CST4125: Blockchain Development
Week: 5
Title: Introduction to Node.js

Dr Ian Mitchell

smerf.net
Bedfordshire,
UK

2023

Aims & Objectives

- VSCode and Hyperledger Composer
- Overview of Nodejs [2]
- A/Synchronous Programming
- Examples

Deadlines

				Feedback	
Description	Submission	Weight	Deadline	Formative	Summative
1. Hyperledger	MyLearning	50%	14th April 2023	LW11-12	10/05/2023
2. Ethereum	MyLearning	50%	8th July 2023	LW23-24	28/07/2023
Resits	MyLearning	50-100%	14th August 2023	None	None
Deferals	MyLearning	50-100%	14th August 2023	None	None

Contact and Office Hours

Contact Details

- Name: Dr Ian Mitchell
- Room: TG10
- Address: Middlesex University, Computer Science, London, NW4 4BT
- email: smerf.net

Contact and Office Hours

Contact Details

- Name: Dr Ian Mitchell
- Room: TG10
- Address: Middlesex University, Computer Science, London, NW4 4BT
- email: smerf.net

Office Hours

- During term time only
- When: Winter Term: Mondays 1100-1300hrs
- Please read notifications or emails
- There are occassions that these could be arranged online, e.g., due to industrial action or inclement weather

OTC

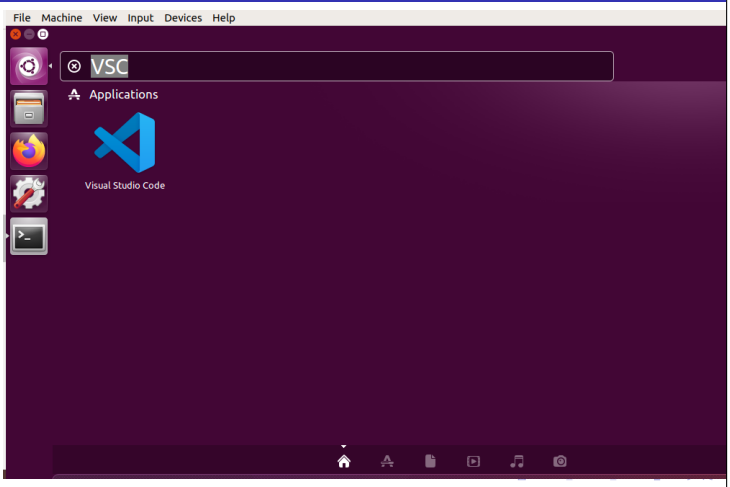
MDX App

Install VSCode

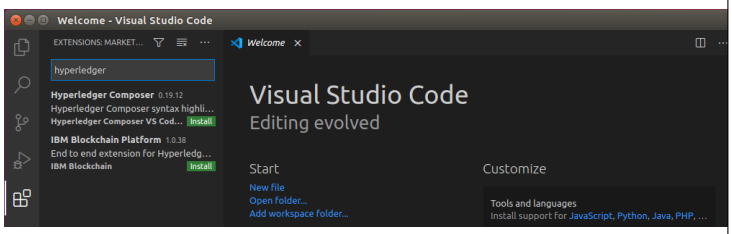
cli

```
sudo snap install --classic code
```


Run VSCode

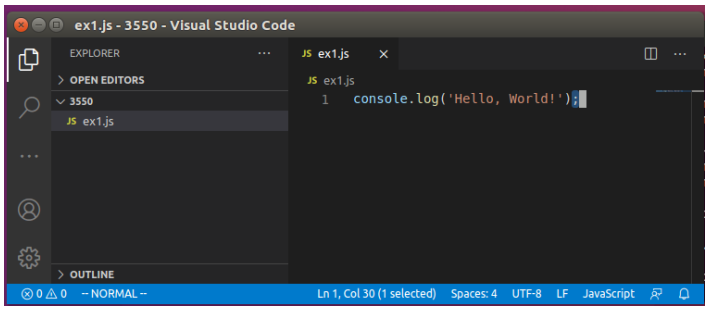


Install Extensions for Hyperledger Composer

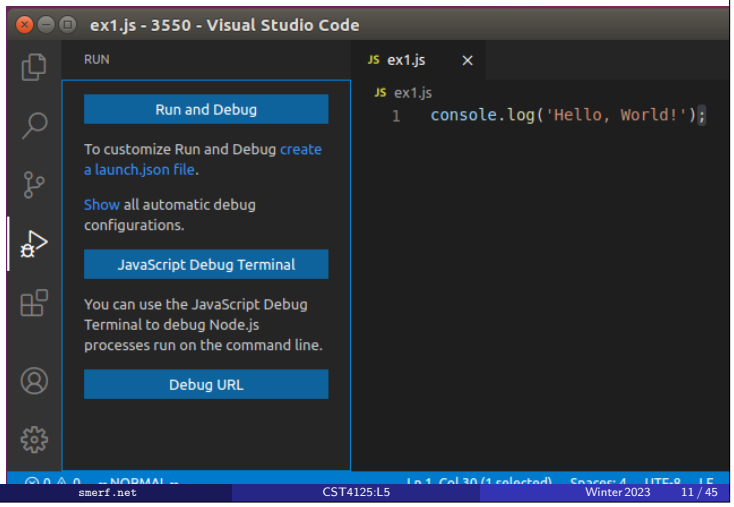


Lock to Launch bar

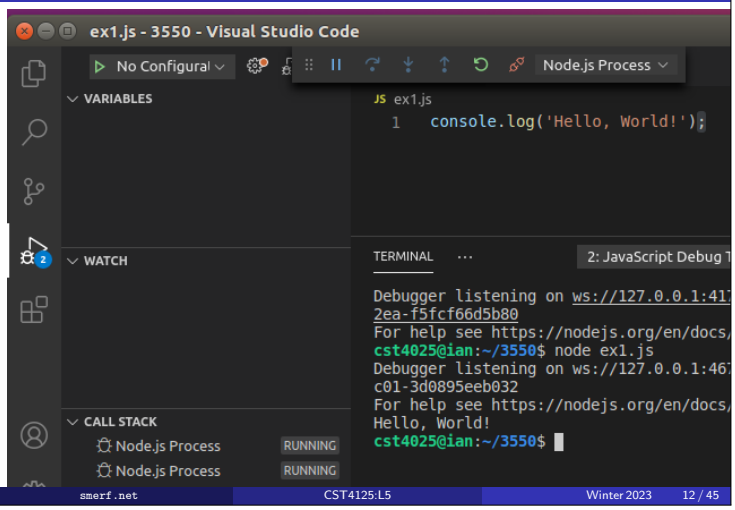
Open a Folder and a File



Run and Debug Environment



Run and Debug



Synchronous
Blocking

- Sequence
- First-come-first-serve
- Queue
- Supermarket
- Blocks

Definition

happening at the same time

Listing

```
1 console.log('Start');
2 console.log('End');
```


Synchronous
Blocking

- Sequence
- First-come-first-serve
- Queue
- Supermarket
- Blocks

Definition

happening at the same time

Output

Start
End

Listing

```
1 console.log('Start');
2 console.log('End');
```


Asynchronous
Non-blocking

- Stack
- process whilst available
- complete shortest job first
- Dynamic
- Supermarket
- non-blocking

Definition

not happening at the same times

Listing

```
1 setTimeout( ()=>{console.log("A")},250);
2 setTimeout( ()=>{console.log("B")},50);
```


Asynchronous
Non-blocking

- Stack
- process whilst available
- complete shortest job first
- Dynamic
- Supermarket
- non-blocking

Definition

not happening at the same times

Output

B
A

Listing

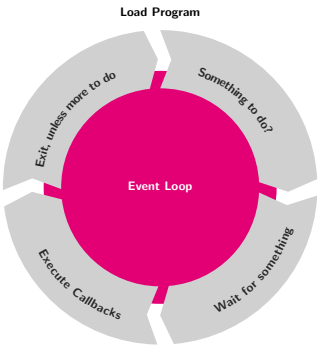
```
1 setTimeout( ()=>{console.log("A")},250);
2 setTimeout( ()=>{console.log("B")},50);
```


Node.js
Application

Felix Geisendörfer

"Everything runs in parallel except your code"

- Events
- Callbacks
- Listening
- Create callback functions that get executed in response to listening to events
- Non-blocking



Node.js

Single-Threaded and Highly Parallel

- Run code
- Non-blocking
- Callback function
- succeeds or fails

Backwardism

- Maths: $3x + 2 = 11$
- Maths: x can only be one value
- Code: $x = x + 1$
- x can be any value
- x can have different values at different times

Why?

- Composer
- Asynchronous
- Non-blocking
- Single-Threaded
- Event-based

Node.js

Single-Threaded and Highly Parallel

- Run code
- Non-blocking
- Callback function
- succeeds or fails

Backwardism

- Maths: $3x + 2 = 11$
- Maths: x can only be one value
- Code: $x = x + 1$
- x can be any value
- x can have different values at different times

Why?

- Composer
- Asynchronous
- Non-blocking
- Single-Threaded
- Event-based
- *There are times when you need to synchronise your asynchronous code*

Node.js

Single-Threaded and Highly Parallel

- Run code
- Non-blocking
- Callback function
- succeeds or fails

Backwardism

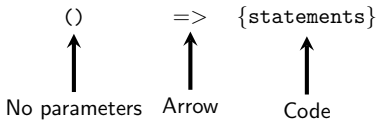
- Maths: $3x + 2 = 11$
- Maths: x can only be one value
- Code: $x = x + 1$
- x can be any value
- x can have different values at different times

Why?

- Composer
- Asynchronous
- Non-blocking
- Single-Threaded
- Event-based
- Functions: Arrow functions; callback functions and Promises.
- Run functions asychronously
- Sychronising your asynchronous code

Arrow Function: No Parameter

Syntax



Arrow Functions

Comparison

Listing without

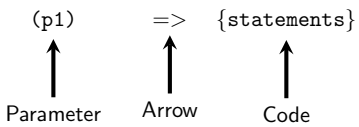
```
1 var add = function(x,y){return x+y;}
2 console.log(add(3,7));
```

Listing with

```
1 var add = (x,y) => x+y;
2 console.log(add(3,7));
```

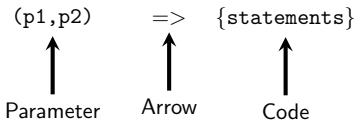

Arrow Function: Single Parameter

Syntax



Arrow Function: Multiple Parameter

Syntax



Arrow Functions

Benefits

- Shorter
- Bind `this` lexically
- It is becoming standard

Anonymous Callback

Definition

Passing a function as an argument

Example

```
function mathOperate(x,y,callback){
  var result=callback(x,y);
  console.log("result: "+result);
}

mathOperate(10,5,function(u,v){return u*v;});
mathOperate(10,5,function(u,v){return u+v;});
```

- Why?
- Dynamic

Anonymous Callback

Output

result: 50
result: 15

Example

```
function mathOperate(x,y,callback){
  var result=callback(x,y);
  console.log("result: "+result);
}

mathOperate(10,5,function(u,v){return u*v;});
mathOperate(10,5,function(u,v){return u+v;});
```

- Why?
- Dynamic

Asynchronous

- `SetTimeout(`*fn, ms*`)`
- Exec. fn after ms
- Order $\neq$ Code
- Non-blocking, continue to execute program

Listing

```
console.log('Start');
setTimeout( () => {console.log('Callback');},2000);
console.log('End');
```


Asynchronous

- `SetTimeout(`*fn, ms*`)`
- Exec. fn after ms
- Order $\neq$ Code
- Non-blocking, continue to execute program

Listing

```
console.log('Start');
setTimeout( () => {console.log('Callback');},2000);
console.log('End');
```

Output

Start
End
Callback

Asynchronous

Listing

```
1 console.log('Start');
2 setTimeout( () => {console.log('1st Callback')
      ;},2000);
3 setTimeout( () => {console.log('2nd Callback');},
      0);
4 console.log('End');
```

- Again
- Order $\neq$ Code
- Non-blocking, continue to execute program

Asynchronous

Output

```
Start
End
2nd Callback
1st Callback
```

Listing

```
1 console.log('Start');
2 setTimeout( () => {console.log('1st Callback')
      ;},2000);
3 setTimeout( () => {console.log('2nd Callback');},
      0);
4 console.log('End');
```

- Again
- Order $\neq$ Code
- Non-blocking, continue to execute program

Named Callback

- Why?
- Dynamic
- trigger automatic updates
- `setInterval(fn,ms)`

Example

```
1  function mathOperate(x,y,callback){
2    var result=callback(x,y);
3    console.log("result: "+result);
4  }
5
6  function times(u,v){return u*v;}
7  function add(u,v){return u+v;}
8  function mod(u,v){return u%v;}
9
10 mathOperate(10,7,times);
11 mathOperate(10,7,add);
12 mathOperate(10,7,mod);
```


Named Callback

- Why?
- Dynamic
- trigger automatic updates
- `setInterval(fn,ms)`

Output

```
result: 70
result: 17
result: 3
```

Example

```
1  function mathOperate(x,y,callback){
2    var result=callback(x,y);
3    console.log("result: "+result);
4  }
5
6  function times(u,v){return u*v;}
7  function add(u,v){return u+v;}
8  function mod(u,v){return u%v;}
9
10 mathOperate(10,7,times);
11 mathOperate(10,7,add);
12 mathOperate(10,7,mod);
```


Callback Hell

```
1 setTimeout( ()=>{
2   console.log('1: 3 second delay');
3   setTimeout( ()=>{
4     console.log('2: 2 second delay');
5     setTimeout( ()=>{
6       console.log('3: 1 second delay');
7     }, 1000);
8   },2000);
9 }, 3000);
```


Callback Hell!

```
1  function doSomething1(){
2    setTimeout( ()=>{
3      console.log('1: 3 second delay');
4      doSomething2();
5    },3000);
6  }
7  function doSomething2(){
8    setTimeout( ()=>{
9      console.log('2: 2 second delay');
10     doSomething3();
11   }, 2000);
12 }
13 function doSomething3(){
14   setTimeout( ()=>{
15     console.log('3: 1 second delay');
16   }, 1000);
17 }
18 doSomething1();
```

Parameter Passing

```
function doSomething(string,t){
  setTimeout( ()=>{
    console.log(string);
  },t);
}

doSomething('A',3000);
doSomething('B',2000);
doSomething('C',1000);
```


Promises [1]

Definition [3]

A promise is an object that serves as a placeholder for a value. That value is usually the result of an async[hronous] operation.... When an async function is called it can immediately return a promise object. Using that object, you can register callbacks that will run when the operation succeeds or an error occurs.

Promise States [3]

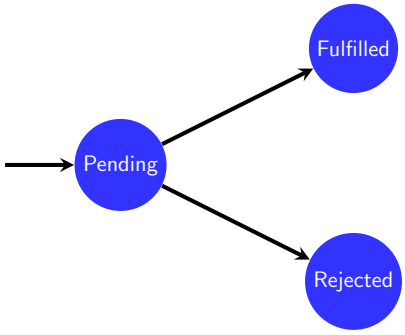
Definitions

Pending: The operation has not begun or is in progress.

Fulfilled: The operation has completed.

Rejected: The operation could not be completed.

Promise States Relationships[3]



Promise Analogy

Child: Please can I have some sweets?

Parent: I will give you some when you complete your homework

Promise Pending

Promise Analogy

Child: Please can I have some sweets?

Parent: I will give you some when you complete your homework

Promise Pending

⋮

Child: Can I have some sweets now!

Parent: Have you completed your homework?

Child: No.

Parent: Then you cannot have any sweets.

Promise Rejected

Promise Analogy

Child: Please can I have some sweets?

Parent: I will give you some when you complete your homework

Promise Pending

Child: Can I have some sweets now?

Parent: Have you completed your homework?

Child: Yes.

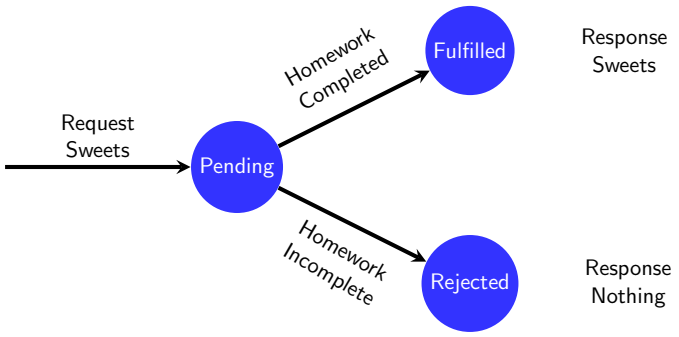
Parent: Well done, I will go and get you some.

Promise Pending

Parent: They are on the table.

Promise Fulfilled

Promise States Relationships[3]



Promise Syntax[2]

- Creation: object

Listing

```
1 var somePromise = new
    promise((resolve,reject)
    =>{
2 //do asynchronous stuff
    here
3   });
4
```


Promise Syntax[2]

- Creation: object
- Anonymous Arrow Fn

Listing

```
1 var somePromise = new
    promise((resolve,reject)
    =>{
2 //do asynchronous stuff
    here
3   });
4
```


Promise Syntax[2]

- Creation: object
- Anonymous Arrow Fn
- Asynchronous

Listing

```
1 var somePromise = new
    promise((resolve,reject)
    =>{
2 //do asynchronous stuff
    here
3   });
4
```


Promise Syntax[2]

- Creation: object
- Anonymous Arrow Fn
- Asynchronous
- Resolve, Reject

Listing

```
1 var somePromise = new
    promise((resolve,reject)
    =>{
2 //do asynchronous stuff
    here
3   });
4
```

Promise Example
Resolve

Output

```
success:  It worked
```

Listing

```
1  var somePromise = new Promise((resolve,reject)=>{
2    setTimeout(()=>{
3        resolve('It worked');
4        resolve('It worked again');//won't run -
         promises can only either be resolved or
         rejected once
5      },500);
6
7  });
8
9  somePromise.then((message) => {
10   console.log('success:',message);},
11   (errorMessage) => {
12   console.log('Failure:',errorMessage);
13 });
```


Promise Example
Reject

Output

```
Failure:  It Failed
```

Listing

```
1  var somePromise = new Promise((resolve,reject)=>{
2    setTimeout(()=>{
3      reject('It Failed');
4        },500);
5
6  });
7
8  somePromise.then((message) => {
9    console.log('success:',message);},
10   (errorMessage) => {
11   console.log('Failure:',errorMessage);
12 });
```


Revisit Callback Hell

```
1  const add = function (a, b, callback) {
2    setTimeout(() => {
3      callback(a + b);
4    }, 1000);
5  };
6
7  add(1, 2, (sum1) => {
8    add(3, sum1, (sum2) => {
9      add(4, sum2, (sum3) => {
10       console.log('Sum of first 4 natural
11       numbers using callback is ${sum3}');
12     });
13   });
14 });
```


Avoid Callback Hell

```
1  const addPromise = function (a, b) {
2    return new Promise((resolve, reject) => {
3      setTimeout(() => {
4        resolve(a + b);
5      }, 2000);
6    });
7  };
8
9  addPromise(1, 2)
10   .then((sum1) => {
11     return addPromise(3, sum1);
12   })
13   .then((sum2) => {
14     return addPromise(4, sum2);
15   })
16   .then((sum3) => {
17     console.log(
18       'Sum of first 4 natural numbers using
19        promise and then() is ${sum3}'
20     );
21   });
22 (async () => {
23   const sum1 = await addPromise(1, 2);
24   const sum2 = await addPromise(3, sum1);
25   const sum3 = await addPromise(4, sum2);
26   console.log(
27     'Sum of first 4 natural numbers using
28      promise and async/await is ${sum3}'
29   );
30 })();
```


Call a Promise
.then

- .then is a callback function
  - success
  - failure
- two callback functions
- Reject or Resolve
- Only reject once
- Only resolve once
- Pending for 500ms

Listing

```
8  somePromise.then((message) => {
9    console.log('success:',message);},
10   (errorMessage) => {
11   console.log('Failure:',errorMessage);
12 });
```


Return a Promise
Resolve

Output

```
sum:109
```

Listing

```
1  var asyncAdd = (a,b) => {
2    return new Promise((resolve,reject) =>{
3      setTimeout(()=>{
4        if( (typeof a === 'number') && (typeof b === 'number')) {
5          resolve(a+b);
6        } else {
7          reject('enter two numbers');
8        }
9      },500);
10   });
11
12 };
13
14 asyncAdd(34,75).then(
15   //first callback is the success - resolve case
16   (message)=>{console.log('Sum:',message);},
17   //second callback is the failure - reject case
18   (errorMessage)=>{console.log('Error:',errorMessage);}
19   );
```

Return a Promise

Resolve

Output

Error: enter two numbers

Listing

```
var asyncAdd = (a,b) => {
  return new Promise((resolve,reject) =>{
    setTimeout(()=>{
      if( (typeof a === 'number') && (typeof b === 'number')) {
        resolve(a+b);
      } else {
        reject('enter two numbers');
      }
    },500);
  });

};

asyncAdd(5,'a').then(
  //first callback is the success - resolve case
  (message)=>{console.log('Sum:',message);},
  //second callback is the failure - reject case
  (errorMessage)=>{console.log('Error:',errorMessage);}
  );
```


Promise Chaining [3]

Listing

```
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()},5000)
```


Promise Chaining [3]

Output

ABC
XYZ

Listing

```
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()},5000)
```


Summary

- Promises
- passing functions as parameters
- Asynchronous code
- Synchronous code
- Promise chaining
- callback hell
- async, await, let

References I

[1] B. Liskov and L. Shrira. "Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems". In: *Int. Conf. on Programming Language Design and Implementation (SIGPLAN'88)*. 1988, pp. 260–267.

[2] A. Mead. *Learning Node.js Development*. Packt, 2018.

[3] D. Parker. *Javascript with Promises*. 1st ed. O'Reilly, 2015.

Web Resources

- `http://hyperledger.org`
- `https://nodejs.org`