Web Hosting Services

https://www.hostinger.in/

https://www.goolhost.com/

You are free to purchase web hosting with any service provider. The above companies offer hosting services often with cheap prices.

Note : Take Your Instructors help to purchase domain and setup Cpanel Web Hosting, connecting Nameservers, to host static websites built so far and to deploy the upcoming web challenges. Try to register a domain for your personal portfolio. Ex : yourname.com or yourname.me etc.,

However, in our further sessions, we will migrate to Google Cloud Services. CPanel is a fundamental tool for web developers to know.

Cpanel Resources

https://www.youtube.com/playlist?list=PLZk46idJS6s54hAX8K79_AY8brgBwPtHd

---

Setup Development Environment for JavaScript:

These are the following tools to work with Javascript:

a. Web Browser (Google Chrome is recommended)
b. Node.js Runtime Environment
c. Online Editors (like https://compiler.code.in/)

Method A : (Google Chrome Web Browser)

- ➢ Quick way to compile Javascript code is your web browser console.
- ➢ In Google Chrome, press F12 or press [Ctrl+Shift+i], You'll get developer tools.
- ➢ Open the 'Console' tab and start writing your Javascript code.

Method BQA: Install Node.js software, a javascript runtime environment built on using Chrome V8 Javascript Engine

- ➢ You can download and install Node.js software based on your operating system.
- ➢ https://nodejs.org/en/download/
- ➢ node file_name.js

*For the IDE suggestion, **Use Visual Studio Code**

https://code.visualstudio.com/download

Javascript Lexical Structures

Character Set :

- JavaScript programs are written using the *Unicode character set*. Unicode is a superset of ASCII and Latin-1 http://www.unicode.org/versions/Unicode13.0.0/
- **Javascript uses the UTF-16** (16-bit Unicode Transformation Format) Unicode character set.
  https://en.wikipedia.org/wiki/UTF-16
- JavaScript *strings* are sequences of unsigned 16-bit values.
- Unicode characters have codepoints that fit in 16 bits and can be represented by a single element of a string.

```
var string = 'p';
console.log(string.length); //1
```

- If your Unicode characters codepoints do not fit in 16 bits then they are encoded following the rules of UTF-16 as a sequence (known as a "**surrogate pair**") of two 16-bit values.

```
var string = '😍';
console.log(string.length); //2
```

Case Sensitivity and Reserved Keywords :

➢ JavaScript is a case-sensitive language.
➢ JavaScript ignores additional white-spaces and additional new lines in the program.
➢ This way, you can write a program more readable and clean for others to understand.

```
1. var string='we';
2. var string =  'we';
3. var String='we';
4. var String  = 'we';
```

From the above example,

➢ Line 1 & 2 are the same because whitespaces are ignored.
➢ Line 1 & 3 are not the same because it's case sensitive at the letter "s to S".
➢ Line 3 & 4 are the same because whitespaces are ignored.
➢ Line 2 & 4 are not the same because it's case sensitive at the letter "s to S". And also they are two different variables.

Unicode Escape Sequences:

- JavaScript defines special sequences of six ASCII characters to represent any 16-bit Unicode codepoint.
- These Unicode escapes begin with the *characters \u and are followed by exactly four hexadecimal digits.*

Let's try to understand with an example. Let's take a string in Spanish called **`¿cómo estás?`**

```
var string1 = '¿cómo estás?';
var string2 = '\u00bfc\u00f3mo est\u00e1s?';
console.log(string1);
console.log(string2);
console.log(string1 === string2); //true
/*
The result is true when you compare string1 and string2 with equality
operator because Javascript interpretes Unicode escapes.
Here the Unicode escape for the characters :
¿ is \u00bf
ó is \u00f3
á is \u00e1
*/
```

You can find the list of Unicode Characters here :
https://en.wikipedia.org/wiki/List_of_Unicode_characters

Comments :

JavaScript supports two styles of comments.

Style 1: Single Line Comment //

- ➢ Typing anything after **//** is ignored in Javascript. It is considered to be a comment line.
- ➢ And it is valid only till that single line.

```
// This is a single-line comment.
```

Style2 :Multi Line Comments: /* */

- ➢ Typing anything between **/* */** is ignored in Javascript. It is considered to be a multi-line comment.
- ➢ It is valid until it is opened with **/*** and closed with ***/.**

```
/*
* This is an example for multi-line comment.
* This is another to explain the program
* Last line of comment
*/
```

Literals :

A literal is a data value that can be used directly in a Javascript program.

Example:

```
11; // The numerical literal
1.2; // Floating Literal
'Hey There!'; // A string literal
```

```
true; // A Boolean Literal
false / // A Boolean Literal
/hackingschool/g; // Regular Expression" literal
null; // used to represent empty objects
```

Identifiers and Reserved Words:

Identifiers:

- ➢ Identifiers are the names given by us for the variables, labels for the loops and functions in Javascript.
- ➢ Rules to be an identifier:
  - ➔ Should start with an alphabet or an underscore (_), or a dollar sign ($).
  - ➔ Reserved words(explained below) cannot be used as Identifiers.
- ➢ Rest all characters can be letters, digits, special characters, underscores, or dollar signs.

Example:

```
hi; // valid
M_d1.2; // valid
$hey; // valid
_lol // valid
'Hey There!'; // Invalid
12dad; // Invalid
.hola // Invalid
@hey // Invalid
```

Reserved Words:

- ➢ Reserved Keywords are those which are reserved for inbuilt Javascript operations purposes. It means you are not authorized to use them as your identifiers.
- ➢ Here is the list of reserved words that you cannot use as an identifier.

abstract	arguments	await*	boolean
break	byte	case	catch
char	class*	const	continue
debugger	default	delete	do
double	else	enum*	eval
export*	extends*	false	final
finally	float	for	function
goto	if	implements	import*
in	instanceof	int	interface
let*	long	native	new
null	package	private	protected
public	return	short	static
super*	switch	synchronized	this
throw	throws	transient	true
try	typeof	var	void
volatile	while	with	yield

Image source: w3schools.com

You can find a few more list here: https://www.w3schools.com/js/js_reserved.asp

---

Semicolons in Javascript:

- Semicolons are used as separators or terminators between the line of codes.
- Semicolons should be assigned with proper attention on operations.
- JavaScript considers the separation of a line with a semicolon if the next nonspace character cannot be interpreted as a continuation of the current statement.

  Example:

```
x = 5  ;
x = 5;
```

  From the above example, it is clear that semicolons ignore whitespaces. Hence, both are the same.

  Example:
  Below codes are equivalent to each other irrespective of the position of the semicolon.
  i) The semicolon after x=5 is not mandatory as the next line has got no connection to it.

```
x = 5;
y = 6
```

  ii) Declaration of these two variables is valid without the semicolon.

```
x = 5
y = 6
```

  iii) Variable declarations are valid only with the mandatory semicolon between them because of two different variable declarations in a single line. It needs separation.

```
x = 5 ; y = 6
```

iv) Below codes are equivalent to each other.

```
//Type - 1
x = 5 ; y = 6;
//Type - 2
x
=
5
;
y=

6;
```

➢ Semicolon separation is not allowed between **return, break** or **continue** and the expression that follows the keyword.

Example 1 :

```
return false; // It returns boolean false
```

Example 2 :

```
return; false; //It terminates at return; and doesn't pass to false
```

Example 3 :

```
return
false
// It terminates at first line return and doesn't pass to false
```

Example 4:

```
return;
false;
// It terminates at first line return and doesn't pass to false
```

Example: 5

```
function dodo1()
{
 return {
     bar: "hello"
 };
}
/* function foo1 returns "hello" as there is a continuation with a
"{" beside return */

function dodo2()
{
 return
 {
     bar: "hello"
 }
//function dodo2 doesn't return "hello" as it terminates at return.
```

➢ Post and pre-increment/decrement operators should be used on the same line.
➢ Line Break in before or after pre-increment/decrement operators will be treated as a semicolon.

Example: 6

```
a
++
b
```

```
//This will interpret as a; ++b;
//Taking a++;b; is WRONG
```

Types and Operators

Variable:

- Variables are placeholder names for different types of values.
- Keywords like **var, const, let** will be used to define variables
- Unlike the other programming languages, they don't determine the **data types** of variables. These keywords can be used for any datatype.
- JavaScript is a **dynamically-typed** language. It means that variables created using var or let keywords can be dynamically re-assigned to a value of another type.
- Variables declaration allows storing a single type of data that can be modified later.
- Variables must be initialized **before using**.
- Multiple variables can be defined in a single line.

Example :

```
var a = 'Apple';
var b = 11;
```

Statements :

- A statement is the smallest building block of a computer program. In this chapter, we will explore a few common cases.
- An empty statement with semicolon evaluates to undefined.
- Any statement that doesn't produce value will be evaluated as undefined.

```
let a = 1; //undefined
1; //1
'text'; //"text"
[]; //undefined
{} //undefined
let a = 1; //undefined
let b = []; //undefined
let c = {}; //undefined
```

Block Statement :

A statement block is simply a sequence of statements enclosed within curly braces.

```
{
    var x = 2;
    var y = 3;
    console.log('2 power 3 = ' + 2 ** 3);
}
```

Note: Empty statement looks like this

```
;  // ; only
```

Declaration Statements:

- Declarations statements are **var, let, const** and **functions.** These statements are defined or declared in order to be used in the program further.

var: The ‘var’ statement declares a single variable or multi variables. The var keyword is followed by a comma-separated list of variables to declare.

Examples:

```
var a; // a simple variable
var b = 2; // One var, one value
var i, j; // Two variables
var tell = 'hello' + b; // A complex initializer
var x = 3.14,
    y = Math.sqrt(16),
    p,
    xeta; // Many variables
var s = 2,
    p = s * s; // Second var uses the first
var d = 4, // Multiple variables...
    k = function(d) {
        return d * d;
    }, // each on its own line
var  y = k(d);
```

Function:

- ➢ If you want to define a function, then it must use the **function** keyword.
- ➢ A JavaScript function is a block of code designed to perform a particular task which executes when "something" invokes it (calls it).
- ➢ A JavaScript function syntax 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: { }

Syntax :

```
function name(parameter1, parameter2, parameter3) {
    // code to be executed
  }
```

Note :

- ➢ Function **parameters** are listed inside the parentheses () in the function definition.
- ➢ Function **arguments** are the values received by the function when it is invoked.

Conditional Statements:

“if - else” condition:

- ➢ When **“ if ”** expression is truthy - statement1 is executed;
- ➢ When **“ if”** expression is falsy - statement2 is executed;

Syntax:

```
if (expression){
```

```
    Statement1;
}
else{
    Statement2;
}
```

Example 1:

```
var mails = 2;
if (mails > 3) {
    console.log('You have 1 new mails.');
} else {
    console.log('You have ' + mails + ' new mails.');
}
//Output : You have 2 new mails.
```

Example 2 :

```
var i = 1;
var j = 2;
var k = 2;
if (i == j) {
    if (j == k) {
        console.log('i equals k');
    }
} else {
    console.log("i doesn't equal j");
}
//Output : i doesn't equal j
```

“if - else if - else” condition:

- ➢ When **“ if ”** expression is truthy - statement1 is executed;
- ➢ When **“ if”** expression is falsy - go to **“else if ”** expression;
- ➢ When **“ else if ”** expression is truthy - statement2 is executed;
- ➢ When **“ else if”** expression is falsy - go to **“else”** expression;
- ➢ **“else”** expression is a default statement when all the above conditions are false.

Syntax :

```
if (condition1) {
    //  statement-1  (block of code to be executed if condition1 is
true)
} else if (condition2) {
    //  statement-2 (block of code to be executed if the condition1 is
false and condition2 is true)
} else {
    //  statement-3 (block of code to be executed if the condition1 is
false and condition2 is false)
}
```

Exercise “ else if ” :

Write a program that prints the numbers from 1 to 100 and for multiples of '3' print "Fizz" instead of the number and for the multiples of '5' print "Buzz".

```
for (i = 1; i <= 100; i++) {
    if (i % 3 === 0 && i % 5 === 0) {
```

```
        console.log('fizz buzz');
    } else if (i % 3 === 0) {
        console.log('fizz');
    } else if (i % 5 === 0) {
        console.log('buzz');
    } else {
        console.log(i);
    }
}
```

“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
}
```

Example :

```
var day;
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';
}
console.log(day);
```

“break” Keyword

- During execution of JavaScript if it reaches a **break** keyword, it comes out of the switch block.
- This will stop the execution of inside the block.

“default” Keyword

- The **default** keyword works when none of the cases matches.
- If there is no **break** then the execution continues with the next **case** without any checks.

An example without “break”:

```
var a = 2 + 2;
switch (a) {
    case 3:
        console.log('Too small');
    case 4:
        console.log('Equals!');
    case 5:
        console.log('Too big');
    default:
        console.log("I don't know such values");
}
//Output :
Equals!
Too big
I don't know such values
```

Any expression can be a **switch/case** argument

Example :

```
let a = 1;
let b = 0;
switch (++a) {
    case 1:
        console.log('When case equals to 1, this will print.');
        break;
    default:
        console.log("Nothing Matched!");
}
//Output : Nothing Matched!
```

Group Case :

```
let a = 3;
switch (a) {
    case 4:
        console.log('Right!');
        break;
    case 3: // grouping cases
    case 5:
        console.log('First Sentence!');
        console.log('Second Sentence');
        break;
    default:
        console.log('The result is strange. Really.');
}
//Output :
First Sentence!
Second Sentence
```

Exercise: Convert the below **switch case** to **if else**

```
switch (browser) {
    case 'Edge':
        console.log('You are Microsoft Edge!');
        break;
    case 'Chrome':
    case 'Firefox':
    case 'Safari':
    case 'Opera':
        console.log('Website supports for all these browsers.');
        break;
    default:
        console.log('OOPS!');
}
```

above **switch** can be converted to below **if else** condition :

```
if (browser == 'Edge') {
    console.log('You are Microsoft Edge!');
} else if (browser == 'Chrome' || browser == 'Firefox' || browser == 'Safari' || browser == 'Opera') {
    console.log('Website supports for all these browsers.');
} else {
    console.log('OOPS!');
}
```

Exercise: Convert the **if** code to **switch** code :

```
let a = 2;
if (a == 0) {
    console.log(0);
}
if (a == 1) {
    console.log(1);
}
if (a == 2 || a == 3) {
    console.log('2,3');
}
```

The above **if** code can be written in **switch** like below :

```
switch (a) {
    case 0:
        console.log(0);
        break;
    case 1:
        console.log(1);
        break;
    case 2:
```

```
    case 3:
        console.log('2,3');
        break;
}
```

Loop Statements (Loops):

➢ Using the Loop statement is the easiest way to repeat the same code multiple times without doing it condition by condition.

While loop:

➢ While the loop is one of the types of loops.

The **while** loop has the following syntax:

```
while (condition) {
    // code
    // so-called "loop body"
}
```

Example :

```
let i = 0;
while (i < 3) {
    console.log(i);
    i++;
}

//output :
0
1
2
```

“do...while” loop :

Syntax :

```
do {
    // loop body
} while (condition);
```

Exercise : Write an algorithm to calculate **(M^N)** using **do while.**

```
var M = 2;
var N = 3;
var result = 1;
do {
    result *= M;
    N--;

} while (N > 0);
console.log(result);
```

“for” loop:

```
for (begin; condition; step) {
    // ... loop body ...
  }
```

Example :

```
for (let i = 0; i < 3; i++) {
    console.log(i);
}
//output : 0 1 2
```

Let’s examine the **for** statement part-by-part:

begin	i = 0	Executes once upon entering the loop.
condition	i < 3	Checked before every loop iteration. If false, the loop stops.
body	console.log(i)	Runs again and again while the condition is truthy.
step	i++	Executes after the body on each iteration.

Exercise: Write a program to find the sum of all the elements in an array using a for loop.

```
var arr = [ 1, 2, 3, 4, 5 ];
var sum = 0;
for (i = 0; i < arr.length; i++) {
    sum += arr[i];
}
console.log(sum);
```

Different forms of for … loop:

➢ Any part of **for** can be skipped. All the below for loops are equivalent.

Example 1: Skip initialisation

```
var i = 0;
for (; i < 3; i++) {
        console.log(i);
}
```

Example 2: Skip increment and write in the body of the loop

```
var i = 0;
for (; i < 3; ) {
    console.log(i);
    i++;
}
```

Example 3 : skip everything in **for**

```
var i = 0;
for (;;) {
    if (i < 3) {
        console.log(i);
        i++;
    } else {
        break;
    }
}
```

Note: Any Empty loop like this goes infinite and leaks memory of your computer.

```
for (;;) {
```

```
    // will go infinite
  }
```

break keyword in **loop:**

```
var sum = 0;
while (true) {
    let value = 0;

    if (!value) break;

    sum += value;
}
console.log('Sum: ' + sum);
```

continue :

The **continue** keyword is a “lighter version” of a **break**. It doesn’t stop the whole loop. Instead, it stops the current iteration.

Example :

```
for (let i = 0; i <= 10; i++) {

    if (i % 2 == 0) continue;

    console.log(i);

}
```

Note: continue/break shouldn’t be used with a **ternary operator.**

---

label:

A label is an identifier with a colon before a loop:

```
labelName: for (...) {
    ...
}
```

Example :

```
first: for (var i = 0; i < 3; i++) {

    second: for (var j = 0; j < 3; j++) {

        if (i === 1) continue first;

        if (j === 1) break second;

        console.log(`${i} & ${j}`);

    }

}
//Output :
0 & 0
2 & 0
```

“for in” loop:

➢ The **for...in** statement iterates over all enumerable properties of an object that are keyed by strings.

```
const object = { a: 1, b: 2, c: 3 };

for (const property in object) {

    console.log(`${property}: ${object[property]}`);

}
```

“try..catch...finally” statement :

Syntax :

```
try {

    // code...

} catch (err) {

    // error handling
}
```

It works like this:

1. First, the code in **try {...}** is executed.
2. If there were no errors, then **catch(err)** is ignored: the execution reaches the end of try and goes on, skipping **catch.**
3. If an error occurs, then the **try** execution is stopped, and control flows to the beginning of **catch(err).** The **err** variable (we can use any name for it) will contain an error object with details about what happened.

Example 1: “try” without “errors”

```
try {
    console.log('Start of try block.');
    console.log('End of try block.');
} catch (err) {
    console.log('Catch is ignored, because there are no errors');
}
//output
Start of try block.

End of try block.
```

Example 2: “try” with “error”

```
try {
    console.log('Start of try runs'); // (1)

     slipped_banana; // error, the variable is not defined occurred so it goes to the catch block!

    console.log('End of try block.'); // (2)
```

```
} catch (err) {
    console.log(`Error has occurred!`); // (3)
}

//output :
Start of try runs
Error has occurred!
```

Note: “try..catch” only works for runtime errors

Properties of error object :
For all built-in errors, the error object has two main properties:

name:
Error name. For instance, for an undefined variable that’s **"ReferenceError".**

message:
Textual message about error details.

There are other non-standard properties available in most environments. One of the most widely used and supported is:

stack:
Current call stack: A string with information about the sequence of nested calls that led to the error. Used for debugging purposes.

Example :

```
try {
    slipped_banana; // error, variable is not defined!
} catch (err) {
    console.log(err.name); // ReferenceError
    console.log(err.message); // slipped_banana is not defined
      console.log(err.stack);  // ReferenceError: slipped_banana is not
defined at (...call stack)
}
```

Nested “try... catch” Example:

```
(function() {
    try {
        try {
            throw new Error('oops');
        } catch (ex) {
            console.error('inner', ex.message);
            throw ex;
        } finally {
            console.log('finally');
            return;
        }
    } catch (ex) {
        console.error('outer', ex.message);
    }
} ) ( );
```

“try…catch…finally” Example:

The **try...catch** construct may have one more code clause finally.

If it exists, it runs in all cases:

- ➢ After the try, if there were no errors,
- ➢ After the catch, if there were errors.

Syntax :

```
try {
    ... try to execute the code ...
 } catch(e) {
    ... handle errors ...
 } finally {
    ... execute always ...
 }
```

Example :

```
try {
    console.log('try');
    if (dodo('Write an error!')) ERROR_CODE();
} catch (e) {
    console.log('catch');
} finally {
    console.log('finally');
}
```

---

debugger statement :

- ➢ The **debugger** keyword stops the execution of JavaScript, and calls (if available) the debugging function.
- ➢ This has the same function as setting a breakpoint in the debugger.

“use strict” statement : (Very Important)

- ➢ **"use strict";** Defines that JavaScript code should be executed in **"strict mode**".
- ➢ Strict mode makes it easier to write "secure" JavaScript.
- ➢ Strict mode changes previously accepted "bad syntax" into real errors.
- ➢ As an example, in normal JavaScript, mistyping a variable name creates a new global variable. In strict mode, this will throw an error, making it impossible to accidentally create a global variable.
- ➢ In normal JavaScript, a developer will not receive any error feedback assigning values to non-writable properties.
- ➢ In strict mode, any assignment to a non-writable property, a getter-only property, a non-existing property, a non-existing variable, or a non-existing object, will throw an error.
- ➢ Using a variable, without declaring it, is not allowed.
- ➢ Using an object, without declaring it, is not allowed.
- ➢ Deleting a variable (or object) is not allowed(with delete keyword)
- ➢ Duplicating a parameter name is not allowed in functions.

Example 1:

```
"use strict";
x = 3.14;      // This will cause an error because x is not declared
```

Example 2 :

```
"use strict";
```

```
myFunction();
function myFunction() {
  y = 3.14; // This will cause an error because y is not declared
}
```

Expression :

A statement that produces a single value other than undefined can be referred to as an expression

```
11+2; //14
```

Operators :

JavaScript has the following types of operators.

1. Assignment operators
2. Comparison operators
3. Arithmetic operators
4. Bitwise operators
5. Logical operators
6. String operators
7. Conditional (ternary) operator
8. Comma operator
9. Unary operators
10. Relational operators

1. Assignment operators:

Name	Shorthand operator	Expression
Assignment	x = y	x = y
Addition assignment	x += y	x = x + y
Subtraction assignment	x -= y	x = x - y
Multiplication assignment	x *= y	x = x * y
Division assignment	x /= y	x = x / y
Remainder assignment	x %= y	x = x % y
Exponentiation assignment	x **= y	x = x ** y
Left shift assignment	x <<= y	x = x << y
Right shift assignment	x >>= y	x = x >> y
Unsigned right shift assignment	x >>>= y	x = x >>> y
Bitwise AND assignment	x &= y	x = x & y
Bitwise XOR assignment	x ^= y	x = x ^ y
Bitwise OR assignment	x |= y	x = x | y

Exponentiation Operator Example:

```
var value = 2 ** 3;
var number = 625;
```

```
console.log('Square Root of number :', number ** 0.5);
```

Unsigned Left Shift Assignment Example:

```
var num = 25;
console.log(num << 2);//100
/*
Here 25 in binary is 11001
Left shift goes on shifting left side by appending “0” to its right.
It gives 1100100 which is 100 in decimal
*/
```

Signed Left Shift Assignment Example:

```
var num = -25;
console.log(num << 2);//100
/*
Here 25 in binary is -(11001)
And the sign remains the same as “-”
The left shift goes on shifting left side by appending “0” to its right.
It gives -(1100100) which is -100 in decimal
*/
```

Unsigned Right Shift Assignment Example:

```
var num = 25;
console.log(num >> 2);//6
/*
Here 25 in binary is 11001
The right shift goes on shifting right side by eliminating the binary
values which are to its right.
It gives 110 which is 6 in decimal
*/
```

Signed Right Shift Assignment Example:

```
var num = -25;
console.log(num >> 2);//-7
/*
Here 25 in binary is 0001 1001
As the number is negative, we need to do 2’s complement first, then
shifting right and 2’s complementing again to get the result.
11001 2’s complement is 1110 0110
                              1
                         ----------
                         1110 01~~11~~ (eliminate shifting number = 2)
                         1111 1001 (above step is converted to 8 Bit)
                         0000 0110
                                1
                        -----------
                         0000 0111 = (-7) in decimal
*/
```

Bitwise AND assignment:

```
var a = 2, b = 5
console.log(a&b);//0
/*
 a = 2 = 0010
 b = 5 = 0101
         ----
   a&b = 0000  (Using AND truth table)
*/
```

Bitwise XOR assignment:

```
var a = 2, b = 5
console.log(a^b);//7
/*
 a = 2 = 0010
 b = 5 = 0101
         ----
   a^b = 0111  (Using XOR truth table)
*/
```

Bitwise OR assignment:

```
var a = 2, b = 5
console.log(a|b);//7
/*
 a = 2 = 0010
 b = 5 = 0101
         ----
   a|b = 0111  (Using OR truth table)
*/
```

---

2. Comparison Operators

Operator	Description
Equal (==)	Returns true if the operands are equal.
Not equal (!=)	Returns true if the operands are not equal.
Strict equal (===)	Returns true if the operands are equal and of the same type. See also Object.is and sameness in JS.
Strict not equal (!==)	Returns true if the operands are of the same type but not equal, or are of a different type.
Greater than (>)	Returns true if the left operand is greater than the right operand.
Greater than or equal (>=)	Returns true if the left operand is greater than or equal to the right operand.
Less than (<)	Returns true if the left operand is less than the right operand.

Less than or equal (<=)	Returns true if the left operand is less than or equal to the right operand.

3. Arithmetic Operators

Operator	Description
Remainder (%)	Binary operator. Returns the integer remainder of dividing the two operands.
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.
Decrement (--)	Unary operator. Subtracts one from its operand. The return value is analogous to that for the increment operator.
Unary plus (+)	Unary operator. Attempts to convert the operand to a number if it is not already.

Remainder Example:

```
var a = 2, b = 5
console.log(a%b);//2
/*
     5)2(2
       0
     ----
       2
   a%b = 2
*/
```

PRE and POST Increment Example:

```
var num = 0;
console.log(num++ + ++num); //2
var num = 0;
console.log(++num + num); //2
var num = 0;
console.log(num++ + num++); //1
var num = 0;
console.log(num++ + num++ + ++num); //4
```

PRE and POST Decrement Example:

```
var num = 0;
console.log(num-- - --num); //2
var num = 0;
console.log(--num - num); //0
var num = 0;
console.log(num-- - num--); //1
var num = 0;
console.log(num-- - num-- - --num); //4
```

---

4. 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.

5. 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.

Logical AND

```
console.log(1 && 2 && 3 && 4); // 4
console.log(9 && 8); //8
console.log(9 && 9 && 9 && 0); //0
console.log(0 && 1 && 2 && 3); //0
```

Logical OR

```
console.log(1 || 2 || 3 || 4); // 1
console.log(9 || 8); //9
console.log(9 || 9 || 9 || 0); //9
console.log(0 || 1 || 2 || 3); //1
```

Logical NOT

```
console.log(!0); //true
console.log(!false);//true
console.log(!2);//false
console.log(!true); //false
console.log(!1); //false
```

---

6. “String” Operators :

```
console.log('Hello ' + 'World'); // concatenates two string "Hello World".
var string1 = 'Hello';
var string2 = 'World!';
var result = `Strings can be concatenated like this : ${string1} ${string2}`;
console.log(result);
```

“typeof” operator:

The **typeof** operator is used in either of the following ways:

```
var myFun = new Function('5 + 2');
var shape = 'round';
var size = 1;
var foo = [ 'Apple', 'Mango', 'Orange' ];
var today = new Date();
console.log(typeof myFun); // returns "function"
console.log(typeof shape); // returns "string"
console.log(typeof size); // returns "number"
console.log(typeof foo); // returns "object"
console.log(typeof today); // returns "object"
```

---

7. “Conditional Ternary” Operator

```
var a=4,b=2;
(a>b)?console.log("a is greater"):console.log("b is greater");

/* Conditional territory operator returns the value after? till: if it is true, and returns the value after: if it false.*/
```

---

8. “Comma” operator

The comma operator (,) simply evaluates both of its operands and returns the value of the last operand.

```
var a = (1, 2, 3, 4, 5, 6);
console.log(a); //6
```

---

9. Unary plus Example:

```
var num = "2";
console.log(+num)//2
```

---

10. **Relational Operator:** A relational operator compares its operands and returns a Boolean value based on whether the comparison is true.

```
var trees = [ 'redwood', 'bay', 'cedar', 'oak', 'maple' ];
0 in trees; // returns true
3 in trees; // returns true
```

```
6 in trees; // returns false
```

Javascript Operators Precedence

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence

---

Javascript Data Types:

Javascript types are divided into two types.

- Primitive Data Types
- Object Data Types (Composite Data Types)

Primitive Data Types :

1. null
2. undefined
3. number
4. bigint
5. string
6. boolean
7. symbol

1. **null:** The value null represents the intentional absence of any object value. It is one of JavaScript's primitive values and is treated as falsy for boolean operations.
2. **undefined:** undefined is a primitive value automatically assigned to variables that have just been declared,
3. **number:** In JavaScript, Number is a numeric data type in the double-precision 64-bit floating-point format (IEEE 754).
4. **string:** A string is a sequence of characters used to represent text. In JavaScript, a String is one of the primitive values**.**
5. **bigint:** In JavaScript, BigInt is a numeric data type that can represent integers in the arbitrary precision(Big Number Arithmetic) format.
6. **boolean:** A Boolean is a logical data type that can have only the values true or false.
7. **symbol:** The Symbol primitive provides a way to define a completely unique key. It is used for an anonymous, unique value, mostly in objects and to make object properties as private

Object Data Types :

1. Object
2. Array
3. Function

Array: JavaScript arrays are used to store multiple values in a single variable.

- ➢ An array is an ordered collection of values.
- ➢ Each value is called an **“element”**, and each element has a numeric position in the array, known as its **“index”.**

  Note: Array index starts from **“0”.**

- ➢ JavaScript array elements can be of any type, and different elements of the same array may be of different types.

➢ Array elements may even be objects or other arrays, arrays of objects and arrays of arrays which allows you to create complex data structures.

The easiest way to create an array is with an array literal, which is simply a comma-separated list of array elements within square brackets.

Types of Array Declarations :

1. var arr = new Array();
2. var arr = [ ] ;

➔ Call it with no arguments:

```
var a = new Array( );
```

This method creates an empty array with no elements and is equivalent to the array literal [ ].

➔ Call it with a single numeric argument, which specifies a length:

```
var a = new Array(10);
```

This technique creates an array with the specified length. This form of the Array() can be used to preallocate an array when you know in advance how many elements will be required.

Example :

```
var arr = ["The", "Hacking", "School"];
```

➢ The strings 'The' , 'Hacking', 'School' are known as Array Elements.
➢ Array Elements can be accessed using the following syntax

```
var arr = [ 'The', 'Hacking', 'School' ];
console.log(arr[0]); //The
console.log(arr[1]); //Hacking
console.log(arr[2]); //School
```

➢ The values may be arbitrary expressions than constants: var base = 1024; var table = [base, base+1, base+2, base+3];
➢ Array literals can contain object literals or other array literals:

```
var b = [ [1,{x:1, y:2} ], [2, {x:3, y:4} ] ];
```

➢ If you omit a value from an array literal, the omitted element is given the value undefined:

```
var count = [1,,3]; // An array with 3 elements, the middle one undefined.
var undefs = [,,];  // An array with 2 elements, both undefined.
```

➢ An array literal syntax allows an optional trailing comma, so [, ,] has only two elements, not three.
➢ Another way to create an array is with the Array() constructor. You can invoke this constructor in three distinct ways:

We can update the values in array :

```
var arr = [ 'The', 'Hacking', 'School' ];
arr[2] = 'School - Bootcamp';
console.log(arr[2]);
console.log(arr);
//output :
```

```
School - Bootcamp
[ 'The', 'Hacking', 'School - Bootcamp' ]
```

We can add new values to the array :

```
var arr = [ 'The', 'Hacking', 'School' ];
arr[3] = 'Bootcamp';
console.log(arr);
```

To find the total number of elements in an array

```
var arr = [ 'The', 'Hacking', 'School' ];
console.log(arr.length); //3
```

You can store any data type of elements inside array in Javascript :

```
var  arr = [
    'The Hacking School',
    { address: 'Hyderabad' },
    true,
    function() {
        console.log('Best Bootcamp');
    }
];
```

The above array elements can be accessed like this

```
console.log(arr[0]); //Prints the string 'The Hacking School'
console.log(arr[1]); //Prints the object {address : 'Hyderabad}
console.log(arr[1].address); //Prints Hyderabad
console.log(arr[2]); //prints the boolean true
arr[3](); //Prints Best Bootcamp by invoking the function
```

These are the operations allowed on Array :

1. pop
2. shift
3. push
4. unshift

1. pop

➢ Extracts the last element of the array and returns it.
➢ Removes the last element from the array.

```
var arr = [ 'The', 'Hacking', 'School', 'Bootcamp' ];
console.log(arr.pop()); //Bootcamp
console.log(arr); //[ 'The', 'Hacking', 'School' ]
```

2. shift

➢ Extracts the first element of the array and returns it:
➢ Removes the first element from the array.

```
var arr = [ 'The', 'Hacking', 'School', 'Bootcamp' ];
console.log(arr.shift()); //The
console.log(arr); //[ 'Hacking', 'School', 'Bootcamp' ]
```

3. push

➢ Adds an element to the end of the array.

```
var arr = [ 'The', 'Hacking', 'School' ];
arr.push('Bootcamp');
console.log(arr); //[ 'The', 'Hacking', 'School', 'Bootcamp' ]
```

> **Note :**
>
> arr.push('Bootcamp') is equivalent to
>
> arr[arr.length] = 'Bootcamp'

4. unshift

➢ Adds an element to the beginning of the array.

```
var arr = [ 'The', 'Hacking', 'School' ];
arr.unshift('Bootcamp');
console.log(arr); //[ 'Bootcamp','The', 'Hacking', 'School' ]
```

Few Key Points on Array :

➢ An array is an object in Javascript.
➢ Because of its object type, The square brackets used to access a property **arr[0]** actually come from the object syntax.
➢ Array operations **push/pop** works fast, while **shift/unshift** are slow.
➢ If we copy an array, it will use the same reference

```
var bootcamp = [ 'The', 'Hacking' ];
var school = bootcamp;
// copy by reference (two variables reference the same array)
console.log(bootcamp === school); // true
bootcamp.push('School');
// modify the array and that will update in both arrays
console.log(school); // [ 'The', 'Hacking', 'School' ]
console.log(bootcamp); //[ 'The', 'Hacking', 'School' ]
```

Loops to access Array Elements

“for loop”:

```
var bootcamp = [ 'The', 'Hacking', 'School' ];
for (let i = 0; i < bootcamp.length; i++) {
    console.log(bootcamp[i]);
}

//Output:
The
Hacking
School
```

“for..of” loop :

```
var bootcamp = [ 'The', 'Hacking', 'School' ];
for (let elements of bootcamp) {
    console.log(elements);
}
```

```
//Output:
The
Hacking
School
```

Few Key Points on array length property :

Key point 1:

```
var bootcamp = [];
bootcamp[11] = 'The Hacking School';
console.log(bootcamp.length); // 12 but not 1
```

Key point 2:

```
let bootcamp = [ 'The', 'Hacking', 'School' ];
console.log(bootcamp[3]); // 'School'
bootcamp.length = 2; // truncates to 2 elements
console.log(bootcamp); // [ 'The', 'Hacking' ]
console.log(bootcamp[3]); // gives undefined and the old values do not return because its deleted from memory
```

Multidimensional arrays:

1d Array: A group of elements.

Example :

```
let arr = [ 1, 2, 3, 4, 5, 6 ];
console.log(arr);
//output
 [ 1, 2, 3, 4, 5, 6 ];
```

2d Array: A group of 1d arrays.

```
let arr =
    [
        [ 1, 2, 3 ],
        [ 4, 5, 6 ],
        [ 7, 8, 9 ]
    ];
console.log(arr);

//output
[ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ]
```

3d Array: Group of 2d arrays.

```
let arr =
    [
        [ [ 1, 2, 3 ], [ 4, 5, 6 ] ],
        [ [ 7, 8, 9 ], [ 10, 11,12 ] ]
    ];
console.log(arr);
```

```
//output
[ [ [ 1, 2, 3 ], [ 4, 5, 6 ] ], [ [ 7, 8, 9 ], [ 10, 11, 12 ] ] ]
```

Exercise 1:

2d Array: Write an algorithm to sum two matrices.

```
//Addition of Matrices
var a = [ [ 1, 2 ], [ 3, 4 ] ];
var b = [ [ 2, 3 ], [ 4, 5 ] ];
var c = [ [], [] ];
for (i = 0; i < a.length; i++) {
    for (j = 0; j < a[i].length; j++) {
        c[i][j] = a[i][j] + b[i][j];
    }
}
console.log(c);
//Output :
[ [ 3, 5 ], [ 7, 9 ] ]
```

Exercise 2: Write an algorithm to remove duplicate elements in an array?

```
var n = [ 1, 1, 2, 2, 2, 3, 3, 3, 4, 5 ];
var arr = [];
function dup(n) {
    n = n.sort((a, b) => a - b);
    for (i = 0; i < n.length; i++) {
        if (n[i] - n[i + 1] !== 0) {
            arr.push(n[i]);
        }
    }
    console.log(arr);
}
dup(n)
//Output
[ 1, 2, 3, 4, 5 ]
```

Exercise 3 : Write an algorithm to sort all elements in 2d array.

```
//Sorting of elements in the 2d array :
var a = [ [ 1, 3, 2 ], [ 3, 4, 2, 1 ], [ 2, 1, 3 ] ];
function sort(a) {
    for (i = 0; i < a.length; i++) {
        a[i] = a[i].sort((a, b) => a - b);
    }
    return a;
}
console.log(sort(a));
//Output : [ [ 1, 2, 3 ], [ 1, 2, 3, 4 ], [ 1, 2, 3 ] ]
```

Exercise 4 : Write an algorithm to print the frequency of odd and even numbers in a given 2d array.

```
var a = [ [ 2, 2, 7 ], [ 3, 4, 3, 3, 7, 9 ] ];
```

```
function dodo(a) {
    var even = [];
    var odd = [];
    for (var i = 0; i < a.length; i++) {
        for (var j = 0; j < a[i].length; j++) {
            if (a[i][j] % 2 === 0) {
                even.push(a[i][j]);
            } else {
                odd.push(a[i][j]);
            }
        }
    }
    console.log('Even numbers Frequency: ' + even.length);
    console.log('Odd numbers Frequency: ' + odd.length);
}
dodo(a);
//Output :
Even numbers Frequency: 3
Odd numbers Frequency: 6
```

Object:

Object: An object is a collection of data represented in property-value or key-value pairs.

- An empty object ("empty { }") can be created using one of two syntaxes:
  1. let bootcamp = new Object();
  2. let bootcamp = { }

```
let bootcamp = {
    name: 'The Hacking School',
    address: 'Hyderabad',
    year: 2015
};
/* Here the object name is bootcamp
The properties/keys are name, address, year
And the values are The Hacking School, Hyderabad,2015
*/
```

Accessing object properties

1. dot operator **(.)**
2. Square Brackets **[ ]**

Examples:

```
let bootcamp = {
    name: 'The Hacking School',
    address: 'Hyderabad',
    year: 2015
};
console.log(bootcamp.name); //The Hacking School
console.log(bootcamp['name']); //The Hacking School
console.log(bootcamp.address); //Hyderabad
```

```
console.log(bootcamp['address']); //Hyderabad
console.log(bootcamp.year); //2015
console.log(bootcamp['year']); //2015
```

Note: dot**(.)** operator cannot access if object property names are numerical values, instead [ ] Square brackets should be used to access them.

Example :

```
let obj = {
    1: 2,
    3: 4
};
console.log(obj.1); //Is Invalid
console.log(obj[1]); //Valid Syntax
```

Object Computed properties :

Square brackets[ ] are used to a property when creating an object which implies its predefined value . That's called **computed properties.**

```
let str = 'value';
let obj = {
      [str]: 11 // the name of the property is substituted from the variable str
};
console.log(obj.value); // Computes [str] and gives 11
```

Loops (for … in) to access object properties :

```
let bootcamp = {
    name: 'The Hacking School',
    address: 'Hyderabad',
    year: 2015
};
for (element in bootcamp) {
    console.log(element, bootcamp[element]);
}
//Output :
name The Hacking School
address Hyderabad
year 2015
```

Nested Objects :

Objects can be nested likewise:

```
let bootcamp = {
    name: 'The Hacking School',
    address: 'Hyderabad',
    year: 2015,
    obj: {
        stack: 'MERN',
        language: 'Javascript',
        database: {
            db1: 'MongoDB',
```

```
            db2: 'MySQL'
        }
    }
};
console.log(bootcamp);
console.log(bootcamp.obj.stack); //MERN
console.log(bootcamp.obj.database.db1); //MongoDB
```

Exercise: Check if the given object is empty or not?

```
var obj = {};
function objectEmptyCheck(obj) {
    for (let key in obj) {
        // if the loop has started, there is a property
        console.log(false);
        return;
    }
    console.log(true);
}
objectEmptyCheck(obj);
```

Function:

If you want to define a function, then it must use the **function** keyword.

- ➢ A JavaScript function is a block of code designed to perform a particular task which executes when "something" invokes it (calls it).
- ➢ A JavaScript function syntax 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: { }

Syntax :

```
function name(parameter1, parameter2, parameter3) {
    // code to be executed
}
```

Example :

```
function hai() {
    console.log('Hello World!');
}
hai();
//Output :
Hello World!
```

Function can be called multiple times without rewriting statements

```
function hai() {
    console.log('Hello World!');
}
for (let i = 1; i <= 5; i++) {
```

```
    hai();
}
//Output :
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

Note :

- ➢ Function **parameters** are listed inside the parentheses( ) in the function definition.
- ➢ Function **arguments** are the values received by the function when it is invoked.

Function Default Parameters :

Default function parameters allow named parameters to be initialized with default values if no value or **undefined** is passed.

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

console.log(multiply(5, 2));
// expected output: 10

console.log(multiply(5));
// expected output: 5
```

return keyword in function **:**

```
function sum(a, b) {
    return a + b;
}
let result = sum(1, 2);
console.log(result); // 3
```

Note:

- ➢ Don’t put any statements after return. Javascript programs won’t consider statements after return and the function will be terminated.

Function Expressions

```
var x = function(a, b) {
    return a * b;
};
```

Immediately Invoked Function Expression - IIFE

```
(function(a, b) {
    console.log(a * b);
})(2, 3);
```

Recursions and Understanding Call Stacks :

Recursion :

The process in which a function calls itself directly or indirectly is called **recursion**.

Direct Recursion :

```
"use strict";
function factorial
(num) {
    if (num <= 1) {
        return 1;
    } else {
        return num * factorial(num-1);
    }
}
console.log(factorial(5));
```

The above code can be written using an iterative method for a loop.

```
"use strict";
var result = 1;
var num = 5;
for (var i = 1; i <= num; i++) {
    result *= i;
}
console.log(result);
```

Indirect Recursion :

```
function abc(n) {
    def(n-1);
}
function def(n) {
    if (n >= 1) {
        console.log(n);
        abc(n);
    }
}
abc(10);
```

Exercise 1 :

Build recursive call stacks for the following recursive function code.

```
function fun(n) {

    if (n == 4) return n;

    else return 2 * fun(n + 1);

}

fun(2);
```

Solution :

Stack :

fun(4)	4	4
fun(3)	2*fun(4)	8
fun(2)	2*fun(3)	**16**

Result : **16**

Exercise 2 :

Build recursive call stacks for the following recursive function code.

```
function fun(x, y) {
    if (x == 0) return y;
    return fun(x - 1, x + y);
}
fun(4, 3);
```

Solution :

Stack :

fun(0,13)	13	
fun(1,12)	fun(0,13)	x=1,y=12
fun(2,10)	fun(1,12)	x=2,y=10
fun(3,7)	fun(2,10)	x=3,y=7
fun(4,3)	fun(3,7)	x=4,y=3

Result : **13**

Exercise 3 :

Build recursive call stacks for the following recursive function code.

```
function fun(n) {
    if (n == 0) return;
    console.log(n % 2);
    fun(n / 2);
}
fun(25);
```

Solution :

Stack :

fun(1)	1 ~~fun(0)~~	1 nothing
fun(3)	1 fun(1)	1 1
fun(6)	0 fun(3)	0 1 1

fun(12)	0 fun(6)	0 0 1 1
fun(25)	1 fun(12)	1 0 0 1 1

Result :

1

0

0

1

1

Exercise 4 :

What does the following recursive function do? What does fun() evaluate? Check options

```
function fun( x,  y){
        if (y == 0)    return 0;
        return (x + fun(x, y-1));
}
```

Check Options:

A)x+y B)x+x*y C)x*y D)x^y

Solution :

Stack :

assume x=1, y=1			
fun(1,0)	0	0	option A is eliminated, because (1+1) !=1
fun (1,1)	1+fun(1,0)	1	
assume x=2, y=1			
fun(2,0)	0	0	option B is eliminated, because 2+2*1 !=2
fun (2,1)	2+fun(2,0)	2	
assume x=1, y=2			
fun(1,0)	0	0	option D is eliminated because, 1**2 != 2
fun(1,1)	1+fun(1,0)	1	
fun (1,2)	1+fun(1,1)	2	
x*y =1*2 = 2			

Result : x*y =1*2 = **2**

option **C** is correct

Exercise 5 :

What does the following recursive function do? What does fun2() evaluate? Check options

Check Options.

```
function fun(x, y) {
    if (y == 0) return 0;
    return x + fun(x, y - 1);
}

function fun2(a, b) {
    if (b == 0) return 1;
    return fun(a, fun2(a, b - 1));
}
```

A)x+y B)x+x*y C)x*y D)x^y

Solution :

Stack :

As we know that fun (a,b) = a*b from the above question.

assume a=1,b=1

fun (1,1)	1	1
fun (1,fun2(1,0))	fun (1,1)	1
fun(a, fun2(a,b-1))	fun(1,fun2(1,0))	1

option A is eliminated, because (1+1) !=1

assume a=1,b=2

fun (1,1)	1	1
fun (1,fun2(1,1))	fun (1,1)	1
fun2(1,2)	fun(1,fun2(1,1))	1

Option B & C are eliminated as
B) 1+1*2 != 1
C) 1*2 != 1

assume a=3,b=2

fun2(3,0)	1	1
fun2(3,1))	fun(3,fun2(3,0)	3
fun2(3,2)	fun(3,fun2(3,1))	9

or

assume a=4,b=2

fun2(4,0)	1	1
fun2(4,1))	fun(4,fun2(4,0)	4
fun2(4,2)	fun(4,fun2(4,1))	16

Result :

D) 12 = 1 is correct**

Exercise 6 :

What does the following recursive algorithm do? Analyse outputs for multiple inputs and find out what the below algorithm is solution for?

```
function fun(n) {
    if (n == 0 || n == 1) return n;
```

```
    if (n % 3 != 0) return 0;
    return fun(n / 3);
}
```

Solution :

Stack :

n=2	fun(2)	0	0
n=3	fun(1)	1	1
	fun(3)	fun(1)	1
n=4	fun(4)	0	0
n=5	fun(5)	0	0
n=6	fun(2)	0	0
	fun(6)	fun(2)	0
n=9	fun(1)	1	1
	fun(3)	fun(1)	1
	fun(9)	fun(3)	1
It's seen that only **3** power series is '1'			

Result: **Its 3 power series function.**

Exercise 7 :

What does the following recursive algorithm do?

```
function f(n) {
    if (n <= 1) return 1;
    if (n % 2 == 0) return f(n / 2);
    return f(n / 2) + f(n / 2 + 1);
}
f(11);
```

Solution:
Stack :

f(6)	f(3)	2	
f(3)	f(1)+f(2)	2	
f(1)	1	1	
f(2)	~~f(1)~~	1	As f(1)=1 f(2)=1 f(3)=2
f(5)	~~f(2)~~+f(3)	3	
f(11)	f(5)+f(6)	5	

Result: **5**

Exercise 8 :

What does the following recursive algorithm do?

```
function foo(n, r) {
    if (n > 0) return n % r + foo(n / r, r);
    else return 0;
}
foo(513, 2);
```

Solution :
Stack:

foo(0,2)	1	1
foo(1,2)	1+foo(0,2)	1
foo(2,2)	0+foo(1,2)	1
foo(4,2)	0+foo(2,2)	1
foo(8,2)	0+foo(4,2)	1
foo(16,2)	0+foo(8,2)	1
foo(32,2)	0+foo(16,2)	1
foo(64,2)	0+foo(32,2)	1
foo(128,2)	0+foo(64,2)	1
foo(256,2)	0+foo(128,2)	1
foo(513, 2);	1+foo(256,2)	2

Result: **2**

Exercise 9 :

What does the following recursive algorithm do?

```
function robot(n, a, b) {
    if (n <= 0) return;
    robot(n - 1, a, b + n);
    console.log(n, a, b);
    robot(n - 1, b, a + n);
}
robot(2, 5, 2);
```

Solution:

Stack :

	robot(0,2,8) 1,2,7	
robot(1,2,7)	robot(0,7,3)	1,2,7

robot(1,5,4)	robot(0,5,5) 1,5,4 robot(0,4,6)	1,5,4
robot(2,5,2)	robot(1,5,4); 2,5,2 robot(1,2,7)	1,5,4 2,5,2 1,2,7

Result :

1,5,4
2,5,2
1,2,7

Exercise 10 :

What does the following recursive algorithm do?

```
function f(n) {
    var i = 1;
    if (n >= 5) return n;
    n = n + i;
    i++;
    return f(n);
}
f(1);
```

Solution :

Stack :

f(4)	f(5)	5
f(3)	f(4)	5
f(2)	f(3)	5
f(1)	f(2)	5

Result: **5**

Exercise 11 :

What does the following recursive algorithm do?

```
function ths(n) {
    if (n < 1) return;
    ths(n - 1);
    ths(n - 3);
    console.log(n);
}
ths(8);
```

Solution :

Stack :

ths(1)	~~ths(0)~~ ~~ths(-2)~~ 1	 1
ths(2)	ths(1) ~~ths(-1)~~ 2	 1 2
ths(3)	ths(2) ~~ths(0)~~ 3	1 2 3
ths(4)	 ths(3) ths(1) 4	1 2 3 1 4
ths(5)	 ths(4) ths(2) 5	1 2 3 1 4 1 2 5
ths(6)	 ths(5) ths(3) 6	1 2 3 1 4 1 2 5 1 2 3 6
ths(7)	 ths(6) ths(4) 7	1 2 3 1 4 1 2 5 1 2 3 6 1 2

		3 1 4 7
ths(8)	ths(7) ths(5) 8	1 2 3 1 4 1 2 5 1 2 3 6 1 2 3 1 4 7 1 2 3 1 4 1 2 5 8

Result :

1
2
3
1
4
1
2
5
1
2
3
6
1
2
3
1
4

7
1
2
3
1
4
1
2
5
8

Exercise 12 :

What does the following recursive algorithm do?

```
function count(n) {
    var d = 1;
    console.log(n);
    console.log(d);
    d++;
    if (n > 1) count(n - 1);
    console.log(d);
}
count(3);
```

Solution:

Stack:

count(1)	1 1 2	1 1 2
count(2)	2 1 count(1) 2	2 1 1 1 2 2
count(3)	3 1 count(2) 2	3 1 2 1 1 1 2 2 2

Result :

3

1
2
1
1
1
2
2
2

Exercise 13 :

What does the following recursive algorithm do?

```
function f(n) {
    if (n <= 1) {
        console.log(n);
    } else {
        f(n / 2);
        console.log(n % 2);
    }
}
f(1024);
```

Solution :
Stack :

f(1)	1	1
f(2)	f(1) 0	1 0
f(4)	f(2) 0	1 0 0
f(8)	f(4) 0	1 0 0 0
f(16)	f(8) 0	1 0 0 0 0
f(32)	f(16) 0	1 0 0 0 0 0

f(64)	f(32) 0	1 0 0 0 0 0 0 0
f(128)	f(64) 0	1 0 0 0 0 0 0 0 0
f(256)	f(128) 0	1 0 0 0 0 0 0 0 0 0
f(512)	f(256) 0	1 0 0 0 0 0 0 0 0 0 0
f(1024)	f(512) 0	1 0 0 0 0 0 0 0 0 0 0 0

Result :

1
0

0
0
0
0
0
0
0
0
0

Exercise 14:

What does the following recursive algorithm do?

```
function f(n) {
    if (n / 2) {
        f(n / 2);
    }
    console.log(n % 2);
}
f(1024);
```

Solution:
Stack:

f(1)	1	1
f(2)	f(1) 0	1 0
f(4)	f(2) 0	1 0 0
f(8)	f(4) 0	1 0 0 0
f(16)	f(8) 0	1 0 0 0 0
f(32)	f(16) 0	1 0 0 0 0 0

f(64)	f(32) 0	1 0 0 0 0 0 0 0
f(128)	f(64) 0	1 0 0 0 0 0 0 0 0
f(256)	f(128) 0	1 0 0 0 0 0 0 0 0 0
f(512)	f(256) 0	1 0 0 0 0 0 0 0 0 0 0
f(1024)	f(512) 0	1 0 0 0 0 0 0 0 0 0 0 0

Result :

1

0

0
0
0
0
0
0
0
0
0

Note : Take your instructors help in understanding the above recursion call stacks using Chrome Debugging Tools graphically. Solve each and every stack to know the underhood of javascript call stack and recursion interpretations.

Type Coercions :

- Type coercion means, when the operands of an operator are different types, one of them will be converted to an "equivalent" value of the other operand's type.
- When Javascript was developed, it was meant to be for web programming. When there is any syntax or type error, JS is designed to produce something rather than an error so that websites don't break. Coercion is all about that.

If you want to understand more - Try this example in your Node js shell directly:

```
{} + [] + {} + [ 11 ]
//Output will be : '0[object Object]11'
```

Try another example in your Node js shell directly :

```
console.log(null + {} + true + []+ [5]);
//Output will be
null[object Object]true5
```

Surprising ? Final output is coming as a string. That's called '**Coercion**'

Lets try a few more mini examples and will try to understand the above two statements of coercion logic.

Example 1:

```
console.log(true + 1);
// 2 (boolean and number, producing number,so 1+1=2)
console.log(true+true);
//2 (boolean true considered as 1, so 1+1 = 2)
console.log(false + 0);
// 0 (boolean and number, producing number, so 0+0=0)
console.log(true + false);
//1 (boolean false considered as 0, so 1+0 = 1)
```

Example 2:

```
console.log('The' + ' ' + 'Hacking' + ' ' + 'School');
//The Hacking School
//Adding strings with +  will concatenate
```

Example 3:

```
console.log('TheHackingSchool'+2020); //TheHackingSchool2020
//A String + A number = A String
console.log(2020+'TheHackingSchool'); //2020TheHackingSchool
//A Number + A String = A String
console.log('11' + '22'); //1122
//A Number inside a string + A number inside a string = A String
(Concatenates)
console.log("100"-2); //98
// A Number inside a string - A number = A Number
```

Example 4:

```
console.log('100' * '2'); //200 converted to a number
console.log('100' + null); // 100null
//A number inside string and null will be a string
console.log(100+null); //100 (null be taken as 0)
```

Example 5:

```
console.log(1/"THS"); //NaN
//NaN - Not A Number
console.log(NaN === NaN); //false
```

Example 6:

```
console.log([1] + [2]); //12
console.log([1,2,3]+ [4,5,6]);//1,2,34,5,6
```

Example 7:

```
console.log(1/Infinity); //0
console.log(Infinity/Infinity); //NaN
console.log(-Infinity/0); //-Infinity
console.log(Infinity/0); //Infinity
```

Example 8:

```
console.log([]+{}); //[object Object]
console.log(-~!+"hello" << '1' >> '1')
```

Example 9:

```
console.log(Boolean(true)); //true
console.log(Boolean([])); //true
console.log(Boolean({}));//true
console.log(Boolean(function() {})); //true
console.log(Boolean(Infinity));//true
console.log(Boolean(false)); //false
console.log(Boolean(NaN)); //false
console.log(Boolean(null)); //false
console.log(Boolean(undefined)); //false
console.log(Boolean('')); //false
console.log(Boolean(0)); //false
console.log(Boolean(-0)); //false
```

Example 10:

```
var str = 'The Hacking School';
console.log(-str); //NaN
```

Javascript Scope and Hoisting

Scope

Scope is simply the area enclosed by { } brackets.

There are 3 unique scope types:
1.The global scope,
2.block scope and
3.function scope

variable definitions - case sensitivity.
variables are case sensitive.

```
let a = 1;
let A = "hello";
```

Variable Definitions

Variables can be defined using var, let or const keywords. Of course, if you tried to refer to a variable that wasn't defined anywhere, you would generate a ReferenceError error "variable name is not defined":

Prior to let and const the traditional model allowed only var definitions:

```
var apple = 1;
{
    console.log(apple);
}
```

- Here apple is defined in global scope.
- But it can also be accessed from an inner block-scope. Anything (even a function definition) defined in global scope becomes available anywhere in your program. The value propagates into all inner scopes. When a variable is defined in global scope using var keyword, it also automatically becomes available as a property on window object.

Hoisting

If apple was defined using var keyword inside a block-scope, it would be hoisted back to global scope! Hoisting simply means "raised" or "placed on top of".

Note : Hoisting is limited to variables defined using var keyword and function name defined using function keyword.

- Variables defined using let and const are not hoisted and their use remains limited only to the scope in which they were defined.
- As an exception, variables defined var keyword inside function-level scope are not // hoisted. Commonly, when we talk about hoisting block-scope is implied.
- Likewise, variables defined in global scope will propagate to pretty much every other scope defined in global context, including block-level scope, for-loop scope, function-level scope, and event callback functions created using setTimeout, setInterval or addEventListener functions.

```
console.log(apple);
{
    var apple = 1;
}
```

- Variable apple is hoisted to global scope. But the value of the hoisted variable is now undefined – not 1. Only its name definition was hoisted.
- Hoisting is like a safety feature. You should not rely on it when writing code. You may not retain the value of a hoisted variable in global scope, but you will still save your program from generating an error and halting execution flow.

Function Name Hoisting:

Hoisting also applies to function names. But variable hoisting always takes precedence.

```
fun();
function fun(){
    console.log("Hello from fun() function.");
};
```

Note that the function was defined after it was called. This is legal in JavaScript.

```
function fun(){
    console.log("Hello from fun() function 1.");
}

//now same as
var fun = function(){
    console.log("Hello from fun() function 2.");
}
```

- It is possible to assign an anonymous function expression to a variable name.

- It's important to note, however, that anonymous functions that were assigned to variable names are not hoisted unlike named functions.
- This valid JavaScript code will not produce a function redefinition error. The function will be simply overwritten by second definition.

Having said this, what do you think will happen if we call fun() at this point?

```
// Now check with following code

function fun(){
    console.log("Hello from fun() function 1.");
}

function fun(){
    console.log("Hello from fun() function 2.");
}
```

However, this is still perfectly valid code – no error is generated. Whenever you // have two functions defined using a function keyword and they happen to share the // same name, the function that was defined last will take precedence.

```
var fun = function(){
    console.log("Hello from fun() function 1.");
}

function fun(){
    console.log("Hello from fun() function 2.");
}
```

And now let's call fun() to see what happens in this case:

Bingo : variable name will take precedence over function definitions even if it was defined prior to the second function definition with the same name:

Now Lets see the order in which JavaScript hoists variables and functions. Functions are hoisted first. Then variables.

```
fun();
var fun = function(){
    console.log("Hello from fun() function 1.");
```

```
}

function fun(){
    console.log("Hello from fun() function 2.");
}
```

Now Call fun() again.

DEFINING VARIABLES INSIDE FUNCTION SCOPE

At this point you might want to know that variables defined inside a function will be limited only to the scope of that function. Trying to access them outside of the function will result in a reference error:

```
function fun(){
    var apple = 1;
}
console.log(apple);
```

Rule 1 : Do not use var unless for some reason you want to hoist the variable name.
Rule 2 : Do use let and const instead of var, wherever possible
Rule 3 : Do use const to define constants such as PI, speed of light, tax rate, etc. – values that you know shouldn't change during the lifetime of your application.

Chrome Debugging Tips

https://developers.google.com/web/tools/chrome-devtools/javascript

https://youtu.be/H0XScE08hy8

Problem Solving and Algorithms with JavaScript

1) Write a program whether given input Check whether Even or odd?

```
var x;
function odev(x){
    if( x % 2 == 0){
        console.log('Its an Even number');
```

```
    }
    else
    console.log('Its an Odd number');
} odev(20);
```

2) Write a program to Generate Even and Odd Numbers less than N and Generate 'N' Even and Odd Numbers.

```
var x;
var even=[];
var odd=[];
function nodev(x){
for(i=0;i<x;i++){
        if(i%2==0){
             even.push(i);
                }
        else{
            odd.push(i);
            }
}
console.log('Even numbers are: ' + even);
console.log('Odd numbers are: '+ odd);
} nodev(20)
```

3) Write a program to decide given N is Prime or not?

```
var x,y,prime,cnt=0;
function prime(x){
   if (x>2) {
        for(i=1;i<=x;i++) {
      if(x%i==0) {
             cnt++;
         }
    }
  }
  if(cnt==2){
        console.log('Its a prime number');
    }
  else{
        console.log("Its not a prime number");
    }
}
prime(32);
```

BONUS QUESTION:
Print a new array of prime numbers from the given array elements.

```
var arr = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15];
var newarr=[];
```

```
   for(var i=0;i<arr.length;i++){
       var cnt=0;
       for(var j=1;j<=arr[i];j++){
           if(arr[i]%j===0){
             cnt++;
           }
         }
         if(cnt===2){
                 newarr.push(arr[i]);
         }

   }console.log(newarr);
```

4) Write a program to subtract two integers without using the Minus (-) operator ?

```
var a,b;
var cnt=0;
function sub(a,b){
    if(a==b){
        console.log(0);
    }
if(a<b){
    for(i=a;i<b;i++){
        cnt+=1;
    }
    console.log('-'+cnt);
    }
if(a>b) {
    for(i=b;i<a;i++){
        cnt+=1;
    }
    console.log('+'+cnt);
    }
}
sub(40,20);


//Best solution:
/*
var a=40,b=20;
console.log(a+(~b)+1);
*/
```

5) Write a program to find the remainder of two numbers without using a modulus (%) operator?

```
var a,b,c;
function rem(a,b){
  c=Math.floor(a/b);
console.log(a-(b*c));
}rem(20,4)
```

6) Write a program to generate Prime Numbers less than N and Generate 'N' Prime Numbers in a given range.

```
//Program to generate prime Numbers less than a value :
function primelessthan(arr){
    let newarr=[];
    let cnt;
   for(let i=1;i<=arr;i++){
        cnt=0;
        for(let j=1;j<=arr;j++){
          if(i%j===0){
             cnt++;
         }
         }
         if(cnt===2){
                newarr.push(i);
         }
   }
         console.log(newarr);
   }
   primelessthan(20);
```

```
//Program to print the prime values within a range(start,end) :
function primelessthan(start,end){
    let newarr=[];
    let cnt;
   for(let i=start;i<=end;i++){
        cnt=0;
        for(let j=1;j<=end;j++){
          if(i%j===0){
             cnt++;
         }
         }
         if(cnt===2){
                newarr.push(i);
         }
   }
         console.log(newarr);
   }
   primelessthan(1,50)
```

7) Write a program that prints the numbers from 1 to 100 and for multiples of '3' print "Fizz" instead of the number and for the multiples of '5' print "Buzz".

```
for(i=1;i<=100;i++){
    var arr=[];
    if((i%3===0) && (i%5===0)){
        console.log('fizz buzz');
    }
   else if(i%3===0){
```

```
        console.log('fizz');
    }
   else if(i%5===0){
        console.log('buzz');
    }
    else{
        console.log(i);
    }
    arr.push(i);
}
```

8) Write a program to find the Sum of Array Elements.

```
var arr=[1,2,3,4,5];
var sum=0;
function summ(){
    for(i=0;i<arr.length;i++){
        sum+=arr[i];
    }
    console.log(sum);
} summ(arr);
```

Note : The above algorithms are basic and approach is bruteforce to solve the problem. Discuss if you have any other approach to solve the problems with your instructor. In our upcoming sessions, we will go through Algorithm Analysis concepts with Asymptotic Notations like Big O to improve the logic and write optimised code.