/\*\*

\* `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`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Uint8Array) class and

\* extends it with methods that cover additional use cases. Node.js APIs accept

\* plain [`Uint8Array`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/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.

\*

\* ```js

\* import { Buffer } from '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 &#x26; 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');

\* ```

\* @see [source](https://github.com/nodejs/node/blob/v17.0.0/lib/buffer.js)

\*/

declare module 'buffer' {

import { BinaryLike } from 'node:crypto';

export const INSPECT\_MAX\_BYTES: number;

export const kMaxLength: number;

export const kStringMaxLength: number;

export const constants: {

MAX\_LENGTH: number;

MAX\_STRING\_LENGTH: number;

};

export type TranscodeEncoding = 'ascii' | 'utf8' | 'utf16le' | 'ucs2' | 'latin1' | 'binary';

/\*\*

\* 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:

\*

\* ```js

\* import { Buffer, transcode } from 'buffer';

\*

\* const newBuf = transcode(Buffer.from('€'), 'utf8', 'ascii');

\* console.log(newBuf.toString('ascii'));

\* // Prints: '?'

\* ```

\*

\* Because the Euro (`€`) sign is not representable in US-ASCII, it is replaced

\* with `?` in the transcoded `Buffer`.

\* @since v7.1.0

\* @param source A `Buffer` or `Uint8Array` instance.

\* @param fromEnc The current encoding.

\* @param toEnc To target encoding.

\*/

export function transcode(source: Uint8Array, fromEnc: TranscodeEncoding, toEnc: TranscodeEncoding): Buffer;

export const SlowBuffer: {

/\*\* @deprecated since v6.0.0, use `Buffer.allocUnsafeSlow()` \*/

new (size: number): Buffer;

prototype: Buffer;

};

/\*\*

\* Resolves a `'blob:nodedata:...'` an associated `Blob` object registered using

\* a prior call to `URL.createObjectURL()`.

\* @since v16.7.0

\* @experimental

\* @param id A `'blob:nodedata:...` URL string returned by a prior call to `URL.createObjectURL()`.

\*/

export function resolveObjectURL(id: string): Blob | undefined;

export { Buffer };

/\*\*

\* @experimental

\*/

export interface BlobOptions {

/\*\*

\* @default 'utf8'

\*/

encoding?: BufferEncoding | undefined;

/\*\*

\* 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.

\*/

type?: string | undefined;

}

/\*\*

\* A [`Blob`](https://developer.mozilla.org/en-US/docs/Web/API/Blob) encapsulates immutable, raw data that can be safely shared across

\* multiple worker threads.

\* @since v15.7.0, v14.18.0

\* @experimental

\*/

export class Blob {

/\*\*

\* The total size of the `Blob` in bytes.

\* @since v15.7.0, v14.18.0

\*/

readonly size: number;

/\*\*

\* The content-type of the `Blob`.

\* @since v15.7.0, v14.18.0

\*/

readonly type: string;

/\*\*

\* 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 also copied into the `Blob`.

\*/

constructor(sources: Array<BinaryLike | Blob>, options?: BlobOptions);

/\*\*

\* Returns a promise that fulfills with an [ArrayBuffer](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/ArrayBuffer) containing a copy of

\* the `Blob` data.

\* @since v15.7.0, v14.18.0

\*/

arrayBuffer(): Promise<ArrayBuffer>;

/\*\*

\* Creates and returns a new `Blob` containing a subset of this `Blob` objects

\* data. The original `Blob` is not altered.

\* @since v15.7.0, v14.18.0

\* @param start The starting index.

\* @param end The ending index.

\* @param type The content-type for the new `Blob`

\*/

slice(start?: number, end?: number, type?: string): Blob;

/\*\*

\* Returns a promise that fulfills with the contents of the `Blob` decoded as a

\* UTF-8 string.

\* @since v15.7.0, v14.18.0

\*/

text(): Promise<string>;

/\*\*

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

\* @since v16.7.0

\*/

stream(): unknown; // pending web streams types

}

export import atob = globalThis.atob;

export import btoa = globalThis.btoa;

global {

// Buffer class

type BufferEncoding = 'ascii' | 'utf8' | 'utf-8' | 'utf16le' | 'ucs2' | 'ucs-2' | 'base64' | 'base64url' | 'latin1' | 'binary' | 'hex';

type WithImplicitCoercion<T> =

| T

| {

valueOf(): T;

};

/\*\*

\* Raw data is stored in instances of the Buffer class.

\* A Buffer is similar to an array of integers but corresponds to a raw memory allocation outside the V8 heap. A Buffer cannot be resized.

\* Valid string encodings: 'ascii'|'utf8'|'utf16le'|'ucs2'(alias of 'utf16le')|'base64'|'base64url'|'binary'(deprecated)|'hex'

\*/

interface BufferConstructor {

/\*\*

\* Allocates a new buffer containing the given {str}.

\*

\* @param str String to store in buffer.

\* @param encoding encoding to use, optional. Default is 'utf8'

\* @deprecated since v10.0.0 - Use `Buffer.from(string[, encoding])` instead.

\*/

new (str: string, encoding?: BufferEncoding): Buffer;

/\*\*

\* Allocates a new buffer of {size} octets.

\*

\* @param size count of octets to allocate.

\* @deprecated since v10.0.0 - Use `Buffer.alloc()` instead (also see `Buffer.allocUnsafe()`).

\*/

new (size: number): Buffer;

/\*\*

\* Allocates a new buffer containing the given {array} of octets.

\*

\* @param array The octets to store.

\* @deprecated since v10.0.0 - Use `Buffer.from(array)` instead.

\*/

new (array: Uint8Array): Buffer;

/\*\*

\* Produces a Buffer backed by the same allocated memory as

\* the given {ArrayBuffer}/{SharedArrayBuffer}.

\*

\*

\* @param arrayBuffer The ArrayBuffer with which to share memory.

\* @deprecated since v10.0.0 - Use `Buffer.from(arrayBuffer[, byteOffset[, length]])` instead.

\*/

new (arrayBuffer: ArrayBuffer | SharedArrayBuffer): Buffer;

/\*\*

\* Allocates a new buffer containing the given {array} of octets.

\*

\* @param array The octets to store.

\* @deprecated since v10.0.0 - Use `Buffer.from(array)` instead.

\*/

new (array: ReadonlyArray<any>): Buffer;

/\*\*

\* Copies the passed {buffer} data onto a new {Buffer} instance.

\*

\* @param buffer The buffer to copy.

\* @deprecated since v10.0.0 - Use `Buffer.from(buffer)` instead.

\*/

new (buffer: Buffer): Buffer;

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

\* // Creates a new Buffer containing the UTF-8 bytes of the string 'buffer'.

\* const buf = Buffer.from([0x62, 0x75, 0x66, 0x66, 0x65, 0x72]);

\* ```

\*

\* 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.

\* @since v5.10.0

\*/

from(arrayBuffer: WithImplicitCoercion<ArrayBuffer | SharedArrayBuffer>, byteOffset?: number, length?: number): Buffer;

/\*\*

\* Creates a new Buffer using the passed {data}

\* @param data data to create a new Buffer

\*/

from(data: Uint8Array | ReadonlyArray<number>): Buffer;

from(data: WithImplicitCoercion<Uint8Array | ReadonlyArray<number> | string>): Buffer;

/\*\*

\* Creates a new Buffer containing the given JavaScript string {str}.

\* If provided, the {encoding} parameter identifies the character encoding.

\* If not provided, {encoding} defaults to 'utf8'.

\*/

from(

str:

| WithImplicitCoercion<string>

| {

[Symbol.toPrimitive](hint: 'string'): string;

},

encoding?: BufferEncoding

): Buffer;

/\*\*

\* Creates a new Buffer using the passed {data}

\* @param values to create a new Buffer

\*/

of(...items: number[]): Buffer;

/\*\*

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

\*

\* ```js

\* import { Buffer } from '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

\* ```

\* @since v0.1.101

\*/

isBuffer(obj: any): obj is Buffer;

/\*\*

\* Returns `true` if `encoding` is the name of a supported character encoding,

\* or `false` otherwise.

\*

\* ```js

\* import { Buffer } from '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

\* ```

\* @since v0.9.1

\* @param encoding A character encoding name to check.

\*/

isEncoding(encoding: string): encoding is BufferEncoding;

/\*\*

\* Returns the byte length of a string when encoded using `encoding`.

\* This is not the same as [`String.prototype.length`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/String/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.

\*

\* ```js

\* import { Buffer } from '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

\* ```

\*

\* When `string` is a

\* `Buffer`/[`DataView`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/DataView)/[`TypedArray`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/-

\* Reference/Global\_Objects/TypedArray)/[`ArrayBuffer`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/ArrayBuffer)/[`SharedArrayBuffer`](https://develop-

\* er.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/SharedArrayBuffer), the byte length as reported by `.byteLength`is returned.

\* @since v0.1.90

\* @param string A value to calculate the length of.

\* @param [encoding='utf8'] If `string` is a string, this is its encoding.

\* @return The number of bytes contained within `string`.

\*/

byteLength(string: string | NodeJS.ArrayBufferView | ArrayBuffer | SharedArrayBuffer, encoding?: BufferEncoding): number;

/\*\*

\* 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`.

\*

\* ```js

\* import { Buffer } from '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

\* ```

\*

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

\* @since v0.7.11

\* @param list List of `Buffer` or {@link Uint8Array} instances to concatenate.

\* @param totalLength Total length of the `Buffer` instances in `list` when concatenated.

\*/

concat(list: ReadonlyArray<Uint8Array>, totalLength?: number): Buffer;

/\*\*

\* Compares `buf1` to `buf2`, typically for the purpose of sorting arrays of`Buffer` instances. This is equivalent to calling `buf1.compare(buf2)`.

\*

\* ```js

\* import { Buffer } from '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].)

\* ```

\* @since v0.11.13

\* @return Either `-1`, `0`, or `1`, depending on the result of the comparison. See `compare` for details.

\*/

compare(buf1: Uint8Array, buf2: Uint8Array): number;

/\*\*

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

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

\* const buf = Buffer.alloc(5);

\*

\* console.log(buf);

\* // Prints: <Buffer 00 00 00 00 00>

\* ```

\*

\* If `size` is larger than {@link constants.MAX\_LENGTH} or smaller than 0, `ERR\_INVALID\_ARG\_VALUE` is thrown.

\*

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

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

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

\*

\* console.log(buf);

\* // Prints: <Buffer 61 61 61 61 61>

\* ```

\*

\* If both `fill` and `encoding` are specified, the allocated `Buffer` will be

\* initialized by calling `buf.fill(fill, encoding)`.

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

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

\*

\* console.log(buf);

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

\* ```

\*

\* 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.

\* @since v5.10.0

\* @param size The desired length of the new `Buffer`.

\* @param [fill=0] A value to pre-fill the new `Buffer` with.

\* @param [encoding='utf8'] If `fill` is a string, this is its encoding.

\*/

alloc(size: number, fill?: string | Buffer | number, encoding?: BufferEncoding): Buffer;

/\*\*

\* Allocates a new `Buffer` of `size` bytes. If `size` is larger than {@link constants.MAX\_LENGTH} or smaller than 0, `ERR\_INVALID\_ARG\_VALUE` is thrown.

\*

\* The underlying memory for `Buffer` instances created in this way is \_not\_

\* \_initialized\_. The contents of the newly created `Buffer` are unknown and\_may contain sensitive data\_. Use `Buffer.alloc()` instead to initialize`Buffer` instances with zeroes.

\*

\* ```js

\* import { Buffer } from '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>

\* ```

\*

\* 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.concat()`, and the

\* deprecated`new Buffer(size)` constructor only when `size` is less than or equal

\* to `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.

\* @since v5.10.0

\* @param size The desired length of the new `Buffer`.

\*/

allocUnsafe(size: number): Buffer;

/\*\*

\* Allocates a new `Buffer` of `size` bytes. If `size` is larger than {@link constants.MAX\_LENGTH} or smaller than 0, `ERR\_INVALID\_ARG\_VALUE` 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 under 4 KB 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.

\*

\* ```js

\* import { Buffer } from '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);

\* }

\* });

\* ```

\*

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

\* @since v5.12.0

\* @param size The desired length of the new `Buffer`.

\*/

allocUnsafeSlow(size: number): Buffer;

/\*\*

\* This is the size (in bytes) of pre-allocated internal `Buffer` instances used

\* for pooling. This value may be modified.

\* @since v0.11.3

\*/

poolSize: number;

}

interface Buffer extends Uint8Array {

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from '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

\* ```

\* @since v0.1.90

\* @param string String to write to `buf`.

\* @param [offset=0] Number of bytes to skip before starting to write `string`.

\* @param [length=buf.length - offset] Maximum number of bytes to write (written bytes will not exceed `buf.length - offset`).

\* @param [encoding='utf8'] The character encoding of `string`.

\* @return Number of bytes written.

\*/

write(string: string, encoding?: BufferEncoding): number;

write(string: string, offset: number, encoding?: BufferEncoding): number;

write(string: string, offset: number, length: number, encoding?: BufferEncoding): number;

/\*\*

\* 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 {@link constants.MAX\_STRING\_LENGTH}.

\*

\* ```js

\* import { Buffer } from '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

\* ```

\* @since v0.1.90

\* @param [encoding='utf8'] The character encoding to use.

\* @param [start=0] The byte offset to start decoding at.

\* @param [end=buf.length] The byte offset to stop decoding at (not inclusive).

\*/

toString(encoding?: BufferEncoding, start?: number, end?: number): string;

/\*\*

\* Returns a JSON representation of `buf`. [`JSON.stringify()`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/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)`.

\*

\* ```js

\* import { Buffer } from '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 &#x26;&#x26; value.type === 'Buffer' ?

\* Buffer.from(value) :

\* value;

\* });

\*

\* console.log(copy);

\* // Prints: <Buffer 01 02 03 04 05>

\* ```

\* @since v0.9.2

\*/

toJSON(): {

type: 'Buffer';

data: number[];

};

/\*\*

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

\*

\* ```js

\* import { Buffer } from '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

\* ```

\* @since v0.11.13

\* @param otherBuffer A `Buffer` or {@link Uint8Array} with which to compare `buf`.

\*/

equals(otherBuffer: Uint8Array): boolean;

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from '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].)

\* ```

\*

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

\*

\* ```js

\* import { Buffer } from '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

\* ```

\*

\* `ERR\_OUT\_OF\_RANGE` is thrown if `targetStart < 0`, `sourceStart < 0`,`targetEnd > target.byteLength`, or `sourceEnd > source.byteLength`.

\* @since v0.11.13

\* @param target A `Buffer` or {@link Uint8Array} with which to compare `buf`.

\* @param [targetStart=0] The offset within `target` at which to begin comparison.

\* @param [targetEnd=target.length] The offset within `target` at which to end comparison (not inclusive).

\* @param [sourceStart=0] The offset within `buf` at which to begin comparison.

\* @param [sourceEnd=buf.length] The offset within `buf` at which to end comparison (not inclusive).

\*/

compare(target: Uint8Array, targetStart?: number, targetEnd?: number, sourceStart?: number, sourceEnd?: number): number;

/\*\*

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

\*

\* [`TypedArray.prototype.set()`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/TypedArray/set) performs the same operation, and is available

\* for all TypedArrays, including Node.js `Buffer`s, although it takes

\* different function arguments.

\*

\* ```js

\* import { Buffer } from '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!!!!!!!!!!!!!

\* ```

\*

\* ```js

\* import { Buffer } from '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

\* ```

\* @since v0.1.90

\* @param target A `Buffer` or {@link Uint8Array} to copy into.

\* @param [targetStart=0] The offset within `target` at which to begin writing.

\* @param [sourceStart=0] The offset within `buf` from which to begin copying.

\* @param [sourceEnd=buf.length] The offset within `buf` at which to stop copying (not inclusive).

\* @return The number of bytes copied.

\*/

copy(target: Uint8Array, targetStart?: number, sourceStart?: number, sourceEnd?: number): number;

/\*\*

\* Returns a new `Buffer` that references the same memory as the original, but

\* offset and cropped by the `start` and `end` indices.

\*

\* This is the same behavior as `buf.subarray()`.

\*

\* This method is not compatible with the `Uint8Array.prototype.slice()`,

\* which is a superclass of `Buffer`. To copy the slice, use`Uint8Array.prototype.slice()`.

\*

\* ```js

\* import { Buffer } from '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

\* ```

\* @since v0.3.0

\* @param [start=0] Where the new `Buffer` will start.

\* @param [end=buf.length] Where the new `Buffer` will end (not inclusive).

\*/

slice(start?: number, end?: number): Buffer;

/\*\*

\* Returns a new `Buffer` that references the same memory as the original, but

\* offset and cropped by the `start` and `end` indices.

\*

\* 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()`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/TypedArray/subarray).

\*

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

\*

\* ```js

\* import { Buffer } from '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

\* ```

\*

\* Specifying negative indexes causes the slice to be generated relative to the

\* end of `buf` rather than the beginning.

\*

\* ```js

\* import { Buffer } from '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).)

\* ```

\* @since v3.0.0

\* @param [start=0] Where the new `Buffer` will start.

\* @param [end=buf.length] Where the new `Buffer` will end (not inclusive).

\*/

subarray(start?: number, end?: number): Buffer;

/\*\*

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

\*

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

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

\* const buf = Buffer.allocUnsafe(8);

\*

\* buf.writeBigInt64BE(0x0102030405060708n, 0);

\*

\* console.log(buf);

\* // Prints: <Buffer 01 02 03 04 05 06 07 08>

\* ```

\* @since v12.0.0, v10.20.0

\* @param value Number to be written to `buf`.

\* @param [offset=0] Number of bytes to skip before starting to write. Must satisfy: `0 <= offset <= buf.length - 8`.

\* @return `offset` plus the number of bytes written.

\*/

writeBigInt64BE(value: bigint, offset?: number): number;

/\*\*

\* Writes `value` to `buf` at the specified `offset` as little-endian.

\*

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

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

\* const buf = Buffer.allocUnsafe(8);

\*

\* buf.writeBigInt64LE(0x0102030405060708n, 0);

\*

\* console.log(buf);

\* // Prints: <Buffer 08 07 06 05 04 03 02 01>

\* ```

\* @since v12.0.0, v10.20.0

\* @param value Number to be written to `buf`.

\* @param [offset=0] Number of bytes to skip before starting to write. Must satisfy: `0 <= offset <= buf.length - 8`.

\* @return `offset` plus the number of bytes written.

\*/

writeBigInt64LE(value: bigint, offset?: number): number;

/\*\*

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

\*

\* This function is also available under the `writeBigUint64BE` alias.

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

\* const buf = Buffer.allocUnsafe(8);

\*

\* buf.writeBigUInt64BE(0xdecafafecacefaden, 0);

\*

\* console.log(buf);

\* // Prints: <Buffer de ca fa fe ca ce fa de>

\* ```

\* @since v12.0.0, v10.20.0

\* @param value Number to be written to `buf`.

\* @param [offset=0] Number of bytes to skip before starting to write. Must satisfy: `0 <= offset <= buf.length - 8`.

\* @return `offset` plus the number of bytes written.

\*/

writeBigUInt64BE(value: bigint, offset?: number): number;

/\*\*

\* Writes `value` to `buf` at the specified `offset` as little-endian

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

\* const buf = Buffer.allocUnsafe(8);

\*

\* buf.writeBigUInt64LE(0xdecafafecacefaden, 0);

\*

\* console.log(buf);

\* // Prints: <Buffer de fa ce ca fe fa ca de>

\* ```

\*

\* This function is also available under the `writeBigUint64LE` alias.

\* @since v12.0.0, v10.20.0

\* @param value Number to be written to `buf`.

\* @param [offset=0] Number of bytes to skip before starting to write. Must satisfy: `0 <= offset <= buf.length - 8`.

\* @return `offset` plus the number of bytes written.

\*/

writeBigUInt64LE(value: bigint, offset?: number): number;

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

\* const buf = Buffer.allocUnsafe(6);

\*

\* buf.writeUIntLE(0x1234567890ab, 0, 6);

\*

\* console.log(buf);

\* // Prints: <Buffer ab 90 78 56 34 12>

\* ```

\* @since v0.5.5

\* @param value Number to be written to `buf`.

\* @param offset Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - byteLength`.

\* @param byteLength Number of bytes to write. Must satisfy `0 < byteLength <= 6`.

\* @return `offset` plus the number of bytes written.

\*/

writeUIntLE(value: number, offset: number, byteLength: number): number;

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

\* const buf = Buffer.allocUnsafe(6);

\*

\* buf.writeUIntBE(0x1234567890ab, 0, 6);

\*

\* console.log(buf);

\* // Prints: <Buffer 12 34 56 78 90 ab>

\* ```

\* @since v0.5.5

\* @param value Number to be written to `buf`.

\* @param offset Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - byteLength`.

\* @param byteLength Number of bytes to write. Must satisfy `0 < byteLength <= 6`.

\* @return `offset` plus the number of bytes written.

\*/

writeUIntBE(value: number, offset: number, byteLength: number): number;

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

\* const buf = Buffer.allocUnsafe(6);

\*

\* buf.writeIntLE(0x1234567890ab, 0, 6);

\*

\* console.log(buf);

\* // Prints: <Buffer ab 90 78 56 34 12>

\* ```

\* @since v0.11.15

\* @param value Number to be written to `buf`.

\* @param offset Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - byteLength`.

\* @param byteLength Number of bytes to write. Must satisfy `0 < byteLength <= 6`.

\* @return `offset` plus the number of bytes written.

\*/

writeIntLE(value: number, offset: number, byteLength: number): number;

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

\* const buf = Buffer.allocUnsafe(6);

\*

\* buf.writeIntBE(0x1234567890ab, 0, 6);

\*

\* console.log(buf);

\* // Prints: <Buffer 12 34 56 78 90 ab>

\* ```

\* @since v0.11.15

\* @param value Number to be written to `buf`.

\* @param offset Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - byteLength`.

\* @param byteLength Number of bytes to write. Must satisfy `0 < byteLength <= 6`.

\* @return `offset` plus the number of bytes written.

\*/

writeIntBE(value: number, offset: number, byteLength: number): number;

/\*\*

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

\*

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

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

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

\*

\* console.log(buf.readBigUInt64BE(0));

\* // Prints: 4294967295n

\* ```

\* @since v12.0.0, v10.20.0

\* @param [offset=0] Number of bytes to skip before starting to read. Must satisfy: `0 <= offset <= buf.length - 8`.

\*/

readBigUInt64BE(offset?: number): bigint;

/\*\*

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

\*

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

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

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

\*

\* console.log(buf.readBigUInt64LE(0));

\* // Prints: 18446744069414584320n

\* ```

\* @since v12.0.0, v10.20.0

\* @param [offset=0] Number of bytes to skip before starting to read. Must satisfy: `0 <= offset <= buf.length - 8`.

\*/

readBigUInt64LE(offset?: number): 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.

\* @since v12.0.0, v10.20.0

\* @param [offset=0] Number of bytes to skip before starting to read. Must satisfy: `0 <= offset <= buf.length - 8`.

\*/

readBigInt64BE(offset?: number): 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.

\* @since v12.0.0, v10.20.0

\* @param [offset=0] Number of bytes to skip before starting to read. Must satisfy: `0 <= offset <= buf.length - 8`.

\*/

readBigInt64LE(offset?: number): bigint;

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

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

\*

\* console.log(buf.readUIntLE(0, 6).toString(16));

\* // Prints: ab9078563412

\* ```

\* @since v0.11.15

\* @param offset Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - byteLength`.

\* @param byteLength Number of bytes to read. Must satisfy `0 < byteLength <= 6`.

\*/

readUIntLE(offset: number, byteLength: number): number;

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from '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.

\* ```

\* @since v0.11.15

\* @param offset Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - byteLength`.

\* @param byteLength Number of bytes to read. Must satisfy `0 < byteLength <= 6`.

\*/

readUIntBE(offset: number, byteLength: number): number;

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

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

\*

\* console.log(buf.readIntLE(0, 6).toString(16));

\* // Prints: -546f87a9cbee

\* ```

\* @since v0.11.15

\* @param offset Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - byteLength`.

\* @param byteLength Number of bytes to read. Must satisfy `0 < byteLength <= 6`.

\*/

readIntLE(offset: number, byteLength: number): number;

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from '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.

\* ```

\* @since v0.11.15

\* @param offset Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - byteLength`.

\* @param byteLength Number of bytes to read. Must satisfy `0 < byteLength <= 6`.

\*/

readIntBE(offset: number, byteLength: number): number;

/\*\*

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

\*

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

\*

\* ```js

\* import { Buffer } from '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.

\* ```

\* @since v0.5.0

\* @param [offset=0] Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 1`.

\*/

readUInt8(offset?: number): number;

/\*\*

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

\*

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

\*

\* ```js

\* import { Buffer } from '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.

\* ```

\* @since v0.5.5

\* @param [offset=0] Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 2`.

\*/

readUInt16LE(offset?: number): number;

/\*\*

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

\*

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

\*

\* ```js

\* import { Buffer } from '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

\* ```

\* @since v0.5.5

\* @param [offset=0] Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 2`.

\*/

readUInt16BE(offset?: number): number;

/\*\*

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

\*

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

\*

\* ```js

\* import { Buffer } from '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.

\* ```

\* @since v0.5.5

\* @param [offset=0] Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 4`.

\*/

readUInt32LE(offset?: number): number;

/\*\*

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

\*

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

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

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

\*

\* console.log(buf.readUInt32BE(0).toString(16));

\* // Prints: 12345678

\* ```

\* @since v0.5.5

\* @param [offset=0] Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 4`.

\*/

readUInt32BE(offset?: number): number;

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from '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.

\* ```

\* @since v0.5.0

\* @param [offset=0] Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 1`.

\*/

readInt8(offset?: number): number;

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

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

\*

\* console.log(buf.readInt16LE(0));

\* // Prints: 1280

\* console.log(buf.readInt16LE(1));

\* // Throws ERR\_OUT\_OF\_RANGE.

\* ```

\* @since v0.5.5

\* @param [offset=0] Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 2`.

\*/

readInt16LE(offset?: number): number;

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

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

\*

\* console.log(buf.readInt16BE(0));

\* // Prints: 5

\* ```

\* @since v0.5.5

\* @param [offset=0] Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 2`.

\*/

readInt16BE(offset?: number): number;

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from '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.

\* ```

\* @since v0.5.5

\* @param [offset=0] Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 4`.

\*/

readInt32LE(offset?: number): number;

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

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

\*

\* console.log(buf.readInt32BE(0));

\* // Prints: 5

\* ```

\* @since v0.5.5

\* @param [offset=0] Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 4`.

\*/

readInt32BE(offset?: number): number;

/\*\*

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

\*

\* ```js

\* import { Buffer } from '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.

\* ```

\* @since v0.11.15

\* @param [offset=0] Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 4`.

\*/

readFloatLE(offset?: number): number;

/\*\*

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

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

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

\*

\* console.log(buf.readFloatBE(0));

\* // Prints: 2.387939260590663e-38

\* ```

\* @since v0.11.15

\* @param [offset=0] Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 4`.

\*/

readFloatBE(offset?: number): number;

/\*\*

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

\*

\* ```js

\* import { Buffer } from '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.

\* ```

\* @since v0.11.15

\* @param [offset=0] Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 8`.

\*/

readDoubleLE(offset?: number): number;

/\*\*

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

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

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

\*

\* console.log(buf.readDoubleBE(0));

\* // Prints: 8.20788039913184e-304

\* ```

\* @since v0.11.15

\* @param [offset=0] Number of bytes to skip before starting to read. Must satisfy `0 <= offset <= buf.length - 8`.

\*/

readDoubleBE(offset?: number): number;

reverse(): this;

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from '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.

\* ```

\*

\* One convenient use of `buf.swap16()` is to perform a fast in-place conversion

\* between UTF-16 little-endian and UTF-16 big-endian:

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

\* const buf = Buffer.from('This is little-endian UTF-16', 'utf16le');

\* buf.swap16(); // Convert to big-endian UTF-16 text.

\* ```

\* @since v5.10.0

\* @return A reference to `buf`.

\*/

swap16(): Buffer;

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from '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.

\* ```

\* @since v5.10.0

\* @return A reference to `buf`.

\*/

swap32(): Buffer;

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from '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.

\* ```

\* @since v6.3.0

\* @return A reference to `buf`.

\*/

swap64(): Buffer;

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from '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>

\* ```

\* @since v0.5.0

\* @param value Number to be written to `buf`.

\* @param [offset=0] Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 1`.

\* @return `offset` plus the number of bytes written.

\*/

writeUInt8(value: number, offset?: number): number;

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

\* const buf = Buffer.allocUnsafe(4);

\*

\* buf.writeUInt16LE(0xdead, 0);

\* buf.writeUInt16LE(0xbeef, 2);

\*

\* console.log(buf);

\* // Prints: <Buffer ad de ef be>

\* ```

\* @since v0.5.5

\* @param value Number to be written to `buf`.

\* @param [offset=0] Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 2`.

\* @return `offset` plus the number of bytes written.

\*/

writeUInt16LE(value: number, offset?: number): number;

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

\* const buf = Buffer.allocUnsafe(4);

\*

\* buf.writeUInt16BE(0xdead, 0);

\* buf.writeUInt16BE(0xbeef, 2);

\*

\* console.log(buf);

\* // Prints: <Buffer de ad be ef>

\* ```

\* @since v0.5.5

\* @param value Number to be written to `buf`.

\* @param [offset=0] Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 2`.

\* @return `offset` plus the number of bytes written.

\*/

writeUInt16BE(value: number, offset?: number): number;

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

\* const buf = Buffer.allocUnsafe(4);

\*

\* buf.writeUInt32LE(0xfeedface, 0);

\*

\* console.log(buf);

\* // Prints: <Buffer ce fa ed fe>

\* ```

\* @since v0.5.5

\* @param value Number to be written to `buf`.

\* @param [offset=0] Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 4`.

\* @return `offset` plus the number of bytes written.

\*/

writeUInt32LE(value: number, offset?: number): number;

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

\* const buf = Buffer.allocUnsafe(4);

\*

\* buf.writeUInt32BE(0xfeedface, 0);

\*

\* console.log(buf);

\* // Prints: <Buffer fe ed fa ce>

\* ```

\* @since v0.5.5

\* @param value Number to be written to `buf`.

\* @param [offset=0] Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 4`.

\* @return `offset` plus the number of bytes written.

\*/

writeUInt32BE(value: number, offset?: number): number;

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

\* const buf = Buffer.allocUnsafe(2);

\*

\* buf.writeInt8(2, 0);

\* buf.writeInt8(-2, 1);

\*

\* console.log(buf);

\* // Prints: <Buffer 02 fe>

\* ```

\* @since v0.5.0

\* @param value Number to be written to `buf`.

\* @param [offset=0] Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 1`.

\* @return `offset` plus the number of bytes written.

\*/

writeInt8(value: number, offset?: number): number;

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

\* const buf = Buffer.allocUnsafe(2);

\*

\* buf.writeInt16LE(0x0304, 0);

\*

\* console.log(buf);

\* // Prints: <Buffer 04 03>

\* ```

\* @since v0.5.5

\* @param value Number to be written to `buf`.

\* @param [offset=0] Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 2`.

\* @return `offset` plus the number of bytes written.

\*/

writeInt16LE(value: number, offset?: number): number;

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

\* const buf = Buffer.allocUnsafe(2);

\*

\* buf.writeInt16BE(0x0102, 0);

\*

\* console.log(buf);

\* // Prints: <Buffer 01 02>

\* ```

\* @since v0.5.5

\* @param value Number to be written to `buf`.

\* @param [offset=0] Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 2`.

\* @return `offset` plus the number of bytes written.

\*/

writeInt16BE(value: number, offset?: number): number;

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

\* const buf = Buffer.allocUnsafe(4);

\*

\* buf.writeInt32LE(0x05060708, 0);

\*

\* console.log(buf);

\* // Prints: <Buffer 08 07 06 05>

\* ```

\* @since v0.5.5

\* @param value Number to be written to `buf`.

\* @param [offset=0] Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 4`.

\* @return `offset` plus the number of bytes written.

\*/

writeInt32LE(value: number, offset?: number): number;

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

\* const buf = Buffer.allocUnsafe(4);

\*

\* buf.writeInt32BE(0x01020304, 0);

\*

\* console.log(buf);

\* // Prints: <Buffer 01 02 03 04>

\* ```

\* @since v0.5.5

\* @param value Number to be written to `buf`.

\* @param [offset=0] Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 4`.

\* @return `offset` plus the number of bytes written.

\*/

writeInt32BE(value: number, offset?: number): number;

/\*\*

\* Writes `value` to `buf` at the specified `offset` as little-endian. Behavior is

\* undefined when `value` is anything other than a JavaScript number.

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

\* const buf = Buffer.allocUnsafe(4);

\*

\* buf.writeFloatLE(0xcafebabe, 0);

\*

\* console.log(buf);

\* // Prints: <Buffer bb fe 4a 4f>

\* ```

\* @since v0.11.15

\* @param value Number to be written to `buf`.

\* @param [offset=0] Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 4`.

\* @return `offset` plus the number of bytes written.

\*/

writeFloatLE(value: number, offset?: number): number;

/\*\*

\* Writes `value` to `buf` at the specified `offset` as big-endian. Behavior is

\* undefined when `value` is anything other than a JavaScript number.

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

\* const buf = Buffer.allocUnsafe(4);

\*

\* buf.writeFloatBE(0xcafebabe, 0);

\*

\* console.log(buf);

\* // Prints: <Buffer 4f 4a fe bb>

\* ```

\* @since v0.11.15

\* @param value Number to be written to `buf`.

\* @param [offset=0] Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 4`.

\* @return `offset` plus the number of bytes written.

\*/

writeFloatBE(value: number, offset?: number): number;

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

\* const buf = Buffer.allocUnsafe(8);

\*

\* buf.writeDoubleLE(123.456, 0);

\*

\* console.log(buf);

\* // Prints: <Buffer 77 be 9f 1a 2f dd 5e 40>

\* ```

\* @since v0.11.15

\* @param value Number to be written to `buf`.

\* @param [offset=0] Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 8`.

\* @return `offset` plus the number of bytes written.

\*/

writeDoubleLE(value: number, offset?: number): number;

/\*\*

\* 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.

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

\* const buf = Buffer.allocUnsafe(8);

\*

\* buf.writeDoubleBE(123.456, 0);

\*

\* console.log(buf);

\* // Prints: <Buffer 40 5e dd 2f 1a 9f be 77>

\* ```

\* @since v0.11.15

\* @param value Number to be written to `buf`.

\* @param [offset=0] Number of bytes to skip before starting to write. Must satisfy `0 <= offset <= buf.length - 8`.

\* @return `offset` plus the number of bytes written.

\*/

writeDoubleBE(value: number, offset?: number): number;

/\*\*

\* Fills `buf` with the specified `value`. If the `offset` and `end` are not given,

\* the entire `buf` will be filled:

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

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

\*

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

\*

\* console.log(b.toString());

\* // Prints: hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh

\* ```

\*

\* `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 &#x26; 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:

\*

\* ```js

\* import { Buffer } from '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>

\* ```

\*

\* If `value` contains invalid characters, it is truncated; if no valid

\* fill data remains, an exception is thrown:

\*

\* ```js

\* import { Buffer } from '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.

\* ```

\* @since v0.5.0

\* @param value The value with which to fill `buf`.

\* @param [offset=0] Number of bytes to skip before starting to fill `buf`.

\* @param [end=buf.length] Where to stop filling `buf` (not inclusive).

\* @param [encoding='utf8'] The encoding for `value` if `value` is a string.

\* @return A reference to `buf`.

\*/

fill(value: string | Uint8Array | number, offset?: number, end?: number, encoding?: BufferEncoding): this;

/\*\*

\* If `value` is:

\*

\* \* a string, `value` is interpreted according to the character encoding in`encoding`.

\* \* a `Buffer` or [`Uint8Array`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Uint8Array), `value` will be used in its entirety.

\* To compare a partial `Buffer`, use `buf.slice()`.

\* \* a number, `value` will be interpreted as an unsigned 8-bit integer

\* value between `0` and `255`.

\*

\* ```js

\* import { Buffer } from '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

\* ```

\*

\* 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()`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/String/indexOf).

\*

\* ```js

\* import { Buffer } from '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', []));

\* ```

\*

\* 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.

\* @since v1.5.0

\* @param value What to search for.

\* @param [byteOffset=0] Where to begin searching in `buf`. If negative, then offset is calculated from the end of `buf`.

\* @param [encoding='utf8'] 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`.

\* @return The index of the first occurrence of `value` in `buf`, or `-1` if `buf` does not contain `value`.

\*/

indexOf(value: string | number | Uint8Array, byteOffset?: number, encoding?: BufferEncoding): number;

/\*\*

\* Identical to `buf.indexOf()`, except the last occurrence of `value` is found

\* rather than the first occurrence.

\*

\* ```js

\* import { Buffer } from '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

\* ```

\*

\* 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()`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/String/lastIndexOf).

\*

\* ```js

\* import { Buffer } from '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', []));

\* ```

\*

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

\* @since v6.0.0

\* @param value What to search for.

\* @param [byteOffset=buf.length - 1] Where to begin searching in `buf`. If negative, then offset is calculated from the end of `buf`.

\* @param [encoding='utf8'] 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`.

\* @return The index of the last occurrence of `value` in `buf`, or `-1` if `buf` does not contain `value`.

\*/

lastIndexOf(value: string | number | Uint8Array, byteOffset?: number, encoding?: BufferEncoding): number;

/\*\*

\* Creates and returns an [iterator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration\_protocols) of `[index, byte]` pairs from the contents

\* of `buf`.

\*

\* ```js

\* import { Buffer } from '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]

\* ```

\* @since v1.1.0

\*/

entries(): IterableIterator<[number, number]>;

/\*\*

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

\*

\* ```js

\* import { Buffer } from '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

\* ```

\* @since v5.3.0

\* @param value What to search for.

\* @param [byteOffset=0] Where to begin searching in `buf`. If negative, then offset is calculated from the end of `buf`.

\* @param [encoding='utf8'] If `value` is a string, this is its encoding.

\* @return `true` if `value` was found in `buf`, `false` otherwise.

\*/

includes(value: string | number | Buffer, byteOffset?: number, encoding?: BufferEncoding): boolean;

/\*\*

\* Creates and returns an [iterator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration\_protocols) of `buf` keys (indices).

\*

\* ```js

\* import { Buffer } from 'buffer';

\*

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

\*

\* for (const key of buf.keys()) {

\* console.log(key);

\* }

\* // Prints:

\* // 0

\* // 1

\* // 2

\* // 3

\* // 4

\* // 5

\* ```

\* @since v1.1.0

\*/

keys(): IterableIterator<number>;

/\*\*

\* Creates and returns an [iterator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration\_protocols) for `buf` values (bytes). This function is

\* called automatically when a `Buffer` is used in a `for..of` statement.

\*

\* ```js

\* import { Buffer } from '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

\* ```

\* @since v1.1.0

\*/

values(): IterableIterator<number>;

}

var Buffer: BufferConstructor;

/\*\*

\* 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')`.\*\*

\* @since v15.13.0, v14.17.0

\* @deprecated Use `Buffer.from(data, 'base64')` instead.

\* @param data The Base64-encoded input string.

\*/

function atob(data: string): 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')`.\*\*

\* @since v15.13.0, v14.17.0

\* @deprecated Use `buf.toString('base64')` instead.

\* @param data An ASCII (Latin1) string.

\*/

function btoa(data: string): string;

}

}

declare module 'node:buffer' {

export \* from 'buffer';

}