SCS 2207 – Programming Language Concepts Assignment 01

SCS 2207 – Programming Language Concepts Assignment 01

Name : Nethsara Sandeepa Elvitigala

Index No: 20000482

Reg No: 2020cs048

1. Asynchronous Programming

Asynchronous programming is a technique to make sure that some long running programs can be run without blocking other programs from running.Although may not seem that useful in a multi-threaded programming language, this technique is very important in a single threaded langauge like JavaScript.By default, Javascript is synchronous, meaning that next tasj only runs after completing the previous task.But this is problematic in single threaded language like Javascript which is specially used to build user interfaces.Because completing tasks requested by user, makes the UI unresponsive because JS will be occupied with completing the ongoing task, thus preventing reacting to events.

Following is a example of a example of a synchronous (blocking) js program

```
    const params = { someKey: "Some Value" }
    const result = getDataFromLongRunningFunction(params)
    console.log(result)
    // assume this function is a blocking function, whiche means that it will hang execution until it comp
```

Assuming the "getDataFromLongRunningFunction" takes 10s, "result" will only be logged after 10s (this may seem unproblematic at first, but it's very problematic for a language like JS which is used as language to make UIs.)

But looking at the following example of asynchrnous js function,

```
async function getDataFromLongRunningAsyncFunction(params) {
    const result = await longRunningTask(params) // takes 10s to complete
    console.log(result)
}

const params = {someKey: "Some Value"}
getDataFromLongRunningAsyncFunction(params)
console.log("Hello World")
```

Here , instead of blocking the printing of "Hello World", the function sort of runs in the background and js continues to run other synchronous tasks.So the User Interface is always responsive.Only after completing the "longRunningTask()", the "result" will be logged.

At the start, JS had one way of handling asynchronous programming and it was Callbacks.

2. Callbacks

As a functional prgramming language, JS has the ability to write functions that can take other functions as inputs, so JS uses this behavior to define functions that can take other functions as input and only runs them after the initial task is completed.However, tasks outside of the function is carried out without blocking them.Thus making the page not unresponsive.

Below is an example,

```
console.log(1)
console.log(2)
setTimeout(() => {
    console.log(3)
}, 2000)  // 2000ms is the waiting time
console.log(4)
```

This set time out is a builtin asynchronous function,so instead of waiting until 2s finishes, the program prints 4 and only prints 3 after the 2 seconds.So the out put is like below,

```
1
2
4
3
```

In this example the callback function is a arrow function which is,

```
() => {
    console.log(3)
}
```

However, this callback technique lead to something called as callback hell after a while, which are lots of nested functions which lead the programs to be unreadable and hard to debug.So a new technique called promises were introduced in modern Javascript.

3. Promises

This pattern includes returning a promise which either “resolves” meaning it completed it’s task successfully or “rejects” which means it failed to complete successfully.The promise can choose to resolve with anything (mosty used to get data from API calls) and this lead to a pattern where asynchrnous programming in JS is concise and readable.

Looking at the following example,

```
function asyncLoadData() {
    return new Promise((resolve, reject) => {
        try {
            const data = someLongRunningTask()
            //if successfull
            resolve(data) // passign the data to be returned
        } catch (error) {
            // if there is an error,
            reject(error) // passing the reject reason
        }

    })
}
```

There are two ways to use this function in JS, the more traditional one is below,

```
asyncLoadData().
    then((data) => { console.log(data) }).
    catch((err) => { console.log(err)  })

    // on completing either the data (if succeeds) or error(if failed) will be logged.
```

However, this doesn’t solve the callback hell issue.So a new syntax was introduced, but as of now it’s mostly used inside other functions marked with async .The support to do this globally without functions is also introduced but still not adopted in a lot of browsers.

```
async function doSomethingAfterAsyncLoadData() {
    data = await doSomethingAfterAsyncLoadData() // assume this takes 10s
    console.log(data)
}
```

Here , console.log will only be called after getting the data from previously defined “asyncLoadData” function.So doing something lke below wouldn’t block the program execution.

```
doSomethingAfterAsyncLoadData()
console.log("I ran")
```

Here “I ran” will be logged first and then data will be logged after API call finishes. Also most modern JS builting functions that takes a long time and most libraries this by default now.So it’s rare to write promises by hand unless there isn’t already a solution for the problem.