Topics to be covered: Expression and Operators in JS, Types of Operators, other operators and Keyword, Important points on Bigint and String operations; Functions in JS- Function declaration, Function expression, IIFE, Function declaration – using constructor, based on condition, Calling or Invoking function, Recursive function, Function hoisting, Function Scope – Nested functions, Multiply-nested functions – preservation of variables, name conflict, using the arguments object, Function Parameter – Default and Rest Parameters, Arrow Function

Expression and Operators in JS

In JavaScript at high level, an *expression* is a valid unit of code that resolves to a value.
There are two types of expressions:

- those that have side effects (such as assigning values)
  Example x = 7
- those that purely *evaluate*.
  Example 3 + 4

As the examples above also illustrate, all complex expressions are joined by *operators*, such as = and +.

Types of operators in JavaScript

- Assignment operators
- Comparison operators
- Arithmetic operators
- Bitwise operators
- Logical operators
- Conditional (ternary) operator
- Comma operator
- Other operators

Assignment operators

Name	Shorthand operator	Meaning
Assignment	x = f()	x = f()
Addition assignment	x += f()	x = x + f()
Subtraction assignment	x -= f()	x = x - f()
Multiplication assignment	x *= f()	x = x * f()
Division assignment	x /= f()	x = x / f()
Remainder assignment	x %= f()	x = x % f()
Exponentiation assignment	x **= f()	x = x ** f()
Left shift assignment	x <<= f()	x = x << f()
Right shift assignment	x >>= f()	x = x >> f()
Unsigned right shift assignment	x >>>= f()	x = x >>> f()
Bitwise AND assignment	x &= f()	x = x & f()
Bitwise XOR assignment	x ^= f()	x = x ^ f()

Bitwise OR assignment	x |= f()	x = x | f()
Logical AND assignment	x &&= f()	x && (x = f())
Logical OR assignment	x ||= f()	x || (x = f())
Nullish coalescing assignment	x ??= f()	x ?? (x = f())

Comparison operators

Operator	Description	Examples returning true
Equal (==)	Returns true if the operands are equal.	“3” == var1
Not equal (!=)	Returns true if the operands are not equal.	var1 != 4 var2 != "3"
Strict equal (===)	Returns true if the operands are equal and of the same type.	3 === var1
Strict not equal (!==)	Returns true if the operands are of the same type but not equal, or are of different type.	var1!== "3" 3 !== '3'
Greater than (>)	Returns true if the left operand is greater than the right operand.	var2 > var1 "12" > 2
Greater than or equal (>=)	Returns true if the left operand is greater than or equal to the right operand.	var2 >= var1 var1 >= 3
Less than (<)	Returns true if the left operand is less than the right operand.	var1 < var2 "2" < 12
Less than or equal (<=)	Returns true if the left operand is less than or equal to the right operand.	var1 <= var2 var2 <= 5

Arithmetic operators

Operator	Description	Example
Remainder (%)	Binary operator. Returns the integer remainder of dividing the two operands.	12 % 5 returns 2.
Increment (++)	Unary operator. Adds one to its operand. If used as a prefix operator (++x), returns the value of its operand after adding one; if used as a postfix operator (x++), returns the value of its operand before adding one.	If x is 3, then ++x sets x to 4 and returns 4, whereas x++ returns 3 and, only then, sets x to 4.
Decrement (--)	Unary operator. Subtracts one from its operand. The return value is analogous to that for the increment operator.	If x is 3, then --x sets x to 2 and returns 2, whereas x-- returns 3 and, only then, sets x to 2.
Unary negation (-)	Unary operator. Returns the negation of its operand.	If x is 3, then -x returns -3.

Operator	Description	Example
Unary plus (+)	Unary operator. Attempts to convert the operand to a number, if it is not already.	+"3" returns 3. +true returns 1.
Exponentiation operator (**)	Calculates the base to the exponent power, that is, base^exponent	2 ** 3 returns 8. 10 ** -1 returns 0.1.
Division operator (/)	Division '/' operator performs division on two operands (divide the numerator by the denominator).	10 / 6 returns 1.6666666666666667

Bitwise Operators

Operator	Usage	Description
Bitwise AND	a & b	Returns a one in each bit position for which the corresponding bits of both operands are ones.
Bitwise OR	a | b	Returns a zero in each bit position for which the corresponding bits of both operands are zeros.
Bitwise XOR	a ^ b	Returns a zero in each bit position for which the corresponding bits are the same. [Returns a one in each bit position for which the corresponding bits are different.]
Bitwise NOT	~ a	Inverts the bits of its operand.
Left shift	a << b	Shifts a in binary representation b bits to the left, shifting in zeros from the right.
Sign-propagating right shift	a >> b	Shifts a in binary representation b bits to the right, discarding bits shifted off.
Zero-fill right shift	a >>> b	Shifts a in binary representation b bits to the right, discarding bits shifted off, and shifting in zeros from the left.

Logical operators

Operator	Usage	Description
Logical AND (&&)	expr1 && expr2	Returns expr1 if it can be converted to false; otherwise, returns expr2. Thus, when used with Boolean values, && returns true if both operands are true; otherwise, returns false.
Logical OR (||)	expr1 || expr2	Returns expr1 if it can be converted to true; otherwise, returns expr2. Thus, when used with Boolean values, || returns true if either operand is true; if both are false, returns false.
Logical NOT (!)	!expr	Returns false if its single operand that can be converted to true; otherwise, returns true.

Conditional (Ternary) Operator

The conditional operator is the only JavaScript operator that takes three operands. The operator can have one of two values based on a condition.

The syntax is: condition ? val1 : val2

Comma operator

The comma operator (,) evaluates both of its operands and returns the value of the last operand. This operator is primarily used inside a for loop, to allow multiple variables to be updated each time through the loop

Example:

```
for (let i = 0, j = 9; i <= j; i++, j--) {
        console.log(`a[${i}][${j}]= ${a[i][j]}`);
}
```

Other operators and keywords

Operator	Description
delete	The *delete* operator deletes an object's property **Syntax:** delete object.property; delete object[propertyKey]; delete objectName[index];

typeof	The *typeof* operator returns a string indicating the type of the unevaluated operand
	typeof true; // returns "boolean"
	typeof null; // returns "object"

void	The *void* operator specifies an expression to be evaluated without returning a value
in	The *in* operator returns true if the specified property is in the specified object. The syntax is: propNameOrNumber in objectName
instanceof	The *instanceof* operator returns true if the specified object is of the specified object type. The syntax is: objectName instanceof objectType
Grouping operator ( )	The grouping operator ( ) controls the precedence of evaluation in expressions.
new	*new* operator to create an instance of a user-defined object type or of one of the built-in object types
this	*this* keyword to refer to the current object
super	The *super* keyword is used to call functions on an object's parent. It is useful with classes to call the parent constructor

Important points on Bigint operations

- unsigned right shift (>>>), which is not defined for BigInt values
  Example: const d = 8n >>> 2n; // TypeError: BigInts have no unsigned right shift, use >> instead

- BigInts and numbers are not mutually replaceable — you cannot mix them in calculations.

  Example: const a = 1n + 2; // TypeError: Cannot mix BigInt and other types

  This is because BigInt is neither a subset nor a superset of numbers. BigInts have higher precision than numbers when representing large integers, but cannot represent decimals, so implicit conversion on either side might lose precision. Use explicit conversion to signal whether you wish the operation to be a number operation or a BigInt one.

  Example: const a = Number(1n) + 2; // 3
  const b = 1n + BigInt(2); // 3n

- Comparison of Bigint with numbers can be done

  Example: const a = 1n > 2; // false
  const b = 3 > 2n; // true

Important points on String operations

- In addition to the comparison operators, which can be used on string values, the concatenation operator (+) concatenates two string values together, returning another string that is the union of the two operand strings.

  Example: console.log("my " + "string");

  The shorthand assignment operator += can also be used to concatenate strings.

  Example: let mystring = "alpha";
  mystring += "bet";

Functions in JS

Function Declaration

A function definition (also called a **function declaration**, or **function statement**) consists of the function keyword, followed by:

- The name of the function.
- A list of parameters to the function, enclosed in parentheses and separated by commas.
- The JavaScript statements that define the function, enclosed in curly braces, { /* … */ }.

Example:

```
function square(number) {
        return number * number;
}
```

Parameters are essentially passed to functions by value — so if the code within the body of a function assigns a completely new value to a parameter that was passed to the function, the change is not reflected globally or in the code which called that function.

Important Note:

- ✓ When you pass an object as a parameter, if the function changes the object's properties, that change is visible outside the function.
- ✓ When you pass an array as a parameter, if the function changes any of the array's values, that change is visible outside the function.

Function Expression

Functions can also be created by a function expression. Such a function can be **anonymous**; it does not have to have a name.

For example, the function square could have been defined as:

```
const square = function (number) {
  return number * number;
};

console.log(square(4));     // 16
```

However, a name can be provided with a function expression. Providing a name allows the function to refer to itself, and also makes it easier to identify the function in a debugger's stack traces:

```
const factorial = function fac(n) {
  return n < 2 ? 1 : n * fac(n - 1);
};

console.log(factorial(3)); // 6
```

Immediately Invoked Function Expression (IIFE)

An IIFE (Immediately Invoked Function Expression) is a JavaScript function that runs as soon as it is defined.

```
(function square(number) {
      console.log(number * number);
})(5);
```

OR

```
((number)=> {
   console.log(number * number);
})(5);
```

Function declaration using constructor

In addition to defining functions as described here, you can also use the Function constructor to create functions from a string at runtime

Example:

```
const square = new Function("x", "console.log(x*x)");
square(5);
```

Function declaration based on condition

In JavaScript, a function can be defined based on a condition. For example, the following function definition defines myFunc only if num equals 0:

Example:

```
let myFunc;
if (num === 0) {
  myFunc = function (theObject) {
    theObject.make = "Toyota";
  };
}
```

Note: A method is a function that is a property of an object.

Calling or Invoking Function

Defining a function does not execute it. Defining it names the function and specifies what to do when the function is called.

Calling the function actually performs the specified actions with the indicated parameters. For example, if you define the function square, you could call it as follows:

```
function square(number) {
        return number * number;
}

square(5);   //function calling
```

Functions must be in scope when they are called, but the function declaration can be hoisted (appear below the call in the code). The scope of a function declaration is the function in which it is declared (or the entire program, if it is declared at the top level).

Recursive Functions

A function can call itself. For example, here is a function that computes factorials recursively: In fact, recursion itself uses a stack: the function stack.

Example:

```
function factorial(n) {
  if (n === 0 || n === 1) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
```

A function can refer to and call itself. There are three ways for a function to refer to itself:

- The function's name
- arguments.callee
- An in-scope variable that refers to the function

```
const foo = function bar(x) {
   console.log(`${x} Recursion`);
   if(x==1)
      return 1;
   else
      foo(x-1);
}

foo(3);
```

```
const foo = function bar(x) {
   console.log(`${x} Recursion`);
   if(x==1)
      return 1;
   else
      arguments.callee(x-1);
}

foo(3);
```

```
const foo = function bar(x)
{
console.log(`${x}
Recursion`);
   if(x==1)
      return 1;
   else
      bar(x-1);
}

foo(3);
```

Note: It turns out that functions are themselves objects — and in turn, these objects have methods. The call() and apply() methods can be used to achieve this goal.

Function Hoisting

Function declaration can come after function call

```
console.log(square(5)); // 25
function square(n) {
  return n * n;
}
```

Note: Function hoisting only works with function declarations — not with function expressions. The following code will not work:

```
console.log(square(5));  // ReferenceError: Cannot access 'square' before initialization
const square = function (n) {
  return n * n;
};
```

Function Scope

Variables defined inside a function cannot be accessed from anywhere outside the function, because the variable is defined only in the scope of the function. However, a function can access all variables and functions defined inside the scope in which it is defined.

In other words, a function defined in the global scope can access all variables defined in the global scope. A function defined inside another function can also access all variables defined in its parent function, and any other variables to which the parent function has access.

Nested Function

Function can be nested inside another function. The nested (inner) function is private to its containing (outer) function.

- The inner function can be accessed only from statements in the outer function.
- The inner function forms a **closure:** the inner function can use the arguments and variables of the outer function, while the outer function cannot use the arguments and variables of the inner function.

```
function addSquares(a, b) {
  function square(x) {
    return x * x;
  }
  return square(a) + square(b);
}
console.log(addSquares(2, 3)); // 13
console.log(addSquares(3, 4)); // 25
console.log(addSquares(4, 5)); // 41
```

Multiply-nested functions

Functions can be multiply-nested. For example:

- A function (A) contains a function (B), which itself contains a function (C).
- Both functions B and C form closures here. So, B can access A, and C can access B.
- In addition, since C can access B which can access A, C can also access A.

Thus, the closures can contain multiple scopes; they recursively contain the scope of the functions containing it. This is called **scope chaining.**

Preservation of variables

Consider the following example:

```
function A(x) {
  function B(y) {
    function C(z) {
      console.log(x + y + z);
    }
    C(3);
  }
  B(2);
}
A(1); // Logs 6 (which is 1 + 2 + 3)
```

In this example, C accesses B's y and A's x.

This can be done because:

- B forms a closure including A (i.e., B can access A's arguments and variables).
- C forms a closure including B.
- Because C's closure includes B and B's closure includes A, then C's closure also includes A. This means C can access both B and A's arguments and variables. In other words, C chains the scopes of B and A, in that order.

The reverse, however, is not true. A cannot access C, because A cannot access any argument or variable of B, which C is a variable of. Thus, C remains private to only B.

Name Conflict

When two arguments or variables in the scopes of a closure have the same name, there is a name conflict. More nested scopes take precedence. So, the innermost scope takes the highest precedence, while the outermost scope takes the lowest. This is the scope chain. The first on the chain is the innermost scope, and the last is the outermost scope. Consider the following:

```
function outside() {
  const x = 5;
  function inside(x) {
    return x * 2;
  }
  return inside;}
```

Using the arguments object

The arguments of a function are maintained in an array-like object. Within a function, you can address the arguments passed to it as follows:

```
arguments[i];
```

where i is the ordinal number of the argument, starting at 0. So, the first argument passed to a function would be arguments[0]. The total number of arguments is indicated by arguments.length.

Using the arguments object, you can call a function with more arguments than it is formally declared to accept. This is often useful if you don't know in advance how many arguments will be passed to the function. You can use arguments.length to determine the number of arguments actually passed to the function, and then access each argument using the arguments object.

For example, consider a function that concatenates several strings. The only formal argument for the function is a string that specifies the characters that separate the items to concatenate. The function is defined as follows:

```
function myConcat(separator) {
  let result = "";                          // initialize list
                                            // iterate through arguments
  for (let i = 1; i < arguments.length; i++) {
    result += arguments[i] + separator;
  }
  return result;
}
```

Note: The arguments variable is "array-like", but not an array. It is array-like in that it has a numbered index and a length property. However, it does not possess all of the array-manipulation methods.

Functional Parameter

There are two special kinds of parameter syntax: default parameters and rest parameters.

Default Parameters

In JavaScript, parameters of functions default to undefined. However, in some situations it might be useful to set a different default value. This is exactly what default parameters do.

```
function multiply(a, b = 1) {
  return a * b;
}

console.log(multiply(5)); // 5
```

Rest Parameters

The rest parameter syntax allows us to represent an indefinite number of arguments as an array.

In the following example, the function multiply uses rest parameters to collect arguments from the second one to the end. The function then multiplies these by the first argument.

```
function multiply(multiplier, ...theArgs) {
  return theArgs.map((x) => multiplier * x);
}

const arr = multiply(2, 1, 2, 3);
console.log(arr); // [2, 4, 6]
```

Arrow Function

An arrow function expression (also called a *fat arrow* to distinguish from a hypothetical -> syntax in future JavaScript) has a shorter syntax compared to function expressions and does not have its own this, arguments, super, or new.target. Arrow functions are always anonymous.

Two factors influenced the introduction of arrow functions: *shorter functions* and *non-binding* of this.

```
const a = ["Hydrogen", "Helium", "Lithium", "Beryllium"];
const a3 = a.map((s) => s.length);
console.log(a3);
```

PRACTICE SHEET II

Note: Students, please perform the question in following practice sheet on your own

Q. No.	Description
1.	Which of the operator is used to test if a particular property exists or not? a) in b) exist c) within d) exists

2.	What is the output of following code snippet:
	function output(option)
	{
	return (option ? "yes" : "no");
	}
	bool ans=true;
	console.log(output(ans));

3.	What is the output of following code snippet:
	function test(...args) {
	console.log(typeof args);
	}
	test(12);

4.	What does … operator do in JS? a) It is used to spread iterables to individual elements b) It is used to describe a datatype of undefined size c) No such operator exists d) None of the above

5.	What is the output of following code snippet:
	(function(a){
	return (function(){
	console.log(a);
	a = 6;
	})()
	})(21);

6.	What is the output of following code snippet:
	var x=12;
	var y=8;
	var res=eval("x+y");
	document.write(res);

7.	What is the output of following code snippet: console.log(void(2===2));
8.	What is the output of following code snippet: console.log(void 2===2);
9.	The …… operator expects a left-side operand that is or can be converted to a string. A) comparison B) in C) instanceOf D) logical

10.	If either operand is NaN or converts to NaN, then the comparison operator always returns …… A) False B) True C) NaN D) Undefined

11.	What will be output of following code snippet
	const d = 99n >>> 1n;
	console.log(d);

12.	What will be the output of following code snippet
	const a = 1n +2;
	console.log(a);

13.	What will be the output of following code snippet const a = 4n – BigInt(2); console.log(a);

14.	What will be the output of following code snippet
	const a = 9n > 2;
	console.log(a);

15.	What will be the output of following code snippet
	const obj = {firstname:"Rana",
	secondname: "Rana"
	}
	function correctname(obj2){
	obj2.firstname = "Sandeep";
	}
	correctname(obj);
	console.log(obj.firstname);

16.	What will be the output of following code snippet
	function factorial(n) {
	if (n === 0 || n === 1) {
	return 1;
	} else {
	return n * arguments.callee(n - 1);
	}
	}
	console.log(factorial(5));

17.	What will be the output of following code snippet
	console.log(sum(5,6))
	const sum = function (x,y){
	return x+y;
	}

18.	What will be the output of following code snippet
	console.log(sum(5))
	function sum(x, y){
	return x + y;
	}

19.	What will be the output of following code snippet console.log(sum(5)) function sum(x, y=1){ return x + y;}

20.	What will be the output of following code snippet
	function outside() {
	const x = 5;
	function inside(x) {
	return x * 2;
	}
	return inside;}
	console.log(outside());

21.	What will be the output of following code snippet
	function outside() {
	const x = 5;
	function inside(x) {
	return x * 2;
	}
	return inside;
	}
	
	const result = outside();
	console.log(result(10));

22.	What will be the output of following code snippet
	function outside() {
	const x = 5;
	function inside(x) {
	return x * 2;
	}
	return inside;}
	console.log(outside()(10));

23.	What will be the output of following code snippet
	const double = new Function("x", "console.log(x+x)");
	double(5);

24.	What will be the output of following code snippet
	function myFunc(x){
	console.log(arguments[1]+x);
	}
	
	myFunc(3,4);

25.	function modulation(adder, ...args) {
	return args.map((x) => adder + x);
	}
	
	const arr = modulation(2, -1, -2, -3);
	console.log(arr);