//SPECIAL DATA TYPES

/\* BigInt - This data type is used to store numbers which are above the limitation of the Number data type.It can store large integers and is represented by adding “n” to an integer literal.

Example : \*/

const previouslyMaxSafeInteger = 9007199254740991n

const alsoHuge = BigInt(9007199254740991)

// ↪ 9007199254740991n

/\* Symbol - It is a new data type introduced in the ES6 version of javascript.It is used to store an anonymous and unique value.

Example : \*/

var symbol1 = Symbol('symbol');

/\* By specification, object property keys may be either of string type, or of symbol type.Not numbers, not booleans, only strings or symbols, these two types. A “symbol” represents a unique identifier. A value of this type can be created using Symbol(): \*/

let id = Symbol();

/\* Upon creation, we can give symbol a description(also called a symbol name), mostly useful for debugging purposes: \*/

let id = Symbol("id");

/\* Symbols are guaranteed to be unique.Even if we create many symbols with the same description, they are different values.The description is just a label that doesn’t affect anything. Most values in JavaScript support implicit conversion to a string.For instance, we can alert almost any value, and it will work.Symbols are special. They don’t auto - convert. Any data type that is not a primitive data type, is of Object type in javascript.\*/

//HOISTING

/\*Hoisting is the default behavior of javascript where all the variable and function declarations are moved on top. This means that irrespective of where the variables and functions are declared, they are moved on top of the scope.The scope can be both local and global. Variable initializations are not hoisted, only variable declarations are hoisted. To avoid hoisting, you can run javascript in strict mode by using “use strict” on top of the code: \*/

"use strict";

x = 23; // Gives an error since 'x' is not declared

var x;

//STRING MANIPULATION

/\* In addition of 2 variables, say a and b if both are numbers only then actual addition (numeric) occurs. If both or any one of them are strings, then string concatenation occurs. In subtraction of two vars, actual subtraction as of numbers is done regardless of whether either of a and b are string or numbers. In case if either or both of a, b is bool then actual addition / subtraction takes place considering its value as 1 / 0.\*/

//COERCION

/\*Truthy values are those which will be converted(coerced) to true. Falsy values are those which will be converted to false. All values except 0, 0n, -0, “”, null, undefined and NaN are truthy values. While using the ‘==’ operator, coercion takes place. The ‘==’ operator, converts both the operands to the same type and then compares them.

Example: \*/

var a = 12;

var b = "12";

a == b // Returns true

//NAN

/\* NaN property represents “Not - a - Number” value.It indicates a value which is not a legal number. typeof of a NaN will return a Number. To check if a value is NaN, we use the isNaN() function. isNaN() function converts the given value to a Number type, and then equates to NaN. \*/

isNaN("Hello") // Returns true

isNaN(345) // Returns false

isNaN('1') // Returns false, since '1' is converted to Number type which results in 0 ( a number)

isNaN(true) // Returns false, since true converted to Number type results in 1 ( a number)

isNaN(false) // Returns false

isNaN(undefined) // Returns true

//PASSING BY REFERENCE

/\* Suppose a is an argument passed to a function. If a is primitive, then a copy of 'a' is received. While on the other hand if a is non primitive in nature, a reference to is received. But if the function is‘nt tweaking the internal components(obj prop, array element) of the non primitive argument a, but rather altering the arg or its value itself(reassigning etc) then the reference is neglected and the the arg is treated as a copy. Other than passing parameters to functions, same is the case with assigning variables. \*/

//IMMDIATELY INVOKED FUNCTIONS

/\*An Immediately Invoked Function is a function that runs as soon as it is defined. Syntax of IIFE: \*/

(function () {

console.log('kk');

})(); //prints kk

//HIGHER ORDER FUNCTIONS

/\* Functions that operate on other functions, either by taking them as arguments or by returning them, are called higher - order functions. Higher order functions are a result of functions being first - class citizens in javascript. Examples of higher order functions: \*/

function higherOrder(fn) {

fn();

}

higherOrder(function () { console.log("Hello world") });

function higherOrder2() {

return function () {

return "Do something";

}

}

var x = higherOrder2();

x() // Returns "Do something"

//THIS

/\* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this \*/

//FUNCTION FUNCTIONS

/\* The bind() method creates a new function that, when called, has its this keyword refer to the provided object, with a given sequence of arguments preceding the obj when the new function is called. The bind func works only once. Calling bind() on a func that is itself a return of bind() is allowed but doesn't work. The this of the func under concern will still refer to the scope it was originally binded to. Also, bind, call and apply do not work if the function is declared using arrow operator. \*/

const module = {

x: 42,

getX: function () {

return this.x;

}

};

const unboundGetX = module.getX;

console.log(unboundGetX()); // The function gets invoked at the global scope

// expected output: undefined

const boundGetX = unboundGetX.bind(module);

console.log(boundGetX());

// expected output: 42

let o = {

a: 'a',

b: function () {

console.log(this.a);

}

};

function f(c) { c(); };

let tmp = o.b;

tmp();//undefined

tmp = tmp.bind(o);

tmp();//'a'

/\* obj is the scope of 'this' in function binded to. \*/

let obj = {

name: 'johnny'

}

//call

func.call(obj, 'straight')

//apply takes in an array of args rather than many args and maps 'em with parameters

func.apply(obj, ['gay']);

//bind returns a func which when called will actually call the original function

(func.bind(obj, 'asexual'))();

//CURRYING

/\* Currying is an advanced technique to transform a function of arguments n, to n functions of one or less arguments. Example of a curried function: \*/

function add(a) {

return function (b) {

return a + b;

}

}

add(3)(4)

/\* For Example, if we have a function f(a, b) , then the function after currying, will be transformed to f(a)(b). By using the currying technique, we do not change the functionality of a function, we just change the way it is invoked. Let’s see currying in action: \*/

function multiply(a, b) {

return a \* b;

}

function currying(fn) {

return function (a) {

return function (b) {

return fn(a, b);

}

}

}

var curriedMultiply = currying(multiply);

multiply(4, 3); // Returns 12

curriedMultiply(4)(3); // Also returns 12

/\* As one can see in the code above, we have transformed the function multiply(a, b) to a function curriedMultiply , which takes in one parameter at a time. \*/

//COSURE

let x = () => {

let y = '4';

return () => {

console.log(y);

}

}

x()();

/\* Closure is a prop of js by virtue of which a reference to the lexical scope of a function is attached to it when the func is defined. This scope consists of all the variables required by the func and not defined within it's own scope.

x(), instead of destroying the value of y after execution, saves the value in the memory for further reference. This is the reason why the returning function is able to use the variable declared in the outer scope even after the function is already executed. \*/

//PROTOTYPE

/\* https://javascript.info/prototypes

suppose an obj o={a:'a',f:function(){return this.a},g:function(){this.a='A'}}

another obj m={...somePropOwn,\_\_proto\_\_:o}

Calling a getter func such as f on m will return parent's value of key found through protypical inheritance, ie val of a from o. But calling a setter func such as g on m will have 'this' binded to child obj and thus affects contents of its own and not of the parent/prototype. \*/

let arr = [4, 'h'];

arr.push(false);

/\* Object has a property(obj) - prototype. Any obj that we create(array, js obj, etc) inherits all the properties from this global Object's property, prototype obj. But this obj that we create also has a protype of its own, possibly containing props. We can add to the props of this proto. So when we access a property on an obj which possibly we created no failure is resulted unless this prop that we access is neither present in the obj's proto (or any of the protos that this obj inherits from) nor have we defined it ourselves in the obj. For example, the push on arr works because even though no such function has been defined by us, but its present in the proto of Array obj. \*/

//ARROW

/\* By general definition, the this keyword always refers to the object that is calling the function. In the arrow functions, there is no binding of the this keyword. The this keyword inside an arrow function, does not refer to the object calling it. It rather inherits its value from the parent scope. \*/

//REST AND SPREAD

/\* Rest provides an improved way of handling parameters of a function. Using the rest parameter syntax, we can create functions that can take a variable number of arguments. Any number of arguments will be converted into an array using the rest parameter. It also helps in extracting all or some parts of the arguments. Rest parameter can be used by applying three dots (...) before the parameters. \*/

function extractingArgs(...args) {

return args[1];

}

// extractingArgs(8,9,1); // Returns 9

function addAllArgs(...args) {

let sumOfArgs = 0;

let i = 0;

while (i < args.length) {

sumOfArgs += args[i];

i++;

}

return sumOfArgs;

}

addAllArgs(6, 5, 7, 99); // Returns 117

addAllArgs(1, 3, 4); // Returns 8

// Rest parameter should always be used at the last parameter of a function:

// Incorrect way to use rest parameter

function randomFunc(a, ...args, c) {

//Do something

}

// Correct way to use rest parameter

function randomFunc2(a, b, ...args) {

//Do something

}

//GENERATOR`

/\* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Generator \*/

//CLASS

/\* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes \*/

//Object destructuring:

const classDetails = {

strength: 78,

benches: 39,

blackBoard: 1

}

const { strength: classStrength, benches: classBenches, blackBoard: classBlackBoard } = classDetails;

console.log(classStrength); // Outputs 78

console.log(classBenches); // Outputs 39

console.log(classBlackBoard); // Outputs 1

/\* As one can see, using object destructuring we have extracted all the elements inside an object in one line of code. If we want our new variable to have the same name as the property of an object we can remove the colon: \*/

const { strength: strength } = classDetails;

// The above line of code can be written as:

const { strength } = classDetails;

//Array destructuring:

/\* The same example using object destructuring: \*/

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

const [first, second, third, fourth] = arr;

console.log(first); // Outputs 1

console.log(second); // Outputs 2

//TEMPORAL DEAD ZONE

/\* Temporal Dead Zone is a behaviour that occurs with variables declared using let and const keywords.

It is a behaviour where we try to access a variable before it is initialized. \*/