</html>
```

e. Using Ternary Operator (?:)

The conditional operator, also referred to as the ternary operator (?:), is a shortcut for expressing conditional statements in JavaScript.

Syntax:

condition ? value if true : value if false

Example:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <h1>Conditional Statement</h1>
    <script>
      let age = 21;
      const result age >= 18
          ? "You are eligible to vote."
          : "You are not eligible to vote.";
      alert(result);
    </script>
  </body>
</html>
```

4. **Loops Statement in Javascript**
   JavaScript loops are essential for efficiently handling repetitive tasks. They execute a block of code repeatedly as long as a specified condition remains true. These loops are powerful tools for automating tasks and streamlining your code.

Example:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <h1>loops Statement</h1>
    <script>
      for (let i = 0; i < 5; i++) {
        document.write("<h1>Hello World!</h1>");
      }
    </script>
  </body>
</html>
```

a. **JavaScript for Loop**

The JS for loop provides a concise way of writing the loop structure. The for loop contains initialization, condition, and increment/decrement in one line thereby providing a shorter, easy-to-debug structure of looping.

Syntax

```
for (initialization; testing condition; increment/decrement) {
   statement(s)
}
```

Example:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <h1>loops Statement</h1>
    <script>
      let x;
      for (x = 2; x <= 4; x++) {
        document.write(`<h1>value of ${x}</h1>`);
      }
    </script>
  </body>
```

```
</html>
```

b. JavaScript while Loop

The JS while loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement.

Syntax

```
while (boolean condition) {
   loop statements...
}
```

Example:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <h1>loops Statement</h1>
    <script>
      let val = 1;
      while (val < 6) {
        document.write(`<h1>${val}</h1>`);
        val += 1;
```

```
    }
  </script>
 </body>
</html>
```

c. JavaScript do-while Loop

The JS do-while loop is similar to the while loop with the only difference is that it checks for the condition after executing the statements, and therefore is an example of an Exit Control Loop. It executes loop content at least once event the condition is false.

Syntax

```
do {
   Statements...
}
while (condition);
```

Example:

```
<!DOCTYPE html>
<html lang="en">
 <head>
   <meta charset="UTF-8" />
   <meta name="viewport" content="width=device-width, initial-scale=1.0" />
   <title>Document</title>
 </head>
 <body>
   <h1>loops Statement</h1>
```

```
    <script>
      let test = 1;
      do {
        document.write(`<h1>${test}</h1>`);
        test++;
      } while (test <= 5);
    </script>
  </body>
</html>
```

4. JavaScript for-in Loop

JS for-in loop is used to iterate over the properties of an object.

Syntax

```
for(let variable_name in object_name) {
    // Statement
}
```

Example:

```
let myObj = {
   x: 1,
   y: 2,
   z: 3
};
for (let key in myObj) {
    console.log(key, myObj[key]);
}
```

5. JavaScript for-of Loop

JS for-of loop is used to iterate the iterable objects for example – array, object, set and map. It directly iterate the value of the given iterable object and has more concise syntax than for loop.

Syntax:

```
for(let variable_name of  object_name) {
   // Statement
}
```

Example: This example shows the use of for-of loop.

```
let arr = [1, 2, 3, 4, 5];
for (let value of arr) {
   console.log(value);
}
```

Four type of user defined function in js

In JavaScript, a function can be defined without parameters and without a return type. Such functions are typically used to perform a task that does not require input and does not need to provide any output. These functions simply execute the code within their body when called.

Program Examples

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

greet(); // Call the function
```

Arrow Function

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

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

greet(); // Call the function
```

In JavaScript, a function can be defined without parameters but still return a value. These functions perform a task and provide a result back to the caller. The return value is specified using the return statement within the function body.

Program Examples

```
function getCurrentYear() {
    return new Date().getFullYear();
}
```

```
const year = getCurrentYear(); // Call the function and store the return value
console.log(year); // Outputs the current year
```

Arrow Function

```
const getCurrentYear = () => {
    return new Date().getFullYear();
};

const year = getCurrentYear(); // Call the function and store the return value
console.log(year); // Outputs the current year
```

Objects in JavaScript

In JavaScript, objects are containers for named values (properties) that can be of any data type, including other objects, arrays, functions, and primitive data types (like strings and numbers).

There are several ways to create objects in JavaScript:

Object Literal Syntax:

```
let person = {
    firstName: 'Ram',
    lastName: 'Sharma',
    age: 30,
    greet: function() {
        return 'Hello, ' + this.firstName;
```

```
    }
};
```

Using the new Object() Syntax:

```
let person = new Object();
person.firstName = 'Ram ';
person.lastName = 'Sharma';
person.age = 30;
person.greet = function() {
    return 'Hello, ' + this.firstName;
};
```

Accessing Object Properties

You can access object properties using dot notation (objectName.propertyName) or bracket notation (objectName['propertyName']):

```
console.log(person.firstName);
console.log(person['lastName']);
console.log(person.greet());
```

Examples of Built-in Objects

1. Math Object:

The Math object in JavaScript provides mathematical constants and functions. It is not instantiated like other objects but accessed directly:

```
console.log(Math.PI); // Output: 3.141592653589793
console.log(Math.sqrt(16)); // Output: 4 (square root of 16)
```

console.log(Math.random()); // Output: A random number between 0 and 1

2. String Object:

Strings in JavaScript are primitive values, but you can create String objects using the new String() constructor. These objects have methods for manipulating strings:

```
let str = new String('Hello');
console.log(str.length); // Output: 5 (length of the string)
console.log(str.toUpperCase()); // Output: 'HELLO'
console.log(str.charAt(0)); // Output: 'H' (character at index 0)
```

3. Date Object:

The Date object in JavaScript is used for working with dates and times:

```
let currentDate = new Date();
console.log(currentDate); // Output: Current date and time
let specificDate = new Date('2024-06-25T12:00:00');
console.log(specificDate.getFullYear()); // Output: 2024
console.log(specificDate.getMonth()); // Output: 5 (June, because months are zero-indexed)
```

Regular expression

Regular expressions (regex or regexp) in JavaScript are powerful tools for pattern matching and manipulating strings. They provide a concise and flexible way to search, replace, and validate text based on patterns defined using a formal syntax.

1. **Using a Regular Expression Literal:**

let regex = /pattern/; // where "pattern" is the regular expression pattern

let regex = /hello/;

2. **Using the RegExp Constructor:**

let regex = new RegExp('pattern'); // where "pattern" is the regular expression pattern as a string

let regex = new RegExp('hello');

Regular Expression Patterns

Regular expressions consist of:

Literals: Characters that match exactly themselves, like a, 1, or @.

Metacharacters: Special symbols with specific meanings:

^: Matches the start of a string.

$: Matches the end of a string.

Quantifiers: Specify how many times a character or group should occur:

*: Matches zero or more times.

+: Matches one or more times.

? : Matches zero or one time (optional).

{min,max}: Matches at least min and at most max times.

Character Classes: Sets of characters to match:

\d: Any digit (0-9).

\w: Any word character (letters, digits, underscore).

\s: Any whitespace character (space, tab, newline).

[abc]: Matches any character within the brackets.

[^abc]: Matches any character not in the brackets.

Flags

Flags modify how the regular expression behaves:

g (global): Matches all occurrences (not just the first).

i (case insensitive): Ignores case while matching.

m (multiline): Treats beginning (^) and end ($) characters as working across multiple lines.

Practical Uses

Search and Replace: Find specific patterns and replace them with new text.

Validation: Check if user input matches expected formats like emails, phone numbers, or dates.

Data Extraction: Capture specific parts of text for further processing.

Text Manipulation: Splitting strings or formatting text based on patterns.

Matching emails:

let emailRegex = /\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}\b/i;

The provided regex pattern is used to match email addresses. Here's a breakdown of its components:

5. **\b:** A word boundary, ensuring that the email address is matched as a whole word and not as part of a larger string.

6. **[A-Z0.0-9._%+-]+:** Matches one or more characters that can be uppercase letters (A-Z), digits (0-9), dots (.), underscores (_), percent signs (%), plus signs (+), or hyphens (-). This part corresponds to the local part of the email address (the part before the @).
7. **@:** Matches the "@" character.
8. **[A-Z0-9.-]+:** Matches one or more characters that can be uppercase letters (A-Z), digits (0-9), dots (.), or hyphens (-). This part corresponds to the domain name.
9. **\.:** Matches the dot (.) character before the domain suffix.
10. **[A-Z]{2,}:** Matches two or more uppercase letters (A-Z) for the domain suffix (e.g., com, org, net).

Garbage collection

Garbage collection in JavaScript is the process by which the JavaScript engine automatically manages memory. It helps ensure that unused memory (or garbage) is reclaimed and made available for future use, preventing memory leaks and optimizing performance.

How Garbage Collection Works

Automatic Memory Management: JavaScript uses an automatic garbage collection mechanism, meaning developers do not need to explicitly allocate or deallocate memory. Memory management is handled behind the scenes by the JavaScript engine.

Mark-and-Sweep Algorithm: The most common method of garbage collection in JavaScript (and many other languages) is the Mark-and-Sweep algorithm.

Detection of Unused Objects: JavaScript determines whether an object is still needed based on its reachability from the root of the application. If an object cannot be accessed by any part of the code (i.e., it is not reachable), it is considered eligible for garbage collection.

Timing of Garbage Collection: The JavaScript engine decides when to perform garbage collection based on various factors such as available memory, allocation rate, and execution context. Modern engines (like V8 in Chrome or Firefox) optimize garbage collection to minimize interruptions to the main execution thread.

Managing Memory Efficiently

While JavaScript handles memory management automatically, developers can follow best practices to optimize memory usage and improve performance:

Avoid Global Variables: Minimize the use of global variables to reduce the scope of reachability and make it easier for the garbage collector to determine unused objects.

Dispose of Event Listeners: Remove event listeners when they are no longer needed to prevent potential memory leaks, especially when binding events to DOM elements.

Nullifying Unused References: Set object references to null when they are no longer needed to explicitly signal to the garbage collector that the object can be reclaimed.

Objects in JavaScript

In JavaScript, objects are key-value pairs where keys are strings (or Symbols) and values can be any data type, including other objects, arrays, functions, and primitive types (like strings and numbers). Objects are a cornerstone of JavaScript programming, allowing for structured data representation and manipulation.

1. Object Creation

There are multiple ways to create objects in JavaScript:

Object Literal Syntax:

```
let person = {
    name: 'Ram',
    age: 30,
    greet: function() {
        return 'Hello, my name is ' + this.name;
    }
};
```

Using the new Object() Syntax:

Creating objects with new Object() gives you an empty object that you can then add properties and methods to dynamically.

```
let person = new Object();
```

```
person.name = 'Ram';
person.age = 30;
person.greet = function() {
    return 'Hello, my name is ' + this.name;
};

console.log(person.name); // Output: 'Ram'
console.log(person.age); // Output: 30
console.log(person.greet()); // Output: 'Hello, my name is Ram'
```

Constructor Functions

Constructor functions are used to create multiple instances of objects with the same properties and methods.ES6 classes and are widely used in JavaScript.

Constructor functions are essentially regular functions that are used with the new keyword to create instances of objects. They initialize object properties using this.

```
function Person(name, age) {
    this.name = name;
    this.age = age;
    this.greet = function() {
        return 'Hello, my name is ' + this.name;
    };
}

let ram = new Person('Ram', 30);
```

```
console.log(ram.name); // Output: 'Ram'
console.log(ram.age); // Output: 30
console.log(ram.greet()); // Output: 'Hello, my name is Ram'
```

ES6 Classes

ES6 introduced a more structured way to define objects and their behaviors using classes.

Classes in JavaScript are syntactic sugar over constructor functions and prototype-based inheritance. They provide a clearer and more familiar syntax for defining object-oriented patterns.

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

    greet() {
        return 'Hello, my name is ' + this.name;
    }
}

let ram = new Person('Ram', 30);
console.log(ram.name); // Output: 'Ram'
console.log(ram.age); // Output: 30
console.log(ram.greet()); // Output: 'Hello, my name is Ram'
```

2. Properties and Methods

Properties

Properties in JavaScript objects are variables that are attached to the object. They consist of a key (name) and a value, which can be of any data type.

Properties define the characteristics or attributes of an object. They can be accessed, modified, added, and deleted dynamically during the execution of a program.

```
let person = {
    name: 'Ram',
    age: 30,
    city: 'Kathmandu'
};


console.log(person.name); // Output: 'Ram'
person.age = 31; // Modifying property
person.job = 'Developer'; // Adding new property
delete person.city; // Deleting property
```

3. Prototypes and Inheritance

Prototype-based Inheritance

JavaScript objects inherit properties and methods from a prototype. Each object has a prototype from which it inherits methods and properties.

Prototypes allow objects to inherit properties and methods from other objects. They form a chain of inheritance, facilitating code reusability and maintaining object relationships.

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

Person.prototype.greet = function() {
    return 'Hello, my name is ' + this.name;
};

let ram = new Person('Ram', 30);
console.log(john.greet()); // Output: 'Hello, my name is Ram'
```

Object as an associative array:

In JavaScript, objects can indeed be treated as associative arrays, where keys (or property names) are treated as strings and can be used to access values associated with those keys.

Associative Arrays in JavaScript

Object Literal Syntax:

Objects in JavaScript can be created using literal syntax, where keys are specified as strings within curly braces {}.

```
let person = {
    name: 'John',
    age: 30,
```

```
    city: 'New York'
};
```

In this example:

name, age, and city are keys (or properties) of the person object.

'John', 30, and 'New York' are corresponding values.

Accessing Properties:

Properties of an object can be accessed using dot notation (objectName.propertyName) or bracket notation (objectName['propertyName']).

```
console.log(person.name); // Output: 'John'
console.log(person['age']); // Output: 30
```

Adding and Modifying Properties:

Properties can be added or modified after an object is created:

```
person.job = 'Developer'; // Adding a new property
person.age = 31; // Modifying an existing property
```

Deleting Properties:

Properties can be deleted using the delete keyword:

```
delete person.city; // Deleting a property
```

Using Objects as Associative Arrays

Associative arrays refer to using keys that are not necessarily sequential integers (as in typical arrays), but arbitrary strings or symbols. JavaScript objects naturally support this behavior:

Associative arrays in JavaScript refer to using keys that are not limited to numeric indices (like traditional arrays), but can be arbitrary strings or symbols. This means you can use meaningful names as keys to access values stored in an object. JavaScript objects are naturally designed to support this behavior.

```
let car = {
    'make': 'Toyota',
    'model': 'Camry',
    'year': 2020
};


console.log(car['make']); // Output: 'Toyota'
console.log(car.model); // Output: 'Camry'
console.log(car['year']); // Output: 2020
```

Traditional Arrays vs. Associative Arrays:

Traditional Arrays: Use numeric indices (0, 1, 2, ...) to access elements stored in a sequential order.

Associative Arrays (JavaScript Objects): Use arbitrary strings or symbols as keys to access corresponding values stored in an object.

```
// Traditional Array
let traditionalArray = ['apple', 'banana', 'orange'];
console.log(traditionalArray[0]); // Output: 'apple'
```

```
// Associative Array (JavaScript Object)
let fruitPrices = {
    'apple': 1.99,
    'banana': 0.99,
    'orange': 1.49
};

console.log(fruitPrices['apple']); // Output: 1.99
console.log(fruitPrices['banana']); // Output: 0.99
```

Usage in JavaScript Objects:

JavaScript objects allow you to define properties using strings (or symbols), making it easier to create data structures where each key has a meaningful identifier.

This approach is useful for mapping names or identifiers to values, representing configurations, dictionaries, or any scenario where a key-value relationship is needed.

Flexibility and Versatility:

Unlike traditional arrays that are mainly used for ordered collections of elements, associative arrays (JavaScript objects) provide flexibility in how data is organized and accessed based on meaningful keys.

You can dynamically add, modify, and delete properties, making objects versatile for various programming tasks.

Dom and Event Handling

Understanding the Document Object Model (DOM) and event handling in JavaScript is crucial for building interactive and dynamic web applications.

Document Object Model (DOM)

What is the DOM?

The Document Object Model (DOM) is a programming interface for web documents. JavaScript can interact with the DOM to dynamically modify the content, structure, and style of web pages.

DOM, or **Document Object Model**, is a programming interface that represents structured documents like HTML and XML as a tree of objects. It defines how to access, manipulate, and modify document elements using scripting languages like JavaScript.

Why is DOM Required?

HTML is used to structure the web pages and Javascript is used to add behavior to our web pages. When an HTML file is loaded into the browser, the JavaScript cannot understand the HTML document directly. So

it interprets and interacts with the Document Object Model (DOM), which is created by the browser based on the HTML document.

DOM is basically the representation of the same HTML document but in a tree-like structure composed of objects. JavaScript can not understand the tags(<h1>H</h1>) in HTML document but can understand object h1 in DOM.

Key Concepts:

DOM Tree Structure:

Every HTML element, attribute, and text node is represented as an object in the DOM tree.

DOM Nodes:

Element Nodes: Represent HTML elements (e.g., , <p>, <a>).

Text Nodes: Represent text within elements.

Attribute Nodes: Represent attributes of elements (e.g., id, class, src).

Accessing DOM Elements:

You can access and manipulate DOM elements using JavaScript methods like getElementById, getElementsByClassName, getElementsByTagName, querySelector, and querySelectorAll.

```
let element = document.getElementById('myElement');

element.textContent = 'Hello, DOM!';
```

DOM Events:

DOM events are actions or occurrences that happen in the browser (e.g., clicking a button, hovering over an element, submitting a form).

Callback in javaScript

A callback function in JavaScript is a function that is passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

```
function greet(name, callback) {
   console.log('Hello, ' + name);
   callback();
}
function sayGoodbye() {
   console.log('Goodbye!');
}
greet('Alice', sayGoodbye);
```

```
// Output:
// Hello, Alice
// Goodbye!
```

Using Promises

you can use promises or async/await. Here's how you can refactor the previous example using promises:

```
function firstTask() {
    return new Promise((resolve) => {
        setTimeout(() => {
            console.log('First task completed');
            resolve();
        }, 1000);
    });
}

function secondTask() {
    return new Promise((resolve) => {
        setTimeout(() => {
            console.log('Second task completed');
            resolve();
        }, 1000);
    });
}

function thirdTask() {
```

```
    return new Promise((resolve) => {
        setTimeout(() => {
            console.log('Third task completed');
            resolve();
        }, 1000);
    });
}

firstTask()
    .then(secondTask)
    .then(thirdTask)
    .then(() => {
        console.log('All tasks completed');
    });

// Output after 3 seconds:
// First task completed
// Second task completed
// Third task completed
// All tasks completed
```

Event Handling in JavaScript

What are Events?

Events are actions or occurrences that happen in the browser or web page. Examples include mouse clicks, keyboard presses, form submissions, page load, etc. JavaScript allows you to respond to these events by attaching event listeners to DOM elements.

Key Concepts:

Event Listener:

An event listener is a function that listens for a specific event on a DOM element and executes code when that event occurs.

Adding an event listener

```
let button = document.getElementById('myButton');

button.addEventListener('click', function() {
    alert('Button clicked!');
});
```

Types of Events:

Mouse Events: click, mouseover, mouseout, mousemove, etc.

Keyboard Events: keydown, keypress, keyup.

Form Events: submit, change, input.

Window Events: load, resize, scroll.

Custom Events: Events created programmatically using CustomEvent.

Introduction to JSON

JSON stands for JavaScript Object Notation. It is a format for structuring data. This format is used by different web applications to communicate with each other. JSON is the replacement of the XML data exchange format in JSON. It is easy to struct the data compare to XML. It supports data structures like arrays and objects and the JSON documents that are rapidly executed on the server. It is also a Language-Independent format that is derived from JavaScript. The official media type for the JSON is application/json and to save those file .json extension.

Features of JSON:

Easy to understand: JSON is easy to read and write.

Format: It is a text-based interchange format. It can store any kind of data in an array of video, audio, and image anything that you required.

Support: It is light-weighted and supported by almost every language and OS. It has a wide range of support for the browsers approx each browser supported by JSON.

Dependency: It is an Independent language that is text-based. It is much faster compared to other text-based structured data.

JSON Syntax Rules: Data is in name/value pairs and they are separated by commas. It uses curly brackets to hold the objects and square brackets to hold the arrays.

```
{
"Courses": [
      {
         "Name" : "Web Development",
         "Created by" : "BIT",
         "Content" : [ "Java Core", "JSP", "Servlets", "Collections" ]
      },
      {
         "Name" : "Data Structures",
         "also known as" : "Interview Preparation Course",
         "Topics" : [ "Trees", "Graphs", "Maps" ]
      }
   ]
}
```

Advantages of JSON:

- JSON stores all the data in an array so data transfer makes easier. That's why JSON is the best for sharing data of any size even audio, video, etc.
- Its syntax is very easy to use. Its syntax is very small and light-weighted that's the reason that it executes and response in a faster way.
- JSON has a wide range for the browser support compatibility with the operating systems, it doesn't require much effort to make it all browser compatible.
- On the server-side parsing the most important part that developers want, if the parsing will be fast on the server side then the user can get the fast response, so in this case JSON server-side parsing is the strong point compare to others.

Disadvantages of JSON:

- The main disadvantage for JSON is that there is no error handling in JSON, if there was a slight mistake in the JSON script then you will not get the structured data.

- JSON becomes quite dangerous when you used it with some unauthorized browsers. Like JSON service return a JSON file wrapped in a function call that has to be executed by the browsers if the browsers are unauthorized then your data can be hacked.
- JSON has limited supported tools that we can use during JSON development.

Introduction to JQuery

- jQuery is a small and lightweight JavaScript library.
- jQuery is cross-platform.
- jQuery means "write less do more".
- jQuery simplifies AJAX call and DOM manipulation.

jQuery is a small, light-weight and fast JavaScript library. It is cross-platform and supports different types of browsers. It is also referred as ?write less do more? because it takes a lot of common tasks that requires many lines of JavaScript code to accomplish, and binds them into methods that can be called with a single line of code whenever needed. It is also very useful to simplify a lot of the complicated things from JavaScript, like AJAX calls and DOM manipulation.

jQuery Features

Following are the important features of jQuery.

- HTML manipulation
- DOM manipulation
- DOM element selection

- CSS manipulation
- Effects and Animations
- Utilities
- AJAX
- HTML event methods
- JSON Parsing

Why jQuery is required

Sometimes, a question can arise that what is the need of jQuery or what difference it makes on bringing jQuery instead of AJAX/ JavaScript? If jQuery is the replacement of AJAX and JavaScript? For all these questions, you can state the following answers.

It is very fast and extensible.

It facilitates the users to write UI related function codes in minimum possible lines.

It improves the performance of an application.

Browser's compatible web applications can be developed.

It uses mostly new features of new browsers.

jQuery is the most popular and the most extendable. Many of the biggest companies on the web use jQuery.

Some of these companies are

Microsoft

Google

IBM

Netflix

Query Example

jQuery is developed by Google. To create the first jQuery example, you need to use JavaScript file for jQuery. You can download the jQuery file from jquery.com or use the absolute URL of jQuery file.

In this jQuery example, we are using the absolute URL of jQuery file. The jQuery example is written inside the script tag.

```
<!DOCTYPE html>
<html>
<head>
 <title>First jQuery Example</title>



<script src="https://code.jquery.com/jquery-3.7.1.min.js" integrity="sha256-/JqT3SQfawRcv/BIHPThkBvs0OEvtFFmqPF/lYI/Cxo=" crossorigin="anonymous"></script>
 <script type="text/javascript" language="javascript">
 $(document).ready(function() {
 $("p").css("background-color", "cyan");
 });
 </script>
 </head>
<body>
<p>The first paragraph is selected.</p>
<p>The second paragraph is selected.</p>
<p>The third paragraph is selected.</p>
</body>
```

</html>

How to use Selectors

The jQuery selectors can be used single or with the combination of other selectors. They are required at every steps while using jQuery. They are used to select the exact element that you want from your HTML document.

S.No.	Selector	Description
1)	Name:	It selects all elements that match with the given element name.
2)	#ID:	It selects a single element that matches with the given id.
3)	.Class:	It selects all elements that matches with the given class.
4)	Universal(*)	It selects all elements available in a DOM.
5)	Multiple Elements A,B,C	It selects the combined results of all the specified selectors A,B and C.

Different jQuery Selectors

Selector	Example	Description
*	$("*")	It is used to select all elements.
#id	$("#firstname")	It will select the element with id="firstname"

.class	$(".primary")	It will select all elements with class="primary"
class,.class	$(".primary,.secondary")	It will select all elements with the class "primary" or "secondary
element	$("p")	It will select all p elements.
el1,el2,el3	$("h1,div,p")	It will select all h1, div, and p elements.
:first	$("p:first")	This will select the first p element
:last	$("p:last")	This will select he last p element

jQuery Syntax For Event Methods

In jQuery, most DOM events have an equivalent jQuery method.

To assign a click event to all paragraphs on a page, you can do this:

```
$("p").click();
```

The next step is to define what should happen when the event fires. You must pass a function to the event:

```
$("p").click(function(){
  // action goes here!!
});
```

Commonly Used jQuery Event Methods

$(document).ready()

The $(document).ready() method allows us to execute a function when the document is fully loaded. This event is already explained in the jQuery Syntax chapter.

click()

The click() method attaches an event handler function to an HTML element.

The function is executed when the user clicks on the HTML element.

The following example says: When a click event fires on a <p> element; hide the current <p> element:

```
$(document).ready(function(){
  $("p").click(function(){
    $(this).hide();
  });
});
```

dblclick()

The dblclick() method attaches an event handler function to an HTML element.

The function is executed when the user double-clicks on the HTML element:

```
$(document).ready(function(){
  $("p").dblclick(function(){
    $(this).hide();
  });
});
```

mouseenter()

The mouseenter() method attaches an event handler function to an HTML element.

The function is executed when the mouse pointer enters the HTML element:

```
$(document).ready(function(){
```

```
  $("#p1").mouseenter(function(){
    alert("You entered p1!");
  });
});
```

mouseleave()

The mouseleave() method attaches an event handler function to an HTML element.

mousedown()

The mousedown() method attaches an event handler function to an HTML element.

mouseup()

The mouseup() method attaches an event handler function to an HTML element.

hover()

The hover() method takes two functions and is a combination of the mouseenter() and mouseleave() methods.

```
$(document).ready(function(){
  $("#p1").hover(function(){
    alert("You entered p1!");
  },
  function(){
    alert("Bye! You now leave p1!");
  });
});
```

focus()

The focus() method attaches an event handler function to an HTML form field.

```
<!DOCTYPE html>
<html>

<head>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.7.1/jquery.min.js"></script>
<script>
$(document).ready(function(){
  $("input").focus(function(){
    $(this).css("background-color", "yellow");
  });
  $("input").blur(function(){
    $(this).css("background-color", "green");
  });
});
</script>
</head>
<body>

Name: <input type="text" name="fullname"><br>
Email: <input type="text" name="email">
```

```
</body>
</html>
```

The on() Method

The on() method attaches one or more event handlers for the selected elements.

```
<!DOCTYPE html>
<html>
<head>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.7.1/jquery.min.js"></script>
<script>
$(document).ready(function(){
  $("p").on("click", function(){
    $(this).hide();
  });
});
</script>
</head>
<body>

<p>If you click on me, I will disappear.</p>
<p>Click me away!</p>
<p>Click me too!</p>
```

```
</body>
</html>
```

jQuery hide() and show()

With jQuery, you can hide and show HTML elements with the hide() and show() methods:

```
<!DOCTYPE html>
<html>
<head>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.7.1/jquery.min.js"></script>
<script>
$(document).ready(function(){
  $("#hide").click(function(){
    $("p").hide();
  });
  $("#show").click(function(){
    $("p").show();
  });
});
</script>
</head>
<body>
<p>If you click on the "Hide" button, I will disappear.</p>
```

```
<button id="hide">Hide</button>
<button id="show">Show</button>
</body>
</html>
```

```
$(document).ready(function(){
  $("button").click(function(){
    $("p").hide(1000);
  });
});
```

jQuery toggle()

You can also toggle between hiding and showing an element with the toggle() method.

```
<!DOCTYPE html>
<html>
<head>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.7.1/jquery.min.js"></script>
<script>
$(document).ready(function(){
  $("button").click(function(){
    $("p").toggle();
  });
```

```
});
</script>
</head>
<body>
<button>Toggle between hiding and showing the paragraphs</button>
<p>This is a paragraph with little content.</p>
<p>This is another small paragraph.</p>
</body>
</html>
```

Callback Functions

JavaScript statements are executed line by line. However, with effects, the next line of code can be run even though the effect is not finished. This can create errors.

To prevent this, you can create a callback function.

A callback function is executed after the current effect is finished.

Typical syntax: $(selector).hide(speed,callback);

A callback function is executed after the current effect is 100% finished.

```
<!DOCTYPE html>
<html>
<head>
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.7.1/jquery.min.js"
></script>
```

```
<script>
$(document).ready(function(){
  $("button").click(function(){
    $("p").hide("slow", function(){
      alert("The paragraph is now hidden");
    });
  });
});
</script>
</head>
<body>
<button>Hide</button>
<p>This is a paragraph with little content.</p>
</body>
</html>
```

Integrating jQuery in a Project

You can include jQuery in your project by downloading it or by including it via a CDN (Content Delivery Network).

Example 1: Including jQuery via CDN

Include the following line in the <head> section of your HTML file

```
<head>
  <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"
></script>
</head>
```

Example 2: Basic jQuery

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>jQuery Integration Example</title>
  <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
  <script>
    $(document).ready(function() {
      // jQuery code goes here
      $("button").click(function() {
        $("p").text("Hello, jQuery!");
      });
    });
  </script>
</head>
<body>
  <p>Click the button to change this text.</p>
  <button>Click Me</button>
</body>
</html>
```

Saving States with Cookies in JavaScript

Cookies are small pieces of data stored by a web browser that are sent back to the server with every subsequent request. They are used to remember information about the user between sessions or across different pages.

Use of Cookies.

Persistence: Cookies can persist data even after the browser is closed, allowing state to be maintained between sessions.

State Management: They help in maintaining user states, such as login status, preferences, and other personalized settings.

Cross-page Storage: Cookies allow information to be shared across different pages of a website.

Creating and Using Cookies

Creating a Cookie: You can create a cookie using JavaScript by setting the document.cookie property.

Reading a Cookie: To read a cookie, you can access the document.cookie property and parse the string.

Deleting a Cookie: To delete a cookie, you set its expiration date to a past date.

Example: Creating, Reading, and Deleting Cookies

Creating a cookie

```
// Function to set a
 cookie
function setCookie(name, value, days) {
  let date = new Date();
  date.setTime(date.getTime() + (days * 24 * 60 * 60 * 1000));
  let expires = "expires=" + date.toUTCString();
  document.cookie = name + "=" + value + ";" + expires + ";path=/";
}


// Usage
```

setCookie("username", "JohnDoe", 7); // Creates a cookie 'username' with value 'JohnDoe' that expires in 7 days

Reading a Cookie

```
// Function to get a cookie
function getCookie(name) {
  let nameEQ = name + "=";
  let cookiesArray = document.cookie.split(';');
  for(let i = 0; i < cookiesArray.length; i++) {
      let cookie = cookiesArray[i].trim();
      if (cookie.indexOf(nameEQ) === 0)
          return cookie.substring(nameEQ.length, cookie.length);
  }
  return null;
}

// Usage
let username = getCookie("username");
if (username) {
  console.log("Welcome back, " + username);
} else {
  console.log("Welcome, new user!");
}
```

Deleting cookie

```
// Function to delete a cookie
function deleteCookie(name) {
```

```
 document.cookie = name + "=;expires=Thu, 01 Jan 1970 00:00:00 UTC;path=/;";
}

// Usage
deleteCookie("username"); // Deletes the 'username' cookie
```

Using cookies to save user Preferences

Html file

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Cookie Example</title>
  <script src="cookies.js"></script>
</head>
<body>
  <label for="theme">Choose a theme:</label>
  <select id="theme">
    <option value="light">Light</option>
    <option value="dark">Dark</option>
  </select>
  <button onclick="savePreference()">Save Preference</button>

  <script>
    // Function to save theme preference
```

```
    function savePreference() {
      let theme = document.getElementById('theme').value;
      setCookie("theme", theme, 30); // Save theme preference in a cookie for 30 days
      applyTheme();
    }

    // Function to apply theme based on cookie
    function applyTheme() {
      let theme = getCookie("theme");
      if (theme) {
        document.body.className = theme;
        document.getElementById('theme').value = theme;
      }
    }

    // Apply theme on page load
    window.onload = applyTheme;
  </script>
</body>
</html>
```

Css file

```
body.light {
  background-color: white;
  color: black;
}

body.dark {
```

```
  background-color: black;
  color: white;
}
```

Cookies.js

```
// Function to set a cookie
function setCookie(name, value, days) {
  let date = new Date();
  date.setTime(date.getTime() + (days * 24 * 60 * 60 * 1000));
  let expires = "expires=" + date.toUTCString();
  document.cookie = name + "=" + value + ";" + expires + ";path=/";
}

// Function to get a cookie
function getCookie(name) {
  let nameEQ = name + "=";
  let cookiesArray = document.cookie.split(';');
  for(let i = 0; i < cookiesArray.length; i++) {
    let cookie = cookiesArray[i].trim();
    if (cookie.indexOf(nameEQ) === 0) return cookie.substring(nameEQ.length, cookie.length);
  }
  return null;
}

// Function to delete a cookie
function deleteCookie(name) {
  document.cookie = name + "=;expires=Thu, 01 Jan 1970 00:00:00 UTC;path=/;";
```

}

Cookies are a powerful way to store small amounts of data persistently on the client side. They can be used to save user preferences, login information, and other state data across sessions and page loads. By effectively managing cookies, you can enhance the user experience on your website.

localStorage in js

localStorage is a feature in JavaScript that allows web applications to store data locally within the user's browser.

localStorage stores data as key-value pairs. It can only store strings, so objects or arrays need to be converted using JSON.stringify() and JSON.parse().

Setting Data

localStorage.setItem('key', 'value');

This stores a value with a specific key in localStorage.

Getting Data:

const value = localStorage.getItem('key');

Removing Data:

localStorage.removeItem('key');

Example:

// Storing data

localStorage.setItem('username', 'ram_joshi');

// Retrieving data

const username = localStorage.getItem('username');

console.log('Username:', username); // Outputs: Username: john_doe

// Removing data

localStorage.removeItem('username');