# Readable Node.js Code Using JavaScript Promises

Norman Wilde
http://normanwilde.net

```
Promise.resolve().
   then(() => /* do A */ ...).
   then(() => /* do B */ ...).
   then(() => /* do C */ ...).
   then(() => /* do D */ ...).
   catch(() => /* do error */ ...);
```

# The author:

Writing programs for 50+ years

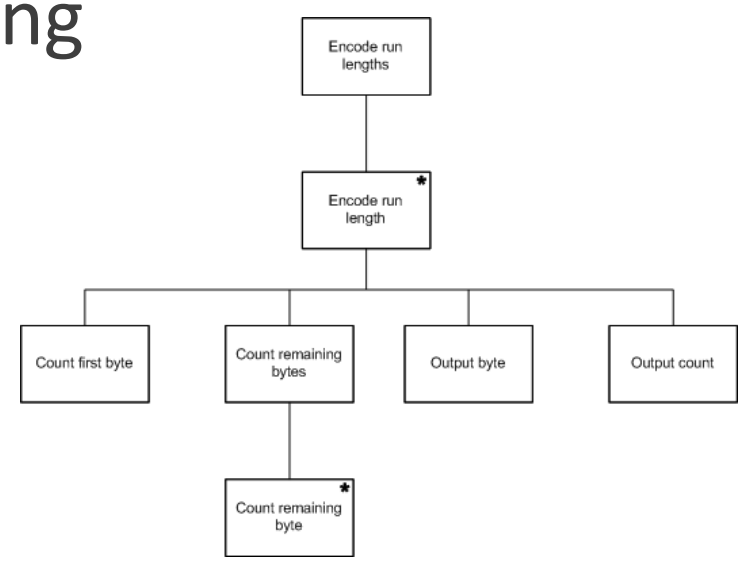
Teaching Software Engineering for 30+ years

Doing  node.js for just 18 months!

*So what follows is just my opinion ...*

# Good Code === Maintainable Code
## Maintainable Code ~ Readable Code

- Modularity - High Cohesion, Low Coupling
- Simple, Readable Style:
  - Organization
  - Data and Function Names
  - Comments to Explain Purpose



# Doing Tasks in Sequence

## Traditional Program - Call and Return

```
function doABCD(){
 doA();
 doB();
 doC();
 doD();
}
```

## Traditional JavaScript - Callbacks

```
function doABCD(callback){
 taskABCD(() => {
  // do A, then do BCD
  taskBCD(() => {
   // do B, then do CD
   taskCD(() => {
    // do C, then do D
    taskD(() => {
     // do D, then finish
     callback(result);
    })
   })
  })
 })
}
```

# Doing Tasks in Sequence

## Traditional Program - Call and Return

```
function doABCD(){
 doA();
 doB();
 doC();
 doD();
}
```

Bad Performance for Microservices

## Traditional JavaScript - Callbacks

```
function doABCD(callback){
 taskABCD(() => {
  // do A, then do BCD
  taskBCD(() => {
   // do B, then do CD
   taskCD(() => {
    // do C, then do D
    taskD(() => {
     // do D, then finish
     callback(result);
    })
   })
  })
 })
}
```

# Doing Tasks in Sequence

**Traditional Program - Call and Return**

```
function doABCD(){
 doA();
 doB();
 doC();
 doD();
}
```

**Traditional JavaScript - Callbacks**

```
function doABCD(callback){
 taskABCD(() => {
  // do A, then do BCD
  taskBCD(() => {
   // do B, then do CD
   taskCD(() => {
    // do C, then do D
    taskD(() => {
     // do D, then finish
     callback(result);
    })
   })
```

Unreadable
ESPECIALLY when Error
Handling is Added

# The Solution: "Promise Chains"?
## *Code that looks like this?*

```
function doABCD_Promise(){
    Promise.resolve().
    then(() => /* do A */ ...).
    then(() => /* do B */ ...).
    then(() => /* do C */ ...).
    then(() => /* do D */ ...).
    catch(() => /* error */ );
}
```

*Initial Promise*

*Do these functions in sequence*

*Handle errors here*

# JavaScript Promises
## *A Complex Mechanism*

### Doing Three Things

1. **Sequencing** - asynchronous and synchronous tasks
2. **Error handling** - in the *catch()*
3. **Passing data** - from earlier tasks to later ones

Standardized at **Promises/A+**
https://promisesaplus.com/

*Not an easy read!*

# Keeping it Simple:
## My *KISS* Suggestions

1. Encapsulate the Promise chain in a class
2. For each task in the chain:
   a) If synchronous, return null or throw an exception
   b) If asynchronous, instantiate and return a new Promise object
3. Avoid passing data down the chain

# Why Encapsulate the Promise?

*Because promise code can get ugly*

```
.then(() => {
    http.get(testUrl).on('error', (e) => {
        errorMessageOnGet = e.message;
        errorOnGet = true;
    });
}).
then(() => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if (!errorOnGet) {
                resolve();
            } else {
                reject(errorMessageOnGet);
            };
        }, ERROR_WAIT);
    });
}).
```

- Functions tangled in the chain
  - Anonymous
  - Uncommented
  - Can't unit test

*Hard to read!*

# Encapsulate Promise Chain in a Class

*Example: A test driver class*

```
const sst = new SSTester("foo.com", ...);
Promise.resolve().
then( ()=> sst.setupTest("test01",...) ).
then( ()=> sst.doGet() ).
then( ()=> sst.checkResult(...) ).
then( ()=> sst.writeResult() ).
then( ()=> console.log("test01 done") ).
// ... more tests here
catch((err)=> console.log("Error ...") ).
finally(()=> console.log("Ended tests") );
```

- Each task calls one member function
  - Function names provide documentation
- Calling code is simple
- Comments can go in the class
  - Just like normal OOP
- Unit test the class

# The Class Looks As Usual

*Using TypeScript, but ECMAScript 2015 (ES6) will do*

```
import http = require("http");
//// more imports as needed
/**
 * Driver class for integration tests
 */
export class SSTester {
    //// commented data members
    host: string; // host part of the URL
    port: string; // port number for URL
    //// constructor
    constructor(theHost:string,
      thePort:string) {
        this.host = theHost;
        this.port = thePort;
    } // end constructor
```

```
 //// commented member functions
    /**
     * Initialize new service test
     * @param tId - test Identifier
     * @param tPath - test path
     */
    setupTest(tId:string, tPath: string) {
        // ...
        return null;
    } // end setupTest()
    //// etc.
} // end class SSTester
```

*For each task in the chain*

# Is it Asynchronous or Synchronous?

- Asynchronous tasks are long-running
  - e.g. read a file, call a web service
  - Don't block the node event loop!
  - The API's provide *callback* function to handle results

```
S3.headBucket(params, (err, data) => {
    if(err) /* handle error */;
    else /* do something with data*/;
})
```

*Checking status of an AWS S3 Bucket*

- Synchronous tasks are short-running
  - e.g. do a computation
  - You implement with conventional procedural code

```
let sum = 0;
myArray.forEach((x)=> {
    sum += x;
});
```

*Sum up an array*

*That determines*

# What Goes in *p.then( … ??? … )*

- You provide a function
- It contains the code for the task
- **The type of its return value determines how *then()* works!**

```
Promise.resolve().
then( (args) => {
  /* code for this task */
  return retval;
}).
then( ... ).
```

Is **retval** a Promise? Or is it of some other type?

# For an Asynchronous Task - A Promise

## *My suggestion - there are other approaches*

- Wrap your code in a function inside a new Promise object
- The function takes two function arguments *resolve* and *reject*
  - These are the "hooks" to say when task is complete
  - Your code calls *resolve()* on success; calls *reject()* on an error
- **Return the Promise object**
- then() stops the chain until you call *resolve* or *reject**

* This is grossly simplified. See https://promisesaplus.com/ for the full story

# Example of Asynchronous Task: Write to S3

*Call as: then( () => sst.writeResult() )*

```
writeResult(): Promise<any> {
        let wPromise = new Promise((resolve, reject) => {
            let params = {
                Body: "",
                Bucket: this.s3bucketName,
                Key: key,
            };
            S3.putObject(params, function (err, data) {
                if (err) reject(err); // an error occurred
                else resolve(null);   // successful response
            });
        });
        return wPromise;
    } // end writeResult
```

# Example of Asynchronous Task: Write to S3

*Call as: then( () => sst.writeResult() )*

```
writeResult(): Promise<any> {
        let wPromise = new Promise((resolve, reject) => {
            let params = {
                Body: "",
                Bucket: this.s3bucketName,
                Key: key,
            };
            S3.putObject(params, function (err, data) {
                if (err) reject(err); // an error occurred
                else resolve(null);   // successful response
            });
        });
        return wPromise;
    } // end writeResult
```

The writeResult() function passed to then() creates a new Promise object

# Example of Asynchronous Task: Write to S3

*Call as: then( () => sst.writeResult() )*

```
writeResult(): Promise<any> {
        let wPromise = new Promise((resolve, reject) => {
            let params = {
                Body: "",
                Bucket: this.s3bucketName,
                Key: key,
            };
            S3.putObject(params, function (err, data) {
                if (err) reject(err); // an error occurred
                else resolve(null);   // successful response
            });
        });
        return wPromise;
    } // end writeResult
```

The constructor takes a function that has two function arguments

# Example of Asynchronous Task: Write to S3

*Call as: then( () => sst.writeResult() )*

```
writeResult(): Promise<any> {
        let wPromise = new Promise((resolve, reject) => {
            let params = {
                Body: "",
                Bucket: this.s3bucketName,
                Key: key,
            };
            S3.putObject(params, function (err, data) {
                if (err) reject(err); // an error occurred
                else resolve(null);   // successful response
            });
        });
        return wPromise;
    } // end writeResult
```

The body of the function performs your asynchronous task

# Example of Asynchronous Task: Write to S3

*Call as: then( () => sst.writeResult() )*

```
writeResult(): Promise<any> {
        let wPromise = new Promise((resolve, reject) => {
            let params = {
                Body: "",
                Bucket: this.s3bucketName,
                Key: key,
            };
            S3.putObject(params, function (err, data) {
                if (err) reject(err); // an error occurred
                else resolve(null);   // successful response
            });
        });
        return wPromise;
    } // end writeResult
```

The callback from the task calls resolve() or reject()

# Example of Asynchronous Task: Write to S3

*Call as: then( () => sst.writeResult() )*

```
writeResult(): Promise<any> {
        let wPromise = new Promise((resolve, reject) => {
            let params = {
                Body: "",
                Bucket: this.s3bucketName,
                Key: key,
            };
            S3.putObject(params, function (err, data) {
                if (err) reject(err); // an error occurred
                else resolve(null);   // successful response
            });
        });
        return wPromise;
    } // end writeResult
```

writeResult returns the new Promise

# Example of Synchronous Task

*Call as: then( () => sst.checkResult() )*

- Regular procedural code
- Return anything **but** a Promise

I suggest `null`

```
checkResult(matchString: string): any {
        if (-1 == this.testReturned.indexOf(matchString)) {
            this.testPassed = false;
        } else {
            this.testPassed = true;
        }
        return null;
    } // end checkResult
```

checkResult returns null

# To Summarize My Suggestions

*For each task in the chain*

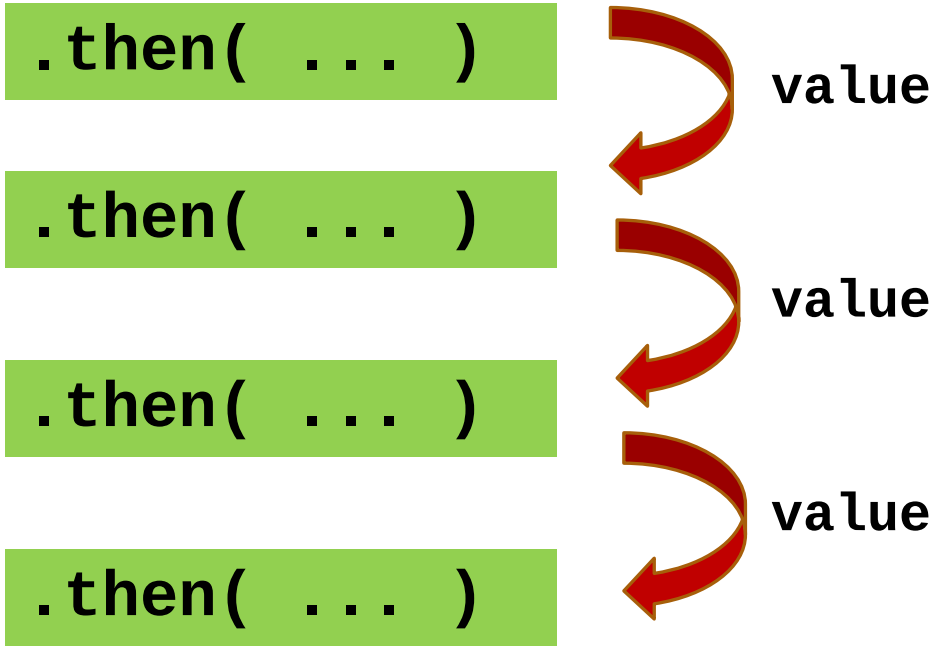
- For an asynchronous task
  - Instantiate a Promise object that wraps a function that does the task
  - When the task completes (in its callback) call resolve() or reject()
  - Return the Promise object
- For a synchronous task
  - Just write regular procedural code
  - Return anything **but** a Promise - I suggest *null*

# Passing Data Down the Chain

*results from one task -> later tasks*

The Promise "value" from one task can be passed on

**I suggest you DON'T do this!**



# Here's How

*if you REALLY insist ...*

Function in first `then()`

```
.then( ... )
```

Asynchronous Case

```
...
resolve(value);
...
```

Synchronous Case

```
...
return value
...
```

Function in following `then()`

```
.then( ... )
```

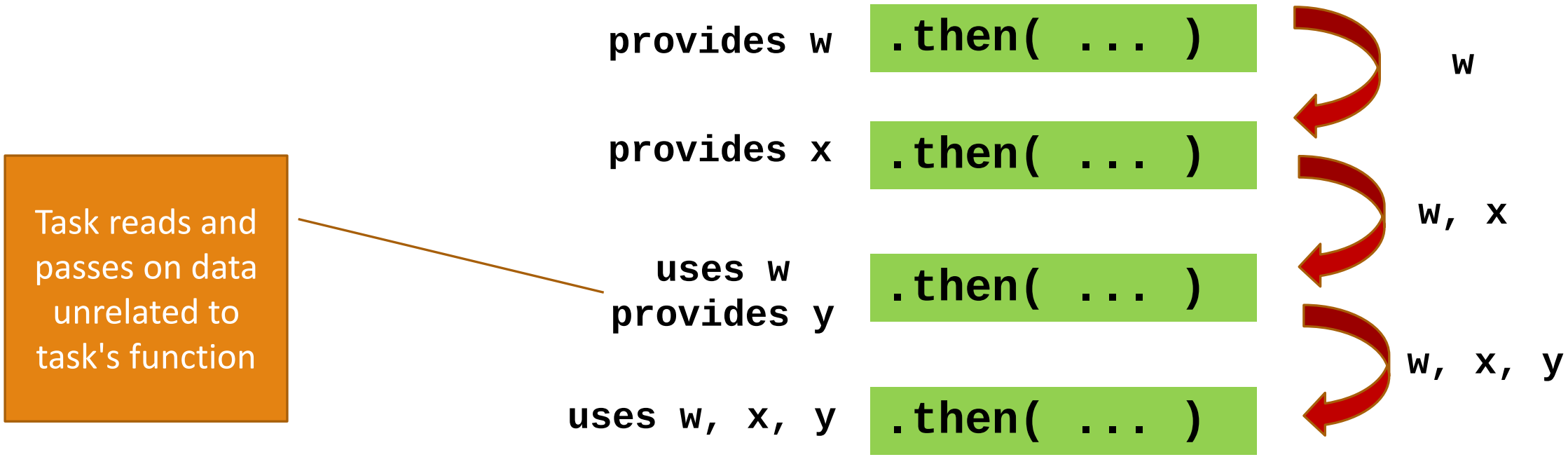
```
(x) => {
  // x has value
}
```

**`value`** is passed as first argument of the function

# Why I Don't Like This
## *spooky action at a distance …*

- Hard to trace data flow
- Poor coupling, poor information hiding
  - First function needs to know exactly what the second needs
  - The second function needs to know what the first will provide
  - Both are thus locked in to a particular place in a specific Promise chain

# It Get's Uglier as the Chain Gets Longer



*My Suggestion*

# Use the Object to Pass Data

```
const sst = new SSTester("foo.com", ...);
Promise.resolve().
then( ()=> sst.setupTest("test01",...) ).
then( ()=> sst.doGet() ).
// etc.
```

Sets the value of `testQuery`

```
this.testQuery = tQuery;
```

**Easier to track the data flow**

Uses the value of `testQuery`

```
if (this.testQuery !== null) {
        url += "/";
        url += this.testQuery;
    }
```

# The take away suggestions:

1. Encapsulate the Promise chain in a class
2. For each task in the chain:
   a) If synchronous, return null or throw an exception
   b) If asynchronous, instantiate and return a new Promise object
3. Avoid passing data down the chain

# To go deeper:

- Sam Roberts, *Node's Event Loop from the Inside Out*, Node Summit 2017, https://vimeo.com/229535344
- MDN Web Docs, *Making asynchronous programming easier with async and await*, https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Async_await
- Adam Barr, *The Problem With Software: Why Smart Engineers Write Bad Code*, MIT Press, 2018, ISBN-13: 978-0262038515

# Thank You!