Lesson 6: Buffers and Streams

Buffers:

Buffer objects are used to represent a fixed-length sequence of bytes. Many Node.js APIs support Buffers.

The Buffer class is a subclass of JavaScript's Uint8Array class and extends it with methods that cover additional use cases. Node.js APIs accept plain Uint8Arrays wherever Buffers are supported as well.

The Buffer class is within the global scope, making it unlikely that one would need to ever use require('buffer').Buffer.

When converting between Buffers and strings, a character encoding may be specified. If no character encoding is specified, UTF-8 will be used as the default.

Class: Buffer

Buffer.alloc(size[, fill[, encoding]])

- size <integer> The desired length of the new Buffer.
- fill <string> | <Buffer> | <Uint8Array> | <integer> A value to pre-fill the new Buffer with. **Default:** 0.
- encoding <string> If fill is a string, this is its encoding. **Default:** 'utf8'.

Allocates a new Buffer of size bytes. If fill is undefined, the Buffer will be zero-filled.

```
const buf = Buffer.alloc(5);

console.log(buf);
// Prints: <Buffer 00 00 00 00 00>
```

Calling Buffer.alloc() can be measurably slower than the alternative Buffer.allocUnsafe() but ensures that the newly created Buffer instance contents will never contain sensitive data from previous allocations, including data that might not have been allocated for Buffers.

Buffer.allocUnsafe(size)

- size <integer> The desired length of the new Buffer.

Allocates a new Buffer of size bytes. If size is larger than buffer.constants.MAX_LENGTH or smaller than 0, ERR_INVALID_OPT_VALUE is thrown.

The underlying memory for Buffer instances created in this way is *not initialized*. The contents of the newly created Buffer are unknown and *may contain sensitive data*. Use Buffer.alloc() instead to initialize Buffer instances with zeroes.

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

console.log(buf);
// Prints (contents may vary): <Buffer a0 8b 28 3f 01 00 00 00 50 32>
```

```
buf.fill(0);

console.log(buf);
// Prints: <Buffer 00 00 00 00 00 00 00 00 00 00>
```

Buffer.compare(buf1, buf2)

- buf1 <Buffer> | <Uint8Array>
- buf2 <Buffer> | <Uint8Array>
- Returns: <integer> Either -1, 0, or 1, depending on the result of the comparison. See buf.compare() for details.

Compares buf1 to buf2, typically for the purpose of sorting arrays of Buffer instances. This is

equivalent to calling buf1.compare(buf2).

```
const buf1 = Buffer.from('1234');
const buf2 = Buffer.from('0123');
const arr = [buf1, buf2];

console.log(arr.sort(Buffer.compare));
// Prints: [ <Buffer 30 31 32 33>, <Buffer 31 32 33 34> ]
// (This result is equal to: [buf2, buf1].)
```

Buffer.concat(list[, totalLength])

- list <Buffer[]> | <Uint8Array[]> List of Buffer or Uint8Array instances to concatenate.
- totalLength <integer> Total length of the Buffer instances in list when concatenated.
- Returns: <Buffer>

Returns a new Buffer which is the result of concatenating all the Buffer instances in the list together.

If the list has no items, or if the totalLength is 0, then a new zero-length Buffer is returned.

```
// Create a single `Buffer` from a list of three `Buffer` instances.

const buf1 = Buffer.alloc(10);
const buf2 = Buffer.alloc(14);
const buf3 = Buffer.alloc(18);
const totalLength = buf1.length + buf2.length + buf3.length;

console.log(totalLength);
// Prints: 42

const bufA = Buffer.concat([buf1, buf2, buf3], totalLength);

console.log(bufA);
// Prints: <Buffer 00 00 00 00 ...>
console.log(bufA.length);
// Prints: 42
```

Buffer.from(array)

- array <integer[]>

Allocates a new Buffer using an array of bytes in the range 0 – 255.

Array entries outside that range will be truncated to fit into it.

```
// Creates a new Buffer containing the UTF-8 bytes of the string 'buffer'.
const buf = Buffer.from([0x62, 0x75, 0x66, 0x66, 0x65, 0x72]);
```

buf.compare(target[, targetStart[, targetEnd[, sourceStart[, sourceEnd]]]])

- target <Buffer> | <Uint8Array> A Buffer or Uint8Array with which to compare buf.
- targetStart <integer> The offset within target at which to begin comparison. **Default:** 0.
- targetEnd <integer> The offset within target at which to end comparison (not inclusive). **Default:** target.length.
- sourceStart <integer> The offset within buf at which to begin comparison. **Default:** 0.
- sourceEnd <integer> The offset within buf at which to end comparison (not inclusive). **Default:** buf.length.
- Returns: <integer>

Compares buf with target and returns a number indicating whether buf comes before, after, or is the

same as target in sort order. Comparison is based on the actual sequence of bytes in each Buffer.

- 0 is returned if target is the same as buf
- 1 is returned if target should come before buf when sorted.
- -1 is returned if target should come after buf when sorted.

```
const buf1 = Buffer.from('ABC');
const buf2 = Buffer.from('BCD');
const buf3 = Buffer.from('ABCD');

console.log(buf1.compare(buf1));
// Prints: 0
console.log(buf1.compare(buf2));
// Prints: -1
console.log(buf1.compare(buf3));
// Prints: -1
console.log(buf2.compare(buf1));
// Prints: 1
console.log(buf2.compare(buf3));
// Prints: 1
console.log([buf1, buf2, buf3].sort(Buffer.compare));
// Prints: [ <Buffer 41 42 43>, <Buffer 41 42 43 44>, <Buffer 42 43 44> ]
// (This result is equal to: [buf1, buf3, buf2].)
```

buf.copy(target[, targetStart[, sourceStart[, sourceEnd]]])

- target <Buffer> | <Uint8Array> A Buffer or Uint8Array to copy into.

- targetStart <integer> The offset within target at which to begin writing. **Default:** 0.
- sourceStart <integer> The offset within buf from which to begin copying. **Default:** 0.
- sourceEnd <integer> The offset within buf at which to stop copying (not inclusive). **Default:** buf.length.
- Returns: <integer> The number of bytes copied.

Copies data from a region of buf to a region in target, even if the target memory region overlaps with buf.

```
const buf1 = Buffer.alloc(10, 'hi welcome');
const buf2 = Buffer.alloc(10, 'hi welcome');

var result = buf1.compare(buf2);
if(result < 0)
{
console.log(buf1 + " comes before " +  buf2);
}
else if(result === 0)
{
   console.log(buf1 + " is same as " +  buf2);
}
else
{
   console.log(buf1 + " comes after " +  buf2);
}

var buf3 = Buffer.alloc(25);
buf2.copy(buf3);
console.log("Buffer2 content is copied to Buffer3 "+ buf3.toString());
console.log(buf3.toString());
```

Streams:

A stream is an abstract interface for working with streaming data in Node.js. The stream module

provides an API for implementing the stream interface.

There are many stream objects provided by Node.js. For instance, a request to an HTTP server and process.stdout are both stream instances.

Streams can be readable, writable, or both. All streams are instances of EventEmitter.

To access the stream module:

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

The stream module is useful for creating new types of stream instances. It is usually not necessary to

use the stream module to consume streams.

Types of streams

There are four fundamental stream types within Node.js:

- Writable: streams to which data can be written (for example, fs.createWriteStream()).
- Readable: streams from which data can be read (for example, fs.createReadStream()).
- Duplex: streams that are both Readable and Writable (for example, net.Socket).
- Transform: Duplex streams that can modify or transform the data as it is written and read (for example, zlib.createDeflate()).

Additionally, this module includes the utility functions stream.pipeline(), stream.finished() and stream.Readable.from().

Object mode

All streams created by Node.js APIs operate exclusively on strings and Buffer (or Uint8Array) objects. It is possible, however, for stream implementations to work with other types of JavaScript values (with the exception of null, which serves a special purpose within streams). Such streams are considered to operate in "object mode".

Stream instances are switched into object mode using the objectMode option when the stream is created. Attempting to switch an existing stream into object mode is not safe.

Buffering

Both Writable and Readable streams will store data in an internal buffer that can be retrieved using writable.writableBuffer or readable.readableBuffer, respectively.

The amount of data potentially buffered depends on the highWaterMark option passed into the stream's constructor. For normal streams, the highWaterMark option specifies a total number of bytes. For streams operating in object mode, the highWaterMark specifies a total number of objects.

Data is buffered in Readable streams when the implementation calls stream.push(chunk). If the consumer of the Stream does not call stream.read(), the data will sit in the internal queue until it is consumed.

Once the total size of the internal read buffer reaches the threshold specified by highWaterMark, the stream will temporarily stop reading data from the underlying resource until the data currently buffered can be consumed (that is, the stream will stop calling the internal readable._read() method that is used to fill the read buffer).

Data is buffered in Writable streams when the writable.write(chunk) method is called repeatedly. While the total size of the internal write buffer is below the threshold set by highWaterMark, calls to writable.write() will return true. Once the size of the internal buffer reaches or exceeds the highWaterMark, false will be returned.

A key goal of the stream API, particularly the stream.pipe() method, is to limit the buffering of data to acceptable levels such that sources and destinations of differing speeds will not overwhelm the available memory.

The highWaterMark option is a threshold, not a limit: it dictates the amount of data that a stream buffers before it stops asking for more data. It does not enforce a strict memory limitation in general. Specific stream implementations may choose to enforce stricter limits but doing so is optional.

Because Duplex and Transform streams are both Readable and Writable, each maintains *two* separate internal buffers used for reading and writing, allowing each side to operate independently of the other while maintaining an appropriate and efficient flow of data. For example, net.Socket instances

are Duplex streams whose Readable side allows consumption of data received *from* the socket and whose Writable side allows writing data *to* the socket. Because data may be written to the socket at a faster or slower rate than data is received, each side should operate (and buffer) independently of the other.

Writable streams

Writable streams are an abstraction for a destination to which data is written.

Examples of Writable streams include:

- HTTP requests, on the client
- HTTP responses, on the server
- fs write streams
- zlib streams
- crypto streams
- TCP sockets
- child process stdin
- process.stdout, process.stderr

Some of these examples are actually Duplex streams that implement the Writable interface.

All Writable streams implement the interface defined by the stream.Writable class.

Class: stream.Writable

Readable streams

Readable streams are an abstraction for a *source* from which data is consumed.

Examples of Readable streams include:

- HTTP responses, on the client
- HTTP requests, on the server
- fs read streams
- zlib streams
- crypto streams
- TCP sockets
- child process stdout and stderr
- process.stdin

All Readable streams implement the interface defined by the stream.Readable class.

Two reading modes

Readable streams effectively operate in one of two modes: flowing and paused. These modes are

separate from object mode. A Readable stream can be in object mode or not, regardless of whether it is in flowing mode or paused mode.

- 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.

Class: stream.Readable

Duplex and transform streams

Class: stream.Duplex

Duplex streams are streams that implement both the Readable and Writable interfaces.

Examples of Duplex streams include:

- TCP sockets
- zlib streams
- crypto streams

Class: stream.Transform

Transform streams are Duplex streams where the output is in some way related to the input. Like all Duplex streams, Transform streams implement both the Readable and Writable interfaces.

Examples of Transform streams include:

- zlib streams
- crypto streams