

**NOTES**

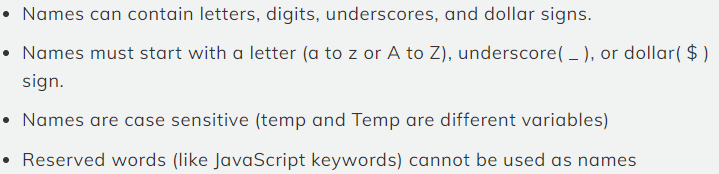
@ashish003

Introduction

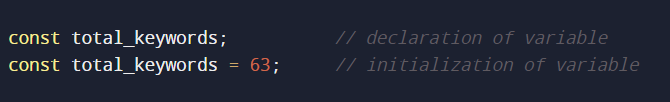
* JavaScript is an interpreted programming language and the programs in this language called scripting language used on the client side and server side.
* It was developed by Brendan Eich in 1995 and became an ECMA standard in 1997.
* Pros: Interpreted language, All browser support, Minimal syntax, Platform Independence
* Cons: weakly typed language, no support of multithreading and multiprocessing (i.e., single-threaded language), read and write of files not allowed.

Variable and Identifiers

* Variables are the name of the memory space where we store our data. which can be any type due to JavaScript's weakly typed language.
* There are four ways to declare variables in JavaScript, automatically, var, let, and const.
* Identifier is the name of the variable or function that we can identify uniquely to access in their scope.

The rules to give a unique name are: 

* Example:



auto global, var, let, and const

* In JavaScript we can declare variables without using any reserved keyword like var, let, or const this concept is called an auto-global declaration.

example: my\_var = 100;

* var is a reserved keyword in JavaScript used for declaring the variables that denote data can vary.

example: var my\_var = 100;

Note: The above two ways of declaring variables are not recommended due to of behavior of scope and security issues which we discuss later.

* let is another reserved keyword in JavaScript used for declaring the variable that also denotes data can vary. It is recommended instead of var.

example: let my\_var = 100;

* const is also a reserved keyword in JavaScript used for initializing the variable at the time of declaration and const denotes data (like name, date of birth) cannot vary at any point.

example: const date\_of\_birth = “07/04/1998”;

Note: The above two ways of declaring a variable are always recommended but always try to declare a variable using const if possible, which makes data secure.

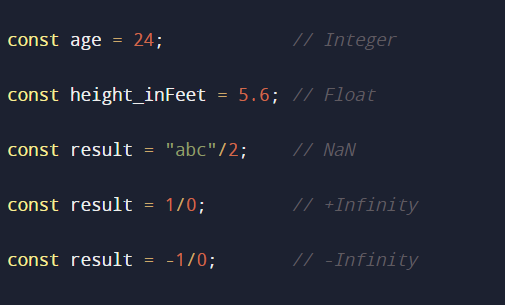
Datatypes

* Datatype means the type of data that we going to store in our variable and due to the dynamic nature of JavaScript it allows us to store any datatype in the variable.
* There are 7 primitive datatypes and 2 non-primitive datatypes.
* Primitive: Number, BigInt, String, Boolean, Undefined, Null, Symbol
* Non-Primitive: Object, function

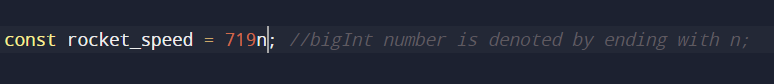
Note: primitive datatypes are immutable and take memory in the stack whereas non-primitive datatypes are mutable and take memory in the heap and reference in the stack.

Primitive datatypes

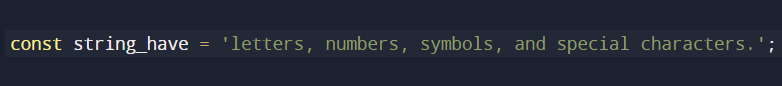
* Number datatype represents a variable whose value is either an integer, float, or JavaScript special numeric value (NaN, +Infinity, -Infinity).



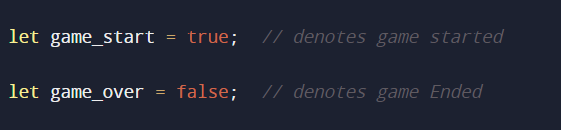
* BigInt datatype is introduced in ECMAScript 2020 and is designed to represent arbitrary precision integers means very large numbers without losing precision.



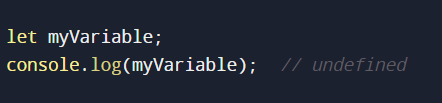
* String datatype represents the sequence of characters inside single quotes, double quotes, or in backticks.



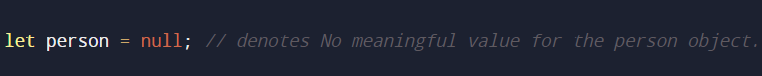
* Boolean datatype represents the two values i.e., true or false. It is used for conditional testing.



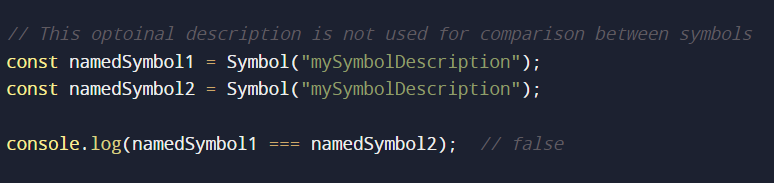
* Undefined datatype that represents the absence of value, and variables that have been declared but not assigned a value automatically have the value ‘undefined’.



* Null is a special value that represents the intentional absence of any object value. Unlike `undefined` which is often the default value of an uninitialized variable, `null` is explicitly assigned to indicate the absence of value.

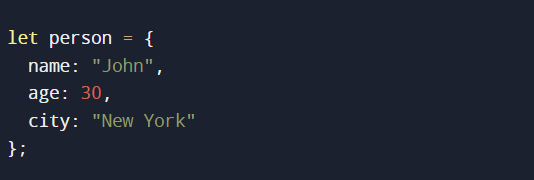


* Symbol datatype is used to provide a way to create unique and immutable identifiers. Symbols are often used to create private or hidden properties on objects and avoid naming collisions in scenarios where string-based property names might clash.

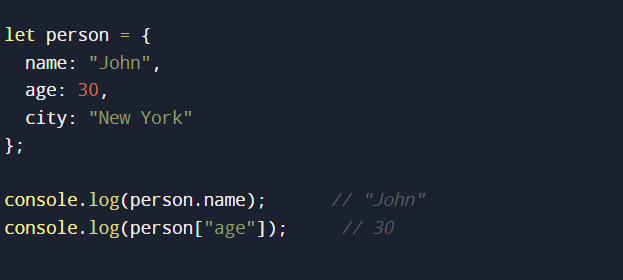


Non-primitive datatypes

* Object datatype is a complex datatype that allows you to group related data and code into a single unit. It represents the data in key-value pairs, where keys are strings or symbols, and the value can be a valid JavaScript datatype, including other objects.



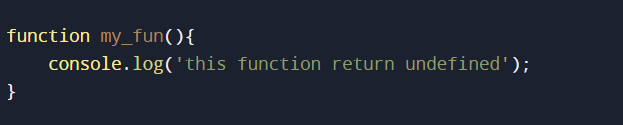
object value can be accessed using dot notation or square bracket notation example:



object datatype is further divided in JavaScript i.e., Array object, Date object, RegExp object, which we discuss later.

* Function is also the type of object and has properties and methods like other objects. functions are used to define reusable blocks of code that can be invoked to solve specific tasks.

when we see the type of `function` using the `typeof` operator shows the string function that is why I consider this function object as a second non-primitive datatype.



Note: the above data types can be verified by using the `typeOf` operator

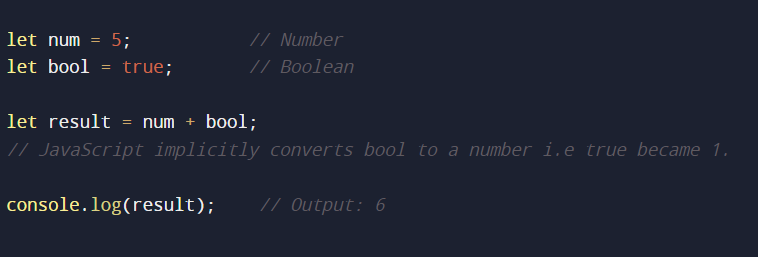
Operators

* Operators are some reserve fundamental symbols used to perform specific mathematical and logical operations on operands.
* There are many operators in JavaScript, and some important ones are on [geeksForGeeks](https://www.geeksforgeeks.org/javascript-operators-reference/)
* Remember: [Operator precedence](https://www.geeksforgeeks.org/operator-precedence-in-javascript/)

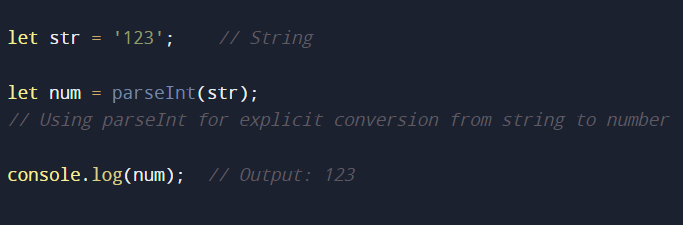
Implicit and Explicit conversion

* Implicit conversion, also known as coercion, happens automatically when JavaScript converts one data type to another without the programmer explicitly requesting the conversion.

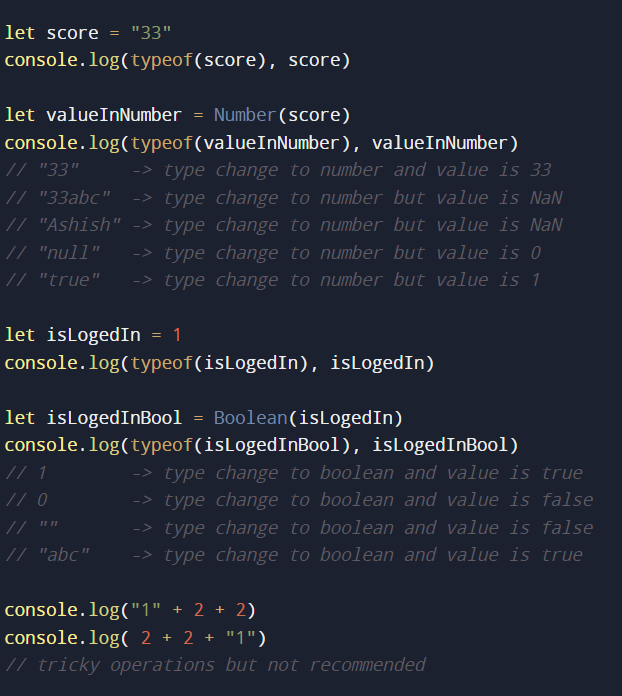
This often occurs in situations where different data types are involved in an operation.



* Explicit conversion, also known as type casting, occurs when the programmer explicitly converts a value from one type to another. This is done using built-in functions or methods.

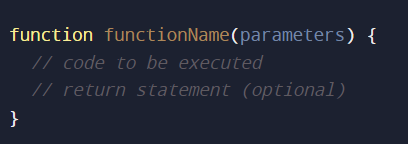


Example:



Functions

* Functions are blocks of reusable code that perform a specific task, here is a basic syntax for creating a function:



function: keyword used to declare a function.

functionName: is the identifier used to call the function later.

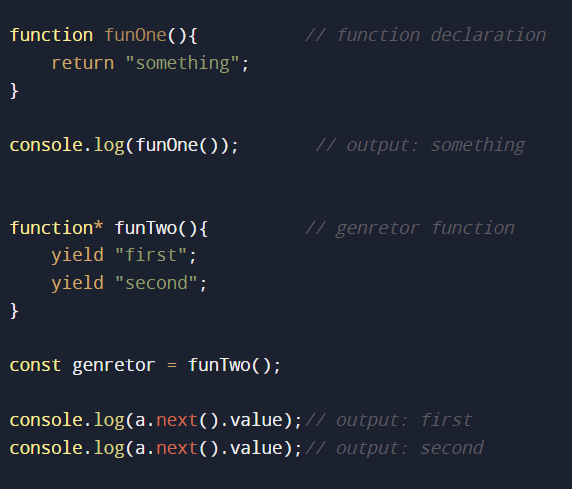
parameters: input values that the function can accept.

return keyword used to return the value explicitly by default function return undefined.

* Default parameter is the value that we assign to the formal argument to handle the absence of an actual argument.

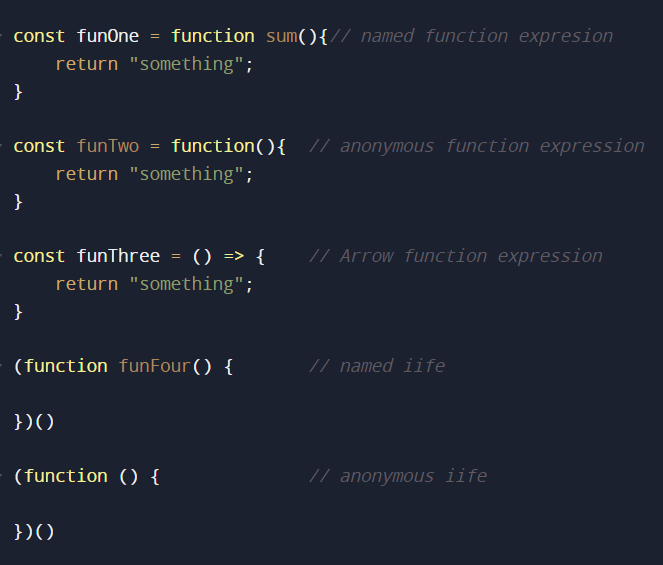


* Actual arguments are values that we pass to the function call and Formal arguments are the values that we take in function definition.
* Whereas, In JavaScript function has an arguments keyword or property that we can use to get parameters from an Array.
* Functions are of two types Function declaration and Function expression.
* Function declaration: if the first word or token of a written function is the function keyword then it is called function declaration. example:



* Function expression: if the first word or token of a written function is not a function keyword then it is called the function expression, it is of three types name function expression, anonymous function expression, and iife (immediately invoked function expression).

example:



* Why do we need function expression?

-- Because JavaScript is heavily influenced by a functional-oriented paradigm shows that we can store the function, pass the function, and return the function.

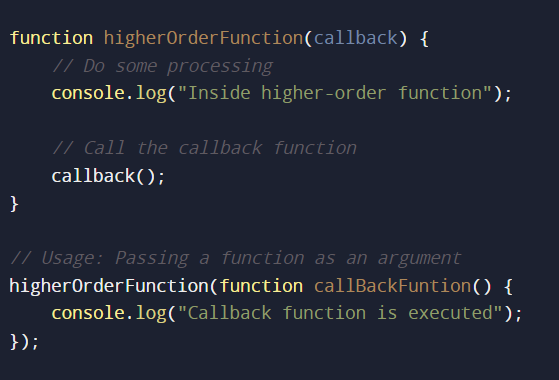
Due to this requirement specification, we need function expression.

* Difference between function declaration and function expression.?

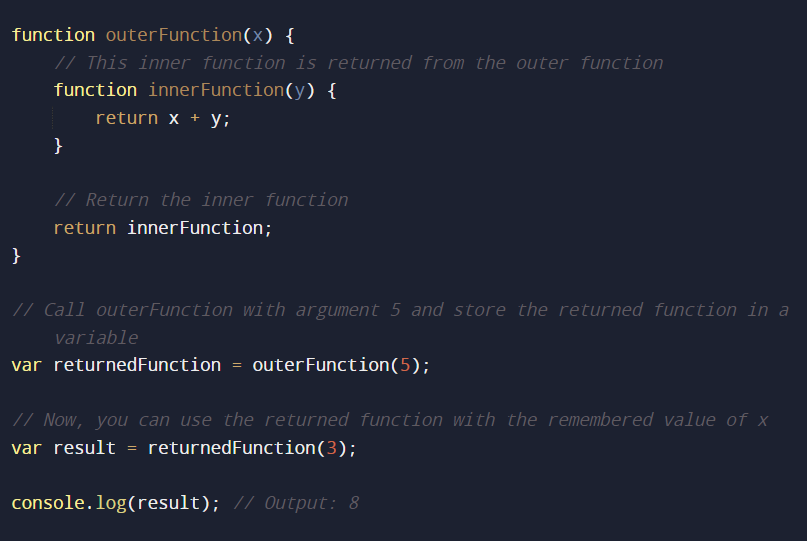
-- The function declaration is the formal declaration that is hoisted before the execution of the code and the assignment of the variables and function expression are not hoisted.

Higher order functions and call-back function

* Higher order function is a function that takes one or more functions as a formal argument or returns a function is known as Higher order function.
* Call-back function is a function that is passed as an actual argument to the higher order function which is executed after the operation or at a later time. known as the call-back function.



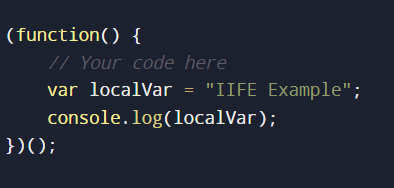
* Returning the function from another function is a powerful and flexible concept in JavaScript. When a function returns another function, the returned function can "remember" the scope in which it was created. This behavior is known as a closure. Here's a simple example:



iife

IIFE (Immediately Invoked Function Expression) is a common JavaScript pattern that involves defining and executing a function immediately after its creation. This pattern is often used to create a private scope for variables to avoid polluting the global namespace.

here's a basic example of an iife:



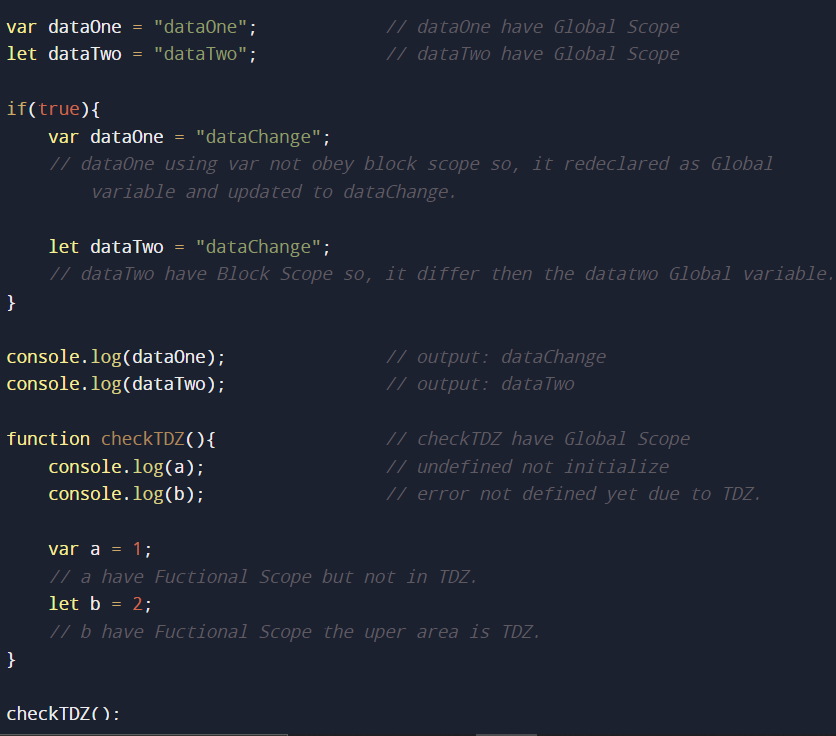
Scope

* Scope in programming means a defined area of variable or data where we can access them.
* There are three types of scope in JavaScript: Global, Functional, and Block scope.
* JavaScript runs programs in one go but actually in two phases i.e., phase one (compilation) and phase two (interpretation).
* And in phase one JavaScript decides the scope of the formally declared variable to their named scope (lexical Scoping), checks syntax errors, etc.
* And in phase two JavaScript assigns the value to the variable and then executes the code.



Note: var does not support the block scope.

* Block Scope {} comes into the picture due to let and const variables in ES6 and also the concept of TDZ (temporal dead zone). means variables in that zone are not reachable.

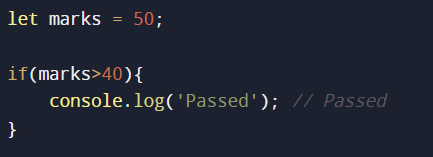


Hoisting

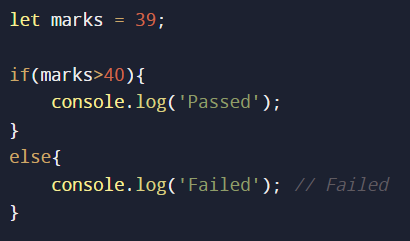
* Hoisting is a consequence due to which variables and functions are available before their declaration. and it happens because of Lexical Scoping.

Control flow

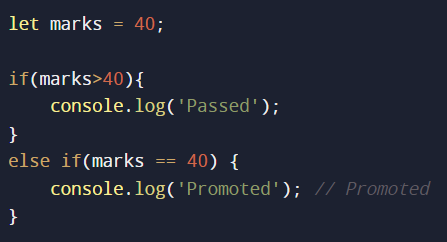
* Conditional statements are used to run different sets of statements according to the conditions**.** In JavaScript, we have the following conditional statements:
* Use**if** to specify a block of code to be executed if a specified condition is true.



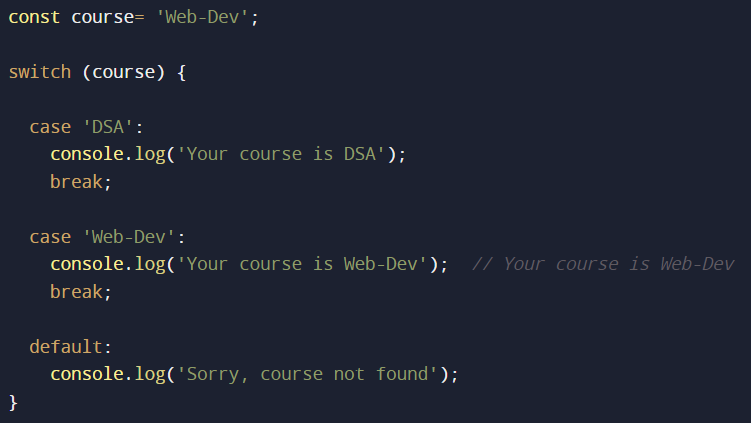
* Use **else**to specify a block of code to be executed if the same condition is false.



* Use **else if** to specify a new condition to test, if the first condition is false.



* Use the **switch**to specify many alternative blocks of code to be executed.



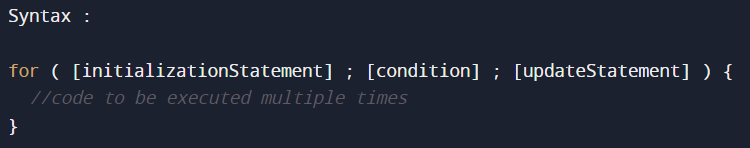
Note: If you omit the break statement, the following case will be executed even if the evaluation does not match the case.

Work Flow of Switch Statements:

* The switch expression is evaluated once.
* The value of the expression is compared with the values of each case.
* If there is a match, the associated block of code is executed.
* If there is no match, the default code block is executed.

Loops

* Loops are used to do something repeatedly. You need to print 'n' numbers, then you can use a loop to do so.
* Types of Loops in JavaScript are: for, while, do-while, for in, and for of.
* A for loop is used to repeat some code until the condition evaluates to false.

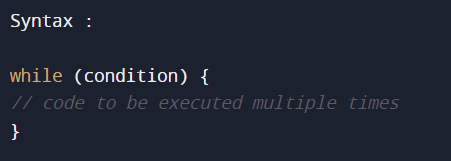


Key points are:

* The initialization statement is used to initialize loop counters.
* The condition is the expression that is evaluated to a Boolean value **true**or **false.**
* The update statement is used to update the loop counters.



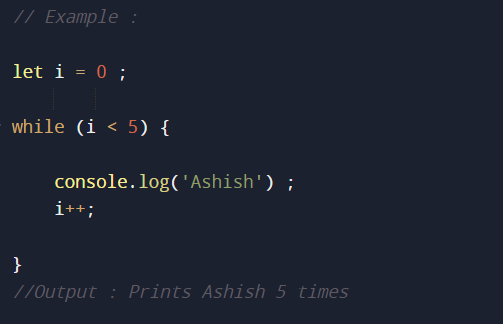
* The while loop statement executes the statements until the condition is not **false**.



First, the condition is evaluated, and if it is true, then the

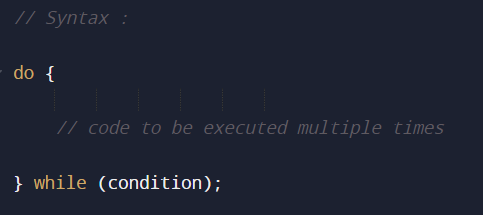
statements are executed, and the condition is tested again.

The execution stops when the condition returns false.



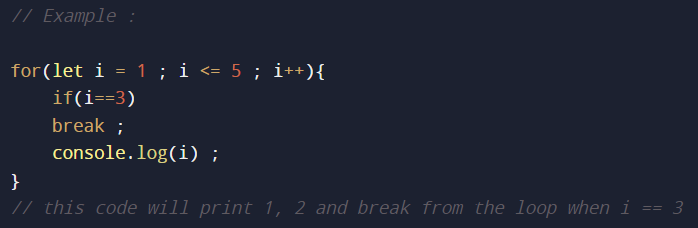
* The do-while loop is similar to the while loop, except that the

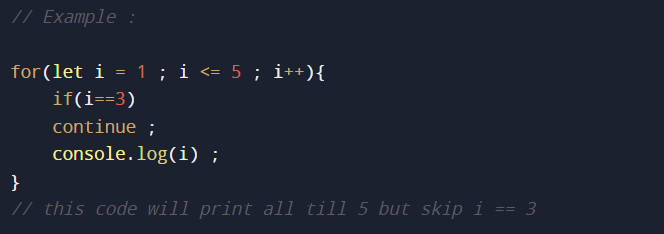
statements are executed at least once





* The for-in and for-of loops are most used in Array and Object so we discuss them later.
* There are two keywords which used often i.e., break and continue to manipulate the loop.





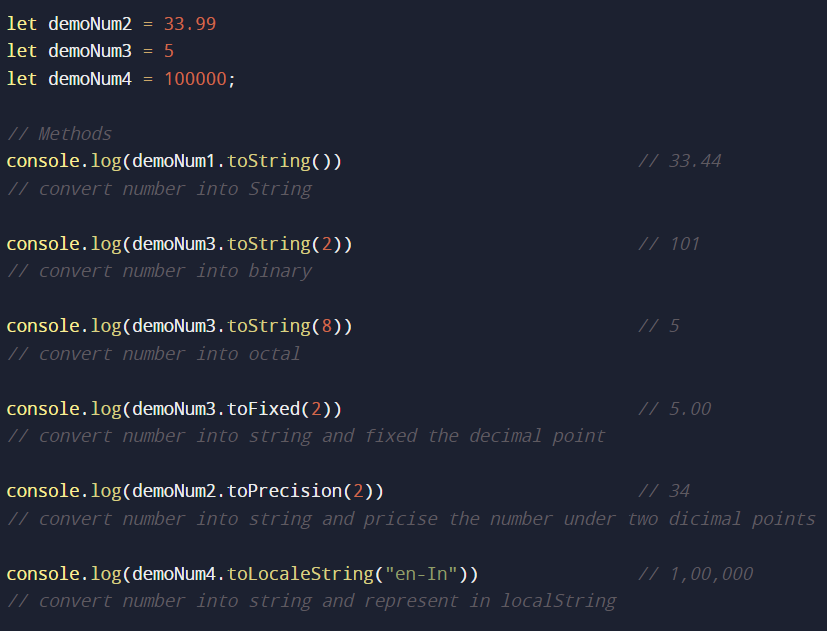
Important datatypes and some operations

* Number and some operations

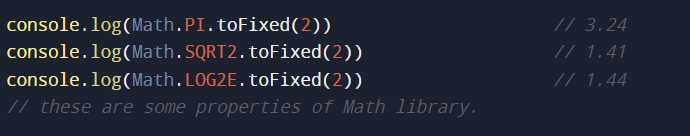
Basic:



Methods:



Math Library:



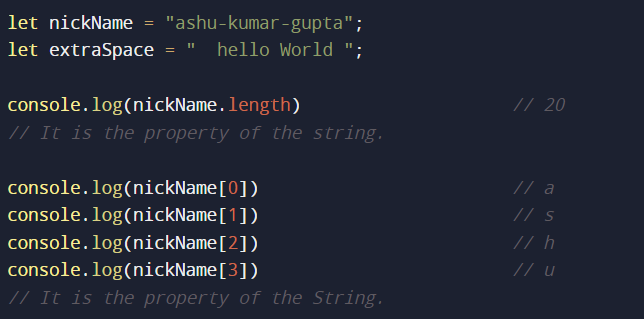


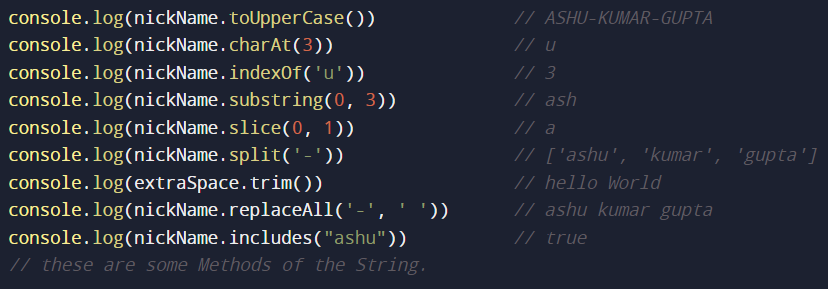
* String and some operations

Basic:



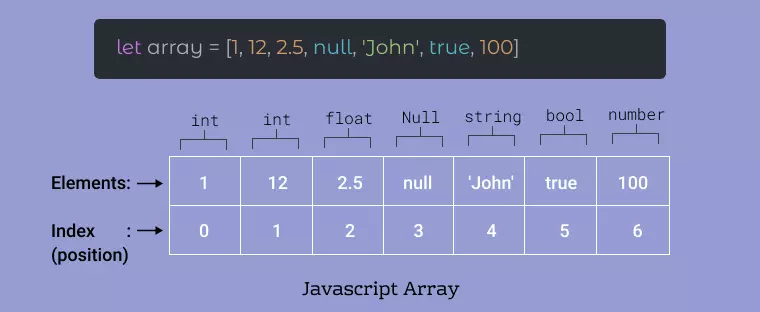
Methods:



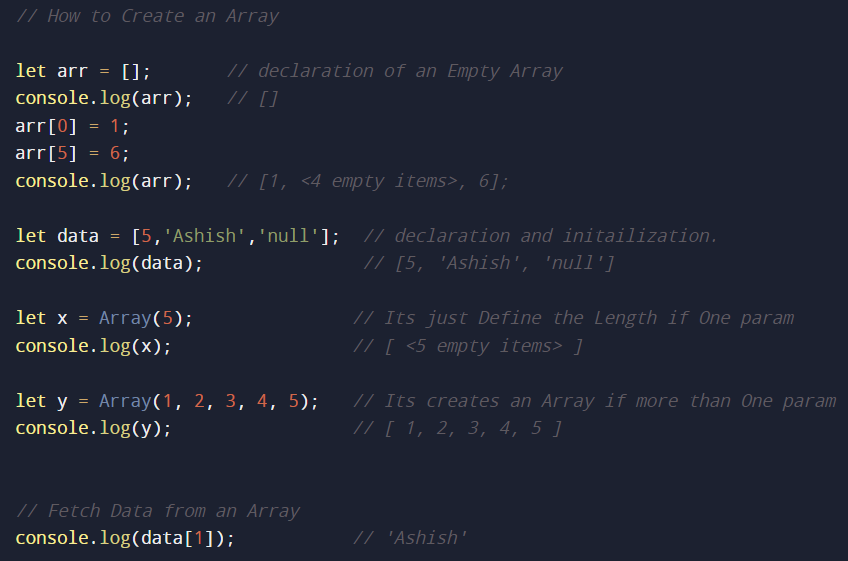


* Array and some operations

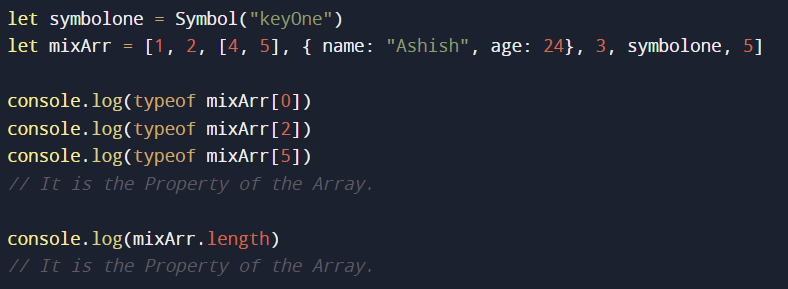
In JavaScript Array is the object that is used as a linear data structure and that is why an array in JavaScript is mutable and heterogenous in nature means it can hold all primitive and non-primitive data types.

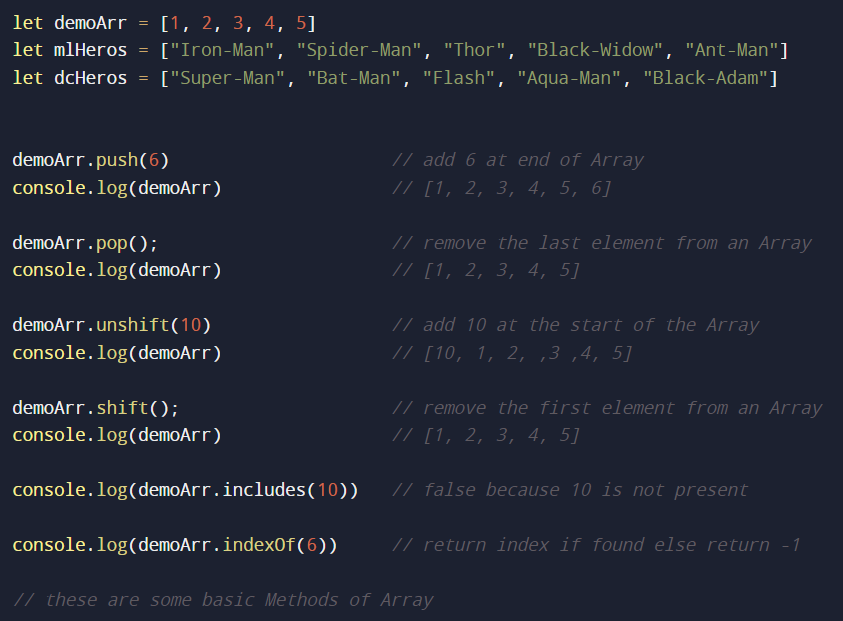
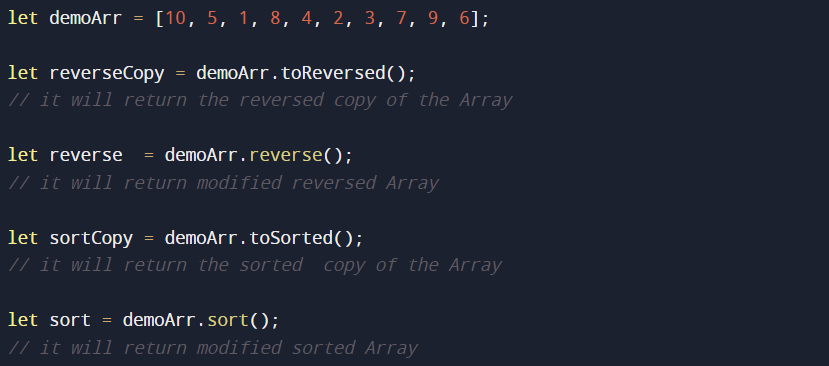


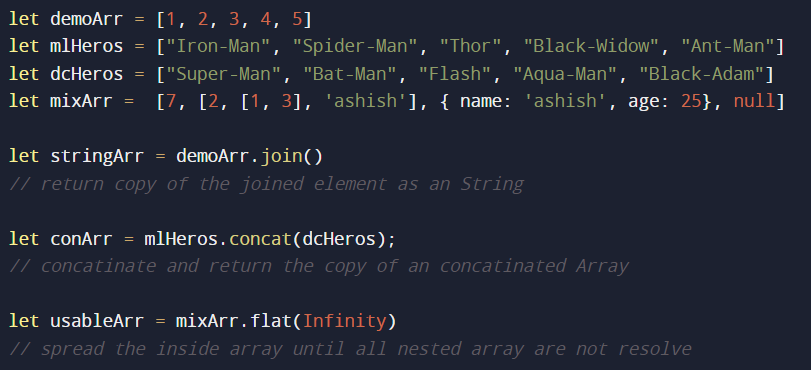
Basic:

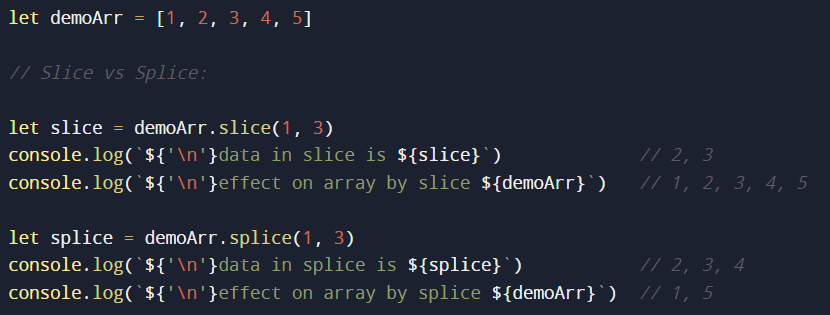


Methods:

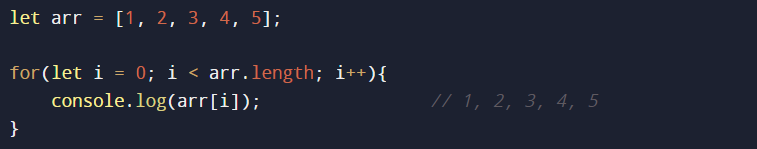


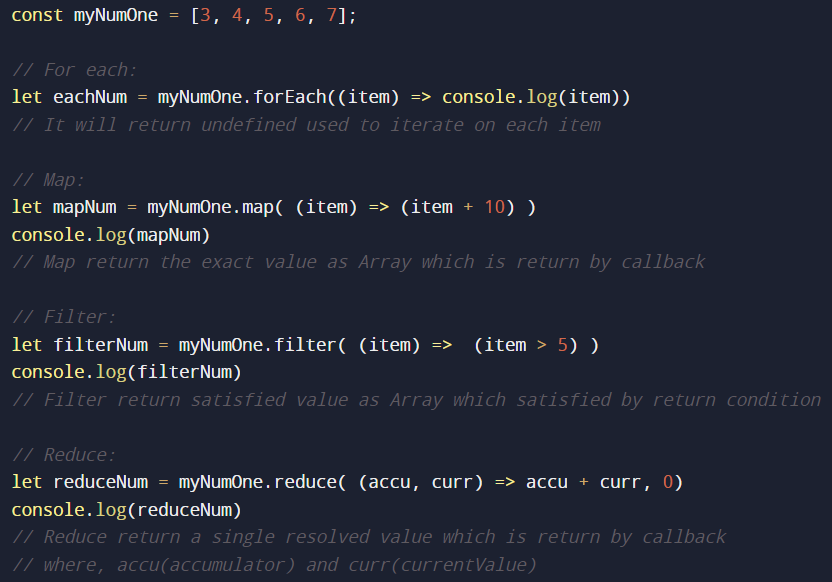


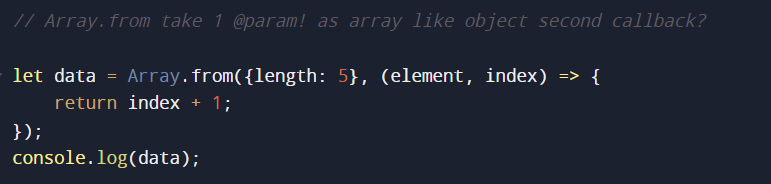




Iteration and Iteration Methods on Array:







* Object and some operations

Basic:

Methods:

Synchronous and Asynchronous JavaScript

* Synchronous means code executes sequentially (line by line) on the main thread whereas Asynchronous means code executes in a non-sequential manner i.e., code that executes along with the main thread without interrupting it.
* JavaScript supports Single-Threaded only due to which it behaves like synchronous for its native codes (means native code which JavaScript understands will execute line by line on the main thread). That’s why JavaScript can handle the synchronous code. Then how it’s able to handle asynchronous code.
* Actually, Asynchronous codes (non-native codes like setTimeout, setInterval, web APIs, etc) are handled by the Runtime Environment, JavaScript only knows how to trigger these non-native codes and immediately comes back to the main thread, but how runtime environment handle it.
* whenever or wherever the request comes from JavaScript the execution context is created for each request and starts executing in that context and after execution it pushes the callback to the event queue.
* and there is an event loop that continuously checks whether the callback stack is empty or not to push the callbacks of the event queue without interrupting the main thread.
* But there are issues with the callback because we passing our function to another piece of context which may affect the code is called Inversion of Control which is the first critical issue of callback and to solve this problem promises come into the picture.
* Second, Callback Hell is essentially nested callbacks stacked below one another forming a pyramid structure. Every callback depends on the previous callback, thereby making a pyramid structure that affects the readability and maintainability of the code.

Promises

* Promise is a Special JavaScript Object.
* Just like callback, promises can also be used as synchronous or asynchronous.
* Promises are also considered as placeholder for future tasks
* To understand, how to create a promise we need to first understand how to consume the promise.

How to consume the Promise?

Promise has some functions and properties that help us to understand how to consume the promise, and the properties are:

* status: The state of the status by default is pending which can be changed by the creator of the promise to fulfill or reject.

Note: Once the state of the status is fulfilled or rejected it will never change.

* value: Initially when the promise is created the status is pending and the value is undefined means if the status is pending forever then the value is also undefined forever.

When the status changes to fulfill or reject, then the value property might change.

Note: If the value of the promise is changed once it will never change again.

* onFulfillment: It is an array that holds all the functions which are defined and registered by the consumer that we want to execute once the promise state goes from pending to fulfill state.
* onRejection: It is the same as the onfulfillment but execution of registered functions is done when the status changes from pending to reject.

Note: at the time of execution the registered functions have only one argument called the value of the promise.

How to register functions in the fulfillment and rejection array??

By using the promise object function called: then

* then is a higher-order function that takes the first parameter as a success callback function which is registered to the onFulfillment array and the second parameter (optional) as a failure callback function which is registered to the onRejection array.
* we can register multiple functions by calling then method multiple times.
* then method returns the brand-new promise object which is independent of the current promise object.
* when the status of the promise changes from pending to fulfill or reject state the function registered into the onFullfillment or onRejection array will be executed and the value property of the promise object is passed as an argument.

 Example of how to consume promise:

How to Create Promise?

Promise can be created using the constructor function with the required parameter called the executor function.

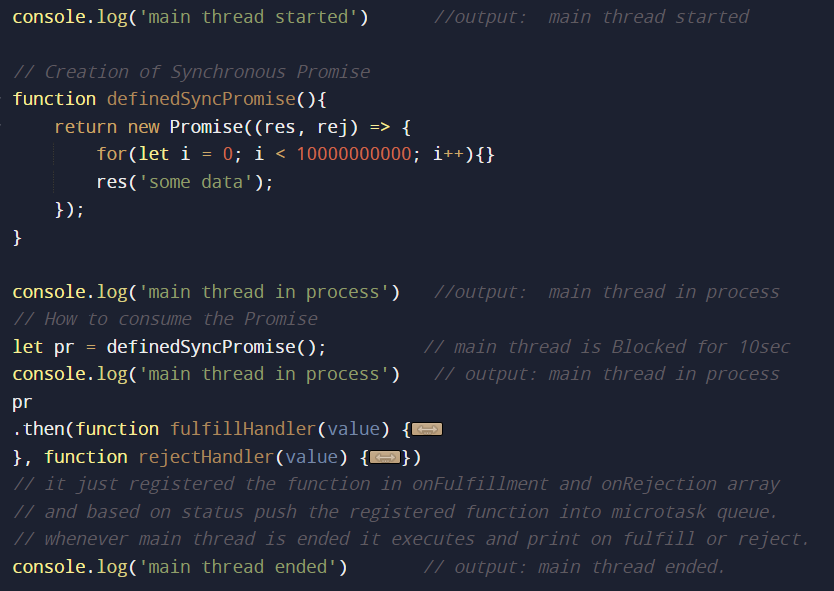
* The executor function has the implementation defining how the promise will be fulfilled or rejected.
* Again, the executor function also takes two arguments called the resolver function and the rejector function.

Note: when the executor function finishes its code execution then the promise object is created.

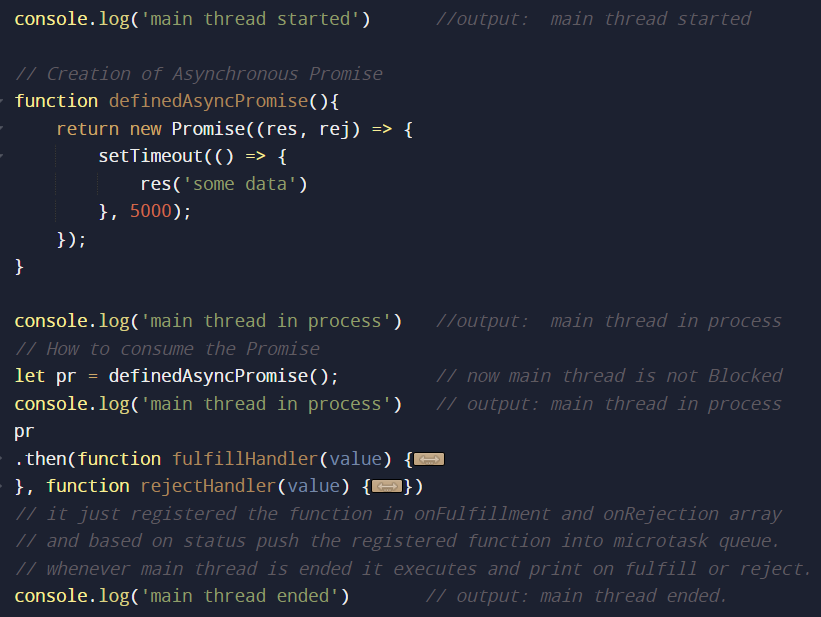
* Whenever we call the resolver function anytime in the executor function it will change the state of status from pending to fulfill and the argument that we pass in the resolver function which change the value property from undefined to that argument. And push the registered onFulfillment array functions into the microtask queue.
* In the case rejector function it will change the state of the status from pending to reject, change the value that we pass as an argument in the rejector function, and push the registered onRejection array functions into the microtask callback queue.

Note: whatever we call first which changes the state of the promise the second call of anyone is irrelevant.

Example of Creating Synchronous Promise:



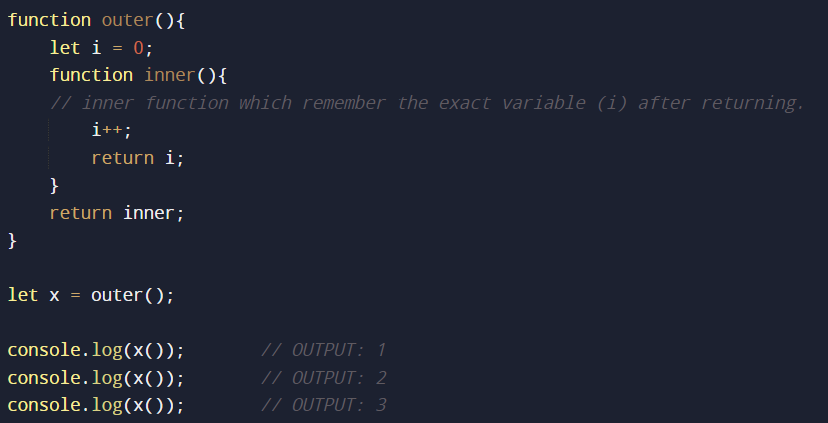
Example of Creating Asynchronous Promise:



Closure

* Closure is the mechanism using which a function remembers the variables that might not be present in its local scope, i.e., the same variable of the outer function persists for the closure function (inner function) if it is used inside the closure function.
* In closure, we close over the variable instead of a static copy of the variable.
* It happens due to lexical scoping.

example:



Async await

* We can make any function as an Async function using async and await keywords.
* It is similar to the generator function with some differences, await always wraps the promise object and jumps out with that from the async function, and after resolving or rejecting the promise starts from where it jumps out of the execution.
* It has better syntax and readability than dot then chaining which also makes promise hell.

Example:



Opps

DOM

API