/\*---------------------------- DIFFERENCES BETWEEN THE var AND let KEYWORD -----------------------\*/

// In ES-6, we can declare variables using "let" and "const" as well, along with "var" keyword.

// "let" doesn't allow to declare a variable with same name twice.

var catName = "Tom";

var catName = "Jerry";

console.log(catName); // No Error

let mouseName = "Tom";

//let mouseName = "Jerry"; // ERROR - TypeError: unknown: Duplicate declaration "mouseName"

// But we can still reset the value of the same variable more than once.

catName = "Jerry";

console.log(catName);

// Note that, even if first you declared a variable using "var" keyword and then we declare another variable with same name using "let" keyword, then also we'll get an error

var dogName = "Rocky";

//let dogName = "Brutus"; // ERROR - TypeError: unknown: Duplicate declaration "dogName"

/\* ----------------------------------------------USE STRICT---------------------------------------\*/

//TO DO

/\*------------------------------------ SCOPES OF var AND let KEYWORDS ----------------------------\*/

/\* ######## 1 \*\*\*\*\*\*\*\*\*\*/

function checkScope(){

"use strict";

var i = "function scope";

if(1){

var i = "block scope";

console.log("Block scope i is: ",i);

}

console.log("Function scope i is: ",i);

return i;

}

checkScope(); // OUTPUT - Block scope i is:,"block scope"

// Function scope i is:,"block scope"

// Here, variable 'i' acts as a Global Variable and we can change it's value from within the block as well even though we've explicitly DECLARED a new variable 'i' using keyword "var" inside the block.

/\*######### 2 ########### \*/

function checkScope(){

"use strict";

let i = "function scope";

let j = "2nd function scope";

if(1){

let i = "block scope";

j = "2nd block scope";

console.log("Block scope i is: ",i); //Block scope i is:,"block scope"

// 'i' here behaves as a different vaiable as it is declared in a different scope. Thus, declaring variables with same name using "let" keyword in different scope is VALID, as both of them will behave as different variables.

console.log("Block scope j is: ",j); //Block scope j is:,"2nd block scope"

// Notice that although 'j' is declared in another scope (which is global to this scope, i.e., we can use 'j' in this block as well), we can still modify it's value as the same variable.

}

console.log("Function scope i is: ",i); // OUTPUT -Function scope i is:,"function scope"

//This is because the variable 'i' declared within the block acts as another variable and the value given to it won't be reflected in the function scope variable 'i' as both behave as different variables.

console.log("Function scope j is: ",j); // Function scope j is:,"2nd block scope"

// 'j' acts as the same variable inside the block as well because it was not re-declared inside the block, thus changes made inside the block are reflected outside as well.

return i;

}

checkScope();

/\*######### 3 ########### \*/

function checkScope(){

"use strict";

if(1){

var i = "var block scope";

let j = "let block scope";

console.log("Block scope i is: ",i); // OUTPUT - Block scope i is:,"var block scope"

console.log("Block scope j is: ",j); // OUTPUT - Block scope j is:,"let block scope"

}

console.log("Function scope i is: ",i); // OUTPUT - Function scope i is:,"var block scope"

// Notice that, variable 'i' is still accessible outside the block although it was declared withing the block. This is because of HOISTING. Refer to -

//console.log("Function scope j is: ",j); // OUTPUT - ReferenceError: j is not defined

// variable 'j' acts more like a local variable as it's not accessible outside the scope in which it was declared. Thus, most users prefer using "let" to declare variables as it's more intuitive.

return i;

}

checkScope();

/\*------------------------------- READ-ONLY VAIABLE WITH "const" KEYWORD--------------------------\*/

// "const" keyword has ALL the features of "let" and it is READ-ONLY

const PI = 3.14; // Note that const variables are declared in all-capital letters

//PI = 5; // ERROR- SyntaxError: unknown: "PI" is read-only

// THUS IT IS ADVISABLE TO USE "let" AND "const" KEYWORDS ONLY IN YOUR CODE

/\*---------------------------------- MUTATE AN ARRAY DECLARED WITH "const" -----------------------\*/

const s = [5,4,3];

//s = [7,8,9]; // ERROR - SyntaxError: unknown: "arr" is read-only

// But we can individually change each element of the const array

s[0] = 7;

s[1] = 8;

s[2] = 9;

console.log(s)

for(let i=0;i<3;++i){

s[i] = i+10;

}

console.log(s)

// Similarly we can do that with OBJECTS

const MATH\_CONST = {

PI : 3.14

};

var constant = MATH\_CONST.PI;

console.log(constant);

MATH\_CONST.PI = 5;

constant = MATH\_CONST.PI;

console.log(constant); // OUTPUT - 5. Value has been MUTATED

/\*---------------------------------------- PREVENT OBJECT MUTATION -------------------------------\*/

// We can use "Object.freeze()" to prevent Data Mutation in Objects

function Obj() {

"use strict";

const MATH\_CONSTANTS = {

ANOTHER\_PI : 3.14

};

Object.freeze(MATH\_CONSTANTS); // "Object.freeze(OBJECT\_NAME)" won't allow any MUTATIONS being made into the object

try {

MATH\_CONSTANTS.ANOTHER\_PI = 58; // We try to MUTATE the value of PI to 58

}

catch(ex){ // Error is caught

console.log(ex); // Error is Displayed

}

return MATH\_CONSTANTS.ANOTHER\_PI;

};

const CONST\_PI = Obj();

//console.log(CONST\_PI);

/\*------------------------ ERROR-HANDLING - TRY, CATCH, FINALLY AND THROW ------------------------\*/

// try and catch only handle RUNTIME ERROR not PARSE TIME/ COMPILE TIME ERROR, i.e., try and catch won't run if SYNTAX OF CODE is WRONG, this kind of error is known as PARSE-TIME/ COMPILE TIME ERROR

// "try" statement is used to test a block of code for errors. Thus, if we've any doubts whether a given block of code has errors in it or not then we put it inside the "try" statement. If any error is encountered then it automatically throws respective error.

// "catch" statement catches the error thrown by the "try" statement and inside "catch" block we can write how we want to deal with that type of error.

// "try" and "catch" statements are used so that our program doesn't terminate abruptly without telling us properly what has happened. Thus, we use "try" and "catch" to run the suspicious lines of code inside "try" and deal with the error using "catch". For e.g., if we don't know what type of error has occurred, we can just output the error using "catch" block, thus our program doesn't terminate abruptly without us knowing what error has occurred.

// "finally" statement let's us execute the code written inside it regardless of whether an error has been thrown by the "try" statement or not, or whether the "catch" statement has handled the error or not.

try{

console.log("Start of try statement ");

unicorn; // "unicorn" variable is not defined anywhere in code thus at this statement our try block will throw an error and the statements after this line of code will not be executed

console.log("This statement is never reached");

}

catch(err){

//console.log("Error has occurred: " + err.stack); // "err.stack" will Not work in all browser

console.log("Error has occurred: " + err); // OUTPUT - Error has occurred: ReferenceError: unicorn is not defined

// Here, "err" contains ReferenceError: unicorn is not defined

// "ReferenceError" is known as the name of error

// "unicorn is not defined" is known as the message of this error.

console.log("Error has occurred: " + err.name); // OUTPUT - Error has occurred: ReferenceError

console.log("Error has occurred: " + err.message); // OUTPUT - Error has occurred: unicorn is not defined

}

finally{

console.log("This line always runs"); // This line of code always runs whether or not error has been thrown or not

}

// "throw" statement is used to make CUSTOM ERRORS and throw them to the "catch" block.

let json = '{"age" : 30}'; // Data

try{

let user = JSON.parse(json);

if(!user.name){ // If there is no "name" property in our data then we throw an error

// We can throw a string, integer, float or anything but we can also throw predefined errors with their Messages changed. Like below, we've a new "SyntaxError" with our CUSTOM error Message

throw new SyntaxError("Incomplete data: no name"); // Name of ERROR is: "SyntaxError"

// Message of ERROR is: "Incomplete data: no name"

}

console.log(user.name)

}

catch(e){

console.log("Error Message: " + e.message); // OUTPUT - Error Message: Incomplete data: no name

console.log("Error Name : " + e.name); // OUTPUT - Error Name : SyntaxError

console.log("JSON Error: " + e); // JSON Error: SyntaxError: Incomplete data: no name

}

/\*------------------------------------ ARRAY ITERATION (8 METHODS) -------------------------------\*/

// forEach Loop

/\*######### 1 ######## \*/

const days = ['Mon','Tue','Wed','Thr','Fri','Sat','Sum'];

// We define an Anonymous Function and do some task in it. Then we use the array which is "days" here and use "forEach" loop, and then PASS this function in that loop as shown below.

// Thus, our function will be executed 'n' number of times, where 'n' is the number of elements in the array.

let func = function(){

console.log("This line will get printed the 7 number of times because 7 is the number of elements in array");

}

days.forEach(func);

/\*######### 2 ######## \*/

// Another shorter way of using a "forEach" loop on an array is by declaring and defining the function inside the "forEach" loop itself, as shown below.

// Notice that, our function takes two Arguments as input which are "day" and "index" (they're just variables, we can name them anything).

// The first argument ("day" here) gets the value stored at each index and the second argument("index" here) gets the index number of the element which is being visited.

// Always remember, the first argument gets the value of the element that is being visited and the second argument gets the index of the element that is being visited.

days.forEach(function(day,index){ // This kind of function which doesn't have a name and is passed is known as "CallBack".

console.log(`starts with ${index+1} -- ${day}`); // We can only use '$' inside " ` ` " (tilt) only, that's why we've used " ` ` ".

})

var name = "Ayush";

console.log("Name ${name}"); // OUTPUT - Name ${name} , here '$' sign can't be used.

// map

// It is used to perform some task on the elements of the array and store the result created in a new array.

const three = [1,2,3];

const doubled = three.map(function(item){

return item\*2;

});

console.log(doubled);

console.log(three);

// filter

// This is used to perform some check on the elements of the array,whether a condition is True or False and the elements that satisfy the condition gets stored in a new array.

const ints = [1,2,3];

const evens = ints.filter(function(item){

return item % 2 === 0; // All the elements of the array will be checked for this condition and those who satisfy will be returned to form a new array.

})

console.log(evens);

// reduce

//The reduce() method reduces the array to a single value.

//SYNTAX - array.reduce(function(total, currentValue), initialValue)

// See w3Schools for more info.

// total is value that is going to be passed to the next element and item is the current item's value.

// Initial value of "total" is also given in reduce(), here it is 0. If initial value is skipped then "total" variable will be equal to the first item's value in the array.

const sum = [1,2,3].reduce(function(total, item){

return total+item;

},0)

console.log(sum); // OUTPUT - 6

const sum\_2 = [1,2,3].reduce(function(total, item){

return total+item;

})

console.log(sum\_2); // OUTPUT - 6

// some

// The some() method checks if any of the elements in an array pass a test

// The some() method executes the function once for each element present in the array:

// If it finds an array element where the function returns a true value, some() returns true (and does not check the remaining values).

// Otherwise it returns false.

const hasNegativeNumbers = [1,2,3,-1,4,5].some(function(item){

return item<0;

});

console.log(hasNegativeNumbers); // OUTPUT - true

// every

// This method is just like "some()" but here All the elements need to satisfy the condition given in the "every()".

// Thus, true is returned if all the elements of the array satisfy the condition else false is returned.

const allPositiveNumbers = [1,2,3].every(function(item){

return item > 0;

});

console.log(allPositiveNumbers);

const notAllPositiveNumbers = [1,2,-3].every(function(item){

return item > 0;

});

console.log(notAllPositiveNumbers);

// find

// "find()" method returns the item that needs to be found in the given array.

const objects = [{id:'a'},{id:'b'},{id:'c'}];

const found = objects.find(function(item){

return item.id === 'b' ;

});

console.log(found); // OUTPUT - {id: "b"}

// If the item is not found in the array then "NULL" is returned

const anotherFound = objects.find(function(item){

return item.id === 'd' ;

});

console.log(anotherFound); // OUTPUT - null

//findIndex()

// "findIndex()" method searches for an item in the given array and returns it's index where the item is situated.

const foundIndex = objects.findIndex(function(item){

return item.id === 'b';

});

console.log(foundIndex); // OUTPUT - 1

// If the element is not found in the array then the index returned is "-1";

const anotherFoundIndex = objects.findIndex(function(item){

return item.id === 'd';

});

console.log(anotherFoundIndex); // OUTPUT - -1

/\*---------------------------------- ANONYMOUS FUNCTION & ARROW FUNCTION--------------------------\*/

// An anonymous function doesn't have any name, it is only assigned to a variable.

var magic = function() {

return new Date();

}

console.log(magic());

// Whenever there's an anonymous function we can convert it into an ARROW FUNCTION

var magic = () => { // Notice that, we removed they keyword "function" and put an ARROW (=>) after the brackets "( )"

return new Date();

};

console.log(magic());

// Notice that the function given above is an ARROW FUNCTION and if this function returns ONLY ONE value then we can shorten it further.

var magic = () => new Date(); // ARROW FUNCTION. Notice that we removed the "return" keyword and the curley braces "{ }" and written only the value that has to be returned after the arrow symbol.

console.log("This is magic: " + magic());

// Parameters can also be passed in an ARROW FUNCTION as well as shown below

var myConcat = (arr1,arr2) => arr1.concat(arr2); // Thus the format of an Arrow function is:

// (arguments) => (returned value);

console.log(myConcat([1,2] , [3,4,5])); // OUTPUT - [1,2,3,4,5]

// ARROW FUNCTIONS work well with Higher Order Array Methods (like forEach, map, filter etc.).

// These Higher Order Array Methods take another function as an argument.

// NOTE THAT - Whenever any function takes another function as an argument then that's a good time to use ARROW FUNCTION

// We've created a function, that takes an array as argument and returns the square of positive integer values from the array.

const realNumberArray = [4,5.6,-9.8,3.14,42,6,8.96,-2];

const squareList = (arr) => {

const squaredIntegers = arr.filter(num => Number.isInteger(num) && num > 0).map(x => x\*x);

return squaredIntegers;

};

const squaredIntegers = squareList(realNumberArray);

console.log(squaredIntegers);

/\*------------------------------------ DEFAULT VALUE IN A FUNCTION -------------------------------\*/

// Notice that we've used an ANONYMOUS FUNCTION here

const increment = (function(){

return function increment(number, value =1){ // Default value of variable "value" is 1

return number + value;

};

})();

console.log(increment(5,2));

console.log(increment(5)); // Default value 1 is used

/\*-------------------------------- REST OPERATOR WITH FUNCTION PARAMETERS ------------------------\*/

// Rest Operator is used to pass any number of arguments to a function.

const sum\_1 = (function(){

return function sum(x,y,z){

const args = [x,y,z];

return args.reduce((a,b) => a+b,0);

};

})();

console.log(sum\_1(3,2,3));

// In the above function, we passed 3 arguments and then made an array consisting of these three and used ".reduce()" method on that array to calculate the result.

// If we have variable number of arguments then we can use REST OPERATOR

// REST OPERATOR is denoted using 3 DOTS (...)

const add = (function(){

return function sum(...args){ // What this REST OPERATOR does is that whatever is passed in the "sum" function, no matter how many values, it makes an array consisting of these values and this array is known as "args" as shown. That's why we don't need to explicitly create an array and add these elements.

return args.reduce((a,b) => a+b,0);

};

})();

console.log(add(3,2,5,6)); // OUTPUT - 16, Notice that we can add any number of arguments now

/\*------------------------------------------ SPREAD OPERATOR -------------------------------------\*/

const arr1 = ["JAN","FEB","MAR","APR","MAY"];

let arr2;

(function(){

arr2 = arr1;

arr1[0] = "Potato";

})();

console.log(arr2);

// Notice that above, we had an array "arr1" and created a new array "arr2" and copied "arr1" to "arr2". If we changed an element of "arr1" then the same element of "arr2" will get changed.

// This is because, The data assigned to array "arr1" will be stored at some place in array and the array "arr1" will act as a pointer to that data, thus "arr1" and "arr2" both act as a REFERENCE VARIABLE to this data.

// Now since both the variables act as reference variable to same data, changes made in data through one variable will get reflected into another one.

// To avoid this, we use SPREAD OPERATOR

// SPREAD OPERATOR is represented similar to REST OPERATOR i.e., "[...]"

const arr3 = ["JAN","FEB","MAR","APR","MAY"];

let arr4;

(function(){

arr4 = [...arr3]; // This is the SPREAD OPERATOR with "[...]"

arr3[0] = "Potato";

})();

console.log(arr4); // OUTPUT - ["JAN", "FEB", "MAR", "APR", "MAY"]

/\*---------------------- DESTRUCTURING ASSIGNMENT TO ASSIGN VARIABLES FROM OBJECTS ---------------\*/

var voxel = {x:3.6 , y:7.4, z:6.54};

// Destructuring is like un-packing a tuple.

// If we had to copy each variable's value of object in some other variable the normal way of doing this would be

var x = voxel.x;

var y = voxel.y;

var z = voxel.z;

// This process is lenghty and not optimal, thus we use DESTRUCTURING as shown below.

// Syntax is {Objects\_Variable\_1 : Variable\_1, Objects\_Variable\_2 : Variable\_2,.....} = Objects\_Name

const {x:a, y:b , z:c} = voxel; // This is DESTRUCTURING, in this value of "x" is stored in variable "a" and value of "y" is stored in variable b and so on.

console.log(a,b,c); // OUPTUT - 3.6 , 7.4 , 6.54

// Another Example

const AVG\_TEMPERATURES = {

today : 77.5,

tomorrow : 79

};

function getTempOfTmrw(avgTemperatures){

"use strict";

const { tomorrow : tempOfTomorrow } = avgTemperatures;

return tempOfTomorrow;

}

console.log(getTempOfTmrw(AVG\_TEMPERATURES)); // OUTPUT - 79

/\*----------------------------- ASSIGNING VARIABLES FROM NESTED OBJECTS --------------------------\*/

// Nested Object

const LOCAL\_FORECAST = {

today : { min:72 , max:83 },

tomorrow : { min:73.3 , max:84.6 }

};

// Syntax is

//{Nested\_Obj\_1 : {Obj\_Variable\_Name\_1 : Variable\_Name , Obj\_Variable\_Name\_1 : Variable\_Name,...} , Nested\_Obj\_2 : {Obj\_Variable\_Name\_1 : Variable\_Name , Obj\_Variable\_Name\_1 : Variable\_Name,...},.....}

function getMaxOfTmrw(forecast){

"use strict";

// Here we've accessed "max" variable of "tomorrow" object inside LOCAL\_FORECAST object

const { tomorrow : {max : maxOfTomorrow }} = forecast; // In this way we can get any variable in a nested object using DESTRUCTURING

return maxOfTomorrow;

}

console.log(getMaxOfTmrw(LOCAL\_FORECAST)); // OUTPUT - 84.6

/\*---------------------- DESTRUCTURING ASSIGNMENT TO ASSIGN VARIABLES FROM ARRAYS ----------------\*/

// We can assign an element of an array to a variable using DESTRUCTURING

// If we don't use DESTRUCTURING we do our task like below

const arr = [1,2,3,4,5];

var A = arr[0];

var B = arr[0];

console.log(A,B);

// But using DESTRUCTURING, we do our task as shown below

var [x,y,,z] = [1,2,3,4,5]; // Notice we can only DESTRUCTURE or unpack an array in the order in which it's elements are present, i.e., "x" will contain 1 only, we can't specify any random random element from array to be stored in "x" as it was with objects.

// We can skip Un-Wanted elements, i.e., elements that we don't want to be stored in a variable using just "," i.e., if we want to skip an element we don't put any variable name in it's turn and just put comma (,) as shown above, we've skipped 3 from the array and "z" contains 4 not 3.

console.log(x,y,z); // OUTPUT - 1,2,4

// We can also do SWAPPING using DESTRUCTURING

var A = 8, B= 6;

(() => {

"use strict";

[A,B] = [B,A]

})();

console.log(A,B); // OUTPUT - 6,8

/\*-------------------------- DESTRUCTURING ASSIGNMENT USING REST OPERATOR ------------------------\*/

// We can use the REST OPERATOR to assign the elements of an array to another array

const source = [1,2,3,4,5,6,7,8,9,10];

function removeFirstTwo(list){

const [, , ...arr] = list; // Here, we've stored all the elements of array "source" except first two elements into array "arr". To skip an element we used blank commas (,). Thus, it says first two elements should not be assigned to any variable and after that all the elements should be assigned to array "arr".

const [A,B,...another\_arr] = list; // This is another way to storing all elements but first two. We've stored first two elements in variables "A" and "B" and rest elements in array "another\_arr".

return arr;

}

const Arr = removeFirstTwo(source);

console.log(Arr); // OUTPUT - [3, 4, 5, 6, 7, 8, 9, 10]

console.log(source); // OUTPUT - [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

/\*------------- DESTRUCTURING ASSIGNMENT TO PASS AN OBJECT AS A FUNCTION'S PARAMETERS ------------\*/

// Object

const stats = {

max : 56.78,

standard\_deviation : 4.34,

median : 34.54,

mode : 23.87,

min : -0.75,

average : 35.85

};

// To use the values of variables of object, we pass in whole object to the function and using dot/square bracket notation we access elements of objects to perform the task as shown below.

const half = (function() {

return function half(obj){

return (obj.max + obj.min) / 2.0;

};

})();

console.log(stats);

console.log(half(stats)); // OUTPUT - 28.015

// Now, we can only pass the values of variables in the objects that are required using DESTRUCTURING.

// This is mainly used in API calling.

const another\_half = (function(){

return function another\_half({max , min , average}){

return ((max + min)/2.0) + average;

};

})();

console.log(stats); // OUTPUT - {max: 56.78, standard\_deviation: 4.34, median: 34.54, mode: 23.87, min: -0.75, average: 35.85}

console.log(another\_half(stats)); // OUTPUT - 63.865

/\*----------------------------- CREATING STRINGS USING TEMPLATE LITERALS -------------------------\*/

// Template Literals are special type of strings that make creation of complex strings easy.

// Template Literals are made using Back-Tick (` `). The string of Template Literal goes inside this Back-Tick.

const person = {

name : "Ayush Kumar Tripathi",

age : 21

};

const greeting = `Hello, My Name is ${person.name}!

I am ${person.age} years old.`;

console.log(greeting);

/\* Template Literal Strings have 3 main advantages:

(1) We can create Multi-Line strings without using "console.log()" again and again. For e.g., in example given above, "Hello, My Name is ${person.name}!" will be printed in one line and then we've used new-line to write "I am ${person.age} years old.". Thus, both these will be printed in different lines. Here, in SCRIMBA it's not printed like that, but in browser it will.

(2) We can add Single-Quotes (' ') and Double-Quotes (" ") without escaping them.

(3) We can put VARIABLES inside these Template Literal Strings using Dollar ($) sign and Curley Braces "{ }". We can put HTML or JavaScript code inside "{}" to processed and added to the string as shown above.

\*/

/\*--------------------- CONCISE OBJECT LITERAL DECLARATIONS USING SIMPLE FIELDS ------------------\*/

// In the below given function, we take some values as input and create an object out of it and return it.

// If the name of the variable in the object is same as name of the variable of function that contains input values as shown below, i.e., Name of variable of function containing values has same name as the name of variables of object to which values are assigned then we can use this technique.

const createPerson = (name,age, gender) => {

return {

name : name,

age : age,

gender : gender

};

};

console.log(createPerson("Ayush Kumar Tripathi" , 21 , "male"));

// Now, using Object Literals we can do it as follows:

const createAnotherPerson = (name,age,gender) => ({name,age,gender}); // It does the exactly same thing

console.log(createAnotherPerson("S.K.Tiwari" , 46 , "male"));

/\*------------------------------------- CONCISE DECLARATIVE FUNCTIONS ----------------------------\*/

// We can add a function inside the object as a Method, traditional way of doing this is

const bicycle = {

gear : 2,

setGear : function(newGear){

"use strict";

this.gear = newGear;

}

};

bicycle.setGear(5);

console.log(bicycle.gear);

// We can do this without using colon (:) and "function" keyword

const anotherBicycle = {

gear : 2,

setGear(newGear){ // Notice that the colon (:) and "function" keyword has been removed

"use strict";

this.gear = newGear;

}

};

anotherBicycle.setGear(10);

console.log(anotherBicycle.gear); // OUTPUT - 10

/\*---------------------------- CLASS SYNTAX TO DEFINE A CONSTRUCTOR FUNCTION ---------------------\*/

/\*----------------------- getters AND setters TO CONTROL ACCESS TO AN OBJECT ---------------------\*/

/\*----------------------------- DIFFERENCE BETWEEN IMPORT AND REQUIRE ----------------------------\*/

// Earlier, "require" function was used to import different functions and codes from another files

// Now, we've got "Import" and "Export", we can export the code in one file and import that file into some another file. It also allows us to import only certain Functions or variables from a file.

// To EXPORT any function, one way of syntax used is

// export function()

// write the function as it is and write the keyword "export" infront of it.

// To IMPORT any function, the syntax used is

// import { function\_name } from "Address of File from which function is import w.r.t this/current file"

// "./file\_name", here, "./" represents that in the same directory w.r.t current file, the file from which we are importing is present.

// Note that "./" represents the same directory w.r.t the current file.

// Note that, we don't write "file\_name.js" as it is evident that the file being import will be .js type.

import { capitalizeString } from "./string\_function" // import from "string\_function" file

const cap = capitalizeString("hello!");

console.log(cap);

/\*----------------------------------- EXPORT TO REUSE A CODE BLOCK -------------------------------\*/

// Another way of syntax used to export different functions and variables from a file are as follows

/\*

const capitalizeString = (string) => {

return string.charAt(0).toUpperCase() + string.slice(1);

}

export { capitalizeString }; // Exporting Function, Notice that previously in file "string\_function.js", we used "export" keyword infront of our function that was to be exported, but here, we export our function after defining it.

export const foo = "bar"; // Exporting a variable

export const bar = "foo"; // Exporting another Variable

\*/

/\*------------------------------ USE \* TO IMPORT EVERYTHING FROM A FILE --------------------------\*/

// We can import everything a file exports individually or we can import everything at once using "\*" operator

// SYNTAX would be

// import \* as Object\_Name from "Address of File from which function is import w.r.t this/current file"

// Notice that, once we import everything, then we've to store it in an Object, thus we've to specify an Object by giving any name to it then we've to tell the address of the file from which we're importing it

// For example

/\*

import \* as capitalizeStrings from "./capitalize\_strings"

\*/

// Here, Object name is "capitalizeStrings" and everything imported from file "capitalize\_strings" will be stored in this object

/\*------------------------------ EXPORT FALLBACK WITH EXPORT DEFAULT -----------------------------\*/

//When we import we have to specify a name and import like:

//import MyDefaultComponent from "./MyDefaultExport";

// We've to specify the name of the function that we want to import

// If we want to give user defined name to the import then we've to make sure that the export is a DEFAULT EXPORT

//The naming of import is completely independent in default export and we can use any name we like

// NOTE that there can be ONLY ONE default export function/variable/class/object etc.

// For example

/\*

export default function subtract(x,y) {return x-y;}

OR

let myExport = function subtract(x,y) {return x-y;}

export default myExport;

\*/

// NOTE that, named exports can be as many as we like but there is only one default export per file.

/\*----------------------------------- IMPORTING A DEFAULT EXPORT --------------------------------\*/

/\*

import subtract from "math\_functions";

subtract(7,4);

\*/

// Notice that, subtract function was a default export thus we didn't use curley braces "{ }" around the function name while importing. Other than that, everything else is same as importing a named export

/\*---------------------- IMMEDIATELY INVOKED FUNCTION EXPRESSION (IIFE)(NOT ES-6)-----------------\*/

// As soon as this function below is written and saved, it runs in the console, that's why it's known as "Immediately Invoked"

/\*

(function(){

console.log("Hello World");

})();

// We can also give our IIFE function a name i.e., store it inside a variable

var favNumber;

(favNumber = function(num = 3){ // 3 is default value for num

console.log("My favorite number is " + num);

})();

favNumber(5); // We can also explicitly pass a value for num in our IIFE

// IIFE are also used to create CLOSURES

var a = 2;

(function(){

var a = 3;

console.log(a); // outputs 3

})();

console.log(a); // outputs 2, as it cannot access the 'a' with value 3 inside IIFE, thus IIFE are used to create closures and to avoid declaring variables in global scope as is the case with "var".

// This is what used to be done before ES-6, but now in ES-6, we've "let" and "const" whose life-time is withing their scope only.

\*/

/\*--------------------------------------------- CLOSURES -----------------------------------------\*/