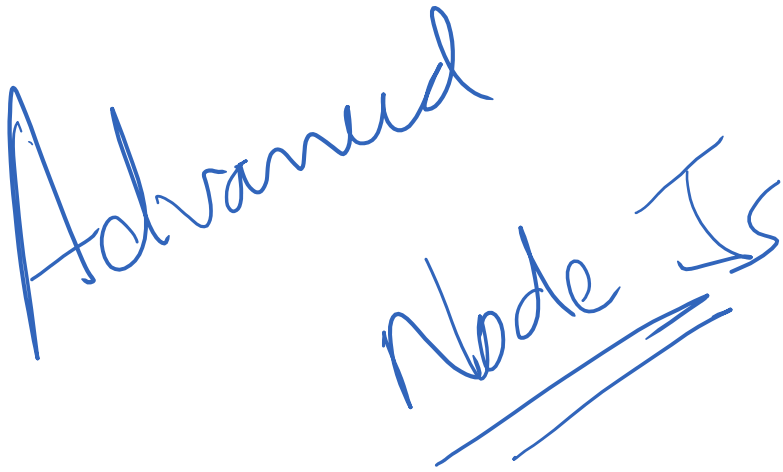
Welcome To Advanced NodeJS

Monday, October 12, 2020 10:38 AM



Assignment01

Monday, October 12, 2020 10:41 AM

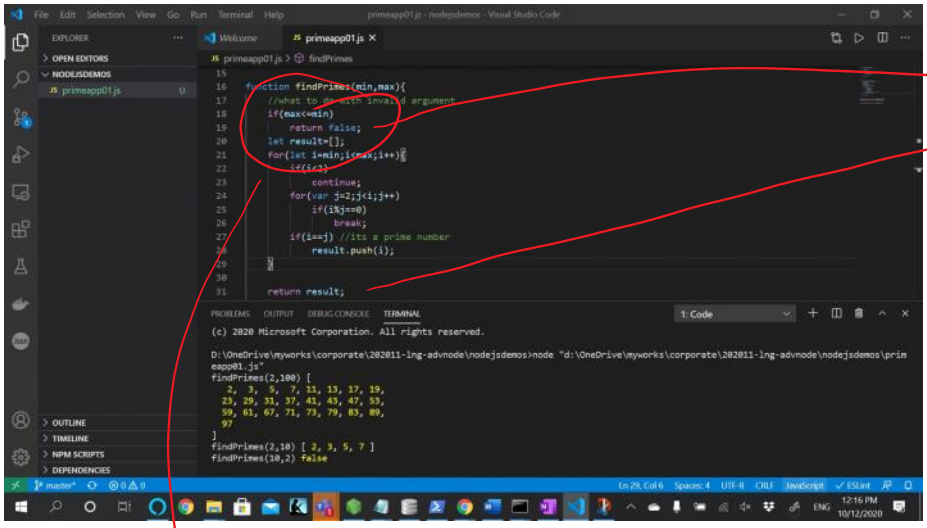
- Create a function to find and return all primes in a given min and max range
  - Example find primes between 2 and 200

- Psudo code of isPrime

```
bool isPrime(int x){

        If(x<2)
                return false;

        for(int i=2;i<x;i++)
                If(x%i==0)
                        return false;

        return true;

}
```

The common problems

Monday, October 12, 2020 12:15 PM



Returning completely different type of values

- Client is forced to check the types

Recommendation!

- **If you function returns an array, always return an array, may be an empty array when you have not value to return instead of returning false or null.**

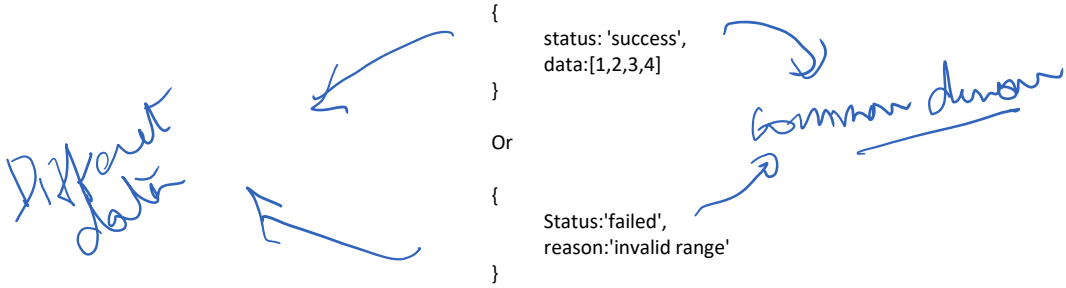
Don't return a value to indicate an error. If possible **throw exception or any standard Mechanism to indicate error.**

Loose types?

- Javascript as loose (dynamic) types.
- But to create a consistent API we must adhere to some common denomniators

- Example a method may return

```
{
    status: 'success',
    data:[1,2,3,4]
}

Or

{
    Status:'failed',
    reason:'invalid range'
}
```



Nodejs is Single threaded Asynchronous Programming model

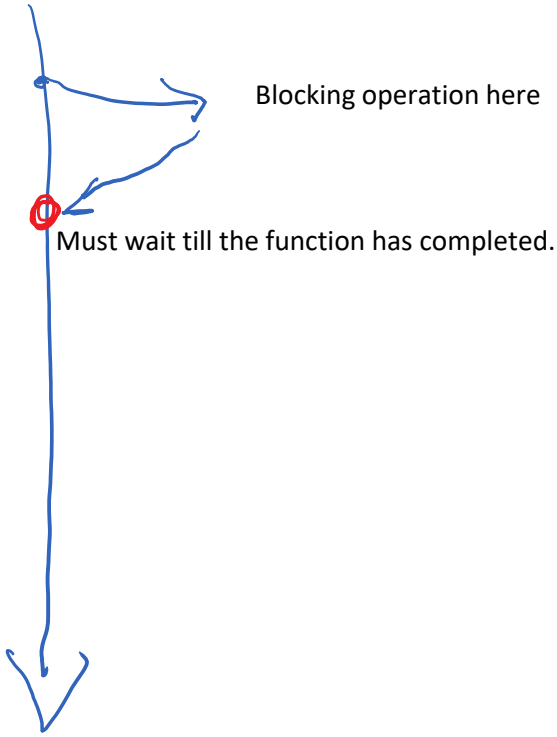
Monday, October 12, 2020 12:30 PM

NodeJS expects your functions to be async by default

- If you function is synchronous for whatever reason, it must be suffixed with the word sync

Note

- **Languages like java and C# using async suffix to mark an asynchronous function.**
- **By default functions are synchronous**
- **NodeJs expects functions to by async by default.**



Javascript Asynchrnous Programming

Monday, October 12, 2020 3:16 PM

- A general paradigm of programming, where we don't need to wait for a function to finish
  - Function returns immediately
  - Continues to work in backgournd
  - Updates the client once it finishes with the help of some kind of call back

Different Types of Asynchrnous Programming Model

1. NodeJS Callback pattern
   a. Callback is not a new concept
   b. NodeJS has a special callback syntax for function : **function callback(err,result);**
      i. **We can use this model anywhere as this is just a pattern and now a NODE JS feature**
      ii. **Most of the NodeJS API follow the same syntax.**

2. **ES2015 Promises**

Assignment 02

Monday, October 12, 2020 12:52 PM

1. Continue with Assignment01 and make the API asynchronous
2. Use Modular approach by separating business and presentation tier

NodeJS Callback Pattern

Monday, October 12, 2020 1:03 PM

1. NodeJS callback architecture

- Nodejs expects your functions not to return using return keyword
- You pass a callback as the last parameter to your function
- Once function finishes it calls the call back
- The callback should take two parameter in order
  - Err
    - Should specificy in case of error
    - Second parameter should be null/undefined
  - Result
    - Err should be null
    - Result should contain the result

```
function findPrimesSync(min,max){

    let result=[];

    return result;
}
```

Should change to

```
function findPrimes(min,max, cb){

    let result=[];
    if(success)
        cb(null, result); //success
    else
        cb('invalid input'); //error
}
```

```
function findPrimes(min, max, cb) {

    setTimeout(() => {
        if (min >= max)
            cb(new Error(`Invalid Range(${min}-${max})`));  //result is undefined
        else {
            let primes = [];
            for (let i = min; i < max; i++)
                isPrime(i, (err, result) => {
                    if (result)
                        primes.push(i);
                });

            cb(null, primes); //first parameter null indicates success
        }

    }, 2); //just to simulate that job may take long time.


}
```

Simulates a long running process

- Is running synchronously as one big chunk of code.
- Once you start, you end only after searching everything
- Not giving any other job time to work
- This is called **selfish** programming

Cooperative Worker Pattern

- A code should allow other codes to work by taking a break
- This should allow vital UI updates and other short worker to complete

How to implement co-operative worker in our code

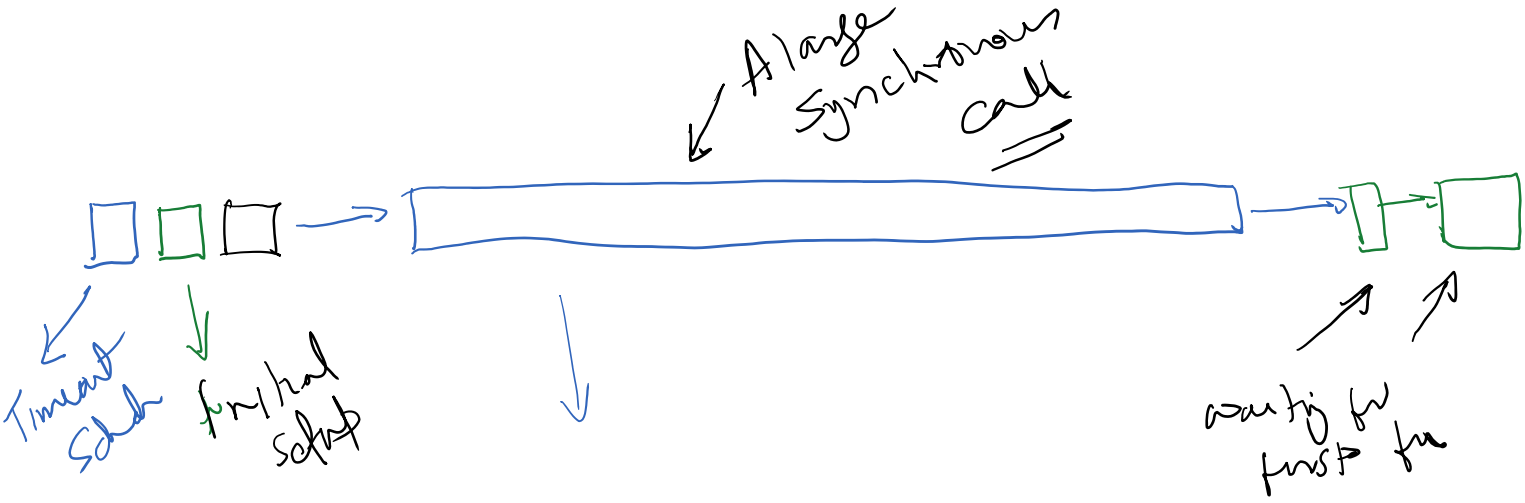
- Say we are finding all primes between 2 and 500000
- We may take a short break of say 10ms after every 1000 iteration.

Assignment03

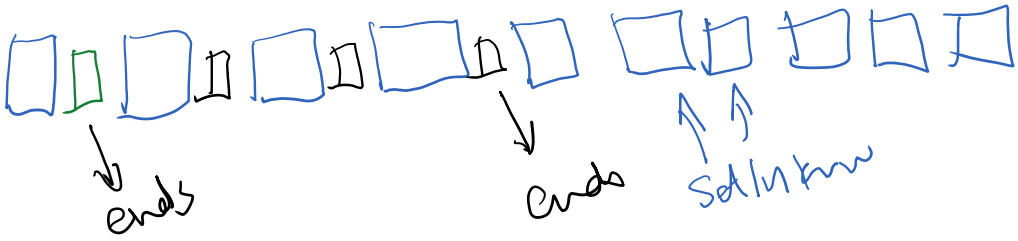
Tuesday, October 13, 2020 10:43 AM

Cooperative Async Pattern

Monday, October 12, 2020 12:52 PM



New Design



ES2015 Promises

Monday, October 12, 2020 3:19 PM

- It is not a NodeJS feature but available in general in all javascript programming
- **Evolved much later**
- **NodeJS was already using its own model of programming**

- Many Nodejs libraries are now slowly moving to Promise rather than node callbacks

A Promise

- It's a bulit in ES2015 (Javascript feature)
- **Promise is an object that promises to get some result in future**
  - **Promise also take a callback with two parameters**
    - **These two parameters are again call backs**
      1. **To call when success**
      2. **To call when failed**

- Promise to get a result asynchronously by calling another function

Promise says let me run this code and I will let you know when we are ready

```
let promise= new Promise(   function_that_will_give_you_a_result );

function  function_that_will_give_you_a_result(   fnResolve, fnReject ){

    ...
    if(success)
        fnResolve( result);   //call when you completed successfully
    else
        fnReject(err_details); //call this function when you fail

}
```

This is your business logic

Creating an api — callback vs Promise

```
function findPrimes(   min, max, cb ){
    //business logic
    ...
    if(success)
        cb( null, result);
    else
        cb(err_details);

    //This function returns nothing
}
```

No callback passed.

```
function findPrimes(   min, max ){

    let promise=new Promise( (resolve,reject) =>{
        //business logic here
        if(success)
            resolve(  result);
        else
            reject(err_details);

    }
    return promise;
}
```

We handle promise once returned

Consuming The Asynchronous operations

```
//callback example
findPrimes( 2, 100 ,   (err,primes) =>{

    if(err){
        console.log('err',err); //on failure
    } else{
        Console.log('primes', primes.length);  //on success
    }

});

//we are free to do whatever we want
//the callback will be called sometimes in future
//same callback will get both err and result
```

```
//promise based design

//function doesn't return result. It returns a future promise
let promise= findPrimes(2,100);

//we can set for future when it completes
//if promise is resolved successfully
promise. then( primes=> console.log('primes', promes.length);

//if promise is rejected because of error
promise.catch( err => console.log( 'err', err);


//we can do whatever we want to do. then() and catch() will
```

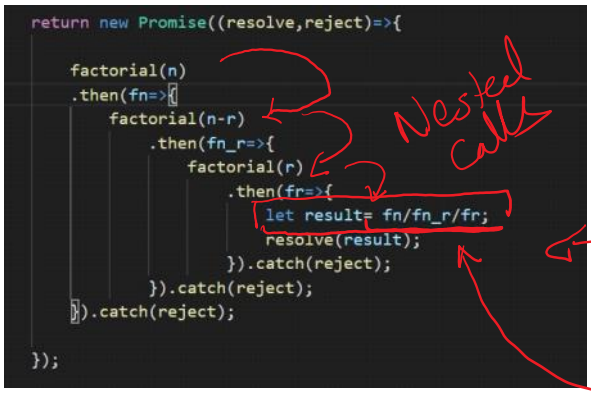
execute asynchrnously when promise is resolved/rejected in future.

//this code will execute immediately.

Promises can Be chained

```
findPrimes(2,100)
        .then(primes=> console.log(primes))
        .catch(err=>console.log(err);
```

Nested Promise Problem



1. Calculate factorial n
2. Calculate factorial of n-r
3. Calculate factorial of r
4. Use the first 3 calculation to calculate combination

- Can you see the sequence in nested promise?

This calculation depends on all the three

Async - Await Keywords

- Since Promise is a javascript feature, javascript has defined a set of keywords that makes working with Promise easy and straight forward.
- **await** is a javascript keyword that automatically resolves the promise and give you resolved result rather than promise
  - Remember this result will not come immediately but sometimes in future
- When you use **await**, the rejection is thrown as an exception that can be handled using standard **catch** keyword
- The function is actually waiting for resolved/rejected, but will finish immediately asynchrnously
  - It will execute the code later.

Manual Promise Resolution

```
function testFactorial(n){

    let p = factorial(n); //it returns a promise

     //wait for promise to complete and get resolved result
    p.then( fn => console.log(fn));

    //if promise is rejected you get rejection message
    p.catch( err=> console.log('err',err) );


}
```

1. Looks like this code is synchrnous. But actually there may be long gap between these two lines
2. This code may run in future but the function will return immediately

- Function having await must be marked **async**
- An async function always returns a Promise implicilty

Using await

```
async function  testFactorial(n){

    try{
        //await will wait for promise resolution.
        let fn = await factorial(n);  //taking n*100ms

        //next piece of code is what you would write in then, to be executed in future
        console.log('result is ',fn);   //typically what you write in then

    }
    catch(err){ //rejection is handled in catch
        console.log('err',err) ; //what you write in .catch()
    }

}
```

Anything that follows await will be executed later and therefore this function creates a Promise and returns immediately

```
let combination=(n,r)=>{

    //factorial(n)/factorial(n-r)/factorial(r)

    return new Promise((resolve,reject)=>{

        factorial(n)
        .then(fn=>{
            factorial(n-r)
                .then(fn_r=>{
                    factorial(r)
                        .then(fr=>{
                            let result= fn/fn_r/fr;
                            resolve(result);
                        }).catch(reject);
                }).catch(reject);
        }).catch(reject);

    });
```

```
async function comibnation(n,r){

    let fn= await factorial(n);
    let fn_r=await factorial(n-r);
    let fr=await factorial(r);
    let c= fn/fn_r/fr;

    return c;

}
```

1. Awaits (resolves then) and gets you resolved result fn
    a. But this will happen in future. So it is just a promise
2. Second will execute once the first promise is resolved.
    a. It is a promise against a promise.
    b. It is also future tense
3.

What is this returninig

- It appears that this function is returning a number
- But this number depends on other calculation which are based on promises
- **Here we are telling that we will return this value to you in future**
- This function is returning a **Promise** that will have this value

- Since an async function always returns a promise
    - We can always use it with then() and catch() if we need

await must always be written inside an async function

- **You can't write await in global**
- **Constructor of a class can't be marked async**
    - **You can't await inside a constructor**
    - **You can use standard then(),catch()**

Understanding Promises

Tuesday, October 13, 2020 9:59 AM

```
function combination(n,r){
    let fn = factorial(n);
    let fn_r = factorial(n - r);
    let fr = factorial(r);
    var comb='waiting for the result...';

    Promise.all([fn,fn_r,fr])  //when all promises are fulfilled (resolved/rejected)
        .then((result) => {
            //result[0] is output of promise fn
            console.log(result[0], result[1], result[2]);

            //we will reach here in apporx 1400ms for comibination(7,2);
            comb = (result[0] / result[1] / result[2]);

    })
    .catch(function(err){
        reject("combination Error: " + err);
    });


    //we reach here immediately without waiting for promise to be fulfilled.
    console.log("Calculate Factorial: " + comb);

}
combination(7, 2);
```

Will be evaluated sometimes in future

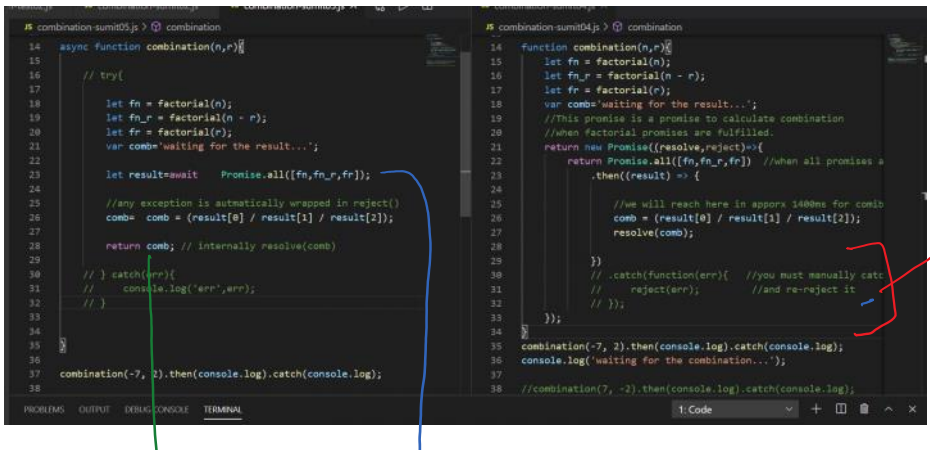
We reach here in present, immediately long before the calculations are done.

To calculate the comination we need another calcuation.

Promise to calculate the combination when other promises are fulfilled

```
//let us make a mega promise  which is a promise of all promises
let megaCombProm = new Promise(function(resolve,reject){

    return  Promise.all([fn,fn_r,fr])  //when all promises are fulfilled (resolved/rejected)
        .then((result) => {
            //result[0] is output of promise fn
            console.log(result[0], result[1], result[2]);

            //we will reach here in apporx 1400ms for comibination(7,2);
            comb = (result[0] / result[1] / result[2]);
            //we must mark the promise resolved.

            resolve(comb); //which promise are we resolving?

    })
    .catch(function(err){
        reject("combination Error: " + err);
    });

});


megaCombProm.then(function (comb){
    console.log("Calculate Factorial: " + comb);

}).catch(function (err) {
    console.log("Calculate Factorial Error: " + err);
});
```

We don't need another promise to wrap this promise!



If an inner promise is rejected

- **You must write catch()**
- **If you don't want to handle rejection you still must**
  - **Write a catch**
  - **Re-reject it**

Async await benefits

1. Code looks sequential.
2. Return is automatically translated to resolve
    a. If no return is specified end of function is resolve
3. Any rejection is an exception thrown.
    a. You don't have to handle the exception if you don't need
    b. If you don't write try catch, it is automatically re-rejected.

Assignment 04

Monday, October 12, 2020 3:41 PM

- Convert findPrimes from callback to Promise model
- Write the test application

Assignment05

Monday, October 12, 2020 4:32 PM

Create a long running factorial function.

- Psudo code for factorial

```
int factorial(int n){
        if(n<0) //error

        let fn=1;

        while(n>1)
                fn*=n--;

        return fn;

}
```

Assume factorial is a long running task and needs n*100 ms to complete

1. Create an asynchrnous factorial function that returns in n*100 ms.
   a. It should return a promise

2. Use the factorial function to calculate comination(n,r); psudocode for combination is

```
int combination(int n, int r){
        int fn=factorial(n);

        int fn_r=factorial(n-r);

        int fr=factorial(r);

        return fn/fn_r/fr;
}
```

Comination will not have any delays programmed.
It will be delayed because of factorial

Assignment06

Tuesday, October 13, 2020 10:41 AM

Convert the factorial function given below to a cooperative function

- It should still take n*100ms to complete successfully
- It should take 100ms if it fails

```
let factorial=(number)=>{
    return new Promise((resolve,reject)=>{
        setTimeout(()=>{
            if(number<0)
                reject('negative numbers do not have factorial');
            let f=1;
            while(number>0)
                f*=number--;
            resolve(f);
        }, (number>0?number:1)*100);
    });
};
```

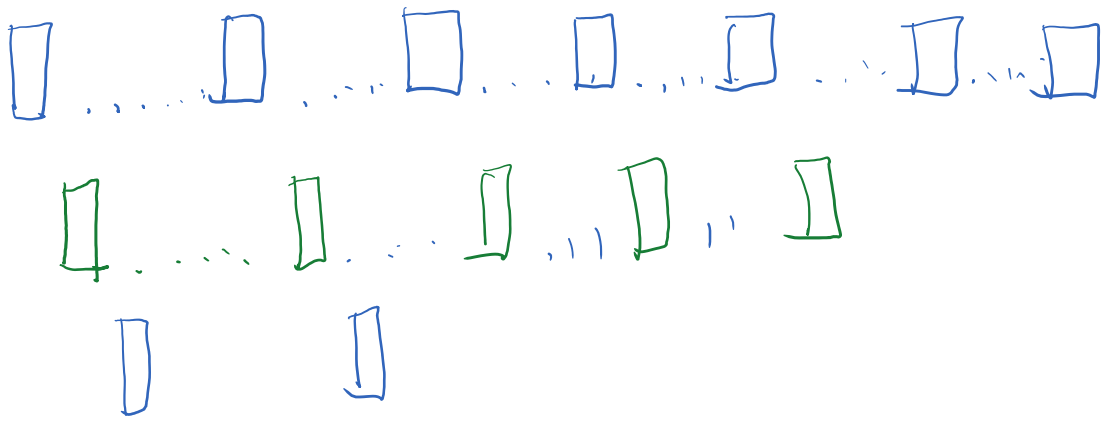
How async code works

Tuesday, October 13, 2020 11:31 AM

factorial(7)

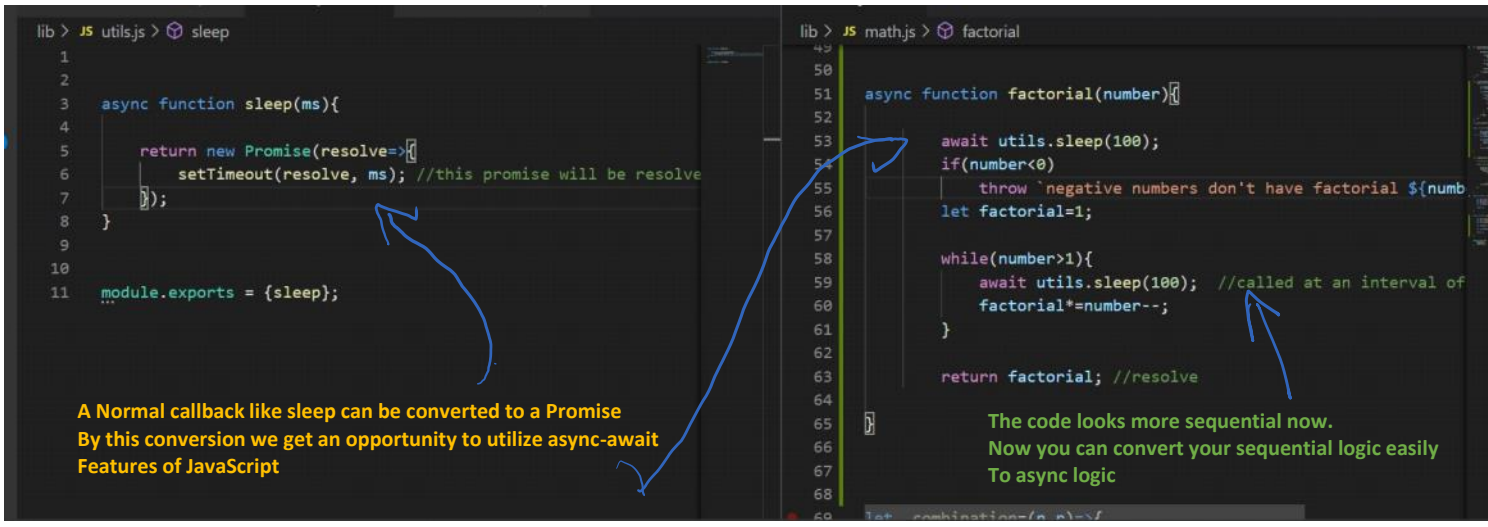
factorial(5)

factorial(2)



Convert Normal Call to Promise

Tuesday, October 13, 2020 11:46 AM



Handle Large Data

Tuesday, October 13, 2020 11:57 AM

Let us revisit our logic to find all primes between 2-500,000

- It takes **roughly** ~44 seconds complete
- It returns a **array** of ~41K+ primes

Use cases -- what will you do after getting 41K primes?

- What are the possible usage of these 41K values?
  - Display all values
  - Save all values to disk
  - Send values across network
  - Calculate the sum of those values
  - Find First 1000 primes ending with 7 eg--> 7,17,37,47,67…
- Think instead of searching for primes, you have searched for products on Amazon or Google
  - Display a list of values
  - Select one of those values

Important Consideration!

- In which of the use cases do you need all those values together?
  - Most of these cases needs values one by one.
- Are you sure you will use all the values
  - After a google/amazon search that returns 100 pages of results, how many pages you actually see?
  - What

Problem

- We may never use the entire data set generated.
- If we use entire dataset we still process **one information at a time**
- **We can't use use the first prime number till we have calculated all the 41K+ prime number**
  - **Can't I use results in smaller chunk and not wait for complete calculation.**

Hadling Large Data Options

Tuesday, October 13, 2020 12:29 PM

We can apply different technqiues

Two important techniques

1. ES2015 generator.
    a. It is like java iterator or c# enumerators
    b.

2. Nodejs Events

Generators

Tuesday, October 13, 2020 12:30 PM

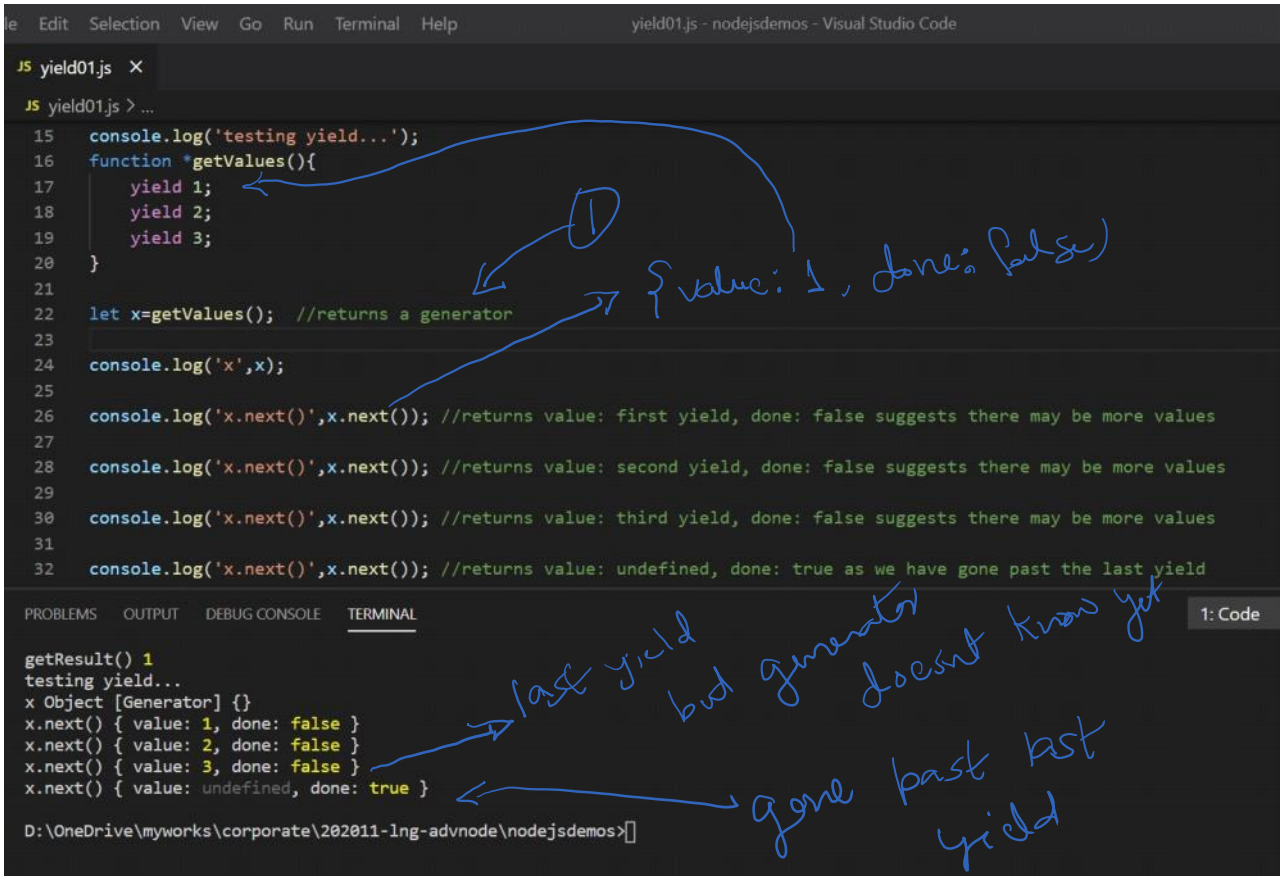
- Javascript has the concept of a generator like C# and Python.
- A generator is based on a new keyword **yield**
- **yield** looks like **return** but works differently

Return statement

```
function getResults(){

        return 1;   // returns 1 and exist the program
        return 2;   //unreachable code
        return 3;  // unreachable code
}


console.log(getResult());  //1
console.log(getResult());  //1
console.log(getResult());  //1
```

Return statement

```
//A function that has yield, must have "*" prefix

function * getResults(){

        yield 1;   // returns 1 and exist the program
        yied 2;   //unreachable code
        yield 3;  // unreachable code
}

let x= getResult(); //you get a result which is not 1
```



eld03.js

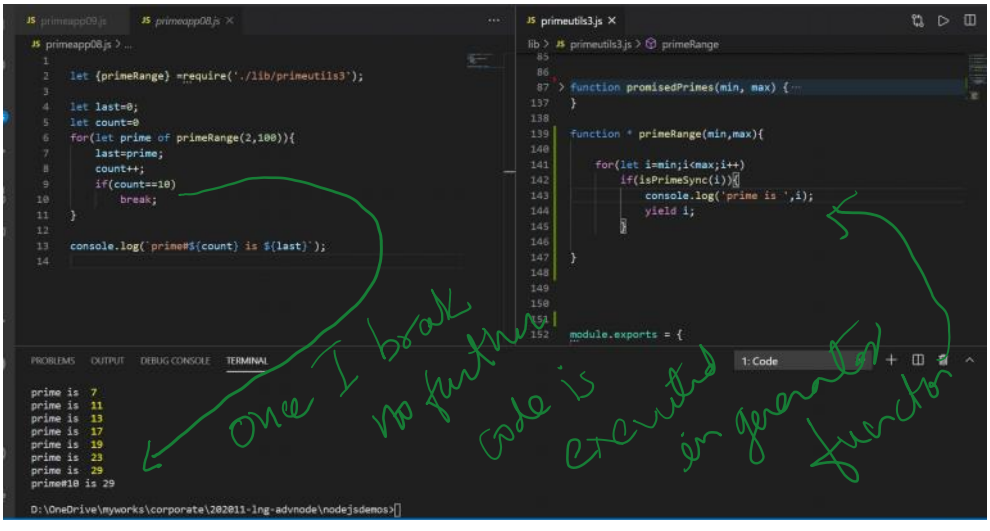
yield03.js > range

```
let range= function *(min,max) {

    console.log('starting the range...');

    for(let x=min; x<max; x++){
        console.log('yeielding ',x);
        yield x;
    }

    console.log('end of range...');
}
let g= range(0,3); //generates 0,1,2

console.log('g',g); //note range function hasn't executed any code yet.

console.log('reaches first yield',g.next());  //here all codes till first yield execute, but no further

//notice we have not completed loop yet. if we don't call next further, no further calculation will happen.

console.log('reaches second yield',g.next()); //executes code till next yield and then wait for another next call

console.log('reaches last yield',g.next());; //this will encounter our last yield, but program hasn't finished yet.

console.log('reaches the end of code',g.next());; //executes the rest of the code to realize that there is no more yield pending.

console.log('once you are past the last line of the code');

console.log('end of code reached by earlier call, so no action here',g.next());; //no more execution as you already gone past last line of
```

Generated Values can easily be stored in an array we need them together using loop or spread operator.

primeapp09.js > ...

```
1
2   let {primeRange} =require('./lib/primeutils3');
3
4   //what if I need array of primes
5   //... spreads any iterator/generator as individual which are collected in a list
6   let primeList= [ ... primeRange(2,100)];
7
8   console.log(primeList);
```



Problem

- If we stop to call next() of a generator, generator code will not execute further
- **BUT IT WILL NOT EXIT EITHER.**
  - **THE GENERATOR FUNCTION WILL REMAIN SUSPENDED**
    - **ALL RESOURCES AND MEMORY ALLOCATED TO IT WILL ALIVE**

Solution -- communicating to generator using next()

- Generator is actually a two way communication!
- We can supply a value to generator function using the call to **next**
- This value is obtained by taking a return from the yield call.

Parameter pass to next()
Can be collected by generator from yield statement.

You may pass signals like

- **Stop**
- **Skip**
- **Reset()**
- **Start**

In our example signal is a call to terminate the generator function

- **Once the function terminates it releases all the resources**

NodeJS Events

Tuesday, October 13, 2020 2:31 PM

- NodeJS has an event mechanism. You code can send information in small chunks to the caller using event rather than return.

Events vs Promises

How are they similar

- An async function may return either Promise or a Events
- User handle the promise in **then()/catch()** and they can listen to events in **on()**

```
function primePromise(){

    return new Promise(…){

        resolve(result_as_bulk);
    };
}

primePromise. then(result_as_build=>doSomething(result_as_bulk))
              .catch(…);
```

```
function primeEvents(){

    Let event=new EventEmitter();
    …
    …
    event.emit( result_chunk);
    return event;

}

primeEvents(). on(  event_chunk => so_something(event_chunk);
```

How are they different

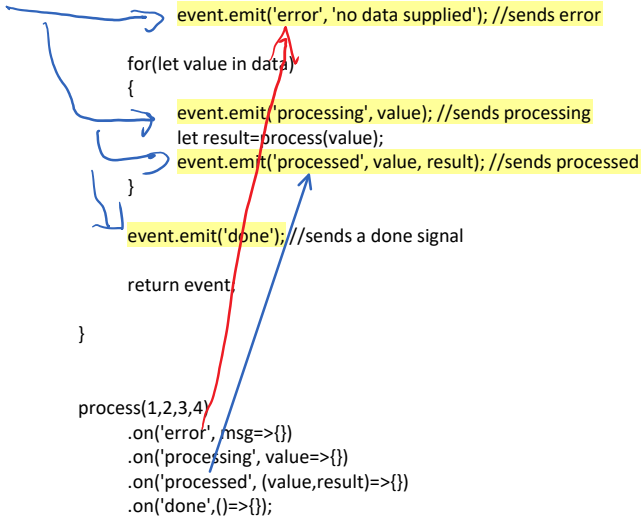
1. **Frequency of call**
   a. **Promise is resolved only once** and returns the entire data in one go.
      i. Not great for large amount data
      ii. Client must wait till entire data is ready
   b. Events can be triggered multiple times
      i. You can send data in small unit multiple times
      ii. You can use fetch and emit loop
      iii. In our example you can emit each prime number one by one
2. **Type of Singals**
   a. **Promise had two fixed types —** resolve and reject
      i. We can't specify what is resolved if there are different type of elements resolved
   b. **Events has no fixed types**
      i. They can define any number of custom events and send different data with each of them.
      ii. There is no separate **reject** equivalent. If error can be considered as a type
3. **Type of object**
   a. **Promise is a ES2015 object available to all javascript programs**
   b. **EventEmitter is a nodejs object which is part of event-emitter module**

Note:

EventEmitter is present in module **event-emitter**

You need to require it

```
function  process( … data){

    let event=new EventEmitter();

    If(data.length==0)
        event.emit('error', 'no data supplied'); //sends error

    for(let value in data)
    {
        event.emit('processing', value); //sends processing
        let result=process(value);
        event.emit('processed', value, result); //sends processed
    }

    event.emit('done');///sends a done signal

    return event;

}


process(1,2,3,4)
    .on('error', msg=>{})
    .on('processing', value=>{})
    .on('processed', (value,result)=>{})
    .on('done',()=>{});
```



```
function primeEvents(){

    Let event=new EventEmitter();
```

```
        …
        …
        event.emit( result_chunk);

        return event;
}
```

Event has already been emitted.

You get event object only when all processing have finished.
It's like inviting you to an event that has ended.

```
function primeEvents(){

        Let event=new EventEmitter();
        …
        setTimeOut(()=>{
                event.emit( result_chunk);
        },0);

        return event;
}


function fetchUrl(url){

        let event=new EventEmitter();

        request.get(url, (err,data)=>{
                if(!err)
                        event.emit('response', data);
        });

        return event;

}
```

Event must be emitted from within some callback that will execute
Sometimes in future after the event object is made available to the client
That callback may be a timed callback or external request callback.
External request callback--

- Talking to OS (file read write)
- Database calls
- Networks calls
- Interprocess communication.

Natural time delays

This event shall be emitted in future

Long after

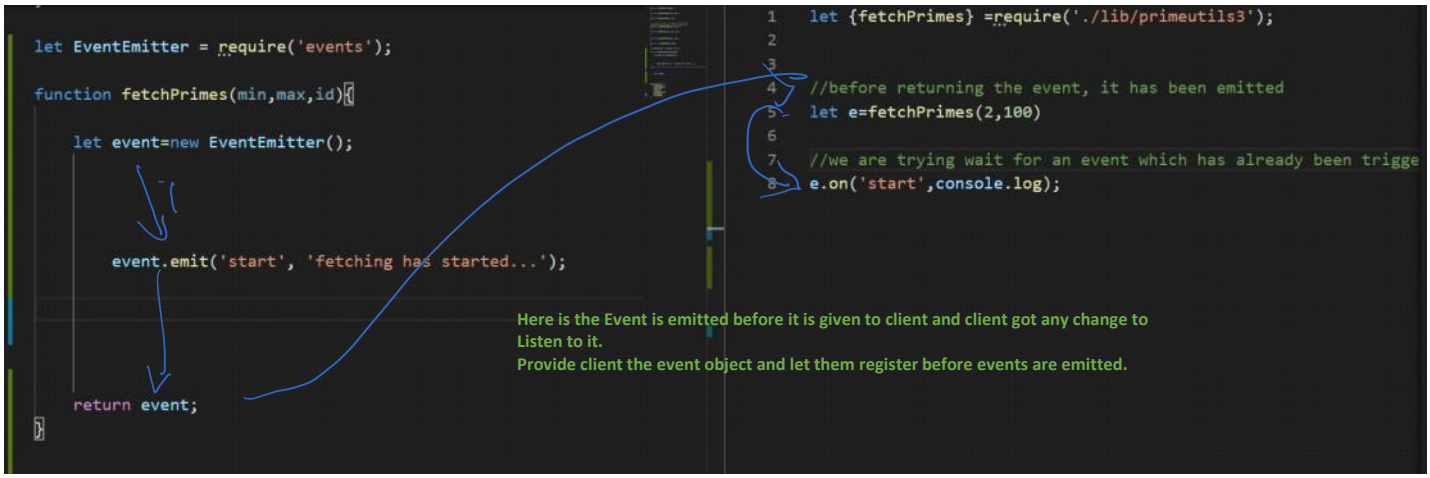
We returned the event object

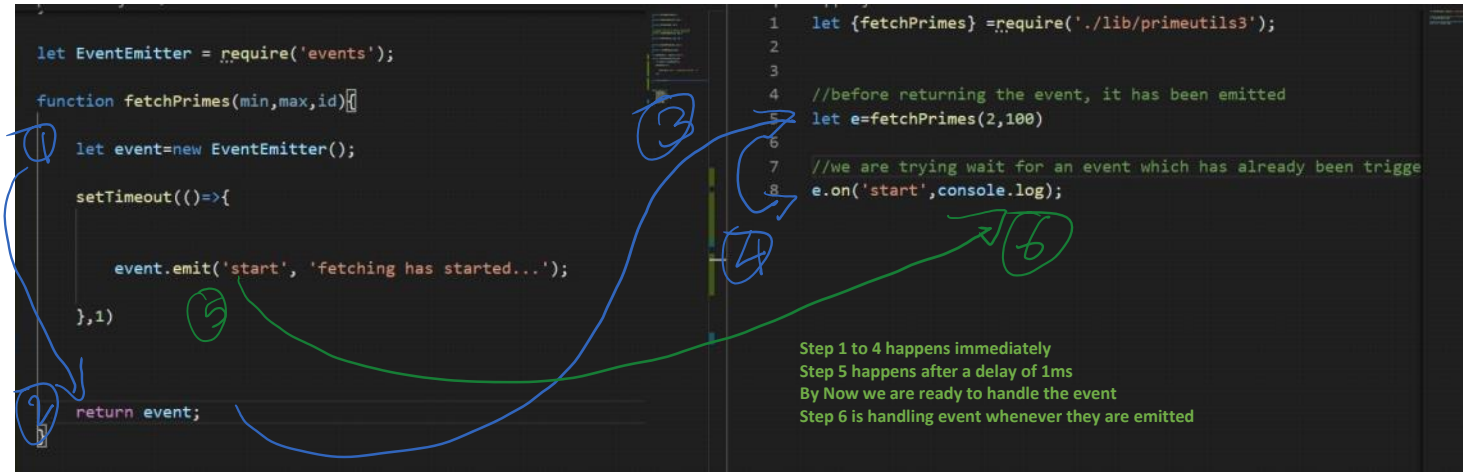
Assignment 07

Tuesday, October 13, 2020 2:52 PM

- Create a function **fetchPrimes** that should be an event based model
  - Should return error as an event
  - The function should take a task id
  - Should return each prime number as they are found with format {id: 1, index:1, prime:2}, {id:2, index=2, prime:3}
  - Should return the progress as an event {id:1, progress:12} <--12% progress
  - Should return completed event

- Write the application to test the events





Assignment 08

Tuesday, October 13, 2020 5:26 PM

- All fetchPrime functions to get aborted on the client request
- Request could be sent through the event emitter

Assignement 09

Tuesday, October 13, 2020 5:27 PM

- Create a js program to read a file and display the progress bar while it is being read
- Verify if all the bytes are read
- Display necessary stats about the file

- Also create a file copy function using createReadStream and createWriteStream

NodeJS Streams

Wednesday, October 14, 2020 10:18 AM

- NodeJS supports the concept of streams.
- There are three broad type of streams
  - Readable Stream
  - Writeable Stream
  - Transform Stream
- Each Stream is typically (like an interface) having a set of
  1. **Standard methods** that should be present in the stream object
  - **A standard set of events** that the stream object may emit.
    - Standard events mean events with a
      - Particular name
      - Particular payload (data that is sent with particular event)

Readable Stream

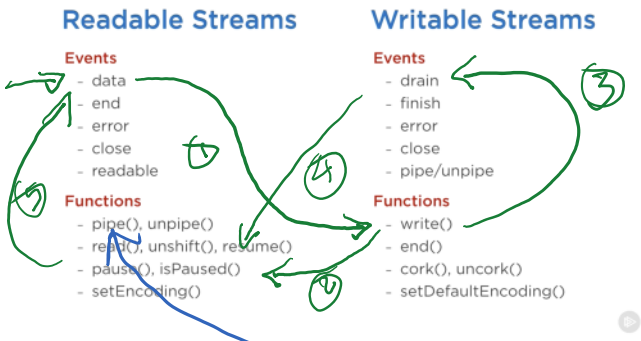
- A Stream from which we can read the data
  - createReadableStream returns stream of data from a file
- It contains
  - Methods
    - read()
      - Read the data from stream
      - Generally done when it is readable
      - Pause()
        - Pause the reading
      - Resume()
        - Resume the reading
    - close()
  - Events
    - 'data'
      - Tells some data is available for reading
    - 'end'
      - Tells we have reached the end of our stream
    - 'error'
      - Informs about the error
    - 'close'
      - Stream has been closed
    - 'readable'
      - Stream is ready to be read

Writeable Stream

Events

- **drain**
  - We have consumed the data earlier supplied
    - It has been written
    - We are ready for more data

Communication between ReadStream and WriteStream



Stream Dance

Pipe()

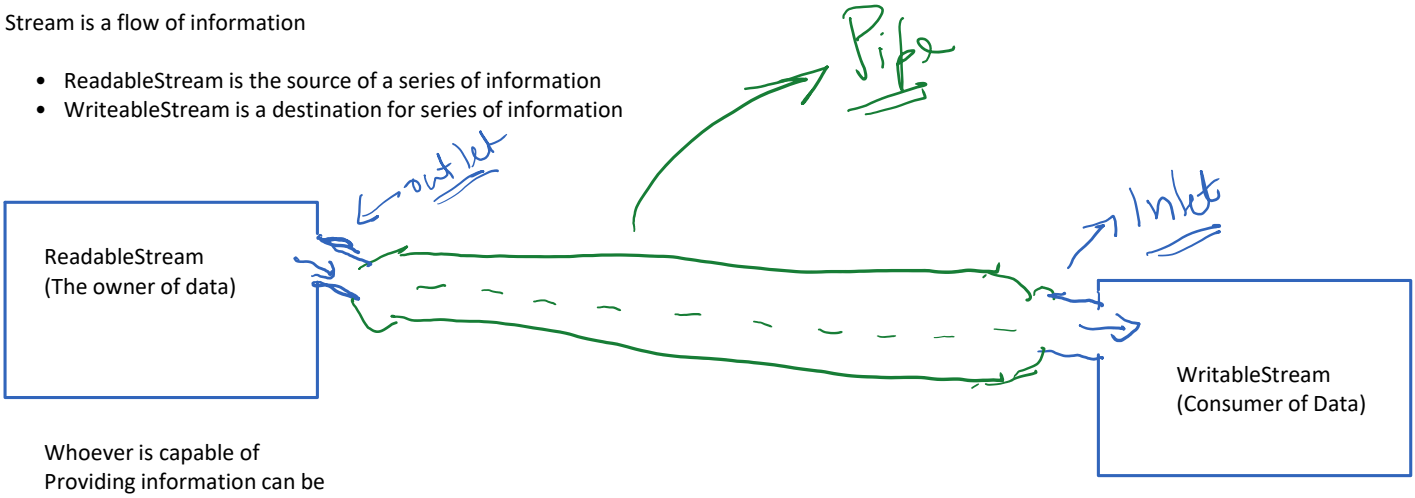
- The **Stream dance** can be automated by using the function **pipe()** on **ReadableStream** which takes a **WritableStream as a parameter**

What is a Stream

Wednesday, October 14, 2020 12:21 PM

Stream is a flow of information

- ReadableStream is the source of a series of information
- WriteableStream is a destination for series of information



Whoever is capable of
Providing information can be
Considered as a ReadableStream

Examples:

1. File
2. Network
3. **Computed Data Source**
4. Standard Input Device

Example
1. File
2. Network
3. Standard output device

Can I consider a process returning a series (like primes) be a ReadableStream?
YES

To be a Stream instead of returning custom events, we need to have standard events like data and end

How to create custom Readable Streams

1. Create your own type that extends Readable type
   a. Chain constructor call
   b. Copy prototype

```
let PrintStream=function(min,max){
    //Javascript Inheritance Step 1 -- chain constructor
    Readbale.call(this);  //chain the constructor to the super

    //user initilazation here


}

//copy the prototype of Redable type to PrintStream type, so th
//PrintStream
util.inherits(PrintStream, Readable); //PrintStream gets all me
```

2. Define _read() method to emit
    a. 'data'
    b. 'end'
    c. 'error'

3. You may translate your custom events to standard events

```
PrintStream.properties._read =function(){

    let self=this; //generall this will be lost in nested callb

    fetchPrimes(this.min,this.max)
        //my api sends 'PRIME', but Readable is supposed send '
        .on('PRIME',data=>{
            //create a buffer from the json data you have
            let buffer= Buffer.from(JSON.stringify(data));
            self.emit('data',buffer);
        })
        .on('FINISHED',()=>{
            self.emit('end');
        })
        .on('ERROR',(error)=>{
            let buffer=Buffer.from(JSON.stringify(error));
            self.emit('error',buffer);
        });
```

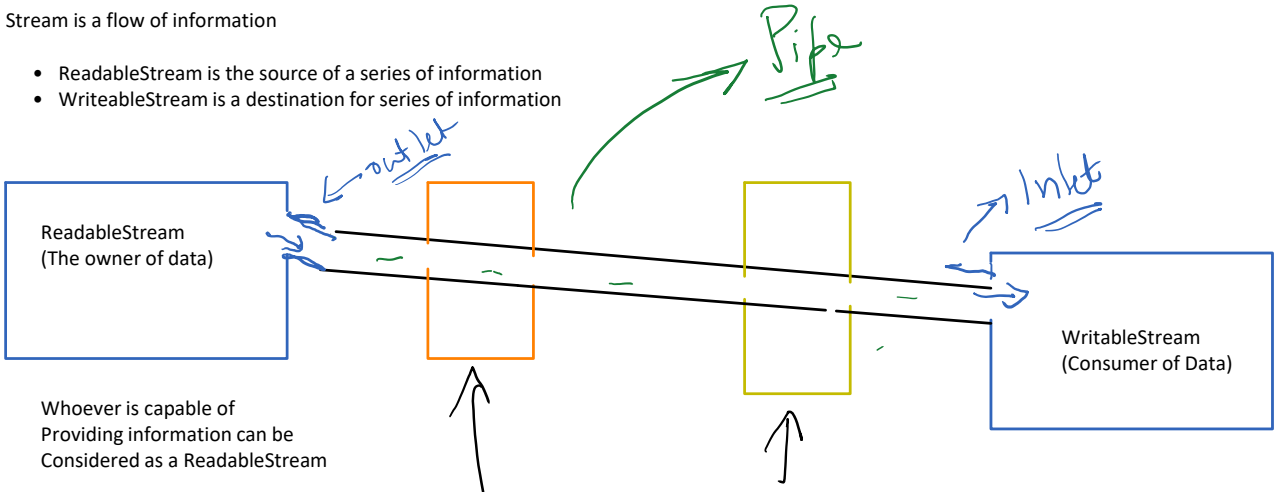
1. Define _read that should eventually emit
    a. data
    b. end
    c. Error
2. Send data as buffer and not as plain data
3. Translate custom events to standard events

Transform Stream

Wednesday, October 14, 2020 12:21 PM

Stream is a flow of information

- ReadableStream is the source of a series of information
- WriteableStream is a destination for series of information



Whoever is capable of
Providing information can be
Considered as a ReadableStream

Filter
That can allows passing
Selected items forward
And remove the rest of them

- Eg. Water purifier

Transformer

Converts the value
Before forwarding

- Eg. Water header

TransformStream

- They have properties of both Readable and Writeable
- They can receive the data using WriteableStream and forward data as RedableStream
- They can be added to pipe making a chain

e.g.

```
getEmployeess()
        .pipe(filter(emp=> emp.dept=='training'))
        .pipe(filter(emp=>emp.salary>100000))
        .pipe(converter(convertToXml))
        .pipe(converter(zipCompress)
        .pipe(fs.createWriteStream())
```

1. Get a list of all employess
2. Pipe it to a filter that takes employees from training department
3. Pipe to another filter that takes employee with a min salary
4. Covert data to xml
5. Compress xml
6. Write to a file

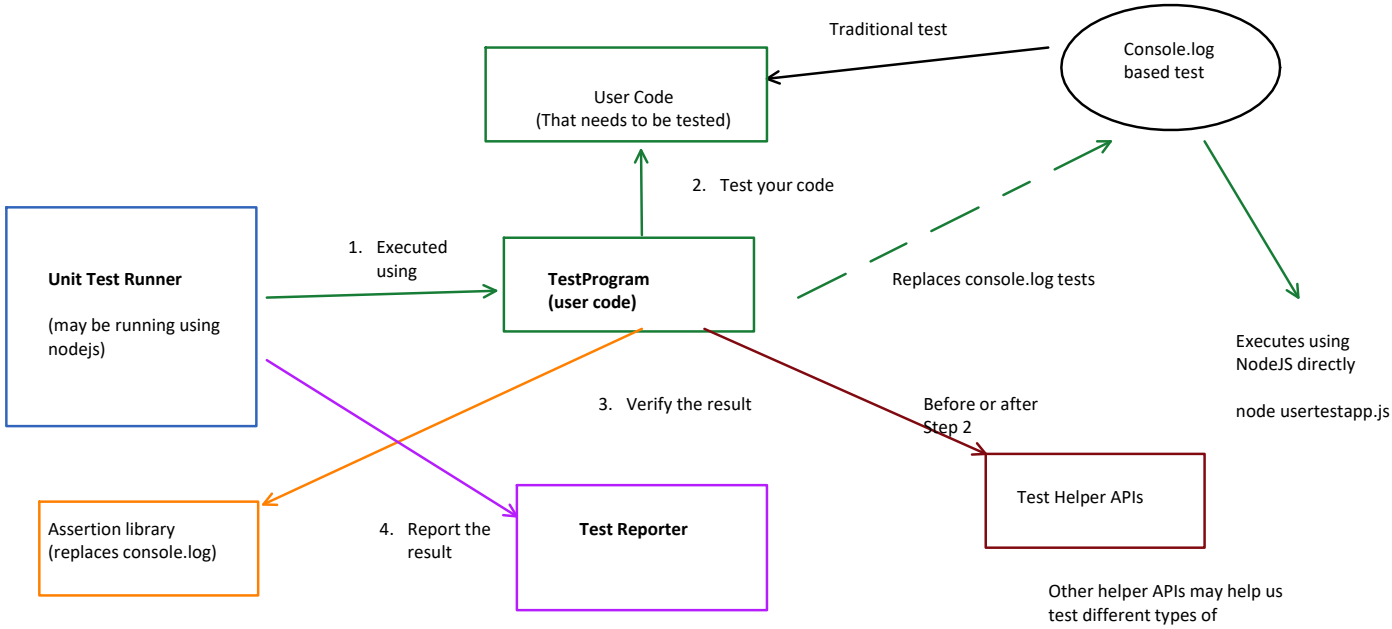
Convert to Transform

```
let Converter=function(convertFunction){

    //Fixed Step 1:chain the constructor
    Transform.call(this);
    this.convertFunction=convertFunction; //this function will actually be used over the stream
}
//Fixed Step 2: inherits
util.inherits(Converter, Transform);

//Fixed Step 3: overwrite the required method

Converter.prototype._transform = function(chunk, enc, cb){

    let original=chunk.toString() ; //convert buffer into data

    let covertedValue= this.convertFunction(original); //covert input data to desired type

    let outputBuffer= Buffer.from(covertedValue.toString()); //create a new buffer

    this.push(outputBuffer); //send it to the client

    cb(); //inform the system that convert is complete
};
```

N

Unit Testing Component

Wednesday, October 14, 2020 2:23 PM



Instead of displaying the result on screen (which only humans can read)

An assert call verifies if we meet certain condition and throws and exception if Expectation is not met

The Test Runner runs the tests and check which expectation pass and which one fails

How will be displaying the result of the test

- It could be as simple as: **console.log**
- Use some logger api
- Display in an html file
- Save it to log file
-

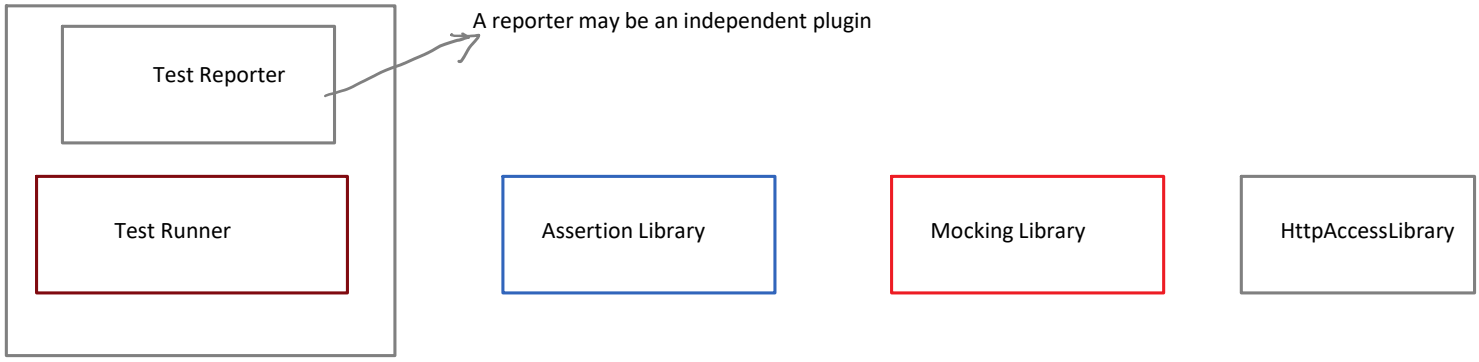
Other helper APIs may help us test different types of elements.

1. Mocking API
2. Network Access API

Unit Testing framework

Wednesday, October 14, 2020 2:44 PM

- Provides one or more re-usables compoent required for unit Testing
- All these components may not be part of the same product
- We may use different protects at different level



Generally a test runner includes

- The runner
- The Reporter (customizable)

Popular Prouduct in this domain

- **Mocha**
- **Karma**
- **Jest**

Popular products

- **Nodejs builtin assertions**
- **Chai**
- **Should**
- **Jasmine**
- **Jest**

Popular products

- **Sinon**
- **should**
- **Jest**

Popular Product

- **supertest**

Note:

- A Testing framework may come with more than one builtin elements
  - Example **Jest**
    - **Is a test runner but also includes**
    - **OsAssertion library**
    - **Mocking library**
    - **Snapstho library**
- **Most test runners can work with most of the other libraries**
  - **Example**
    - **Jest can also use assertions from Chai/Should**
    - **Can use mocking using sinion**
    - **supertest**

Wednesday, October 14, 2020 2:52 PM

o