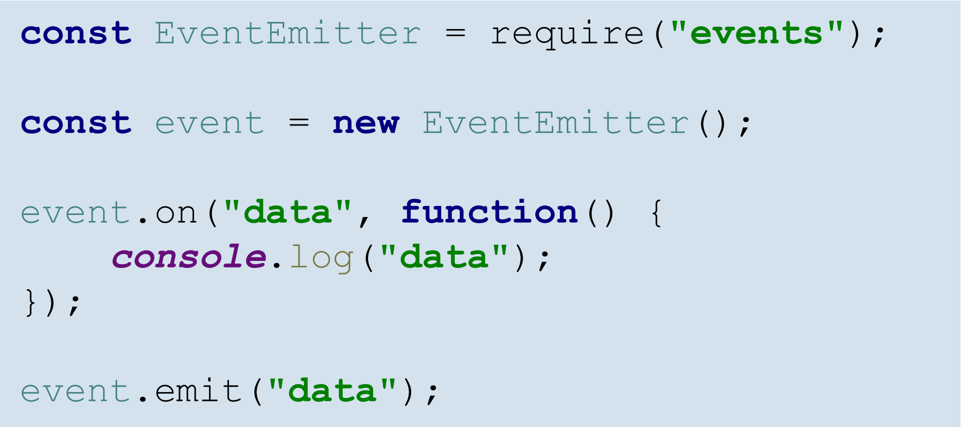
CORE MODULES

Agenda

- Review and use Node.js core modules
- Event Emitter
- File System API
- Stream API
- Buffer and strings
- Cluster API

EventEmitter

- The core class of Node.js asynchronous event-driven architecture
- Allows for publisher/subscriber implementation
- Events are emitted using the emit method
- Listeners can subscribe using the on method



```
const EventEmitter = require("events");

const event = new EventEmitter();

event.on("data", function() {
    console.log("data");
});

event.emit("data");
```

emit is synchronous

- All listeners are notified synchronously in the same order of registration

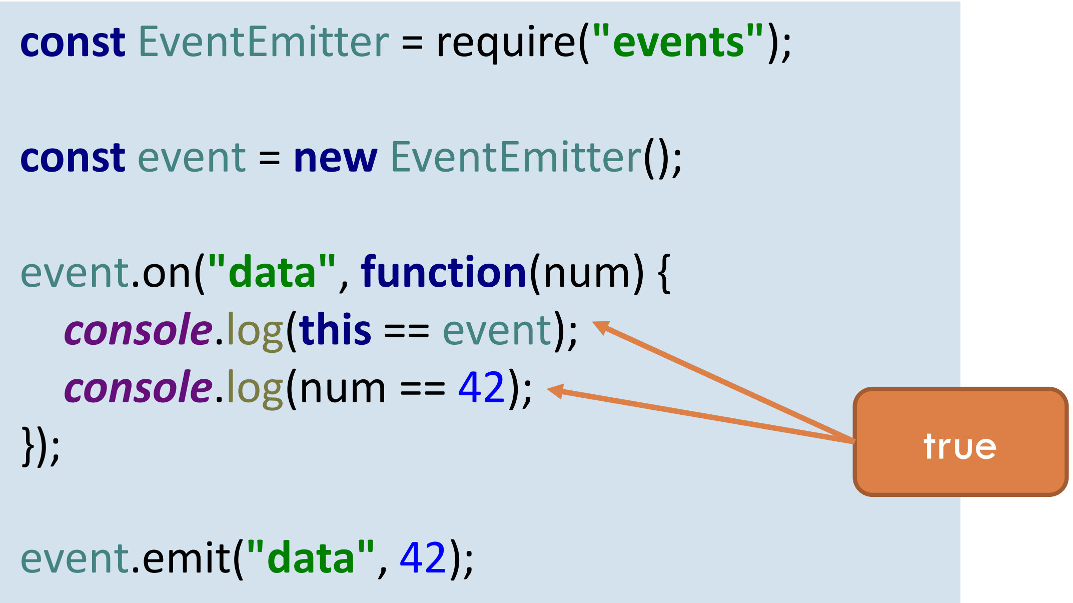
```
const EventEmitter = require("events");

const event = new EventEmitter();

event.on("data", function() {
   console.log("listener1");
});

event.on("data", function() {
   console.log("listener2");
});

console.log("before");
event.emit("data");
console.log("after");
```

Output is:

```
before
listener1
listener2
after
```

Passing Arguments

- □ emit allows arbitrary set of arguments to be passed
- □ Inside the callback, this references the EventEmitter instance



```
const EventEmitter = require("events");

const event = new EventEmitter();

event.on("data", function(num) {
    console.log(this == event);
    console.log(num == 42);
});

event.emit("data", 42);
```

Error Event

- Node.js treats the error event in a special way
- If no listener is registered for the event
  - Stack trace is printed
  - Node.js kills the process

```
const EventEmitter = require("events");

const event = new EventEmitter();

event.emit("error", new Error("Oooops"));

console.log("after");
```

This line will not be executed

Error Event

- Emitting error event without having a listener causes Node.js to throw the error
- You can catch it
  - Not common

```
const EventEmitter = require("events");

const event = new EventEmitter();

try {
    event.emit("error", new Error("Oooops"));
}
catch(err) {
    console.log("after");
}
```

Now, this line will be executed

Removing a Listener

- □ Use the removeListener method
- □ Does not effect the current emit call

```
const EventEmitter = require("events");
const ev = new EventEmitter();

function listener1() {
    console.log("listener1");
    ev.removeListener("data", listener2);
}

function listener2() {
    console.log("listener2");
}

ev.on("data", listener1);
ev.on("data", listener2);

ev.emit("data");
```

Although removed, listener2 will be notified

More

- once
- newListener/removeListener events
- prependListener
- removeAllListeners

ArrayBuffer

- Fixed size
- Raw binary data
- You cannot read/manipulate its content
  - Need to create a typed array view

```
const buf = new ArrayBuffer(16);

console.log(buf.byteLength);
```

TypedArray

- □ An array-like view of an underline ArrayBuffer
- □ No global property with that name
- □ Represents a group of “view” classes

```
const buf = new ArrayBuffer(16);

const view8 = new Uint8Array(buf);
view8[0] = 1;
view8[1] = 1;

const view16 = new Uint16Array(buf);
console.log(view16[0]);
```

Prints 257
Why ?

Buffer

- TypedArray & ArrayBuffer are part of ES6
- Before ES6, Node.js had to offer its own implementation of binary data → Buffer API
- Can think of it as an Uint8Array
- Buffer is
  - Fixed size
  - Raw memory
  - Outside of V8 heap
  - More optimized than Uint8Array

Create Buffer

- □ Do not use constructor
- □ Use static methods from, alloc & allocUnsafe

```
const buf = Buffer.alloc(10);

for(let i=0; i<buf.length; i++) {
    buf[i] = i;
}

for(const byte of buf) {
    console.log(byte);
}
```

Be aware of Truncation

- Each index is of 1 byte size
- Writing data larger than 1 byte → data loss

```
const buf = Buffer.alloc(10);

buf[0] = 1000; //0x000003e8

console.log(buf[0]); //0xe8
```

Only the least significant byte is preserved

Buffer & String

- □ Can be easily transformed from one to the other

```
const buf = Buffer.from("abc");
const str = buf.toString();
console.log(str == "abc");
```

- □ Default encoding is utf8

Base64

- Is considered an encoding

```
const buf = Buffer.from("abc");

const str = buf.toString("base64");

const clone = Buffer.from(str, "base64");

console.log(Buffer.compare(buf, clone));
```

str equals YWJj

Not the same reference

Buffer as View

- □ In some cases a buffer instance is just a view over the raw data

```
const buf = Buffer.from("abcde");

const slice = buf.slice(0, 1);

console.log(slice.buffer == buf.buffer);
```

buf & slice share the same internal buffer

Crazy stuff

- What will be printed ?

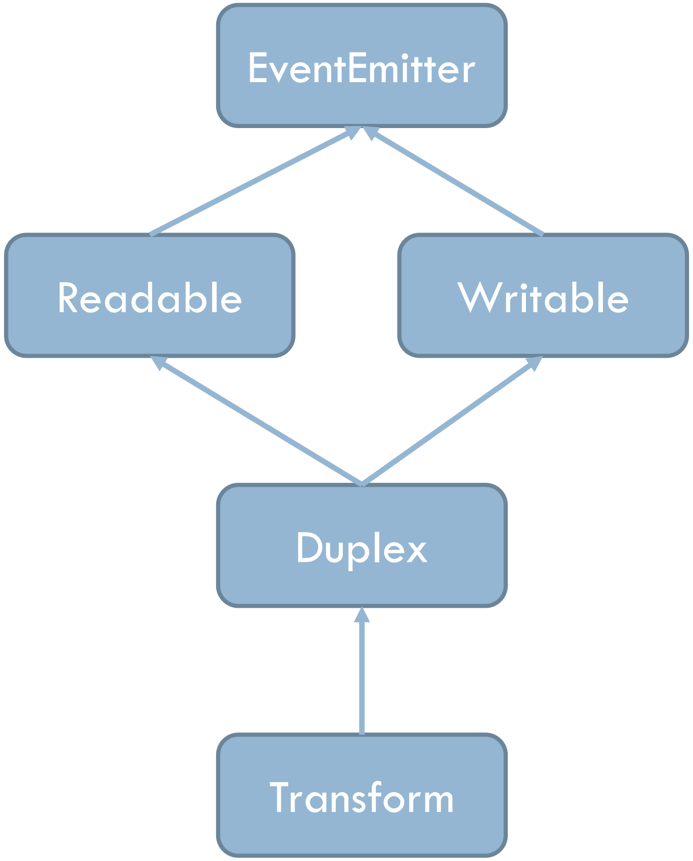
```
const buf1 = Buffer.from("abcdef");

const buf2 = Buffer.from(buf1.buffer, 0, 10);

console.log(buf2.toString());
```

More

- Buffer.compare
- Buffer.concat
- fill
- includes
- indexOf
- readXXX/writeXXX
- swap16/32/64

Type of Streams

- □ Readable
- □ Writable
- □ Duplex
- □ Transform



Consuming Readable Stream

- A.K.A “pull”
- Wait for the readable event
- Pull the buffered data using read

```
const stream = fs.createReadStream("main.js");

stream.on("readable", function() {
    const buf = stream.read();

    console.log(buf);
});
```

read(size)

- You can limit the size of the returned buffer
- Must invoke read multiple times until null is returned

```
const stream = fs.createReadStream("main.js");

stream.on("readable", function() {
    let chunk;

    while(chunk = stream.read(1)) {
        process.stdout.write(chunk.toString());
    }
});
```

Flowing Mode

- □ Readable stream begins at paused state
- □ Registering to data event causes stream to switch to flowing mode

Buffer object

```
const fs = require("fs");

const stream = fs.createReadStream("main.js");

setTimeout(function() {
    //
    //  Data is not lost because of this delay
    //
    stream.on("data", function(buf) {
        console.log(buf);
    });
}, 1500);
```

end Event

- □ Only relevant for readable streams
- □ Signals the end of read operation
- □ on returns the source stream → Use chaining

```
const stream = fs.createReadStream("main.js");

stream
    .on("data", function(buf) {
        console.log("data", buf);
    })
    .on("end", function() {
        console.log("end");
    });
```

error Event

- As for any EventEmitter, you must handle the error event. Else, Node.js kills your process

```
var net = require('net');

var server = net.createServer(function(socket) {
    console.log("New connection");

    socket.pipe(socket).on("error", function(err) {
        console.error(err);
    });
});

server.listen(1337, '127.0.0.1');
```

TCP Client

- Same paradigm

```
const net = require('net');

const client = new net.Socket();

client
    .connect(1337, '127.0.0.1', function () {
        console.log('Connected');
        client.write('Hello, server');
    })
    .on('data', function (data) {
        console.log('Received: ' + data);
        client.destroy();
    })
    .on('close', function () {
        console.log('Connection closed');
    });
```

Pipe

- Instead of handling the data event directly you can pipe into a writable stream

```
const fs = require("fs");

const stream = fs.createReadStream("main.js");

stream.pipe(fs.createWriteStream("main.js.backup"));
```

- The readable stream automatically switches to flowing mode

Pipe Notes

- The flow of data is controller by the pipe
  - For example, backpressure
- Can attach multiple write streams
- Automatically ends the write stream when the readable emits end
  - Can disable it using the option

```
reader.pipe(writer, {
  end: false
});
```

  - In case of an error the write stream is not closed

Chain of Pipes

- pipe method returns a reference to the destination stream
- Therefore, we can chain multiple pipes

```
const fs = require("fs");
const zlib = require("zlib");

fs.createReadStream("main.js")
    .pipe(zlib.createGzip())
    .pipe(fs.createWriteStream("main.js.gz"));
```

finish Event

- The finish event can be used to determine the end of the writing operation

```
async function main() {
    await zip("main.js", "main.js.gz");
    await rename("temp/main.js.gz", "done/main.js.gz");
}

function zip(dest, source) {
    return new Promise((resolve,reject)=> {
        fs.createReadStream("main.js")
            .pipe(zlib.createGzip())
            .pipe(fs.createWriteStream("temp/main.js.gz"))
            .on("finish", function () {
                resolve();
            });
    });
}
```

Pipe & Errors

- □ Errors are not propagated through the pipe chain
- □ Instead, the destination stream is unpiped

```
const stream = new MyReadable();

stream.pipe(fs.createWriteStream("1.txt")).on("finish", function() {
    console.log("finish");
});
```

- □ finish event will never happen

Pipe & Errors

□ Must handle error event after each pipe

```
function compress(source, dest) {
   return new Promise((resolve, reject) => {
      fs.createReadStream(source)
         .on("error", function (err) {
            reject(err);
         })
         .pipe(zlib.createGzip())
         .on("error", function (err) {
            reject(err);
         })
         .pipe(fs.createWriteStream(dest))
         .on("error", function (err) {
            reject(err);
         })
         .on("finish", function () {
            resolve();
         });
   });
}
```

Stream of what ?

- □ Buffer | string | Uint8Array
- □ However, the abstraction model is flexible enough to represent non bytes stream
- □ AKA "Object Mode"

```
const gulp = require("gulp");

gulp.src("*.js*").on("data", function(chunk) {
    console.log(chunk.path);
});
```

chunk is an object not a buffer/string

Summary

□ Streams are quite easy to use

□ Harder to implement

□ Binary data is represented using a Buffer object