Buffer

Stability: 2 - Stable

Source Code: lib/buffer.js

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

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 `Uint8Array`s wherever `Buffer`s are supported as well.

While the `Buffer` class is available within the global scope, it is still recommended to explicitly reference it via an import or require statement.

```
import { Buffer } from 'node:buffer';

// Creates a zero-filled Buffer of length 10.
const buf1 = Buffer.alloc(10);

// Creates a Buffer of length 10,
// filled with bytes which all have the value `1`.
const buf2 = Buffer.alloc(10, 1);

// Creates an uninitialized buffer of length 10.
// This is faster than calling Buffer.alloc() but the returned
// Buffer instance might contain old data that needs to be
// overwritten using fill(), write(), or other functions that fill the Buffer's
// contents.
const buf3 = Buffer.allocUnsafe(10);

// Creates a Buffer containing the bytes [1, 2, 3].
const buf4 = Buffer.from([1, 2, 3]);

// Creates a Buffer containing the bytes [1, 1, 1, 1] – the entries
// are all truncated using `(value & 255)` to fit into the range 0–255.
const buf5 = Buffer.from([257, 257.5, -255, '1']);

// Creates a Buffer containing the UTF-8-encoded bytes for the string 'tést':
// [0x74, 0xc3, 0xa9, 0x73, 0x74] (in hexadecimal notation)
// [116, 195, 169, 115, 116] (in decimal notation)
const buf6 = Buffer.from('tést');

// Creates a Buffer containing the Latin-1 bytes [0x74, 0xe9, 0x73, 0x74].
const buf7 = Buffer.from('tést', 'latin1');
```

```
const { Buffer } = require('node:buffer');

// Creates a zero-filled Buffer of length 10.
const buf1 = Buffer.alloc(10);

// Creates a Buffer of length 10,
// filled with bytes which all have the value `1`.
const buf2 = Buffer.alloc(10, 1);

// Creates an uninitialized buffer of length 10.
// This is faster than calling Buffer.alloc() but the returned
// Buffer instance might contain old data that needs to be
// overwritten using fill(), write(), or other functions that fill the Buffer's
// contents.
const buf3 = Buffer.allocUnsafe(10);

// Creates a Buffer containing the bytes [1, 2, 3].
const buf4 = Buffer.from([1, 2, 3]);

// Creates a Buffer containing the bytes [1, 1, 1, 1] – the entries
// are all truncated using `(value & 255)` to fit into the range 0–255.
const buf5 = Buffer.from([257, 257.5, -255, '1']);

// Creates a Buffer containing the UTF-8-encoded bytes for the string 'tést':
// [0x74, 0xc3, 0xa9, 0x73, 0x74] (in hexadecimal notation)
// [116, 195, 169, 115, 116] (in decimal notation)
const buf6 = Buffer.from('tést');

// Creates a Buffer containing the Latin-1 bytes [0x74, 0xe9, 0x73, 0x74].
const buf7 = Buffer.from('tést', 'latin1');
```

COPY

Buffers and character encodings

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

```
import { Buffer } from 'node:buffer';

const buf = Buffer.from('hello world', 'utf8');

console.log(buf.toString('hex'));
// Prints: 68656c6c6f20776f726c64
console.log(buf.toString('base64'));
// Prints: aGVsbG8gd29ybGQ=

console.log(Buffer.from('fhqwhgads', 'utf8'));
// Prints: <Buffer 66 68 71 77 68 67 61 64 73>
console.log(Buffer.from('fhqwhgads', 'utf16le'));
// Prints: <Buffer 66 00 68 00 71 00 77 00 68 00 67 00 61 00 64 00 73 00>
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.from('hello world', 'utf8');

console.log(buf.toString('hex'));
// Prints: 68656c6c6f20776f726c64
console.log(buf.toString('base64'));
// Prints: aGVsbG8gd29ybGQ=

console.log(Buffer.from('fhqwhgads', 'utf8'));
// Prints: <Buffer 66 68 71 77 68 67 61 64 73>
console.log(Buffer.from('fhqwhgads', 'utf16le'));
// Prints: <Buffer 66 00 68 00 71 00 77 00 68 00 67 00 61 00 64 00 73 00>
```

COPY

Node.js buffers accept all case variations of encoding strings that they receive. For example, UTF-8 can be specified as `'utf8'`, `'UTF8'`, or `'uTf8'`.

The character encodings currently supported by Node.js are the following:

- `'utf8'` (alias: `'utf-8'`): Multi-byte encoded Unicode characters. Many web pages and other document formats use UTF-8. This is the default character encoding. When decoding a `Buffer` into a string that does not exclusively contain valid UTF-8 data, the Unicode replacement character `U+FFFD` `�` will be used to represent those errors.
- `'utf16le'` (alias: `'utf-16le'`): Multi-byte encoded Unicode characters. Unlike `'utf8'`, each character in the string will be encoded using either 2 or 4 bytes. Node.js only supports the little-endian variant of UTF-16.
- `'latin1'`: Latin-1 stands for ISO-8859-1. This character encoding only supports the Unicode characters from `U+0000` to `U+00FF`. Each character is encoded using a single byte. Characters that do not fit into that range are truncated and will be mapped to characters in that range.

Converting a `Buffer` into a string using one of the above is referred to as decoding, and converting a string into a `Buffer` is referred to as encoding.

Node.js also supports the following binary-to-text encodings. For binary-to-text encodings, the naming convention is reversed: Converting a `Buffer` into a string is typically referred to as encoding, and converting a string into a `Buffer` as decoding.

- `'base64'`: Base64 encoding. When creating a `Buffer` from a string, this encoding will also correctly accept "URL and Filename Safe Alphabet" as specified in RFC 4648, Section 5. Whitespace characters such as spaces, tabs, and new lines contained within the base64-encoded string are ignored.
- `'base64url'`: base64url encoding as specified in RFC 4648, Section 5. When creating a `Buffer` from a string, this encoding will also correctly accept regular base64-encoded strings. When encoding a `Buffer` to a string, this encoding will omit padding.
- `'hex'`: Encode each byte as two hexadecimal characters. Data truncation may occur when decoding strings that do not exclusively consist of an even number of hexadecimal characters. See below for an example.

The following legacy character encodings are also supported:

- `'ascii'`: For 7-bit ASCII data only. When encoding a string into a `Buffer`, this is equivalent to using `'latin1'`. When decoding a `Buffer` into a string, using this encoding will additionally unset the highest bit of each byte before decoding as `'latin1'`. Generally, there should be no reason to use this encoding, as `'utf8'` (or, if the data is known to always be ASCII-only, `'latin1'`) will be a better choice when encoding or decoding ASCII-only text. It is only provided for legacy compatibility.
- `'binary'`: Alias for `'latin1'`. The name of this encoding can be very misleading, as all of the encodings listed here convert between strings and binary data. For converting between strings and `Buffer`s, typically `'utf8'` is the right choice.

- `'ucs2'`, `'ucs-2'`: Aliases of `'utf16le'`. UCS-2 used to refer to a variant of UTF-16 that did not support characters that had code points larger than U+FFFF. In Node.js, these code points are always supported.

```mjs
import { Buffer } from 'node:buffer';

Buffer.from('1ag123', 'hex');
// Prints <Buffer 1a>, data truncated when first non-hexadecimal value
// ('g') encountered.

Buffer.from('1a7', 'hex');
// Prints <Buffer 1a>, data truncated when data ends in single digit ('7').

Buffer.from('1634', 'hex');
// Prints <Buffer 16 34>, all data represented.
```

```cjs
const { Buffer } = require('node:buffer');

Buffer.from('1ag123', 'hex');
// Prints <Buffer 1a>, data truncated when first non-hexadecimal value
// ('g') encountered.

Buffer.from('1a7', 'hex');
// Prints <Buffer 1a>, data truncated when data ends in single digit ('7').

Buffer.from('1634', 'hex');
// Prints <Buffer 16 34>, all data represented.
```

COPY

Modern Web browsers follow the WHATWG Encoding Standard which aliases both `'latin1'` and `'ISO-8859-1'` to `'win-1252'`. This means that while doing something like `http.get()`, if the returned charset is one of those listed in the WHATWG specification it is possible that the server actually returned `'win-1252'`-encoded data, and using `'latin1'` encoding may incorrectly decode the characters.

Buffers and TypedArrays

`Buffer` instances are also JavaScript `Uint8Array` and `TypedArray` instances. All `TypedArray` methods are available on `Buffer`s. There are, however, subtle incompatibilities between the `Buffer` API and the `TypedArray` API.

In particular:

- While `TypedArray.prototype.slice()` creates a copy of part of the `TypedArray`, `Buffer.prototype.slice()` creates a view over the existing `Buffer` without copying. This behavior can be surprising, and only exists for legacy compatibility. `TypedArray.prototype.subarray()` can be used to achieve the behavior of `Buffer.prototype.slice()` on both `Buffer`s and other `TypedArray`s and should be preferred.
- `buf.toString()` is incompatible with its `TypedArray` equivalent.
- A number of methods, e.g. `buf.indexOf()`, support additional arguments.

There are two ways to create new `TypedArray` instances from a `Buffer`:

- Passing a `Buffer` to a `TypedArray` constructor will copy the `Buffer`s contents, interpreted as an array of integers, and not as a byte sequence of the target type.

```mjs
import { Buffer } from 'node:buffer';

const buf = Buffer.from([1, 2, 3, 4]);
const uint32array = new Uint32Array(buf);

console.log(uint32array);

// Prints: Uint32Array(4) [ 1, 2, 3, 4 ]
```

```cjs
const { Buffer } = require('node:buffer');

const buf = Buffer.from([1, 2, 3, 4]);
const uint32array = new Uint32Array(buf);

console.log(uint32array);

// Prints: Uint32Array(4) [ 1, 2, 3, 4 ]
```

COPY

- Passing the `Buffer` s underlying `ArrayBuffer` will create a `TypedArray` that shares its memory with the `Buffer` .

```mjs
import { Buffer } from 'node:buffer';

const buf = Buffer.from('hello', 'utf16le');
const uint16array = new Uint16Array(
  buf.buffer,
  buf.byteOffset,
  buf.length / Uint16Array.BYTES_PER_ELEMENT);

console.log(uint16array);

// Prints: Uint16Array(5) [ 104, 101, 108, 108, 111 ]
```

```cjs
const { Buffer } = require('node:buffer');

const buf = Buffer.from('hello', 'utf16le');
const uint16array = new Uint16Array(
  buf.buffer,
  buf.byteOffset,
  buf.length / Uint16Array.BYTES_PER_ELEMENT);

console.log(uint16array);

// Prints: Uint16Array(5) [ 104, 101, 108, 108, 111 ]
```

COPY

It is possible to create a new `Buffer` that shares the same allocated memory as a `TypedArray` instance by using the `TypedArray` object's `.buffer` property in the same way. `Buffer.from()` behaves like `new Uint8Array()` in this context.

```mjs
import { Buffer } from 'node:buffer';

const arr = new Uint16Array(2);

arr[0] = 5000;
arr[1] = 4000;

// Copies the contents of `arr`.
const buf1 = Buffer.from(arr);

// Shares memory with `arr`.
const buf2 = Buffer.from(arr.buffer);

console.log(buf1);
// Prints: <Buffer 88 a0>
console.log(buf2);
// Prints: <Buffer 88 13 a0 0f>

arr[1] = 6000;

console.log(buf1);
// Prints: <Buffer 88 a0>
console.log(buf2);
// Prints: <Buffer 88 13 70 17>
```

```cjs
const { Buffer } = require('node:buffer');

const arr = new Uint16Array(2);

arr[0] = 5000;
arr[1] = 4000;

// Copies the contents of `arr`.
const buf1 = Buffer.from(arr);

// Shares memory with `arr`.
const buf2 = Buffer.from(arr.buffer);

console.log(buf1);
// Prints: <Buffer 88 a0>
console.log(buf2);
// Prints: <Buffer 88 13 a0 0f>

arr[1] = 6000;

console.log(buf1);
// Prints: <Buffer 88 a0>
console.log(buf2);
// Prints: <Buffer 88 13 70 17>
```

When creating a `Buffer` using a `TypedArray`'s `.buffer`, it is possible to use only a portion of the underlying `ArrayBuffer` by passing in `byteOffset` and `length` parameters.

```mjs
import { Buffer } from 'node:buffer';

const arr = new Uint16Array(20);
const buf = Buffer.from(arr.buffer, 0, 16);

console.log(buf.length);
// Prints: 16
```

```cjs
const { Buffer } = require('node:buffer');

const arr = new Uint16Array(20);
const buf = Buffer.from(arr.buffer, 0, 16);

console.log(buf.length);
// Prints: 16
```

COPY

The `Buffer.from()` and `TypedArray.from()` have different signatures and implementations. Specifically, the `TypedArray` variants accept a second argument that is a mapping function that is invoked on every element of the typed array:

- `TypedArray.from(source[, mapFn[, thisArg]])`

The `Buffer.from()` method, however, does not support the use of a mapping function:

- `Buffer.from(array)`
- `Buffer.from(buffer)`
- `Buffer.from(arrayBuffer[, byteOffset[, length]])`
- `Buffer.from(string[, encoding])`

Buffers and iteration

`Buffer` instances can be iterated over using `for..of` syntax:

```mjs
import { Buffer } from 'node:buffer';

const buf = Buffer.from([1, 2, 3]);

for (const b of buf) {
  console.log(b);
}
// Prints:
//   1
//   2
//   3
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.from([1, 2, 3]);

for (const b of buf) {
  console.log(b);
}
// Prints:
//   1
//   2
//   3
```

COPY

Additionally, the `buf.values()`, `buf.keys()`, and `buf.entries()` methods can be used to create iterators.

Class: Blob

A `Blob` encapsulates immutable, raw data that can be safely shared across multiple worker threads.

new buffer.Blob([sources[, options]])

- `sources` `<string[]>` | `<ArrayBuffer[]>` | `<TypedArray[]>` | `<DataView[]>` | `<Blob[]>` An array of string, `<ArrayBuffer>`, `<TypedArray>`, `<DataView>`, or `<Blob>` objects, or any mix of such objects, that will be stored within the `Blob`.
- `options` `<Object>`
  - `endings` `<string>` One of either `'transparent'` or `'native'`. When set to `'native'`, line endings in string source parts will be converted to the platform native line-ending as specified by `require('node:os').EOL`.
  - `type` `<string>` The Blob content-type. The intent is for `type` to convey the MIME media type of the data, however no validation of the type format is performed.

Creates a new `Blob` object containing a concatenation of the given sources.

`<ArrayBuffer>`, `<TypedArray>`, `<DataView>`, and `<Buffer>` sources are copied into the 'Blob' and can therefore be safely modified after the 'Blob' is created.

String sources are encoded as UTF-8 byte sequences and copied into the Blob. Unmatched surrogate pairs within each string part will be replaced by Unicode U+FFFD replacement characters.

blob.arrayBuffer()

- Returns: `<Promise>`

Returns a promise that fulfills with an `<ArrayBuffer>` containing a copy of the `Blob` data.

blob.bytes()

The `blob.bytes()` method returns the byte of the `Blob` object as a `Promise<Uint8Array>`.

```
const blob = new Blob(['hello']);
blob.bytes().then((bytes) => {
  console.log(bytes); // Outputs: Uint8Array(5) [ 104, 101, 108, 108, 111 ]
});
```

COPY

blob.size

The total size of the `Blob` in bytes.

blob.slice([start[, end[, type]]])

- `start` <number> The starting index.
- `end` <number> The ending index.
- `type` <string> The content-type for the new `Blob`

Creates and returns a new `Blob` containing a subset of this `Blob` objects data. The original `Blob` is not altered.

blob.stream()

- Returns: <ReadableStream>

Returns a new `ReadableStream` that allows the content of the `Blob` to be read.

blob.text()

- Returns: <Promise>

Returns a promise that fulfills with the contents of the `Blob` decoded as a UTF-8 string.

blob.type

- Type: <string>

The content-type of the `Blob` .

Blob objects and MessageChannel

Once a <Blob> object is created, it can be sent via `MessagePort` to multiple destinations without transferring or immediately copying the data. The data contained by the `Blob` is copied only when the `arrayBuffer()` or `text()` methods are called.

```
import { Blob } from 'node:buffer';
import { setTimeout as delay } from 'node:timers/promises';

const blob = new Blob(['hello there']);

const mc1 = new MessageChannel();
const mc2 = new MessageChannel();

mc1.port1.onmessage = async ({ data }) => {
  console.log(await data.arrayBuffer());
  mc1.port1.close();
};

mc2.port1.onmessage = async ({ data }) => {
  await delay(1000);
  console.log(await data.arrayBuffer());
  mc2.port1.close();
};

mc1.port2.postMessage(blob);
mc2.port2.postMessage(blob);

// The Blob is still usable after posting.
blob.text().then(console.log);
```

```
const { Blob } = require('node:buffer');
const { setTimeout: delay } = require('node:timers/promises');

const blob = new Blob(['hello there']);

const mc1 = new MessageChannel();
const mc2 = new MessageChannel();

mc1.port1.onmessage = async ({ data }) => {
  console.log(await data.arrayBuffer());
  mc1.port1.close();
};

mc2.port1.onmessage = async ({ data }) => {
  await delay(1000);
  console.log(await data.arrayBuffer());
  mc2.port1.close();
};

mc1.port2.postMessage(blob);
mc2.port2.postMessage(blob);

// The Blob is still usable after posting.
blob.text().then(console.log);
```

COPY

Class: Buffer

The `Buffer` class is a global type for dealing with binary data directly. It can be constructed in a variety of ways.

Static method: 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.

```
import { Buffer } from 'node:buffer';

const buf = Buffer.alloc(5);

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

```
const { Buffer } = require('node:buffer');

const buf = Buffer.alloc(5);

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

COPY

If `size` is larger than `buffer.constants.MAX_LENGTH` or smaller than 0, `ERR_OUT_OF_RANGE` is thrown.

If `fill` is specified, the allocated `Buffer` will be initialized by calling `buf.fill(fill)`.

```
import { Buffer } from 'node:buffer';

const buf = Buffer.alloc(5, 'a');

console.log(buf);
// Prints: <Buffer 61 61 61 61 61>
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.alloc(5, 'a');

console.log(buf);
// Prints: <Buffer 61 61 61 61 61>
```

COPY

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

```
import { Buffer } from 'node:buffer';

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

console.log(buf);
// Prints: <Buffer 68 65 6c 6c 6f 20 77 6f 72 6c 64>
```

```
const { Buffer } = require('node:buffer');

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

console.log(buf);
// Prints: <Buffer 68 65 6c 6c 6f 20 77 6f 72 6c 64>
```

COPY

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 `Buffer`s.

A `TypeError` will be thrown if `size` is not a number.

Static method: 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_OUT_OF_RANGE` 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.

```
import { Buffer } from 'node:buffer';

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>
```

```
const { Buffer } = require('node:buffer');

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>
```

COPY

A `TypeError` will be thrown if `size` is not a number.

The `Buffer` module pre-allocates an internal `Buffer` instance of size `Buffer.poolSize` that is used as a pool for the fast allocation of new `Buffer` instances created using `Buffer.allocUnsafe()`, `Buffer.from(array)`, `Buffer.from(string)`, and `Buffer.concat()` only when `size` is less than `Buffer.poolSize >>> 1` (floor of `Buffer.poolSize` divided by two).

Use of this pre-allocated internal memory pool is a key difference between calling `Buffer.alloc(size, fill)` vs. `Buffer.allocUnsafe(size).fill(fill)`. Specifically, `Buffer.alloc(size, fill)` will *never* use the internal `Buffer` pool, while `Buffer.allocUnsafe(size).fill(fill)` *will* use the internal `Buffer` pool if `size` is less than or equal to half `Buffer.poolSize`. The difference is subtle but can be important when an application requires the additional performance that `Buffer.allocUnsafe()` provides.

Static method: Buffer.allocUnsafeSlow(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_OUT_OF_RANGE` is thrown. A zero-length `Buffer` is created if `size` is 0.

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 `buf.fill(0)` to initialize such `Buffer` instances with zeroes.

When using `Buffer.allocUnsafe()` to allocate new `Buffer` instances, allocations less than `Buffer.poolSize >>> 1` (4KiB when default poolSize is used) are sliced from a single pre-allocated `Buffer`. This allows applications to avoid the garbage collection overhead of creating

many individually allocated `Buffer` instances. This approach improves both performance and memory usage by eliminating the need to track and clean up as many individual `ArrayBuffer` objects.

However, in the case where a developer may need to retain a small chunk of memory from a pool for an indeterminate amount of time, it may be appropriate to create an un-pooled `Buffer` instance using `Buffer.allocUnsafeSlow()` and then copying out the relevant bits.

```mjs
import { Buffer } from 'node:buffer';

// Need to keep around a few small chunks of memory.
const store = [];

socket.on('readable', () => {
  let data;
  while (null !== (data = readable.read())) {
    // Allocate for retained data.
    const sb = Buffer.allocUnsafeSlow(10);

    // Copy the data into the new allocation.
    data.copy(sb, 0, 0, 10);

    store.push(sb);
  }
});
```

```cjs
const { Buffer } = require('node:buffer');

// Need to keep around a few small chunks of memory.
const store = [];

socket.on('readable', () => {
  let data;
  while (null !== (data = readable.read())) {
    // Allocate for retained data.
    const sb = Buffer.allocUnsafeSlow(10);

    // Copy the data into the new allocation.
    data.copy(sb, 0, 0, 10);

    store.push(sb);
  }
});
```

COPY

A `TypeError` will be thrown if `size` is not a number.

Static method: Buffer.byteLength(string[, encoding])

- `string` <string> | <Buffer> | <TypedArray> | <DataView> | <ArrayBuffer> | <SharedArrayBuffer> A value to calculate the length of.
- `encoding` <string> If `string` is a string, this is its encoding. **Default:** `'utf8'`.
- Returns: <integer> The number of bytes contained within `string`.

Returns the byte length of a string when encoded using `encoding`. This is not the same as `String.prototype.length`, which does not account for the encoding that is used to convert the string into bytes.

For `'base64'`, `'base64url'`, and `'hex'`, this function assumes valid input. For strings that contain non-base64/hex-encoded data (e.g. whitespace), the return value might be greater than the length of a `Buffer` created from the string.

```
import { Buffer } from 'node:buffer';

const str = '\u00bd + \u00bc = \u00be';

console.log(`${str}: ${str.length} characters, ` +
            `${Buffer.byteLength(str, 'utf8')} bytes`);
// Prints: ½ + ¼ = ¾: 9 characters, 12 bytes
```

```
const { Buffer } = require('node:buffer');

const str = '\u00bd + \u00bc = \u00be';

console.log(`${str}: ${str.length} characters, ` +
            `${Buffer.byteLength(str, 'utf8')} bytes`);
// Prints: ½ + ¼ = ¾: 9 characters, 12 bytes
```

COPY

When `string` is a `Buffer` / `DataView` / `TypedArray` / `ArrayBuffer` / `SharedArrayBuffer`, the byte length as reported by `.byteLength` is returned.

Static method: 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)`.

```
import { Buffer } from 'node:buffer';

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].)
```

```
const { Buffer } = require('node:buffer');

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].)
```

COPY

Static method: 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.

If `totalLength` is not provided, it is calculated from the `Buffer` instances in `list` by adding their lengths.

If `totalLength` is provided, it is coerced to an unsigned integer. If the combined length of the `Buffer`s in `list` exceeds `totalLength`, the result is truncated to `totalLength`.

```
import { Buffer } from 'node:buffer';

// 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
```

```
const { Buffer } = require('node:buffer');

// 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
```

COPY

`Buffer.concat()` may also use the internal `Buffer` pool like `Buffer.allocUnsafe()` does.

Static method: Buffer.copyBytesFrom(view[, offset[, length]])

- `view` `<TypedArray>` The `<TypedArray>` to copy.
- `offset` `<integer>` The starting offset within `view`. **Default::** `0`.
- `length` `<integer>` The number of elements from `view` to copy. **Default:** `view.length - offset`.

Copies the underlying memory of `view` into a new `Buffer`.

```
const u16 = new Uint16Array([0, 0xffff]);
const buf = Buffer.copyBytesFrom(u16, 1, 1);
u16[1] = 0;
console.log(buf.length); // 2
console.log(buf[0]); // 255
console.log(buf[1]); // 255
```

COPY

Static method: 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.

```
import { Buffer } from 'node:buffer';

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

```
const { Buffer } = require('node:buffer');

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

COPY

If `array` is an `Array`-like object (that is, one with a `length` property of type `number`), it is treated as if it is an array, unless it is a `Buffer` or a `Uint8Array`. This means all other `TypedArray` variants get treated as an `Array`. To create a `Buffer` from the bytes backing a `TypedArray`, use `Buffer.copyBytesFrom()`.

A `TypeError` will be thrown if `array` is not an `Array` or another type appropriate for `Buffer.from()` variants.

`Buffer.from(array)` and `Buffer.from(string)` may also use the internal `Buffer` pool like `Buffer.allocUnsafe()` does.

Static method: Buffer.from(arrayBuffer[, byteOffset[, length]])

- `arrayBuffer` `<ArrayBuffer>` | `<SharedArrayBuffer>` An `ArrayBuffer`, `SharedArrayBuffer`, for example the `.buffer` property of a `TypedArray`.
- `byteOffset` `<integer>` Index of first byte to expose. **Default:** `0`.
- `length` `<integer>` Number of bytes to expose. **Default:** `arrayBuffer.byteLength - byteOffset`.

This creates a view of the `ArrayBuffer` without copying the underlying memory. For example, when passed a reference to the `.buffer` property of a `TypedArray` instance, the newly created `Buffer` will share the same allocated memory as the `TypedArray`'s underlying `ArrayBuffer`.

```mjs
import { Buffer } from 'node:buffer';

const arr = new Uint16Array(2);

arr[0] = 5000;
arr[1] = 4000;

// Shares memory with `arr`.
const buf = Buffer.from(arr.buffer);

console.log(buf);
// Prints: <Buffer 88 13 a0 0f>

// Changing the original Uint16Array changes the Buffer also.
arr[1] = 6000;

console.log(buf);
// Prints: <Buffer 88 13 70 17>
```

```cjs
const { Buffer } = require('node:buffer');

const arr = new Uint16Array(2);

arr[0] = 5000;
arr[1] = 4000;

// Shares memory with `arr`.
const buf = Buffer.from(arr.buffer);

console.log(buf);
// Prints: <Buffer 88 13 a0 0f>

// Changing the original Uint16Array changes the Buffer also.
arr[1] = 6000;

console.log(buf);
// Prints: <Buffer 88 13 70 17>
```

COPY

The optional `byteOffset` and `length` arguments specify a memory range within the `arrayBuffer` that will be shared by the `Buffer`.

```mjs
import { Buffer } from 'node:buffer';

const ab = new ArrayBuffer(10);
const buf = Buffer.from(ab, 0, 2);

console.log(buf.length);
// Prints: 2
```

```cjs
const { Buffer } = require('node:buffer');

const ab = new ArrayBuffer(10);
const buf = Buffer.from(ab, 0, 2);

console.log(buf.length);
// Prints: 2
```

COPY

A `TypeError` will be thrown if `arrayBuffer` is not an `ArrayBuffer` or a `SharedArrayBuffer` or another type appropriate for `Buffer.from()` variants.

It is important to remember that a backing `ArrayBuffer` can cover a range of memory that extends beyond the bounds of a `TypedArray` view. A new `Buffer` created using the `buffer` property of a `TypedArray` may extend beyond the range of the `TypedArray`:

```mjs
import { Buffer } from 'node:buffer';

const arrA = Uint8Array.from([0x63, 0x64, 0x65, 0x66]); // 4 elements
const arrB = new Uint8Array(arrA.buffer, 1, 2); // 2 elements
console.log(arrA.buffer === arrB.buffer); // true

const buf = Buffer.from(arrB.buffer);
console.log(buf);
// Prints: <Buffer 63 64 65 66>
```

```cjs
const { Buffer } = require('node:buffer');

const arrA = Uint8Array.from([0x63, 0x64, 0x65, 0x66]); // 4 elements
const arrB = new Uint8Array(arrA.buffer, 1, 2); // 2 elements
console.log(arrA.buffer === arrB.buffer); // true

const buf = Buffer.from(arrB.buffer);
console.log(buf);
// Prints: <Buffer 63 64 65 66>
```

COPY

Static method: Buffer.from(buffer)

- `buffer` `<Buffer>` | `<Uint8Array>` An existing `Buffer` or `Uint8Array` from which to copy data.

Copies the passed `buffer` data onto a new `Buffer` instance.

```mjs
import { Buffer } from 'node:buffer';

const buf1 = Buffer.from('buffer');
const buf2 = Buffer.from(buf1);

buf1[0] = 0x61;

console.log(buf1.toString());
// Prints: auffer
console.log(buf2.toString());
// Prints: buffer
```

```cjs
const { Buffer } = require('node:buffer');

const buf1 = Buffer.from('buffer');
const buf2 = Buffer.from(buf1);

buf1[0] = 0x61;

console.log(buf1.toString());
// Prints: auffer
console.log(buf2.toString());
// Prints: buffer
```

COPY

A `TypeError` will be thrown if `buffer` is not a `Buffer` or another type appropriate for `Buffer.from()` variants.

Static method: Buffer.from(object[, offsetOrEncoding[, length]])

- `object` <Object> An object supporting `Symbol.toPrimitive` or `valueOf()`.
- `offsetOrEncoding` <integer> | <string> A byte-offset or encoding.
- `length` <integer> A length.

For objects whose `valueOf()` function returns a value not strictly equal to `object`, returns `Buffer.from(object.valueOf(), offsetOrEncoding, length)`.

```mjs
import { Buffer } from 'node:buffer';

const buf = Buffer.from(new String('this is a test'));
// Prints: <Buffer 74 68 69 73 20 69 73 20 61 20 74 65 73 74>
```

```cjs
const { Buffer } = require('node:buffer');

const buf = Buffer.from(new String('this is a test'));
// Prints: <Buffer 74 68 69 73 20 69 73 20 61 20 74 65 73 74>
```

COPY

For objects that support `Symbol.toPrimitive`, returns `Buffer.from(object[Symbol.toPrimitive]('string'), offsetOrEncoding)`.

```mjs
import { Buffer } from 'node:buffer';

class Foo {
  [Symbol.toPrimitive]() {
    return 'this is a test';
  }
}

const buf = Buffer.from(new Foo(), 'utf8');
// Prints: <Buffer 74 68 69 73 20 69 73 20 61 20 74 65 73 74>
```

```cjs
const { Buffer } = require('node:buffer');

class Foo {
  [Symbol.toPrimitive]() {
    return 'this is a test';
  }
}

const buf = Buffer.from(new Foo(), 'utf8');
// Prints: <Buffer 74 68 69 73 20 69 73 20 61 20 74 65 73 74>
```

COPY

A `TypeError` will be thrown if `object` does not have the mentioned methods or is not of another type appropriate for `Buffer.from()` variants.

Static method: Buffer.from(string[, encoding])

- `string` `<string>` A string to encode.
- `encoding` `<string>` The encoding of `string`. **Default:** `'utf8'`.

Creates a new `Buffer` containing `string`. The `encoding` parameter identifies the character encoding to be used when converting `string` into bytes.

```mjs
import { Buffer } from 'node:buffer';

const buf1 = Buffer.from('this is a tést');
const buf2 = Buffer.from('7468697320697320612074c3a97374', 'hex');

console.log(buf1.toString());
// Prints: this is a tést
console.log(buf2.toString());
// Prints: this is a tést
console.log(buf1.toString('latin1'));
// Prints: this is a tÃ©st
```

```cjs
const { Buffer } = require('node:buffer');

const buf1 = Buffer.from('this is a tést');
const buf2 = Buffer.from('7468697320697320612074c3a97374', 'hex');
```

```
console.log(buf1.toString());
// Prints: this is a tést
console.log(buf2.toString());
// Prints: this is a tést
console.log(buf1.toString('latin1'));
// Prints: this is a tÃ©st
```

COPY

A `TypeError` will be thrown if `string` is not a string or another type appropriate for `Buffer.from()` variants.

`Buffer.from(string)` may also use the internal `Buffer` pool like `Buffer.allocUnsafe()` does.

Static method: Buffer.isBuffer(obj)

- `obj` <Object>
- Returns: <boolean>

Returns `true` if `obj` is a `Buffer`, `false` otherwise.

```
import { Buffer } from 'node:buffer';

Buffer.isBuffer(Buffer.alloc(10)); // true
Buffer.isBuffer(Buffer.from('foo')); // true
Buffer.isBuffer('a string'); // false
Buffer.isBuffer([]); // false
Buffer.isBuffer(new Uint8Array(1024)); // false
```

```
const { Buffer } = require('node:buffer');

Buffer.isBuffer(Buffer.alloc(10)); // true
Buffer.isBuffer(Buffer.from('foo')); // true
Buffer.isBuffer('a string'); // false
Buffer.isBuffer([]); // false
Buffer.isBuffer(new Uint8Array(1024)); // false
```

COPY

Static method: Buffer.isEncoding(encoding)

- `encoding` <string> A character encoding name to check.
- Returns: <boolean>

Returns `true` if `encoding` is the name of a supported character encoding, or `false` otherwise.

```
import { Buffer } from 'node:buffer';

console.log(Buffer.isEncoding('utf8'));
// Prints: true

console.log(Buffer.isEncoding('hex'));
// Prints: true
```

```
console.log(Buffer.isEncoding('utf/8'));
// Prints: false

console.log(Buffer.isEncoding(''));
// Prints: false
```

```
const { Buffer } = require('node:buffer');

console.log(Buffer.isEncoding('utf8'));
// Prints: true

console.log(Buffer.isEncoding('hex'));
// Prints: true

console.log(Buffer.isEncoding('utf/8'));
// Prints: false

console.log(Buffer.isEncoding(''));
// Prints: false
```

COPY

Class property: Buffer.poolSize

- <integer> **Default:** `8192`

This is the size (in bytes) of pre-allocated internal `Buffer` instances used for pooling. This value may be modified.

buf[index]

- `index` <integer>

The index operator `[index]` can be used to get and set the octet at position `index` in `buf`. The values refer to individual bytes, so the legal value range is between `0x00` and `0xFF` (hex) or `0` and `255` (decimal).

This operator is inherited from `Uint8Array`, so its behavior on out-of-bounds access is the same as `Uint8Array`. In other words, `buf[index]` returns `undefined` when `index` is negative or greater or equal to `buf.length`, and `buf[index] = value` does not modify the buffer if `index` is negative or `>= buf.length`.

```
import { Buffer } from 'node:buffer';

// Copy an ASCII string into a `Buffer` one byte at a time.
// (This only works for ASCII-only strings. In general, one should use
// `Buffer.from()` to perform this conversion.)

const str = 'Node.js';
const buf = Buffer.allocUnsafe(str.length);

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

```
console.log(buf.toString('utf8'));
// Prints: Node.js
```

```
const { Buffer } = require('node:buffer');

// Copy an ASCII string into a `Buffer` one byte at a time.
// (This only works for ASCII-only strings. In general, one should use
// `Buffer.from()` to perform this conversion.)

const str = 'Node.js';
const buf = Buffer.allocUnsafe(str.length);

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

console.log(buf.toString('utf8'));
// Prints: Node.js
```

COPY

buf.buffer

- <ArrayBuffer> The underlying `ArrayBuffer` object based on which this `Buffer` object is created.

This `ArrayBuffer` is not guaranteed to correspond exactly to the original `Buffer`. See the notes on `buf.byteOffset` for details.

```
import { Buffer } from 'node:buffer';

const arrayBuffer = new ArrayBuffer(16);
const buffer = Buffer.from(arrayBuffer);

console.log(buffer.buffer === arrayBuffer);
// Prints: true
```

```
const { Buffer } = require('node:buffer');

const arrayBuffer = new ArrayBuffer(16);
const buffer = Buffer.from(arrayBuffer);

console.log(buffer.buffer === arrayBuffer);
// Prints: true
```

COPY

buf.byteOffset

- <integer> The `byteOffset` of the `Buffer`s underlying `ArrayBuffer` object.

When setting `byteOffset` in `Buffer.from(ArrayBuffer, byteOffset, length)`, or sometimes when allocating a `Buffer` smaller than `Buffer.poolSize`, the buffer does not start from a zero offset on the underlying `ArrayBuffer`.

This can cause problems when accessing the underlying `ArrayBuffer` directly using `buf.buffer`, as other parts of the `ArrayBuffer` may be unrelated to the `Buffer` object itself.

A common issue when creating a `TypedArray` object that shares its memory with a `Buffer` is that in this case one needs to specify the `byteOffset` correctly:

```mjs
import { Buffer } from 'node:buffer';

// Create a buffer smaller than `Buffer.poolSize`.
const nodeBuffer = Buffer.from([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]);

// When casting the Node.js Buffer to an Int8Array, use the byteOffset
// to refer only to the part of `nodeBuffer.buffer` that contains the memory
// for `nodeBuffer`.
new Int8Array(nodeBuffer.buffer, nodeBuffer.byteOffset, nodeBuffer.length);
```

```cjs
const { Buffer } = require('node:buffer');

// Create a buffer smaller than `Buffer.poolSize`.
const nodeBuffer = Buffer.from([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]);

// When casting the Node.js Buffer to an Int8Array, use the byteOffset
// to refer only to the part of `nodeBuffer.buffer` that contains the memory
// for `nodeBuffer`.
new Int8Array(nodeBuffer.buffer, nodeBuffer.byteOffset, nodeBuffer.length);
```

COPY

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.

```mjs
import { Buffer } from 'node:buffer';

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].)
```

```
const { Buffer } = require('node:buffer');

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].)
```

COPY

The optional `targetStart`, `targetEnd`, `sourceStart`, and `sourceEnd` arguments can be used to limit the comparison to specific ranges within `target` and `buf` respectively.

```
import { Buffer } from 'node:buffer';

const buf1 = Buffer.from([1, 2, 3, 4, 5, 6, 7, 8, 9]);
const buf2 = Buffer.from([5, 6, 7, 8, 9, 1, 2, 3, 4]);

console.log(buf1.compare(buf2, 5, 9, 0, 4));
// Prints: 0
console.log(buf1.compare(buf2, 0, 6, 4));
// Prints: -1
console.log(buf1.compare(buf2, 5, 6, 5));
// Prints: 1
```

```
const { Buffer } = require('node:buffer');

const buf1 = Buffer.from([1, 2, 3, 4, 5, 6, 7, 8, 9]);
const buf2 = Buffer.from([5, 6, 7, 8, 9, 1, 2, 3, 4]);

console.log(buf1.compare(buf2, 5, 9, 0, 4));
// Prints: 0
console.log(buf1.compare(buf2, 0, 6, 4));
// Prints: -1
console.log(buf1.compare(buf2, 5, 6, 5));
// Prints: 1
```

COPY

ERR_OUT_OF_RANGE is thrown if `targetStart < 0`, `sourceStart < 0`, `targetEnd > target.byteLength`, or `sourceEnd > source.byteLength`.

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`.

TypedArray.prototype.set() performs the same operation, and is available for all TypedArrays, including Node.js `Buffer`s, although it takes different function arguments.

```
import { Buffer } from 'node:buffer';

// Create two `Buffer` instances.
const buf1 = Buffer.allocUnsafe(26);
const buf2 = Buffer.allocUnsafe(26).fill('!');

for (let i = 0; i < 26; i++) {
  // 97 is the decimal ASCII value for 'a'.
  buf1[i] = i + 97;
}

// Copy `buf1` bytes 16 through 19 into `buf2` starting at byte 8 of `buf2`.
buf1.copy(buf2, 8, 16, 20);
// This is equivalent to:
// buf2.set(buf1.subarray(16, 20), 8);

console.log(buf2.toString('ascii', 0, 25));
// Prints: !!!!!!!!qrst!!!!!!!!!!!!!
```

```
const { Buffer } = require('node:buffer');

// Create two `Buffer` instances.
```

```
const buf1 = Buffer.allocUnsafe(26);
const buf2 = Buffer.allocUnsafe(26).fill('!');

for (let i = 0; i < 26; i++) {
  // 97 is the decimal ASCII value for 'a'.
  buf1[i] = i + 97;
}

// Copy `buf1` bytes 16 through 19 into `buf2` starting at byte 8 of `buf2`.
buf1.copy(buf2, 8, 16, 20);
// This is equivalent to:
// buf2.set(buf1.subarray(16, 20), 8);

console.log(buf2.toString('ascii', 0, 25));
// Prints: !!!!!!!!qrst!!!!!!!!!!!!!
```

COPY

```
import { Buffer } from 'node:buffer';

// Create a `Buffer` and copy data from one region to an overlapping region
// within the same `Buffer`.

const buf = Buffer.allocUnsafe(26);

for (let i = 0; i < 26; i++) {
  // 97 is the decimal ASCII value for 'a'.
  buf[i] = i + 97;
}

buf.copy(buf, 0, 4, 10);

console.log(buf.toString());
// Prints: efghijghijklmnopqrstuvwxyz
```

```
const { Buffer } = require('node:buffer');

// Create a `Buffer` and copy data from one region to an overlapping region
// within the same `Buffer`.

const buf = Buffer.allocUnsafe(26);

for (let i = 0; i < 26; i++) {
  // 97 is the decimal ASCII value for 'a'.
  buf[i] = i + 97;
}

buf.copy(buf, 0, 4, 10);

console.log(buf.toString());
// Prints: efghijghijklmnopqrstuvwxyz
```

COPY

buf.entries()

- Returns: <Iterator>

Creates and returns an iterator of `[index, byte]` pairs from the contents of `buf`.

```mjs
import { Buffer } from 'node:buffer';

// Log the entire contents of a `Buffer`.

const buf = Buffer.from('buffer');

for (const pair of buf.entries()) {
  console.log(pair);
}
// Prints:
//   [0, 98]
//   [1, 117]
//   [2, 102]
//   [3, 102]
//   [4, 101]
//   [5, 114]
```

```cjs
const { Buffer } = require('node:buffer');

// Log the entire contents of a `Buffer`.

const buf = Buffer.from('buffer');

for (const pair of buf.entries()) {
  console.log(pair);
}
// Prints:
//   [0, 98]
//   [1, 117]
//   [2, 102]
//   [3, 102]
//   [4, 101]
//   [5, 114]
```

COPY

buf.equals(otherBuffer)

- `otherBuffer` <Buffer> | <Uint8Array> A `Buffer` or Uint8Array with which to compare `buf`.
- Returns: <boolean>

Returns `true` if both `buf` and `otherBuffer` have exactly the same bytes, `false` otherwise. Equivalent to `buf.compare(otherBuffer) === 0`.

```mjs
import { Buffer } from 'node:buffer';

const buf1 = Buffer.from('ABC');
const buf2 = Buffer.from('414243', 'hex');
```

```
const buf3 = Buffer.from('ABCD');

console.log(buf1.equals(buf2));
// Prints: true
console.log(buf1.equals(buf3));
// Prints: false
```

```
const { Buffer } = require('node:buffer');

const buf1 = Buffer.from('ABC');
const buf2 = Buffer.from('414243', 'hex');
const buf3 = Buffer.from('ABCD');

console.log(buf1.equals(buf2));
// Prints: true
console.log(buf1.equals(buf3));
// Prints: false
```

COPY

buf.fill(value[, offset[, end]][, encoding])

- `value` <string> | <Buffer> | <Uint8Array> | <integer> The value with which to fill `buf`. Empty value (string, Uint8Array, Buffer) is coerced to `0`.
- `offset` <integer> Number of bytes to skip before starting to fill `buf`. **Default:** `0`.
- `end` <integer> Where to stop filling `buf` (not inclusive). **Default:** `buf.length`.
- `encoding` <string> The encoding for `value` if `value` is a string. **Default:** `'utf8'`.
- Returns: <Buffer> A reference to `buf`.

Fills `buf` with the specified `value`. If the `offset` and `end` are not given, the entire `buf` will be filled:

```
import { Buffer } from 'node:buffer';

// Fill a `Buffer` with the ASCII character 'h'.

const b = Buffer.allocUnsafe(50).fill('h');

console.log(b.toString());
// Prints: hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh

// Fill a buffer with empty string
const c = Buffer.allocUnsafe(5).fill('');

console.log(c.fill(''));
// Prints: <Buffer 00 00 00 00 00>
```

```
const { Buffer } = require('node:buffer');

// Fill a `Buffer` with the ASCII character 'h'.
```

```
const b = Buffer.allocUnsafe(50).fill('h');

console.log(b.toString());
// Prints: hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh

// Fill a buffer with empty string
const c = Buffer.allocUnsafe(5).fill('');

console.log(c.fill(''));
// Prints: <Buffer 00 00 00 00 00>
```

COPY

`value` is coerced to a `uint32` value if it is not a string, `Buffer`, or integer. If the resulting integer is greater than `255` (decimal), `buf` will be filled with `value & 255`.

If the final write of a `fill()` operation falls on a multi-byte character, then only the bytes of that character that fit into `buf` are written:

```
import { Buffer } from 'node:buffer';

// Fill a `Buffer` with character that takes up two bytes in UTF-8.

console.log(Buffer.allocUnsafe(5).fill('\u0222'));
// Prints: <Buffer c8 a2 c8 a2 c8>
```

```
const { Buffer } = require('node:buffer');

// Fill a `Buffer` with character that takes up two bytes in UTF-8.

console.log(Buffer.allocUnsafe(5).fill('\u0222'));
// Prints: <Buffer c8 a2 c8 a2 c8>
```

COPY

If `value` contains invalid characters, it is truncated; if no valid fill data remains, an exception is thrown:

```
import { Buffer } from 'node:buffer';

const buf = Buffer.allocUnsafe(5);

console.log(buf.fill('a'));
// Prints: <Buffer 61 61 61 61 61>
console.log(buf.fill('aazz', 'hex'));
// Prints: <Buffer aa aa aa aa aa>
console.log(buf.fill('zz', 'hex'));
// Throws an exception.
```

```
const { Buffer } = require('node:buffer');

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

```
console.log(buf.fill('a'));
// Prints: <Buffer 61 61 61 61 61>
console.log(buf.fill('aazz', 'hex'));
// Prints: <Buffer aa aa aa aa aa>
console.log(buf.fill('zz', 'hex'));
// Throws an exception.
```

COPY

buf.includes(value[, byteOffset][, encoding])

- `value` <string> | <Buffer> | <Uint8Array> | <integer> What to search for.
- `byteOffset` <integer> Where to begin searching in `buf`. If negative, then offset is calculated from the end of `buf`. **Default:** `0`.
- `encoding` <string> If `value` is a string, this is its encoding. **Default:** `'utf8'`.
- Returns: <boolean> `true` if `value` was found in `buf`, `false` otherwise.

Equivalent to `buf.indexOf() !== -1`.

```
import { Buffer } from 'node:buffer';

const buf = Buffer.from('this is a buffer');

console.log(buf.includes('this'));
// Prints: true
console.log(buf.includes('is'));
// Prints: true
console.log(buf.includes(Buffer.from('a buffer')));
// Prints: true
console.log(buf.includes(97));
// Prints: true (97 is the decimal ASCII value for 'a')
console.log(buf.includes(Buffer.from('a buffer example')));
// Prints: false
console.log(buf.includes(Buffer.from('a buffer example').slice(0, 8)));
// Prints: true
console.log(buf.includes('this', 4));
// Prints: false
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.from('this is a buffer');

console.log(buf.includes('this'));
// Prints: true
console.log(buf.includes('is'));
// Prints: true
console.log(buf.includes(Buffer.from('a buffer')));
// Prints: true
console.log(buf.includes(97));
// Prints: true (97 is the decimal ASCII value for 'a')
console.log(buf.includes(Buffer.from('a buffer example')));
// Prints: false
console.log(buf.includes(Buffer.from('a buffer example').slice(0, 8)));
```

```
// Prints: true
console.log(buf.includes('this', 4));
// Prints: false
```

COPY

buf.indexOf(value[, byteOffset][, encoding])

- `value` <string> | <Buffer> | <Uint8Array> | <integer> What to search for.
- `byteOffset` <integer> Where to begin searching in `buf`. If negative, then offset is calculated from the end of `buf`. **Default:** `0`.
- `encoding` <string> If `value` is a string, this is the encoding used to determine the binary representation of the string that will be searched for in `buf`. **Default:** `'utf8'`.
- Returns: <integer> The index of the first occurrence of `value` in `buf`, or `-1` if `buf` does not contain `value`.

If `value` is:

- a string, `value` is interpreted according to the character encoding in `encoding`.
- a `Buffer` or Uint8Array, `value` will be used in its entirety. To compare a partial `Buffer`, use buf.subarray.
- a number, `value` will be interpreted as an unsigned 8-bit integer value between `0` and `255`.

```
import { Buffer } from 'node:buffer';

const buf = Buffer.from('this is a buffer');

console.log(buf.indexOf('this'));
// Prints: 0
console.log(buf.indexOf('is'));
// Prints: 2
console.log(buf.indexOf(Buffer.from('a buffer')));
// Prints: 8
console.log(buf.indexOf(97));
// Prints: 8 (97 is the decimal ASCII value for 'a')
console.log(buf.indexOf(Buffer.from('a buffer example')));
// Prints: -1
console.log(buf.indexOf(Buffer.from('a buffer example').slice(0, 8)));
// Prints: 8

const utf16Buffer = Buffer.from('\u039a\u0391\u03a3\u03a3\u0395', 'utf16le');

console.log(utf16Buffer.indexOf('\u03a3', 0, 'utf16le'));
// Prints: 4
console.log(utf16Buffer.indexOf('\u03a3', -4, 'utf16le'));
// Prints: 6
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.from('this is a buffer');

console.log(buf.indexOf('this'));
// Prints: 0
console.log(buf.indexOf('is'));
// Prints: 2
```

```
console.log(buf.indexOf(Buffer.from('a buffer')));
// Prints: 8
console.log(buf.indexOf(97));
// Prints: 8 (97 is the decimal ASCII value for 'a')
console.log(buf.indexOf(Buffer.from('a buffer example')));
// Prints: -1
console.log(buf.indexOf(Buffer.from('a buffer example').slice(0, 8)));
// Prints: 8

const utf16Buffer = Buffer.from('\u039a\u0391\u03a3\u03a3\u0395', 'utf16le');

console.log(utf16Buffer.indexOf('\u03a3', 0, 'utf16le'));
// Prints: 4
console.log(utf16Buffer.indexOf('\u03a3', -4, 'utf16le'));
// Prints: 6
```

COPY

If `value` is not a string, number, or `Buffer`, this method will throw a `TypeError`. If `value` is a number, it will be coerced to a valid byte value, an integer between 0 and 255.

If `byteOffset` is not a number, it will be coerced to a number. If the result of coercion is `NaN` or `0`, then the entire buffer will be searched. This behavior matches `String.prototype.indexOf()`.

```
import { Buffer } from 'node:buffer';

const b = Buffer.from('abcdef');

// Passing a value that's a number, but not a valid byte.
// Prints: 2, equivalent to searching for 99 or 'c'.
console.log(b.indexOf(99.9));
console.log(b.indexOf(256 + 99));

// Passing a byteOffset that coerces to NaN or 0.
// Prints: 1, searching the whole buffer.
console.log(b.indexOf('b', undefined));
console.log(b.indexOf('b', {}));
console.log(b.indexOf('b', null));
console.log(b.indexOf('b', []));
```

```
const { Buffer } = require('node:buffer');

const b = Buffer.from('abcdef');

// Passing a value that's a number, but not a valid byte.
// Prints: 2, equivalent to searching for 99 or 'c'.
console.log(b.indexOf(99.9));
console.log(b.indexOf(256 + 99));

// Passing a byteOffset that coerces to NaN or 0.
// Prints: 1, searching the whole buffer.
console.log(b.indexOf('b', undefined));
console.log(b.indexOf('b', {}));
```

```
console.log(b.indexOf('b', null));
console.log(b.indexOf('b', []));
```

COPY

If `value` is an empty string or empty `Buffer` and `byteOffset` is less than `buf.length`, `byteOffset` will be returned. If `value` is empty and `byteOffset` is at least `buf.length`, `buf.length` will be returned.

buf.keys()

- Returns: `<Iterator>`

Creates and returns an iterator of `buf` keys (indexes).

```
import { Buffer } from 'node:buffer';

const buf = Buffer.from('buffer');

for (const key of buf.keys()) {
  console.log(key);
}
// Prints:
//   0
//   1
//   2
//   3
//   4
//   5
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.from('buffer');

for (const key of buf.keys()) {
  console.log(key);
}
// Prints:
//   0
//   1
//   2
//   3
//   4
//   5
```

COPY

buf.lastIndexOf(value[, byteOffset][, encoding])

- `value` `<string>` | `<Buffer>` | `<Uint8Array>` | `<integer>` What to search for.
- `byteOffset` `<integer>` Where to begin searching in `buf`. If negative, then offset is calculated from the end of `buf`. **Default:** `buf.length - 1`.
- `encoding` `<string>` If `value` is a string, this is the encoding used to determine the binary representation of the string that will be searched for in `buf`. **Default:** `'utf8'`.

- Returns: <integer> The index of the last occurrence of `value` in `buf`, or `-1` if `buf` does not contain `value`.

Identical to `buf.indexOf()`, except the last occurrence of `value` is found rather than the first occurrence.

```mjs
import { Buffer } from 'node:buffer';

const buf = Buffer.from('this buffer is a buffer');

console.log(buf.lastIndexOf('this'));
// Prints: 0
console.log(buf.lastIndexOf('buffer'));
// Prints: 17
console.log(buf.lastIndexOf(Buffer.from('buffer')));
// Prints: 17
console.log(buf.lastIndexOf(97));
// Prints: 15 (97 is the decimal ASCII value for 'a')
console.log(buf.lastIndexOf(Buffer.from('yolo')));
// Prints: -1
console.log(buf.lastIndexOf('buffer', 5));
// Prints: 5
console.log(buf.lastIndexOf('buffer', 4));
// Prints: -1

const utf16Buffer = Buffer.from('\u039a\u0391\u03a3\u03a3\u0395', 'utf16le');

console.log(utf16Buffer.lastIndexOf('\u03a3', undefined, 'utf16le'));
// Prints: 6
console.log(utf16Buffer.lastIndexOf('\u03a3', -5, 'utf16le'));
// Prints: 4
```

```cjs
const { Buffer } = require('node:buffer');

const buf = Buffer.from('this buffer is a buffer');

console.log(buf.lastIndexOf('this'));
// Prints: 0
console.log(buf.lastIndexOf('buffer'));
// Prints: 17
console.log(buf.lastIndexOf(Buffer.from('buffer')));
// Prints: 17
console.log(buf.lastIndexOf(97));
// Prints: 15 (97 is the decimal ASCII value for 'a')
console.log(buf.lastIndexOf(Buffer.from('yolo')));
// Prints: -1
console.log(buf.lastIndexOf('buffer', 5));
// Prints: 5
console.log(buf.lastIndexOf('buffer', 4));
// Prints: -1

const utf16Buffer = Buffer.from('\u039a\u0391\u03a3\u03a3\u0395', 'utf16le');

console.log(utf16Buffer.lastIndexOf('\u03a3', undefined, 'utf16le'));
```

```
// Prints: 6
console.log(utf16Buffer.lastIndexOf('\u03a3', -5, 'utf16le'));
// Prints: 4
```

COPY

If `value` is not a string, number, or `Buffer`, this method will throw a `TypeError`. If `value` is a number, it will be coerced to a valid byte value, an integer between 0 and 255.

If `byteOffset` is not a number, it will be coerced to a number. Any arguments that coerce to `NaN`, like `{}` or `undefined`, will search the whole buffer. This behavior matches `String.prototype.lastIndexOf()`.

```
import { Buffer } from 'node:buffer';

const b = Buffer.from('abcdef');

// Passing a value that's a number, but not a valid byte.
// Prints: 2, equivalent to searching for 99 or 'c'.
console.log(b.lastIndexOf(99.9));
console.log(b.lastIndexOf(256 + 99));

// Passing a byteOffset that coerces to NaN.
// Prints: 1, searching the whole buffer.
console.log(b.lastIndexOf('b', undefined));
console.log(b.lastIndexOf('b', {}));

// Passing a byteOffset that coerces to 0.
// Prints: -1, equivalent to passing 0.
console.log(b.lastIndexOf('b', null));
console.log(b.lastIndexOf('b', []));
```

```
const { Buffer } = require('node:buffer');

const b = Buffer.from('abcdef');

// Passing a value that's a number, but not a valid byte.
// Prints: 2, equivalent to searching for 99 or 'c'.
console.log(b.lastIndexOf(99.9));
console.log(b.lastIndexOf(256 + 99));

// Passing a byteOffset that coerces to NaN.
// Prints: 1, searching the whole buffer.
console.log(b.lastIndexOf('b', undefined));
console.log(b.lastIndexOf('b', {}));

// Passing a byteOffset that coerces to 0.
// Prints: -1, equivalent to passing 0.
console.log(b.lastIndexOf('b', null));
console.log(b.lastIndexOf('b', []));
```

COPY

If `value` is an empty string or empty `Buffer`, `byteOffset` will be returned.

buf.length

- <integer>

Returns the number of bytes in `buf`.

```
import { Buffer } from 'node:buffer';

// Create a `Buffer` and write a shorter string to it using UTF-8.

const buf = Buffer.alloc(1234);

console.log(buf.length);
// Prints: 1234

buf.write('some string', 0, 'utf8');

console.log(buf.length);
// Prints: 1234
```

```
const { Buffer } = require('node:buffer');

// Create a `Buffer` and write a shorter string to it using UTF-8.

const buf = Buffer.alloc(1234);

console.log(buf.length);
// Prints: 1234

buf.write('some string', 0, 'utf8');

console.log(buf.length);
// Prints: 1234
```

COPY

buf.parent

Stability: 0 - Deprecated: Use `buf.buffer` instead.

The `buf.parent` property is a deprecated alias for `buf.buffer`.

buf.readBigInt64BE([offset])

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy: `0 <= offset <= buf.length - 8`. **Default:** `0`.
- Returns: <bigint>

Reads a signed, big-endian 64-bit integer from `buf` at the specified `offset`.

Integers read from a `Buffer` are interpreted as two's complement signed values.

buf.readBigInt64LE([offset])

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy: `0 <= offset <= buf.length - 8`. **Default:** `0`.
- Returns: <bigint>

Reads a signed, little-endian 64-bit integer from `buf` at the specified `offset`.

Integers read from a `Buffer` are interpreted as two's complement signed values.

buf.readBigUInt64BE([offset])

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy: `0 <= offset <= buf.length - 8`. **Default:** `0`.
- Returns: <bigint>

Reads an unsigned, big-endian 64-bit integer from `buf` at the specified `offset`.

This function is also available under the `readBigUint64BE` alias.

```mjs
import { Buffer } from 'node:buffer';

const buf = Buffer.from([0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff]);

console.log(buf.readBigUInt64BE(0));
// Prints: 4294967295n
```

```cjs
const { Buffer } = require('node:buffer');

const buf = Buffer.from([0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff]);

console.log(buf.readBigUInt64BE(0));
// Prints: 4294967295n
```

COPY

buf.readBigUInt64LE([offset])

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy: `0 <= offset <= buf.length - 8`. **Default:** `0`.
- Returns: <bigint>

Reads an unsigned, little-endian 64-bit integer from `buf` at the specified `offset`.

This function is also available under the `readBigUint64LE` alias.

```mjs
import { Buffer } from 'node:buffer';

const buf = Buffer.from([0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff]);

console.log(buf.readBigUInt64LE(0));
// Prints: 18446744069414584320n
```

```cjs
const { Buffer } = require('node:buffer');

const buf = Buffer.from([0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff]);
```

```
console.log(buf.readBigUInt64LE(0));
// Prints: 18446744069414584320n
```

COPY

buf.readDoubleBE([offset])

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 8`. **Default:** `0`.
- Returns: <number>

Reads a 64-bit, big-endian double from `buf` at the specified `offset`.

```
import { Buffer } from 'node:buffer';

const buf = Buffer.from([1, 2, 3, 4, 5, 6, 7, 8]);

console.log(buf.readDoubleBE(0));
// Prints: 8.20788039913184e-304
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.from([1, 2, 3, 4, 5, 6, 7, 8]);

console.log(buf.readDoubleBE(0));
// Prints: 8.20788039913184e-304
```

COPY

buf.readDoubleLE([offset])

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 8`. **Default:** `0`.
- Returns: <number>

Reads a 64-bit, little-endian double from `buf` at the specified `offset`.

```
import { Buffer } from 'node:buffer';

const buf = Buffer.from([1, 2, 3, 4, 5, 6, 7, 8]);

console.log(buf.readDoubleLE(0));
// Prints: 5.447603722011605e-270
console.log(buf.readDoubleLE(1));
// Throws ERR_OUT_OF_RANGE.
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.from([1, 2, 3, 4, 5, 6, 7, 8]);

console.log(buf.readDoubleLE(0));
// Prints: 5.447603722011605e-270
```

```
console.log(buf.readDoubleLE(1));
// Throws ERR_OUT_OF_RANGE.
```

COPY

buf.readFloatBE([offset])

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 4`. **Default:** `0`.
- Returns: <number>

Reads a 32-bit, big-endian float from `buf` at the specified `offset`.

```
import { Buffer } from 'node:buffer';

const buf = Buffer.from([1, 2, 3, 4]);

console.log(buf.readFloatBE(0));
// Prints: 2.387939260590663e-38
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.from([1, 2, 3, 4]);

console.log(buf.readFloatBE(0));
// Prints: 2.387939260590663e-38
```

COPY

buf.readFloatLE([offset])

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 4`. **Default:** `0`.
- Returns: <number>

Reads a 32-bit, little-endian float from `buf` at the specified `offset`.

```
import { Buffer } from 'node:buffer';

const buf = Buffer.from([1, 2, 3, 4]);

console.log(buf.readFloatLE(0));
// Prints: 1.539989614439558e-36
console.log(buf.readFloatLE(1));
// Throws ERR_OUT_OF_RANGE.
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.from([1, 2, 3, 4]);

console.log(buf.readFloatLE(0));
// Prints: 1.539989614439558e-36
```

```
console.log(buf.readFloatLE(1));
// Throws ERR_OUT_OF_RANGE.
```

COPY

buf.readInt8([offset])

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 1`. **Default:** `0`.
- Returns: <integer>

Reads a signed 8-bit integer from `buf` at the specified `offset`.

Integers read from a `Buffer` are interpreted as two's complement signed values.

```
import { Buffer } from 'node:buffer';

const buf = Buffer.from([-1, 5]);

console.log(buf.readInt8(0));
// Prints: -1
console.log(buf.readInt8(1));
// Prints: 5
console.log(buf.readInt8(2));
// Throws ERR_OUT_OF_RANGE.
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.from([-1, 5]);

console.log(buf.readInt8(0));
// Prints: -1
console.log(buf.readInt8(1));
// Prints: 5
console.log(buf.readInt8(2));
// Throws ERR_OUT_OF_RANGE.
```

COPY

buf.readInt16BE([offset])

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 2`. **Default:** `0`.
- Returns: <integer>

Reads a signed, big-endian 16-bit integer from `buf` at the specified `offset`.

Integers read from a `Buffer` are interpreted as two's complement signed values.

```
import { Buffer } from 'node:buffer';

const buf = Buffer.from([0, 5]);
```

```
console.log(buf.readInt16BE(0));
// Prints: 5
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.from([0, 5]);

console.log(buf.readInt16BE(0));
// Prints: 5
```

COPY

buf.readInt16LE([offset])

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 2`. **Default:** `0`.
- Returns: <integer>

Reads a signed, little-endian 16-bit integer from `buf` at the specified `offset`.

Integers read from a `Buffer` are interpreted as two's complement signed values.

```
import { Buffer } from 'node:buffer';

const buf = Buffer.from([0, 5]);

console.log(buf.readInt16LE(0));
// Prints: 1280
console.log(buf.readInt16LE(1));
// Throws ERR_OUT_OF_RANGE.
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.from([0, 5]);

console.log(buf.readInt16LE(0));
// Prints: 1280
console.log(buf.readInt16LE(1));
// Throws ERR_OUT_OF_RANGE.
```

COPY

buf.readInt32BE([offset])

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 4`. **Default:** `0`.
- Returns: <integer>

Reads a signed, big-endian 32-bit integer from `buf` at the specified `offset`.

Integers read from a `Buffer` are interpreted as two's complement signed values.

```
import { Buffer } from 'node:buffer';
```

```
const buf = Buffer.from([0, 0, 0, 5]);

console.log(buf.readInt32BE(0));
// Prints: 5
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.from([0, 0, 0, 5]);

console.log(buf.readInt32BE(0));
// Prints: 5
```

COPY

buf.readInt32LE([offset])

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 4`. **Default:** `0`.
- Returns: <integer>

Reads a signed, little-endian 32-bit integer from `buf` at the specified `offset`.

Integers read from a `Buffer` are interpreted as two's complement signed values.

```
import { Buffer } from 'node:buffer';

const buf = Buffer.from([0, 0, 0, 5]);

console.log(buf.readInt32LE(0));
// Prints: 83886080
console.log(buf.readInt32LE(1));
// Throws ERR_OUT_OF_RANGE.
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.from([0, 0, 0, 5]);

console.log(buf.readInt32LE(0));
// Prints: 83886080
console.log(buf.readInt32LE(1));
// Throws ERR_OUT_OF_RANGE.
```

COPY

buf.readIntBE(offset, byteLength)

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - byteLength`.
- `byteLength` <integer> Number of bytes to read. Must satisfy `0 < byteLength <= 6`.
- Returns: <integer>

Reads `byteLength` number of bytes from `buf` at the specified `offset` and interprets the result as a big-endian, two's complement signed value supporting up to 48 bits of accuracy.

```mjs
import { Buffer } from 'node:buffer';

const buf = Buffer.from([0x12, 0x34, 0x56, 0x78, 0x90, 0xab]);

console.log(buf.readIntBE(0, 6).toString(16));
// Prints: 1234567890ab
console.log(buf.readIntBE(1, 6).toString(16));
// Throws ERR_OUT_OF_RANGE.
console.log(buf.readIntBE(1, 0).toString(16));
// Throws ERR_OUT_OF_RANGE.
```

```cjs
const { Buffer } = require('node:buffer');

const buf = Buffer.from([0x12, 0x34, 0x56, 0x78, 0x90, 0xab]);

console.log(buf.readIntBE(0, 6).toString(16));
// Prints: 1234567890ab
console.log(buf.readIntBE(1, 6).toString(16));
// Throws ERR_OUT_OF_RANGE.
console.log(buf.readIntBE(1, 0).toString(16));
// Throws ERR_OUT_OF_RANGE.
```

COPY

buf.readIntLE(offset, byteLength)

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - byteLength`.
- `byteLength` <integer> Number of bytes to read. Must satisfy `0 < byteLength <= 6`.
- Returns: <integer>

Reads `byteLength` number of bytes from `buf` at the specified `offset` and interprets the result as a little-endian, two's complement signed value supporting up to 48 bits of accuracy.

```mjs
import { Buffer } from 'node:buffer';

const buf = Buffer.from([0x12, 0x34, 0x56, 0x78, 0x90, 0xab]);

console.log(buf.readIntLE(0, 6).toString(16));
// Prints: -546f87a9cbee
```

```cjs
const { Buffer } = require('node:buffer');

const buf = Buffer.from([0x12, 0x34, 0x56, 0x78, 0x90, 0xab]);

console.log(buf.readIntLE(0, 6).toString(16));
// Prints: -546f87a9cbee
```

COPY

buf.readUInt8([offset])

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 1`. **Default:** `0`.
- Returns: <integer>

Reads an unsigned 8-bit integer from `buf` at the specified `offset`.

This function is also available under the `readUint8` alias.

```mjs
import { Buffer } from 'node:buffer';

const buf = Buffer.from([1, -2]);

console.log(buf.readUInt8(0));
// Prints: 1
console.log(buf.readUInt8(1));
// Prints: 254
console.log(buf.readUInt8(2));
// Throws ERR_OUT_OF_RANGE.
```

```cjs
const { Buffer } = require('node:buffer');

const buf = Buffer.from([1, -2]);

console.log(buf.readUInt8(0));
// Prints: 1
console.log(buf.readUInt8(1));
// Prints: 254
console.log(buf.readUInt8(2));
// Throws ERR_OUT_OF_RANGE.
```

COPY

buf.readUInt16BE([offset])

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 2`. **Default:** `0`.
- Returns: <integer>

Reads an unsigned, big-endian 16-bit integer from `buf` at the specified `offset`.

This function is also available under the `readUint16BE` alias.

```mjs
import { Buffer } from 'node:buffer';

const buf = Buffer.from([0x12, 0x34, 0x56]);

console.log(buf.readUInt16BE(0).toString(16));
// Prints: 1234
console.log(buf.readUInt16BE(1).toString(16));
// Prints: 3456
```

```cjs
const { Buffer } = require('node:buffer');
```

```
const buf = Buffer.from([0x12, 0x34, 0x56]);

console.log(buf.readUInt16BE(0).toString(16));
// Prints: 1234
console.log(buf.readUInt16BE(1).toString(16));
// Prints: 3456
```

COPY

buf.readUInt16LE([offset])

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 2`. **Default:** `0`.
- Returns: <integer>

Reads an unsigned, little-endian 16-bit integer from `buf` at the specified `offset`.

This function is also available under the `readUint16LE` alias.

```
import { Buffer } from 'node:buffer';

const buf = Buffer.from([0x12, 0x34, 0x56]);

console.log(buf.readUInt16LE(0).toString(16));
// Prints: 3412
console.log(buf.readUInt16LE(1).toString(16));
// Prints: 5634
console.log(buf.readUInt16LE(2).toString(16));
// Throws ERR_OUT_OF_RANGE.
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.from([0x12, 0x34, 0x56]);

console.log(buf.readUInt16LE(0).toString(16));
// Prints: 3412
console.log(buf.readUInt16LE(1).toString(16));
// Prints: 5634
console.log(buf.readUInt16LE(2).toString(16));
// Throws ERR_OUT_OF_RANGE.
```

COPY

buf.readUInt32BE([offset])

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 4`. **Default:** `0`.
- Returns: <integer>

Reads an unsigned, big-endian 32-bit integer from `buf` at the specified `offset`.

This function is also available under the `readUint32BE` alias.

```
import { Buffer } from 'node:buffer';
```

```
const buf = Buffer.from([0x12, 0x34, 0x56, 0x78]);

console.log(buf.readUInt32BE(0).toString(16));
// Prints: 12345678
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.from([0x12, 0x34, 0x56, 0x78]);

console.log(buf.readUInt32BE(0).toString(16));
// Prints: 12345678
```

COPY

buf.readUInt32LE([offset])

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 4`. **Default:** `0`.
- Returns: <integer>

Reads an unsigned, little-endian 32-bit integer from `buf` at the specified `offset`.

This function is also available under the `readUint32LE` alias.

```
import { Buffer } from 'node:buffer';

const buf = Buffer.from([0x12, 0x34, 0x56, 0x78]);

console.log(buf.readUInt32LE(0).toString(16));
// Prints: 78563412
console.log(buf.readUInt32LE(1).toString(16));
// Throws ERR_OUT_OF_RANGE.
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.from([0x12, 0x34, 0x56, 0x78]);

console.log(buf.readUInt32LE(0).toString(16));
// Prints: 78563412
console.log(buf.readUInt32LE(1).toString(16));
// Throws ERR_OUT_OF_RANGE.
```

COPY

buf.readUIntBE(offset, byteLength)

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - byteLength`.
- `byteLength` <integer> Number of bytes to read. Must satisfy `0 < byteLength <= 6`.
- Returns: <integer>

Reads `byteLength` number of bytes from `buf` at the specified `offset` and interprets the result as an unsigned big-endian integer supporting up to 48 bits of accuracy.

This function is also available under the `readUintBE` alias.

```mjs
import { Buffer } from 'node:buffer';

const buf = Buffer.from([0x12, 0x34, 0x56, 0x78, 0x90, 0xab]);

console.log(buf.readUIntBE(0, 6).toString(16));
// Prints: 1234567890ab
console.log(buf.readUIntBE(1, 6).toString(16));
// Throws ERR_OUT_OF_RANGE.
```

```cjs
const { Buffer } = require('node:buffer');

const buf = Buffer.from([0x12, 0x34, 0x56, 0x78, 0x90, 0xab]);

console.log(buf.readUIntBE(0, 6).toString(16));
// Prints: 1234567890ab
console.log(buf.readUIntBE(1, 6).toString(16));
// Throws ERR_OUT_OF_RANGE.
```

COPY

buf.readUIntLE(offset, byteLength)

- `offset` <integer> Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - byteLength`.
- `byteLength` <integer> Number of bytes to read. Must satisfy `0 < byteLength <= 6`.
- Returns: <integer>

Reads `byteLength` number of bytes from `buf` at the specified `offset` and interprets the result as an unsigned, little-endian integer supporting up to 48 bits of accuracy.

This function is also available under the `readUintLE` alias.

```mjs
import { Buffer } from 'node:buffer';

const buf = Buffer.from([0x12, 0x34, 0x56, 0x78, 0x90, 0xab]);

console.log(buf.readUIntLE(0, 6).toString(16));
// Prints: ab9078563412
```

```cjs
const { Buffer } = require('node:buffer');

const buf = Buffer.from([0x12, 0x34, 0x56, 0x78, 0x90, 0xab]);

console.log(buf.readUIntLE(0, 6).toString(16));
// Prints: ab9078563412
```

COPY

buf.subarray([start[, end]])

- `start` <integer> Where the new `Buffer` will start. **Default:** `0`.
- `end` <integer> Where the new `Buffer` will end (not inclusive). **Default:** `buf.length`.
- Returns: <Buffer>

Returns a new `Buffer` that references the same memory as the original, but offset and cropped by the `start` and `end` indexes.

Specifying `end` greater than `buf.length` will return the same result as that of `end` equal to `buf.length`.

This method is inherited from `TypedArray.prototype.subarray()`.

Modifying the new `Buffer` slice will modify the memory in the original `Buffer` because the allocated memory of the two objects overlap.

```mjs
import { Buffer } from 'node:buffer';

// Create a `Buffer` with the ASCII alphabet, take a slice, and modify one byte
// from the original `Buffer`.

const buf1 = Buffer.allocUnsafe(26);

for (let i = 0; i < 26; i++) {
  // 97 is the decimal ASCII value for 'a'.
  buf1[i] = i + 97;
}

const buf2 = buf1.subarray(0, 3);

console.log(buf2.toString('ascii', 0, buf2.length));
// Prints: abc

buf1[0] = 33;

console.log(buf2.toString('ascii', 0, buf2.length));
// Prints: !bc
```

```cjs
const { Buffer } = require('node:buffer');

// Create a `Buffer` with the ASCII alphabet, take a slice, and modify one byte
// from the original `Buffer`.

const buf1 = Buffer.allocUnsafe(26);

for (let i = 0; i < 26; i++) {
  // 97 is the decimal ASCII value for 'a'.
  buf1[i] = i + 97;
}

const buf2 = buf1.subarray(0, 3);

console.log(buf2.toString('ascii', 0, buf2.length));
// Prints: abc

buf1[0] = 33;
```

```
console.log(buf2.toString('ascii', 0, buf2.length));
// Prints: !bc
```

COPY

Specifying negative indexes causes the slice to be generated relative to the end of `buf` rather than the beginning.

```
import { Buffer } from 'node:buffer';

const buf = Buffer.from('buffer');

console.log(buf.subarray(-6, -1).toString());
// Prints: buffe
// (Equivalent to buf.subarray(0, 5).)

console.log(buf.subarray(-6, -2).toString());
// Prints: buff
// (Equivalent to buf.subarray(0, 4).)

console.log(buf.subarray(-5, -2).toString());
// Prints: uff
// (Equivalent to buf.subarray(1, 4).)
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.from('buffer');

console.log(buf.subarray(-6, -1).toString());
// Prints: buffe
// (Equivalent to buf.subarray(0, 5).)

console.log(buf.subarray(-6, -2).toString());
// Prints: buff
// (Equivalent to buf.subarray(0, 4).)

console.log(buf.subarray(-5, -2).toString());
// Prints: uff
// (Equivalent to buf.subarray(1, 4).)
```

COPY

buf.slice([start[, end]])

- `start` <integer> Where the new `Buffer` will start. **Default:** `0`.
- `end` <integer> Where the new `Buffer` will end (not inclusive). **Default:** `buf.length`.
- Returns: <Buffer>

Stability: 0 - Deprecated: Use `buf.subarray` instead.

Returns a new `Buffer` that references the same memory as the original, but offset and cropped by the `start` and `end` indexes.

This method is not compatible with the `Uint8Array.prototype.slice()`, which is a superclass of `Buffer`. To copy the slice, use `Uint8Array.prototype.slice()`.

```mjs
import { Buffer } from 'node:buffer';

const buf = Buffer.from('buffer');

const copiedBuf = Uint8Array.prototype.slice.call(buf);
copiedBuf[0]++;
console.log(copiedBuf.toString());
// Prints: cuffer

console.log(buf.toString());
// Prints: buffer

// With buf.slice(), the original buffer is modified.
const notReallyCopiedBuf = buf.slice();
notReallyCopiedBuf[0]++;
console.log(notReallyCopiedBuf.toString());
// Prints: cuffer
console.log(buf.toString());
// Also prints: cuffer (!)
```

```cjs
const { Buffer } = require('node:buffer');

const buf = Buffer.from('buffer');

const copiedBuf = Uint8Array.prototype.slice.call(buf);
copiedBuf[0]++;
console.log(copiedBuf.toString());
// Prints: cuffer

console.log(buf.toString());
// Prints: buffer

// With buf.slice(), the original buffer is modified.
const notReallyCopiedBuf = buf.slice();
notReallyCopiedBuf[0]++;
console.log(notReallyCopiedBuf.toString());
// Prints: cuffer
console.log(buf.toString());
// Also prints: cuffer (!)
```

COPY

buf.swap16()

- Returns: <Buffer> A reference to `buf`.

Interprets `buf` as an array of unsigned 16-bit integers and swaps the byte order *in-place*. Throws `ERR_INVALID_BUFFER_SIZE` if `buf.length` is not a multiple of 2.

```mjs
import { Buffer } from 'node:buffer';

const buf1 = Buffer.from([0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8]);

console.log(buf1);
// Prints: <Buffer 01 02 03 04 05 06 07 08>

buf1.swap16();

console.log(buf1);
// Prints: <Buffer 02 01 04 03 06 05 08 07>

const buf2 = Buffer.from([0x1, 0x2, 0x3]);

buf2.swap16();
// Throws ERR_INVALID_BUFFER_SIZE.
```

```cjs
const { Buffer } = require('node:buffer');

const buf1 = Buffer.from([0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8]);

console.log(buf1);
// Prints: <Buffer 01 02 03 04 05 06 07 08>

buf1.swap16();

console.log(buf1);
// Prints: <Buffer 02 01 04 03 06 05 08 07>

const buf2 = Buffer.from([0x1, 0x2, 0x3]);

buf2.swap16();
// Throws ERR_INVALID_BUFFER_SIZE.
```

COPY

One convenient use of `buf.swap16()` is to perform a fast in-place conversion between UTF-16 little-endian and UTF-16 big-endian:

```mjs
import { Buffer } from 'node:buffer';

const buf = Buffer.from('This is little-endian UTF-16', 'utf16le');
buf.swap16(); // Convert to big-endian UTF-16 text.
```

```cjs
const { Buffer } = require('node:buffer');

const buf = Buffer.from('This is little-endian UTF-16', 'utf16le');
buf.swap16(); // Convert to big-endian UTF-16 text.
```

COPY

buf.swap32()

- Returns: <Buffer> A reference to buf.

Interprets buf as an array of unsigned 32-bit integers and swaps the byte order *in-place*. Throws ERR_INVALID_BUFFER_SIZE if buf.length is not a multiple of 4.

```
import { Buffer } from 'node:buffer';

const buf1 = Buffer.from([0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8]);

console.log(buf1);
// Prints: <Buffer 01 02 03 04 05 06 07 08>

buf1.swap32();

console.log(buf1);
// Prints: <Buffer 04 03 02 01 08 07 06 05>

const buf2 = Buffer.from([0x1, 0x2, 0x3]);

buf2.swap32();
// Throws ERR_INVALID_BUFFER_SIZE.
```

```
const { Buffer } = require('node:buffer');

const buf1 = Buffer.from([0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8]);

console.log(buf1);
// Prints: <Buffer 01 02 03 04 05 06 07 08>

buf1.swap32();

console.log(buf1);
// Prints: <Buffer 04 03 02 01 08 07 06 05>

const buf2 = Buffer.from([0x1, 0x2, 0x3]);

buf2.swap32();
// Throws ERR_INVALID_BUFFER_SIZE.
```

COPY

buf.swap64()

- Returns: <Buffer> A reference to buf.

Interprets buf as an array of 64-bit numbers and swaps byte order *in-place*. Throws ERR_INVALID_BUFFER_SIZE if buf.length is not a multiple of 8.

```
import { Buffer } from 'node:buffer';

const buf1 = Buffer.from([0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8]);
```

```
console.log(buf1);
// Prints: <Buffer 01 02 03 04 05 06 07 08>

buf1.swap64();

console.log(buf1);
// Prints: <Buffer 08 07 06 05 04 03 02 01>

const buf2 = Buffer.from([0x1, 0x2, 0x3]);

buf2.swap64();
// Throws ERR_INVALID_BUFFER_SIZE.
```

```
const { Buffer } = require('node:buffer');

const buf1 = Buffer.from([0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8]);

console.log(buf1);
// Prints: <Buffer 01 02 03 04 05 06 07 08>

buf1.swap64();

console.log(buf1);
// Prints: <Buffer 08 07 06 05 04 03 02 01>

const buf2 = Buffer.from([0x1, 0x2, 0x3]);

buf2.swap64();
// Throws ERR_INVALID_BUFFER_SIZE.
```

COPY

buf.toJSON()

- Returns: <Object>

Returns a JSON representation of `buf`. `JSON.stringify()` implicitly calls this function when stringifying a `Buffer` instance.

`Buffer.from()` accepts objects in the format returned from this method. In particular, `Buffer.from(buf.toJSON())` works like `Buffer.from(buf)`.

```
import { Buffer } from 'node:buffer';

const buf = Buffer.from([0x1, 0x2, 0x3, 0x4, 0x5]);
const json = JSON.stringify(buf);

console.log(json);
// Prints: {"type":"Buffer","data":[1,2,3,4,5]}

const copy = JSON.parse(json, (key, value) => {
  return value && value.type === 'Buffer' ?
    Buffer.from(value) :
```

```
    value;
});

console.log(copy);
// Prints: <Buffer 01 02 03 04 05>
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.from([0x1, 0x2, 0x3, 0x4, 0x5]);
const json = JSON.stringify(buf);

console.log(json);
// Prints: {"type":"Buffer","data":[1,2,3,4,5]}

const copy = JSON.parse(json, (key, value) => {
  return value && value.type === 'Buffer' ?
    Buffer.from(value) :
    value;
});

console.log(copy);
// Prints: <Buffer 01 02 03 04 05>
```

COPY

buf.toString([encoding[, start[, end]]])

- `encoding` <string> The character encoding to use. **Default:** `'utf8'`.
- `start` <integer> The byte offset to start decoding at. **Default:** `0`.
- `end` <integer> The byte offset to stop decoding at (not inclusive). **Default:** `buf.length`.
- Returns: <string>

Decodes `buf` to a string according to the specified character encoding in `encoding`. `start` and `end` may be passed to decode only a subset of `buf`.

If `encoding` is `'utf8'` and a byte sequence in the input is not valid UTF-8, then each invalid byte is replaced with the replacement character `U+FFFD`.

The maximum length of a string instance (in UTF-16 code units) is available as `buffer.constants.MAX_STRING_LENGTH`.

```
import { Buffer } from 'node:buffer';

const buf1 = Buffer.allocUnsafe(26);

for (let i = 0; i < 26; i++) {
  // 97 is the decimal ASCII value for 'a'.
  buf1[i] = i + 97;
}

console.log(buf1.toString('utf8'));
// Prints: abcdefghijklmnopqrstuvwxyz
console.log(buf1.toString('utf8', 0, 5));
```

```
// Prints: abcde

const buf2 = Buffer.from('tést');

console.log(buf2.toString('hex'));
// Prints: 74c3a97374
console.log(buf2.toString('utf8', 0, 3));
// Prints: té
console.log(buf2.toString(undefined, 0, 3));
// Prints: té
```

```
const { Buffer } = require('node:buffer');

const buf1 = Buffer.allocUnsafe(26);

for (let i = 0; i < 26; i++) {
  // 97 is the decimal ASCII value for 'a'.
  buf1[i] = i + 97;
}

console.log(buf1.toString('utf8'));
// Prints: abcdefghijklmnopqrstuvwxyz
console.log(buf1.toString('utf8', 0, 5));
// Prints: abcde

const buf2 = Buffer.from('tést');

console.log(buf2.toString('hex'));
// Prints: 74c3a97374
console.log(buf2.toString('utf8', 0, 3));
// Prints: té
console.log(buf2.toString(undefined, 0, 3));
// Prints: té
```

COPY

buf.values()

- Returns: <Iterator>

Creates and returns an iterator for `buf` values (bytes). This function is called automatically when a `Buffer` is used in a `for..of` statement.

```
import { Buffer } from 'node:buffer';

const buf = Buffer.from('buffer');

for (const value of buf.values()) {
  console.log(value);
}
// Prints:
//   98
//   117
//   102
```

```
//   102
//   101
//   114

for (const value of buf) {
  console.log(value);
}
// Prints:
//   98
//   117
//   102
//   102
//   101
//   114
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.from('buffer');

for (const value of buf.values()) {
  console.log(value);
}
// Prints:
//   98
//   117
//   102
//   102
//   101
//   114

for (const value of buf) {
  console.log(value);
}
// Prints:
//   98
//   117
//   102
//   102
//   101
//   114
```

COPY

buf.write(string[, offset[, length]][, encoding])

- `string` <string> String to write to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write `string`. **Default:** `0`.
- `length` <integer> Maximum number of bytes to write (written bytes will not exceed `buf.length - offset`). **Default:** `buf.length - offset`.
- `encoding` <string> The character encoding of `string`. **Default:** `'utf8'`.
- Returns: <integer> Number of bytes written.

Writes `string` to `buf` at `offset` according to the character encoding in `encoding`. The `length` parameter is the number of bytes to write. If `buf` did not contain enough space to fit the entire string, only part of `string` will be written. However, partially encoded characters will not be written.

```mjs
import { Buffer } from 'node:buffer';

const buf = Buffer.alloc(256);

const len = buf.write('\u00bd + \u00bc = \u00be', 0);

console.log(`${len} bytes: ${buf.toString('utf8', 0, len)}`);
// Prints: 12 bytes: ½ + ¼ = ¾

const buffer = Buffer.alloc(10);

const length = buffer.write('abcd', 8);

console.log(`${length} bytes: ${buffer.toString('utf8', 8, 10)}`);
// Prints: 2 bytes : ab
```

```cjs
const { Buffer } = require('node:buffer');

const buf = Buffer.alloc(256);

const len = buf.write('\u00bd + \u00bc = \u00be', 0);

console.log(`${len} bytes: ${buf.toString('utf8', 0, len)}`);
// Prints: 12 bytes: ½ + ¼ = ¾

const buffer = Buffer.alloc(10);

const length = buffer.write('abcd', 8);

console.log(`${length} bytes: ${buffer.toString('utf8', 8, 10)}`);
// Prints: 2 bytes : ab
```

COPY

buf.writeBigInt64BE(value[, offset])

- `value` <bigint> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy: `0 <= offset <= buf.length - 8`. **Default:** `0`.
- Returns: <integer> `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as big-endian.

`value` is interpreted and written as a two's complement signed integer.

```mjs
import { Buffer } from 'node:buffer';

const buf = Buffer.allocUnsafe(8);
```

```
buf.writeBigInt64BE(0x0102030405060708n, 0);

console.log(buf);
// Prints: <Buffer 01 02 03 04 05 06 07 08>
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.allocUnsafe(8);

buf.writeBigInt64BE(0x0102030405060708n, 0);

console.log(buf);
// Prints: <Buffer 01 02 03 04 05 06 07 08>
```

COPY

buf.writeBigInt64LE(value[, offset])

- `value` <bigint> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy: `0 <= offset <= buf.length - 8`. **Default:** `0`.
- Returns: <integer> `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as little-endian.

`value` is interpreted and written as a two's complement signed integer.

```
import { Buffer } from 'node:buffer';

const buf = Buffer.allocUnsafe(8);

buf.writeBigInt64LE(0x0102030405060708n, 0);

console.log(buf);
// Prints: <Buffer 08 07 06 05 04 03 02 01>
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.allocUnsafe(8);

buf.writeBigInt64LE(0x0102030405060708n, 0);

console.log(buf);
// Prints: <Buffer 08 07 06 05 04 03 02 01>
```

COPY

buf.writeBigUInt64BE(value[, offset])

- `value` <bigint> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy: `0 <= offset <= buf.length - 8`. **Default:** `0`.
- Returns: <integer> `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as big-endian.

This function is also available under the `writeBigUint64BE` alias.

```
import { Buffer } from 'node:buffer';

const buf = Buffer.allocUnsafe(8);

buf.writeBigUInt64BE(0xdecafafecacefaden, 0);

console.log(buf);
// Prints: <Buffer de ca fa fe ca ce fa de>
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.allocUnsafe(8);

buf.writeBigUInt64BE(0xdecafafecacefaden, 0);

console.log(buf);
// Prints: <Buffer de ca fa fe ca ce fa de>
```

COPY

buf.writeBigUInt64LE(value[, offset])

- `value` <bigint> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy: `0 <= offset <= buf.length - 8`. **Default:** `0`.
- Returns: <integer> `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as little-endian

```
import { Buffer } from 'node:buffer';

const buf = Buffer.allocUnsafe(8);

buf.writeBigUInt64LE(0xdecafafecacefaden, 0);

console.log(buf);
// Prints: <Buffer de fa ce ca fe fa ca de>
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.allocUnsafe(8);

buf.writeBigUInt64LE(0xdecafafecacefaden, 0);

console.log(buf);
// Prints: <Buffer de fa ce ca fe fa ca de>
```

COPY

This function is also available under the `writeBigUint64LE` alias.

buf.writeDoubleBE(value[, offset])

- `value` <number> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 8`. **Default:** `0`.
- Returns: <integer> `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as big-endian. The `value` must be a JavaScript number. Behavior is undefined when `value` is anything other than a JavaScript number.

```mjs
import { Buffer } from 'node:buffer';

const buf = Buffer.allocUnsafe(8);

buf.writeDoubleBE(123.456, 0);

console.log(buf);
// Prints: <Buffer 40 5e dd 2f 1a 9f be 77>
```

```cjs
const { Buffer } = require('node:buffer');

const buf = Buffer.allocUnsafe(8);

buf.writeDoubleBE(123.456, 0);

console.log(buf);
// Prints: <Buffer 40 5e dd 2f 1a 9f be 77>
```

COPY

buf.writeDoubleLE(value[, offset])

- `value` <number> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 8`. **Default:** `0`.
- Returns: <integer> `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as little-endian. The `value` must be a JavaScript number. Behavior is undefined when `value` is anything other than a JavaScript number.

```mjs
import { Buffer } from 'node:buffer';

const buf = Buffer.allocUnsafe(8);

buf.writeDoubleLE(123.456, 0);

console.log(buf);
// Prints: <Buffer 77 be 9f 1a 2f dd 5e 40>
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.allocUnsafe(8);

buf.writeDoubleLE(123.456, 0);

console.log(buf);
// Prints: <Buffer 77 be 9f 1a 2f dd 5e 40>
```

COPY

buf.writeFloatBE(value[, offset])

- `value` <number> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 4`. **Default:** `0`.
- Returns: <integer> `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as big-endian. Behavior is undefined when `value` is anything other than a JavaScript number.

```
import { Buffer } from 'node:buffer';

const buf = Buffer.allocUnsafe(4);

buf.writeFloatBE(0xcafebabe, 0);

console.log(buf);
// Prints: <Buffer 4f 4a fe bb>
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.allocUnsafe(4);

buf.writeFloatBE(0xcafebabe, 0);

console.log(buf);
// Prints: <Buffer 4f 4a fe bb>
```

COPY

buf.writeFloatLE(value[, offset])

- `value` <number> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 4`. **Default:** `0`.
- Returns: <integer> `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as little-endian. Behavior is undefined when `value` is anything other than a JavaScript number.

```
import { Buffer } from 'node:buffer';

const buf = Buffer.allocUnsafe(4);

buf.writeFloatLE(0xcafebabe, 0);
```

```
console.log(buf);
// Prints: <Buffer bb fe 4a 4f>
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.allocUnsafe(4);

buf.writeFloatLE(0xcafebabe, 0);

console.log(buf);
// Prints: <Buffer bb fe 4a 4f>
```

COPY

buf.writeInt8(value[, offset])

- `value` <integer> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 1`. **Default:** `0`.
- Returns: <integer> `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset`. `value` must be a valid signed 8-bit integer. Behavior is undefined when `value` is anything other than a signed 8-bit integer.

`value` is interpreted and written as a two's complement signed integer.

```
import { Buffer } from 'node:buffer';

const buf = Buffer.allocUnsafe(2);

buf.writeInt8(2, 0);
buf.writeInt8(-2, 1);

console.log(buf);
// Prints: <Buffer 02 fe>
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.allocUnsafe(2);

buf.writeInt8(2, 0);
buf.writeInt8(-2, 1);

console.log(buf);
// Prints: <Buffer 02 fe>
```

COPY

buf.writeInt16BE(value[, offset])

- `value` <integer> Number to be written to `buf`.

- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 2`. **Default:** `0`.
- Returns: <integer> `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as big-endian. The `value` must be a valid signed 16-bit integer. Behavior is undefined when `value` is anything other than a signed 16-bit integer.

The `value` is interpreted and written as a two's complement signed integer.

```mjs
import { Buffer } from 'node:buffer';

const buf = Buffer.allocUnsafe(2);

buf.writeInt16BE(0x0102, 0);

console.log(buf);
// Prints: <Buffer 01 02>
```

```cjs
const { Buffer } = require('node:buffer');

const buf = Buffer.allocUnsafe(2);

buf.writeInt16BE(0x0102, 0);

console.log(buf);
// Prints: <Buffer 01 02>
```

COPY

buf.writeInt16LE(value[, offset])

- `value` <integer> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 2`. **Default:** `0`.
- Returns: <integer> `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as little-endian. The `value` must be a valid signed 16-bit integer. Behavior is undefined when `value` is anything other than a signed 16-bit integer.

The `value` is interpreted and written as a two's complement signed integer.

```mjs
import { Buffer } from 'node:buffer';

const buf = Buffer.allocUnsafe(2);

buf.writeInt16LE(0x0304, 0);

console.log(buf);
// Prints: <Buffer 04 03>
```

```cjs
const { Buffer } = require('node:buffer');

const buf = Buffer.allocUnsafe(2);
```

```
buf.writeInt16LE(0x0304, 0);

console.log(buf);
// Prints: <Buffer 04 03>
```

COPY

buf.writeInt32BE(value[, offset])

- `value` <integer> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 4`. **Default:** `0`.
- Returns: <integer> `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as big-endian. The `value` must be a valid signed 32-bit integer. Behavior is undefined when `value` is anything other than a signed 32-bit integer.

The `value` is interpreted and written as a two's complement signed integer.

```
import { Buffer } from 'node:buffer';

const buf = Buffer.allocUnsafe(4);

buf.writeInt32BE(0x01020304, 0);

console.log(buf);
// Prints: <Buffer 01 02 03 04>
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.allocUnsafe(4);

buf.writeInt32BE(0x01020304, 0);

console.log(buf);
// Prints: <Buffer 01 02 03 04>
```

COPY

buf.writeInt32LE(value[, offset])

- `value` <integer> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 4`. **Default:** `0`.
- Returns: <integer> `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as little-endian. The `value` must be a valid signed 32-bit integer. Behavior is undefined when `value` is anything other than a signed 32-bit integer.

The `value` is interpreted and written as a two's complement signed integer.

```
import { Buffer } from 'node:buffer';

const buf = Buffer.allocUnsafe(4);
```

```
buf.writeInt32LE(0x05060708, 0);

console.log(buf);
// Prints: <Buffer 08 07 06 05>
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.allocUnsafe(4);

buf.writeInt32LE(0x05060708, 0);

console.log(buf);
// Prints: <Buffer 08 07 06 05>
```

COPY

buf.writeIntBE(value, offset, byteLength)

- `value` <integer> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - byteLength`.
- `byteLength` <integer> Number of bytes to write. Must satisfy `0 < byteLength <= 6`.
- Returns: <integer> `offset` plus the number of bytes written.

Writes `byteLength` bytes of `value` to `buf` at the specified `offset` as big-endian. Supports up to 48 bits of accuracy. Behavior is undefined when `value` is anything other than a signed integer.

```
import { Buffer } from 'node:buffer';

const buf = Buffer.allocUnsafe(6);

buf.writeIntBE(0x1234567890ab, 0, 6);

console.log(buf);
// Prints: <Buffer 12 34 56 78 90 ab>
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.allocUnsafe(6);

buf.writeIntBE(0x1234567890ab, 0, 6);

console.log(buf);
// Prints: <Buffer 12 34 56 78 90 ab>
```

COPY

buf.writeIntLE(value, offset, byteLength)

- `value` <integer> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - byteLength`.

- `byteLength` <integer> Number of bytes to write. Must satisfy `0 < byteLength <= 6`.
- Returns: <integer> `offset` plus the number of bytes written.

Writes `byteLength` bytes of `value` to `buf` at the specified `offset` as little-endian. Supports up to 48 bits of accuracy. Behavior is undefined when `value` is anything other than a signed integer.

```mjs
import { Buffer } from 'node:buffer';

const buf = Buffer.allocUnsafe(6);

buf.writeIntLE(0x1234567890ab, 0, 6);

console.log(buf);
// Prints: <Buffer ab 90 78 56 34 12>
```

```cjs
const { Buffer } = require('node:buffer');

const buf = Buffer.allocUnsafe(6);

buf.writeIntLE(0x1234567890ab, 0, 6);

console.log(buf);
// Prints: <Buffer ab 90 78 56 34 12>
```

COPY

buf.writeUInt8(value[, offset])

- `value` <integer> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 1`. **Default:** `0`.
- Returns: <integer> `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset`. `value` must be a valid unsigned 8-bit integer. Behavior is undefined when `value` is anything other than an unsigned 8-bit integer.

This function is also available under the `writeUint8` alias.

```mjs
import { Buffer } from 'node:buffer';

const buf = Buffer.allocUnsafe(4);

buf.writeUInt8(0x3, 0);
buf.writeUInt8(0x4, 1);
buf.writeUInt8(0x23, 2);
buf.writeUInt8(0x42, 3);

console.log(buf);
// Prints: <Buffer 03 04 23 42>
```

```cjs
const { Buffer } = require('node:buffer');
```

```
const buf = Buffer.allocUnsafe(4);

buf.writeUInt8(0x3, 0);
buf.writeUInt8(0x4, 1);
buf.writeUInt8(0x23, 2);
buf.writeUInt8(0x42, 3);

console.log(buf);
// Prints: <Buffer 03 04 23 42>
```

COPY

buf.writeUInt16BE(value[, offset])

- `value` <integer> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 2`. **Default:** `0`.
- Returns: <integer> `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as big-endian. The `value` must be a valid unsigned 16-bit integer. Behavior is undefined when `value` is anything other than an unsigned 16-bit integer.

This function is also available under the `writeUint16BE` alias.

```
import { Buffer } from 'node:buffer';

const buf = Buffer.allocUnsafe(4);

buf.writeUInt16BE(0xdead, 0);
buf.writeUInt16BE(0xbeef, 2);

console.log(buf);
// Prints: <Buffer de ad be ef>
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.allocUnsafe(4);

buf.writeUInt16BE(0xdead, 0);
buf.writeUInt16BE(0xbeef, 2);

console.log(buf);
// Prints: <Buffer de ad be ef>
```

COPY

buf.writeUInt16LE(value[, offset])

- `value` <integer> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 2`. **Default:** `0`.
- Returns: <integer> `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as little-endian. The `value` must be a valid unsigned 16-bit integer. Behavior is undefined when `value` is anything other than an unsigned 16-bit integer.

This function is also available under the `writeUint16LE` alias.

```mjs
import { Buffer } from 'node:buffer';

const buf = Buffer.allocUnsafe(4);

buf.writeUInt16LE(0xdead, 0);
buf.writeUInt16LE(0xbeef, 2);

console.log(buf);
// Prints: <Buffer ad de ef be>
```

```cjs
const { Buffer } = require('node:buffer');

const buf = Buffer.allocUnsafe(4);

buf.writeUInt16LE(0xdead, 0);
buf.writeUInt16LE(0xbeef, 2);

console.log(buf);
// Prints: <Buffer ad de ef be>
```

COPY

buf.writeUInt32BE(value[, offset])

- `value` <integer> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 4`. **Default:** `0`.
- Returns: <integer> `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as big-endian. The `value` must be a valid unsigned 32-bit integer. Behavior is undefined when `value` is anything other than an unsigned 32-bit integer.

This function is also available under the `writeUint32BE` alias.

```mjs
import { Buffer } from 'node:buffer';

const buf = Buffer.allocUnsafe(4);

buf.writeUInt32BE(0xfeedface, 0);

console.log(buf);
// Prints: <Buffer fe ed fa ce>
```

```cjs
const { Buffer } = require('node:buffer');

const buf = Buffer.allocUnsafe(4);

buf.writeUInt32BE(0xfeedface, 0);
```

```
console.log(buf);
// Prints: <Buffer fe ed fa ce>
```

COPY

buf.writeUInt32LE(value[, offset])

- `value` <integer> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 4`. **Default:** `0`.
- Returns: <integer> `offset` plus the number of bytes written.

Writes `value` to `buf` at the specified `offset` as little-endian. The `value` must be a valid unsigned 32-bit integer. Behavior is undefined when `value` is anything other than an unsigned 32-bit integer.

This function is also available under the `writeUint32LE` alias.

```
import { Buffer } from 'node:buffer';

const buf = Buffer.allocUnsafe(4);

buf.writeUInt32LE(0xfeedface, 0);

console.log(buf);
// Prints: <Buffer ce fa ed fe>
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.allocUnsafe(4);

buf.writeUInt32LE(0xfeedface, 0);

console.log(buf);
// Prints: <Buffer ce fa ed fe>
```

COPY

buf.writeUIntBE(value, offset, byteLength)

- `value` <integer> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - byteLength`.
- `byteLength` <integer> Number of bytes to write. Must satisfy `0 < byteLength <= 6`.
- Returns: <integer> `offset` plus the number of bytes written.

Writes `byteLength` bytes of `value` to `buf` at the specified `offset` as big-endian. Supports up to 48 bits of accuracy. Behavior is undefined when `value` is anything other than an unsigned integer.

This function is also available under the `writeUintBE` alias.

```
import { Buffer } from 'node:buffer';

const buf = Buffer.allocUnsafe(6);
```

```
buf.writeUIntBE(0x1234567890ab, 0, 6);

console.log(buf);
// Prints: <Buffer 12 34 56 78 90 ab>
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.allocUnsafe(6);

buf.writeUIntBE(0x1234567890ab, 0, 6);

console.log(buf);
// Prints: <Buffer 12 34 56 78 90 ab>
```

COPY

buf.writeUIntLE(value, offset, byteLength)

- `value` <integer> Number to be written to `buf`.
- `offset` <integer> Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - byteLength`.
- `byteLength` <integer> Number of bytes to write. Must satisfy `0 < byteLength <= 6`.
- Returns: <integer> `offset` plus the number of bytes written.

Writes `byteLength` bytes of `value` to `buf` at the specified `offset` as little-endian. Supports up to 48 bits of accuracy. Behavior is undefined when `value` is anything other than an unsigned integer.

This function is also available under the `writeUintLE` alias.

```
import { Buffer } from 'node:buffer';

const buf = Buffer.allocUnsafe(6);

buf.writeUIntLE(0x1234567890ab, 0, 6);

console.log(buf);
// Prints: <Buffer ab 90 78 56 34 12>
```

```
const { Buffer } = require('node:buffer');

const buf = Buffer.allocUnsafe(6);

buf.writeUIntLE(0x1234567890ab, 0, 6);

console.log(buf);
// Prints: <Buffer ab 90 78 56 34 12>
```

COPY

new Buffer(array)

Stability: 0 - Deprecated: Use `Buffer.from(array)` instead.

- `array` <integer[]> An array of bytes to copy from.

See `Buffer.from(array)`.

new Buffer(arrayBuffer[, byteOffset[, length]])

Stability: 0 - Deprecated: Use `Buffer.from(arrayBuffer[, byteOffset[, length]])` instead.

- `arrayBuffer` <ArrayBuffer> | <SharedArrayBuffer> An `ArrayBuffer`, `SharedArrayBuffer` or the `.buffer` property of a `TypedArray`.
- `byteOffset` <integer> Index of first byte to expose. **Default:** `0`.
- `length` <integer> Number of bytes to expose. **Default:** `arrayBuffer.byteLength - byteOffset`.

See `Buffer.from(arrayBuffer[, byteOffset[, length]])`.

new Buffer(buffer)

Stability: 0 - Deprecated: Use `Buffer.from(buffer)` instead.

- `buffer` <Buffer> | <Uint8Array> An existing `Buffer` or `Uint8Array` from which to copy data.

See `Buffer.from(buffer)`.

new Buffer(size)

Stability: 0 - Deprecated: Use `Buffer.alloc()` instead (also see `Buffer.allocUnsafe()`).

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

See `Buffer.alloc()` and `Buffer.allocUnsafe()`. This variant of the constructor is equivalent to `Buffer.alloc()`.

new Buffer(string[, encoding])

Stability: 0 - Deprecated: Use `Buffer.from(string[, encoding])` instead.

- `string` <string> String to encode.
- `encoding` <string> The encoding of `string`. **Default:** `'utf8'`.

See `Buffer.from(string[, encoding])`.

Class: File

- Extends: <Blob>

A `File` provides information about files.

new buffer.File(sources, fileName[, options])

- `sources` <string[]> | <ArrayBuffer[]> | <TypedArray[]> | <DataView[]> | <Blob[]> | <File[]> An array of string, <ArrayBuffer>, <TypedArray>, <DataView>, <File>, or <Blob> objects, or any mix of such objects, that will be stored within the `File`.
- `fileName` <string> The name of the file.
- `options` <Object>
  - `endings` <string> One of either `'transparent'` or `'native'`. When set to `'native'`, line endings in string source parts will be converted to the platform native line-ending as specified by `require('node:os').EOL`.
  - `type` <string> The File content-type.
  - `lastModified` <number> The last modified date of the file. **Default:** `Date.now()`.

file.name

- Type: <string>

The name of the `File`.

file.lastModified

- Type: <number>

The last modified date of the `File`.

node:buffer module APIs

While, the `Buffer` object is available as a global, there are additional `Buffer`-related APIs that are available only via the `node:buffer` module accessed using `require('node:buffer')`.

buffer.atob(data)

> Stability: 3 - Legacy. Use `Buffer.from(data, 'base64')` instead.

- `data` <any> The Base64-encoded input string.

Decodes a string of Base64-encoded data into bytes, and encodes those bytes into a string using Latin-1 (ISO-8859-1).

The `data` may be any JavaScript-value that can be coerced into a string.

This function is only provided for compatibility with legacy web platform APIs and should never be used in new code, because they use strings to represent binary data and predate the introduction of typed arrays in JavaScript. For code running using Node.js APIs, converting between base64-encoded strings and binary data should be performed using `Buffer.from(str, 'base64')` and `buf.toString('base64')`.

buffer.btoa(data)

> Stability: 3 - Legacy. Use `buf.toString('base64')` instead.

- `data` <any> An ASCII (Latin1) string.

Decodes a string into bytes using Latin-1 (ISO-8859), and encodes those bytes into a string using Base64.

The `data` may be any JavaScript-value that can be coerced into a string.

This function is only provided for compatibility with legacy web platform APIs and should never be used in new code, because they use strings to represent binary data and predate the introduction of typed arrays in JavaScript. For code running using Node.js APIs, converting between base64-encoded strings and binary data should be performed using `Buffer.from(str, 'base64')` and `buf.toString('base64')`.

buffer.isAscii(input)

- input <Buffer> | <ArrayBuffer> | <TypedArray> The input to validate.
- Returns: <boolean>

This function returns `true` if `input` contains only valid ASCII-encoded data, including the case in which `input` is empty.

Throws if the `input` is a detached array buffer.

buffer.isUtf8(input)

- input <Buffer> | <ArrayBuffer> | <TypedArray> The input to validate.
- Returns: <boolean>

This function returns `true` if `input` contains only valid UTF-8-encoded data, including the case in which `input` is empty.

Throws if the `input` is a detached array buffer.

buffer.INSPECT_MAX_BYTES

- <integer> **Default:** `50`

Returns the maximum number of bytes that will be returned when `buf.inspect()` is called. This can be overridden by user modules. See `util.inspect()` for more details on `buf.inspect()` behavior.

buffer.kMaxLength

- <integer> The largest size allowed for a single `Buffer` instance.

An alias for `buffer.constants.MAX_LENGTH`.

buffer.kStringMaxLength

- <integer> The largest length allowed for a single `string` instance.

An alias for `buffer.constants.MAX_STRING_LENGTH`.

buffer.resolveObjectURL(id)

Stability: 1 - Experimental

- `id` <string> A `'blob:nodedata:...` URL string returned by a prior call to `URL.createObjectURL()`.
- Returns: <Blob>

Resolves a `'blob:nodedata:...'` an associated <Blob> object registered using a prior call to `URL.createObjectURL()`.

buffer.transcode(source, fromEnc, toEnc)

- `source` <Buffer> | <Uint8Array> A `Buffer` or `Uint8Array` instance.
- `fromEnc` <string> The current encoding.
- `toEnc` <string> To target encoding.
- Returns: <Buffer>

Re-encodes the given `Buffer` or `Uint8Array` instance from one character encoding to another. Returns a new `Buffer` instance.

Throws if the `fromEnc` or `toEnc` specify invalid character encodings or if conversion from `fromEnc` to `toEnc` is not permitted.

Encodings supported by `buffer.transcode()` are: `'ascii'`, `'utf8'`, `'utf16le'`, `'ucs2'`, `'latin1'`, and `'binary'`.

The transcoding process will use substitution characters if a given byte sequence cannot be adequately represented in the target encoding. For instance:

```mjs
import { Buffer, transcode } from 'node:buffer';

const newBuf = transcode(Buffer.from('€'), 'utf8', 'ascii');
console.log(newBuf.toString('ascii'));
// Prints: '?'
```

```cjs
const { Buffer, transcode } = require('node:buffer');

const newBuf = transcode(Buffer.from('€'), 'utf8', 'ascii');
console.log(newBuf.toString('ascii'));
// Prints: '?'
```

COPY

Because the Euro (`€`) sign is not representable in US-ASCII, it is replaced with `?` in the transcoded `Buffer`.

Class: SlowBuffer

Stability: 0 - Deprecated: Use `Buffer.allocUnsafeSlow()` instead.

See `Buffer.allocUnsafeSlow()`. This was never a class in the sense that the constructor always returned a `Buffer` instance, rather than a `SlowBuffer` instance.

new SlowBuffer(size)

Stability: 0 - Deprecated: Use `Buffer.allocUnsafeSlow()` instead.

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

See `Buffer.allocUnsafeSlow()`.

Buffer constants

buffer.constants.MAX_LENGTH

- `<integer>` The largest size allowed for a single `Buffer` instance.

On 32-bit architectures, this value currently is 2^{30} - 1 (about 1 GiB).

On 64-bit architectures, this value currently is 2^{32} (about 4 GiB).

It reflects `v8::TypedArray::kMaxLength` under the hood.

This value is also available as `buffer.kMaxLength`.

buffer.constants.MAX_STRING_LENGTH

- <integer> The largest length allowed for a single `string` instance.

Represents the largest `length` that a `string` primitive can have, counted in UTF-16 code units.

This value may depend on the JS engine that is being used.

Buffer.from(), Buffer.alloc(), and Buffer.allocUnsafe()

In versions of Node.js prior to 6.0.0, `Buffer` instances were created using the `Buffer` constructor function, which allocates the returned `Buffer` differently based on what arguments are provided:

- Passing a number as the first argument to `Buffer()` (e.g. `new Buffer(10)`) allocates a new `Buffer` object of the specified size. Prior to Node.js 8.0.0, the memory allocated for such `Buffer` instances is *not* initialized and *can contain sensitive data*. Such `Buffer` instances *must* be subsequently initialized by using either `buf.fill(0)` or by writing to the entire `Buffer` before reading data from the `Buffer`. While this behavior is *intentional* to improve performance, development experience has demonstrated that a more explicit distinction is required between creating a fast-but-uninitialized `Buffer` versus creating a slower-but-safer `Buffer`. Since Node.js 8.0.0, `Buffer(num)` and `new Buffer(num)` return a `Buffer` with initialized memory.
- Passing a string, array, or `Buffer` as the first argument copies the passed object's data into the `Buffer`.
- Passing an `ArrayBuffer` or a `SharedArrayBuffer` returns a `Buffer` that shares allocated memory with the given array buffer.

Because the behavior of `new Buffer()` is different depending on the type of the first argument, security and reliability issues can be inadvertently introduced into applications when argument validation or `Buffer` initialization is not performed.

For example, if an attacker can cause an application to receive a number where a string is expected, the application may call `new Buffer(100)` instead of `new Buffer("100")`, leading it to allocate a 100 byte buffer instead of allocating a 3 byte buffer with content `"100"`. This is commonly possible using JSON API calls. Since JSON distinguishes between numeric and string types, it allows injection of numbers where a naively written application that does not validate its input sufficiently might expect to always receive a string. Before Node.js 8.0.0, the 100 byte buffer might contain arbitrary pre-existing in-memory data, so may be used to expose in-memory secrets to a remote attacker. Since Node.js 8.0.0, exposure of memory cannot occur because the data is zero-filled. However, other attacks are still possible, such as causing very large buffers to be allocated by the server, leading to performance degradation or crashing on memory exhaustion.

To make the creation of `Buffer` instances more reliable and less error-prone, the various forms of the `new Buffer()` constructor have been **deprecated** and replaced by separate `Buffer.from()`, `Buffer.alloc()`, and `Buffer.allocUnsafe()` methods.

Developers should migrate all existing uses of the `new Buffer()` *constructors to one of these new APIs.*

- `Buffer.from(array)` returns a new `Buffer` that *contains a copy* of the provided octets.
- `Buffer.from(arrayBuffer[, byteOffset[, length]])` returns a new `Buffer` that *shares the same allocated memory* as the given `ArrayBuffer`.
- `Buffer.from(buffer)` returns a new `Buffer` that *contains a copy* of the contents of the given `Buffer`.
- `Buffer.from(string[, encoding])` returns a new `Buffer` that *contains a copy* of the provided string.
- `Buffer.alloc(size[, fill[, encoding]])` returns a new initialized `Buffer` of the specified size. This method is slower than `Buffer.allocUnsafe(size)` but guarantees that newly created `Buffer` instances never contain old data that is potentially sensitive. A `TypeError` will be thrown if `size` is not a number.
- `Buffer.allocUnsafe(size)` and `Buffer.allocUnsafeSlow(size)` each return a new uninitialized `Buffer` of the specified `size`. Because the `Buffer` is uninitialized, the allocated segment of memory might contain old data that is potentially sensitive.

`Buffer` instances returned by `Buffer.allocUnsafe()`, `Buffer.from(string)`, `Buffer.concat()` and `Buffer.from(array)` *may* be allocated off a shared internal memory pool if `size` is less than or equal to half `Buffer.poolSize`. Instances returned by `Buffer.allocUnsafeSlow()` *never* use the shared internal memory pool.

The `--zero-fill-buffers` command-line option

Node.js can be started using the `--zero-fill-buffers` command-line option to cause all newly-allocated `Buffer` instances to be zero-filled upon creation by default. Without the option, buffers created with `Buffer.allocUnsafe()`, `Buffer.allocUnsafeSlow()`, and `new`

`SlowBuffer(size)` are not zero-filled. Use of this flag can have a measurable negative impact on performance. Use the `--zero-fill-buffers` option only when necessary to enforce that newly allocated `Buffer` instances cannot contain old data that is potentially sensitive.

```
$ node --zero-fill-buffers
> Buffer.allocUnsafe(5);
<Buffer 00 00 00 00 00>
```

COPY

What makes Buffer.allocUnsafe() and Buffer.allocUnsafeSlow() "unsafe"?

When calling `Buffer.allocUnsafe()` and `Buffer.allocUnsafeSlow()`, the segment of allocated memory is *uninitialized* (it is not zeroed-out). While this design makes the allocation of memory quite fast, the allocated segment of memory might contain old data that is potentially sensitive. Using a `Buffer` created by `Buffer.allocUnsafe()` without *completely* overwriting the memory can allow this old data to be leaked when the `Buffer` memory is read.

While there are clear performance advantages to using `Buffer.allocUnsafe()`, extra care *must* be taken in order to avoid introducing security vulnerabilities into an application.