JavaScript

ADC502 Web Development

References :
https://eloquentjavascript.net/
https://www.w3schools.com/js/
https://www.javatpoint.com/javascript-tutorial

JavaScript : Introduction

- JavaScript was introduced in 1995 as a way to add programs to web pages in the Netscape Navigator browser.
- The language has since been adopted by all other major graphical web browsers.
- It has made modern web applications possible—applications with which you can interact directly without doing a page reload for every action.
- JavaScript is also used in more traditional websites to provide various forms of interactivity and cleverness.

JavaScript : Introduction

- It is important to note that JavaScript has almost nothing to do with the programming language named Java.
- The similar name was inspired by marketing considerations rather than good judgment.
- When JavaScript was being introduced, the Java language was being heavily marketed and was gaining popularity.
- Someone thought it was a good idea to try to ride along on this success. Now we are stuck with the name.

JavaScript : Introduction

- In the beginning, at the birth of computing, there were no programming languages. Programs looked something like this:

```
00110001 00000000 00000000
00110001 00000001 00000001
00110011 00000001 00000010
01010001 00001011 00000010
00100010 00000010 00001000
01000011 00000001 00000000
01000001 00000001 00000001
00010000 00000010 00000000
01100010 00000000 00000000
```

- That is a program to add the numbers from 1 to 10 together and print out the result: 1 + 2 + ... + 10 = 55.

JavaScript : Introduction

- Here is the same program in JavaScript:

```
let total = 0, count = 1;
while (count <= 10) {
  total += count;
  count += 1;
}
console.log(total);
// → 55
```

JavaScript : Numbers & Fractions

- Numbers
  - In a JavaScript program, they are written as follows: 13
  - Use that in a program, and it will cause the bit pattern for the number 13 to come into existence inside the computer's memory.
  - JavaScript uses a fixed number of bits, 64 of them, to store a single number value. There are only so many patterns you can make with 64 bits, which means that the number of different numbers that can be represented is limited.
- Fractional numbers are written by using a dot: 9.81
  - For very big or very small numbers, you may also use scientific notation by adding an e (for exponent), followed by the exponent of the number.
  - 2.998e8 is 2.998 × 108 = 299,800,000.

JavaScript : Arithmetic

- Arithmetic:
    - Arithmetic operations such as addition or multiplication take two number values and produce a new number from them. Here is what they look like in JavaScript:

```
100 + 4 * 11
```

    - The + and * symbols are called operators. The first stands for addition, and the second stands for multiplication.
    - But as in mathematics, you can change this by wrapping the addition in parentheses.

```
(100 + 4) * 11
```

JavaScript : Arithmetic

- When operators appear together without parentheses, the order in which they are applied is determined by the precedence of the operators.
- The example shows that multiplication comes before addition. The / operator has the same precedence as *. Likewise for + and -. When multiple operators with the same precedence appear next to each other, as in 1 - 2 + 1, they are applied left to right: (1 - 2) + 1.
- The % symbol is used to represent the remainder operation. X % Y is the remainder of dividing X by Y. For example, 314 % 100 produces 14, and 144 % 12 gives 0. The remainder operator’s precedence is the same as that of multiplication and division. You’ll also often see this operator referred to as modulo.

JavaScript : Arithmetic

- Not all operators are symbols. Some are written as words. One example is the typeof operator, which produces a string value naming the type of the value you give it.
- console.log(typeof 4.5)
- // → number
- console.log(typeof "x")
- // → string
- The other operators shown all operated on two values, but typeof takes only one. Operators that use two values are called binary operators, while those that take one are called unary operators. The minus operator can be used both as a binary operator and as a unary operator.
- console.log(- (10 - 2))
- // → -8

JavaScript : Strings

- Strings
  - The next basic data type is the string. Strings are used to represent text. They are written by enclosing their content in quotes.

```
`Down on the sea`
"Lie on the ocean"
'Float on the ocean'
```

- You can use single quotes, double quotes, or backticks to mark strings, as long as the quotes at the start and the end of the string match.
- Almost anything can be put between quotes, and JavaScript will make a string value out of it.

JavaScript : Strings

- Newlines (the characters you get when you press enter) can be included without escaping only when the string is quoted with backticks (`).
- To include such characters in a string, the following notation is used: whenever a backslash (\) is found inside quoted text, it indicates that the character after it has a special meaning.
- This is called escaping the character. A quote that is preceded by a backslash will not end the string but be part of it. When n character occurs after a backslash, it is interpreted as a newline. Similarly, a t after a backslash means a tab character. Take the following string:
- "This is the first line\nAnd this is the second"
- The actual text contained is this:

    This is the first line

    And this is the second

JavaScript : Strings

- Strings cannot be divided, multiplied, or subtracted, but the + operator can be used on them. It does not add, but it concatenates—it glues two strings together. The following line will produce the string "concatenate":

```
"con" + "cat" + "e" + "nate"
```

- Backtick-quoted strings, usually called template literals, can do a few more tricks. Apart from being able to span lines, they can also embed other values.

```
`half of 100 is ${100 / 2}`
```

- When you write something inside ${} in a template literal, its result will be computed, converted to a string, and included at that position. The example produces “half of 100 is 50”.

JavaScript : Strings

Note

- `console.log("half of 100 is ${100 / 2}")`

// → `half of 100 is ${100 / 2}`

- `console.log(‘half of 100 is ${100 / 2}’)`

// → `half of 100 is ${100 / 2}`

- ``console.log(`half of 100 is ${100 / 2}`)``

// → `half of 100 is 50`

JavaScript : Boolean Values

- Boolean values
  - It is often useful to have a value that distinguishes between only two possibilities, like “yes” and “no” or “on” and “off”. For this purpose, JavaScript has a Boolean type, which has just two values, true and false, which are written as those words.
- Comparison
  - Here is one way to produce Boolean values:

```
console.log(3 > 2)
// → true
console.log(3 < 2)
// → false
```

String Comparison

- Strings can be compared in the same way.

```
console.log(“Amit" < “Singh")
// → true
```

- The way strings are ordered is roughly alphabetic but not really what you’d expect to see in a dictionary: uppercase letters are always “less” than lowercase ones, so "Z" < "a", and nonalphabetic characters (!, -, and so on) are also included in the ordering.
- When comparing strings, JavaScript goes over the characters from left to right, comparing the Unicode codes one by one.

Logical Operators

- Logical operators
  - There are also some operations that can be applied to Boolean values themselves. JavaScript supports three logical operators: and, or, and not. These can be used to “reason” about Booleans.
- The && operator represents logical and. It is a binary operator, and its result is true only if both the values given to it are true.

```
console.log(true && false)
// → false
console.log(true && true)
// → true
```

Logical Operators

- The || operator denotes logical or. It produces true if either of the values given to it is true.

```
console.log(false || true)
// → true
console.log(false || false)
// → false
```

- ternary, operating on three values. It is written with a question mark and a colon, like this:

```
console.log(true ? 1 : 2);
// → 1
console.log(false ? 1 : 2);
// → 2
```

Type Conversion

- Automatic type conversion:

```
console.log(8 * null)
// → 0
console.log("5" - 1)
// → 4
console.log("5" + 1)
// → 51
console.log("five" * 2)
// → NaN
console.log(false == 0)
// → true
```

Bindings or Variables

- How does a program keep an internal state? How does it remember things? To catch and hold values, JavaScript provides a thing called a binding, or variable:

```
let caught = 5 * 5;
```

- The special word (keyword) let indicates that this sentence is going to define a binding.
- It is followed by the name of the binding and, if we want to immediately give it a value, by an = operator and an expression.

Bindings or Variables

```
let mood = "light";
console.log(mood);
// → light
mood = "dark";
console.log(mood);
// → dark

var name = "Ada";
const greeting = "Hello ";
console.log(greeting + name);
// → Hello Ada
```

A binding name may include dollar signs ($) or underscores (_) but no other punctuation or special characters.

Keywords

- Words with a special meaning, such as let, are keywords, and they may not be used as binding names.
- There are also a number of words that are “reserved for use” in future versions of JavaScript, which also can’t be used as binding names.

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

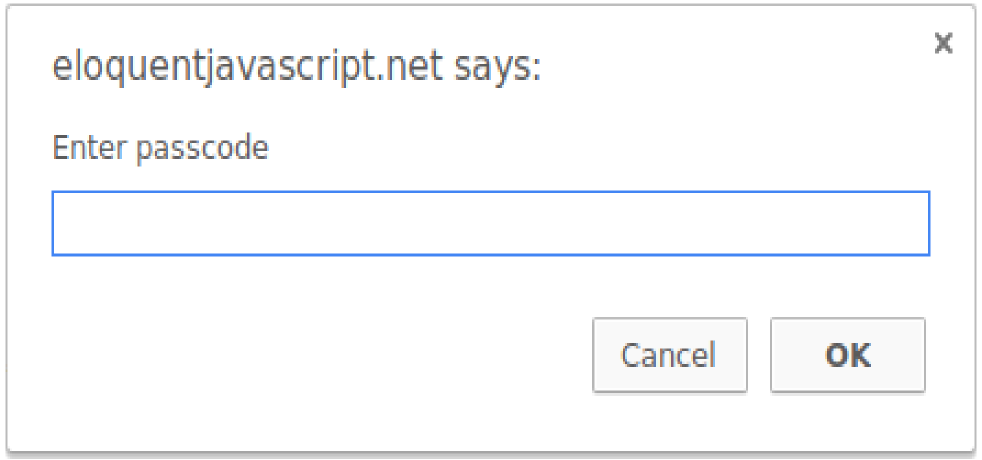
- The full list of keywords and reserved words is rather long.

Working with prompt

- in a browser environment, the binding prompt holds a function that shows a little dialog box asking for user input. It is used like this:

  prompt("Enter passcode");

  A prompt dialog


eloquentjavascript.net says:

Enter passcode

- P.N. prompt() is a function for window object.

Example of the JavaScript prompt() method

```
<!DOCTYPE html>
<html>
  <head>
    <title>JavaScript prompt() method</title>
    <script>
      function fun() {
        var a = prompt("Enter some text", "Hello World");
        if (a != null) {
          document.getElementById("para").innerHTML = "Welcome to " + a;
        }
      }
    </script>
  </head>
  <body style="text-align: center">
    <h2>Example of the JavaScript prompt() method</h2>
    <button onclick="fun()">Click me</button>
    <p id="para"></p>
  </body>
</html>
```

console.log()

- Most JavaScript systems (including all modern web browsers and Node.js) provide a console.log function that writes out its arguments to some text output device.
- In browsers, the output lands in the JavaScript console.
- This part of the browser interface is hidden by default

```
let x = 30;
console.log("the value of x is", x);
// → the value of x is 30
```

- P.N. Console object is a property of window object.

Return Values

- the function Math.max takes any amount of number arguments and gives back the greatest.

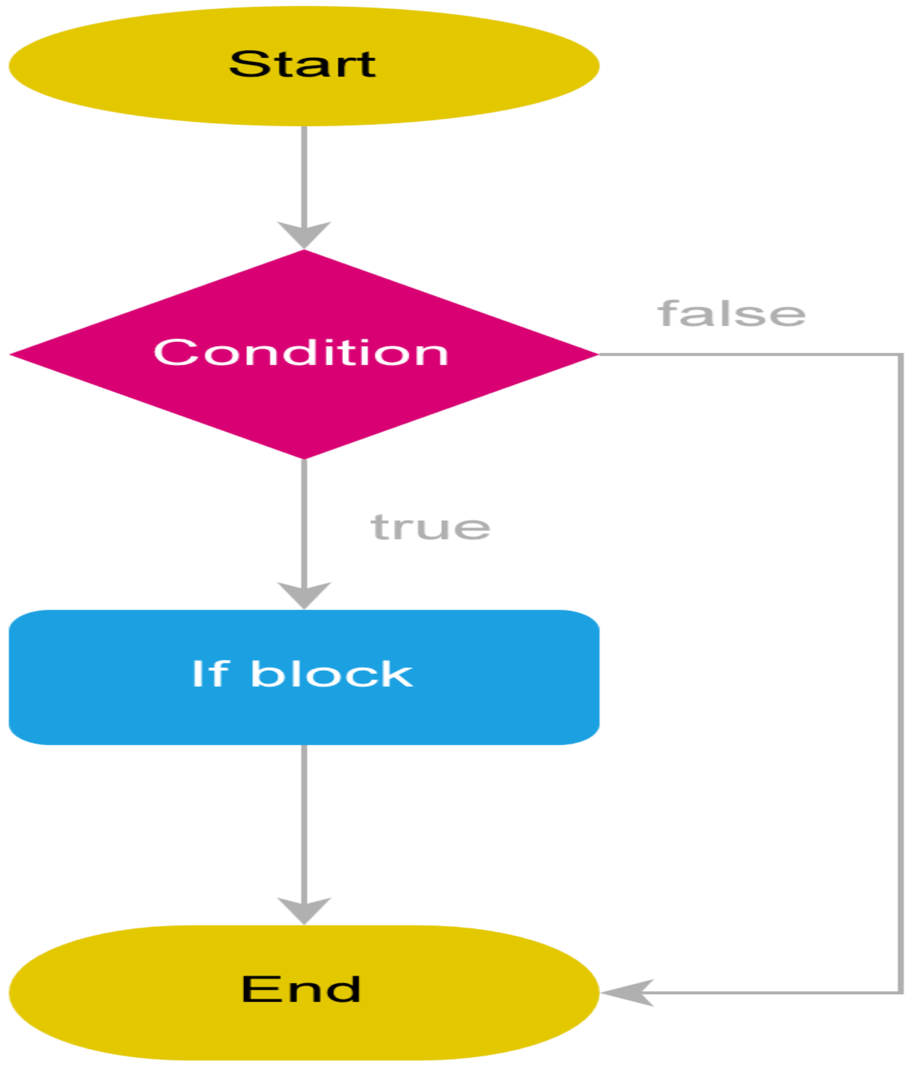
```
console.log(Math.max(2, 4));
// → 4
```

- When a function produces a value, it is said to return that value.
- Anything that produces a value is an expression in JavaScript, which means function calls can be used within larger expressions.
- Here a call to Math.min, which is the opposite of Math.max, is used as part of a plus expression:

```
console.log(Math.min(2, 4) + 100);
// → 102
```

Conditional Statements

```
if( condition )
  statement;
```



Conditional Statements

```
let age = 18;
if (age >= 18) {
  console.log('You can sign up');
}
```

```
Output: You can sign up
```

Conditional Statements

Nested if statement:

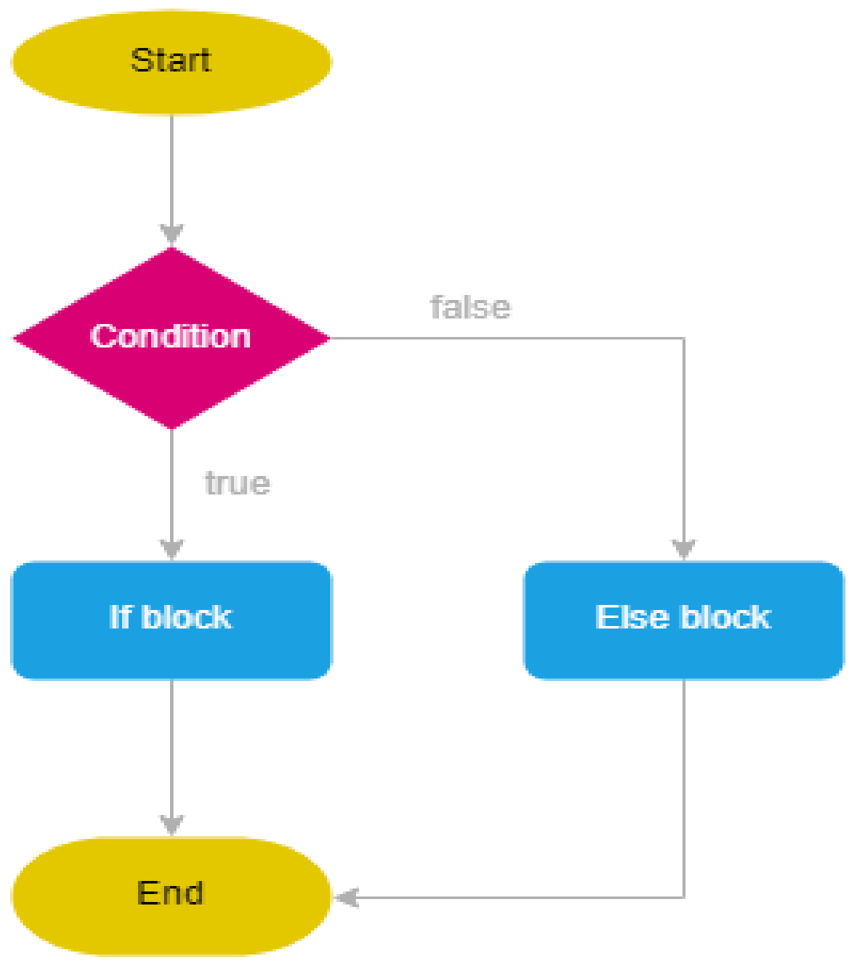
```
let age = 16;
let state = 'CA';

if (state == 'CA') {
  if (age >= 16) {
    console.log('You can drive.');
  }
}
```

Output:

```
You can drive.
```

Conditional Statements

```
if( condition ) {
  // ...
} else {
  // ...
}
```

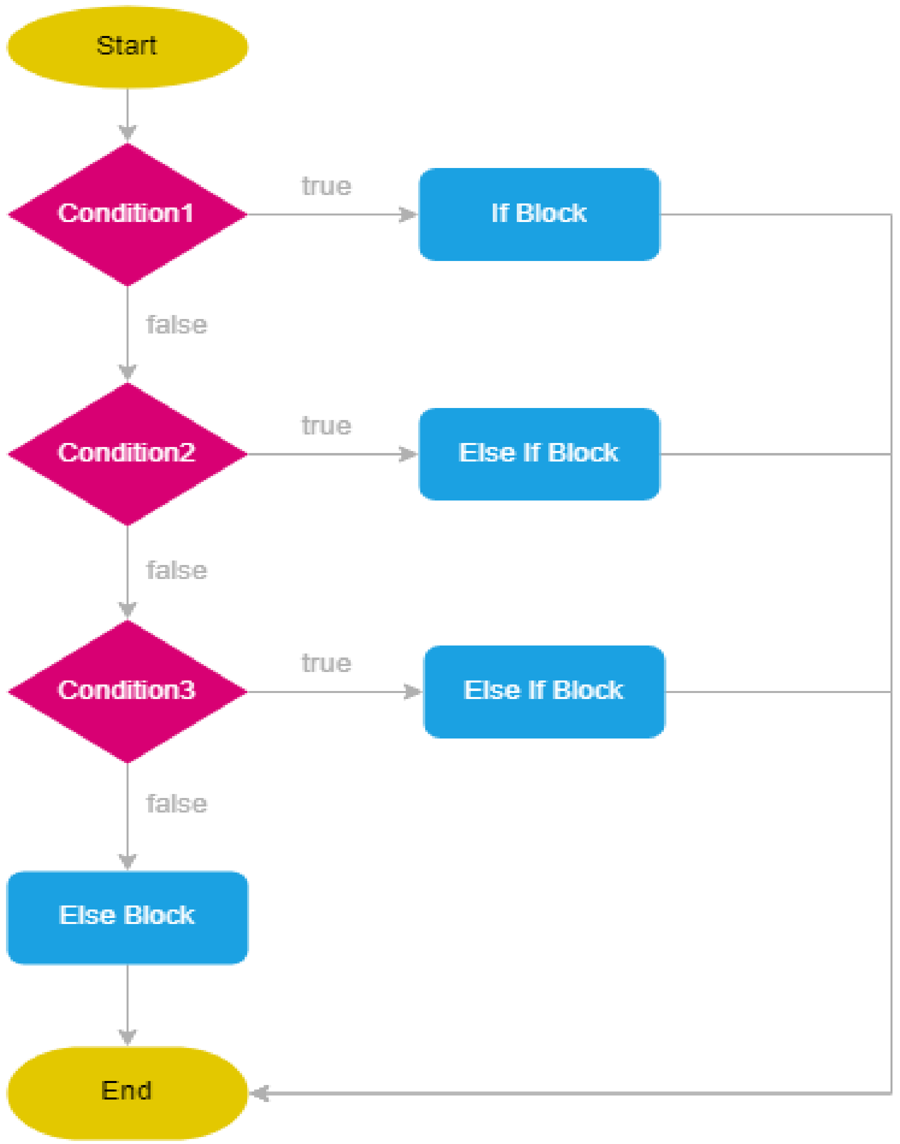


Conditional Statements

```
let age = 18;

if (age >= 18) {
  console.log('You can sign up.');
} else {
  console.log('You must be at least 18 to sign up.');
}
```

Conditional Statements

```
if (condition1) {
  // ...
} else if (condition2) {
  // ...
} else if (condition3) {
  //...
} else {
  //...
}
```



Conditional Statements

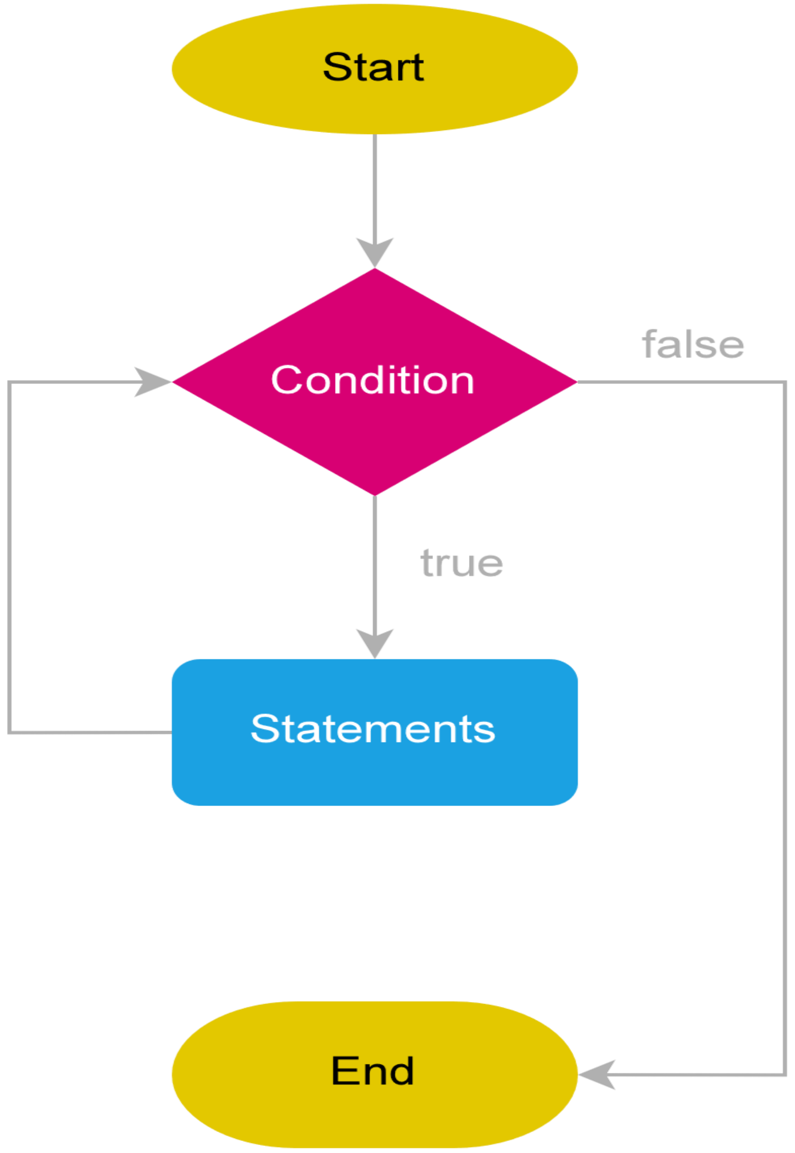
```
let month = 2;
let monthName;
if (month == 1) {
  monthName = 'Jan';
} else if (month == 2) {
  monthName = 'Feb';
} else if (month == 3) {
  monthName = 'Mar';
} else if (month == 4) {
  monthName = 'Apr';
} else {
  monthName = 'Invalid month';
}
console.log(monthName);
```

Loops

while and do loops

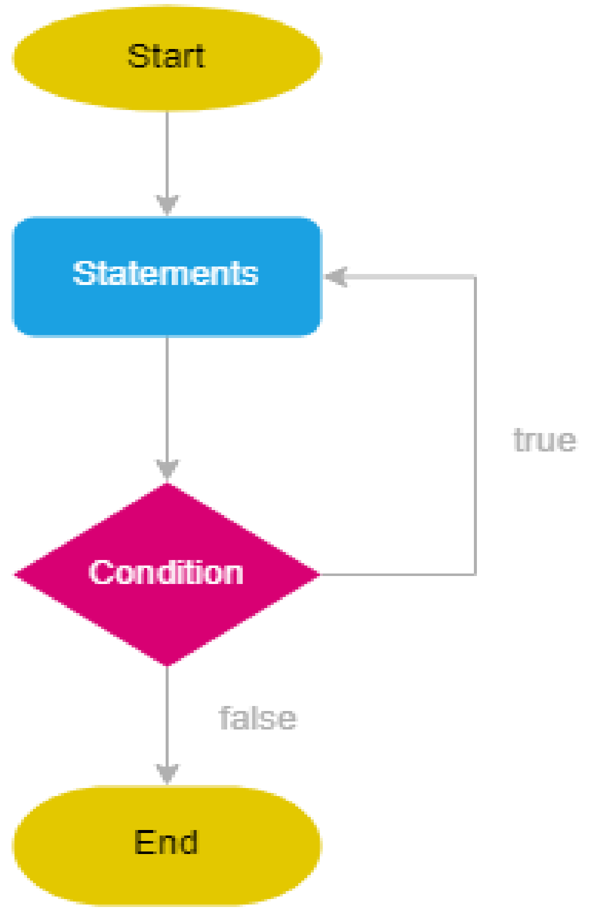
```
while (expression) {
    // statement
}

let count = 1;
while (count < 10) {
    console.log(count);
    count +=2;
}
```

Output:1 3 5 7 9



Loops

```
do {
  statement;
} while(expression);

let count = 0;
do {
  console.log(count);
  count++;
} while (count < 5)

Output: 0 1  2 3 4
```



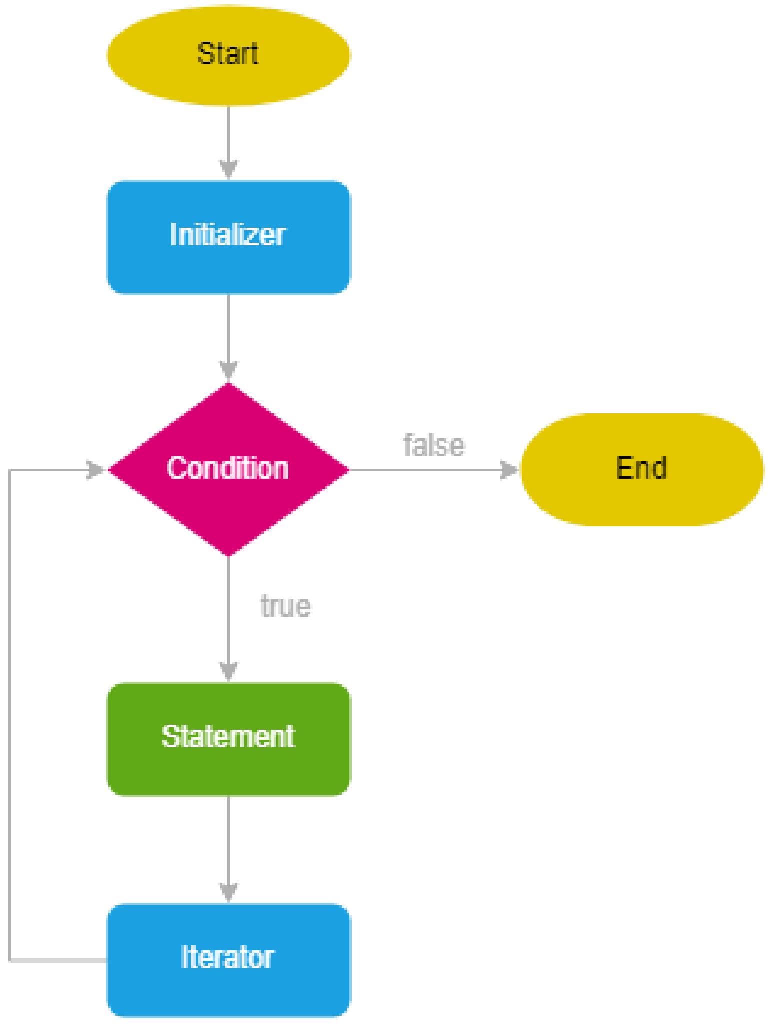
Loops

for loops

```
for (initializer; condition; iterator) {
    // statements
}

for (let i = 1; i < 5; i++) {
  console.log(i);
}
```

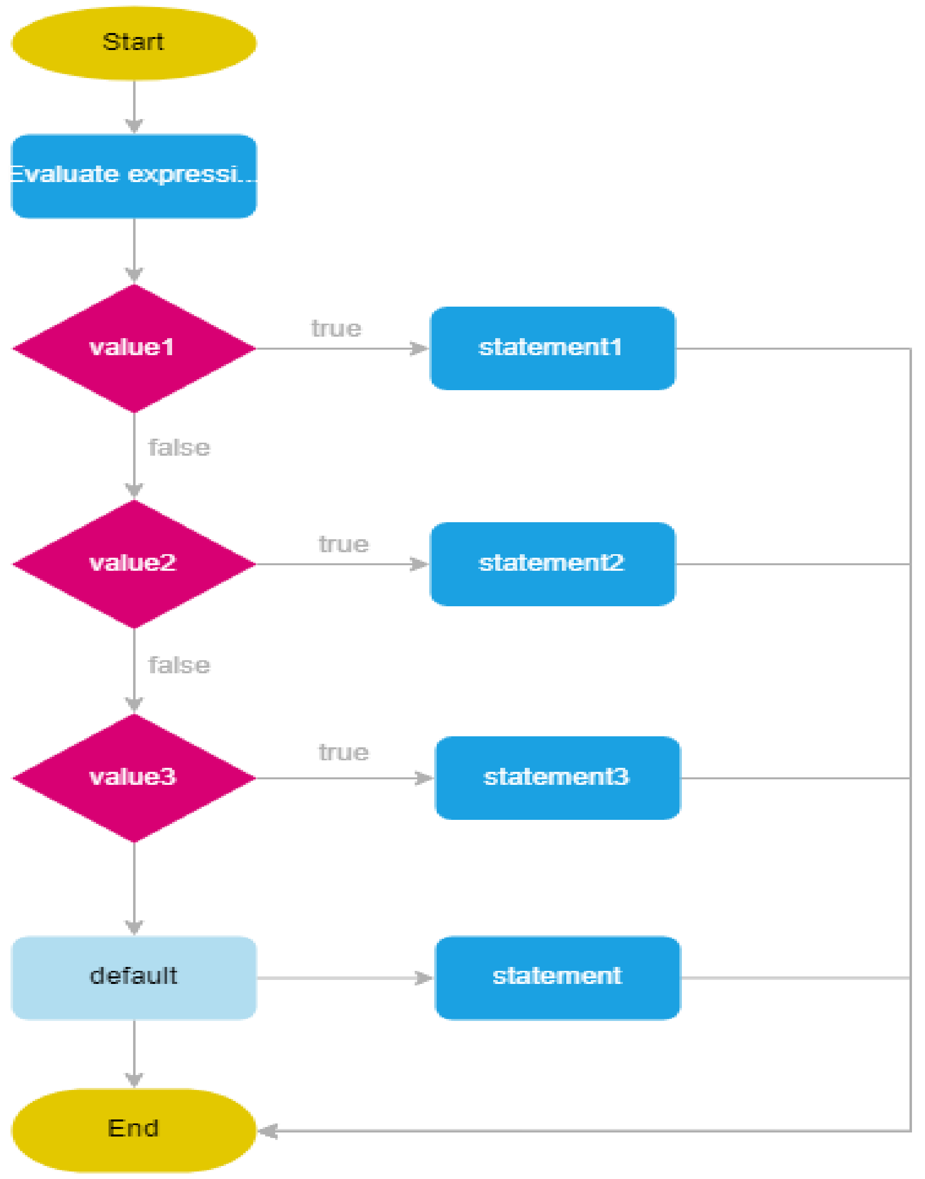
Output:

```
1
2
3
4
```



switch .. case

```
switch (expression) {
    case value1:
        statement1;
        break;
    case value2:
        statement2;
        break;
    case value3:
        statement3;
        break;
    default:
        statement;
}
```



switch .. case

```
let day = 3;
let dayName;
switch (day) {
  case 1:
    dayName = 'Sunday';
    break;
  case 2:
    dayName = 'Monday';
    break;
  case 3:
    dayName = 'Tuesday';
    break;
  default:
    dayName = 'Invalid day';
}
console.log(dayName); // Tuesday
```

Comments

A // comment goes only to the end of the line.

A section of text between /* and */ will be ignored in its entirety, regardless of whether it contains line breaks. This is useful for adding blocks of information about a file or a chunk of program.

```
/*
  I first found this number scrawled on the back of an old
  notebook. Since then, it has often dropped by, showing up in
  phone numbers and the serial numbers of products that I've
  bought. It obviously likes me, so I've decided to keep it.
*/
```

Scope of Variables

- In JavaScript, a variable has two types of scope:
  - Global Scope
  - Local Scope

- Global Scope
  - A variable declared at the top of a program or outside of a function is considered a global scope variable.

Objects

- JavaScript object is a non-primitive data-type that allows you to store multiple collections of data.

JavaScript Object Declaration
The syntax to declare an object is:

```
const object_name = {
  key1: value1,
  key2: value2
}
```

```
const ves = {
    name: 'VESIT',
    address: 'chembur'
}
console.log(typeof person);
```

Accessing Object Properties

1. Using dot Notation

```
const ves = {
   name: 'VESIT',
   address: 'chembur'
}
console.log(ves.name);
```

2. Using bracket Notation

```
const ves = {
   name: 'VESIT',
   address: 'chembur'
}
console.log(ves[‘name’]);
```

Nested Objects

```
const ves = {
    name: 'VESIT',
    address: 'chembur',
    phone: 4563728282
    branch: {
            name: 'AIDS',
            strength: 70,
    }
}
```

Object

JavaScript Object Methods

```
const ves = {
    name: 'VESIT',
    address: 'chembur',
    phone: 4563728282
branch: {
        name: 'AIDS',
        strength: 70,
}
  greeting: function () {
        console.log("welcome to VESIT”);
        }
}
ves.greeting();
```

JavaScript this Keyword

To access a property of an object from within a method of the same object, you need to use the this keyword

```
const ves = {
  name: "VESIT",
  address: "chembur",
  phone: 4563728282,
  types: ["mca", "mba", "pharmacy"],
  branch: {
    name: "AI&DS",
    strength: 70,
  },
  greet: function () {
    console.log("welcome to" + " " + this.name);
    console.log("branch " + this.branch.name);
  },
};
ves.greet();
```

JavaScript this Keyword

However, the function inside of an object can access it's variable in a similar way as a normal function would. For example,

```
const ves = {
   name: 'VESIT',
   address: 'chembur',
   phone: 4563728282,
   types: ['mca','mba','pharmacy'],
   branch: {
      name: 'AIDS',
      strength: 70,
      greeting: function () {
         a= 'ai&ds';
      console.log("welcome to"+ " " + a);
      }
   },
   greet: function () {
      console.log("welcome to"+ " " + this.name);
   }
}
ves.greet();
ves.branch.greeting();
```

JavaScript Constructor Function

- Constructor is a block of codes similar to the method.
- A constructor is a special method that is used to initialize objects.
- Constructor is called when an object of a class is created. It can be used to set initial values for object attributes.
- At the time of calling the constructor, memory for the object is allocated.
- Every time an object is created using the new() keyword, at least one constructor is called.

JavaScript Constructor Function

- What Happens When A Constructor Gets Called?
- When a constructor gets invoked in JavaScript, the following sequence of operations take place:
  - A new empty object gets created.
  - The this keyword begins to refer to the new object and it becomes the current instance object.
  - The new object is then returned as the return value of the constructor.

JavaScript Constructor Function

- How Constructors are Different From Methods ?
  - Constructors must have the same name as the class within which it is defined while it is not necessary for the method.
  - Constructors do not return any type while method(s) have the return type or void if does not return any value.
  - Constructors are called only once at the time of Object creation while method(s) can be called any number of times.

JavaScript Constructor Function

```
// This is how a constructor looks like in Java
Main() {
   System.out.println("Constructor Called:"); // java code
   this.name = "hello";
 }
Main obj = new Main();
System.out.println("The name is " + obj.name);
```

- Note constructors are declared differently in JavaScript

JavaScript Constructor Function

```
// constructor function
function Person () {
    this.name = 'John’;
}

// create object
const person1 = new Person();

// access properties
console.log(person1.name);  // John
```

In a constructor function this does not have a value. It is a substitute for the new object. The value of this will become the new object when a new object is created

JavaScript Constructor Function

- The examples from the previous chapters are limited. They only create single objects.
- Sometimes we need a "blueprint" for creating many objects of the same "type".
- The way to create an "object type", is to use an object constructor function.
- In the example above, function Person() is an object constructor function.
- Objects of the same type are created by calling the constructor function with the new keyword:

JavaScript Constructor Function

```
function Person (person_name, person_age, person_gender) {
    this.name = person_name,
    this.age = person_age,
    this.gender = person_gender,
    this.greet = function () {
        return ('Hi' + ' ' + this.name);
    }
}
const person1 = new Person('John', 23, 'male');
const person2 = new Person('Sam', 25, 'female');
console.log(person1.name); // "John"
console.log(person2.name); // "Sam"
```

JavaScript Constructor Function

Create Objects: Constructor Function Vs Object Literal

Object Literal is generally used to create a single object. The constructor function is useful if you want to create multiple objects. For example,

```
// using object literal
let person = {
   name: 'Sam'
}
// using constructor function
function Person () {
   this.name = 'Sam'
}
let person1 = new Person();
let person2 = new Person();
```

JavaScript Constructor Function

Each object created from the constructor function is unique. You can have the same properties as the constructor function or add a new property to one particular object. For example,

```
// using constructor function
function Person () {
    this.name = 'Sam'
}
let person1 = new Person();
let person2 = new Person();
// adding new property to person1
person1.age = 20;
```

Now this age property is unique to person1 object and is not available to person2 object.

JavaScript Constructor Function

However, if an object is created with an object literal, and if a variable is defined with that object value, any changes in variable value will change the original object. For example,

```
// using object lateral
let person = {
   name: 'Sam'
}
console.log(person.name); // Sam
let student = person; //reference
// changes the property of an object
student.name = 'John';
// changes the origins object property
console.log(person.name); // John
```

JavaScript Constructor Function

- Note: In JavaScript, the keyword class was introduced in ES6 (ES2015) that also allows us to create objects.
- Classes are similar to constructor functions in JavaScript.

JavaScript

- Accessor Property
- In JavaScript, accessor properties are methods that get or set the value of an object. For that, we use these two keywords:
  - get - to define a getter method to get the property value
  - set - to define a setter method to set the property value
- JavaScript Getter
  - In JavaScript, getter methods are used to access the properties of an object. For example,

JavaScript

```
const student = {
   // data property
   firstName: 'Monica',
   // accessor property(getter)
   get getName() {
      return this.firstName;
   }
};
// accessing data property
console.log(student.firstName); // Monica
// accessing getter methods
console.log(student.getName); // Monica
// trying to access as a method
console.log(student.getName()); // error,
//TypeError: student.getName is not a function
```

JavaScript

In the above program, a getter method getName() is created to access the property of an object.

```
get getName() {
   return this.firstName;
}
```

Note: To create a getter method, the get keyword is used.

And also when accessing the value, we access the value as a property.

```
student.getName;
```

When you try to access the value as a method, an error occurs.

```
console.log(student.getName()); // error
```

JavaScript

JavaScript Setter

In JavaScript, setter methods are used to change the values of an object. For example,

```
const student = {
    firstName: 'Monica',
    //accessor property(setter)
    set changeName(newName) {
        this.firstName = newName;
    }
};
console.log(student.firstName); // Monica
// change(set) object property using a setter
student.changeName = 'Sarah';
console.log(student.firstName); // Sarah
```

JavaScript

In the above example, the setter method is used to change the value of an object.

```
set changeName(newName) {
    this.firstName = newName;
}
```

Note: To create a setter method, the set keyword is used.

As shown in the above program, the value of firstName is Monica.

Then the value is changed to Sarah.

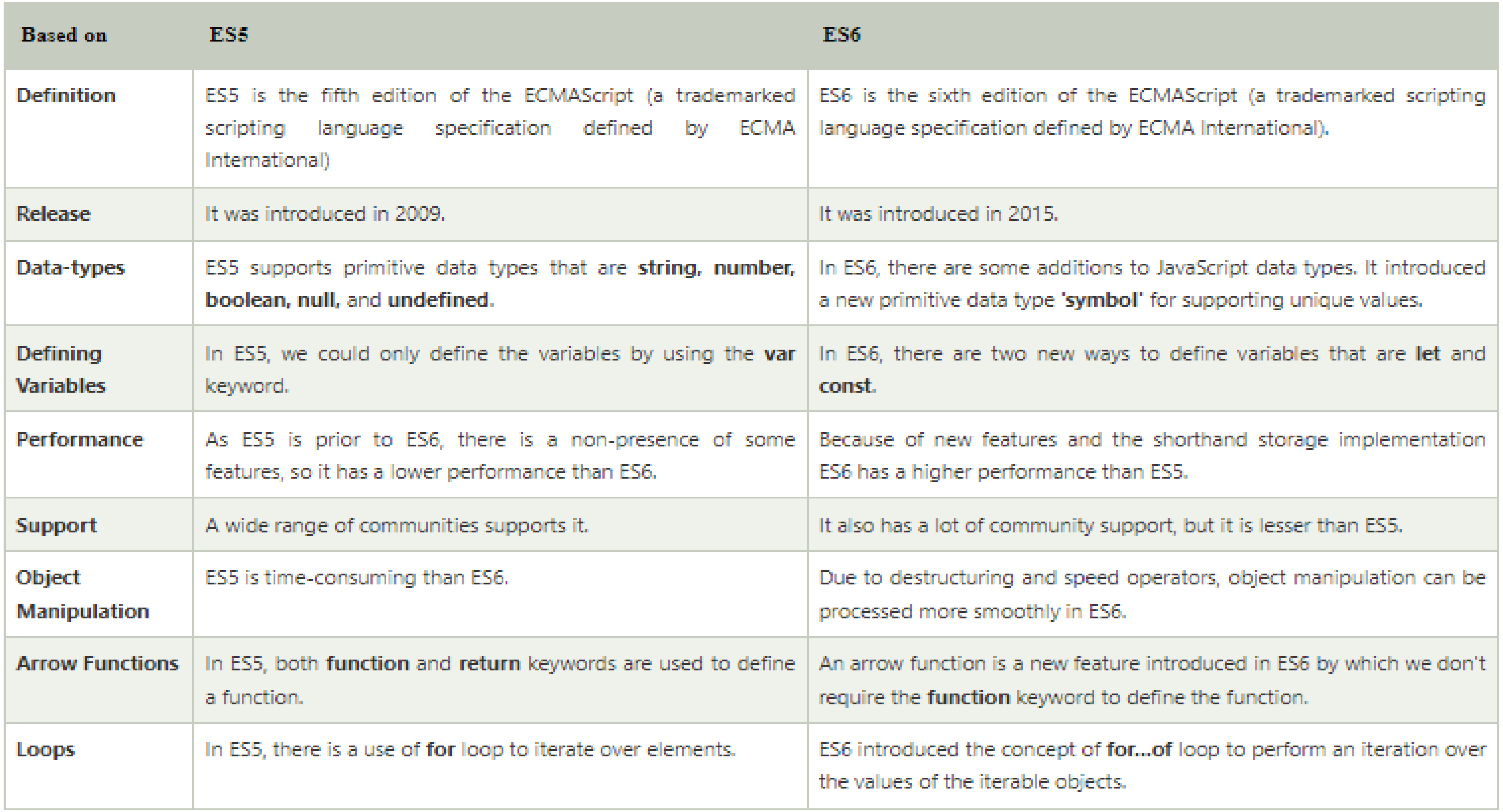
```
student.changeName = 'Sarah';
```

Note: Setter must have exactly one formal parameter.

JavaScript

- ECMAScript is a trademarked scripting language specification that is defined by ECMA International. It was created to standardize JavaScript. The ES scripting language has many implementations, and the popular one is JavaScript. Generally, ECMAScript is used for client-side scripting of the World Wide Web.
- ES5 is an abbreviation of ECMAScript 5 and also known as ECMAScript 2009. The sixth edition of the ECMAScript standard is ES6 or ECMAScript 6. It is also known as ECMAScript 2015. ES6 is a major enhancement in the JavaScript language that allows us to write programs for complex applications.
- Although ES5 and ES6 have some similarities in their nature, there are also so many differences between them.

JavaScript



Based on	ES5	ES6
Definition	ES5 is the fifth edition of the ECMAScript (a trademarked scripting language specification defined by ECMA International)	ES6 is the sixth edition of the ECMAScript (a trademarked scripting language specification defined by ECMA International).
Release	It was introduced in 2009.	It was introduced in 2015.
Data-types	ES5 supports primitive data types that are **string, number, boolean, null,** and **undefined**.	In ES6, there are some additions to JavaScript data types. It introduced a new primitive data type **'symbol'** for supporting unique values.
Defining Variables	In ES5, we could only define the variables by using the **var** keyword.	In ES6, there are two new ways to define variables that are **let** and **const**.
Performance	As ES5 is prior to ES6, there is a non-presence of some features, so it has a lower performance than ES6.	Because of new features and the shorthand storage implementation ES6 has a higher performance than ES5.
Support	A wide range of communities supports it.	It also has a lot of community support, but it is lesser than ES5.
Object Manipulation	ES5 is time-consuming than ES6.	Due to destructuring and speed operators, object manipulation can be processed more smoothly in ES6.
Arrow Functions	In ES5, both **function** and **return** keywords are used to define a function.	An arrow function is a new feature introduced in ES6 by which we don't require the **function** keyword to define the function.
Loops	In ES5, there is a use of **for** loop to iterate over elements.	ES6 introduced the concept of **for...of** loop to perform an iteration over the values of the iterable objects.

ECMAScript Latest edition

- (as on 24 August, 2023)
- 14th Edition – ECMAScript 2023
- The 14th edition, ECMAScript 2023, was published in June 2023.
- This version introduces the toSorted, toReversed, with, findLast, and findLastIndex methods on Array.prototype and TypedArray.prototype, as well as the toSpliced method on Array.prototype;
- added support for #! comments at the beginning of files to better facilitate executable ECMAScript files;
- and allowed the use of most Symbols as keys in weak collections.

JavaScript Arrow Function

- Arrow function is one of the features introduced in the ES6 version of JavaScript. It allows you to create functions in a cleaner way compared to regular functions. For example,
- This function

```
// function expression
let x = function(x, y) {
   return x * y;
}
```

- can be written as

```
// using arrow functions
let x = (x, y) => x * y;
using an arrow function.
```

JavaScript Arrow Function

Arrow Function Syntax

The syntax of the arrow function is:

```
let myFunction = (arg1, arg2, ...argN) => {
    statement(s)
}
```

Here, myFunction is the name of the function

arg1, arg2, ...argN are the function arguments

statement(s) is the function body

If the body has single statement or expression, you can write arrow function as:

```
let myFunction = (arg1, arg2, ...argN) => expression
```

JavaScript Arrow Function

Example 1: Arrow Function with No Argument

If a function doesn't take any argument, then you should use empty parentheses. For example,

```
let greet = () => console.log('Hello');
greet(); // Hello
```

Example 2: Arrow Function with One Argument

If a function has only one argument, you can omit the parentheses. For example,

```
let greet = x => console.log(x);
greet('Hello'); // Hello
```

JavaScript Arrow Function

Multiline Arrow Functions

If a function body has multiple statements, you need to put them inside curly brackets {}. For example,

```
let sum = (a, b) => {
    let result = a + b;
    return result;
}

let result1 = sum(5,7);
console.log(result1); // 12
```

JavaScript for Loops

- In JavaScript, there are three ways we can use a for loop.
  - JavaScript for loop
  - JavaScript for...in loop
  - JavaScript for...of loop
- The for...of loop was introduced in the later versions of JavaScript ES6.

- The for..of loop in JavaScript allows you to iterate over iterable objects (arrays, sets, maps, strings etc).

JavaScript for... in Loop

The for…in loop

The for…in loop is similar to for loop, which iterates through the properties of an object, i.e., when you require to visit the properties or keys of the object, then you can use for…in loop.

```
for (variable_name in object_name) //Here in is the keyword

{

  // statement or block to execute

}
```

In every iteration, one property from the object is assigned to the name of the variable, and this loop continues till all of the object properties get covered.

JavaScript for... in Loop

```
function Mobile(model_no){
    this.Model = model_no;
    this.Color = 'White';
    this.RAM = '4GB';
}
var Samsung = new Mobile("Galaxy");
for(var props in Samsung)
{
    console.log(props+ " : " +Samsung[props]);
}
```

```
Model : Galaxy
Color : White
RAM : 4GB
```

JavaScript for... of Loop

- If you pass the function in the properties of the object then this loop will give you the complete function in the output.
- You can see it's illustration in the following code:

JavaScript for... of Loop

```
function Mobile(model_no){
        this.Model = model_no;
        this.Color = 'White';
        this.RAM = '4GB';
        this.Price = function price()
        {
          console.log(this.model + "Price = Rs. 3300");
        }
}
var Samsung = new Mobile("Galaxy");
for(var props in Samsung)
{
   console.log(props+ " : " +Samsung[props]);
 }
```

```
Model : Galaxy
Color : White
RAM : 4GB
Price : function price()
{
   console.log(this.model + "Price =
Rs. 3300");
}
```

JavaScript for... of Loop

The for…of loop

Unlike the object literals, this loop is used to iterate the iterables (arrays, string, etc.).

Syntax

```
for(variable_name of object_name) // Here of is a keyword
{
  //statement or block to execute
}
```

In every iteration, one property from the iterables gets assigned to the variable_name, and the loop continues till the end of the iteration.

JavaScript for... of Loop

```
var fruits = ['Apple', 'Banana', 'Mango', 'Orange'];
for(let value of fruits)
{
  console.log(value);
}
```

Output

```
Apple
Banana
Mango
Orange
```

JavaScript Classes

- Classes are one of the features introduced in the ES6 version of JavaScript.
- A class is a blueprint for the object. You can create an object from the class.
- You can think of the class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions, you build the house. House is the object.
- Since many houses can be made from the same description, we can create many objects from a class.
- Before ES6, it was hard to create a class in JavaScript. But in ES6, we can create the class by using the class keyword. We can include classes in our code either by class expression or by using a class declaration.
- A class definition can only include constructors and functions. These components are together called as the data members of a class. The classes contain constructors that allocates the memory to the objects of a class. Classes contain functions that are responsible for performing the actions to the objects.

JavaScript Classes

```
class Student {
   constructor(name, age) {
      this.n = name;
      this.a = age;
   }
   stu() {
      console.log("The Name of the student is: ", this.n)
      console.log("The Age of the student is: ",this. a)
   }
}
 let stuObj = new Student('Peter',20);
stuObj.stu();
```

JavaScript Classes

Hoisting

A class should be defined before using it. Unlike functions and other JavaScript declarations, the class is not hoisted. For example,

```
// accessing class
const p = new Person(); // ReferenceError

// defining class
class Person {
  constructor(name) {
    this.name = name;
  }
}
```

JavaScript Classes

- Note: JavaScript class is a special type of function. And the typeof operator returns function for a class.

- For example,

```
class Person {}
console.log(typeof Person); // function
```

JavaScript Class Inheritance

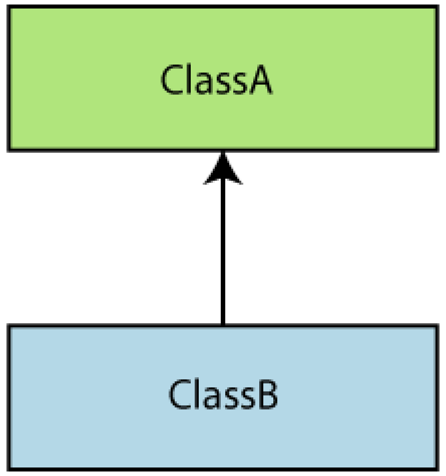
- Inheritance enables you to define a class that takes all the functionality from a parent class and allows you to add more.
- Using class inheritance, a class can inherit all the methods and properties of another class.
- Inheritance is a useful feature that allows code reusability.
- To use class inheritance, you use the extends keyword. For example,

JavaScript Class Inheritance

- Before the ES6, the implementation of inheritance required several steps. But ES6 simplified the implementation of inheritance by using the extends and super keyword.
- Inheritance is the ability to create new entities from an existing one. The class that is extended for creating newer classes is referred to as superclass/parent class, while the newly created classes are called subclass/child class.
- A class can be inherited from another class by using the 'extends' keyword. Except for the constructors from the parent class, child class inherits all properties and methods.

JavaScript Class Inheritance

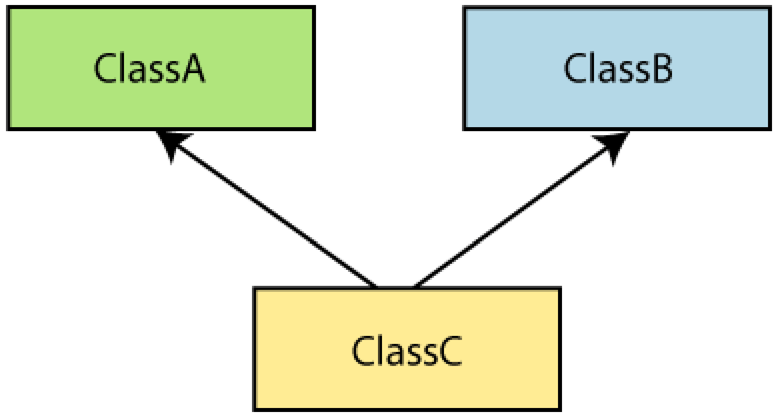
- Types of inheritance
- Inheritance can be categorized as Single-level inheritance, Multiple inheritance, and Multi-level inheritance. Multiple inheritance is not supported in ES6.
- Single-level Inheritance
  - It is defined as the inheritance in which a derived class can only be inherited from only one base class. It allows a derived class to inherit the behavior and properties of a base class, which enables the reusability of code as well as adding the new features to the existing code. It makes the code less repetitive.



Single Inheritance

JavaScript Class Inheritance

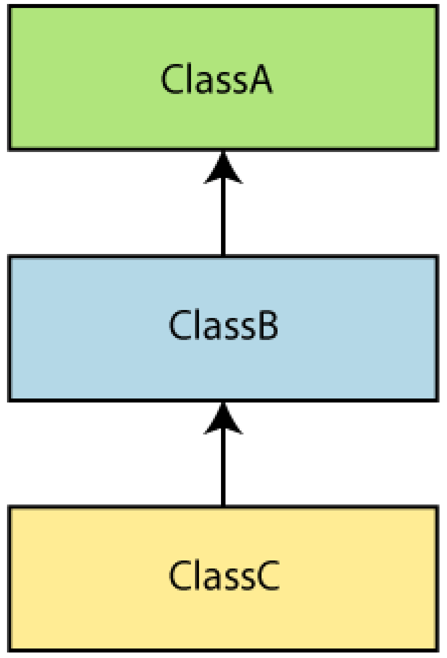
- Multiple Inheritance
- In multiple inheritance, a class can be inherited from several classes. It is not supported in ES6.



Multiple Inheritance

JavaScript Class Inheritance

- Multi-level Inheritance
- In Multi-level inheritance, a derived class is created from another derived class. Thus, a multi-level inheritance has more than one parent class.



Multilevel Inheritance

JavaScript Class Inheritance

```
'use strict'
class Student {
  constructor(a) {
   this.name = a;
  }
}
class User extends Student {
  show() {
    console.log("The name of the student is:  "+this.name)
  }
}
var obj = new User('Sahil');
obj.show()
```

In the above example, we have declared a class student. By using the extends keyword, we can create a new class User that shares the same characteristics as its parent class Student. So, we can see that there is an inheritance relationship between these classes.

Output

```
The name of the student is:  Sahil
```

JavaScript Class Inheritance

```
class Animal{
   eat(){ console.log("eating...");  }}
   class Dog extends Animal{
    bark(){ console.log("barking..."); }}
   class BabyDog extends Dog{
    weep(){console.log("weeping..."); }}
   var d=new BabyDog();
   d.eat();
   d.bark();
   d.weep();
// Output
eating...
barking...
weeping...
```

JavaScript Class Inheritance

- The super keyword
- It allows the child class to invoke the properties, methods, and constructors of the immediate parent class. It is introduced in ECMAScript 2015 or ES6. The super.prop and super[expr] expressions are readable in the definition of any method in both object literals and classes.

- Syntax

- super(arguments);
- In this example, the characteristics of the parent class have been extended to its child class. Both classes have their unique properties. Here, we are using the super keyword to access the property from parent class to the child class.

JavaScript Class Inheritance

```
class Car {
  constructor() {
    console.log('This is a car')
  }
}
const myCar = new Car() //'This is a car'
class Tesla extends Car {
  constructor() {
    super()
    console.log('This is a Tesla')
  }
}
const myCar = new Tesla()
```

```
'This is a car'
'This is a Tesla'
```

JavaScript Class Inheritance

```
class Parent {
  show() {
    console.log("It is the show() method from the parent class");
  }
}
class Child extends Parent {
  show() {
    super.show();
    console.log("It is the show() method from the child class");
  }
}
var obj = new Child();
obj.show();
```

```
It is the show() method from the
parent class
It is the show() method from the
child class
```

JavaScript Iterators and Iterables

- JavaScript Iterables and Iterators
  - JavaScript provides a protocol to iterate over data structures. This protocol defines how these data structures are iterated over using the for...of loop.
- The concept of the protocol can be split into:
  - iterable
  - iterator
- The iterable protocol mentions that an iterable should have the Symbol.iterator key.

JavaScript Iterators and Iterables

- JavaScript Iterables
  - The data structures that have the Symbol.iterator() method are called iterables. For example, Arrays, Strings, Sets, etc.
- JavaScript Iterators
  - An iterator is an object that is returned by the Symbol.iterator() method.
  - The iterator protocol provides the next() method to access each element of the iterable (data structure) one at a time.

JavaScript Iterators and Iterables

Let's look at an example of iterables having Symbol.Iterator()

```
const arr = [1, 2 ,3];
// calling the Symbol.iterator() method
const arrIterator = arr[Symbol.iterator]();
// gives Array Iterator
console.log(arrIterator);
const str = 'hello';
// calling the Symbol.iterator() method
const strIterator = str[Symbol.iterator]();
// gives String Iterator
console.log(strIterator);
```

Output

```
Array Iterator {}
StringIterator {}
```

Here, calling the Symbol.iterator() method of both the array and string returns their respective iterators.

JavaScript Iterators and Iterables

Iterate Through Iterables

You can use the for...of loop to iterate through these iterable objects. You can iterate through the Symbol.iterator() method like this:

```
const number = [ 1, 2, 3];

for (let n of  number[Symbol.iterator]()) {
   console.log(n);
}
```

Output

```
1
2
3
```

JavaScript Iterators and Iterables

Or you can simply iterate through the array like this:

```
const number = [ 1, 2, 3];

for (let n of  number) {
   console.log(n);
}
```

Here, the iterator allows the for...of loop to iterate over an array and return each value.

JavaScript Iterators and Iterables

- JavaScript next() Method
- The iterator object has a next() method that returns the next item in the sequence.
- The next() method contains two properties: value and done.
- value
    - The value property can be of any data type and represents the current value in the sequence.
- done
    - The done property is a boolean value that indicates whether the iteration is complete or not. If the iteration is incomplete, the done property is set to false, else it is set to true.

JavaScript Iterators and Iterables

```
const arr = ['h', 'e', 'l', 'l', 'o'];

let arrIterator = arr[Symbol.iterator]();
console.log(arrIterator.next()); // {value: "h", done: false}
console.log(arrIterator.next()); // {value: "e", done: false}
console.log(arrIterator.next()); // {value: "l", done: false}
console.log(arrIterator.next()); // {value: "l", done: false}
console.log(arrIterator.next()); // {value: "o", done: false}
console.log(arrIterator.next()); // {value: undefined, done: true}
```

You can call next() repeatedly to iterate over an arrIterator object.

- The next() method returns an object with two properties: value and done.
- When the next() method reaches the end of the sequence, then the done property is set to false.

JavaScript Iterators and Iterables

```
const arr = ['h', 'e', 'l', 'l', 'o'];

for (let i of arr) {
   console.log(i);
}
h
e
l
l
o
```

The for...of loop does exactly the same as the program above.

The for...of loop keeps calling the next() method on the iterator. Once it reaches done:true, the for...of loop terminates.

JavaScript Iterators and Iterables

User Defined Iterator

You can also create your own iterator and call next() to access the next element. For example,

```
function displayElements(arr) {
    // to update the iteration
    let n = 0;
    return {
        // implementing the next() function
        next() {
            if(n < arr.length) {
                return {
                    value: arr[n++],
                    done: false}}
```

JavaScript Iterators and Iterables

```
   return {
            value: undefined,
            done: true}}}}
const arr = ['h', 'e', 'l', 'l', 'o'];
const arrIterator = displayElements(arr);
console.log(arrIterator.next());
console.log(arrIterator.next());
console.log(arrIterator.next());
console.log(arrIterator.next());
console.log(arrIterator.next());
console.log(arrIterator.next());
```

JavaScript Iterators and Iterables

- Output

```
{value: "h", done: false}
{value: "e", done: false}
{value: "l", done: false}
{value: "l", done: false}
{value: "o", done: false}
{value: undefined, done: true}
```

- In the above program, we have created our own iterator. The displayElements() function returns value and done property.
- Each time the next() method is called, the function gets executed once and displays the value of an array.
- Finally, when all the elements of an array are exhausted, the done property is set to true, with value as undefined.

JavaScript Generators

- In JavaScript, generators provide a new way to work with functions and iterators.
  - Using a generator, you can stop the execution of a function from anywhere inside the function and continue executing code from a halted position
- Create JavaScript Generators
  - To create a generator, you need to first define a generator function with function* symbol. The objects of generator functions are called generators.

JavaScript Generators

```
// define a generator function
function* generator_function() {
   ... .. ...
}


// creating a generator
const generator_obj = generator_function();
```

- Note: The generator function is denoted by *. You can either use function* generatorFunc() {...} or function *generatorFunc(){...} to create them.

JavaScript Generators

Using yield to Pause Execution

As mentioned above, you can pause the execution of a generator function without executing the whole function body. For that, we use the yield keyword. For example,

```
// generator function
function* generatorFunc() {
    console.log("1. code before the first yield");
    yield 100;
   console.log("2. code before the second yield");
    yield 200;
}
// returns generator object
const generator = generatorFunc();
console.log(generator.next());
```

JavaScript Generators

Output

```
1. code before the first yield
{value: 100, done: false}
```

Here,

- A generator object named generator is created.
- When generator.next() is called, the code up to the first yield is executed. When yield is encountered, the program returns the value and pauses the generator function.
- Note: You need to assign generator objects to a variable before you use it.

JavaScript Generators

- Working of multiple yield Statements
- The yield expression returns a value. However, unlike the return statement, it doesn't terminate the program. That's why you can continue executing code from the last yielded position. For example,

```
function* generatorFunc() {
    console.log("1. code before first yield");
    yield 100;
   console.log("2. code before the second yield");
    yield 200;
    console.log("3. code after the second yield");
}
const generator = generatorFunc();
console.log(generator.next());
console.log(generator.next());
console.log(generator.next());
```

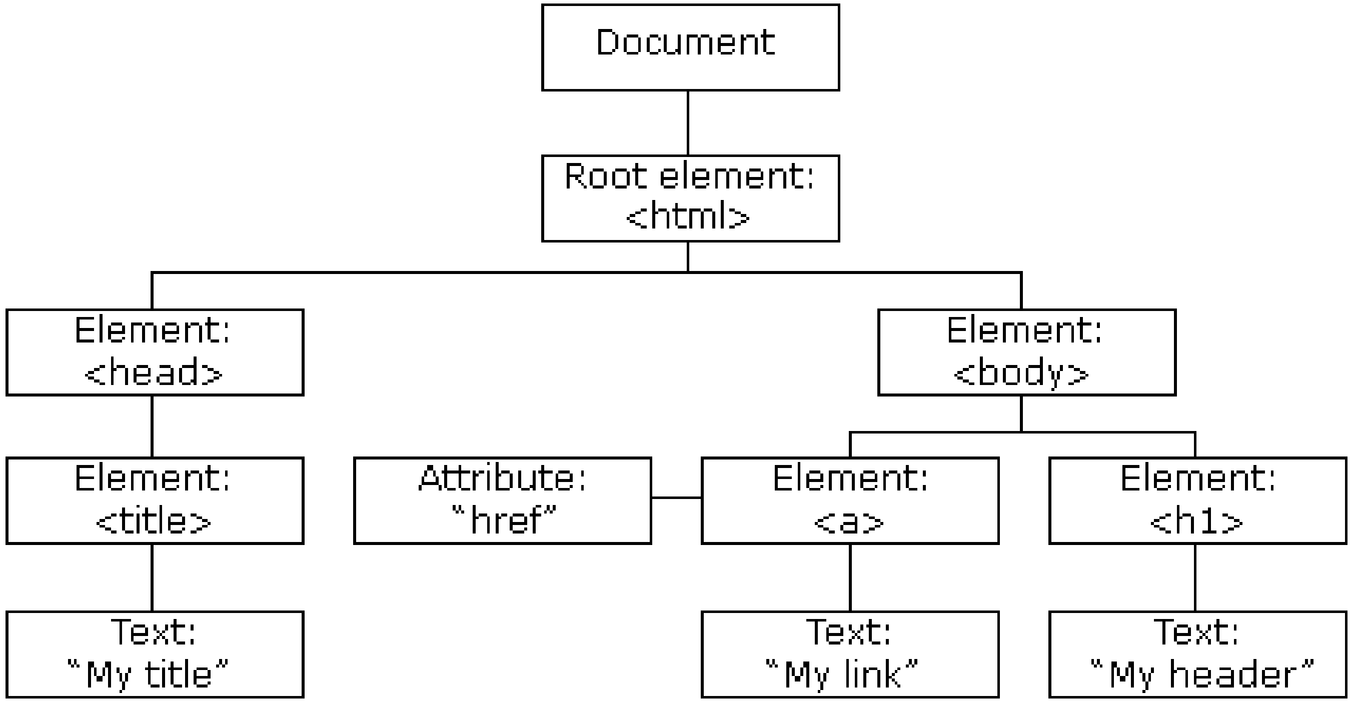
BOM and DOM

Document Object Model

- The Document Object Model (DOM) is a programming interface for HTML and XML(Extensible markup language) documents.
- It defines the logical structure of documents and the way a document is accessed and manipulated.
- DOM is a way to represent the webpage in the structured hierarchical way so that it will become easier for programmers and users to glide through the document.
- With DOM, we can easily access and manipulate tags, IDs, classes, Attributes or Elements using commands or methods provided by Document object.

Document Object Model

- When a web page is loaded, the browser creates a Document Object Model of the page.
- The HTML DOM model is constructed as a tree of Objects. E.g.



Document Object Model

- Using DOM JavaScript can
  - change all the HTML elements in the page
  - change all the HTML attributes in the page
  - change all the CSS styles in the page
  - remove existing HTML elements and attributes
  - add new HTML elements and attributes
  - react to all existing HTML events in the page
  - create new HTML events in the page

The DOM Programming Interface

- The HTML DOM can be accessed with JavaScript (and with other programming languages).
- In the DOM, all HTML elements are defined as objects.
- The programming interface is the properties and methods of each object.
    - A property is a value that one can get or set.
    - A method is an action one can do.

DOM Example

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

In the example above,

- getElementById is a **method**
- innerHTML is a **property**.

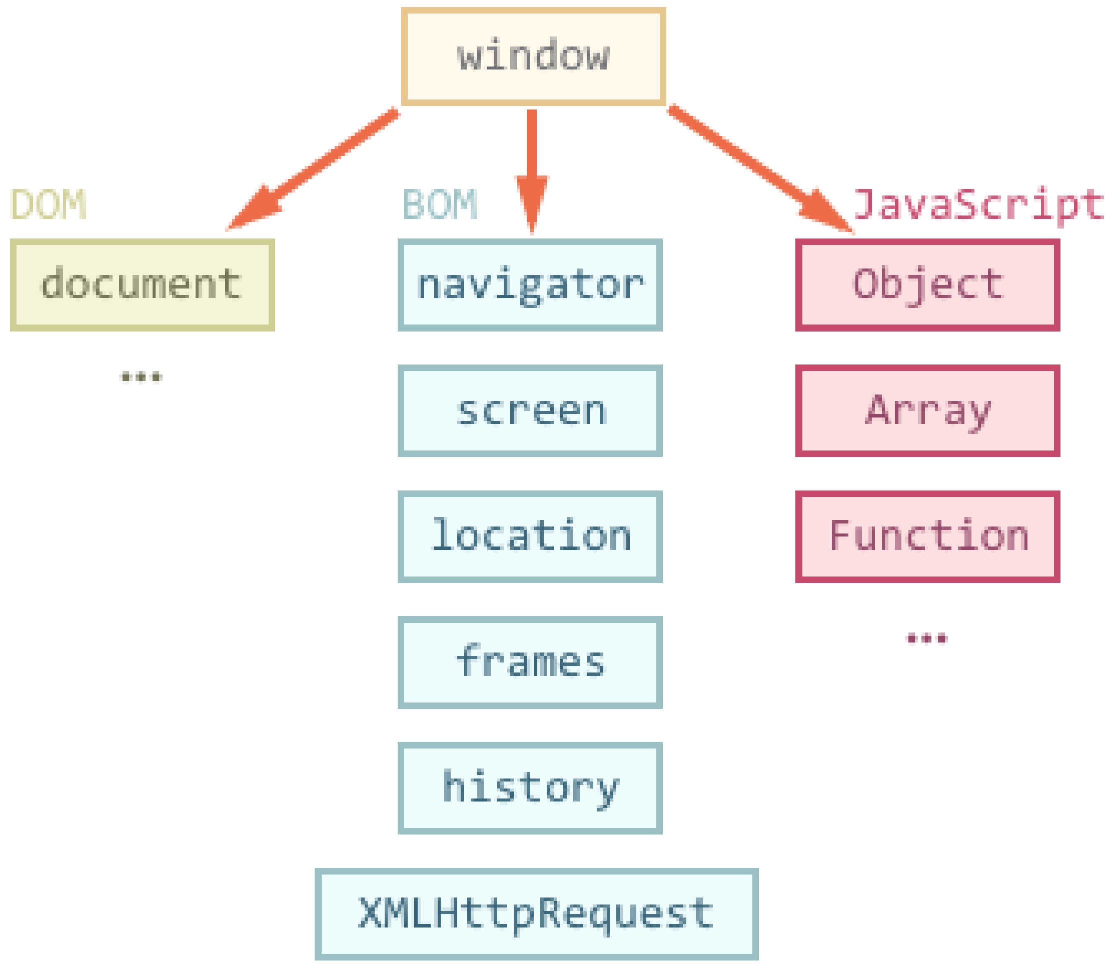
Finding and Changing HTML Elements using DOM

- Finding HTML Elements
    - document.getElementById(id)
    - document.getElementsByTagName(name)
    - document.getElementsByClassName(name)
- Changing HTML Elements
    - Property
        - element.innerHTML = new html content
        - element.attribute = new value
        - element.style.property = new style
    - Method
        - element.setAttribute(attribute, value)

Browser Object Model

- The Browser Object Model (BOM) is a browser-specific convention referring to all the objects exposed by the web browser. The BOM allows JavaScript to “interact with” the browser.
- A window object
  - is created automatically by the browser itself.
  - represents the browser window and all its corresponding features.
  - is the default object of browser
- Window is the object of browser, it is not the object of javascript.

The Window Object



- The window object is supported by all browsers.
- It represents the browser's window.
- All global JavaScript objects, functions, and variables automatically become members of the window object.
  - Global variables are properties of the window object.
  - Global functions are methods of the window object.
- Even the document object (of the HTML DOM) is a property of the window object.

Determining Window Size

- Two properties can be used to determine the size of the browser window.
- Both properties return the sizes in pixels:
  - window.innerHeight - the inner height of the browser window (in pixels)
  - window.innerWidth - the inner width of the browser window (in pixels)
- The browser window (the browser viewport) is NOT including toolbars and scrollbars.
- Example
  - let w = window.innerWidth;
  - let h = window.innerHeight;

Browser Object Model

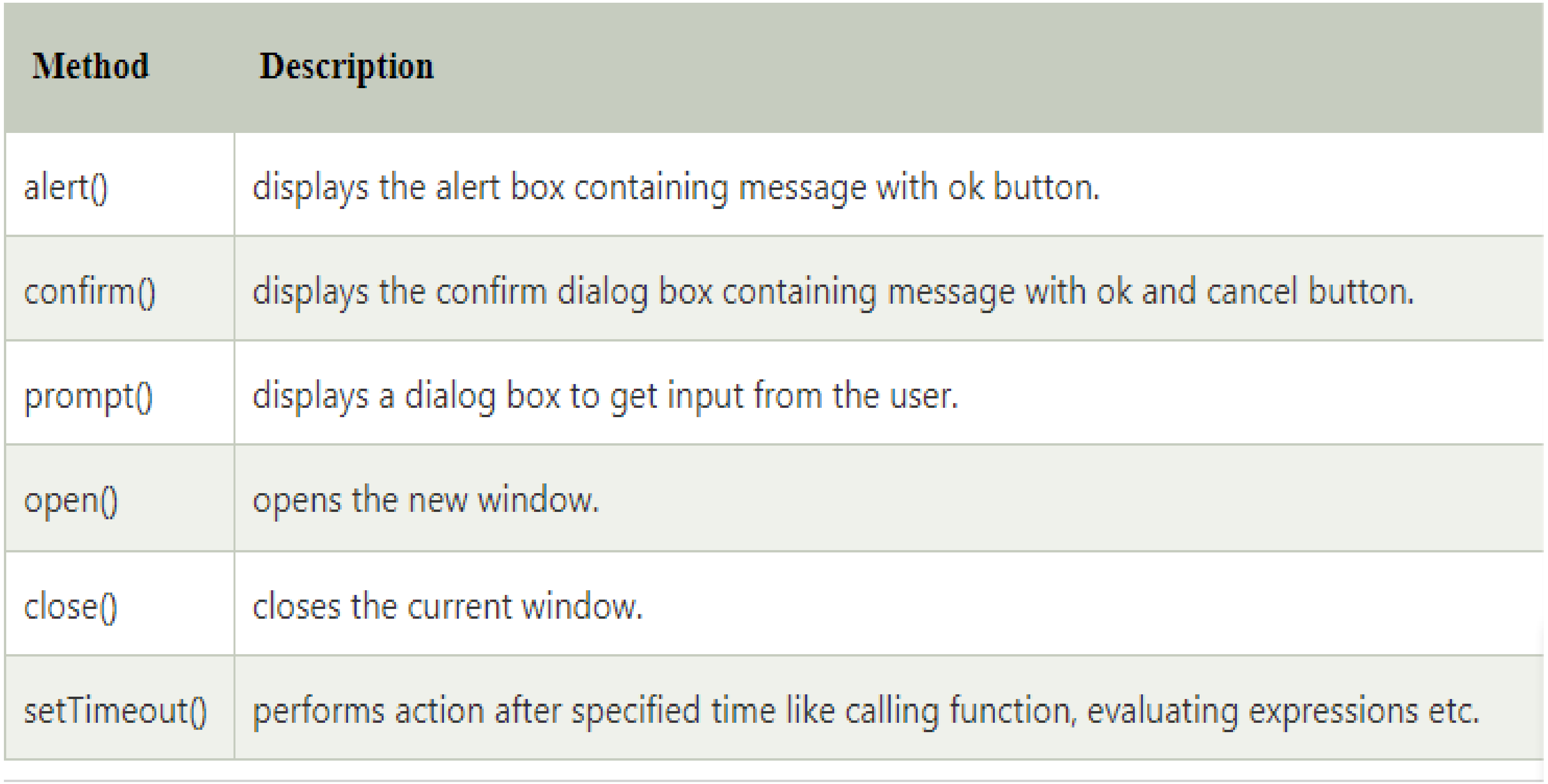
- The default object of browser is window, means you can call all the functions of window directly. For example:

```
window.alert("hello world");
// is same as:
alert("hello world ");
```

- You can use a lot of properties (other objects) defined underneath

Browser Object Model

- The important methods of window object are as follows:



Method	Description
alert()	displays the alert box containing message with ok button.
confirm()	displays the confirm dialog box containing message with ok and cancel button.
prompt()	displays a dialog box to get input from the user.
open()	opens the new window.
close()	closes the current window.
setTimeout()	performs action after specified time like calling function, evaluating expressions etc.

Window History

- The Window History Object
  - The history object contains the URLs visited by the user (in the browser window).
  - The history object is a property of the window object.
  - The history object is accessed with: window.history or just history:
- Example

```
let len = window.history.length;
//or
let len = history.length;
```

P.N. You can get the number of previous sites visited (history.length) but not the urls. As it may be a security risk.

Window History

- There is only 1 property of history object.
- There are only 3 methods of history object.

Property/Method	Description
back()	Loads the previous URL (page) in the history list
forward()	Loads the next URL (page) in the history list
go()	Loads a specific URL (page) from the history list
length	Returns the number of URLs (pages) in the history list

Navigator Object

- The window navigator object is used for browser detection.
- It can be used to get browser information such as appName, appCodeName, userAgent etc.
- The navigator object is the window property, so it can be accessed by:

```
window.navigator
Or,
navigator
```

Navigator Object Properties

- Properties of navigator object that returns information of the browser.

Property	Description
appCodeName	Returns browser code name
appName	Returns browser name
appVersion	Returns browser version
cookieEnabled	Returns true if browser cookies are enabled
geolocation	Returns a geolocation object for the user's location
language	Returns browser language
onLine	Returns true if the browser is online
platform	Returns browser platform
product	Returns browser engine name
userAgent	Returns browser user-agent header

Screen Object

- The window screen object holds information of browser screen.
- It can be used to display screen width, height, colorDepth, pixelDepth etc.
- The navigator object is the window property, so it can be accessed by:

  window.screen

  Or,

  screen

Screen Object Properties

- There are many properties of screen object that returns information of the browser.

Property	Description
availHeight	Returns the height of the screen (excluding the Windows Taskbar)
availWidth	Returns the width of the screen (excluding the Windows Taskbar)
colorDepth	Returns the bit depth of the color palette for displaying images
height	Returns the total height of the screen
pixelDepth	Returns the color resolution (in bits per pixel) of the screen
width	Returns the total width of the screen

Location Object

- The location object contains information about the current URL.
- The location object is a property of the window object.
- The location object is accessed with:

  window.location

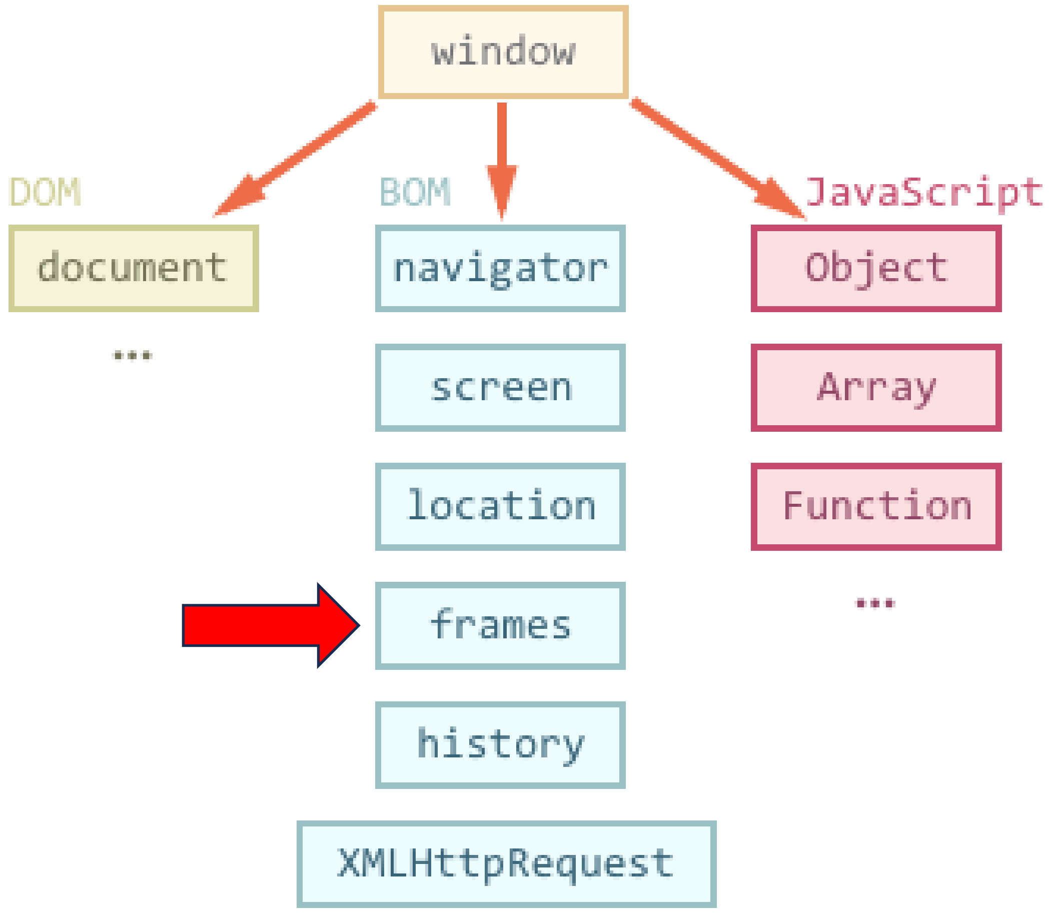
  or

  location

Location Object Properties

Property	Description
hash	Sets or returns the anchor part (#) of a URL
host	Sets or returns the hostname and port number of a URL
hostname	Sets or returns the hostname of a URL
href	Sets or returns the entire URL
origin	Returns the protocol, hostname and port number of a URL
pathname	Sets or returns the path name of a URL
port	Sets or returns the port number of a URL
protocol	Sets or returns the protocol of a URL
search	Sets or returns the querystring part of a URL

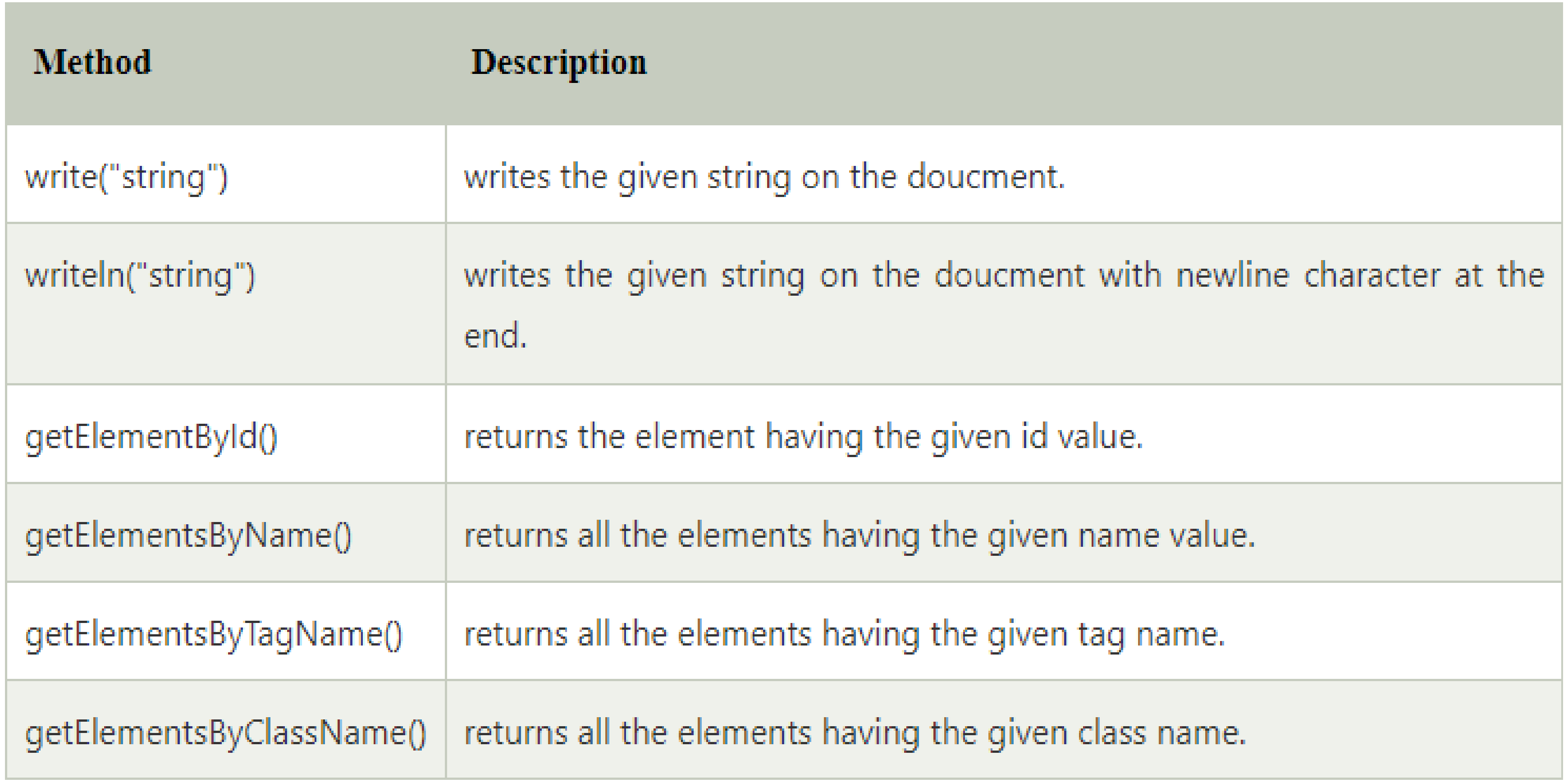
window frames property



- P.N. frames is a property, whereas location, navigator, screen, history, document are objects in window.
- The frames property returns an array with all window objects in the window.
- The frames property is read-only.
- The windows can be accessed by index numbers. The first index is 0.

Document Object

- Properties of document object



Method	Description
write("string")	writes the given string on the doucment.
writeln("string")	writes the given string on the doucment with newline character at the end.
getElementById()	returns the element having the given id value.
getElementsByName()	returns all the elements having the given name value.
getElementsByTagName()	returns all the elements having the given tag name.
getElementsByClassName()	returns all the elements having the given class name.

Event Handling

Event Handling using JavaScript

- Event handling in JavaScript refers to the process of capturing and responding to events that occur in a web page or application.
- Events can be user interactions, such as clicking a button, moving the mouse, typing on the keyboard, or system-generated events like the page finishing loading.
- JavaScript provides a way to listen for these events and execute code (event handlers) in response to them.
- Event handling is a fundamental aspect of creating interactive and dynamic web pages.

Event Handling Key concepts

- **Event**: An event is a specific occurrence or action that takes place in a web page. Examples include clicking a button, hovering over an element, submitting a form, or resizing the window.
- **Event Listener**: An event listener is a function that "listens" for a specific event on a particular HTML element. It waits for the event to occur and then executes a callback function when the event happens.
- **Callback Function**: A callback function is a JavaScript function that you define and pass to an event listener. It gets executed when the associated event occurs.
- **Event Object**: When an event occurs, an event object is created that contains information about the event, such as the type of event, the target element, and any additional data related to the event.

Example of Event handling in JavaScript

```
// Get a reference to a button element with the id "myButton"
const button = document.getElementById("myButton");

// Add an event listener to the button element
button.addEventListener("click", function(event) {
  // This code will execute when the button is clicked
  alert("Button clicked!");
});
```

Example of Event handling in JavaScript

- This examples
  - Gets a reference to an HTML element with the id "myButton" using document.getElementById.
  - Attaches an event listener to the button using the addEventListener method.
  - Specifies that we want to listen for the "click" event.
  - Provides a callback function that displays an alert when the button is clicked.
- Event handling in JavaScript is crucial for building interactive web applications, as it allows you to respond to user actions and create dynamic, responsive user interfaces.

Event Propagation

- Event propagation in JavaScript refers to the order in which events are processed and propagated (or "bubbled" and "captured") through the DOM (Document Object Model) hierarchy.
- Understanding event propagation is important for handling events effectively.
- There are two phases of event propagation in the DOM:
  - **Event Capturing (Capture Phase):** In this phase, the event travels from the outermost ancestor element (the root) to the target element. It goes through each ancestor in the hierarchy until it reaches the target element. This phase allows to capture events at a higher-level container before they reach the actual target element.
  - **Event Bubbling (Bubble Phase):** After the event reaches the target element, it starts to bubble back up through the ancestor elements in the reverse order. It goes from the target element to the root of the DOM. This phase allows to catch events as they bubble up through the hierarchy.

By default, most event handlers in JavaScript use the event bubbling phase.

However, you can also specify the capture phase by setting the useCapture argument to true when using the addEventListener method. Here's an example:

```
const parent = document.getElementById("parent");
const child = document.getElementById("child");
parent.addEventListener("click", function() {
  console.log("Parent Clicked (Bubbling Phase)");
});
child.addEventListener("click", function() {
  console.log("Child Clicked (Bubbling Phase)");
}, false); // The third argument 'false' indicates the use of the bubbling phase
// If you click the child element, you will see the following output:
// "Child Clicked (Bubbling Phase)"
// "Parent Clicked (Bubbling Phase)"
```

In the earlier example, we have a parent element (<div id="parent">) and a child element (<div id="child">), nested inside the parent. We attach click event listeners to both elements.
The event on the child element bubbles up to the parent element because we set useCapture to false in the addEventListener method for the child element.

If you set useCapture to true for the child element, the event would be captured during the capture phase and logged in the opposite order:

```
child.addEventListener("click", function() {
  console.log("Child Clicked (Capture Phase)");
}, true);

// If you click the child element with the above code, you will see the following output:
// "Child Clicked (Capture Phase)"
// "Parent Clicked (Bubbling Phase)"
```

preventDefault()

- The preventDefault() method is used to prevent the default behavior of an event.
- Many HTML elements have default behaviors associated with certain events. For example, clicking on a link (<a> element) by default navigates to a new page, and submitting a form (<form> element) by default sends data to the server and reloads the page.
- By calling preventDefault () on an event object within an event handler, you can stop the default behavior from occurring.
- It's commonly used with events like click, submit, and keydown, among others.

preventDefault()

```
const link = document.getElementById("myLink");

link.addEventListener("click", function(event) {
  // Prevent the default behavior (navigating to a new page)
  event.preventDefault();

  // You can now perform custom actions here.
  console.log("Link clicked, but default behavior prevented.");
});
```

In this example, clicking the link won't navigate to a new page because preventDefault is called within the event handler.

stopPropagation()

- The stopPropagation() method is used to stop the propagation of an event through the DOM hierarchy.
- As an event propagates from an element to its ancestors (capture phase) and then back down to the target element's descendants (bubbling phase), it can trigger multiple event handlers.
- Calling stopPropagation() prevents the event from propagating further, which means that event handlers on ancestor or descendant elements won't be executed.
- This is useful when you want to control which event handlers should run, especially in cases with nested elements and event listeners.

stopPropagation()

```
const parent = document.getElementById("parent");
const child = document.getElementById("child");
parent.addEventListener("click", function() {
  console.log("Parent clicked.");
});
child.addEventListener("click", function(event) {
  // Prevent the click event from propagating to the parent.
  event.stopPropagation();
  console.log("Child clicked, propagation stopped.");
});
```

In this example, clicking the child element will trigger only the child's click event handler. The parent's event handler won't be executed because stopPropagation() prevents the event from propagating up to the parent.

A word of caution

- Both preventDefault() and stopPropagation() are powerful tools for controlling and customizing event behavior in JavaScript, especially in scenarios where you need fine-grained control over event handling.
- However, they should be used with caution, as overusing them can lead to unexpected user experiences and make the application behavior less intuitive.

JavaScript Form Validation

JavaScript Form Validation

- Form validation in JavaScript is the process of ensuring that user input in web forms meets specific criteria before it's submitted to a server.
- Validating user input is crucial for data accuracy, security, and providing a good user experience.
- JavaScript provides facility to validate the form on the client-side so data processing will be faster than server-side validation. Most of the web developers prefer JavaScript form validation.

JavaScript Form Validation

- Through JavaScript, we can validate name, password, email, date, mobile numbers etc..
- Basic Validation − First of all, the form must be checked to make sure all the mandatory fields are filled in. It would require just a loop through each field in the form and check for data.
- Data Format Validation − Secondly, the data that is entered must be checked for correct form and value. Your code must include appropriate logic to test correctness of data.

```
<!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>
    <form id="myForm">
      <label for="name">Name:</label>
      <input type="text" id="name" name="name" required />

      <label for="email">Email:</label>
      <input type="email" id="email" name="email" required />

      <input type="submit" value="Submit" />
    </form>
```

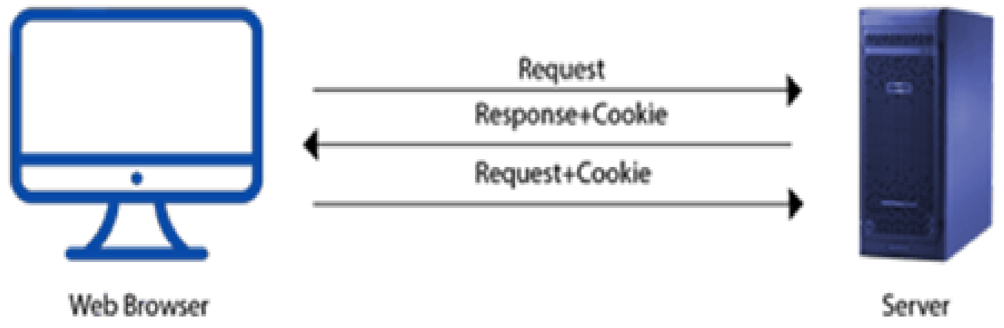
```
<script type="text/javascript">
  const myForm = document.getElementById("myForm");
  myForm.addEventListener("submit", validateForm);

  function validateForm(event) {
    console.log("validateForm function called");
    event.preventDefault(); // Prevent the form from submitting

    const name = document.getElementById("name").value;
    const email = document.getElementById("email").value;

    // Simple validation checks
    if (name === "") {
      alert("Name is required.");
      return false;
    }
    if (email === "") {
      alert("Email is required.");
      return false;
    }
```

```
        if (!isValidEmail(email)) {
          alert("Please enter a valid email address.");
          return false;
        }
        // If all validation checks pass, you can submit the form to the server
        alert("Form submitted successfully!!!");
      }

      // Function to validate email format
      function isValidEmail(email) {
        const emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
        return emailPattern.test(email);
      }
    </script>
  </body>
</html>
```

Cookies

Cookies in JavaScript

- Cookies in JavaScript are small pieces of data that can be stored on a user's device, typically in a web browser.
- They are often used to store information temporarily, such as user preferences, session data, or tracking information.
- Cookies are sent with every HTTP request to the same domain that set them, allowing you to persist data between different pages or sessions.
- A cookie contains the information as a string generally in the form of a name-value pair separated by semi-colons. It maintains the state of a user and remembers the user's information among all the web pages



Working with Cookies : Setting Cookies

- Cookies are established using the document.cookie property. For instance, a developer can set a cookie like this:

document.cookie = "username=John; expires=Thu, 18 Dec 2023 12:00:00 UTC; path=/";

- In this example, a cookie named "username" with the value "John" is set. It has an expiration date of December 18, 2023, and is accessible from all paths on the current domain.

Working with Cookies : Getting Cookies

- Retrieving cookies involves reading the document.cookie property:

```
var cookies = document.cookie;
```

- The variable cookies will contain a string with all the cookies associated with the current domain.

Working with Cookies : Parsing Cookies

As document.cookie returns a string, developers often need to parse it to access specific cookies. This can be done by creating a JavaScript function:

```
function getCookie(name) {
   var value = "; " + document.cookie;
   var parts = value.split("; " + name + "=");
   if (parts.length === 2) {
      return parts.pop().split(";").shift();
   }
   return null;
}
var username = getCookie("username");
```

In this code, the getCookie("username") function retrieves the value of the "username" cookie.

Working with Cookies : Modifying & Deleting Cookies

- Cookies can be updated by setting them again with the same name. This effectively replaces the previous value.

```
document.cookie = "username=Jane; expires=Thu, 18 Dec 2023
12:00:00 UTC; path=/";
```

- To delete a cookie, one can set its expiration date to a past date:

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

Working with Cookies : Deleting Cookies

- It's important to note that cookies have limitations and potential security concerns, such as limited storage capacity.
- For more robust client-side data storage, developers should consider alternatives like Web Storage (localStorage or sessionStorage) or IndexedDB, depending on their specific requirements and browser support.
- Furthermore, maintaining user privacy and data security should be a primary consideration when working with cookies, particularly for sensitive information.

Promise

Promise in JavaScript

- In JavaScript, a Promise is an object that represents the eventual completion or failure of an asynchronous operation.
- Promises are used to handle asynchronous operations such as data fetching, file reading, or network requests in a more structured and manageable way.
- They provide a way to write asynchronous code that is more readable and easier to reason about.
- Promises have three states:
  - **Pending**: Initial state, neither fulfilled nor rejected.
  - **Fulfilled**: The operation completed successfully, and a result is available.
  - **Rejected**: The operation failed, and an error reason is available.

.then()

- This method is used to register callbacks for when the Promise is fulfilled or rejected.
- It takes two optional arguments, one for the success case and another for the error case.

```
promiseObject.then(
  function (result) {
    // Handle the successful result
  },
  function (error) {
    // Handle the error
  }
);
```

.catch()

- This method is used to handle errors in a more concise way.
- It is equivalent to calling then(null, errorCallback)

```
promiseObject.catch(function (error) {
  // Handle the error
});
```

Example

```
// Simulated function to fetch data asynchronously
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const randomNum = Math.random();
      if (randomNum < 0.7) {
        // Simulating a successful response
        resolve(`Data is successfully fetched: ${randomNum}`);
      } else {
        // Simulating an error response
        reject('Error: Data could not be fetched');
      }
    }, 1000); // Simulating a 1-second delay
  });
}

// Usage of the fetchData function with Promises
fetchData()
  .then((result) => {
    console.log(result); // Handling the successful result
  })
  .catch((error) => {
    console.error(error); // Handling errors
  });
```

Example

- fetchData() returns a Promise. Within the Promise, there is a simulated asynchronous operation using setTimeout.
- The **resolve function** is used to signify a successful response and to provide the **result as an argument to resolve**. Conversely, the **reject function** indicates an error response, with the **error message passed as an argument to reject**.
- The **fetchData()** is called, and the then method is used to manage the successful result when the Promise is resolved. If the random number generated is less than 0.7, the operation is successful. Otherwise, it results in rejection with an error.
- The catch method is employed to manage errors when the Promise is rejected. In cases where the random number is greater than or equal to 0.7, it triggers an error.
- This code demonstrates how Promises can be utilized to handle asynchronous operations in a structured manner, making it straightforward to handle both successful outcomes and errors.

Promise : Summary

- Promises provide a structured way to work with asynchronous code and avoid callback hell (also known as the "pyramid of doom") where multiple nested callbacks can make code hard to read and maintain.
- They make it easier to handle errors and provide a clear separation between the success and error cases of an asynchronous operation.
- Promises have been a fundamental part of JavaScript for handling asynchronous operations, and they serve as the foundation for more advanced async/await syntax introduced in later versions of JavaScript.

Client Server Communication

Client Server Communication

- Client-server communication in JavaScript typically involves making network requests from a client-side application (running in a web browser) to a server and handling the responses.
- This interaction allows web applications to fetch data, send data, and update server-side resources.

Client Server Communication : Concepts

- HTTP Requests:
  - HTTP (Hypertext Transfer Protocol) is the foundation for communication between clients (web browsers) and servers. JavaScript can send HTTP requests to servers using various methods, such as GET, POST, PUT, DELETE, etc.
  - The most common way to make HTTP requests in modern JavaScript is by using the fetch() API or XMLHttpRequest.
- Fetch API:
  - The Fetch API provides a more modern and flexible way to make HTTP requests in JavaScript.
  - It returns Promises, making it easier to work with asynchronous code.

Client Server Communication : Concepts

- XMLHttpRequest (XHR):
- This is an older technique for making HTTP requests in JavaScript.
- It provides more control over request headers and parameters. However, it is less user-friendly compared to the Fetch API.

```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'https://api.example.com/data', true);
xhr.onreadystatechange = function() {
  if (xhr.readyState === 4 && xhr.status === 200) {
    var data = JSON.parse(xhr.responseText);
    console.log(data);
  }
};
xhr.send();
```

Client Server Communication : Concepts

- WebSockets:
  - WebSockets provide full-duplex communication channels over a single TCP connection. They enable real-time, bidirectional communication between the client and server.
  - Libraries like Socket.io simplify WebSocket usage in JavaScript.

```
const socket = new
WebSocket('wss://api.example.com/socket');
socket.onmessage = event => {
  console.log('Received message:', event.data);
};
```

Client Server Communication : Concepts

- AJAX (Asynchronous JavaScript and XML):
  - AJAX is a technique that uses XMLHttpRequest or the Fetch API to update parts of a web page without requiring a full page reload.
  - It's often used for dynamic content loading.
- JSON (JavaScript Object Notation):
  - JSON is a common data format used for sending structured data between the client and server.
  - It's easy to work with in JavaScript and is often used in API responses.

Client Server Communication : Concepts

- CORS (Cross-Origin Resource Sharing):
  - When making requests to a different domain (cross-origin requests), CORS restrictions may apply.
  - The server should be configured to allow such requests or use server-side solutions like JSONP or proxy servers.
- Authentication and Security:
  - Protect client-server communication with appropriate authentication methods (e.g., tokens, OAuth) and encryption (e.g., HTTPS).
  - Security measures are crucial to prevent unauthorized access and data breaches.

fetch()

fetch() function

- The fetch() function in JavaScript is a modern API used for making network requests to fetch resources, typically from a web server.
- It provides a more flexible and powerful way to perform HTTP requests compared to traditional approaches using XMLHttpRequest.

Basic Usage

- The fetch() function returns a Promise that resolves to the Response to the request.
- Methods like .json(), .text(), or .blob() on the Response can be used to parse and process the data.

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

HTTP Methods

- By default, fetch() uses the GET method.
- Other HTTP methods may be specified like POST, PUT, DELETE, etc. using the method option.

```
fetch('https://api.example.com/data', {
  method: 'POST',
  body: JSON.stringify({ key: 'value' })
})
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

Customizing Headers

- Headers like Content-Type, Authorization, etc. may be customized using the headers option.

```
fetch('https://api.example.com/data', {
  method: 'GET',
  headers: {
    'Authorization': 'Bearer token'
  }
})
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

fetch() : Handling Response and Error

- Response Handling:
  - The Response object returned by fetch() contains information like status code, headers, and methods to process the response data.
- Error Handling:
  - The catch() method is used to handle network errors or rejected Promises.
  - It's important to include error handling to gracefully manage issues that might arise during the fetch operation.

Fetching from other servers

- CORS and Same-Origin Policy:
  - Just like XMLHttpRequest, fetch requests are subject to the same-origin policy and Cross-Origin Resource Sharing (CORS) restrictions.
  - The server may be configured to allow such requests, or use server-side solutions like JSONP or proxy servers.

Asynchronous Operation

- fetch() returns a Promise, making it non-blocking and well-suited for modern asynchronous programming patterns.
- async/await may be used with fetch() to write more concise and readable asynchronous code.

Asynchronous Operation

```
async function fetchData() {
  try {
    const response = await fetch('https://api.example.com/data');
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('Fetch error:', error);
  }
}
```

fetch() function

- The fetch() function is widely used in modern web development to fetch data from APIs, interact with server-side resources, and update remote databases.
- It is versatile, supports various request and response types, and plays a significant role in building dynamic web applications that communicate with servers.

Manipulating CSS using JavaScript

Manipulating CSS using JavaScript

- JavaScript can be used to
  - manipulate an element's inline styles
  - add/remove CSS classes

Example 1 : Manipulating Inline Styles

In this example, when the button is clicked, the changeColor function is called, which sets the inline color style of the myPara element to 'blue’.

```
<p id="myPara">This is a paragraph.</p>
<button id="changeColorButton">Change Color</button>
```

Example 1 : Manipulating Inline Styles

```
// Get references to the paragraph and button elements
const myPara = document.getElementById('myPara');
const changeColorButton = document.getElementById('changeColorButton');
// Function to change the color
function changeColor() {
  myPara.style.color = 'blue'; // Change the text color to blue
}
                    ------------------------------------------------------------
// Add a click event listener to the button
changeColorButton.addEventListener('click', changeColor);
```

Example 2 : Adding/Removing CSS Classes

- **HTML**

```
<p id="myPara" class="defaultStyle">This is a paragraph.</p>
<button id="toggleStyleButton">Toggle Style</button>
```

- **CSS**

```
/* CSS style classes */
.defaultStyle {
  color: black;
  font-size: 16px;
}
```

```
.newStyle {
  color: red;
  font-size: 20px;
}
```

Example 2 : Adding/Removing CSS Classes

- **JavaScript**

```
// Get references to the paragraph and button elements
const myPara = document.getElementById('myPara');
const toggleStyleButton = document.getElementById('toggleStyleButton’);

// Function to toggle the style class
function toggleStyle() {
  myPara.classList.toggle('newStyle'); // Toggle the "newStyle" class
}

// Add a click event listener to the button
toggleStyleButton.addEventListener('click', toggleStyle);
```

Example 2 : Adding/Removing CSS Classes

- In this example, when the button is clicked, the toggleStyle function toggles the newStyle class on the myPara element using classList.toggle().
- The CSS rules for the newStyle class define the new styles for the paragraph.

End