Java Script

What is a Javascript?

- ✓ JavaScript is used to create client-side dynamic pages.
- ✓ JavaScript is *an* *object-based scripting language* which is lightweight and cross-platform.
- ✓ JavaScript is not a compiled language, but it is a translated language. The JavaScript Translator (embedded in the browser) is responsible for translating the JavaScript code for the web browser.
- ✓ Designed for creating network-centric applications.

Javascript syntax

```
<script>
………..
</script>
```

Example:

```
<script type = “text/javascript”>
Document.write(“Welcome to java script”);
</script>
```

3 Places to put JavaScript code

- ✓ Between the body tag of html
- ✓ Between the head tag of html
- ✓ In .js file (external javaScript)

Example of External JS

Welcome.js

```
alert(“Hello world!”);
```

Index.html

```
<script type=“text/javascript” src=“welcome.js”>
</script>
```

Javascript comment

✓There are two types of comments in javascript.

1. Single-line comment

✓Single line comment represent by

double forword (//)

//single line comment

2. Multi-line comment

✓ It can be used to add single as well as multiline comments.

/* this is a multiline comment*/

What is ECMA script?

- ✓ ECMA stands for European Computer Manufacturers Association
- ✓ ECMAScript is the standard that JavaScript programming language uses.
- ✓ ECMAScript provides the specification on how JavaScript programming language should work.
- ✓ JavaScript **ES6** (also known as **ECMAScript 2015** or **ECMAScript 6**) is the newer version of JavaScript that was introduced in 2015.

Javascript variable

- ✓ Variables are containers for storing data (storing data values).

Rules for designing a variable

1. Name must start with a letter (a to z or A to Z), underscore(_), or dollar($) sign.
2. After first letter we can use digits (0 to 9), for example value1.
3. JavaScript variables are case sensitive, for example x and X are different variables.
4. Reserved words (like JavaScript keywords) cannot be used as variables.

Declare a variable

There are 4 Ways to Declare a JavaScript Variable:

1. Using var
2. Using let
3. Using const
4. Using nothing

Declare a variable

Using var

- ✓ Variables are containers for storing data (storing data values).
- ✓ In this example, x, y, and z, are variables, declared with the var keyword

Example:

```
var x = 5
var y = 6
var z = x + y;
```

Declare a variable

Using var

Can be redeclared.

Example:

With var you can:

```
var x = "John Doe";
var x = 0;
```

Declare a variable

Using let

Variables defined with let can not be redeclared.

Variables defined with let must be declared before use.

Variables defined with let have block scope.

In this example, x, y, and z, are variables, declared with the let keyword:

Example:

```
let x = 5
let y = 6
let z = x + y;
```

Declare a variable

Using let

Cannot be Redeclared

You can not accidentally redeclare a variable declared with let.

Example:

With let you can not do this:

```
let x = "John Doe";
let x = 0;
```

Declare a variable

Using cont

- ✓ Variables defined with const cannot be Redeclared.
- ✓ Variables defined with const cannot be Reassigned.
- ✓ Variables defined with const have Block Scope.
- ✓ In this example, price1, price2, and total, are variables.

Example:

```
const price1 = 5;
const price2 = 6;
let total = price1 + price2;
```

Declare a variable

Using cont

- ✓ A const variable cannot be reassigned.
- ✓ JavaScript const variables must be assigned a value when they are declared:

Example:

```
const PI = 3.141592653589793;  // Correct
 PI = 3.14;      // This will give an error
 PI = PI + 10;   // This will also give an error
```

Javascript Data Types

- ✓ JavaScript provides different **data types** to hold different types of values.
- ✓ There are two types of data types in JavaScript.

1. Primitive data type
2. Non-primitive (reference) data type

Primitive Data types

Data Type	Description
String	represents sequence of characters e.g. "hello"
Number	represents numeric values e.g. 100
Boolean	represents boolean value either false or true
Undefined	represents undefined value
Null	represents null i.e. no value at all
Bigint	represent integer values ($\pm 2^{53}-1$)
Symbol	The Symbol data type is used to create unique identifiers for object properties.

non-primitive data types

Data Type	Description
Object	represents instance through which we can access members
Array	represents group of similar values
RegExp	represents regular expression
Function	The Function data type is used to define a set of statements that perform a task.
Date	The Date data type is used to store dates and times.

Primitive Data Types

Number: The Number data type is used to store numeric values.

Example:

let x = 10; // Here, x is a variable of Number data type

String: The String data type is used to store textual data.

Example:

let str = "Hello World!"; // Here, str is a variable of String data type

Primitive Data Types

Boolean: The Boolean data type is used to store true/false values.

Example:

let bool = true; // Here, bool is a variable of Boolean data type

Null: The Null data type is used to represent the intentional absence of any object value.

Example:

let x = null; // Here, x is a variable of Null data type

Primitive Data Types

Undefined: The Undefined data type is used to represent the absence of any defined value.

Example:

let x; // Here, x is a variable of Undefined data type

Symbol: The Symbol data type is used to create unique identifiers for object properties.

Example:

let sym = Symbol(); // Here, sym is a variable of Symbol data type

Non-Primitive Data Types

Object: The Object data type is used to store collections of data, including arrays and functions.

Example:

let obj = {name: "John", age: 30}; // Here, obj is a variable of Object data type

Array: The Array data type is used to store collections of data.

Example:

let arr = [1, 2, 3, 4]; // Here, arr is a variable of Array data type

Non-Primitive Data Types

Function: The Function data type is used to define a set of statements that perform a task.

Example:

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

Non-Primitive Data Types

Date: The Date data type is used to store dates and times.

Example:

```
let date = new Date(); // Here, date is a variable of
                                      Date data type
```

RegExp: The RegExp data type is used to represent regular expressions.

Example:

```
let pattern = /Hello/gi; // Here, pattern is a
                 variable of RegExp data type
```

JavaScript Operators

- ✓ JavaScript operators are symbols that are used to perform operations on operands.

Types of operators

1. Arithmetic Operators
2. Comparison (Relational) Operators
3. Bitwise Operators
4. Logical Operators
5. Assignment Operators
6. Special Operators

Arithmetic Operators

Operator	Description	Example
+	Addition	3 + 5 returns 8
-	Subtraction	7 - 2 returns 5
*	Multiplication	4 * 6 returns 24
/	Division	12 / 4 returns 3
%	Modulo (remainder)	15 % 4 returns 3
++	Increment (adds 1 to a variable)	let x = 5; x++; returns 6
--	Decrement (subtracts 1 from a variable)	let y = 7; y--; returns 6

Note that the addition operator (+) can also be used for string concatenation, meaning it will join two strings together.
For example, "Hello " + "world" returns "Hello world". However, it's important to note that the other arithmetic operators only work with numbers, not strings.,

Assignment Operators

Operator	Description	Example
=	Assigns a value to a variable	let x = 5; assigns the value 5 to the variable x
+=	Adds a value to a variable	let x = 3; x += 5; assigns the value 8 to the variable x
-=	Subtracts a value from a variable	let y = 7; y -= 2; assigns the value 5 to the variable y
*=	Multiplies a variable by a value	let z = 4; z *= 6; assigns the value 24 to the variable z
/=	Divides a variable by a value	let a = 12; a /= 4; assigns the value 3 to the variable a
%=	Assigns the remainder of a variable divided by a value	let b = 15; b %= 4; assigns the value 3 to the variable b

Note that these operators combine an arithmetic operation with the assignment operation. For example, x += 5; is equivalent to x = x + 5;. The same applies to the other operators in the table.

Comparison Operators

Operator	Description	Example
==	Checks if two values are equal, but does not check data type	3 == 3 returns true
===	Checks if two values are equal, and checks data type	3 === '3' returns false
!=	Checks if two values are not equal, but does not check data type	4 != '4' returns false
!==	Checks if two values are not equal, and checks data type	4 !== '4' returns true
>	Checks if one value is greater than another	7 > 3 returns true
<	Checks if one value is less than another	2 < 5 returns true
>=	Checks if one value is greater than or equal to another	6 >= 6 returns true

Note that the == and != operators can be problematic, as they perform type coercion, which means they try to convert values to the same data type before comparing them. This can sometimes lead to unexpected results. It's generally recommended to use the === and !== operators instead, as they are more strict and do not perform type coercion.

Logical Operators

Operator	Description	Example
&&	Logical AND. Returns true if both operands are true	3 < 5 && 7 > 2 returns true
||	Logical OR. Returns true if at least one operand is true	3 > 5 || 7 > 2 returns true
!	Logical NOT. Returns the opposite boolean value of the operand	!(3 < 5) returns false

Note that the logical AND (&&) and OR (||) operators use short-circuit evaluation. This means that if the left operand of the && operator is false, the right operand will not be evaluated, because the overall expression will always be false. Similarly, if the left operand of the || operator is true, the right operand will not be evaluated, because the overall expression will always be true.

Logical Operators

Here are some examples of short-circuit evaluation:

```
let x = 5;
let y = null;
// The left operand of && is true, so the right operand is evaluated
if (x > 0 && y.length > 0) {
        console.log("This will not be executed");
}
// The left operand of || is false, so the right operand is evaluated
if (x < 0 || y.length > 0) {
        console.log("This will throw an error because y is null");
}
```

The logical NOT (!) operator simply returns the opposite boolean value of the operand. For example, !(3 < 5) returns false, because 3 < 5 is true, and !true is false.

Logical Operators

Operator	Description	Example
&	Bitwise AND. Compares the binary representation of two operands and returns a new binary number where each bit is set to 1 only if both corresponding bits in the operands are 1	5 & 3 returns 1
|	Bitwise OR. Compares the binary representation of two operands and returns a new binary number where each bit is set to 1 if at least one corresponding bit in the operands is 1	5 | 3 returns 7
~	Bitwise NOT. Returns the one's complement of the binary representation of the operand	~5 returns -6
^	Bitwise XOR. Compares the binary representation of two operands and returns a new binary number where each bit is set to 1 only if exactly one corresponding bit in the operands is 1	5 ^ 3 returns 6
<<	Bitwise left shift. Shifts the bits of the left operand to the left by the number of positions specified by the right operand	5 << 1 returns 10
>>	Bitwise right shift. Shifts the bits of the left operand to the right by the number of positions specified by the right operand	5 >> 1 returns 2
>>>	Bitwise zero-fill right shift. Shifts the bits of the left operand to the right by the number of positions specified by the right operand, and fills the leftmost positions with zeros	5 >>> 1 returns 2

Logical Operators

Here are some examples of bitwise operations:

```
let x = 5;
let y = 3;
console.log(x & y); // returns 1  Bitwise AND
console.log(x | y); // returns  7 Bitwise OR
console.log(~x); // returns  -6 Bitwise NOT
console.log(x ^ y); // returns  6 Bitwise XOR
console.log(x << 1); // returns  10 Bitwise left shift
console.log(x >> 1); // returns  2 Bitwise right shift
console.log(x >>> 1); // returns  2 Bitwise zero-fill right shift
```

Special Operators

Operator	Example	Explanation
typeof	typeof "hello"	Returns the data type of a value. In this example, the result is the string "string".
Instanceof	obj instanceof Array	Tests whether an object is an instance of a particular constructor function. In this example, the result is true if obj is an instance of the Array constructor.
Delete	delete obj.prop	Deletes a property from an object. In this example, the property prop is deleted from the object obj.
Void	void 0	Evaluates an expression and always returns undefined. In this example, the expression 0 is evaluated and the result is undefined.
In	"prop" in obj	Checks if a property exists in an object. In this example, the result is true if the property "prop" exists in the object obj.

Special Operators

Operator	Example	Explanation
New	new Object()	Creates a new object from a constructor function. In this example, a new object is created from the Object constructor.
This	this.prop	Refers to the current object. In this example, the prop property of the current object is accessed.
Super	super.method()	Refers to the parent class of an object. In this example, the method function of the parent class is called.
Yield	function* myGenerator() { yield 1; }	Used in generator functions to yield a value. In this example, the generator function yields the value 1.
Await	async function myAsyncFunction() { await somePromise(); }	Used in async functions to wait for a Promise to resolve. In this example, the myAsyncFunction function waits for somePromise to resolve.

Special operators

Operator	Description
(?:)	Conditional Operator returns value based on the condition. It is like if-else.
,	Comma Operator allows multiple expressions to be evaluated as single statement.
delete	Delete Operator deletes a property from the object.
in	In Operator checks if object has the given property
instanceof	checks if the object is an instance of given type
new	creates an instance (object)
typeof	checks the type of object.
void	it discards the expression's return value.
yield	checks what is returned in a generator by the generator's iterator.

JavaScript If-else

- ✓ The **JavaScript if-else statement** is used *to execute the code whether condition is true or false*.
- ✓ There are three forms of if statement in JavaScript.

1. If Statement
2. If else statement
3. if else if statement

if statement

- ✓ Use the if statement to specify a block of JavaScript code to be executed if a condition is true.

Syntax:

```
if(condition)
{
//block of the code
}
```

Example:

```
if (hour < 18) {
  greeting = "Good
day";
}
```

Output:

```
Good day
```

If else Statement

✓ Use the else statement to specify a block of code to be executed if the condition is false.

Syntax:

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

Example:

```
if (hour < 18) {
  greeting = "Good
day";
} else {
  greeting = "Good
evening";
}
```

Output:

```
Good day
```

if else if statement

- ✓ It evaluates the content only if expression is true from several expressions.

Syntax:

```
if(cond1){
//bolck of the cond1 code
} else if(cond2){
//block of the cond2 code
}else{
//else block of the code
}
```

Example:

```
if (hour < 10) {
  greeting = " Good
morning ";
} else if(hour<20){
  greeting = "Good day";
} else {
  greeting = "Good
evening";
}
```

Output:

```
Good day
```

Switch statement

- ✓ The switch statement is used to perform different actions based on different conditions.

Syntax:

```
switch(expression) {
    case x:     // code block
                   break;
    case y:    // code block
                   break;
    default:
               // code block
  }
```

Switch statement

Example

The getDay() method returns the weekday as a number between 0 and 6.
(Sunday=0, Monday=1, Tuesday=2 ..)

```
switch (new Date().getDay()) {
 case 0:
   day = "Sunday";
   break;
 case 1:
   day = "Monday";
   break;
 case 2:
    day = "Tuesday";
   break;
 case 3:
   day = "Wednesday";
   break;
case 4:
   day = "Thursday";
   break;
 case 5:
   day = "Friday";
   break;
 case 6:
   day = "Saturday";
}
```

Switch statement

break Keyword

- ✓ When JavaScript reaches a break keyword, it breaks out of the switch block.
- ✓ This will stop the execution inside the switch block.
- ✓ It is not necessary to break the last case in a switch block.

default Keyword

The default keyword specifies the code to run if there is no case match:

Example

```
switch (new Date().getDay()) {
  case 6:
    text = "Today is Saturday";
    break;
  case 0:
    text = "Today is Sunday";
    break;
  default:
    text = "Looking forward to the Weekend";
}
```

JavaScript Loops

- ✓ The JavaScript loops are used *to iterate the piece of code* using for, while, do while or for-in loops.

Types of loops in JavaScript.

1. for loop
2. while loop
3. do-while loop
4. for-in loop
5. for-of loop(ES6)

for loop

Syntax:

```
for (statement 1; statement 2; statement 3) {
    // code block to be executed
}
```

- ✓ Statement 1 is executed (one time) before the execution of the code block.
- ✓ Statement 2 defines the condition for executing the code block.
- ✓ Statement 3 is executed (every time) after the code block has been executed.

for loop

Example

```
<html>
<body>
<p id="demo"></p>
<script>
var i = 5;
for (var i = 0; i < 10; i++) {
}
document.getElementById("demo").innerHTML = i;
</script>
</body>
</html>
```

Note : In the example, using var, the variable declared in the loop redeclares the variable outside the loop.

for loop

Example

```
<html><body>
<p id="demo"></p>
<script>
let i = 5;
for (let i = 0; i < 10; i++) {
}
document.getElementById("demo").innerHTML = i;
</script>
</body></html>
```

Note : In this example, using let, the variable declared in the loop does not redeclare the variable outside the loop.

When let is used to declare the i variable in a loop, the i variable will only be visible within the loop.

For in Loop

- ✓ This loop is specifically designed to iterate through the properties of an object.

Basic Syntax :

```
for (variable in object) {
// code to be executed
}
```

- ✓ The variable is a variable that will be assigned the property name on each iteration. The object is the object being iterated through.

For in Loop

Example :

```
const person = { name: 'John', age: 30, city: 'New York' };
for (let prop in person) {
  console.log(prop + ': ' + person[prop]);
}
```

In this example, we have an object called person that contains three properties: name, age, and city. The for...in loop iterates through each property in the person object and logs the property name and its corresponding value to the console.

The output of this code will be:

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

For of Loop

- ✓ The for...of loop iterates over the values of an iterable object.

Basic Syntax

```
for (variable of iterable) {
    // code to be executed
}
```

- ✓ The variable is a variable that will be assigned the value on each iteration. The iterable is the iterable object being iterated through.

For of Loop

- ✓ The for...of loop iterates over the values of an iterable object.

Example:

```
const fruits = ['apple', 'banana', 'orange’];
for (let fruit of fruits) {
        console.log(fruit);
}
```

- ✓ In this example, we have an array called fruits that contains three strings. The for...of loop iterates through each value in the fruits array and logs it to the console.

Output

```
apple
banana
orange
```

While Loop

- ✓ The while loop loops through a block of code as long as a specified condition is true.

Syntax

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

Note:

- ✓ If you forget to increase the variable used in the condition, the loop will never end.

While Loop

- ✓ The while loop loops through a block of code as long as a specified condition is true.

Syntax

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

Example

```
while (i < 10) {
        text += "The number is " + i;
        i++;
}
```

Do While Loop

✓ This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax

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

Example:

```
        do {
                  text += "The number is " + i;
                  i++;
        }while (i < 10);
```

for In Loop

- ✓ The JavaScript for in statement loops through the properties of an Object:

Syntax:

```
for (key in object)
{
    // code block to be executed
}
```

- ✓ Each iteration returns a **key** (x)
- ✓ The key is used to access the **value** of the key

for-of loop

- ✓ The JavaScript for of statement loops through the values of an iterable object.
- ✓ It lets you loop over iterable data structures such as Arrays, Strings, Maps, NodeLists, and more:

Syntax

```
for (variable of iterable) {
    // code block to be executed
}
```

- ✓ variable - For every iteration the value of the next property is assigned to the variable. *Variable* can be declared with const, let, or var.
- ✓ iterable - An object that has iterable properties.

Break and Continue

- ✓ The break statement "jumps out" of a loop.
- ✓ The continue statement "jumps over" one iteration in the loop.

Java script Labels

- ✓ To label JavaScript statements you precede the statements with a label name and a colon(:)

```
label:
statements
```

Syntax:

break *labelname*;
continue *labelname*;

Break and Continue

Break Example:

```
for (let i = 0; i < 10; i++) {
  if (i === 3) { break; }
  text += "The number is " + i + "<br>";
}
```

Continue Example:

```
for (let i = 0; i < 10; i++) {
  if (i === 3) { continue; }
  text += "The number is " + i + "<br>";
}
```

Functions

- ✓ Functions are the main “building blocks” of the program. They allow the code to be called many times without repetition.
- ✓ A JavaScript function is executed when "something" invokes it (calls it).
- ✓ A JavaScript function is defined with the function keyword, followed by a name, followed by parentheses ().
- ✓ Function names can contain letters, digits, underscores, and dollar signs (same rules as variables).
- ✓ The parentheses may include parameter names separated by commas:
- ✓ (parameter1, parameter2, ...)
- ✓ The code to be executed, by the function, is placed inside curly brackets: {}
- ✓ Function parameters are listed inside the parentheses () in the function definition.
- ✓ Function arguments are the values received by the function when it is invoked.

Functions

Syntax :

```
function name(parameter1, parameter2, ...parameterN) { // body
}
```

Without parameters Example:

```
function showMessage() {
  alert( 'Hello everyone!' );
}
showMessage();
showMessage();
```

With parameters Example:

```
function showMessage(from, text) {                    // parameters: from, text
  alert(from + ': ' + text);
}
Let from = "yan";
showMessage('Ann', 'Hello!’);       // Ann: Hello! (*)
showMessage(from, 'Hello!’);
```

Functions

Example:

```
function checkAge(age) {
  if (age >= 18) {
    return true;
  } else {
    return confirm('Do you have permission from your parents?');
  }
}

let age = prompt('How old are you?', 18);

if ( checkAge(age) ) {
  alert( 'Access granted' );
} else {
  alert( 'Access denied' );
}
```

Functions

- ✓ Rest parameters is a feature in JavaScript that allows a function to accept an indefinite number of arguments as an array. The rest parameter syntax allows us to represent an indefinite number of arguments as an array.
- ✓ The syntax for rest parameters is the three-dot notation (...) followed by the name of the array that will hold the rest of the parameters.

Example:

```
function sumAll(...args) { // args is the name for the array
  let sum = 0;
  for (let arg of args) sum += arg;
  return sum;
}
alert( sumAll(1) ); // 1
alert( sumAll(1, 2) ); // 3
alert( sumAll(1, 2, 3) ); // 6
```

Functions

Example:

```
function showName(firstName, lastName, ...titles) {
  alert( firstName + ' ' + lastName ); // Julius Caesar

  // the rest go into titles array
  // i.e. titles = ["Consul", "Imperator"]
  alert( titles[0] ); // Consul
  alert( titles[1] ); // Imperator
  alert( titles.length ); // 2
}
showName("Julius", "Caesar", "Consul", "Imperator");
```

Functions

Invokes Function

Example

Convert Fahrenheit to Celsius:

```
function toCelsius(fahrenheit) {
  return (5/9) * (fahrenheit-32);}
document.getElementById("demo").innerHTML = toCelsius(77);
```

- ✓ The () Operator Invokes the Function
- ✓ Using the example above, toCelsius refers to the function object, and toCelsius() refers to the function result.
- ✓ Accessing a function without () will return the function object instead of the function result.

Example

```
function toCelsius(fahrenheit) {
  return (5/9) * (fahrenheit-32);
}
document.getElementById("demo").innerHTML = toCelsius;
```

Functions

Invoking a Function as a Method

- ✓ In JavaScript you can define functions as object methods.

- ✓ The following example creates an object (myObject), with two properties (firstName and lastName), and a method (fullName):

Example:

```
const myObject = {
  firstName:"John",
  lastName: "Doe",
  fullName: function () {
    return this.firstName + " " + this.lastName;
  }
}
myObject.fullName();        // Will return "John Doe"
```

Functions

Invoking a Function with a Function Constructor

- ✓ If a function invocation is preceded with the new keyword, it is a constructor invocation.
- ✓ It looks like you create a new function, but since JavaScript functions are objects you actually create a new object:

Example

```
// This is a function constructor:
function myFunction(arg1, arg2) {
  this.firstName = arg1;
  this.lastName  = arg2;
}
// This creates a new object
const myObj = new myFunction("John", "Doe");
// This will return "John"
myObj.firstName;
```

- ✓ A constructor invocation creates a new object. The new object inherits the properties and methods from its constructor.

Functions

Arrow Function

- ✓ There's another very simple and concise syntax for creating functions, that's often better than Function Expressions.
- ✓ It's called "arrow functions", because it looks like this:

Syntax :

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

Example:

```
let sum = (a, b) => a + b;
/* This arrow function is a shorter form of:
let sum = function(a, b) {
  return a + b;
};
*/
alert( sum(1, 2) ); // 3
```

Functions

- ✓ Nested function is a function that is defined inside another function.
- ✓ A nested function can access variables and parameters of the outer function, which makes it a useful way to encapsulate functionality and create closures. Here's an example of a nested function:

Example:

```
<p id="demo">0</p>
<script>
function outer() {
  const message = 'Hello, ';
  function inner(name) {
    document.getElementById("demo").innerHTML = (message +
name);
  }
  inner("john");
}
const greet = outer();
greet(); // Output: Hello, John
</script>
```

Built in Functions

alert(): The alert() function displays an alert box with a message and an OK button.

Example:

```
alert("Hello, world!");
```

prompt(): The prompt() function displays a dialog box that prompts the user to enter input. The input is returned as a string.

Example:

```
let name = prompt("What is your name?");
alert "Hello, " + name + "!");
```

parseInt(): The parseInt() function parses a string argument and returns an integer.

Example:

```
let num = parseInt("6.8");
document.getElementById("demo").innerHTML=num; // Output: 3.14
```

Built in Functions

parseFloat(): The parseFloat() function parses a string argument and returns a floating-point number.

Example:

```
let num = parseFloat("3.14");
alert(num); // Output: 3.14
```

Math.random(): The Math.random() function returns a random number between 0 (inclusive) and 1 (exclusive).

Example:

```
let randomNum = Math.random();
alert(randomNum);
```

Math.floor(): The Math.floor() function returns the largest integer less than or equal to a given number.

Example:

```
let num = 3.14;
let floorNum = Math.floor(num);
alert(floorNum); // Output: 3
```

Built in Functions

Math.ceil(): The Math.ceil() function returns the smallest integer greater than or equal to a given number.

Example:

```
let num = 3.14;
let ceilNum = Math.ceil(num);
alert(ceilNum); // Output: 4
```

Variable Scope in Functions

In JavaScript, scope refers to the visibility and accessibility of variables and functions within a program. There are two types of scope in JavaScript: global scope and local scope.

Global scope refers to variables and functions that are defined outside of a function, and are therefore accessible from anywhere in the program. Local scope, on the other hand, refers to variables and functions that are defined within a function, and are only accessible from within that function.

Variable Scope in Functions

The difference between global and local scope:

Example:

```
let globalVar = "I am a global variable";
function myFunction() {
  let localVar = "I am a local variable";
  alert(localVar);
  alert (globalVar);
}
myFunction();       // Output: "I am a local variable" and "I am a global
variable"
alert(localVar);    // Output: Uncaught ReferenceError: localVar is not
defined
alert(globalVar);   // Output: "I am a global variable"
```

Classes

- ✓ The new version of JavaScript (ES6) introduced the use of classes instead of functions.
- ✓ Prior to ES6, there were only classes and, functions which are callable objects.
- ✓ A class in JavaScript is basically a blueprint or template of the object.
- ✓ New objects can be created from a class.
- ✓ Classes are similar to functions. Here, a class keyword is used instead of a function keyword.
- ✓ Unlike functions classes in JavaScript are not hoisted.
- ✓ The constructor method is used to initialize. The class name is user-defined.

Syntax:

```
class classname {
  constructor(parameter) {
    this.classname = parameter;
  }
}
```

Classes

Example:

```
class Car {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
}

const myCar1 = new Car("Ford", 2014);
const myCar2 = new Car("Audi", 2019);

document.getElementById("demo").innerHTML =
myCar1.name + " " + myCar2.year;
```

Classes

Class Methods

- ✓ Class methods are created with the same syntax as object methods.
- ✓ Use the keyword class to create a class.
- ✓ Always add a constructor() method.
- ✓ Then add any number of methods.

Syntax

```
class ClassName {
  constructor() { ... }
  method_1() { ... }
  method_2() { ... }
  method_3() { ... }
}
```

Classes

Class Methods

Example:

```
class Car {
  constructor(name, year) {
    this.name = name;
    this.year = year;

  }
  age() {
    const date = new Date();
    return date.getFullYear() - this.year;
  }
}

const myCar = new Car("Ford", 2010);
document.getElementById("demo").innerHTML =
"My car is " + myCar.age() + " years old.";
```

JavaScript Inheritance

- ✓ The JavaScript inheritance is a mechanism that allows us to create new classes on the basis of already existing classes.
- ✓ It provides flexibility to the child class to reuse the methods and variables of a parent class.
- ✓ The JavaScript extends keyword is used to create a child class on the basis of a parent class.

Example:

```
<html><body>
<h1>JavaScript Class Inheritance</h1>
<p>Use the "extends" keyword to inherit all methods from
another class.</p>
<p>Use the "super" method to call the parent's
constructor function.</p>
<p id="demo"></p>
<script>
class Car {
  constructor(brand) {
    this.carname = brand;
  }
  present() {
    return 'I have a ' + this.carname;
  }
}
```

```
class Model extends Car {
  constructor(brand, mod) {
    super(brand);
    this.model = mod;
  }
  show() {
    return this.present() + ', it is a ' +
this.model;
  }}
const myCar = new Model("Ford", "Mustang");
document.getElementById("demo").innerHTML =
myCar.show();
</script>
</body>
</html>
```

Events in JavaScript

Events

An HTML event can be something the browser does, or something a user does.

Here are some examples of HTML events:

- ✓ An HTML web page has finished loading
- ✓ An HTML input field was changed
- ✓ An HTML button was clicked

Example

```
<button
onclick="document.getElementById('demo').innerHTML
= Date()">The time is?</button>
```

Events in JavaScript

Event	Description
onchange	An HTML element has been changed
onclick	The user clicks an HTML element
Onmouseover	The user moves the mouse over an HTML element
Onmouseout	The user moves the mouse away from an HTML element
onkeydown	The user pushes a keyboard key
onload	The browser has finished loading the page

JavaScript Event Handlers

- ✓ Event handlers can be used to handle and verify user input, user actions, and browser actions:
- ✓ Things that should be done every time a page loads
- ✓ Things that should be done when the page is closed
- ✓ Action that should be performed when a user clicks a button
- ✓ Content that should be verified when a user inputs data

Objects

- ✓ object is a collection of key-value pairs where each key maps to a value.
- ✓ There are different ways to create objects in JavaScript, including object literals, constructor functions, and classes.
- ✓ An object literal is a comma-separated list of key-value pairs enclosed in curly braces {}.

Objects

Example:

```
let person = {
  name: "John Doe",
  age: 25,
  address: {
    street: "123 Main St",
    city: "Anytown",
    state: "CA",
    zip: "12345"
  }
};

document.getElementById("demo").innerHTML=(person.name); // "John Doe"
// document.getElementById("demo").innerHTML=(person["age"]); // 25
// document.getElementById("demo").innerHTML=(person.address.city); //
"Anytown"
// document.getElementById("demo").innerHTML=(person["address"]["zip"]); //
"12345"
//document.getElementById("demo").innerHTML=Object.values(person);
```

Objects

Object Methods:

In addition to properties, objects can also have methods. A method is a function that is a property of an object.

Example:

```
let person = {
  name: "John Doe",
  age: 25,
  address: {
    street: "123 Main St",
    city: "Anytown",
    state: "CA",
    zip: "12345"
  },
  sayHello: function() {
    document.getElementById("demo").innerHTML =("Hello, my name is " +
this.address.city);
  }
};

person.sayHello(); // "Hello, my name is John Doe"
```

Objects

Constructor Functions

Constructor functions are a way to create multiple objects with similar properties and methods. A constructor function is defined using the function keyword and the this keyword is used to assign values to object properties.

Example:

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

let person = new Person("John Doe", 25, {
  street: "123 Main St",
  city: "Anytown",
  state: "CA",
  zip: "12345"
});
```

spread operator

- ✓ The spread operator (...) is a powerful tool that allows you to manipulate arrays and objects in various ways. When used with objects, it allows you to clone an object and combine multiple objects into a single object.

Cloning an Object with Spread Operator

- ✓ When you want to create a new object with the same properties as an existing object, you can use the spread operator to clone the object. The syntax for cloning an object with the spread operator is as follows.

Example

```
const originalObj = { key1: 'value1', key2: 'value2' };
const clonedObj = { ...originalObj };
```

spread operator

Combining Objects with Spread Operator

The spread operator can also be used to combine multiple objects into a single object. The syntax for combining objects with the spread operator is as follows.

Example:

```
const obj1 = { key1: 'value1', key2: 'value2' };
const obj2 = { key3: 'value3', key4: 'value4' };
const combinedObj = { ...obj1, ...obj2 };
```

Destructuring Objects

Basic Object Destructuring

To destructure an object, you use curly braces {} and provide the names of the properties you want to extract as variable names.

Example:

```
const person = { name: 'John', age: 30, location: 'New York' };
const { name, age } = person;
alert(name); // output: 'John'
alert(age); // output: 30
```

Destructuring Objects

Renaming Destructured Variables

You can also rename the variables when destructuring an object by using a colon : followed by the new variable name.

Example:

```
const person = { name: 'John', age: 30, location: 'New York' };
const { name: personName, age: personAge } = person;
alert(personName); // output: 'John'
alert(personAge); // output: 30
```

Destructuring Objects

Default Values

You can also provide default values for the variables when destructuring an object by using the equal sign =.

Example:

```
const person = { name: 'John', location: 'New York' };
const { name, age = 30 } = person;
alert(name); // output: 'John'
alert(age); // output: 30
```

Destructuring Objects

Nested Object Destructuring

You can also destructure nested objects in JavaScript.

Example:

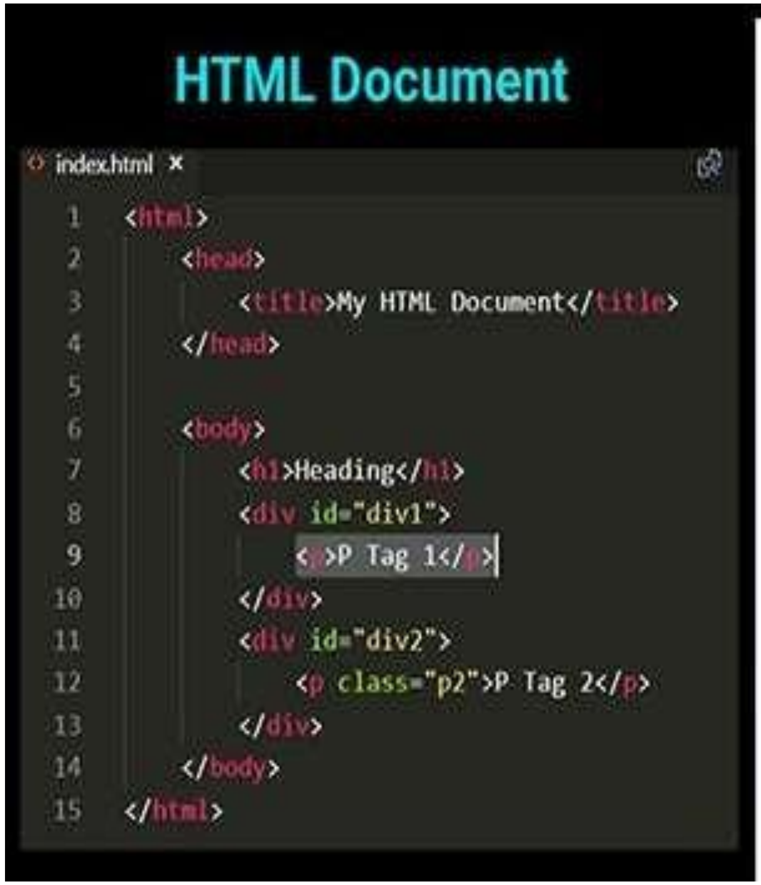
```
const person = { name: 'John', address: { city: 'New York', state: 'NY' } };
const { name, address: { city } } = person;
alert(name); // output: 'John'
alert(city); // output: 'New York'
```

Window Object

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.

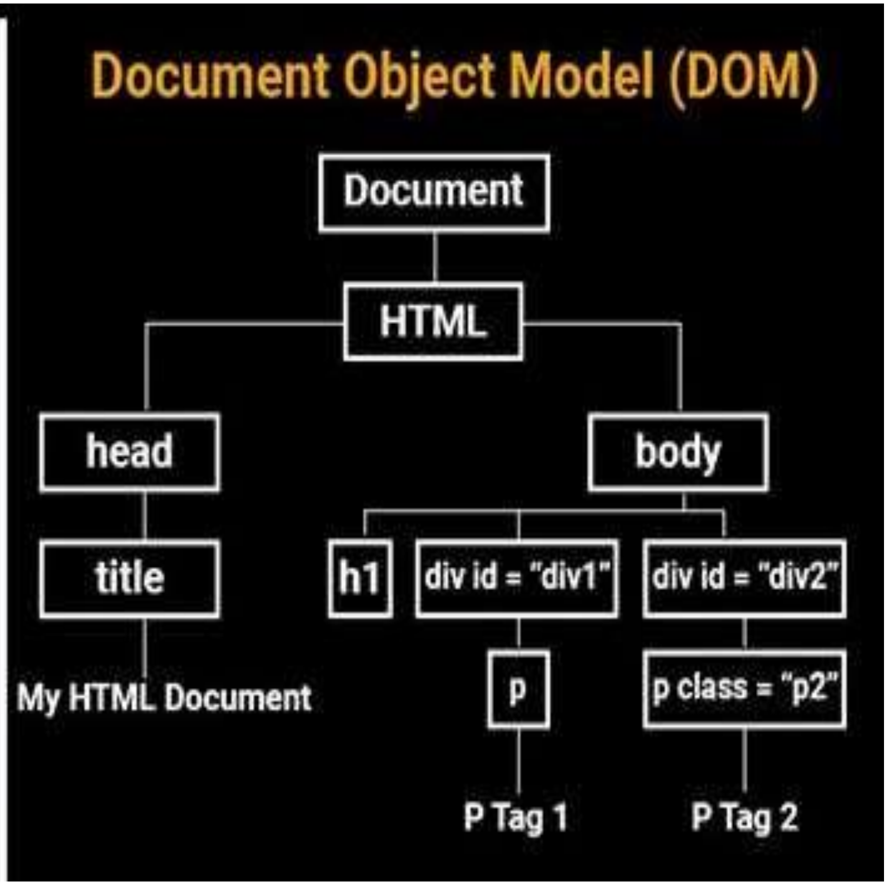
Document Object Model

✓ The document object represents the whole html document.



```html
<html>
    <head>
        <title>My HTML Document</title>
    </head>

    <body>
        <h1>Heading</h1>
        <div id="div1">
            <p>P Tag 1</p>
        </div>
        <div id="div2">
            <p class="p2">P Tag 2</p>
        </div>
    </body>
</html>
```



Methods of document object

Method	**Description**
write("string")	writes the given string on the doucment.
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.

getElementById()

Example:

example.html

```
 <p id = "demo">Hai</p>
```

example..js

```
window.onload = function(){
document.getElementById("demo").style.c
olor = "Red";
}
```

getElementsByName()

Example

Example.html

```
<p name = "exa">Hello</p>
```

Example.js

```
window.onload = function(){
document.getElementsByName("exa")[0].s
tyle.color = "red";
}
```

getElementsByTagName()

Example

Example.html

```
<p class ="exa">Hello</p>
```

Example.js

```
window.onload = function(){
document.getElementsByTagName("p")[0]
.style.backgroundColor = "green";
}
```

getElementsByClassName()

Example

Example.html

```
<p class ="exa">Hello</p>
```

Example.js

```
window.onload = function(){
document.getElementsByClassName("exa")
[0].style.border = "1px solid red";
}
```

Arrays

An array is an ordered collection of values, which can be of any data type, including other arrays.

Creating an Array

You can create an array in JavaScript by using square brackets [] and separating the values with commas.

Example1:

```
        const myArray = [1, 2, 3, 'four', true];
```

Example2:

```
        const myArray = [];
        myArray[0]= 1;
        myArray[1]= 2;
        myArray[2]= “four";
        myArray [3]=true;
```

Arrays

Accessing Array Elements

You can access the elements of an array using their index. The index of the first element in an array is 0, and the index of the last element is the array's length minus one.

Example:

```
const myArray = [1, 2, 3, 'four', true];
alert(myArray[0]); // output: 1
alert(myArray[3]); // output: 'four'
alert(myArray[4]); // output: true
```

Arrays

Modifying Array Elements

You can modify the elements of an array by assigning new values to their indexes.

Example:

```
const myArray = [1, 2, 3, 'four', true];
myArray[1] = 'two';
myArray[3] = 4;
alert(myArray); // output: [1, 'two', 3, 4, true]
```

Arrays

Array Methods

JavaScript provides many built-in methods for working with arrays, such as push(), pop(), shift(), unshift(), splice(), concat(), slice(), forEach(), and more.

push()

The push() method adds one or more elements to the end of an array and returns the new length of the array.

Example:

```
const numbers = [1, 2, 3];
numbers.push(4, 5);
alert(numbers); // [1, 2, 3, 4, 5]
```

pop()

The pop() method removes the last element from an array and returns that element.

Example:

```
const numbers = [1, 2, 3];
const lastNumber = numbers.pop();
alert(lastNumber); // 3
alert(numbers); // [1, 2]
```

Arrays

shift()

The shift() method removes the first element from an array and returns that element.

Example:

```
const numbers = [1, 2, 3];
const firstNumber = numbers.shift();
alert(firstNumber); // 1
alert(numbers); // [2, 3]
```

unshift()

The unshift() method adds one or more elements to the beginning of an array and returns the new length of the array.

Example:

```
const numbers = [1, 2, 3];
numbers.unshift(0, -1);
alert(numbers); // [-1, 0, 1, 2, 3]
```

slice()

The slice() method returns a shallow copy of a portion of an array into a new array. The original array is not modified.

Example:

```
const numbers = [1, 2, 3, 4, 5];
const subset = numbers.slice(1, 4);
alert(subset); // [2, 3, 4]
alert(numbers); // [1, 2, 3, 4, 5]
```

splice()

The splice() method adds or removes elements from an array at a specific index.

Example:

```
const numbers = [1, 2, 3, 4, 5];
numbers.splice(2, 1); // remove 1 element starting at index 2
alert(numbers); // [1, 2, 4, 5]
numbers.splice(2, 0, 'a', 'b'); // add 'a' and 'b' at index 2
alert(numbers); // [1, 2, 'a', 'b', 4, 5]
```

forEach()

The forEach() method executes a provided function once for each element in an array.

Example:

```
const numbers = [1, 2, 3];
numbers.forEach(function(number) {
 alert(number);
});
```

map()

The map() method creates a new array with the results of calling a provided function on every element in the calling array.

Example:

```
const numbers = [1, 2, 3];
const doubledNumbers = numbers.map(function(number) {
  return number * 2;
});
alert(doubledNumbers); // [2, 4, 6]
```

filter()

The filter() method creates a new array with all elements that pass the test implemented by the provided function.

Example:

```
const numbers = [1, 2, 3, 4, 5];
const evenNumbers = numbers.filter(function(number) {
  return number % 2 === 0;
});
console.log(evenNumbers); // [2, 4]
```

Js Callbacks

- ✓ A callback is a function passed as an argument to another function This technique allows a function to call another function
- ✓ A callback function can run after another function has finished

Function Sequence

JavaScript functions are executed in the sequence they are called. Not in the sequence they are defined.

Example:

```
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}
function myFirst() {
  myDisplayer("Hello");
}
function mySecond() {
  myDisplayer("Goodbye");
}
myFirst();
mySecond();
```

Js Callbacks

Sequence Control

- ✓ Sometimes you would like to have better control over when to execute a function.
- ✓ Suppose you want to do a calculation, and then display the result.
- ✓ You could call a calculator function (myCalculator), save the result, and then call another function (myDisplayer) to display the result.

Example:

```
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}
function myCalculator(num1, num2) {
  let sum = num1 + num2;
  return sum;
}
let result = myCalculator(5, 5);
myDisplayer(result);
```

Js Callbacks

JavaScript Callbacks

Using a callback, you could call the calculator function (myCalculator) with a callback (myCallback), and let the calculator function run the callback after the calculation is finished:

Example:

```
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}
function myCalculator(num1, num2, myCallback) {
  let sum = num1 + num2;
  myCallback(sum);
}
myCalculator(5, 5, myDisplayer);
```

Js Callbacks

Asynchronous

- ✓ Functions running in parallel with other functions are called asynchronous

Example:

```
function myDisplayer(something) {
  document.getElementById("demo").innerHTML = something;
}
function myCalculator(num1, num2, myCallback) {
  let sum = num1 + num2;
  myCallback(sum);
}
myCalculator(5, 5, myDisplayer);
```

Js Callbacks

Await

- ✓ When using the JavaScript function setTimeout(), you can specify a callback function to be executed on time-out.

Example:

```
setTimeout(myFunction, 3000);

function myFunction() {
  document.getElementById("demo").innerHTML = "I love India !!";
}
```