ASSIGNMENT

Question:1 Blocking and Non-Blocking: Asynchronous Nature of Node.js

Ans- Blocking refers to operations that **block** further execution until that operation finishes while non-blocking refers to code that doesn't **block** execution.

Node. js docs puts it, blocking is when the execution of additional JavaScript in the Node. js process must wait until a non-JavaScript operation completes.

Blocking is when the execution of additional JavaScript in the Node.js process must wait until a non-JavaScript operation completes. This happens because the event loop is unable to continue running JavaScript while a blocking operation is occurring.

In Node.js, JavaScript that exhibits poor performance due to being CPU intensive rather than waiting on a non-JavaScript operation, such as I/O, isn't typically referred to as blocking**.** Synchronous methods in the Node.js standard library that use libuv are the most commonly used blocking operations. Native modules may also have blocking methods.

All of the I/O methods in the Node.js standard library provide asynchronous versions, which are non-blocking, and accept call-back functions. Some methods also have blocking counterparts, which have names that end with Sync.

Blocking assignment executes "in series" because a **blocking** assignment blocks execution of the next statement until it completes. **Non-blocking** assignment executes in parallel because it describes assignments that all occur at the same time.

BLOCKING

A Blocking term is originally originated from the operating system process model. It is actually a multitasking operating system, each labeled process with a state depending on how ready those processes are to be put on the CPU for execution.

When JavaScript execution in the Nodejs process has to wait until a Non-JavaScript operation to complete is called blocking.

A process is labeled as **blocked** if it is not ready for execution. So it will wait for an event to occur. Each Input/Output event indicates either progress or completion in an I/O operation.

Performing a blocking system call forces the entire process to enter into the blocked state.

NON-BLOCKING

A non-blocking operation in Node js does not wait for I/O to complete. Whenever a blocking operation happens in the process, all other operations will put on hold at the operating system level. Performing such non-blocking I/O operation, the process continues to run in the non-blocking mechanism.

A non-blocking call initiates the operation and leaves it for OS to complete returning immediately without any results.

Let's see how this can be done with non-blocking code in Nodejs:

Here we try to read simple files: hosts and users and printing their contents, meanwhile printing a few welcome messages.

Question:2 Difference between let, const and var

Ans- **Variable Declaration with let and const**

In ES6 finally we can declare variables with **let** and constants with **const** keywords. From name variables should be able to change the values and constants should not allow to change the values.

```
//ES5
var x = 14;
x = 12;
console.log(x);//12
12;
```

from the above code we successfully changed the value of variable from 14 to 12

```
//ES6
let x = 14;
x = 12;
console.log(x); //12
```

from the code above we successfully changed the value of variable from 14 to 12

```
//ES6
const y = 14;
```

```
y = 12; //TypeError: Assignment to constant variable.
console.log(y);
```

From the code above with const when we run the program got the type error that value of

const cannot be changed.

Var

The JavaScript variables statement is used to declare a variable and, optionally, we can initialize the value of that variable.

Example: var a =10;

- Variable declarations are processed before the execution of the code.
- The scope of a JavaScript variable declared with var is its current execution context.
- The scope of a JavaScript variable declared outside the function is global.In the above code, you can find, when the variable is updated inside the if loop, that the value of variable "a" updated 20 globally, hence outside the if loop the value persists. It is similar to the Global variable present in other languages. But, be sure to use this functionality with great care because there is the possibility of overriding an existing value.

let

The **let** statement declares a local variable in a block scope. It is similar to **var**, in that we can optionally initialize the variable.

Example: let a =10;

- The let statement allows you to create a variable with the scope limited to the block on which it is used.
- It is similar to the variable we declare in other languages like Java, .NET, etc.

const

const statement values can be assigned once and they cannot be reassigned. The scope of const statement works similar to let statements.

Example: const a =10;

As per usual, naming standards dictated that we declare the const variable in capital letters. **const a =10** will work the same way as the code given above. Naming standards should be followed to maintain the code for the long run.

keyword	const	let	var
global scope	NO	NO	YES
function scope	YES	YES	YES
block scope	YES	YES	NO
can be reassigned	NO	YES	YES

Scope of let, const and var

To move forward we need to understand what is declaration and initialization

var a *//declaration*
a = 5 *// initialization*let b *//declaration*
b = 10 *//initialization*var c = 5 *//declaration plus initialization in one step*
let d = 5 *//declaration plus initialization in one step*const a ; *// SyntaxError: Missing initializer in const declaration*
a = 5;
console.log(a);const a = 5
console.log(a) *//5*
1: when we start our variable with var, let is called declaration. e.g: var a; or let a;
2: when we start our variable and assigning value it is declaration and initialization with value
3: const cannot be declared only, you need to initialize it with declaration

In the scope section, we will **differentiate between var and let**
Let and const have a block scope but var has function scope.

If you did not understand the above line it is completely ok we are going to understand this below.

//ES5
function adult5(age) {

```
  if (age > 18) {
    var status = 'adult';
  }
  console.log(status); //adult
}
adult5(20);
```

The above example gives output ‘adult’, the var status can be accessed anywhere in the function.

Before moving forward the block in javascript is anything between parenthesis like {…}.So one set of parenthesis makes one block. example of a block is below.

```
{
.....
.....
.....
.....
}
```

lets move forward to the important point.

```
// ES6
function adult6(age) {
  if (age > 18) {
    let status = 'adult';
  }
  console.log(status); //ReferenceError: status is not defined
}
adult6(20);
```