Node.js Streams & Buffers

Mohamed Mukthar Ahmed

1

Streams & Buffers

Outline

- What are Streams?
- The stream Module
- Why streams?
- Stream-powered APIs
- Different Types of Streams
- Creating a readable stream
- Creating a writeable stream
- Piping
- What is a Buffer?
- Why do we need a buffer?
- Accessing buffer content
- Changing content of a buffer
- Buffer Slicing
- Copy a buffer

Mohamed Mukthar Ahmed

2

What are streams?

- **Streams** are one of the **fundamental concepts** that **power** **Node.js** applications.
- They are a **way** to handle
  - reading/writing files,
  - network communications, or
  - any kind of end-to-end information exchange
- in an efficient way.
- Streams are not a concept unique to Node.js. They were introduced in the **UNIX** operating system decades ago.
- Programs can interact with each other passing streams through the **pipe operator** (**|**).

Use Case

- In the traditional way, when you tell the program to read a file, the file is read into memory, from start to finish, and then you process it.
- Using streams you **read it piece by piece**, processing its content without keeping it all in memory.
- **Use Case**
  - Let's take a "**streaming**" services such as **YouTube** or **Netflix** for example: these services don't make you download the video and audio feed all at once. Instead, your browser receives the video as a continuous flow of chunks, allowing the recipients to start watching and/or listening almost immediately.
- However, streams are not only about working with media or big data. They also give us the power of '**composability**' in our code.

Node.js stream Module

- The Node.js **stream** module provides the foundation upon which all **streaming APIs** are built.
- All **streams** are instances of **EventEmitter**

Why Streams?

- Streams basically provide **two major advantages** over using other data handling methods:
- **Memory efficiency**
  - You don't need to load large amounts of data in memory before you are able to process it
- **Time efficiency**
  - It takes way less time to start processing data, since you can start processing as soon as you have it, rather than waiting till the whole data payload is available

Stream-powered Node.js APIs

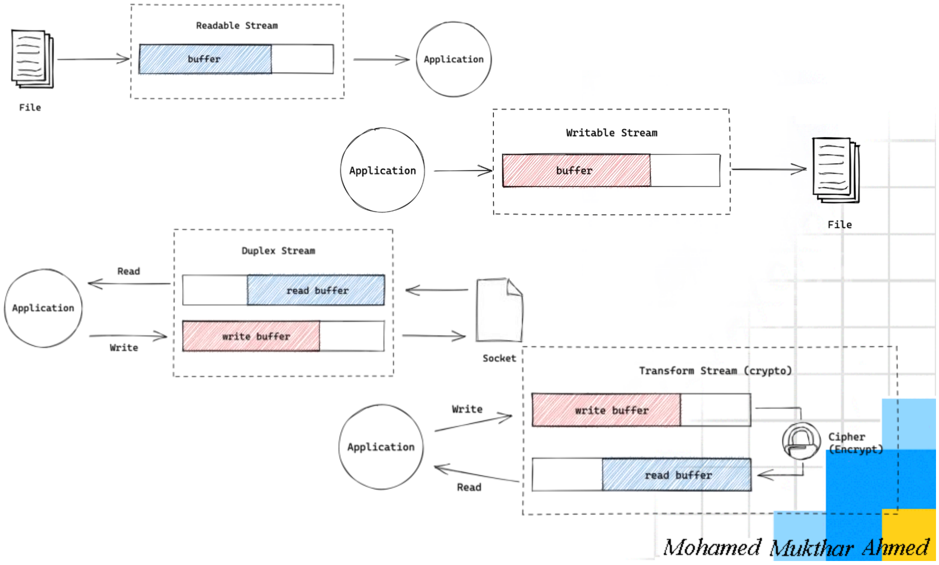
- Due to their **advantages**, many Node.js core modules provide native stream handling capabilities, most notably:
- **process.stdin** returns a stream connected to **stdin**
- **process.stdout** returns a stream connected to **stdout**
- **process.stderr** returns a stream connected to **stderr**
- **fs.createReadStream()** creates a **readable stream** to a file
- **fs.createWriteStream()** creates a **writable stream** to a file

Different Types of Streams

- There are **FOUR** classes of streams:
- **Readable**: a stream which could be used for read data from it. In other words, its **readonly**.
- **Writable**: a stream which could be used for write data to it. It is **writeonly**.
- **Duplex**: a stream which can read and write data, basically its a combination of a Readable and Writable stream.
- **Transform**: a Duplex stream which reads data, transforms the data, and then writes the transformed data in the desired format.

Different Types of Streams

- There are **FOUR** classes of streams:



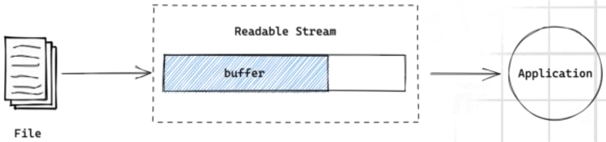
Creating a readable stream

- We first require the Readable stream, and we initialize it.

```
const Stream = require('stream')

const readableStream = new Stream.Readable()
```

- Now that the stream is initialized, we can send data to it:

```
readableStream.push('ping!')

readableStream.push('pong!')
```



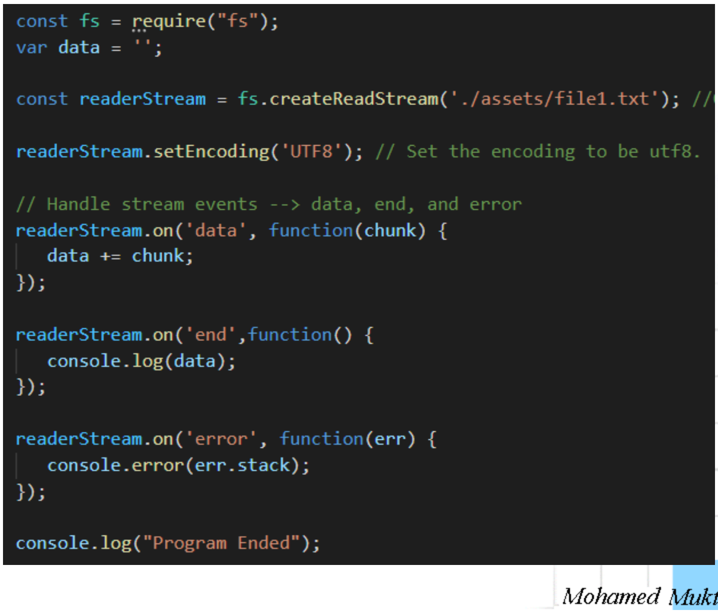
Two Reading Modes

- According to Streams API, readable streams effectively operate in one of two modes: **flowing** and **paused**.

- In **flowing mode**, data is read from the underlying system automatically and provided to an application as quickly as possible using events via the **EventEmitter** interface.

- In **paused mode**, the **stream.read()** method must be called explicitly to read chunks of data from the stream.

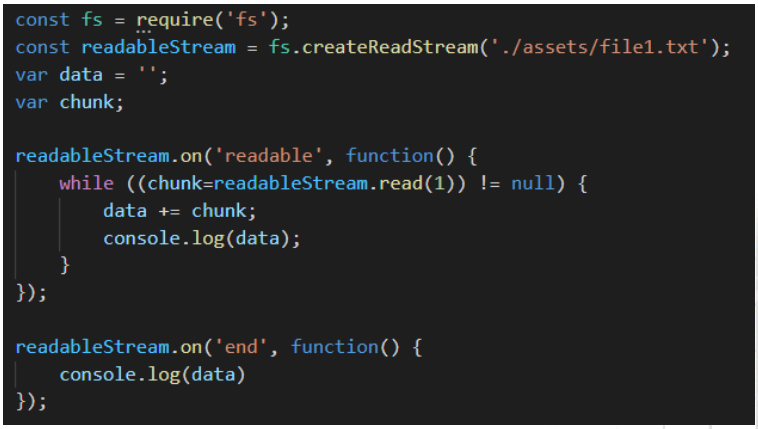
Code Example



```
const fs = require("fs");
var data = '';

const readerStream = fs.createReadStream('./assets/file1.txt'); //

readerStream.setEncoding('UTF8'); // Set the encoding to be utf8.

// Handle stream events --> data, end, and error
readerStream.on('data', function(chunk) {
   data += chunk;
});

readerStream.on('end',function() {
   console.log(data);
});

readerStream.on('error', function(err) {
   console.error(err.stack);
});

console.log("Program Ended");
```

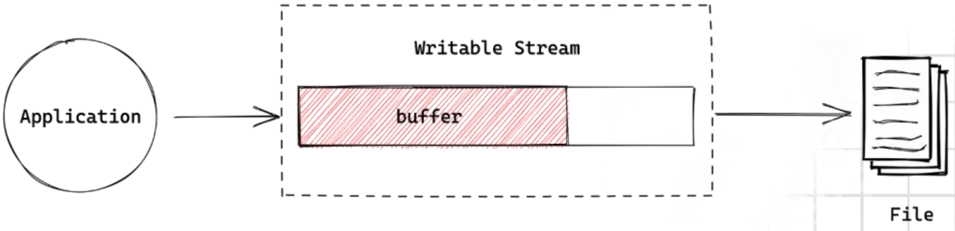
Mohamed Mukthar Ahmed

Code Example



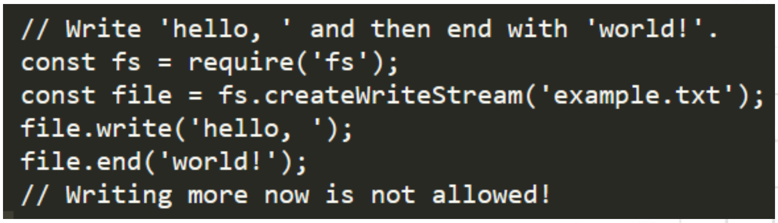
```
const fs = require('fs');
const readableStream = fs.createReadStream('./assets/file1.txt');
var data = '';
var chunk;

readableStream.on('readable', function() {
    while ((chunk=readableStream.read(1)) != null) {
        data += chunk;
        console.log(data);
    }
});

readableStream.on('end', function() {
    console.log(data)
});
```

Mohamed Mukthar Ahmed

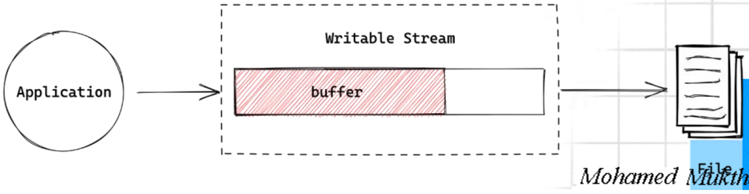


Creating a writeable stream

- To write data to a **writable stream** you need to call **write()** on the stream instance.
- Calling the **writable.end()** method signals that no more data will be written to the Writable.



```
// Write 'hello, ' and then end with 'world!'.
const fs = require('fs');
const file = fs.createWriteStream('example.txt');
file.write('hello, ');
file.end('world!');
// Writing more now is not allowed!
```

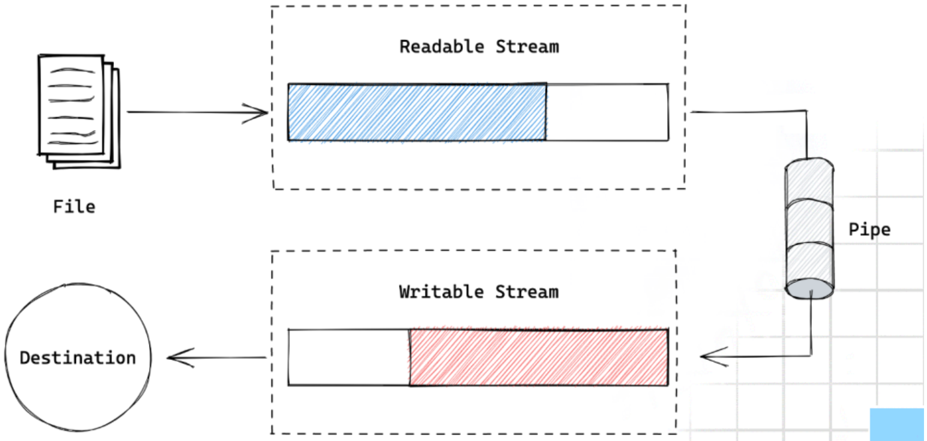


Piping

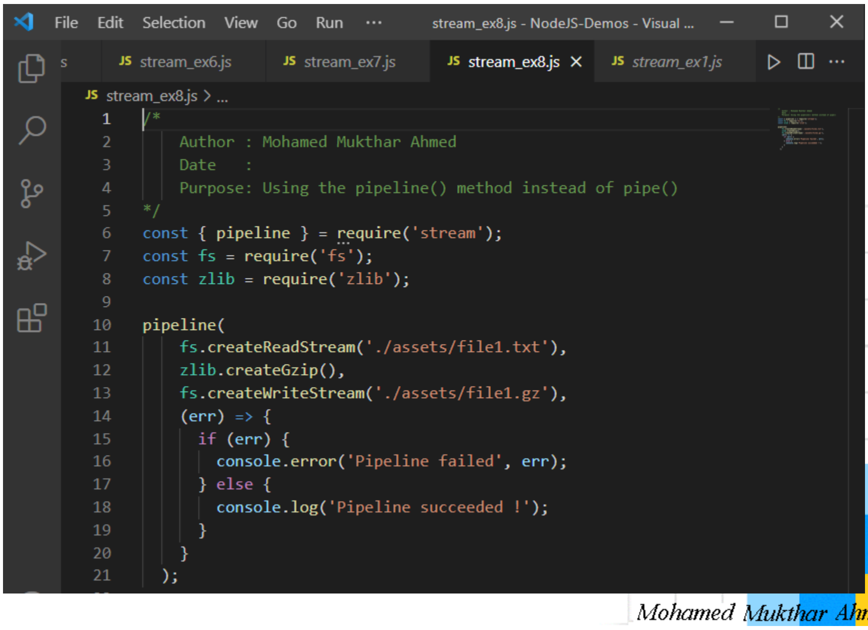
- **Piping** is a mechanism where we provide the output of one stream as the input to another stream.
- It is normally used to get data from one stream and to pass the output of that stream to another stream.
- There is no limit on piping operations. In other words, piping is used to process streamed data in multiple steps.
- In **Node 10.x** was introduced **stream.pipeline()**. This is a module method to pipe between streams.

The ***pipeline*** *should be used instead of* ***pipe****, as pipe is unsafe.*

Piping



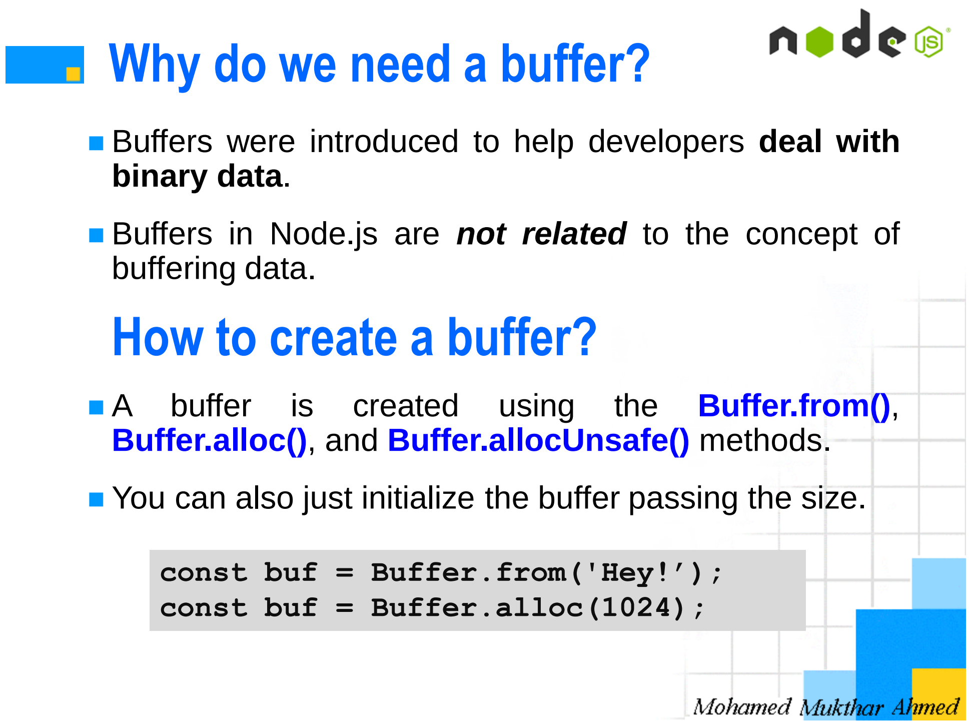
Piping



```
/*
    Author : Mohamed Mukthar Ahmed
    Date    :
    Purpose: Using the pipeline() method instead of pipe()
*/
const { pipeline } = require('stream');
const fs = require('fs');
const zlib = require('zlib');

pipeline(
    fs.createReadStream('./assets/file1.txt'),
    zlib.createGzip(),
    fs.createWriteStream('./assets/file1.gz'),
    (err) => {
      if (err) {
        console.error('Pipeline failed', err);
      } else {
        console.log('Pipeline succeeded !');
      }
    }
  );
```

What is a Buffer?

- A **buffer** is an **area of memory**.
- Most JavaScript developers are much less familiar with this concept.
- It represents a fixed-size chunk of memory (can't be resized) allocated outside of the V8 JavaScript engine.
- You can think of a buffer like an array of integers, each representing a byte of data.
- It is implemented by the Node.js **Buffer** class.



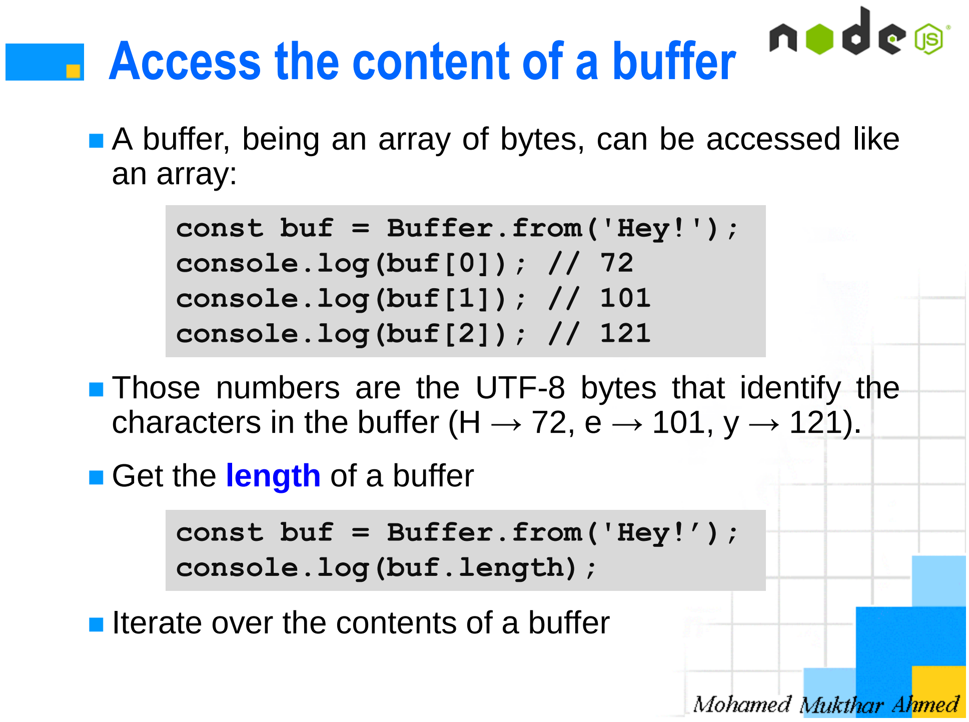
Why do we need a buffer?

- Buffers were introduced to help developers **deal with binary data**.
- Buffers in Node.js are ***not related*** to the concept of buffering data.

How to create a buffer?

- A buffer is created using the **Buffer.from()**, **Buffer.alloc()**, and **Buffer.allocUnsafe()** methods.
- You can also just initialize the buffer passing the size.

```
const buf = Buffer.from('Hey!’);
const buf = Buffer.alloc(1024);
```



Access the content of a buffer

- A buffer, being an array of bytes, can be accessed like an array:

```
const buf = Buffer.from('Hey!');
console.log(buf[0]); // 72
console.log(buf[1]); // 101
console.log(buf[2]); // 121
```

- Those numbers are the UTF-8 bytes that identify the characters in the buffer (H → 72, e → 101, y → 121).
- Get the **length** of a buffer

```
const buf = Buffer.from('Hey!’);
console.log(buf.length);
```

- Iterate over the contents of a buffer

Access the content of a buffer

- Iterate over the contents of a buffer

```
const buf = Buffer.from('Hey!');
for (const item of buf) {
  console.log(item); // 72 101 121 33
}
```

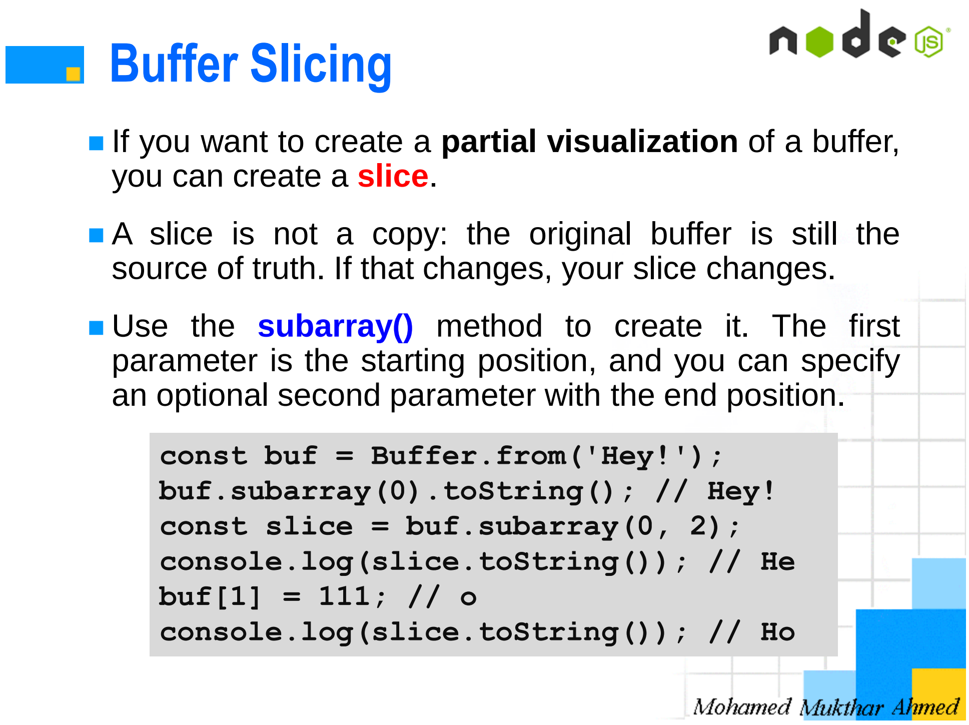
- Changing the content of a buffer
- You can write to a buffer a whole string of data by using the **write()** method:

```
const buf = Buffer.alloc(4);
buf.write('Hey!');
```

Changing the content of a buffer

- Just like you can access a buffer with an array syntax, you can also set the contents of the buffer in the same way:

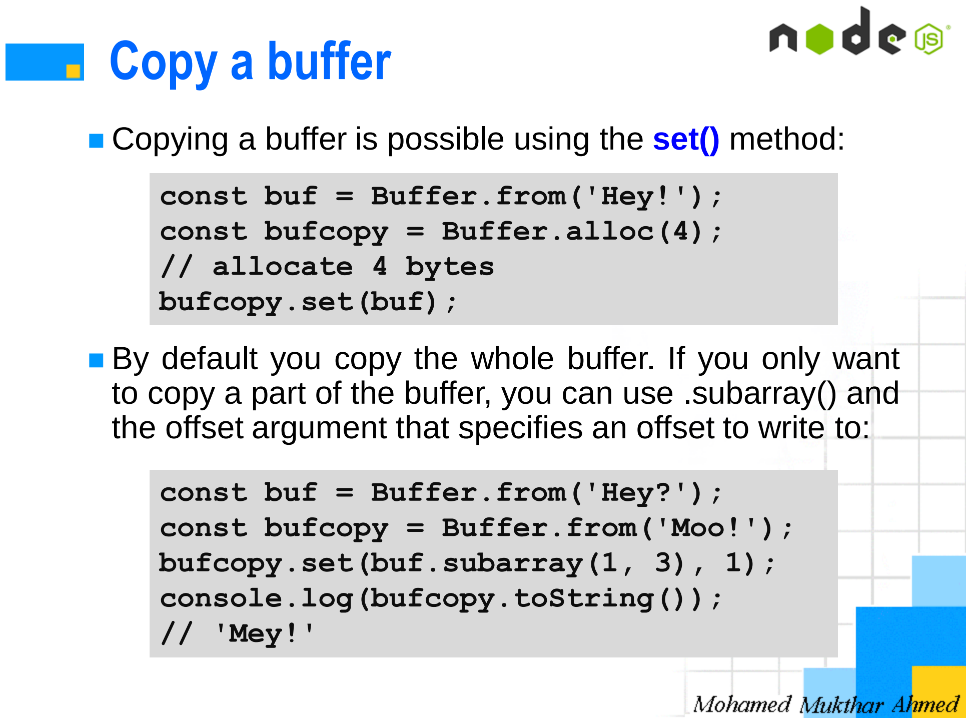
```
const buf = Buffer.from('Hey!');
buf[1] = 111; // o in UTF-8
console.log(buf.toString()); // Hoy!
```



Buffer Slicing

- If you want to create a **partial visualization** of a buffer, you can create a **slice**.
- A slice is not a copy: the original buffer is still the source of truth. If that changes, your slice changes.
- Use the **subarray()** method to create it. The first parameter is the starting position, and you can specify an optional second parameter with the end position.

```
const buf = Buffer.from('Hey!');
buf.subarray(0).toString(); // Hey!
const slice = buf.subarray(0, 2);
console.log(slice.toString()); // He
buf[1] = 111; // o
console.log(slice.toString()); // Ho
```



Copy a buffer

- Copying a buffer is possible using the **set()** method:

```
const buf = Buffer.from('Hey!');
const bufcopy = Buffer.alloc(4);
// allocate 4 bytes
bufcopy.set(buf);
```

- By default you copy the whole buffer. If you only want to copy a part of the buffer, you can use .subarray() and the offset argument that specifies an offset to write to:

```
const buf = Buffer.from('Hey?');
const bufcopy = Buffer.from('Moo!');
bufcopy.set(buf.subarray(1, 3), 1);
console.log(bufcopy.toString());
// 'Mey!'
```