Date / /
Page No.

* JAVASCRIPT :-

→ Javascript is scripting and programming language.

→ It is purely object based language. This means that variables, functions, and even primitive data types like numbers and strings are object, everything is object in Javascript.

→ It is dynamically typed language, it means type of value stored in memory block is checked at runtime because of this nature we can store any type of value in variable.

→ It is object oriented programming language, it means we can create our own object. (It is not purely object oriented programming language.

→ It is interepreted language.

→ Js helps to provide behavior and functionality to webpage and helps to develop dynamic webpage.

→ Mainly introduced to instruct the browser.

* HISTORY OF JAVASCRIPT :-

→ Javascript was first created by 'Brendan Eich' in just 10 days in 'May 1995' while he was working at 'Netscape Communication Corporation'.

→ The initial release was called 'Mocha' and later renamed to 'Live Script', and finally 'JavaScript'.

* JAVASCRIPT RUNTIME ENVIRONMENT [JRE] :-

→ JRE provides the environment where we can run our Javascript code.

→ There are two type of JRE :-

(i) Browser,

(ii) Node.JS.

* **BROWSER:-**

→ A browser is a software application that is used to access and view information on the world wide web.

→ The browser acts as a JavaScript runtime environment because it includes a javascript engine that interprets and executes JavaScript code.

* **NODE.JS:-**

→ Node.JS is a software application that executes JavaScript code. It is not a framework or a library.

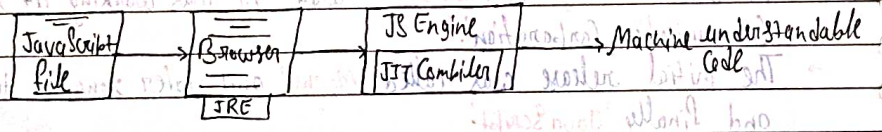
→ It allows developers to run JavaScript code outside of a browser, such as on a server or command line interface.

→ Node.JS uses the V8 JavaScript engine, which also used in Google Chrome.

* **JAVASCRIPT ENGINE (JS ENGINE):-**

→ A JavaScript engine is a computer program that executes JavaScript code.

→ It is a program which reads javascript code line by line and convert it into machine understandable code.



Browser	JSEngine
Chrome	V8 Engine
FireFox	Spider Monkey
Microsoft (IE, edge)	Chakra
Safari	JavaScript core

* **TOKEN:-**

→ It is the smallest unit of Programming Language.

→ We have five types of:-

(i) Keywords:- for, loop, do-while, if, else, etc. These are reserved words.

(ii) Identifiers:- These are user given names to variable, functions and other objects in the code.

→ Identifier name can not start with number.

→ Identifier name should not be a keyword.

→ If identifier is of multiple word, instead of using space we have to use underscore (_).

→ Identifier name should not have special character but can start with underscore (_) and dollar ($).

(iii) Literals:- These are values used in our program like number (2), string ('Hello Shiv'), etc.

Types of Literals / Datatypes:-

(a) Primitive

(b) Non-Primitive

(a) Primitive Datatypes:- In JavaScript, a primitive data-type is a data type that represents single value.

→ It is immutable values.

→ There are 8 primitive types of literals:- number, bigint, boolean, nan, undefined, null, symbol, string.

Number:- This data type represents a numeric value. It can store both integers and floating-point values.

→ It's range is from '$-2^{53}-1$' to '$2^{53}-1$'.

Bigint:- It is used to represent integers that are larger than the number.

→ It's range is more than '$-2^{53}-1$' and more than '$2^{53}-1$'.

To specify the given integer is bigint we have to suffix 'n' after the integer.

Boolean:-
→ This datatype represents a logical entity and can only have two values:- 'true' or 'false'.

Null:- This datatype represents a null or empty value.
→ It is used to mark the memory block empty intentionally.

Undefined:- This datatype represents an uninitialized value.
→ When memory block is uninitialized, JS engine implicitly initialize that memory block with 'undefined' in variable phase.

* Note:-
→ For variable declared with 'var' it will initialize it in variable phase.
→ For variable declared with 'let' or 'const' it will not initilize it in variable phase.

NAN:-
→ It stands for 'not a number'.
→ It represents computational error.
→ When JS engine is not able to compute result it return 'NaN'.

ex:-
'Hello' + 1 => Hello1 or 'Hello' - 1 => NaN

Symbol:-
→ It represents unique identifier.

String:-
→ It represents collection of characters.
→ We have two types of string:-
(i) Single Line String
(ii) Multi Line String

(i) Singal Line String:-
→ It enclosed with single quotes (' ') and double quotes (" ").
→ It does not allow linebreak and white space.

(ii) Multi Line String:-
→ It is enclosed by backticks (` `).
→ It does allow line break and white space.
→ It is also called as template string.
→ Template strings allows us to insert variables and expression directly in the string using `${variable_Name}` notation.

* PRIMITIVE LITERALS:-

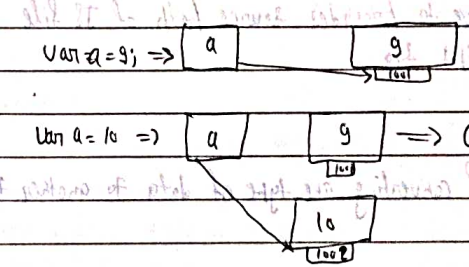


fig:- Immutable

* NON-PRIMITIVE LITERALS:-
→ It represents multi value datatype.
→ It is mutable means that their value can be changed.
(i) Object,
(ii) Array, etc.

ex:-

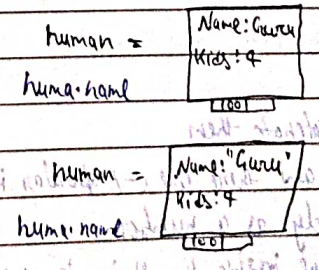


fig: mutable

* Ways to embed JS Code :-
→ There are two ways

(i) Internal JS :-
⇒ The Javascript code are written inside HTML document, is refered as internal JS.
→ To insert JS code, we have <script> --- JS code --- </script>
→ Script tag should be inserted at last of <body> tag.

(ii) External JS :-
→ Here we create separate file to write JS code and file should be saved with '.JS' extension.
→ To link JS file, we have to provides source path of JS file using 'src' attribute of script tag.

Type Coersion :-
→ The process of converting one type of data to another type of data.
→ Two type of coersion :-
(i) Implicit :- By JS Engine
(ii) Explicit :- By user

* Steps of Implicit Type Coersion :-
"A" + 1 ⇒ A1
"A" - 1 ⇒ NaN
"111" + 1 ⇒ 1111
"11" - 1 ⇒ 10
(i) It first check the datatype.
(ii) If (+) operator is in use it concatenate them
(iii) If a string have digits value and there are (-) operator is in use it type change the datatype implicitely as a number.
(iv) If datatype is string and the value inside the string is string it returns NaN.

* Steps of Explicite Type Coersion :-
→ We can use the datatype which we want to type cast in.
ex:- "11" + 1 ⇒ 111
Number("11") + 1 = 12
Number("11") - 1 = 10

Syntax:- Datatype(value);
Boolean(1) ⇒ true
String(1) ⇒ 1 (String)
Number("11") ⇒ 11 (number)
Number("A") ⇒ NaN

* Scope :-
→ Scope defines the visibility of or accessibility of a variable (state and behaviour)
→ There are 3 scope in Javascript :-
(a) Global Scope.
(b) Local / block scope / function scope.
(c) Script Scope.

* Global Scope :-
→ The variable declared in global scope can be accessed anywhere in the program.
→ Global scope has the highest accessibility.
→ Variable declared with var goes in global scope.
eg:-

```
var a = 75;
console.log(a+20);
```

Date
Page No.

* Local Scope:-

→ Local / block / function scope.

→ The variable declared in block scope can be access in that block only ie visibility of member is within the block only.

ex:-

```
console.log("Local Variable");
{
    var x = 20;
    console.log(x);
}
```

* Script Scope:-

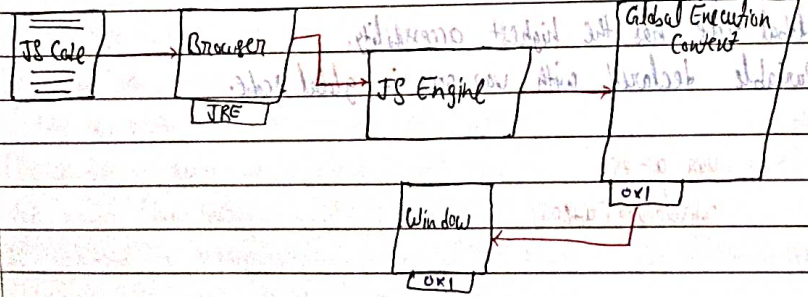
→ Variable declared with let and const goes to (or store in) separate scope called script scope.

ex:-

```
let a = 30;
const z = 40;
console.log(a+z);
```

* Global Execution Context:-

→ When we give JS code to the browser, JS Engine will allocate (create) a global memory block for execution of JavaScript code called 'Global Execution Context'.



Date
Page No.

→ Here, we have a window variable which have reference of Global Execution Context.

* Window Variable:-

→ (Window Object) everything is object in JS

→ Window is a global variable which stores the reference of Global Execution Context.

→ Window object is also known as 'Global Object' because it is available everywhere in the program.

→ Window object have pre-defined state and behaviour. Variable declared with 'var' always goes to global scope and can be accesible by window object.

→ Any variable created in global scope will be added in window object implicitly by JS Engine.

* JavaScript Code Run in Two Phase:-

1→ Variable phase

2→ Execution phase

* Variable Phase:-

→ In variable phase, JS Engine will check the complete JS code and it will search for variable declaration statement.

→ If variable is declared then JS Engine allocate (provide) memory for them.

→ Variable is declared with 'var' will be initialized storing "undefined" at the time of memory block creation.

→ Variable declared with 'let' and 'const' will remain "uninitialized (empty)" at the time of memory block creation.

* Execution Phase:-

→ In execution phase, JS Engine will execute the instruction line-by-line.

Date / / Page No.

Example:-

```
Var a = 10;
Console.log(a);
```

Variable Phase	Execution Phase
① a	② a = 10
	③ Console.log(a);

Output:- 10

* Type of Variable Declaration:-

→ There are 3 way to declare variables in Javascript:-

(i) VAR;
(ii) LET;
(iii) CONST

* VAR (awara):-

→ Variable declared with var goes to global scope.
→ We can declare multiple variable with same name in same scope and the most recently created variable will be used.
→ The value of variable can be modified.

ex:-

```
Var a = 10;
Var b = 80;
Var a = 40;
Console.log(b);
Console.log(a);
```

Output:- 80
40

→ With var, we can declare variable without initialization and JS engine will store "undefined", the moment memory block is created (in Variable Phase).

ex:-

```
Var a;
Console.log(a);
```

output:-

V.P.	E.P.
① a	③ console.log(a)
② a undefined	

→ The variable declared with using var belongs to global scope, so we can access using window variable.
→ The variable declared using var is 'hoisted' does not belong to 'temporal dead zone', can be used before initialization (because memory block created with var, will have undefined stored in it - (in variable phase).
→ Declaring variable using var inside function will not goes global scope. It will be visible inside function only, not out side function.

# Multiple declaration, var a = 10;
# Reinitialization, var b = 20;
# Can be Hoisted. { var a = 30;
Console.log(b);
Console.log(a);
Console.log(z);
var z; }

V.P.	E.P.
① a	② a = 10;
③ a 10	③ b = 20;
④ b	
⑤ b 20	

V.P.	E.P.
② a	④ a = 30
③ a 30	⑤ console.log(b)
⑥ z undefined	⑥ console.log(a)
	⑦ console.log(z)

Output:- 20
30
undefined

* LET (bagal):-

→ Variable declared with let is 'block scope' (script scope).
→ We can not declared with multiple variable with the same name within a same scope.
→ The value of variable can be 'modified'.
→ We can declare variable using 'let' without initialization, But JS engine will keep that memory block 'uninitialized (empty)' until JS engine reads that line, after reading JS engine will store in it (execution phase)

```
let a = 28;
a = 60;
Console.log(a);
```

V.P.	E.P.
a Uninitialized	a = 28;
a 28	a = 60;
a 60	Console.log(a);

Output:- 60

→ Because 'let' variable is a 'uninitialized (empty)' in variable phase, it belongs to 'Temporal Dead Zone'.
→ The variable declared using let does not belong to global scope, we can not use them with the help of window variable.
→ The variable declared using 'let' is 'hoisted' and belong to temporal deadzone. Therefore, it can not be used before initialization (because at that moment it is uninitialized - TDZ).

Multible initialization,
One time declaration,
Can be Hoisted,
Temporal Dead Zone.

```
let a = 10;
console.log(a);
a = 20;
console.log(a);
console.log(z);
let z;
```

V.P.	E.P.
① a uninitialize	① a = 10;
③ a 10	② console.log(a);
⑤ a 20	④ console.log(a);
⑥ z uninitialize	⑤ console.log(z);
	⑥ let z;

Output:- 10
20
uninitialize

* CONST (Decorum):-
→ Variable declared with 'const' is a 'block scope' (Script scope).
→ We can not declare multiple variable with same name within a same scope.
→ The value of variable can not be modified.
→ We can not declare const without initialization.
→ The variable declared with using const does not blong to global object, we can not use them the help of 'window'.
→ The variable declared using const is 'hoisted' and belong to 'temporal dead zone'. Therefore, it can not be used before initialization (because at that moment it is uninitialized - TDZ).

One time declaration,
One time Initialization,
Can be Hoisted,
Temporal Dead Zone

```
const a = 28;
console.log(a);
console.log(z);
a = 20;
const a = 30;
```

V.P.	E.P.
a uninitialize	a = 28;
a 28	console.log(a);
z uninitialize	console.log(z);
	a = 20;
	a = 30;

Output:- 28
Error multible declaration/initialization

** HOISTING:-
→ The ability of javascript engine to access a variable before it's declaration statement, it is called 'Hoisting'.
→ Variables can be hoisted:-
(i) var,
(ii) let,
(iii) const.

```
var a = 10;
console.log(b);
var b = 20;
console.log(b);
console.log(c);
console.log(d);
let c = 20;
const d = 30;
```

V.P.	E.P.
① a undefined	① a = 10;
③ a 10	② console.log(b);
⑤ b undefined	④ b = 20;
⑦ b 20	⑤ console.log(b);
⑧ c uninitialized	⑥ console.log(c);
⑨ d uninitialized	⑪ c = 20;
⑫ c 20	⑫ d = 30;
⑬ d 30	

Output:- 20
20
20

** TEMPORAL DEAD ZONE:-
→ The timeframe between variable deleration and initialization, in this timeframe we can not access the variable because it is empty so JS engine gives us error (because we are trying to access memory block which is empty), (we are trying to access value of memory block), this timeframe is refered as 'Temporal Dead Zone'.
→ Variable will come out of the temporal dead zone when we initialize it.
→ Variable declared with 'let' and 'const' goes to the temporal dead zone.

```
console.log(a);
var a = 5;
console.log(b);
let b;
console.log(c);
const c = 10;
```

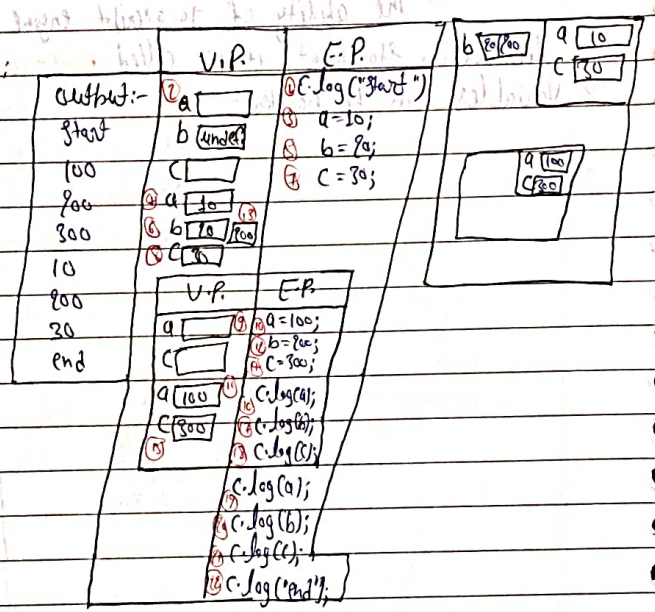
V.P.	E.P.
a undefined	console.log(a);
b ~~undefined~~ uninitialized	console.log(b);
uninitialized	let b;
c 10	console.log(c);
	const c = 10;

Output:- undefined
error
10

Home Work

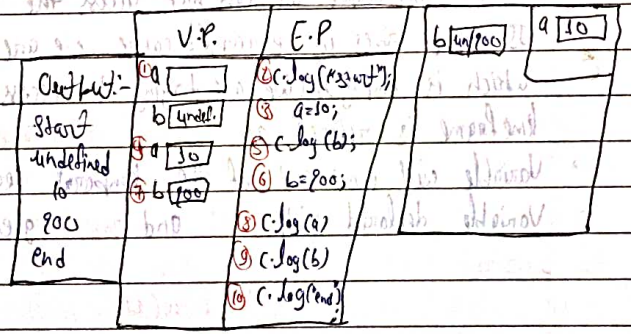
1→

```
console.log("start");
let a = 10;
var b = 20;
const c = 30;
{
    let a = 100;
    var b = 200;
    const c = 300;
    console.log(a);
    console.log(b);
    console.log(c);
}
console.log(a);
console.log(b);
console.log(c);
console.log("end");
```



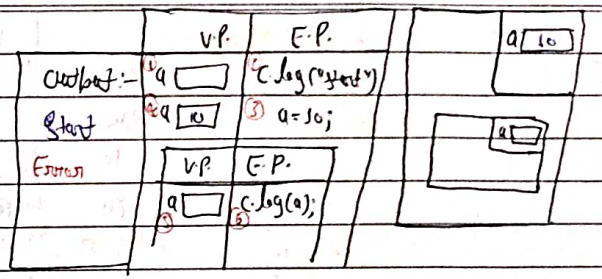
2→

```
console.log("start");
let a = 10;
console.log(b);
{
    var b = 200;
}
console.log(a);
console.log(b);
console.log("end");
```



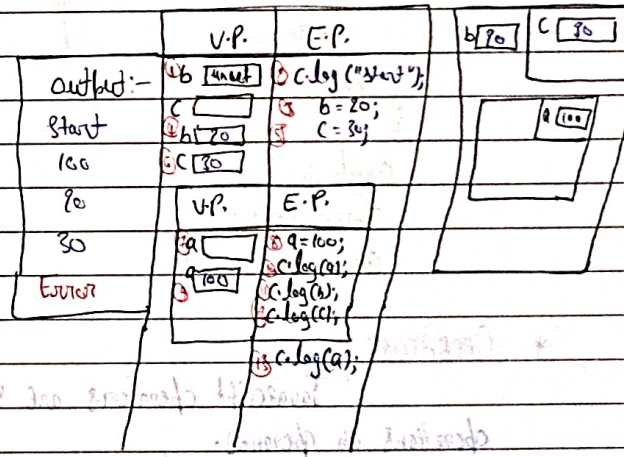
3→

```
console.log("start");
let a = 10;
{
    console.log(a);
    let a = 10;
}
console.log(b);
console.log(b);
console.log("end");
```



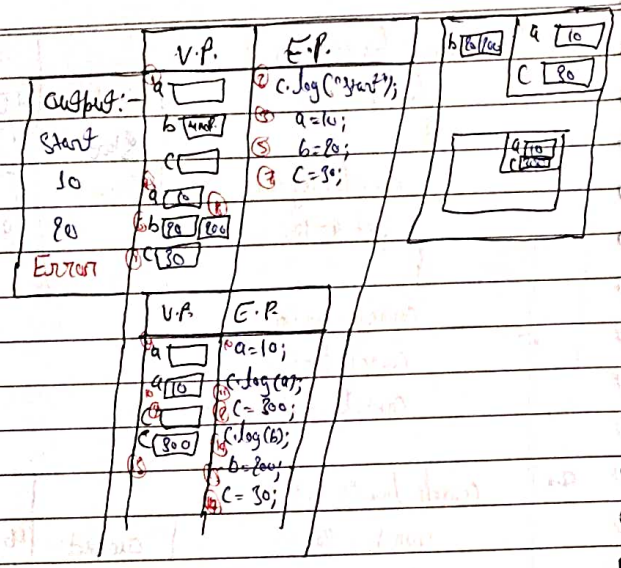
4→

```
console.log("start");
var b = 20;
const c = 30;
{
    let a = 100;
    console.log(a);
    console.log(b);
    console.log(c);
}
console.log(a);
console.log(b);
console.log("end");
```



5→

```
console.log("start");
let a = 10;
var b = 20;
const C = 30;
{
  let a=10;
  console.log(a);
  const C = 300;
  console.log(b);
  b = 200;
  C = 30;
  console.log(b);
}
console.log(a);
console.log(b);
console.log("end");
```



* OPERATORS :-

→ Javascript operators are symbols that are used to perform operations on operands.

→ There are following types of operators in Javascript :-

1→ Arithmatic Operators,
2→ Comparison Operator,
3→ Bitwise Operator,
4→ Logical Operator,
5→ Assignment Operator,
6→ Special Operator.

* ARITHMATIC OPERATORS :-

Arithmatic Operators are used to perform arithmetic operations on the operands.

→ The following operators are known as arithmatic operator :-

Operator	Description	Example
+	Addition	10 + 20 = 30
−	Substraction	20 − 10 = 10
*	Multiplication	10 * 20 = 200
/	Division	20/10 = 2
%	Modulus (Remainder)	20 % 10 = 0
++	Incurement	var a = 10; a++; Now a = 11
−−	Decrement	var a=10 ; a−−; Now a = 9

* COMPARISON OPERATORS :-

→

Operator	Description	Example
==	Equality Operator	"1" == 1 ⇒ true
===	Strictly Equality	"1" === 1 ⇒ false
!=	Not equal	10 != 20 ⇒ true
!==	Not identical	20 !== 20 ⇒ false
>	Greater than	20 > 10 ⇒ true
>=	Greater than equal	20 >= 10 ⇒ true
<	Less than	20 < 10 = false
<=	Less than equal	20 <= 10 ⇒ false

* BITWISE OPERATORS :-

→

Operator	Description	Example
&	Bitwise And	(10 == 20 & 20 == 33) = false

Operator	Description	Example
|	Bitwise OR	(10==20 | 20==33) = false
^	Bitwise XOR	(10==20 ^ 20==33) = false
~	Bitwise NOT	(~10) = -10
<<	Bitwise Left Shift	(10<<2) = 40
>>	Bitwise Right Shift	(10>>2) = 2
>>>	Bitwise Right zero	(10>>>2) = 2

* Logical Operators :-

Operator	Description	Example
&&	Logical AND	(10==20 && 20==30) false
||	Logical OR	(10==10 || 20==30) true
!	Logical NOT	!(10==20) = true

* Special Operators :-

→ (exp1 ? exp2 : exp3) :- 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 instances (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..

* Conditional Statement :-

→ There are 5 conditional statements of Javascript :-

→ If Statement,
→ Else-If Statement,
→ Switch,
→ for Loop,
→ While Loop,
→ Do-While Loop.

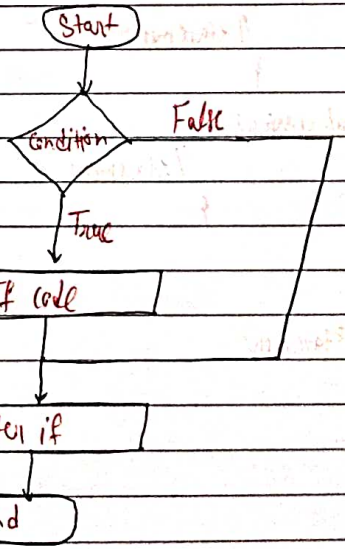
* If Statement :-

→ It evaluates the content only if expression is true.

Syntax :-

```
if (expression){
    // if block statement
}
```

Flow chart :-



* **IF-ELSE STATEMENT:-**

→ It evaluates whether condition true or false

Syntax:-

```
if(expression){
        // if block statement
        }
else{
    // else block statement
    }
// out of else-if block code
```

* **IF-ELSE-IF STATEMENT:-**

→ It evaluates the content only if expression true from several condition. Otherwise else block code execute.

Syntax:-

```
if(expression1){
        // statement
        }
else if(expression2){
        // statement
        }
  : : :
else{
    // statement
    }
```

* **SWITCH STATEMENT:-**

→ The Javascript switch statement is used to execute one code from multible expressions. It is just like else-if statement.

→ It can be used with numbers, characters, etc.

Syntax:-

```
switch(expression){
        case value1: // statement break;
        case value2: // statement break;
        : : : :
        default: // statement
        }
```

Break:- It stops the flow of execution and return to the main scope.

* **LOOPS:-**

* **FOR-LOOP:-**

→ The Javascript for loop iterates the elements for the fixed number of times.

→ It should be used if number of iteration is known.

Syntax:-

```
for(initialization; condition; increment)
{
    // code to be executed
}
```

* **WHILE LOOP:-**

→ The Javascript while loop iterates the elements for the infinite number of times. It should be known of iteration is when unknown then it will prefer for use.

* **Do-While Loop:-**

→ The javascript do-while loop iterates the elements for the infinite number of times like while loop. But, code is executed at least once whether condition is true or false.

Syntax:-

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

* **Some Program based on loop or conditional statement:-**

→ Print number 1 to 100.

Program:-

```
for (var i=1; i<=100; i++)
{
    console.log(i);
}
```

→ Print even number between 1 to 100.

Program:-

```
for (var i=1; i<=100; i++)
{
    if (i%2===0){
        console.log (i is even no.);
    }
}
```

→ Find number of digits in a number.

Program:-

```
var a=100;
console.log (Math.floor(Math.log10(a)+1));
```

→ Print Prime number.

Program:-

```
let count=0, num=5;
for (let i=1; i<=num; i++)
{ if (num%i===0){
        count++;} }
if (count===2)
{
    console.log ("Prime number");
}
```

* **MESSAGE BOXES:-**

→ Javascript provides builtin global functions to display pobup message boxes for different purposes:-

→ Alert (message),

→ Confirm (message),

→ Prompt (message)

* **ALERT (message):-** Display a pop-up box with the specified message with the OK button.

→ In Javascript, global functions can be accessed using the 'window' object.

Syntax:- `alert("message to be print");`

→ It can take any type parameter e.g. string, number, boolean, etc.

→ There is no need to convert a non-string type to a string.

ex:- `alert("This is me:"+100);` => concatenate

`alert(100);` => print 100

`alert(Date());` => display current date

* **CONFIRM (message):-** Use the 'confirm()' function to take the user's confirmation before starting some task.

→ The confirm() function displays a pobup message to the user with two buttons, 'OK' and 'Cancel'.

→ 'OK Button' returns 'true' when clicked and 'Cancel Button' returns 'false' when cliked

ex:-

```
let a,b;
if (confirm("Do you want to enter") == true) { console.log("welcome"); }
```

* PROMPT (message):- Use the 'prompt()' function to take the user's input to do further actions.
Syntax:- String prompt ([message], [default value]);

→ The 'prompt()' function takes two parameters. The first parameter is the message to be displayed, and the second parameter is default value in an input box.
eg:- var name = prompt ("Enter your name:", "Shiv");

* FUNCTIONS:-
→ function is 'object'.
→ Function is block of instruction which is used to perform a specific task.
→ A function get executed only when it is called.
→ The main advantage of function is we can achieve 'code reusability'.
→ To call a function we need its 'reference' and ().
→ Name of function is variable which hold the reference of function object.
→ Creating a function is using 'function keyword' supports function hoisting.
→ Therefore we can also call a function before function declaration
→ When we log to log function name the entire function definition is printed.
→ The scope within function block is known as 'local scope'.
→ Any member with local scope can not used outside the function block.
→ A parameter of a function will have local scope.
→ 'Variable' written inside function even using var have local scope.
→ Inside a function we can use the members of global scope.
→ In Javascript 'this' is a property of every function. (every function will have 'this keyword' except 'arrow function')

* PARAMETERS:-
→ The 'variables' declared in the 'function definition' is known as 'parameter'.
→ The parameters have local scope (can be used only inside function body)
→ Parameters are used to hold the values passed by 'calling a function'.

* ARGUMENTS:-
→ The 'values' passed in the method call statement is known as 'argument'.
→ Note:- An argument can be a literal, variable or an expression which gives a result.

* RETURN KEYWORD:-
→ It is a keyword used as control transfer statement in a function.
→ 'Return' will stop the 'execution' of the function and transfer control along with data to the caller.

Ways to create Functions:-
i) Function Declaration Statement:- Creating function using 'function' keyword.
Syntax:-

```
function func-variable (parameters){
    // statements
}

func_variable()
```

eg:-

```
function greet(){
    console.log("Good Morning");
}
greet(); => Good Morning
```

→ 'Function can be hoisted.'

eg:-

```
greet();
function greet(){
        console.log("Good Morning");
}
```

output:- Good Morning.

→ Function does not belongs to temporal dead zone

(ii) Function as Expression/Expression Function:-

* Functional Programming:- → Function which is passed to an 'variable' as a 'value' is called as 'first class function'.

→ In this approach, we generate 'Generic Function'. Here function task is not predefined. It perform multiple task not only single task.

→ The Function which accepts another function as a parameter or return a function is known as 'Higher Order Function'.

→ The Function which is passed to another function or the function which is returned by another function is known as 'Callback Function'.

→ Functional Programming is a programming technique where we pass a function along with a value to another function.

(ii) Function as Expression/Expression Function:-

→ first class function can not be Hoisted because it's object is created in execution phase.

→ Function does not belong to temporal dead zone.

Syntax:-

```
let variable varName = function (){
        //Statements
}
varName();
```

eg:-

```
let greet = function (){
        //Statements
}
greet();
```

GEC

V.P.	E.P.
greet	greet = func
greet ori	ori

(iii) IMMEDIATE INVOKE FUNCTION:-

→ When a function is called as soon as it's object is created is known as Immediate Invoke Function.

→ We have write the function inside the parenthesis to group it. [using Group operator -> (Function code)]

→ The function is not visible (available) outside the scope.

→ After grouping it, we have to use parenthesis to call this function.

→ Immediate Invoke Function [IIF] executes only once.

eg:-

```
(function (){
        console.log("Good Morning");
})()
```

→ It can not be Hoisted and go to Temporal dead zone.

(iv) ARROW FUNCTION:-

→ The main function of arrow function is to reduce the function syntax.

→ Arrow function is introduced in ES6.

→ If we have only 'single statement parameter', it is not necessary to use curly braces ({}), it is optional.

→ If we have only 'single parameter', it is not necessary to use "paranthesis '()'" for parameter.

→ It does not have it's own 'this' property.

→ There are have two type of return statements:-

(a) Implicit Return:- If there is only one statement and if block is not created then JS Engine will return that statement automatically.

→ Should not use '{}'.

(b) Explicit Return:- If block is created and function is not returning any value, JS Engine will return undefined. to return a value explicitely from block, we have to use 'return' keyword. If block is created then we have to use return keyword to return a value otherwise JS Engine will return undefined.

(v) HIGHER ORDER FUNCTION:-

→ Example of Arrow Function:-

⇒ `let x = () => {` `}` ⇒ `let x = () => { // statements }`

eg:- `let sum = (a,b) => { a+b; }` `sum(10, 20);`

⇒ Sum store refrence of function.

eg:- `let sum = (a,b) => { console.log(a); console.log(b); console.log(a+b); return a+b; }`

⇒ Sum store value or undefined.

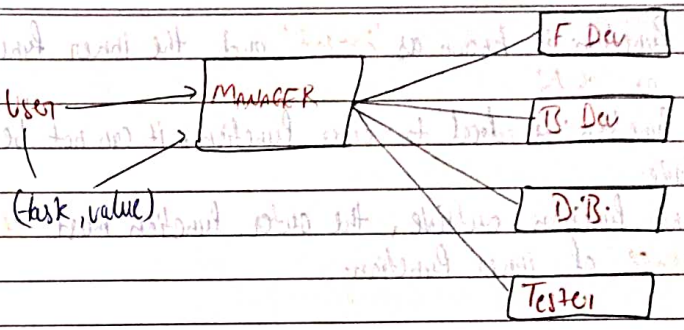


fig:- Functional Programming

eg:-

```
let sum = (a,b) => a+b;
let sub = (a,b) => a-b;
let mul = (a,b) => a*b;
let div = (a,b) => a/b;

let cal = (task, a,b) => {
            return task(a,b);
          }

let res = cal(sum, 10, 20);
```

Higher ORDER Func

Call Back Function.

* NESTED FUNCTION:-

→ The function inside another function is called as 'nested function'.

eg:-

```
function outer() {
    function inner() {
    }
}
```

→ The outer function is known as 'parent' and the inner function is known as 'child'.

→ The inner function is local to outer function, it can not be accessed from outside.

→ To use inner function outside, the outer function must return the reference of inner function.

(i) eg:-

```
function outer(){
        function inner(){
        }
    return inner;
}
```

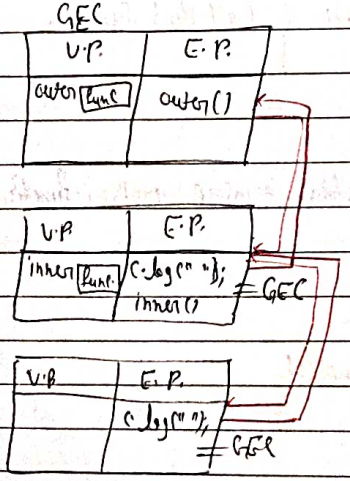
→ We can now call inner function from outside as follows:-

1st way:-

```
let fun = outer();
fun();        => inner is called
```

2nd way:-

```
outer()();    => inner is called
```

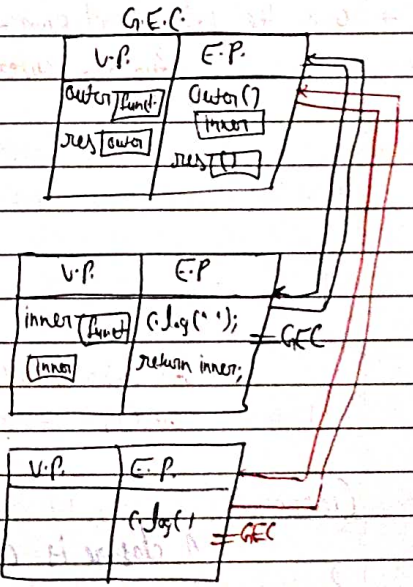


(ii)

```
function outer(){
        console.log("I am outer");
        function inner(){
            console.log("I am Inner");
        }
    inner();
}
outer();
```

(iii)

```
function outer(){
        console.log("I am outer");
        function inner(){
            console.log("I am Inner");
        }
    return inner;
}
let res = outer();
res();
```



* **LEXICAL SCOPE / SCOPE CHAIN :-**

→ The ability of JS Engine to search for a variable in the outer scope when variable is not available in local scope is known as lexical scope or scope chain.

→ It is ability of child to access variable from outside if it is not present in local scope.

→ ex:-

```
let a = 10;
function test(){
        a++;
        console.log(a);
}
test(); => 11
```

→ with the help of closure:-

```
function outer(){
        let a = 10;
        function inner(){
                console.log(a);
        }
        return inner;
}
let res = outer();
res(); => 10
```

* **CLOSURE :-**

→ A closure is created when a function is defined within another function and inner function need to access in outer function's scope.

→ Closure helps to achieve lexical scope from child function to parent function.

→ Closure preserves the state of parent function even after the execution of parent function is completed.

→ A child function will have reference to the parent.

→ Every time a parent function is called the new closure is created.

→ Disadvantage :- High memory Consumption.

* **ARRAY :-**

→ Array is object in javascript.

→ It is non-primitive type of literal.

→ It is a block of memory which is used to store multiple type of value (any type of literal) in same memory block.

→ Array size is dynamic (size is not fixed like JAVA), it means we can store 'N' number of elements and JS Engine will handle memory usage automatically.

→ Values stored inside array are refered as array elements.

→ Array elements are arranged in a sequence ie represented by integer number called as index.

→ Array index starts from zero to array size-1 (suppose array has 5 elements it's first index will be-0 and last index will be 4)

ex:-

```
let arr = [25, 27, 28, 30];
          0   1   2   3
```

size:- 4 (number of elements)

starting index:- 0 ending index:- size-1 = 4-1 = 3

→ We can access the array element with the help of array array object reference, square bracket ([]) and index.

=> array_object_ref[index]

→ If we try to access the index that is greater than array length we will get 'undefined'.

→ Array elements should be separated by Comma (,)

Ways to create Array:-

(i) By using square brackets [ ] and literals:-

```
let arr = [];
// empty array
let arr = [10,20,30];
// array with literals
```

(ii) By using new keyword and Array() constructor:-

```
let arr = new Array();
// empty array
let arr = new Array(10,20,30);
// array with literals -> [10,20,30]
```

Note:- Here 'arr' is a variable which holds the reference of array object. To access array element at index -> 1

Syntax:- array_object_ref[index]

ex:-

```
console.log(arr[1]); => 20
```

* Loop For Iterating Array Element:-

→ For loop:-

Syntax:-

```
for(initialization; condition; increment/decrement)
{
  // Statement
}
```

example:-

```
let arr = [10,20,30];
for(let i=0; i<arr.length; i++)
{
  console.log(arr[i]);
}
```

(iii) FOR-IN-LOOP:- Iterate over index

Syntax:-

```
for(let i in arr.name)
{
  // statements
}
```

example:-

```
let arr = [10,20,30];
for(let i in arr)
{
  console.log(arr[i]);
}
```

(iii) FOR-OF-LOOP:- Iterate over value.

Syntax:-

```
for(let variableName of arrName)
{
  //Statements
}
```

example:-

```
let arr = [20,30,90];
for(let value of arr)
{
  console.log(value);
}
```

* Array Program:-

```
let arr = [10, 20, 30, 40, 50];
Console.log(arr[0]); => [10,20,30,40,50]
Console.log(arr[1]); => 20
Console.log(arr[4]); => 50
Console.log(arr[5]); => undefined
arr[0] = 100;  => //update
for (let i in arr){
        Console.log(arr[i]); =>[100,20,30,40,50]
}
arr[5] = 200;  //inserting
for (let value of arr){
        Console.log(value); =>[100,20,30,40,50,200]
}
arr[7] = 300; //insert
for (let i=0; i<arr.length; i++)
{
 Console.log(arr[i]); =>[100,20,30,40,50,200,undefined,300]
}
```

* ARRAY METHODS :-

1→ push(value) method :-

→ It is used to insert element at last of array.

→ It returns the length of array.

ex:- let arr = [10,20,30,40,50];
arr.push(100);
console.log(arr); => [10,20,30,40,50,100]

2→ pop() method :-

→ It is used to delete element from last index of array.

→ It returns deleted element.

ex:- let arr = [10,20,30,40,50];
arr.pop();
Console.log(arr); => [10,20,30,40]

3→ unshift(value) :-

→ It is used to insert element at first index of array.

→ It return array length.

ex:-
```
let arr = [10,20,30,40,50];
arr.unshift(200);
Console.log(arr); => [10,20,30,40,50,200] [200,10,20,30,40,50]
```

4→ shift() method :-

→ It is used to delete element from first index of array.

→ It returns deleted element.

ex:- let arr = [10,20,30,40,50];
arr.shift();
console.log(arr); => [20,30,40,50]

5→ splice() method :-

→ It is used to perform insertion, deletion and updation in array.

→ It will modify the original array.

→ It returns array of deleted elements.

Syntax:- arr_ref.splice(a,b,c)

a- starting index.

b- number of elements to be deleted.

c- elements to be inserted.

6→ slice() method:- It is used to copy array elements.
→ It will not modify the original array
→ It returns array of copied elements.

Syntax:-
arr.slice(a,b),
a - starting index.
b - last index
=> Here last index is excluded -> last index - 1.

7→ indexOf() method:- It is used to get the index of array element.
→ If element is available -> it returns element's index.
→ If element is not available -> it returns -1.

Syntax:- arr.indexOf(a,b);
a - value to be searched.
b - search starting index.
=> If we does not pass last argument, it will 0 by default.

8→ includes() method:- It is used to check element is available or not.
→ If element is available -> return true.
→ If element is not available -> return false.

Syntax:- arr_ref.includes(a,b);
a - value to be searched
b - value to be starting index
=> If we doesn't pass last argument, it will 0 by default.

9→ reverse() method:- It is used to reverse the array.
→ It will modify the original array.

ex:- let arr = [10,20,30,40,50];
console.log(arr.reverse()); => [50,40,30,20,10]

10→ sort(callback):- It will modify the original array.
→ If callback returns -ve value → it will sort in ascending order.
→ If callback returns +ve value → it will sort in the descending order.
→ If callback returns 0 value → it will not sort.

Example:- sort array in ascending order:-
let arr = [100, 2000, 380, 940, 50, 0, 2];
Console.log(arr.sort((a,b) => a-b));
Output:- [0, 2, 50, 100, 380, 940, 2000]

→ Sort array in descending order:-
let arr = [100, 2000, 380, 940];
console.log(arr.sort((a,b) => b-a));
output:- [2000, 940, 380, 100]

11→ foreach(callback):- It is a higher order function.
→ It is used to iterate over array elements and index.
→ It doesn't return anything, so js engine implicitely return undefined.

Syntax:- arr_ref.foreach((value, idx, array) => {
// statements
})

Example:- arr.foreach((val) => {
if(val%2===0)
{
console.log(val + " " + "is even number;");
}
});

12→ map(callback):- It is a higher order function.
→ It is used to iterate over array.
→ It will not modify original array.

→ It returns new array.
→ The value returned by callback function will be inserted in new array, if it doesn't return anything 'undefined' will be stored.

Syntax:- arr_ref.map((value, index, array)=>{
// statements
})

Example:- create new array where each element of given array is multiple of 8.

```
let arr=[10,20,30,40,50];
let new_arr = arr.map(value => value*8);
console.log(new_arr);
```

Output:- [80, 160, 240, 320, 400]

12→ filter(callback):- It is a higher order function.
→ It is used to iterate over array.
→ It will not modify original array.
→ It returns a new array.
→ Here, element will be inserted in new array only when callback function returns ~~two~~ 'true'.

Syntax:- arr_ref.filter((value, index, array)=>{
// statements
})

Example:- Create new array where elements are greater than 40

```
let arr=[10,20,30,40,50,60,70];
let new_arr = arr.filter(value => {
    if (value > 30)
        return true;
});
console.log(new_arr);
```

Output:- [40,50,60,70]

14→ reduce(callback, initial_value):-
→ It is a higher order function.
→ It is used to iterate and conclude result to a single value.
→ It will not modify original array.
→ It returns a single value.
→ Here, single value is returned after complete iteration of array. Value is stored in a variable which is used to result, we refer it as accumulator.

Syntax:- arr_ref.reduce((accumulator, value, index, array)=>{
// statements
}, initial_value_of_accumulator)

If we does not pass initial value of accumulator first element of array will be stored automatically.

Example:- Find the sum of all elements of array.

```
let arr = [10, 20, 30, 40, 50, 60, 70];
let result = arr.reduce((acc, value)=>{
    acc = acc + value;
    return acc;
}, 0);
console.log("Sum of all elements:", result);
```

Output:- Sum of all element: 280

15→ Array.isArray(literal):-
→ It is used to check given literal is array or not
→ If it is array → it will return true.
→ If it is not array → it will return false.

Example:-

```
console.log(Array.isArray({}));  // false
console.log(Array.isArray(10));  // false
```

```
console.log(Array.isArray([10,20,30])); // true
```

16→ **Array.from(literal):-**

→ It is used to convert iterable literals (like object or string) to array.

→ If literal is iterable -> it returns new array of elements.

→ If literal is not iterable -> it returns empty array.

Example:- Convert string to Array.

```
const str = "hello";
const arr = Array.from(str);
console.log(arr);
```

Output:- ["h", "e", "l", "l", "o"]

* **OBJECT:-**

→ An object is a block of memory which has state (variable), behaviour (methods) and where we can store heterogenous data.

→ An object is a collection of key-value pairs that can contain various datatypes, such as numbers, strings, arrays, functions and other objects.

→ In one object we can have multible key value pair and it should be separated by ',' comma.

→ We can access value of object using (.) operator or square bracket [], object reference and key-name.

* **Object Key (Property]:-**

→ Object key (property) will be automatically converted into string by JS engine.

→ If keys name are in number, JS engine will converted them into string and arrange them in ascending order.

→ To write space separated key names, we have to enclose key name with double quotes.

→ If we want to give computed or user defined property then we have to use square brackets and variable name.

→ If key-name is same as variable name which hold the value, instead of writing two times we can write variable name only once.

Example:-

```
let phono = 8308778884;
let obj = {
    phono,    // phono : phono
};
```

* **Ways to create Object:-**

1→ **By using curly braces {} and literals:-**

Syntax:- `Obj-ref/variable-name = { key: value, }`

Example:-

```
let obj = {};          // empty object
let obj1 = { name: "chombi",
             age: 16 }; // object with literals
```

2→ **By using new keyword and Constructor Object:-**

Syntax:-

```
variable-name/Obj-ref = new Object();
```

Example:-

```
let obj = new Objective(); // {}. empty object
let obj1 = new Object({ name: "chombi" }); // object constructor with
                                              new keyword.
```

3→ By using new keyword and Constructor function :-

Syntax :- function Variable_name (parameters) {
this.key = value;
}
Variable_reference = new function_reference (arguments)

Example :-

```
function CreateObject (name, age) {
    this.name = name;
    this.age = age;
}
obj = new CreateObject (Shiv, 20)
```

4→ By using Class.

* Access Object Value :-

→ There are two ways to access the object value.

1→ By using dot operator (.) and key name :-

Syntax :- obj_ref . key_name

Example :-

```
let obj = { "chombi" : "name", age : 16 }
console.log (obj.chombi)    //name
console.log (obj.age)       //16
```

2→ By using square brackets ([]) and key name :-

Syntax :- obj_ref ["key_name"]

Example :-

```
let obj = { name : "chombi", age : 16 }
console.log (obj["name"])    //chombi
console.log (obj["age"])     //16
```

→ If we try to access property which is not available in object we will get undefined.

* Object Methods :-

→ In Javascript, object methods are functions that are attached to the object, and can be called on that object on # reference.

→ To call a function, we use square brackets instead dot operator.

→ Here, speak is a variable which holds the function reference.

Example :-

```
let obj1 = { name : "chombi", age : 16,
    speak : function () {
        console.log ("i can speak");
    } }
console.log (obj1["speak"] ());    // i can speak
```

→ Access object property inside function declared with function keyword.

Example :-

```
let obj1 = { name : "chombi", age : 16,
    speak : function () {
        console.log ("My name is ${this.name} and i can speak");
    } }
console.log (obj1["speak"] ());    // My name is chombi, age 16 and i can speak.
```

→ Access object property inside function - Arrow function

Example:-

```
let obj1 = { name: "chombi", age: 16,
             speak: () => {
             console.log ('My name is ${obj1.name}, age ${
                 obj1.age} and i can speak');
             }}
console.log(obj1["speak"]());  // My name is chombi, age
                                  16 and i can speak.
```

→ Here we are can access object property, by using object reference because arrow function is not having 'this' property.

* Add Key Value in object:-

→ To add key-value pair we can using dot operator and square brackets.

→ By using dot (.) operator and key name:-

```
let obj = { name: "Chombi", age: 16}
obj.country = "India";      // new key added in object
```

→ By using square ([]) bracket and key name:-

```
let obj = { name: "chombi", age: 16 }
obj["country"] = "India";  // new key added in object
```

* Update key value in object:-

1→ By using dot (.):-

```
let obj = { name: "chombi", age: 20 }
obj.name = "Shiv";  // update name key
```

2→ By using square ([]):-

```
let obj = { name: "chombu", age: 20}
obj["name"] = "Shiv";  // update name key
```

* Delete Key Value in Object:-

1→ By using dot (.):-

```
let obj = { name: "chombi", age: 21 }
delete obj.name;  // object key deleted
```

2→ By using square ([]) bracket:-

```
let obj = { name: "chombi", age: 21 }
delete obj["name"];  // object key deleted
```

Check Property is Available in Object or Not:-

→ We can check using "in" operator.

Syntax:- "property name" in object_name

Example:-

```
let obj = { name: "chombi",
            age: 16 }
console.log("name" in obj)   // true
console.log("age" in obj)    // true
console.log("city" in obj)   // false
```

→ in operator give result in boolean (true/false).

* Copy of Object:-

→ We can create copy of object by two types:-

(a) Shallow Copy
(b) Deep Copy

Shallow Copy:-

→ The copy of object that is directly connected with original object is called as shallow copy or shallow object.

→ Here we store reference of original object in a new variable, now new variable starts pointing same memory block.
→ So if we make any changes in copy, it will be reflected to original object because both variable are pointing to same memory block.

Example:-

```
let obj = { name: "chambi",
            age: 16 }
let obj_cpy = obj;
obj_cpy.age = 20;
console.log(obj_cpy);   // { name: "chambi", age: 20 }
console.log(obj);       // { name: "chambi", age: 20 }
```

Deep Copy:-

→ The copy in which original object is not connected with it's copy, is called as deep copy.
→ Here, we create separate empty object and after that we copy key-value pair of original object into new empty object.
→ Now, if we make any changes in copy, it will not be reflected to original ~~string~~ object. Because we have to create separate memory blocks.
→ Create copy using for loop:-

```
let obj1 = { name: "chambi", age: 16 }
let obj2 = {}                    // empty object
for (prop in obj1) {
    obj2[prop] = obj1[prop]
    console.log(obj2) }
obj2.age = 20;
console.log(obj2);   // { name: "chambi", age: 20 }
console.log(obj1);   // { name: "chambi", age: 16 }
```

* Object In-Built Methods:-

Object.keys(obj_ref):-

→ Returns an array of given object's property names.

```
const obj = { a:1, b:2, c:3 };
console.log(Object.keys(obj));
```

Output:- ["a", "b", "c"]

Object.values(obj_ref):-

→ Returns an array of given object's values.

```
const obj = { a:1, b:2, c:3 };
console.log(Object.values(obj));
```

Output:- [1, 2, 3]

Object.entries(obj_ref):-

→ Returns an array of key-value pairs in an array.

```
const obj = { a:1, b:2, c:3 };
console.log(Object.entries(obj));
```

Output:- [['a',1], ['b',2], ['c',3]]

Object.assign(target_object, src1, src2, ---, srcn):-

→ Copies key-values pair from one or more source objects to a target object.

```
const obj_target = { a:1, b:2 };
const source = { c:3, d:4 };
const result = Object.assign(obj_target, source);
console.log(result);
```

Output:- { a:1, b:2, c:3, d:4 }

* This Keyword:-
→ It is a keyword.
→ It is a variable, which holds the reference.
→ In GEC it holds the address of window object.
→ It is a local variable of every function in JS and holds the address of window object. Except in Arrow Function (for arrow function it stores undefined.
→ Inside object methods, 'this' holds the reference of current object (not in arrow function).

* Destructuring:-
→ The process of extracting the values from the array or object into the variable is known as destructuring.
→ The most used data structures in JavaScript are object and array, both allows us to unpack individual into variables.

Object Destructuring:-
→ The process of extracting values from the object into the variables is known as object destructuring.
→ All the key names provided on LHS are consider as variable and these variables should be declared and written inside curly braces.
→ The variable name should same as object key name.
→ JS engine will search for the key inside the object.
→ If the key is present, the value is extracted and coby into variable.
→ If the key is not present, undefined is store in variable.
→ After the destructuring, we can directly access variable names, without using object reference.

Example:-

```
let obj = { name: "Chombi", age: 16 }
let { name, age, country } = obj;
console.log (name);      // Chombi
Console.log(age);        // 16
Console.log (country);   // undefined
```

→ Here, we are trying extract name, age and country from obj. name and age is present in obj but country is not, so inside country JS engine stored undefined and for name, and age we have respective values.

* Array Destructuring:-
→ The process of extracting the values from the array into the variable is known as array destructuring.
→ All the key names provided on LHS are consider as variable and should be written inside square brackets.
→ JS engine will extract the array values and stored them variable in the same order as they are present inside array.
→ If try to access value which is not present inside array, JS-engine will store undefined inside that variable.

Example:-

```
let arr = [10, 20, 30];
let [a,b,c,d] = arr;
console.log (a);         // 10
Console.log (b);         // 20
console.log (c);         // 30
Console.log (d);         // undefined
```

→ Here we are trying to extract value from array into variable. a,b,c varible have values but d store undefined.

Destructuring In Function :-

→ We can destructure array or object in function parameter so that we can access value directly.

→ **Destructuring Object in function parameter :-**

→ At the time of object destructuring, we have to make sure variable name is same as object key name, and write within curly braces.

```
function details ({name, age}){
    console.log(name);    //Chambi
    console.log(age);     //16
}
let obj = {name: "chambi", age: 16};
details(obj)              //function call
```

→ Here we passed object as an argument to details function, and we have destructured values in parameter only.

→ **Destructuring array in function parameter :-**

→ At the time of array destructuring, we have to keep variables between square brackets. Values will be destructured in the same order, they are available in array.

```
function details ([a,b,c,d]){
    console.log(a);    //10
    console.log(b);    //20
    console.log(c);    //30
    console.log(d);    //40
}
let arr = [10,20,30,40];
details(arr)           //function call
```

→ Here we have passed array as an argument to details function, and we have destructured values in parameter only.

* **Rest and Spread :-**

Rest Parameter :-

→ Rest parameter is used to accept multiple values, stored them in an array and array's preference will stored the variable that we have used for rest.

→ Rest can or accept n number of values and stored them in an array.

→ To make a variable rest, we have to put '...' before variable name.

→ Syntax :- `let ... variable_name;`

→ We can use rest in function when we don't know the exact number of arguments.

→ In function, there can be only one rest parameter and it should be the last parameter.

```
function details (a,b,...z){
    console.log(a);    //10
    console.log(b);    //20
    console.log(z);    //[30,40,50]
}
details(10,20,30,40,50);    //function call
```

→ **Uses of Rest parameter :-**

→ Rest parameter can be used in function definition parameter list to accept multiple values.

→ It can also be used in array and object destructuring.

→ If we pass literals to three dot (...), it will accept all literals

and behave as rest parameter.

Spread Operator :-

→ It is used to unpack element from iterables (like array or object).

→ Use cases:-

→ The unpack data can be sent to the function as an argument by using operator in the function call statement.

```
let arr = [10, 20, 30, 40];
function sum (... data) {
  let acc = 0;
  for (let val of data) {
    acc += val;
  }
  return acc;
}
let result = sum (... arr);
console.log (result);
```

Output:- 561

// ... data - rest parameter
// ... arr - spread operator

→ We can ask the spread operator to store the unpack element in array object by using spread operator inside [ ] brackets.

```
let new_arr = [... arr];
console.log (new_arr);
```

Output:- [10, 20, 30, 40]

→ We can ask the spread operator to store the unpack element in object by using spread operator inside { } brackets.

```
let human1 = { name: "chomba", age: 21, };
let human2 = { ... human1 }
console.log (human2); // { name: "chomba", age: 21 }
```

→ If we do not pass literals to three dots (...), it will unpack all literals and behave as a spread parameter.

Call, Apply, Bind :-

Introduction:-

→ Call, Apply, Bind methods are used to store the object reference in 'this' keyword of function.

→ When function's 'this' have reference of object, then we can access states and behaviours of that object.

→ For practice we will use these objects as reference.

```
let human1 = { name: "chombi", age: 20, };
let human2 = { name: "Dinga", age: 19, };
let human3 = { name: "Nimbi", age: 18, };
```

→ Below function we will use to access objects properties by using call, apply and bind methods.

```
function details (a, b, c) {
  console.log ("Name: " + this.name);
  console.log ("Age: " + this.age);
  console.log ("value of a: " + a);
  console.log ("value of b: " + b);
  console.log ("value of c: " + c);
}
```

Call:-

→ Call method accepts object reference as first argument and accepts 'n' number of arguments.

→ Here, arguments are passed to the function's parameter list.
→ It will call the function immediately.

Example:- print age, name of object human1 and print function arguments.

```
details(human1, 10, 20, 30);
```

Output:-

```
Name : Chinki
Age: 20
value of a: 10
value of b: 20
value of c: 30
```

Apply:-

→ Apply method accepts of 2 arguments where object reference is first argument and 2nd argument is the array of arguments.
→ Here arguments are passed to the function's parameter list.
→ It will call the function immediately.

Example:- print name, age of object human2 and print function arguments.

```
details.apply(human2, [11, 22, 33]);
```

Output:-

```
Name : Dinga
Age : 19
value of a: 11
value of b: 22
value of c: 33
```

Bind:-

→ Bind method accepts object reference as 1st argument and accepts 'n' number of arguments.
→ Here 'n' number of arguments are passed to the function's parameter list.
→ It will not call the function immediately.
→ It returns a new function in which 'this' keyword is pointing to the object reference we have passed.
→ To execute the function we need function reference and parenthesis.

Example:- print name, age of object human3 and print function arguments.

```
let func = details.bind(human3, 77, 88, 99);
func();
```

Output:- Name : Nimbi
Age : 18
value of a: 77
value of b: 88
value of c: 99

If function is available inside the object and use call, apply, bind for all:-

```
let Shanna = { name: "Shanna", age: 21 };
let Chandan = { name: "Chandan", age: 22 };
let Saurabh = { name: "Saurabh", age: 50 };
let Shanna.display = function () { "My name ${this.name} and age
                                   is ${this.age}" };

Shanna.display.call(Shanna);      // My name Shanna and age is 21
Shanna.display.apply(Chandan);    // My name Chandan and age is:
let S = Shanna.display.bind(Saurabh);
console.log(S()); / S();          // My name Saurabh and age is 50
```

* Prototype:-

→ Every function in JS will be associated with an object called as prototype.
→ A function object has a attribute called prototype which holds the

reference of the prototype object.

* **Prototype:-**
→ Every function in JS will be associated with an object called as prototype
→ A function object has a attribute called prototype which holds the reference of the prototype object.
function → prototype object

→ Every function is associated with prototype object.

* **__proto__ :-**
→ When an object is created a prototype object is not created instead the object will have reference of the prototype object referred using __proto__

* **Prototypal inheritance :-**
→ Prototype inheritance is a fundamental concept in JavaScript's object-oriented programming model. In It allows objects to inherit properties and methods from other objects, forming a prototype chain.
⇒ In JS inheritance is achieved using prototype hence it is known as prototypal inheritance.

* **Prototype Chain :-**
→ The prototype chain is a mechanism in JavaScript that allow objects to inherit properties and methods from other objects.
→ The prototype chain forms a hierarchy of object. where each object's prototype is linked to it's parent object's prototype. creating a chain of inheritance. By following this chain, objects can inherit properties and methods from their prototype objects.

Ex:-
→ Prototype inheritance

```
Array.prototype.John = function(){
    console.log("John is a good person");
}

arr.John();
```

→ For checking inherited values:-

```
let arr = [10,20];
console.log(arr.__proto__);
console.log(arr.__proto__.__proto__);
console.log(arr.__proto__.__proto__.__proto__);
```

* **String :-**
→ String methods in JavaScript provide various operations and manipulations that can be performed on strings. Here are explanation for several commonly used string methods:-

1→ **indexOf() :-** This method searches for a specified substring within a string and returns the index of the first occurrence of the substring; if not found, it returns -1.
Syntax:- string.indexOf(substring)
Return:- The index of the first occurence of the substring or -1 if not found.
Example:-

```
let str = "Hello World";
console.log(str.indexOf("e"));   // 1
```

2→ **includes() :-** This method check if a specified substring is present within a string. It returns boolean value indicating whether the substring is found or not.
Syntax:- string.includes(substring);

Return:- true if the substring is found, false otherwise.
Ex:- console.log(str.includes('e')); // true

3→ endsWith():- This method checks if a string ends with a specified substring. It returns a boolean value indicating whether the string ends with substring.
Syntax:- String.endsWith(substring);
Return:- true if the string ends with the substring, false otherwise.
Ex:- console.log(str.endsWith('word')); // true

4→ startWith():- This method checks if a string start with a specified substring. It returns a boolean value indicating whether the string start with the substring.
Syntax:- string.startWith(substring);
Return:- true if the string starts with the substring, false otherwise.
Ex:- console.log(str.startWith('he')); //true

5→ slice():- This method extracts a portion of a string and returns a new string containing the extracted part. It takes two optional arguments: the starting index and the ending index(exclusive). If no ending index is provided, it extracts till the end of the string.
Syntax:- String.slice(startIndex, endIndex);
Return:- The extracted substring as a new string.
Ex:- console.log(str.slice(0, 5)); // Hello

6→ substring():- This method is similar to slice(), but it also allows negative indices. It extracts a portion of a string and returns a new string containing the extracted part. The negative indices represents positions relative to the end of the string.
Syntax:- string.substring(startIndex, endIndex);

Returns:- The extracted substring as a new string.
Ex:- console.log(str.substring(0, 7)); // Hello Wo

7→ split():- This method splits a string into an array of substrings based on a specified separator. It divides the string wherever the separator occurs.
Syntax:- string.split(separator);
Returns:- An array of substrings
Ex:- console.log(str.split(',')); // Array(11): ["H","e","l","l","o"," ","W","o","r","l","d"]

8→ join():- This method joins the elements of an array into a string, using a specified separator. It concatenates the elements together with the separator in between.
Syntax:- array.join(separator);
Returns:- A string containing the joined elements.
Ex:- console.log(a.join(',')); // if a[] a=[1,2,3] then o/p:- 1,2,3

9→ toLowerCase():- toLowerCase() method converts a string to lower case letters. It does not change the original string.
Syntax:- string.toLowerCase();
Returns:- string in lower case.
Ex:- console.log(str.toLowerCase()); // hello world

10→ toUpperCase():- The toUpperCase() method converts a string to upper case letters. It does not change the original string.
Syntax:- String.toUpperCase();
Return:- string in upper case
Ex:- console.log(str.toUpperCase()); // HELLO WORLD