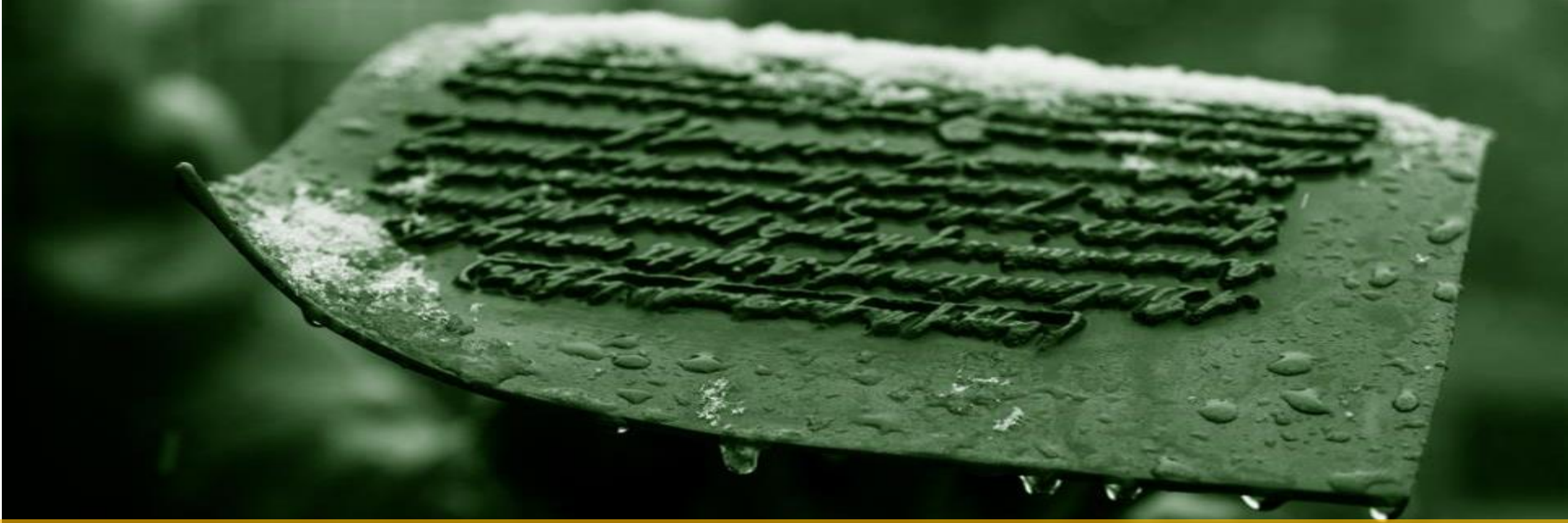


MODULE 3 – NODEJS ASYNCHRONOUS PROGRAMMING TECHNIQUES

IT 207 – IT Programming

GEORGE MASON UNIVERSITY

Lecture Outline

1. Callback functions
2. Asynchronous File API
3. Event listeners and event emitters
4. Streams and pipes

Asynchronous programming in the Web

- Responding to events

Any action in a running application such as opening a file , making a network connection etc.

  - In the browser at the frontend for interacting with elements in a webpage displayed in the browser
    - Clicking a button
    - Checking a box
    - Etc...
  - In **Nodejs** at the Backend for executing logic in response to **blocking** events
    - Querying a database
    - Handling file access
    - Etc...

NODEJS MODEL FOR CONCURRENCY

- Nodejs supports concurrency via
  - Callback functions
  - Event emitters and event listeners

CALLBACKS

What is a callback?

- Callbacks are functions that define logic for one-off response to an event.

Any action in a running application such as opening a file , making a network connection etc.

What is a callback?

- Callbacks are functions that define logic for one-off response to an event.
- A callback function is passed as an argument to an asynchronous function
- A callback function describes what to do when the asynchronous operation has been completed.
- A callback is usually associated to an event and will only be executed when the event happens

TIMERS – AN EXAMPLE FOR A CALLBACK:

- JavaScript defines the `setTimeout` function at the global scope.
- `setTimeout` sets a timer which executes a function or a specified piece of code once the timer expires.
- `setTimeout` syntax:

```
setTimeout(functionRef)
setTimeout(functionRef, delay)
```

  - delay is measured in milliseconds

What is the Output?

Callback implemented as an arrow function

```
setTimeout(()=>{console.log("I've done my work!")}, 2000);

console.log("I'm waiting for all my work to finish.");
```

An **Arrow** function is the new and shorter way of declaring an anonymous function,

An **Anonymous** function is a function that does not have any name associated with it.

What is the Output?

Callback implemented as an anonymous function

```
setTimeout(()=>{console.log("I've done my work!")}, 2000);

console.log("I'm waiting for all my work to finish.");
```

I'm waiting for all my work to finish.
I've done my work!

Callback Pattern in NodeJS

- Callback functions always have at least one parameter that indicates the success or failure status of the last operation.
  - For success the value returned will be null
  - For failure the value returned will be an instance of the Error object class.

- A callback function can have other parameter(s) that hold the result(s) of the operation
  - Results of a database query
  - Content of a file.
  - Etc.

FILESYSTEM ASYNCHRONOUS CALLBACK APIS

Asynchronous vs Synchronous Filesystem APIs

- The asynchronous callback APIs are equivalent to the synchronous APIs with all functions having a callback as the *last* parameter

- Asynchronous
  - Dir Functions
    - fs.mkdir(path[, options], callback)
    - fs.readdir(path[, options], callback)
    - fs.rmdir(path[, options], callback)
  - File Functions
    - fs.readFile(path[, options], callback)
    - fs.read(fd, buffer, offset, length, position, callback)
    - fs.read(fd, buffer[, options], callback)
    - fs.writeFile(file, data[, options], callback)
    - fs.write(fd, buffer, offset[, length[, position]], callback)
    - fs.write(fd, buffer[, options], callback)
    - fs.write(fd, string[, position[, encoding]], callback)
    - fs.appendFile(path, data[, options], callback)

- Synchronous
  - Dir Functions
    - fs.mkdirSync(path[, options])
    - fs.readdirSync(path[, options])
    - fs.rmdirSync (path[, options])
  - File Functions
    - fs.readFileSync(path[, options])
    - fs.readSync(fd, buffer, offset, length[, position])
    - fs.readSync(fd, buffer[, options])
    - fs.writeFileSync(file, data[, options])
    - fs.writeSync(fd, buffer, offset[, length[, position]])
    - fs.writeSync(fd, buffer[, options])
    - fs.writeSync(fd, string[, position[, encoding]])
    - fs.appendFileSync(path, data[, options])

Check Nodejs documentation for function usage and details

Reading from a File

- fs.readFile(path[, options], callback)
  - Asynchronously reads the entire contents of a file.
  - The callback is passed two arguments (err, data), where data is the contents of the file.
  - If no encoding is specified, then the raw buffer is returned.
  - If encoding is specified, then a string is returned

fs.readFile(path[, options], callback)

▶ History

- `path` `<string>` | `<Buffer>` | `<URL>` | `<integer>` filename or file descriptor
- `options` `<Object>` | `<string>`
  - `encoding` `<string>` | `<null>` **Default:** `null`
  - `flag` `<string>` See support of file system `flags`. **Default:** `'r'`.
  - `signal` `<AbortSignal>` allows aborting an in-progress readFile
- `callback` `<Function>`
  - `err` `<Error>` | `<AggregateError>`
  - `data` `<string>` | `<Buffer>`

Reading from a File

- fs.readFile(path[, options], callback)
  - Asynchronously reads the entire contents of a file.
  - The callback is passed two arguments (err, data), where data is the contents of the file.
  - If no encoding is specified, then the raw buffer is returned.
  - If encoding is specified, then a string is returned

```
const fs = require('fs');
//Read the content of the file as one big chunk into a buffer
//Path: String – Default: sequence of bytes
fs.readFile("./Source.txt", (err, content)=>{
if (err) throw err;
console.log(content);
});
```

Reading from a File

- fs.readFile(path[, options], callback)
  - Asynchronously reads the entire contents of a file.
  - The callback is passed two arguments (err, data), where data is the contents of the file.
  - If no encoding is specified, then the raw buffer is returned.
  - If encoding is specified, then a string is returned

```
const fs = require('fs');
//Read the content of the file as one big chunk into a buffer
//Path: String – Default: sequence of bytes
fs.readFile("./Source.txt", (err, content)=>{
if (err) throw err;
console.log(content);
});
```

Output

<Buffer 56 69 64 65 6f 20 70 72 6f 76 69 64 65 73 20 61 20 70 6f 77 65 72 66 75 6c 20 77 61 79 20 74 6f 20 68 65 6c 70 20 79 6f 75 20 70 72 6f 76 65 20 79 6f ... 489 more bytes>

Reading from a File – Using a File Descriptor

- fs.readFile(path[, options], callback)

```
fs.open("./Source.txt", "r", (err, fd)=>{ //fd is returned as a parameter
  if (err) throw err;
  console.log('File descriptpor = '+fd); //print out the fd
  fs.readFile(fd,{encoding:"utf-8",flag:"r"}, (err, content)=>{ //content is a String
    if (err) throw err;
    console.log(content);
    fs.close(fd,(err)=>{
      if (err) throw err;
      console.log('file closed successfully!!');
    });
  });
});
```

Reading from a File – Using a File Descriptor

- fs.readFile(path[, options], callback)

```
fs.open("./Source.txt", "r", (err, fd)=>{ //fd is returned as a parameter
  if (err) throw err;
  console.log('File descriptor = '+fd); //print out the fd
  fs.readFile(fd,{encoding:"utf-8",flag:"r"}, (err, content)=>{ //content is a String
    if (err) throw err;
    console.log(content);
    fs.close(fd,(err)=>{
      if (err) throw err;
      console.log('file closed successfully!!');
    });
  });
});
```

Output

```
File descriptor = 20
Video provides a powerful way to help you prove your point.
When you click Online Video, you can paste in the embed code for the video you want
to add. You can also type a keyword to search online for the video that best fits your
document. To make your document look professionally produced, Word provides
header,
footer, cover page, and text box designs that complement each other.
For example, you can add a matching cover page, header, and sidebar.
Click Insert and then choose the elements you want from the different galleries.
file closed successfully!!
```

Reading from a File – Using a File Descriptor

- fs.readFile(path[, options], callback)

```
fs.open("./Source.txt", "r", (err, fd)=>{ //fd is returned as a parameter
  if (err) throw err;
  console.log('File descriptpor = '+fd); //print out the fd
  fs.readFile(fd,{encoding:"utf-8",flag:"r"}, (err, content)=>{ //content is a String
    if (err) throw err;
    console.log(content);
    fs.close(fd,(err)=>{
      if (err) throw err;
      console.log('file closed successfully!!');
    });
  });
});
```

Note the following:

- Nesting of callbacks can lead to an unreadable and not easy to manageable codebase commonly known as callback hell in Node. js or pyramid of doom.
- The dependence of each file operation on the one before it

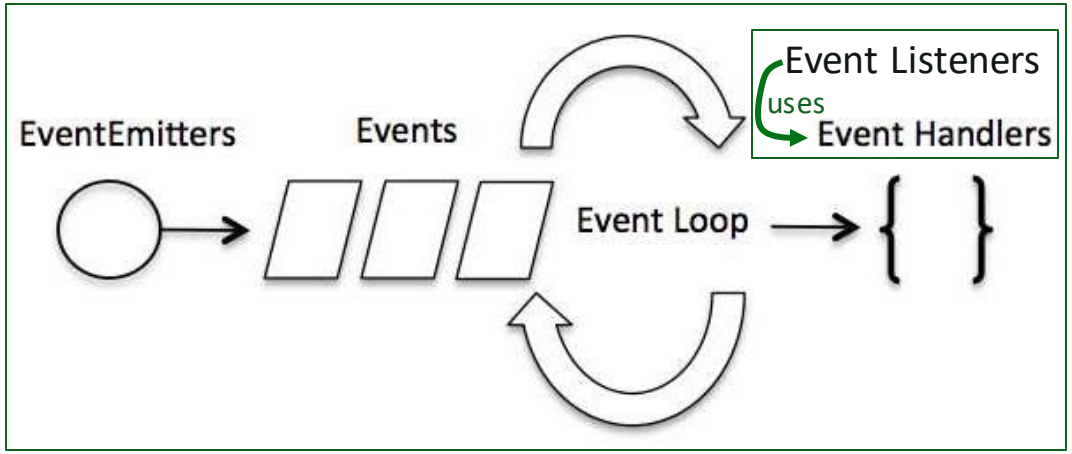
EVENT EMITTERS AND EVENT LISTENERS

Event Emitters and Event Listeners

- Nodejs defines an Event module that allows Nodejs programmers to manipulate their application logic through events.
  - Observer Pattern

Any action in a running application such as opening a file , making a network connection etc.

- The Event module defines two main artefacts: EventEmitters and EventListeners.
- EventEmitters are objects that fire events and include the ability to handle those events when triggered through EventListeners
- EventListeners are callback functions that are associated with an event such that each time an event occur the EventListener is executed.



Observer Pattern

Programming with Event Emitters

- Defining an EventEmitter object

```
const EventEmitter = require('events');
const eventEmitter = new EventEmitter();
```

- EventEmitter methods
  - emit is used to trigger an event
  - addListener / on is used to add a callback function that's going to be executed when the event is triggered
  - once() adds a one-time listener
  - removeListener() / off() removes an event listener from an event
  - removeAllListeners()removes all listeners for an event

A very useful article that develops the EventEmitter class step by set can be found here.

EventEmitter Examples

1. Create an EventEmitter object that defines two events “odd” and “even” and two event listeners eventHandle1 that fires for the “odd” event and eventHandler2 that fires for the “even” event. Once a value is read on the command line the name of the event is decided upon then the corresponding eventhandler is fired.

2. Develop a simple nodejs command line program that takes in the name of a file, some text to be added to the file, and a choice to whether to append the text to the file or overwrite what was in the file.

EventEmitter Example 1

```
//add dependency
const event = require('events');
//declare a new event object that will trigger two events
const myevent1 = new event.EventEmitter();
//define the event handlers
const eventHandler1 = ()=>{console.log('this is an odd number');}
const eventHandler2 = ()=>{console.log('This is an even number');}
//The two events are called "odd" and "even"
myevent1.on("odd",eventHandler1); //listener for "odd"
myevent1.on ("even",eventHandler2);//listener for "even"
//Check the number written on the command line
let ename = Number(process.argv[2])%2 === 0 ? "even" : "odd";
//fire the correct event handler according to the number received
myevent1.emit(ename);
```

EventEmitter Example 2

```
const event = require('events');
const fs = require('fs');
const path = require('path');
if (process.argv.length < 5){
    console.log(`Usage: node ${path.basename(process.argv[1])} [file name] [text]`);
}else{
  const myevent1 = new event.EventEmitter();
  const eventHandler1 = (fname,data)=>{
     fs.writeFile(fname,data,(err)=>{
        if (err) throw err;
       console.log("data has been written to the file");
     });}
  const eventHandler2 = (fname,data)=>{
      fs.appendFile(fname,data,(err)=>{
         if (err) throw err;
         console.log("data has been appended to the file");
      });}
  myevent1.on("write",eventHandler1); //listener for "write"
  myevent1.on ("append",eventHandler2);//listener for "append"
  //5. fire the correct event handler according to the choice received
  myevent1.emit(process.argv[2],process.argv[3],process.argv[4]);
}
```

STREAMS

Streams Basics

- Streams are one of the fundamental concepts in computer science
- Streams are a sequence of data bytes that flow from a source to a destination.
  - Example: live audio/video streaming
- Streams are a way to handle reading/writing files, network communications, or any kind of end-to-end information exchange in an efficient way

Streams in Nodejs

- In Node.js a stream is an abstract interface for working with streaming data.
- Streams are used by several modules in Nodejs.
  - In the http module to implement requests and responses
  - In the fs module to facilitate reading from and writing to files.
- All streams are instances of EventEmitter.
- Streams can be readable, writable, or both.

Why Use Streams

- Streams basically provide two major advantages compared to other data handling methods:
  1. **Memory efficiency**: you don't need to load large amounts of data in memory before you are able to process it
  2. **Time efficiency**: it takes significantly less time to start processing data as soon as you have it, rather than having to wait with processing until the entire payload has been transmitted

https://nodesource.com/blog/understanding-streams-in-nodejs/

Types of Stream

- There are 4 types of streams in Node.js:

1. **Writable**: streams to which we can write data. For example, `fs.createWriteStream()` lets us write data to a file using streams.
2. **Readable**: streams from which data can be read. For example: `fs.createReadStream()` lets us read the contents of a file.
3. **Duplex**: streams that are both Readable and Writable. For example, `net.Socket`
4. **Transform**: streams that can modify or transform the data as it is written and read. For example, in the instance of file-compression, you can write compressed data and read decompressed data to and from a file.

STREAM EXAMPLE 1

- Copying files using streams

```
const fs = require('fs');
let src = './Source.txt';
let dest = './Dest.txt'

console.time('streamCp'); //benchmarking
const rStream = fs.createReadStream(src);
const wStream = fs.createWriteStream(dest);

rStream.on('data',function(chunk){//data is a defined event in the stream class
        wStream.write(chunk);
});

rStream.on('end',function(){//end is a defined event in the stream class
        console.log('finished copying the file');
        wStream.end();
});
console.timeEnd('streamCp');//benchmarking
```

STREAM EXAMPLE 2

- Copying a big file (4 GB) using Asynchronous callback API vs using Streams

```
const fs = require('fs');
console.time('AsyncCopy');
let src = './BigFile.txt';
let dest = './BigFileCopy.txt'
fs.readFile(src,(err, data)=>{
 if (err){
  console.log("ERROR: " + err.code + " (" + err.message + ")");}
 fs.writeFile(dest, data,{encoding:"utf-8"},function(err){
   if (err){
     console.log("ERROR: " + err.code + " (" + err.message + ")");}
   else{
     console.log("The File is copied");}
  });
});
console.timeEnd('AsyncCopy');
```

```
const fs = require('fs');
let src = './BigFile.txt';
let dest = './BigFileCopy.txt'

console.time('streamCp');
const rStream = fs.createReadStream(src);
const wStream = fs.createWriteStream(dest);

rStream.on('data',function(chunk){
      wStream.write(chunk);
});

rStream.on('end',function(){
      console.log('finished copying the file');
      wStream.end();
});

console.timeEnd('streamCp');
```

STREAM EXAMPLE 2

- Copying a big file (4 GB) using Asynchronous callback API vs using Streams

```
const fs = require('fs');
console.time('AsyncCopy');
let src = './BigFile.txt';
let dest = './BigFileCopy.txt'
fs.readFile(src,(err, data)=>{
 if (err){
  console.log("ERROR: " + err.code + " (" + err.message + ")");}
 fs.writeFile(dest, data,{encoding:"utf-8"},function(err){
   if (err){
     console.log("ERROR: " + err.code + " (" + err.message + ")");}
   else{
     console.log("The File is copied");}
  });
});
console.timeEnd('AsyncCopy');
```

```
const fs = require('fs');
let src = './BigFile.txt';
let dest = './BigFileCopy.txt'

console.time('streamCp');
const rStream = fs.createReadStream(src);
const wStream = fs.createWriteStream(dest);

rStream.on('data',function(chunk){
       wStream.write(chunk);
});

rStream.on('end',function(){
        console.log('finished copying the file');
        wStream.end();
});

console.timeEnd('streamCp');
```

```
node streamCp.js
streamCp: 0.675ms
finished copying the file
```

STREAM EXAMPLE 2

- Copying a big file (4 GB) using Asynchronous callback API vs using Streams

```
const fs = require('fs');
console.time('AsyncCopy');
let src = './BigFile.txt';
let dest = './BigFileCopy.txt'
fs.readFile(src,(err, data)=>{
 if (err){
  console.log("ERROR: " + err.code + " (" + err.message + ")");}
 fs.writeFile(dest, data,{encoding:"utf-8"},function(err){
   if (err){
     console.log("ERROR: " + err.code + " (" + err.message + ")");}
   else{
     con
  });
});
console
```

```
node AsyncCopy.js
AsyncCopy: 0.126ms
ERROR: ERR_FS_FILE_TOO_LARGE (File size (4393000000) is greater than 2 GiB)
node:internal/fs/utils:901
```

```
const fs = require('fs');
let src = './BigFile.txt';
let dest = './BigFileCopy.txt'

console.time('streamCp');
const rStream = fs.createReadStream(src);
const wStream = fs.createWriteStream(dest);

rStream.on('data',function(chunk){
      wStream.write(chunk);
});

rStream.on('end',function(){
       console.log('finished copying the file');
       wStream.end();
});
```

```
node streamCp.js
streamCp: 0.675ms
finished copying the file
```

PIPES

Pipes Basics

- In system programming, a pipe is a technique for passing information from one program process or command to another.
  - Uses a temporary software connection between two programs or commands.
  - An area of the main memory is treated like a virtual file to temporarily hold data and pass it from one process to another in a single direction
- Piping was introduced in the Unix operating system, and it can be used on the command line.

Pipe Example

- In Linux, the `wc` is a command the counts the number of lines, words, and characters in a file.

```
%wc Source.txt
    6    93    538 Source.txt
```

- Using pipes, the output of wc can be directed into a text file

```
% wc Source.txt > count.txt
% cat count.txt
    6    93    538 Source.txt
```

Pipes in Nodejs

- In Nodejs, streams can be piped together using the pipe() method defined on readable streams.
- **Piping takes two arguments:**
  - A Required writable stream that acts as the destination for the data and
  - An optional object used to pass in options.
- File transfer can be done using piping

Pipe Example

```
const fs = require('fs');
let src = './Source.txt'
let dest = './DestPipe.txt'
console.time('pipeCopy');
const rdStream = fs.createReadStream(src);
const wtStream = fs.createWriteStream(dest);
rdStream.pipe(wtStream);
console.timeEnd('pipeCopy');
```

PIPE VS STREAMS BENCHMARKING

```
const fs = require('fs');
let src = './Source.txt'
let dest = './DestPipe.txt'
console.time('pipeCopy');
const rdStream = fs.createReadStream(src);
const wtStream = fs.createWriteStream(dest);
rdStream.pipe(wtStream);
console.timeEnd('pipeCopy’);
```

```
% node pipeCopy.js
pipeCopy: 4.709ms
```

```
const fs = require('fs');
let src = './Source.txt';
let dest = './Dest.txt'
console.time('streamCopy');
const rStream = fs.createReadStream(src);
const wStream = fs.createWriteStream(dest);
rStream.on('data',function(chunk){
  wStream.write(chunk);
});
rStream.on('end',function(){
  console.log('finished copying the file’);
  wStream.end();
});
console.timeEnd('streamCopy');
```

PIPE VS STREAMS BENCHMARKING

```
const fs = require('fs');
let src = './Source.txt'
let dest = './DestPipe.txt'
console.time('pipeCopy');
const rdStream = fs.createReadStream(src);
const wtStream = fs.createWriteStream(dest);
rdStream.pipe(wtStream);
console.timeEnd('pipeCopy’);
```

```
% node pipeCopy.js
pipeCopy: 4.709ms
```

```
const fs = require('fs');
let src = './Source.txt';
let dest = './Dest.txt'
console.time('streamCopy');
const rStream = fs.createReadStream(src);
const wStream = fs.createWriteStream(dest);
rStream.on('data',function(chunk){
  wStream.write(chunk);
});
rStream.on('end',function(){
  console.log('finished copying the file’);
  wStream.end();
});
console.timeEnd('streamCopy');
```

```
%node streamCopy.js
streamCopy: 0.98ms
finished copying the file
```

PIPE VS STREAMS BENCHMARKING

```
const fs = require('fs');
let src = './Source.txt'
let dest = './DestPipe.txt'
console.time('pipeCopy');
const rdStream = fs.createReadStream(src);
const wtStream = fs.createWriteStream(dest);
rdStream.pipe(wtStream);
console.timeEnd('pipeCopy’);
```

```
% node pipeCopy.js
pipeCopy: 4.709ms
```

```
const fs = require('fs');
let src = './Source.txt';
let dest = './Dest.txt'
console.time('streamCopy');
const rStream = fs.createReadStream(src);
const wStream = fs.createWriteStream(dest);
rStream.on('data',function(chunk){
  wStream.write(chunk);
});
rStream.on('end',function(){
  console.log('finished copying the file’);
  wStream.end();
});
console.timeEnd('streamCopy');
```

```
%node streamCopy.js
streamCopy: 0.98ms
finished copying the file
```

Using pipes takes more time than when using streams, yet with pipes the memory pressure is less

SUMMARY

- Nodejs supports concurrency via the concept of event and callbacks.
- Callbacks define logic for one-off responses to events.
- Callback functions always has at least one parameter that indicates the success or failure status of the last operation.
- The filesystem provides asynchronous APIs using callbacks.
- Nesting of callbacks can lead to an unreadable and not easy to manageable codebase commonly known as callback hell.
- Nodejs implements event driven programming through the use of the EventEmitter class
- The EventEmitter class implements an Observer pattern that defines an event and and equivalent listener which gets triggered when the event happens
- Streams are instances of the EventEmitter and are used to handle any kind of end-to-end information exchange.
- Streams can be piped together to ease the transfer of information end-to-end.