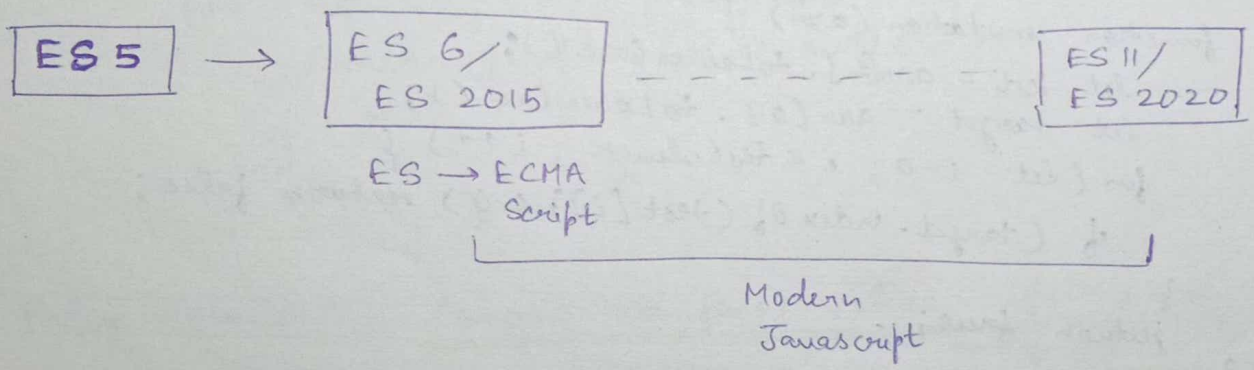
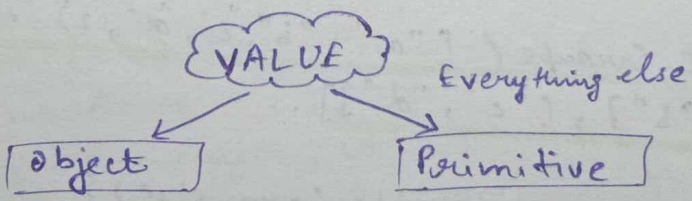
< JAVASCRIPT >

Javascript is a High level, Object Oriented, Multi-Paradigm programming Language.

* High level : don't have to worry about stuff like memory management.
* Object-oriented: Based on objects for storing most kinds of data.
* multi-paradigm: we can use different styles of programming. such as imperetive & declarative.

- JS is used to build - Dynamic effects and web applications in the browser.
  - Web applications on web server.
  - Native mobile applications.
  - Native desktop applications.






```
let me = {
  name: 'Jonas'
};
```

```
let firstName = 'Jonas';
let age = 30;
```

7 Primitive Data Types:

1. Number
2. String
3. Boolean
4. Undefined ('empty value')
5. Null - also 'empty value'
6. Symbol (ES 2015) - value is unique and can't be changed.
7. BigInt (ES 2020) - Larger integers than no's can hold.

Basic Javascript

1. Comment
   - in line //
   - multi-line /* - - - */

2. Declaring variables:
   var myName Is

3. storing values : (=) assignment operator.
   var myVar;
   myVar = a;
   or
   var myVar = a;

4. Adding 2 ~~or more~~ nos => var sum = i+1;
   or var sum += ;

   Subtract -= or i-1;
   multiply i*i or *= ;
   Divide / / =
   Remainder %.

5. Declayring string variable = "string"

6. Esfaping literal quotes in strings =>
   var samplestr = "Alan said, \" Peter is learning Javascript \". " ;

7. Escape sequences in strings:
   \' single quote
   \" double quote
   \\ backslash
   \n new line
   \r carriage return
   \t tab
   \b word boundary
   \f form feed.

(8) when [+] operator is used with a string value, it is called 'concatenation' operator.

Eg: var ourStr = "I come first." + "I come second.";

output : I come first. I come second.

(or)

```
var ourStr = "I come first.";
str += "I come second.";
```

(9)

```
var ourName = "freeCodeCamp";
var ourStr = "Hello, our name is " + ourName +
             ", how are you?";
```

ourStr have the value → Hello, our name is freeCodeCamp, how are you?

(10) Appending variables to Strings —

```
var anAdjective = "awesome!";
var ourStr = "freecodecamp is ";
ourStr += anAdjective;
```

Output: Freecodecamp is awesome!

(11) ~~fing~~ Find Length of string : .length.

(12) Use bracket notation to find first character in a String —

```
var firstName = "Charles";
var firstLetter = firstName[0];
```

output
firstLetter → C

(13) Understand String Immutability :

In Javascript, String values are 'immutable', which means they cannot be altered once created.

```
var myStr = "Bob";
myStr[0] = "J"; ✗
```

```
var myStr = "Bob";
myStr = "Job";
```

Output
Job.

(14) Bracket notation to find Last character :

```
var firstName = "Ada";
var lastLetter = firstName[firstName.length -1];
```

Output: a

(15) Find N^{th} to last Character :

```
var firstName = "Augusta";
var thirdToLastLetter = firstName[firstName.length -3];
```

Output s.

(16) Word Blanks :

```
var sentence = "It was really" + "hot" + ", and we" +
"laughed" + "ourselves" + "silly" + ".";
```

Output: It was really — not — and we — laughed — ourselves — silly — .

(17) Store multiple values in one variable using JS arrays :

```
var sandwich = ["peanut butter", "jelly", "bread"].
```

(18) Nest one array within other :

```
[["Bulls", 23], ["White Sox", 45]]
```

(19) Access Array data with Indexes :

```
var array = [50,60,70];
array[0];
var data = array[1];
```

array[0] is now → 50 and data has value → 60.

(20) Modify Array :

```
var ourArray = [50, 40, 30].
ourArray[0] = 15.
```

Output :

[15, 40, 30].

(21) Access Multi-Dimensional Arrays with Index:

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

Output:

arr[3][0][1]; => 11

(22) Manipulate arrays:

(i) push() • push() takes one or more parameters and "pushes" them onto end of array.

(ii) pop() • pop() removes last element from an array and returns that element.

(iii) shift() - removes first element.

(iv) unshift() - adds element at beginning of array.

(23) Write reusable function in JS.

```
function functionName() {
    console.log("Hello World");
}

functionName()
```

Output: Hello World

(24) Passing values to functions with Arguments:

```
function testFun(param1, param2) {
    console.log(param1, param2);
}

testFun(1, 2);
```

Now, param1 = 1
param2 = 2] - assigned.

(25) Global scope and Functions:

scope refers to → visibility of variables.

Variables which are defined outside of a function block have Global scope.

This means they can be seen everywhere in Javascript.

```
Var myGlobal = 10;
function fun1() {
    oopsGlobal = 5;
}
```

Output:
myGlobal = 10
oopsGlobal = 5.

(26) Local Scope and Functions:

Variables which are declared within a function, as well as f^n parameters, have LOCAL scope. This means they are visible within that function.

```
function myTest() {
    var loc = "foo";
    console.log(loc);
}
myTest();
console.log(loc);
```

- myTest f^n will display string foo in console.
- console.log(loc) will throw an error, as loc is not defined outside function.

(27) Global V/s Local scope in Functions:

It is possible to have both LOCAL and GLOBAL variables with same name. Local variable takes precedence over global variable.

```
Var someVar = "Hat";
function myFun() {
    var someVar = "Head";
    return someVar;
}
```

Output: function myFun will return string Head.

(28) Return value from a Function with Return:

We can pass values into functions with arguments. You can use return statement to ~~send~~ send a value back out of a function.

Eg:

```
function plusThree(num) {
    return num + 3;
}
var answer = plusThree(5);
```

Output - 8

(29) Undefined value returned from function:

function can include return statement but it does not have to. In that case fⁿ doesn't have a return statement, when you call it, the function processes inner code but returned value is undefined.

```
var sum = 0;
function addSum(num) {
    sum = sum + num;
}
addSum(3);
```

Note: addSum is a fⁿ without return statement. The fⁿ will change global sum variable but returned value of function is undefined.

(30) Assignment with a returned value:

Everything to the right of the equal sign is resolved before value is assigned. This means we can take return value of a fⁿ and assign it to a variable.

```
ourSum = sum(5,12);
```

- will call sum function, which returns value of 17 and assign it to ourSum variable.

(31) Stand in Line:

Queue is an abstract Data Structure where items are kept in order. New items can be added at the back of queue and old items are taken off from front of queue.

for example:-

```
var testArr = [1,2,3,4,5];
function nextInLine(arr, item) {
    arr.push(item);
    var item = arr.shift();
    return item;
}
```

(32) Boolean: returns true or false

(33) If statement: ~~function test ( )~~

```
if (condition is true) {
    statement is executed
}
```

(34) Comparison with equality operator:

== It ~~involves~~ compares two values and returns true if they're equivalent or false if they are not.

Eg:

1 == 1	True
1 == 2	False
1 == '1'	true
"3" == 3	True

In order for Javascript to compare two different Datatypes, it must convert one type to other. This is called - Type Coercion.

(35) Strict Equality Operator: (===):

unlike equality operator which attempts to convert both values being compared to common datatype, strict equality operator does not perform - 'type conversion'

3 === 3	True
3 === '3'	False

(36) • Inequality Operator: (!=):

Inequality operator (!=) is opposite of equality operator. It means not equal to returns 'FALSE' where equality would return 'TRUE'.

• Strict Inequality operator (!==).

It is logical opposite of strict equality operator. It means "strictly not equal" and returns 'false' where strict equality would return 'true'

Eg:

3 !== 3	false
3 != = '3'	true
84 != = 3	true.

- Greater than operator (>)
- Greater than or equal to (>=)
- Less than operator (<)
- Less than or equal to (<=)

(37) Logical AND operator: (&&) returns True if and only if operands to left and right of it are true.

```
if (num > 5) {
    if (num < 10) {
        return "yes";
    }
}
return "No";
```

(38) Logical OR operator: (||)

returns true if ~~either of~~ either of operands is true.

(39) • Else statements:

```
if (num > 10) {
    return "Bigger than 10";
} else {
    return "10 or Less";
}
```

• Else if:

```
if (num > 15) {
    return "Bigger than 15";
} else if (num < 5) {
    return "Smaller than 5";
} else {
    return "Between 5 and 15"; }
```

(40) Replacing if Else Chains with Switch:

```
if (val === 1) {
    answer = "a";
} else if (val === 2) {
    answer = "b";
} else {
    answer = "c";
}
```

replacing:

```
switch (val) {
    case 1;
        answer = "a";
        break;
    case 2:
        answer = "b";
        break;
    default:
        answer = "c";
}
```

(41) Objects in Javascript:

→ Objects are similar to ~~of~~ arrays, except that instead of using indexes to access and modify their data, you access ~~them~~ the data in objects through what are called — 'properties.'

Objects are ~~very~~ useful for storing data in structured way.

Example:

```
var cat {
    "name" : "Whiskers"},
    "legs" : 4,
    "tails" : 1,
    "enemies" : ["Water", "Dogs"] };
```

- We can omit quotes for single-word string properties.
- If object has any non-string properties, JS will automatically typecast them as strings.

(42) Accessing Object properties with DOT NOTATION

Eg:

```
var myObj = {
    prop1 : "val1"};
    prop2 : "val2" };
var prop1val = myObj.prop1;
var prop2val = myObj.prop2;
```

BRACKET NOTATION

```
var myObj = {
    "SpaceName" : "Kirk",
    "MoreSpace" : "Spock" };
myObj["SpaceName"];
myObj["MoreSpace"];
```

with VARIABLES

Eg:

```
var dogs = {
    Fido : "Mutt", Hunter : "Doberman", Snoopie : "Beagle" };
var myDog = "Hunter";
var myBreed = dogs[myDog];
console.log(myBreed);
```

Output:

Doberman.

(43) • Updating Object Properties: we can use either dot or bracket notation

Eg:

```
var ourDog = {
    "name" : "Camper",
    "legs" : 4,
    "tails" : 1,
    "friends" : ["everything"] };
```

Output:

ourDog.name = Happy Camper

- Add new properties to JS Object:

```
ourDog.bark = "bow-wow";
```

- Delete properties from JS object:

```
delete ourDog.bark;
```

(44) • Using Objects for Lookups:

```
var alpha = {
  1: "Z",
  2: "Y",
  3: "X",
  ...
  25: "B",
  26: "A" };
alpha[2];
alpha[24];
var value = 2;
alpha[value];
```

Output

alpha[2] is string Y
alpha[24] is string C
alpha[value] is string Y

(45) • Testing Objects for Properties:

We can use .hasOwnProperty(propname) method of objects to determine if that object has the given property name.
.hasOwnProperty() return True or False.

• Manipulating Complex Objects:

Sometimes we may want to store data in flexible Data structure.
JS Object is one way to handle flexible data.
They allow combinations of strings, no's, booleans, arrays, functions, and Objects.

```
var ourMusic = [
  { "artist": "Daft Punk",
    "title": "Homework",
    "release_year": 1997,
    "formats": [
      "CD",
      "cassette",
      "LP" ],
    "gold": true,
  }
];
```

- This is an array which contains one object inside.
- The object has various pieces of metadata about an album.

⇒ JavaScript Object Notation (JSON) is a related data interchange format used to store data.

• Accessing Nested Objects:

Eg:

```
var ourStorage = {
  "desk": { "drawer": "stapler" },
  "cabinet": { "top drawer":
    { "folder1": "a file",
      "folder2": "secrets" }
```

```
        "bottom drawer" : "soda"
    }
};
```

ourStorage.cabinet["top drawer"].folder2; → output: secrets

ourStorage.desk.drawer; → stapler.

- **Accessing Nested Arrays:**

```
ourPets[0].names[1];
ourPets[1].names[0];
```

(46)

- JS while Loops.
- JS for Loops.
- JS Do... while Loops.
- Replace Loops using Recursion:

Recursion is the concept that a function can be expressed in terms of itself. To help understand this, multiply the first n elements of an array to create the product of those elements.

```
function multiply(arr, n) {
  var product = 1;
  for (var i=0; i<n; i++) {
    product *= arr[i];
  } return product;
}
```

we notice, multiply(arr, n) == multiply(arr, n-1) * arr[n-1].

This means we can rewrite multiply in terms of itself and never ~~used~~ to need to use a loop.

```
function multiply(arr, n) {
  if (n <= 0) {
    return 1;
  } else {
    return multiply(arr, n-1) * arr[n-1];
  }
}
```

The recursive version of multiply breaks down like this. In the base case, where n <= 0, it returns 1. For larger values of n, it calls itself, but which n-1. That fn call is evaluated in same way, calling multiply again until n <= 0. At this point, all fn's can return the original multiply return the answer.

Note: Recursive f^ns must have base case when they return without calling the f^n again (in this example when n<=0) otherwise they can never finish executing.

(47) Generate random fractions with JS:

- JS has a Math.random() function that generates a random decimal no b/w 0 (inclusive) and 1 (exclusive).
- Thus, Math.random() can return 0 but never return a 1

Generate Random Whole no's with JS.

- Use math.random() to generate random decimal.
- Math.random() can never quite ~~re~~ return 1 and, because we're rounding down, it's impossible to actually get 20. This technique will give us a whole no b/w 0 and 19.

→ multiple random decimal by 20
→ use another f^n, Math.floor() to round no down to its nearest whole no.

```
Math.floor(Math.random()*20);
```

Generate random Whole no's within RANGE:

```
Math.floor(Math.random()*(max - min + 1)) + min.
```

(48) Use ~~Pre~~ ParseInt Function:

ParseInt() f^n parses a string and returns integer.

```
var a = parseInt("007");
```

- The above f^n converts string 007 to integer 7. If the first character in string can't be converted into a number, then it returns NaN.

- parseInt() f^n parses a string and returns an integer. It takes 2nd argument for radix, which specifies base of number in the string.

Radix → can be integer b/w 2 and 36.

```
parseInt(string, radix);
var a = parseInt("11", 2);
```

Radix variable says that 11 is in binary system, or base 2. This example converts the string 11 to an integer 3