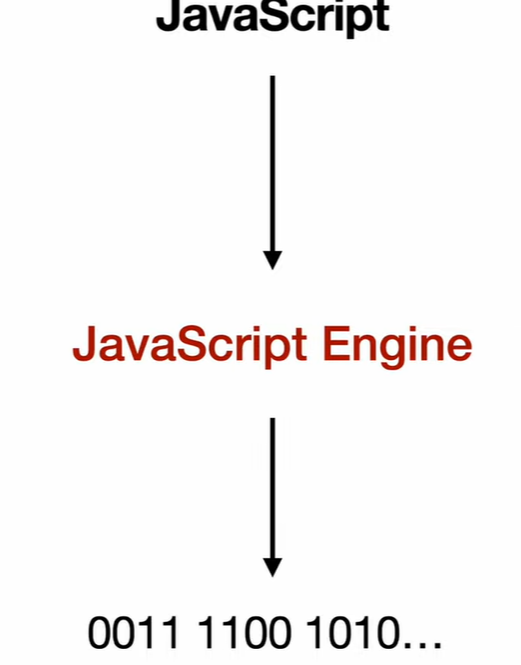
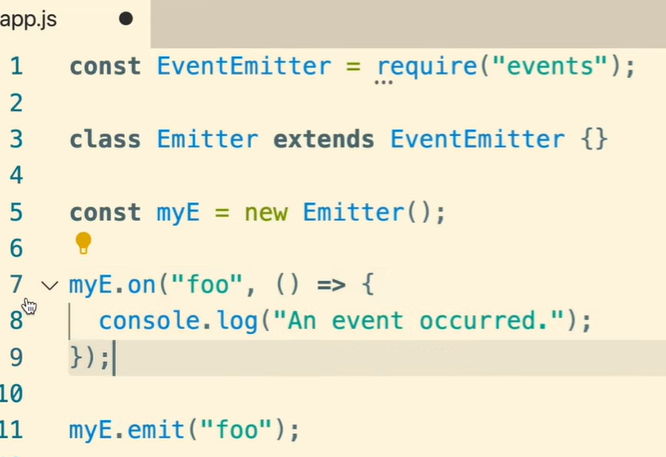


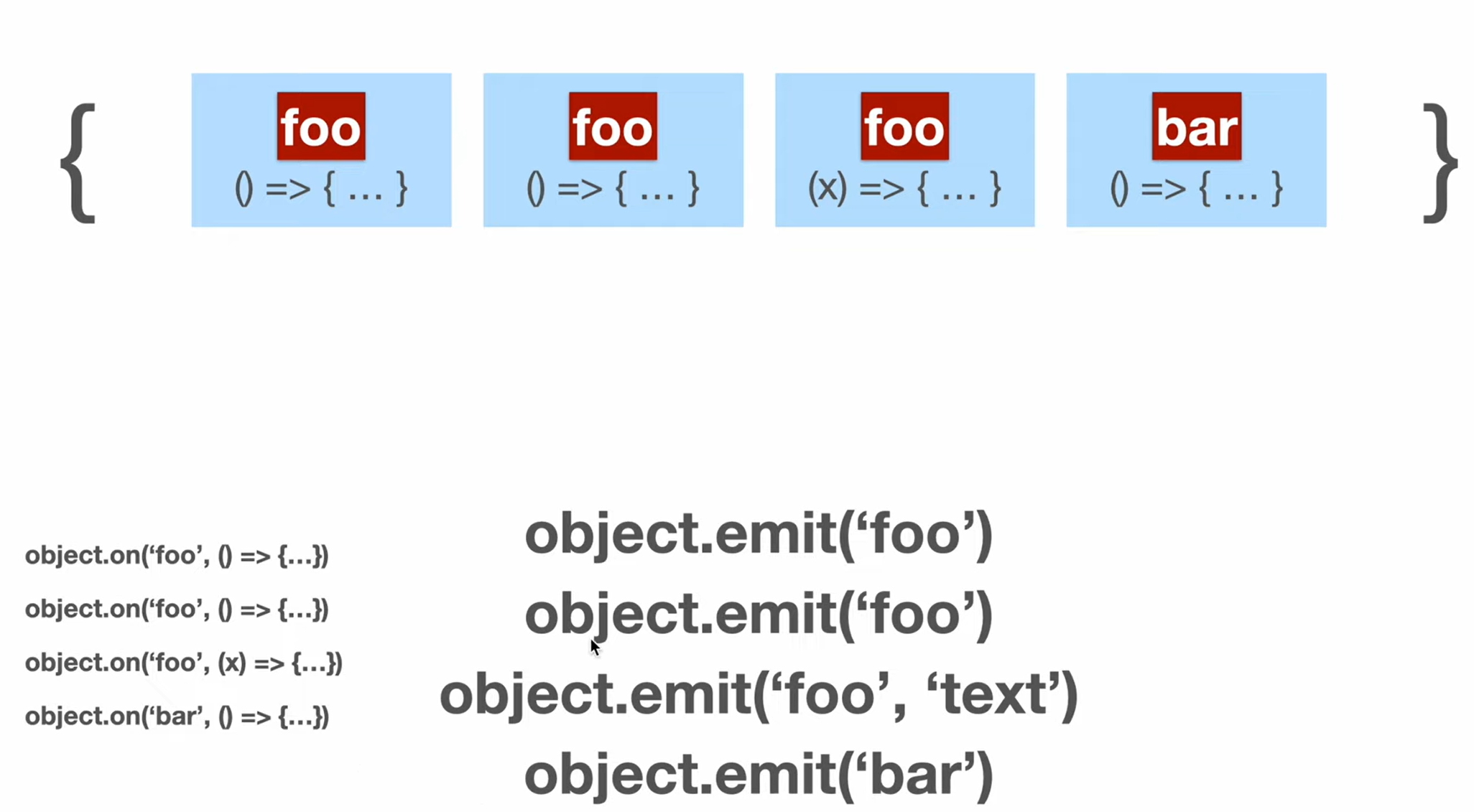
We have to specify processor architecture cuz the instructions vary.



V8 is the latest js engine.

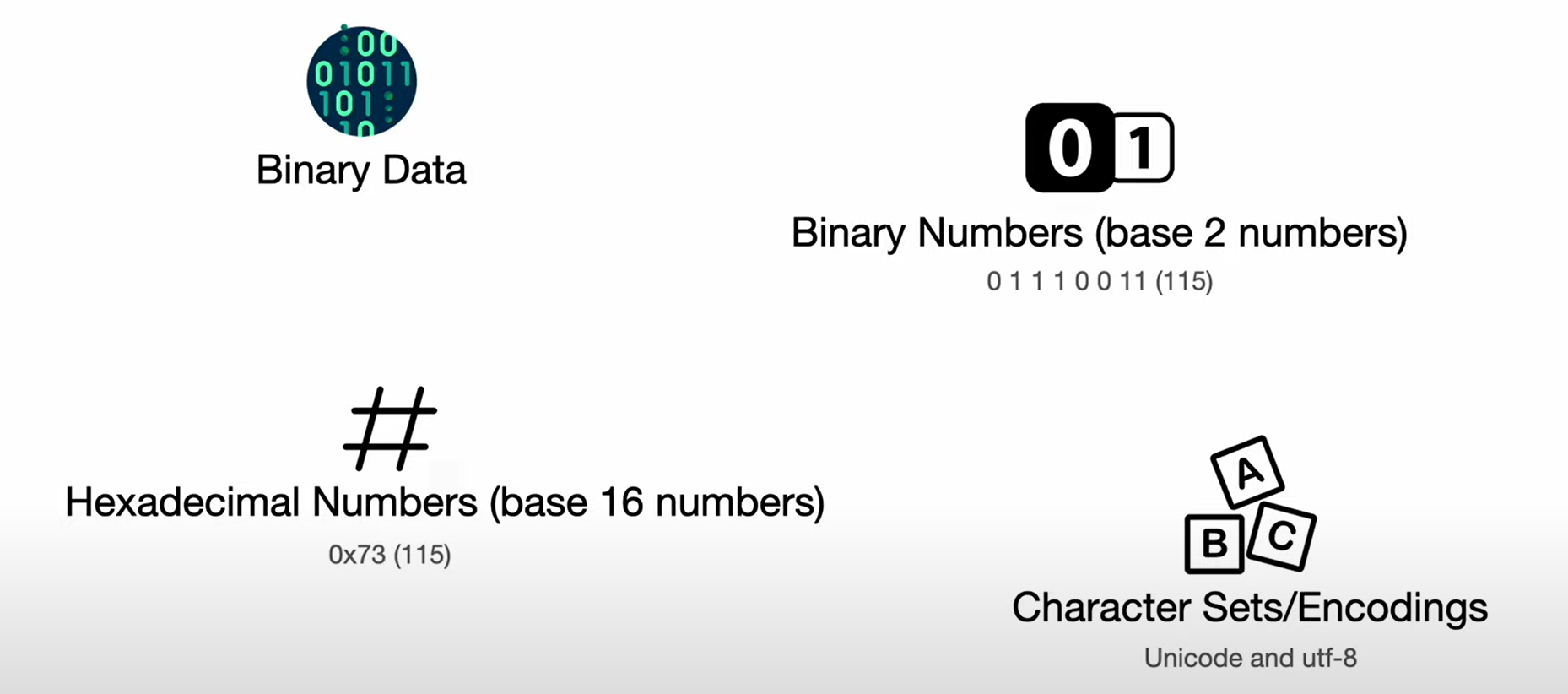
***Event Emitter***:

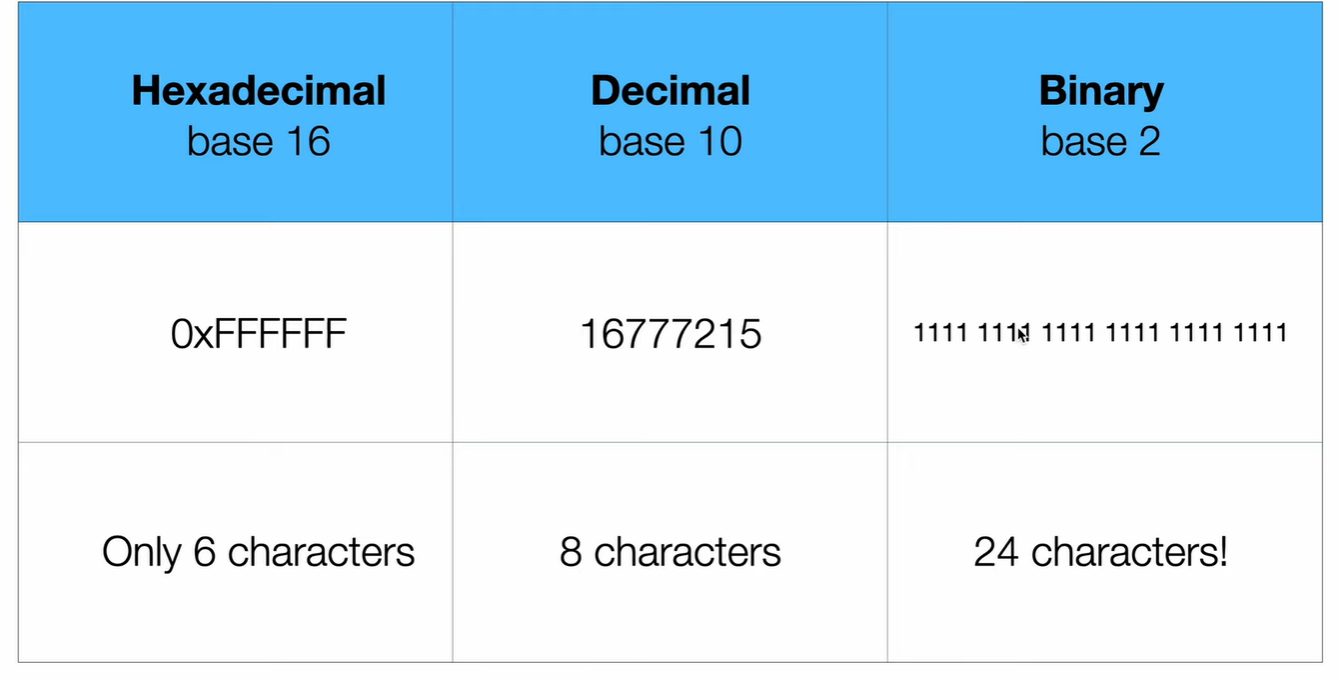


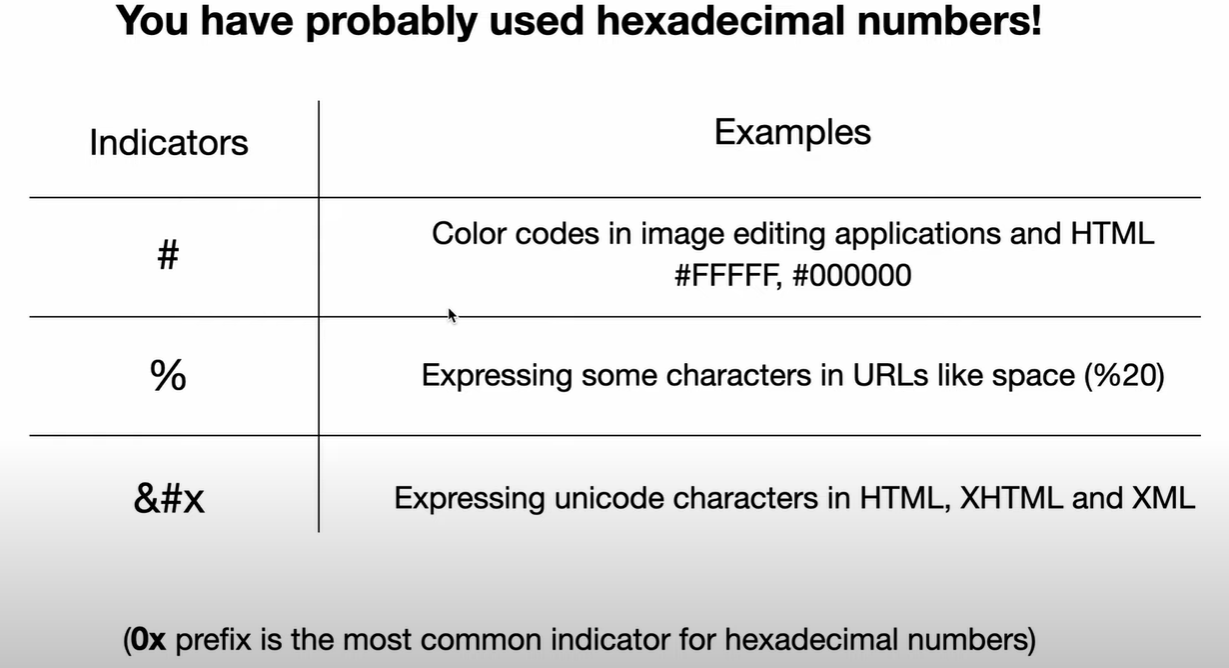


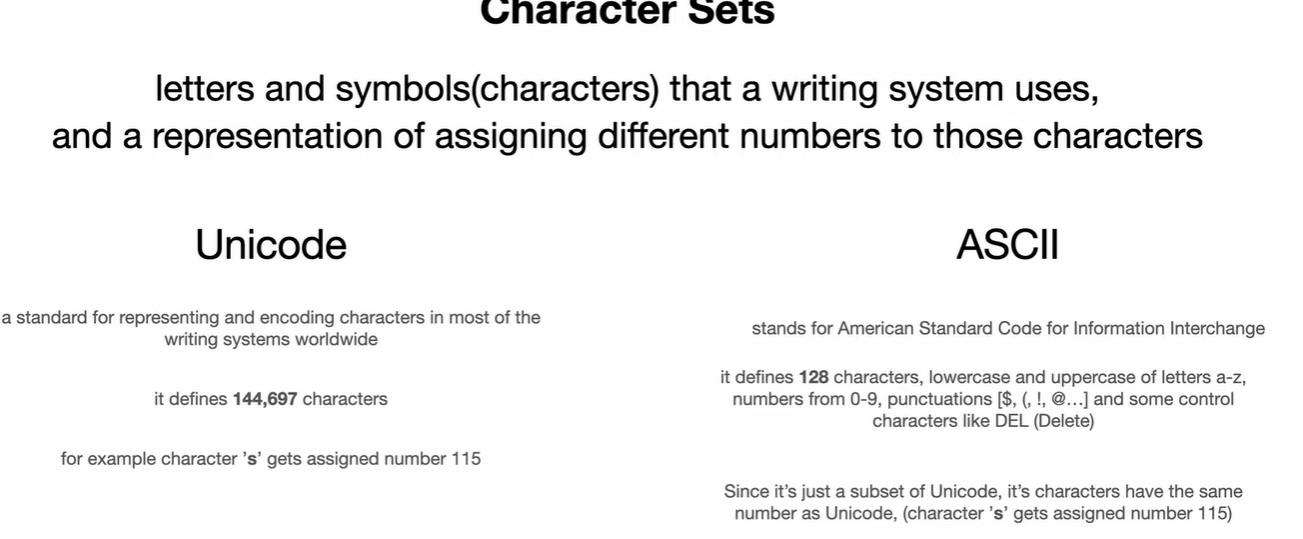
Event Eventer implementation is just a javascript code.

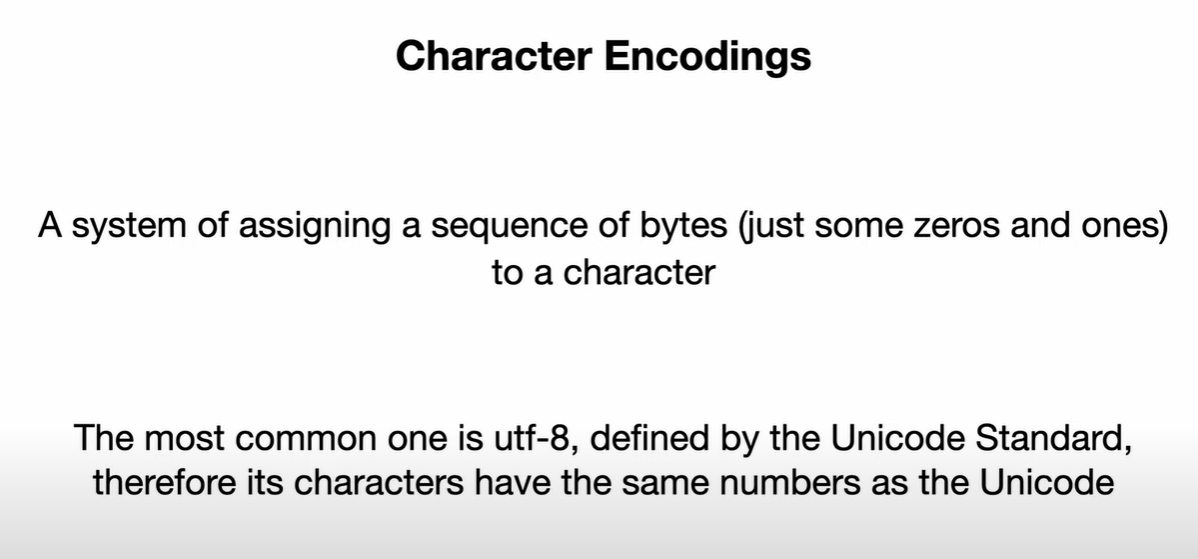
***Buffers***:

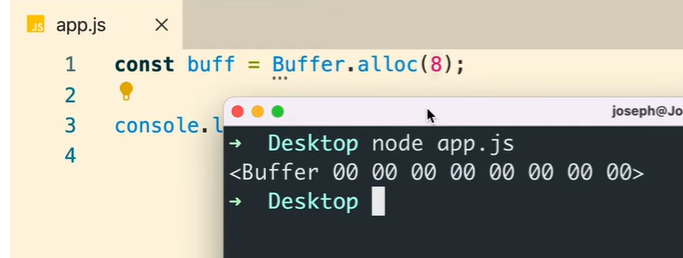




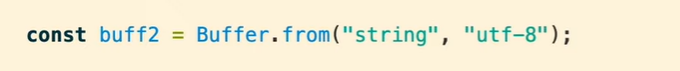








Each number is a hexadecimal number



Find out how many bytes “string” needs and create a buffer

const buf = Buffer.alloc(5);

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

const buf = Buffer.alloc(5, 'a'); // Prints: <Buffer 61 61 61 61 61>

If fill is specified, the allocated Buffer will be initialized by calling buf.fill(fill), encoding defaults to utf-8

const buf = Buffer.alloc(11, 'aGVsbG8gd29ybGQ=', 'base64');

// Prints: <Buffer 68 65 6c 6c 6f 20 77 6f 72 6c 64>

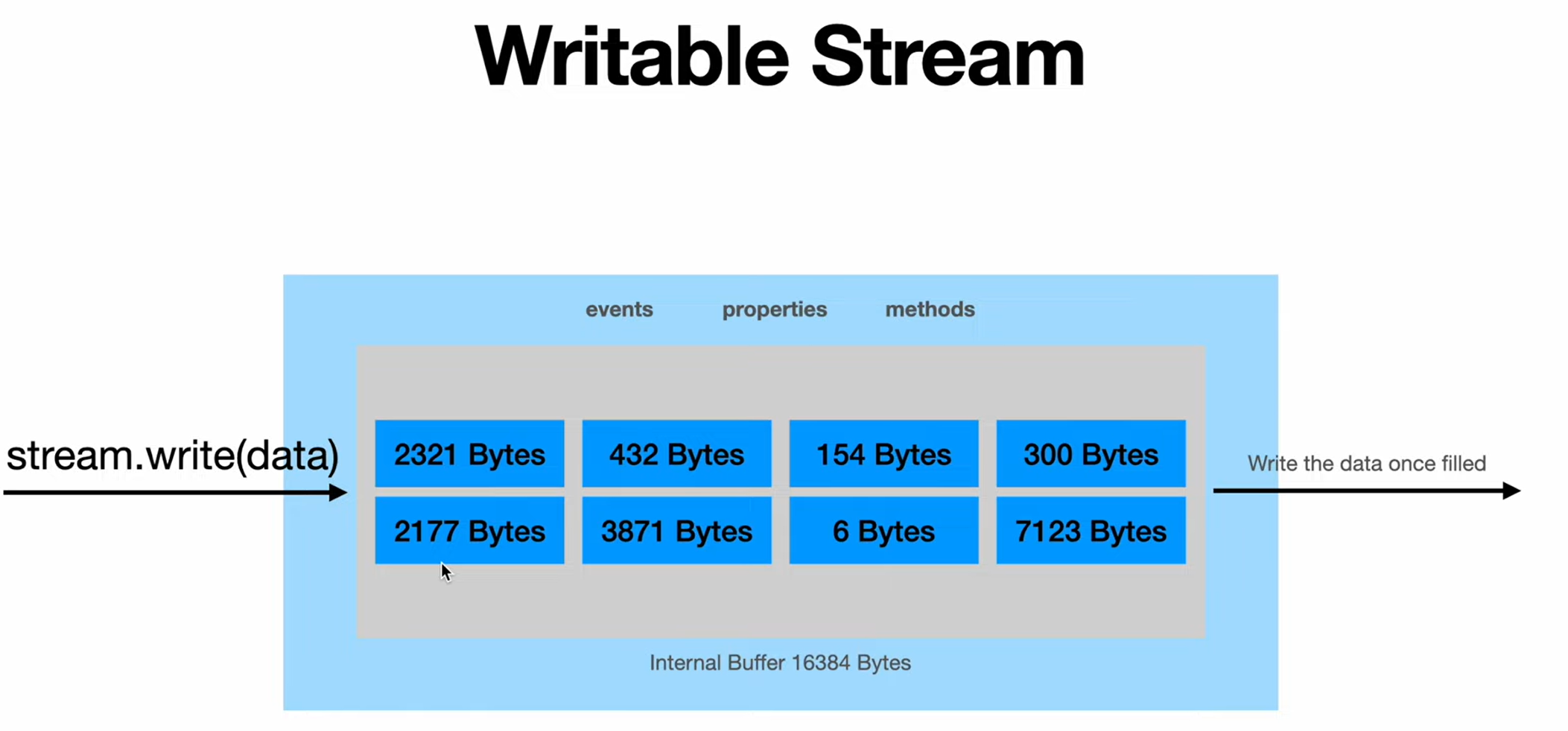
If both fill and encoding are specified, the allocated Buffer will be initialized by calling buf.fill(fill, encoding)

const buf = Buffer.allocUnsafe(10);

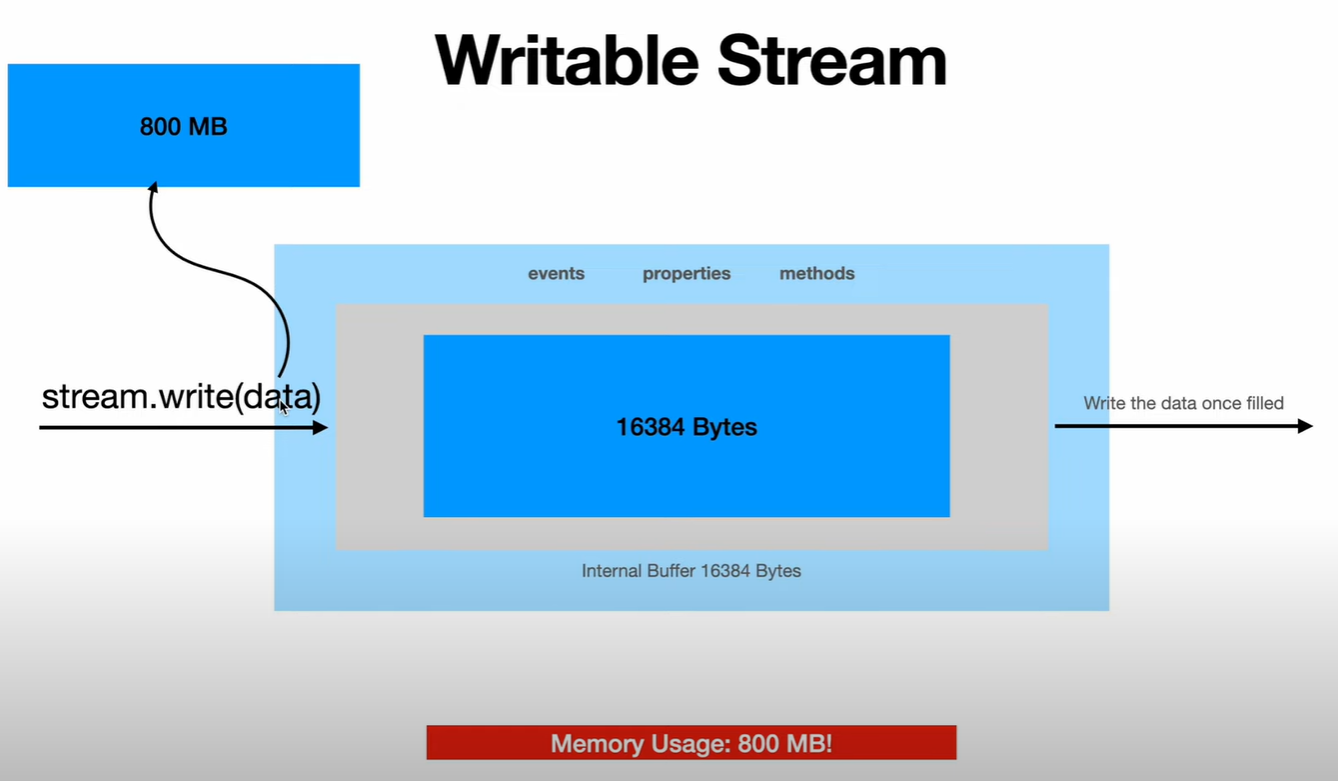
// Prints(contents may vary):<Buffer a0 8b 28 3f 01 00 00 00 50 32>

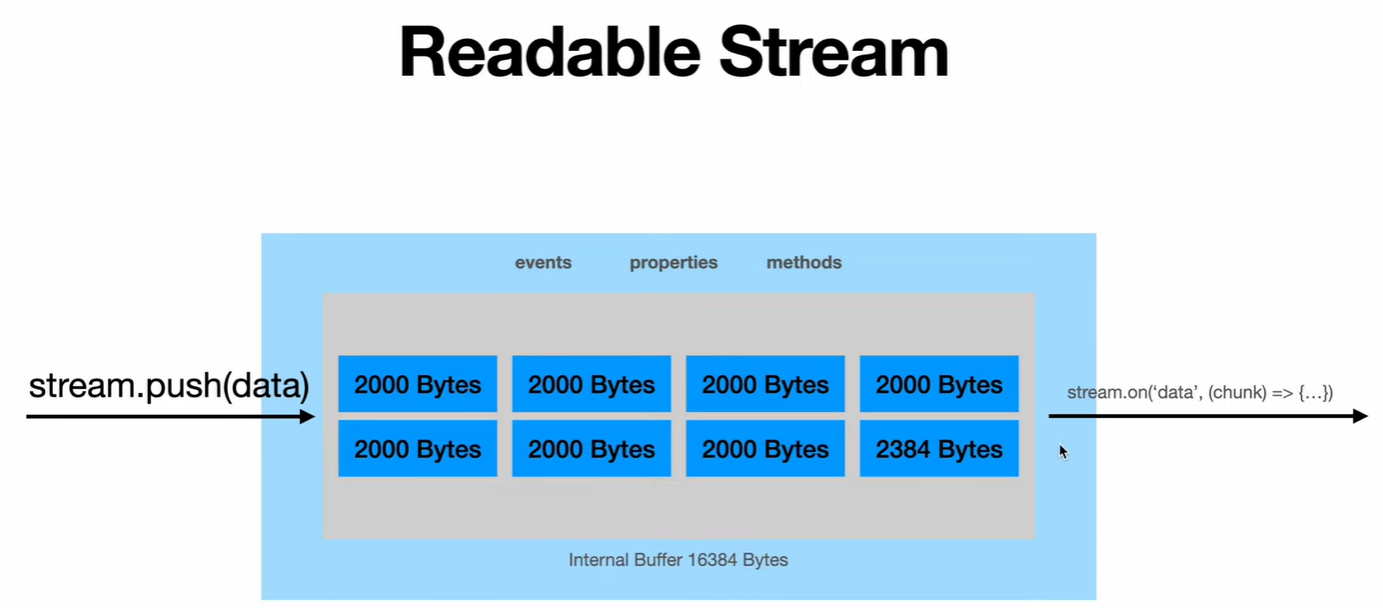
The underlying memory for Buffer instances created in this way is *not initialized*. The contents of the newly created Buffer are unknown and *may contain sensitive data*.

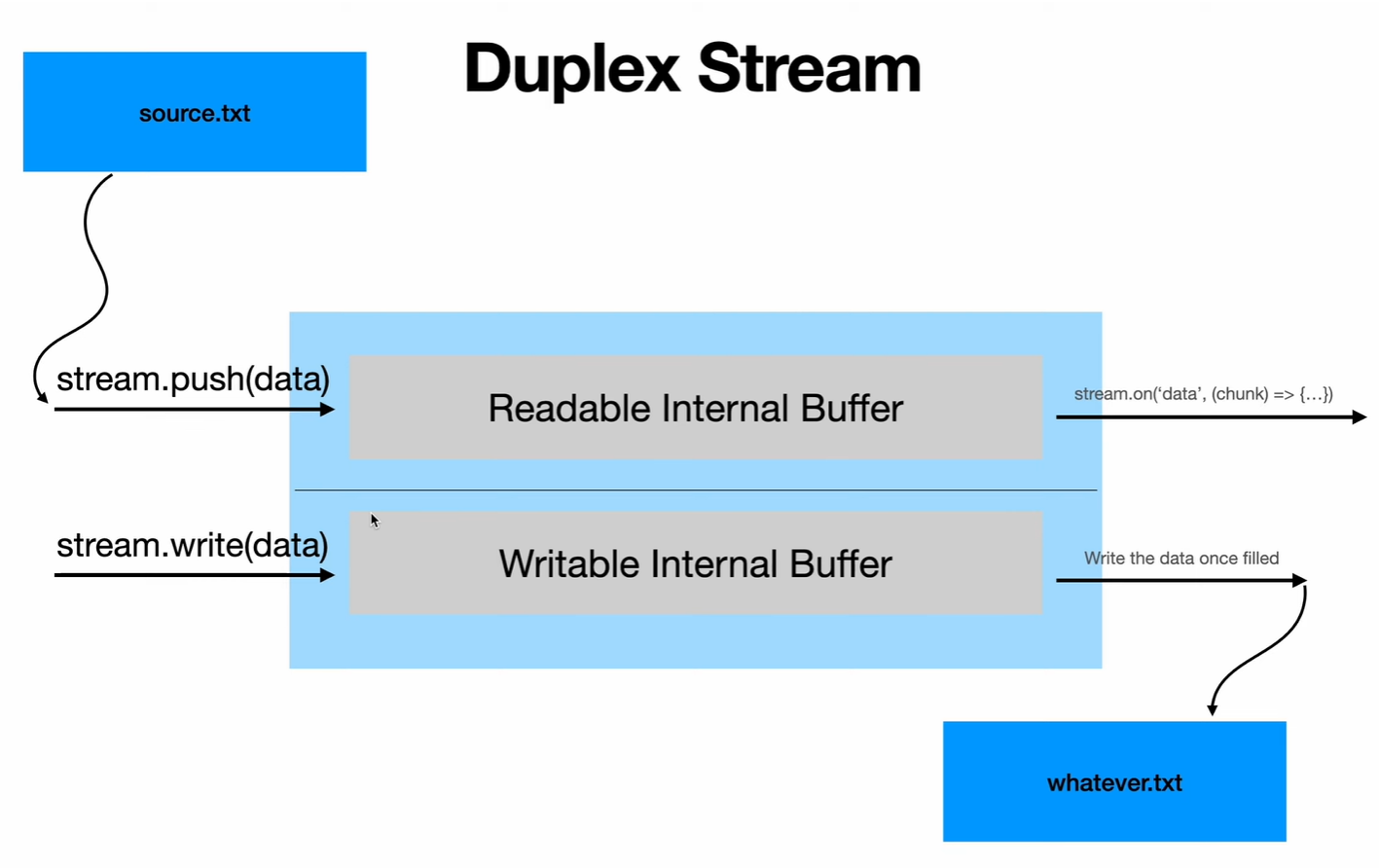
***Streams***:

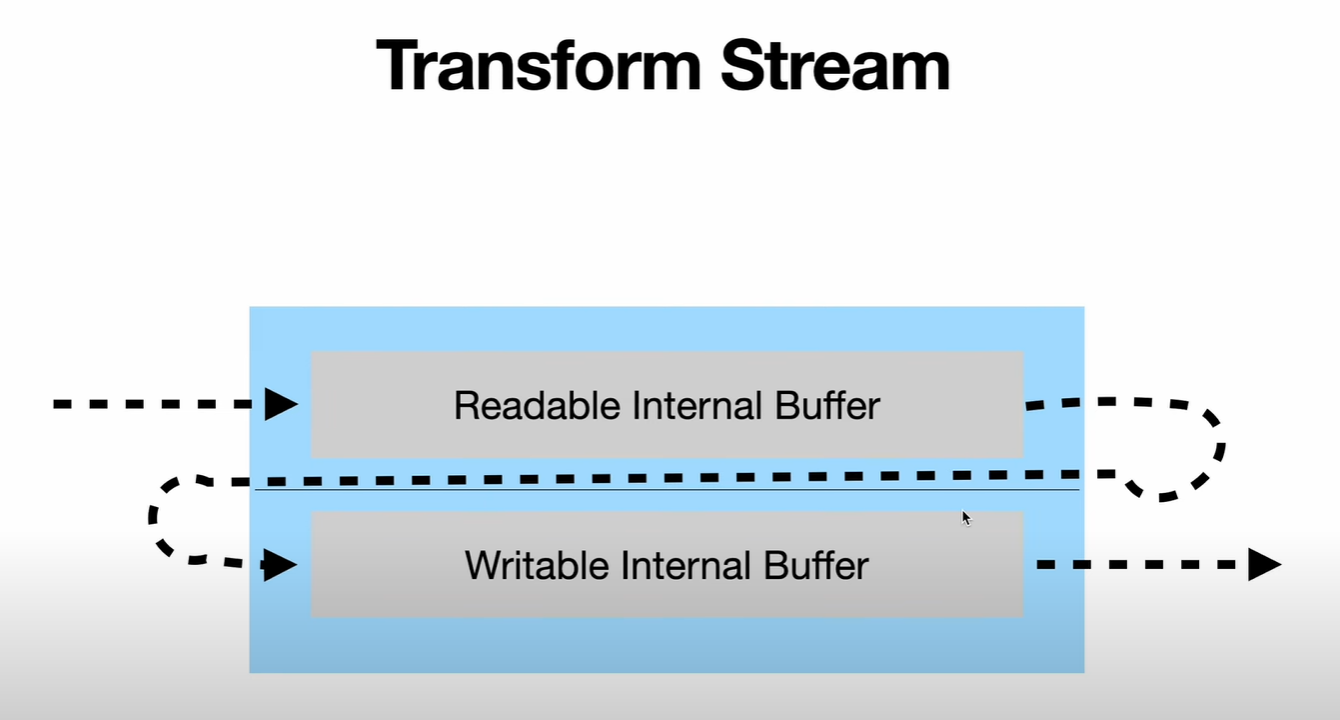


If the data written is more than the internal buffer, nodejs will buffer the remaining data in the memory and pull it out once the internal buffer is drained.



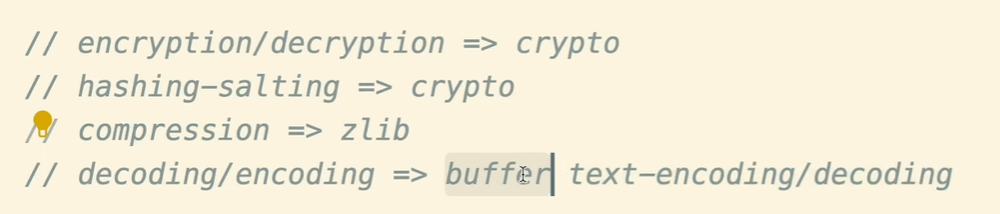






If the output from the readable internal buffer is directly connected to the input of the writable internal buffer it is called Pass Through Stream.

When transform function is implemented which transforms the data given to writable internal buffer then it’s called transform stream, good example would me zlib module.



Streams -> <https://github.com/agile8118/understanding-streams-final>

networking -> <https://github.com/agile8118/node-networking>

In Node.js, there are several ways to create processes that can run concurrently to improve performance and scalability. The main differences between these methods are the scope of the process and the communication mechanisms between them:

* **Child Process**: The child\_process module provides the ability to spawn new processes and communicate with them using standard input/output streams. A child process runs independently of the parent process and can be used to execute external commands or other Node.js scripts. Child processes can be used to perform background tasks or to split a single process into multiple, more manageable parts.
* **Workers**: Worker\_threads module allows you to run JavaScript files or modules in a separate thread, using the SharedArrayBuffer and Atomics API to share memory between threads. A worker runs in the same process as the parent, but in a different thread, allowing to use multiple CPU cores.
* **Cluster**: The cluster module allows you to create a cluster of worker processes that share the same port, allowing your application to take full advantage of multiple cores and improve performance. Each worker process runs in a separate process and can be used to handle different requests. The cluster module provides an easy way to create and manage worker processes, and it also provides a way to share the same state between the workers and the master.