Ethereum
Blockchain Development
Week 5

1 Introduction

We need to get organised and know where our code is and how to access it using command line. Some of you will have completed the steps below, so you only need to add the steps you haven't completed. Make sure you have completed all the steps before beginning this exercise:

- Create a directory on the main volume, say C, named, 'Ethereum'.
- Go to this directory.
- Create sub-directories for each week of the course, say 1,2,3,. . . ,10,11,12
- All today's exercises are to be saved in week 5, so go to sub-directory 5.
- Complete each exercise before moving to the next exercise
- Finally, at the end of today's exercise it is recommended you back up all of your work on the cloud or a peripheral device.

```
1 function displayText(str){console.log(str);}
2 displayText('Hello, World!');
```

Figure 3.1: JavaScript code for 'Hello, World!' exercise

2 Install VSCode

Follow the lecture notes and install VSCode, simply type:
`sudo snap install -classic code`

Then complete the following setup:

- Open VSCode and lock it to the launch bar.
- Import the Hyperledger Composer extension and install.
- Open a folder '3550/5' in VSCode and add then add a new file, ex1.js.
- Add the terminal node in VSCode so that you can run the code in the same environment, simply open a terminal if you are unable to do this.

3 'Hello, World!' and Node.js

Hello, world! is usually the first program you write in a new language. The program's aim is to display the words "Hello, world!" in the output display. In node.js, the display output could be several options, however, it is not the Document Object Model, with its ability to access document components, that is of interest. So, we are going to chose the console.log() as our standard output. The console.log() is access in different ways and depends on how you have installed *npm*, for further installation options and advice see www.nodejs.org. Type the code in Fig. **??** and execute.

```
function printStr(string){
  setTimeout(
    ()=>{
      console.log(string)
    },
    5000);
}
function printAll(){
  printStr('A');
  printStr('B');
  printStr('C');
  }
printAll();
```

Figure 4.1: JavaScript code for exercise 3

4 Arrow Functions

Using a display function, 'printStr(string)', we want to display a few strings, with a 5000ms delay. Type the following code in Fig.**??** and save as 'ex2.js'.

Complete the following modifications to your code:

- Comment your code, explaining and identifying paramets and function calls.
- Once working copy your code to a new file 'ex3.js'
- In 'ex3.js' add a parameter, 't' to the 'printStr' function. Change the function call to 'printStr' in the function 'printAll'.

```
function printStr(string,t){
  return new Promise((resolve,reject)=>{
    setTimeout(
      ()=>{
        console.log(string);
        resolve();
      },
      t);
  });
}
```

Figure 5.1: JavaScript code for exercise 4

5 Promises

The next task is to introduce promises. The first task is to alter the 'printStr' function to include a promise. Create a new file, 'ex4.js' and complete the code in Fig. **??**.

Complete the following modifications to your code:

- As in the previous exercise create a function, 'printAll()', to display the sequence 'A','B' and 'C'. For example, add a function similar to previous examples that completes this task. Then add a call to 'printAll()'.
- Create a new function 'printAll2()', this function should make 3 calls to 'printStr(str,int)'. Each call should be as follows:
  - printStr('A',2500); printStr('B',500); printStr('C',25)
- Create a new function 'printAll3()', this function should make the same 3 calls to 'printStr(str,int)' as the above function. However, this function should be completed using a promise chain and thus synchronise the output, despite the time delays.
- Finally, add a 'catch' command to the promise chain to potentially catch any errors

6 Await

Create a new file and add the code in Fig.**??**. Using the await command create an asynchronous function ‘printAll()’ that passes the following functions calls and outputs the strings ’A’, ’B’ and ’C’.

- printStr(’A’,2500)
- printStr(’B’,250)
- printStr(’C’,25)

```
function printStr(prev,curr,t,cb){
    setTimeout(
      ()=>{
        cb((prev+curr))
        },
      t);
}
function printAll(){
  printStr('','A',2500,result=>{
    printStr(result,'B',250,result=>{
      printStr(result,'C',25,result=>{
        console.log(result)
        })
      })
    })
}
printAll();
```

Figure 7.1: JavaScript code for exercise 6

7 Callback hell

There are reasons not to use callbacks, we like them when they are just at a single level, however for this simple example of outputting three strings we soon see complications arise.

Type the code in Fig.**??** and save as 'ex6.js' then execute.

Add comments to your code to explain the process of the callback and how the output differs.

8 Resolve and Promise

Let us try to write some code to resolve the problem of using complex callbacks. Essentially, there are two approaches of using the 'await' command and promise chaining. Create a new file, 'ex7.js' and enter the code in Fig.**??** and then execute.

Complete the following:

- Comment the code, identifying parameters, function calls and intentions
- Why does the code not work?
- Identify which line of code you comment out to get this working
- Now uncomment the line and add a function 'printAll()' that should complete the same task as function 'printAll2()' but uses a promise chain [hint: see previous exercises for help].

```
function printStr(prev,curr,t){
    return new Promise((resolve,reject)=>{
    setTimeout(
      ()=>{
        resolve(prev+curr)
        },
      t);
    })
}
async function printAll2(){
  let str=''
  str=await printStr(str,'X')
  str=await printStr(str,'Y')
  str=await printStr(str,'Z')
  console.log(str);
}
//printAll();
setTimeout(()=>{printAll2()},5000)
```

Figure 8.1: JavaScript code for exercise 7

9 Promise, Reject and Catch

Exercise 8 uses promises and reject to verify input. Create a new file 'ex8.js' and type the code in Fig.**??**. Once you have this working complete the following tasks:

1. Add comments to your code to explain each step and identify function calls and parameters.
2. Pass an integer as a parameter in the call to 'printStr()' from the 'printAll()' function. Does it catch the error?
3. Pass an integer as a parameter in the call to 'printStr()' from the 'printAll2()' function. Does it catch the error?
4. Modify the code using try and catch methods and adapt function 'printAll2()' to catch the error. Test this and see if it works.

```
function printStr(prev,curr,t){
    return new Promise((resolve,reject)=>{
    setTimeout(
       ()=>{ if((typeof(prev)=='string') && (typeof(
   curr)=='string')){
            resolve(prev+curr)
         } else {
           reject('Enter Strings only')
           }
         },
       t);
    })
}
function printAll(){
  printStr('','A',2500)
  .then( result=>printStr(result,'B',250))
  .then( result=>printStr(result,'C',25))
  .then( result=>console.log(result))
  .catch( result=>console.log(result))
}
async function printAll2(){
  let str=''
  str=await printStr(str,'X',2500);
  str=await printStr(str,'Y',250);
  str=await printStr(str,'Z',25);
  console.log(str);
}
printAll();
setTimeout(()=>{printAll2()},1000)
```

Figure 9.1: Exercise 8. Code to test await and chain promises