/\*\*

\* A stream is an abstract interface for working with streaming data in Node.js.

\* The `stream` module provides an API for implementing the stream interface.

\*

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

\*

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

\*

\* To access the `stream` module:

\*

\* ```js

\* const stream = require('stream');

\* ```

\*

\* The `stream` module is useful for creating new types of stream instances. It is

\* usually not necessary to use the `stream` module to consume streams.

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

\*/

declare module 'stream' {

import { EventEmitter, Abortable } from 'node:events';

import \* as streamPromises from 'node:stream/promises';

import \* as streamConsumers from 'node:stream/consumers';

class internal extends EventEmitter {

pipe<T extends NodeJS.WritableStream>(

destination: T,

options?: {

end?: boolean | undefined;

}

): T;

}

namespace internal {

class Stream extends internal {

constructor(opts?: ReadableOptions);

}

interface StreamOptions<T extends Stream> extends Abortable {

emitClose?: boolean | undefined;

highWaterMark?: number | undefined;

objectMode?: boolean | undefined;

construct?(this: T, callback: (error?: Error | null) => void): void;

destroy?(this: T, error: Error | null, callback: (error: Error | null) => void): void;

autoDestroy?: boolean | undefined;

}

interface ReadableOptions extends StreamOptions<Readable> {

encoding?: BufferEncoding | undefined;

read?(this: Readable, size: number): void;

}

/\*\*

\* @since v0.9.4

\*/

class Readable extends Stream implements NodeJS.ReadableStream {

/\*\*

\* A utility method for creating Readable Streams out of iterators.

\*/

static from(iterable: Iterable<any> | AsyncIterable<any>, options?: ReadableOptions): Readable;

/\*\*

\* Returns whether the stream has been read from or cancelled.

\* @since v16.8.0

\*/

static isDisturbed(stream: Readable | NodeJS.ReadableStream): boolean;

/\*\*

\* Returns whether the stream was destroyed or errored before emitting `'end'`.

\* @since v16.8.0

\* @experimental

\*/

readonly readableAborted: boolean;

/\*\*

\* Is `true` if it is safe to call `readable.read()`, which means

\* the stream has not been destroyed or emitted `'error'` or `'end'`.

\* @since v11.4.0

\*/

readable: boolean;

/\*\*

\* Returns whether `'data'` has been emitted.

\* @since v16.7.0, v14.18.0

\* @experimental

\*/

readonly readableDidRead: boolean;

/\*\*

\* Getter for the property `encoding` of a given `Readable` stream. The `encoding`property can be set using the `readable.setEncoding()` method.

\* @since v12.7.0

\*/

readonly readableEncoding: BufferEncoding | null;

/\*\*

\* Becomes `true` when `'end'` event is emitted.

\* @since v12.9.0

\*/

readonly readableEnded: boolean;

/\*\*

\* This property reflects the current state of a `Readable` stream as described

\* in the `Three states` section.

\* @since v9.4.0

\*/

readonly readableFlowing: boolean | null;

/\*\*

\* Returns the value of `highWaterMark` passed when creating this `Readable`.

\* @since v9.3.0

\*/

readonly readableHighWaterMark: number;

/\*\*

\* This property contains the number of bytes (or objects) in the queue

\* ready to be read. The value provides introspection data regarding

\* the status of the `highWaterMark`.

\* @since v9.4.0

\*/

readonly readableLength: number;

/\*\*

\* Getter for the property `objectMode` of a given `Readable` stream.

\* @since v12.3.0

\*/

readonly readableObjectMode: boolean;

/\*\*

\* Is `true` after `readable.destroy()` has been called.

\* @since v8.0.0

\*/

destroyed: boolean;

constructor(opts?: ReadableOptions);

\_construct?(callback: (error?: Error | null) => void): void;

\_read(size: number): void;

/\*\*

\* The `readable.read()` method pulls some data out of the internal buffer and

\* returns it. If no data available to be read, `null` is returned. By default,

\* the data will be returned as a `Buffer` object unless an encoding has been

\* specified using the `readable.setEncoding()` method or the stream is operating

\* in object mode.

\*

\* The optional `size` argument specifies a specific number of bytes to read. If`size` bytes are not available to be read, `null` will be returned \_unless\_the stream has ended, in which

\* case all of the data remaining in the internal

\* buffer will be returned.

\*

\* If the `size` argument is not specified, all of the data contained in the

\* internal buffer will be returned.

\*

\* The `size` argument must be less than or equal to 1 GiB.

\*

\* The `readable.read()` method should only be called on `Readable` streams

\* operating in paused mode. In flowing mode, `readable.read()` is called

\* automatically until the internal buffer is fully drained.

\*

\* ```js

\* const readable = getReadableStreamSomehow();

\*

\* // 'readable' may be triggered multiple times as data is buffered in

\* readable.on('readable', () => {

\* let chunk;

\* console.log('Stream is readable (new data received in buffer)');

\* // Use a loop to make sure we read all currently available data

\* while (null !== (chunk = readable.read())) {

\* console.log(`Read ${chunk.length} bytes of data...`);

\* }

\* });

\*

\* // 'end' will be triggered once when there is no more data available

\* readable.on('end', () => {

\* console.log('Reached end of stream.');

\* });

\* ```

\*

\* Each call to `readable.read()` returns a chunk of data, or `null`. The chunks

\* are not concatenated. A `while` loop is necessary to consume all data

\* currently in the buffer. When reading a large file `.read()` may return `null`,

\* having consumed all buffered content so far, but there is still more data to

\* come not yet buffered. In this case a new `'readable'` event will be emitted

\* when there is more data in the buffer. Finally the `'end'` event will be

\* emitted when there is no more data to come.

\*

\* Therefore to read a file's whole contents from a `readable`, it is necessary

\* to collect chunks across multiple `'readable'` events:

\*

\* ```js

\* const chunks = [];

\*

\* readable.on('readable', () => {

\* let chunk;

\* while (null !== (chunk = readable.read())) {

\* chunks.push(chunk);

\* }

\* });

\*

\* readable.on('end', () => {

\* const content = chunks.join('');

\* });

\* ```

\*

\* A `Readable` stream in object mode will always return a single item from

\* a call to `readable.read(size)`, regardless of the value of the`size` argument.

\*

\* If the `readable.read()` method returns a chunk of data, a `'data'` event will

\* also be emitted.

\*

\* Calling {@link read} after the `'end'` event has

\* been emitted will return `null`. No runtime error will be raised.

\* @since v0.9.4

\* @param size Optional argument to specify how much data to read.

\*/

read(size?: number): any;

/\*\*

\* The `readable.setEncoding()` method sets the character encoding for

\* data read from the `Readable` stream.

\*

\* By default, no encoding is assigned and stream data will be returned as`Buffer` objects. Setting an encoding causes the stream data

\* to be returned as strings of the specified encoding rather than as `Buffer`objects. For instance, calling `readable.setEncoding('utf8')` will cause the

\* output data to be interpreted as UTF-8 data, and passed as strings. Calling`readable.setEncoding('hex')` will cause the data to be encoded in hexadecimal

\* string format.

\*

\* The `Readable` stream will properly handle multi-byte characters delivered

\* through the stream that would otherwise become improperly decoded if simply

\* pulled from the stream as `Buffer` objects.

\*

\* ```js

\* const readable = getReadableStreamSomehow();

\* readable.setEncoding('utf8');

\* readable.on('data', (chunk) => {

\* assert.equal(typeof chunk, 'string');

\* console.log('Got %d characters of string data:', chunk.length);

\* });

\* ```

\* @since v0.9.4

\* @param encoding The encoding to use.

\*/

setEncoding(encoding: BufferEncoding): this;

/\*\*

\* The `readable.pause()` method will cause a stream in flowing mode to stop

\* emitting `'data'` events, switching out of flowing mode. Any data that

\* becomes available will remain in the internal buffer.

\*

\* ```js

\* const readable = getReadableStreamSomehow();

\* readable.on('data', (chunk) => {

\* console.log(`Received ${chunk.length} bytes of data.`);

\* readable.pause();

\* console.log('There will be no additional data for 1 second.');

\* setTimeout(() => {

\* console.log('Now data will start flowing again.');

\* readable.resume();

\* }, 1000);

\* });

\* ```

\*

\* The `readable.pause()` method has no effect if there is a `'readable'`event listener.

\* @since v0.9.4

\*/

pause(): this;

/\*\*

\* The `readable.resume()` method causes an explicitly paused `Readable` stream to

\* resume emitting `'data'` events, switching the stream into flowing mode.

\*

\* The `readable.resume()` method can be used to fully consume the data from a

\* stream without actually processing any of that data:

\*

\* ```js

\* getReadableStreamSomehow()

\* .resume()

\* .on('end', () => {

\* console.log('Reached the end, but did not read anything.');

\* });

\* ```

\*

\* The `readable.resume()` method has no effect if there is a `'readable'`event listener.

\* @since v0.9.4

\*/

resume(): this;

/\*\*

\* The `readable.isPaused()` method returns the current operating state of the`Readable`. This is used primarily by the mechanism that underlies the`readable.pipe()` method. In most

\* typical cases, there will be no reason to

\* use this method directly.

\*

\* ```js

\* const readable = new stream.Readable();

\*

\* readable.isPaused(); // === false

\* readable.pause();

\* readable.isPaused(); // === true

\* readable.resume();

\* readable.isPaused(); // === false

\* ```

\* @since v0.11.14

\*/

isPaused(): boolean;

/\*\*

\* The `readable.unpipe()` method detaches a `Writable` stream previously attached

\* using the {@link pipe} method.

\*

\* If the `destination` is not specified, then \_all\_ pipes are detached.

\*

\* If the `destination` is specified, but no pipe is set up for it, then

\* the method does nothing.

\*

\* ```js

\* const fs = require('fs');

\* const readable = getReadableStreamSomehow();

\* const writable = fs.createWriteStream('file.txt');

\* // All the data from readable goes into 'file.txt',

\* // but only for the first second.

\* readable.pipe(writable);

\* setTimeout(() => {

\* console.log('Stop writing to file.txt.');

\* readable.unpipe(writable);

\* console.log('Manually close the file stream.');

\* writable.end();

\* }, 1000);

\* ```

\* @since v0.9.4

\* @param destination Optional specific stream to unpipe

\*/

unpipe(destination?: NodeJS.WritableStream): this;

/\*\*

\* Passing `chunk` as `null` signals the end of the stream (EOF) and behaves the

\* same as `readable.push(null)`, after which no more data can be written. The EOF

\* signal is put at the end of the buffer and any buffered data will still be

\* flushed.

\*

\* The `readable.unshift()` method pushes a chunk of data back into the internal

\* buffer. This is useful in certain situations where a stream is being consumed by

\* code that needs to "un-consume" some amount of data that it has optimistically

\* pulled out of the source, so that the data can be passed on to some other party.

\*

\* The `stream.unshift(chunk)` method cannot be called after the `'end'` event

\* has been emitted or a runtime error will be thrown.

\*

\* Developers using `stream.unshift()` often should consider switching to

\* use of a `Transform` stream instead. See the `API for stream implementers` section for more information.

\*

\* ```js

\* // Pull off a header delimited by \n\n.

\* // Use unshift() if we get too much.

\* // Call the callback with (error, header, stream).

\* const { StringDecoder } = require('string\_decoder');

\* function parseHeader(stream, callback) {

\* stream.on('error', callback);

\* stream.on('readable', onReadable);

\* const decoder = new StringDecoder('utf8');

\* let header = '';

\* function onReadable() {

\* let chunk;

\* while (null !== (chunk = stream.read())) {

\* const str = decoder.write(chunk);

\* if (str.match(/\n\n/)) {

\* // Found the header boundary.

\* const split = str.split(/\n\n/);

\* header += split.shift();

\* const remaining = split.join('\n\n');

\* const buf = Buffer.from(remaining, 'utf8');

\* stream.removeListener('error', callback);

\* // Remove the 'readable' listener before unshifting.

\* stream.removeListener('readable', onReadable);

\* if (buf.length)

\* stream.unshift(buf);

\* // Now the body of the message can be read from the stream.

\* callback(null, header, stream);

\* } else {

\* // Still reading the header.

\* header += str;

\* }

\* }

\* }

\* }

\* ```

\*

\* Unlike {@link push}, `stream.unshift(chunk)` will not

\* end the reading process by resetting the internal reading state of the stream.

\* This can cause unexpected results if `readable.unshift()` is called during a

\* read (i.e. from within a {@link \_read} implementation on a

\* custom stream). Following the call to `readable.unshift()` with an immediate {@link push} will reset the reading state appropriately,

\* however it is best to simply avoid calling `readable.unshift()` while in the

\* process of performing a read.

\* @since v0.9.11

\* @param chunk Chunk of data to unshift onto the read queue. For streams not operating in object mode, `chunk` must be a string, `Buffer`, `Uint8Array` or `null`. For object mode

\* streams, `chunk` may be any JavaScript value.

\* @param encoding Encoding of string chunks. Must be a valid `Buffer` encoding, such as `'utf8'` or `'ascii'`.

\*/

unshift(chunk: any, encoding?: BufferEncoding): void;

/\*\*

\* Prior to Node.js 0.10, streams did not implement the entire `stream` module API

\* as it is currently defined. (See `Compatibility` for more information.)

\*

\* When using an older Node.js library that emits `'data'` events and has a {@link pause} method that is advisory only, the`readable.wrap()` method can be used to create a `Readable`

\* stream that uses

\* the old stream as its data source.

\*

\* It will rarely be necessary to use `readable.wrap()` but the method has been

\* provided as a convenience for interacting with older Node.js applications and

\* libraries.

\*

\* ```js

\* const { OldReader } = require('./old-api-module.js');

\* const { Readable } = require('stream');

\* const oreader = new OldReader();

\* const myReader = new Readable().wrap(oreader);

\*

\* myReader.on('readable', () => {

\* myReader.read(); // etc.

\* });

\* ```

\* @since v0.9.4

\* @param stream An "old style" readable stream

\*/

wrap(stream: NodeJS.ReadableStream): this;

push(chunk: any, encoding?: BufferEncoding): boolean;

\_destroy(error: Error | null, callback: (error?: Error | null) => void): void;

/\*\*

\* Destroy the stream. Optionally emit an `'error'` event, and emit a `'close'`event (unless `emitClose` is set to `false`). After this call, the readable

\* stream will release any internal resources and subsequent calls to `push()`will be ignored.

\*

\* Once `destroy()` has been called any further calls will be a no-op and no

\* further errors except from `\_destroy()` may be emitted as `'error'`.

\*

\* Implementors should not override this method, but instead implement `readable.\_destroy()`.

\* @since v8.0.0

\* @param error Error which will be passed as payload in `'error'` event

\*/

destroy(error?: Error): void;

/\*\*

\* Event emitter

\* The defined events on documents including:

\* 1. close

\* 2. data

\* 3. end

\* 4. error

\* 5. pause

\* 6. readable

\* 7. resume

\*/

addListener(event: 'close', listener: () => void): this;

addListener(event: 'data', listener: (chunk: any) => void): this;

addListener(event: 'end', listener: () => void): this;

addListener(event: 'error', listener: (err: Error) => void): this;

addListener(event: 'pause', listener: () => void): this;

addListener(event: 'readable', listener: () => void): this;

addListener(event: 'resume', listener: () => void): this;

addListener(event: string | symbol, listener: (...args: any[]) => void): this;

emit(event: 'close'): boolean;

emit(event: 'data', chunk: any): boolean;

emit(event: 'end'): boolean;

emit(event: 'error', err: Error): boolean;

emit(event: 'pause'): boolean;

emit(event: 'readable'): boolean;

emit(event: 'resume'): boolean;

emit(event: string | symbol, ...args: any[]): boolean;

on(event: 'close', listener: () => void): this;

on(event: 'data', listener: (chunk: any) => void): this;

on(event: 'end', listener: () => void): this;

on(event: 'error', listener: (err: Error) => void): this;

on(event: 'pause', listener: () => void): this;

on(event: 'readable', listener: () => void): this;

on(event: 'resume', listener: () => void): this;

on(event: string | symbol, listener: (...args: any[]) => void): this;

once(event: 'close', listener: () => void): this;

once(event: 'data', listener: (chunk: any) => void): this;

once(event: 'end', listener: () => void): this;

once(event: 'error', listener: (err: Error) => void): this;

once(event: 'pause', listener: () => void): this;

once(event: 'readable', listener: () => void): this;

once(event: 'resume', listener: () => void): this;

once(event: string | symbol, listener: (...args: any[]) => void): this;

prependListener(event: 'close', listener: () => void): this;

prependListener(event: 'data', listener: (chunk: any) => void): this;

prependListener(event: 'end', listener: () => void): this;

prependListener(event: 'error', listener: (err: Error) => void): this;

prependListener(event: 'pause', listener: () => void): this;

prependListener(event: 'readable', listener: () => void): this;

prependListener(event: 'resume', listener: () => void): this;

prependListener(event: string | symbol, listener: (...args: any[]) => void): this;

prependOnceListener(event: 'close', listener: () => void): this;

prependOnceListener(event: 'data', listener: (chunk: any) => void): this;

prependOnceListener(event: 'end', listener: () => void): this;

prependOnceListener(event: 'error', listener: (err: Error) => void): this;

prependOnceListener(event: 'pause', listener: () => void): this;

prependOnceListener(event: 'readable', listener: () => void): this;

prependOnceListener(event: 'resume', listener: () => void): this;

prependOnceListener(event: string | symbol, listener: (...args: any[]) => void): this;

removeListener(event: 'close', listener: () => void): this;

removeListener(event: 'data', listener: (chunk: any) => void): this;

removeListener(event: 'end', listener: () => void): this;

removeListener(event: 'error', listener: (err: Error) => void): this;

removeListener(event: 'pause', listener: () => void): this;

removeListener(event: 'readable', listener: () => void): this;

removeListener(event: 'resume', listener: () => void): this;

removeListener(event: string | symbol, listener: (...args: any[]) => void): this;

[Symbol.asyncIterator](): AsyncIterableIterator<any>;

}

interface WritableOptions extends StreamOptions<Writable> {

decodeStrings?: boolean | undefined;

defaultEncoding?: BufferEncoding | undefined;

write?(this: Writable, chunk: any, encoding: BufferEncoding, callback: (error?: Error | null) => void): void;

writev?(

this: Writable,

chunks: Array<{

chunk: any;

encoding: BufferEncoding;

}>,

callback: (error?: Error | null) => void

): void;

final?(this: Writable, callback: (error?: Error | null) => void): void;

}

/\*\*

\* @since v0.9.4

\*/

class Writable extends Stream implements NodeJS.WritableStream {

/\*\*

\* Is `true` if it is safe to call `writable.write()`, which means

\* the stream has not been destroyed, errored or ended.

\* @since v11.4.0

\*/

readonly writable: boolean;

/\*\*

\* Is `true` after `writable.end()` has been called. This property

\* does not indicate whether the data has been flushed, for this use `writable.writableFinished` instead.

\* @since v12.9.0

\*/

readonly writableEnded: boolean;

/\*\*

\* Is set to `true` immediately before the `'finish'` event is emitted.

\* @since v12.6.0

\*/

readonly writableFinished: boolean;

/\*\*

\* Return the value of `highWaterMark` passed when creating this `Writable`.

\* @since v9.3.0

\*/

readonly writableHighWaterMark: number;

/\*\*

\* This property contains the number of bytes (or objects) in the queue

\* ready to be written. The value provides introspection data regarding

\* the status of the `highWaterMark`.

\* @since v9.4.0

\*/

readonly writableLength: number;

/\*\*

\* Getter for the property `objectMode` of a given `Writable` stream.

\* @since v12.3.0

\*/

readonly writableObjectMode: boolean;

/\*\*

\* Number of times `writable.uncork()` needs to be

\* called in order to fully uncork the stream.

\* @since v13.2.0, v12.16.0

\*/

readonly writableCorked: number;

/\*\*

\* Is `true` after `writable.destroy()` has been called.

\* @since v8.0.0

\*/

destroyed: boolean;

constructor(opts?: WritableOptions);

\_write(chunk: any, encoding: BufferEncoding, callback: (error?: Error | null) => void): void;

\_writev?(

chunks: Array<{

chunk: any;

encoding: BufferEncoding;

}>,

callback: (error?: Error | null) => void

): void;

\_construct?(callback: (error?: Error | null) => void): void;

\_destroy(error: Error | null, callback: (error?: Error | null) => void): void;

\_final(callback: (error?: Error | null) => void): void;

/\*\*

\* The `writable.write()` method writes some data to the stream, and calls the

\* supplied `callback` once the data has been fully handled. If an error

\* occurs, the `callback` will be called with the error as its

\* first argument. The `callback` is called asynchronously and before `'error'` is

\* emitted.

\*

\* The return value is `true` if the internal buffer is less than the`highWaterMark` configured when the stream was created after admitting `chunk`.

\* If `false` is returned, further attempts to write data to the stream should

\* stop until the `'drain'` event is emitted.

\*

\* While a stream is not draining, calls to `write()` will buffer `chunk`, and

\* return false. Once all currently buffered chunks are drained (accepted for

\* delivery by the operating system), the `'drain'` event will be emitted.

\* It is recommended that once `write()` returns false, no more chunks be written

\* until the `'drain'` event is emitted. While calling `write()` on a stream that

\* is not draining is allowed, Node.js will buffer all written chunks until

\* maximum memory usage occurs, at which point it will abort unconditionally.

\* Even before it aborts, high memory usage will cause poor garbage collector

\* performance and high RSS (which is not typically released back to the system,

\* even after the memory is no longer required). Since TCP sockets may never

\* drain if the remote peer does not read the data, writing a socket that is

\* not draining may lead to a remotely exploitable vulnerability.

\*

\* Writing data while the stream is not draining is particularly

\* problematic for a `Transform`, because the `Transform` streams are paused

\* by default until they are piped or a `'data'` or `'readable'` event handler

\* is added.

\*

\* If the data to be written can be generated or fetched on demand, it is

\* recommended to encapsulate the logic into a `Readable` and use {@link pipe}. However, if calling `write()` is preferred, it is

\* possible to respect backpressure and avoid memory issues using the `'drain'` event:

\*

\* ```js

\* function write(data, cb) {

\* if (!stream.write(data)) {

\* stream.once('drain', cb);

\* } else {

\* process.nextTick(cb);

\* }

\* }

\*

\* // Wait for cb to be called before doing any other write.

\* write('hello', () => {

\* console.log('Write completed, do more writes now.');

\* });

\* ```

\*

\* A `Writable` stream in object mode will always ignore the `encoding` argument.

\* @since v0.9.4

\* @param chunk Optional data to write. For streams not operating in object mode, `chunk` must be a string, `Buffer` or `Uint8Array`. For object mode streams, `chunk` may be any

\* JavaScript value other than `null`.

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

\* @param callback Callback for when this chunk of data is flushed.

\* @return `false` if the stream wishes for the calling code to wait for the `'drain'` event to be emitted before continuing to write additional data; otherwise `true`.

\*/

write(chunk: any, callback?: (error: Error | null | undefined) => void): boolean;

write(chunk: any, encoding: BufferEncoding, callback?: (error: Error | null | undefined) => void): boolean;

/\*\*

\* The `writable.setDefaultEncoding()` method sets the default `encoding` for a `Writable` stream.

\* @since v0.11.15

\* @param encoding The new default encoding

\*/

setDefaultEncoding(encoding: BufferEncoding): this;

/\*\*

\* Calling the `writable.end()` method signals that no more data will be written

\* to the `Writable`. The optional `chunk` and `encoding` arguments allow one

\* final additional chunk of data to be written immediately before closing the

\* stream.

\*

\* Calling the {@link write} method after calling {@link end} will raise an error.

\*

\* ```js

\* // Write 'hello, ' and then end with 'world!'.

\* const fs = require('fs');

\* const file = fs.createWriteStream('example.txt');

\* file.write('hello, ');

\* file.end('world!');

\* // Writing more now is not allowed!

\* ```

\* @since v0.9.4

\* @param chunk Optional data to write. For streams not operating in object mode, `chunk` must be a string, `Buffer` or `Uint8Array`. For object mode streams, `chunk` may be any

\* JavaScript value other than `null`.

\* @param encoding The encoding if `chunk` is a string

\* @param callback Callback for when the stream is finished.

\*/

end(cb?: () => void): void;

end(chunk: any, cb?: () => void): void;

end(chunk: any, encoding: BufferEncoding, cb?: () => void): void;

/\*\*

\* The `writable.cork()` method forces all written data to be buffered in memory.

\* The buffered data will be flushed when either the {@link uncork} or {@link end} methods are called.

\*

\* The primary intent of `writable.cork()` is to accommodate a situation in which

\* several small chunks are written to the stream in rapid succession. Instead of

\* immediately forwarding them to the underlying destination, `writable.cork()`buffers all the chunks until `writable.uncork()` is called, which will pass them

\* all to `writable.\_writev()`, if present. This prevents a head-of-line blocking

\* situation where data is being buffered while waiting for the first small chunk

\* to be processed. However, use of `writable.cork()` without implementing`writable.\_writev()` may have an adverse effect on throughput.

\*

\* See also: `writable.uncork()`, `writable.\_writev()`.

\* @since v0.11.2

\*/

cork(): void;

/\*\*

\* The `writable.uncork()` method flushes all data buffered since {@link cork} was called.

\*

\* When using `writable.cork()` and `writable.uncork()` to manage the buffering

\* of writes to a stream, it is recommended that calls to `writable.uncork()` be

\* deferred using `process.nextTick()`. Doing so allows batching of all`writable.write()` calls that occur within a given Node.js event loop phase.

\*

\* ```js

\* stream.cork();

\* stream.write('some ');

\* stream.write('data ');

\* process.nextTick(() => stream.uncork());

\* ```

\*

\* If the `writable.cork()` method is called multiple times on a stream, the

\* same number of calls to `writable.uncork()` must be called to flush the buffered

\* data.

\*

\* ```js

\* stream.cork();

\* stream.write('some ');

\* stream.cork();

\* stream.write('data ');

\* process.nextTick(() => {

\* stream.uncork();

\* // The data will not be flushed until uncork() is called a second time.

\* stream.uncork();

\* });

\* ```

\*

\* See also: `writable.cork()`.

\* @since v0.11.2

\*/

uncork(): void;

/\*\*

\* Destroy the stream. Optionally emit an `'error'` event, and emit a `'close'`event (unless `emitClose` is set to `false`). After this call, the writable

\* stream has ended and subsequent calls to `write()` or `end()` will result in

\* an `ERR\_STREAM\_DESTROYED` error.

\* This is a destructive and immediate way to destroy a stream. Previous calls to`write()` may not have drained, and may trigger an `ERR\_STREAM\_DESTROYED` error.

\* Use `end()` instead of destroy if data should flush before close, or wait for

\* the `'drain'` event before destroying the stream.

\*

\* Once `destroy()` has been called any further calls will be a no-op and no

\* further errors except from `\_destroy()` may be emitted as `'error'`.

\*

\* Implementors should not override this method,

\* but instead implement `writable.\_destroy()`.

\* @since v8.0.0

\* @param error Optional, an error to emit with `'error'` event.

\*/

destroy(error?: Error): void;

/\*\*

\* Event emitter

\* The defined events on documents including:

\* 1. close

\* 2. drain

\* 3. error

\* 4. finish

\* 5. pipe

\* 6. unpipe

\*/

addListener(event: 'close', listener: () => void): this;

addListener(event: 'drain', listener: () => void): this;

addListener(event: 'error', listener: (err: Error) => void): this;

addListener(event: 'finish', listener: () => void): this;

addListener(event: 'pipe', listener: (src: Readable) => void): this;

addListener(event: 'unpipe', listener: (src: Readable) => void): this;

addListener(event: string | symbol, listener: (...args: any[]) => void): this;

emit(event: 'close'): boolean;

emit(event: 'drain'): boolean;

emit(event: 'error', err: Error): boolean;

emit(event: 'finish'): boolean;

emit(event: 'pipe', src: Readable): boolean;

emit(event: 'unpipe', src: Readable): boolean;

emit(event: string | symbol, ...args: any[]): boolean;

on(event: 'close', listener: () => void): this;

on(event: 'drain', listener: () => void): this;

on(event: 'error', listener: (err: Error) => void): this;

on(event: 'finish', listener: () => void): this;

on(event: 'pipe', listener: (src: Readable) => void): this;

on(event: 'unpipe', listener: (src: Readable) => void): this;

on(event: string | symbol, listener: (...args: any[]) => void): this;

once(event: 'close', listener: () => void): this;

once(event: 'drain', listener: () => void): this;

once(event: 'error', listener: (err: Error) => void): this;

once(event: 'finish', listener: () => void): this;

once(event: 'pipe', listener: (src: Readable) => void): this;

once(event: 'unpipe', listener: (src: Readable) => void): this;

once(event: string | symbol, listener: (...args: any[]) => void): this;

prependListener(event: 'close', listener: () => void): this;

prependListener(event: 'drain', listener: () => void): this;

prependListener(event: 'error', listener: (err: Error) => void): this;

prependListener(event: 'finish', listener: () => void): this;

prependListener(event: 'pipe', listener: (src: Readable) => void): this;

prependListener(event: 'unpipe', listener: (src: Readable) => void): this;

prependListener(event: string | symbol, listener: (...args: any[]) => void): this;

prependOnceListener(event: 'close', listener: () => void): this;

prependOnceListener(event: 'drain', listener: () => void): this;

prependOnceListener(event: 'error', listener: (err: Error) => void): this;

prependOnceListener(event: 'finish', listener: () => void): this;

prependOnceListener(event: 'pipe', listener: (src: Readable) => void): this;

prependOnceListener(event: 'unpipe', listener: (src: Readable) => void): this;

prependOnceListener(event: string | symbol, listener: (...args: any[]) => void): this;

removeListener(event: 'close', listener: () => void): this;

removeListener(event: 'drain', listener: () => void): this;

removeListener(event: 'error', listener: (err: Error) => void): this;

removeListener(event: 'finish', listener: () => void): this;

removeListener(event: 'pipe', listener: (src: Readable) => void): this;

removeListener(event: 'unpipe', listener: (src: Readable) => void): this;

removeListener(event: string | symbol, listener: (...args: any[]) => void): this;

}

interface DuplexOptions extends ReadableOptions, WritableOptions {

allowHalfOpen?: boolean | undefined;

readableObjectMode?: boolean | undefined;

writableObjectMode?: boolean | undefined;

readableHighWaterMark?: number | undefined;

writableHighWaterMark?: number | undefined;

writableCorked?: number | undefined;

construct?(this: Duplex, callback: (error?: Error | null) => void): void;

read?(this: Duplex, size: number): void;

write?(this: Duplex, chunk: any, encoding: BufferEncoding, callback: (error?: Error | null) => void): void;

writev?(

this: Duplex,

chunks: Array<{

chunk: any;

encoding: BufferEncoding;

}>,

callback: (error?: Error | null) => void

): void;

final?(this: Duplex, callback: (error?: Error | null) => void): void;

destroy?(this: Duplex, error: Error | null, callback: (error: Error | null) => void): void;

}

/\*\*

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

\*

\* Examples of `Duplex` streams include:

\*

\* \* `TCP sockets`

\* \* `zlib streams`

\* \* `crypto streams`

\* @since v0.9.4

\*/

class Duplex extends Readable implements Writable {

readonly writable: boolean;

readonly writableEnded: boolean;

readonly writableFinished: boolean;

readonly writableHighWaterMark: number;

readonly writableLength: number;

readonly writableObjectMode: boolean;

readonly writableCorked: number;

/\*\*

\* If `false` then the stream will automatically end the writable side when the

\* readable side ends. Set initially by the `allowHalfOpen` constructor option,

\* which defaults to `false`.

\*

\* This can be changed manually to change the half-open behavior of an existing`Duplex` stream instance, but must be changed before the `'end'` event is

\* emitted.

\* @since v0.9.4

\*/

allowHalfOpen: boolean;

constructor(opts?: DuplexOptions);

/\*\*

\* A utility method for creating duplex streams.

\*

\* - `Stream` converts writable stream into writable `Duplex` and readable stream

\* to `Duplex`.

\* - `Blob` converts into readable `Duplex`.

\* - `string` converts into readable `Duplex`.

\* - `ArrayBuffer` converts into readable `Duplex`.

\* - `AsyncIterable` converts into a readable `Duplex`. Cannot yield `null`.

\* - `AsyncGeneratorFunction` converts into a readable/writable transform

\* `Duplex`. Must take a source `AsyncIterable` as first parameter. Cannot yield

\* `null`.

\* - `AsyncFunction` converts into a writable `Duplex`. Must return

\* either `null` or `undefined`

\* - `Object ({ writable, readable })` converts `readable` and

\* `writable` into `Stream` and then combines them into `Duplex` where the

\* `Duplex` will write to the `writable` and read from the `readable`.

\* - `Promise` converts into readable `Duplex`. Value `null` is ignored.

\*

\* @since v16.8.0

\*/

static from(src: Stream | Blob | ArrayBuffer | string | Iterable<any> | AsyncIterable<any> | AsyncGeneratorFunction | Promise<any> | Object): Duplex;

\_write(chunk: any, encoding: BufferEncoding, callback: (error?: Error | null) => void): void;

\_writev?(

chunks: Array<{

chunk: any;

encoding: BufferEncoding;

}>,

callback: (error?: Error | null) => void

): void;

\_destroy(error: Error | null, callback: (error: Error | null) => void): void;

\_final(callback: (error?: Error | null) => void): void;

write(chunk: any, encoding?: BufferEncoding, cb?: (error: Error | null | undefined) => void): boolean;

write(chunk: any, cb?: (error: Error | null | undefined) => void): boolean;

setDefaultEncoding(encoding: BufferEncoding): this;

end(cb?: () => void): void;

end(chunk: any, cb?: () => void): void;

end(chunk: any, encoding?: BufferEncoding, cb?: () => void): void;

cork(): void;

uncork(): void;

}

type TransformCallback = (error?: Error | null, data?: any) => void;

interface TransformOptions extends DuplexOptions {

construct?(this: Transform, callback: (error?: Error | null) => void): void;

read?(this: Transform, size: number): void;

write?(this: Transform, chunk: any, encoding: BufferEncoding, callback: (error?: Error | null) => void): void;

writev?(

this: Transform,

chunks: Array<{

chunk: any;

encoding: BufferEncoding;

}>,

callback: (error?: Error | null) => void

): void;

final?(this: Transform, callback: (error?: Error | null) => void): void;

destroy?(this: Transform, error: Error | null, callback: (error: Error | null) => void): void;

transform?(this: Transform, chunk: any, encoding: BufferEncoding, callback: TransformCallback): void;

flush?(this: Transform, callback: TransformCallback): void;

}

/\*\*

\* Transform streams are `Duplex` streams where the output is in some way

\* related to the input. Like all `Duplex` streams, `Transform` streams

\* implement both the `Readable` and `Writable` interfaces.

\*

\* Examples of `Transform` streams include:

\*

\* \* `zlib streams`

\* \* `crypto streams`

\* @since v0.9.4

\*/

class Transform extends Duplex {

constructor(opts?: TransformOptions);

\_transform(chunk: any, encoding: BufferEncoding, callback: TransformCallback): void;

\_flush(callback: TransformCallback): void;

}

/\*\*

\* The `stream.PassThrough` class is a trivial implementation of a `Transform` stream that simply passes the input bytes across to the output. Its purpose is

\* primarily for examples and testing, but there are some use cases where`stream.PassThrough` is useful as a building block for novel sorts of streams.

\*/

class PassThrough extends Transform {}

/\*\*

\* Attaches an AbortSignal to a readable or writeable stream. This lets code

\* control stream destruction using an `AbortController`.

\*

\* Calling `abort` on the `AbortController` corresponding to the passed`AbortSignal` will behave the same way as calling `.destroy(new AbortError())`on the stream.

\*

\* ```js

\* const fs = require('fs');

\*

\* const controller = new AbortController();

\* const read = addAbortSignal(

\* controller.signal,

\* fs.createReadStream(('object.json'))

\* );

\* // Later, abort the operation closing the stream

\* controller.abort();

\* ```

\*

\* Or using an `AbortSignal` with a readable stream as an async iterable:

\*

\* ```js

\* const controller = new AbortController();

\* setTimeout(() => controller.abort(), 10\_000); // set a timeout

\* const stream = addAbortSignal(

\* controller.signal,

\* fs.createReadStream(('object.json'))

\* );

\* (async () => {

\* try {

\* for await (const chunk of stream) {

\* await process(chunk);

\* }

\* } catch (e) {

\* if (e.name === 'AbortError') {

\* // The operation was cancelled

\* } else {

\* throw e;

\* }

\* }

\* })();

\* ```

\* @since v15.4.0

\* @param signal A signal representing possible cancellation

\* @param stream a stream to attach a signal to

\*/

function addAbortSignal<T extends Stream>(signal: AbortSignal, stream: T): T;

interface FinishedOptions extends Abortable {

error?: boolean | undefined;

readable?: boolean | undefined;

writable?: boolean | undefined;

}

/\*\*

\* A function to get notified when a stream is no longer readable, writable

\* or has experienced an error or a premature close event.

\*

\* ```js

\* const { finished } = require('stream');

\*

\* const rs = fs.createReadStream('archive.tar');

\*

\* finished(rs, (err) => {

\* if (err) {

\* console.error('Stream failed.', err);

\* } else {

\* console.log('Stream is done reading.');

\* }

\* });

\*

\* rs.resume(); // Drain the stream.

\* ```

\*

\* Especially useful in error handling scenarios where a stream is destroyed

\* prematurely (like an aborted HTTP request), and will not emit `'end'`or `'finish'`.

\*

\* The `finished` API provides promise version:

\*

\* ```js

\* const { finished } = require('stream/promises');

\*

\* const rs = fs.createReadStream('archive.tar');

\*

\* async function run() {

\* await finished(rs);

\* console.log('Stream is done reading.');

\* }

\*

\* run().catch(console.error);

\* rs.resume(); // Drain the stream.

\* ```

\*

\* `stream.finished()` leaves dangling event listeners (in particular`'error'`, `'end'`, `'finish'` and `'close'`) after `callback` has been

\* invoked. The reason for this is so that unexpected `'error'` events (due to

\* incorrect stream implementations) do not cause unexpected crashes.

\* If this is unwanted behavior then the returned cleanup function needs to be

\* invoked in the callback:

\*

\* ```js

\* const cleanup = finished(rs, (err) => {

\* cleanup();

\* // ...

\* });

\* ```

\* @since v10.0.0

\* @param stream A readable and/or writable stream.

\* @param callback A callback function that takes an optional error argument.

\* @return A cleanup function which removes all registered listeners.

\*/

function finished(stream: NodeJS.ReadableStream | NodeJS.WritableStream | NodeJS.ReadWriteStream, options: FinishedOptions, callback: (err?: NodeJS.ErrnoException | null) => void): () => void;

function finished(stream: NodeJS.ReadableStream | NodeJS.WritableStream | NodeJS.ReadWriteStream, callback: (err?: NodeJS.ErrnoException | null) => void): () => void;

namespace finished {

function \_\_promisify\_\_(stream: NodeJS.ReadableStream | NodeJS.WritableStream | NodeJS.ReadWriteStream, options?: FinishedOptions): Promise<void>;

}

type PipelineSourceFunction<T> = () => Iterable<T> | AsyncIterable<T>;

type PipelineSource<T> = Iterable<T> | AsyncIterable<T> | NodeJS.ReadableStream | PipelineSourceFunction<T>;

type PipelineTransform<S extends PipelineTransformSource<any>, U> =

| NodeJS.ReadWriteStream

| ((source: S extends (...args: any[]) => Iterable<infer ST> | AsyncIterable<infer ST> ? AsyncIterable<ST> : S) => AsyncIterable<U>);

type PipelineTransformSource<T> = PipelineSource<T> | PipelineTransform<any, T>;

type PipelineDestinationIterableFunction<T> = (source: AsyncIterable<T>) => AsyncIterable<any>;

type PipelineDestinationPromiseFunction<T, P> = (source: AsyncIterable<T>) => Promise<P>;

type PipelineDestination<S extends PipelineTransformSource<any>, P> = S extends PipelineTransformSource<infer ST>

? NodeJS.WritableStream | PipelineDestinationIterableFunction<ST> | PipelineDestinationPromiseFunction<ST, P>

: never;

type PipelineCallback<S extends PipelineDestination<any, any>> = S extends PipelineDestinationPromiseFunction<any, infer P>

? (err: NodeJS.ErrnoException | null, value: P) => void

: (err: NodeJS.ErrnoException | null) => void;

type PipelinePromise<S extends PipelineDestination<any, any>> = S extends PipelineDestinationPromiseFunction<any, infer P> ? Promise<P> : Promise<void>;

interface PipelineOptions {

signal: AbortSignal;

}

/\*\*

\* A module method to pipe between streams and generators forwarding errors and

\* properly cleaning up and provide a callback when the pipeline is complete.

\*

\* ```js

\* const { pipeline } = require('stream');

\* const fs = require('fs');

\* const zlib = require('zlib');

\*

\* // Use the pipeline API to easily pipe a series of streams

\* // together and get notified when the pipeline is fully done.

\*

\* // A pipeline to gzip a potentially huge tar file efficiently:

\*

\* pipeline(

\* fs.createReadStream('archive.tar'),

\* zlib.createGzip(),

\* fs.createWriteStream('archive.tar.gz'),

\* (err) => {

\* if (err) {

\* console.error('Pipeline failed.', err);

\* } else {

\* console.log('Pipeline succeeded.');

\* }

\* }

\* );

\* ```

\*

\* The `pipeline` API provides a promise version, which can also

\* receive an options argument as the last parameter with a`signal` `AbortSignal` property. When the signal is aborted,`destroy` will be called on the underlying pipeline, with

\* an`AbortError`.

\*

\* ```js

\* const { pipeline } = require('stream/promises');

\*

\* async function run() {

\* await pipeline(

\* fs.createReadStream('archive.tar'),

\* zlib.createGzip(),

\* fs.createWriteStream('archive.tar.gz')

\* );

\* console.log('Pipeline succeeded.');

\* }

\*

\* run().catch(console.error);

\* ```

\*

\* To use an `AbortSignal`, pass it inside an options object,

\* as the last argument:

\*

\* ```js

\* const { pipeline } = require('stream/promises');

\*

\* async function run() {

\* const ac = new AbortController();

\* const signal = ac.signal;

\*

\* setTimeout(() => ac.abort(), 1);

\* await pipeline(

\* fs.createReadStream('archive.tar'),

\* zlib.createGzip(),

\* fs.createWriteStream('archive.tar.gz'),

\* { signal },

\* );

\* }

\*

\* run().catch(console.error); // AbortError

\* ```

\*

\* The `pipeline` API also supports async generators:

\*

\* ```js

\* const { pipeline } = require('stream/promises');

\* const fs = require('fs');

\*

\* async function run() {

\* await pipeline(

\* fs.createReadStream('lowercase.txt'),

\* async function\* (source, signal) {

\* source.setEncoding('utf8'); // Work with strings rather than `Buffer`s.

\* for await (const chunk of source) {

\* yield await processChunk(chunk, { signal });

\* }

\* },

\* fs.createWriteStream('uppercase.txt')

\* );

\* console.log('Pipeline succeeded.');

\* }

\*

\* run().catch(console.error);

\* ```

\*

\* Remember to handle the `signal` argument passed into the async generator.

\* Especially in the case where the async generator is the source for the

\* pipeline (i.e. first argument) or the pipeline will never complete.

\*

\* ```js

\* const { pipeline } = require('stream/promises');

\* const fs = require('fs');

\*

\* async function run() {

\* await pipeline(

\* async function \* (signal) {

\* await someLongRunningfn({ signal });

\* yield 'asd';

\* },

\* fs.createWriteStream('uppercase.txt')

\* );

\* console.log('Pipeline succeeded.');

\* }

\*

\* run().catch(console.error);

\* ```

\*

\* `stream.pipeline()` will call `stream.destroy(err)` on all streams except:

\*

\* \* `Readable` streams which have emitted `'end'` or `'close'`.

\* \* `Writable` streams which have emitted `'finish'` or `'close'`.

\*

\* `stream.pipeline()` leaves dangling event listeners on the streams

\* after the `callback` has been invoked. In the case of reuse of streams after

\* failure, this can cause event listener leaks and swallowed errors.

\* @since v10.0.0

\* @param callback Called when the pipeline is fully done.

\*/

function pipeline<A extends PipelineSource<any>, B extends PipelineDestination<A, any>>(

source: A,

destination: B,

callback?: PipelineCallback<B>

): B extends NodeJS.WritableStream ? B : NodeJS.WritableStream;

function pipeline<A extends PipelineSource<any>, T1 extends PipelineTransform<A, any>, B extends PipelineDestination<T1, any>>(

source: A,

transform1: T1,

destination: B,

callback?: PipelineCallback<B>

): B extends NodeJS.WritableStream ? B : NodeJS.WritableStream;

function pipeline<A extends PipelineSource<any>, T1 extends PipelineTransform<A, any>, T2 extends PipelineTransform<T1, any>, B extends PipelineDestination<T2, any>>(

source: A,

transform1: T1,

transform2: T2,

destination: B,

callback?: PipelineCallback<B>

): B extends NodeJS.WritableStream ? B : NodeJS.WritableStream;

function pipeline<

A extends PipelineSource<any>,

T1 extends PipelineTransform<A, any>,

T2 extends PipelineTransform<T1, any>,

T3 extends PipelineTransform<T2, any>,

B extends PipelineDestination<T3, any>

>(source: A, transform1: T1, transform2: T2, transform3: T3, destination: B, callback?: PipelineCallback<B>): B extends NodeJS.WritableStream ? B : NodeJS.WritableStream;

function pipeline<

A extends PipelineSource<any>,

T1 extends PipelineTransform<A, any>,

T2 extends PipelineTransform<T1, any>,

T3 extends PipelineTransform<T2, any>,

T4 extends PipelineTransform<T3, any>,

B extends PipelineDestination<T4, any>

>(source: A, transform1: T1, transform2: T2, transform3: T3, transform4: T4, destination: B, callback?: PipelineCallback<B>): B extends NodeJS.WritableStream ? B : NodeJS.WritableStream;

function pipeline(

streams: ReadonlyArray<NodeJS.ReadableStream | NodeJS.WritableStream | NodeJS.ReadWriteStream>,

callback?: (err: NodeJS.ErrnoException | null) => void

): NodeJS.WritableStream;

function pipeline(

stream1: NodeJS.ReadableStream,

stream2: NodeJS.ReadWriteStream | NodeJS.WritableStream,

...streams: Array<NodeJS.ReadWriteStream | NodeJS.WritableStream | ((err: NodeJS.ErrnoException | null) => void)>

): NodeJS.WritableStream;

namespace pipeline {

function \_\_promisify\_\_<A extends PipelineSource<any>, B extends PipelineDestination<A, any>>(source: A, destination: B, options?: PipelineOptions): PipelinePromise<B>;

function \_\_promisify\_\_<A extends PipelineSource<any>, T1 extends PipelineTransform<A, any>, B extends PipelineDestination<T1, any>>(

source: A,

transform1: T1,

destination: B,

options?: PipelineOptions

): PipelinePromise<B>;

function \_\_promisify\_\_<A extends PipelineSource<any>, T1 extends PipelineTransform<A, any>, T2 extends PipelineTransform<T1, any>, B extends PipelineDestination<T2, any>>(

source: A,

transform1: T1,

transform2: T2,

destination: B,

options?: PipelineOptions

): PipelinePromise<B>;

function \_\_promisify\_\_<

A extends PipelineSource<any>,

T1 extends PipelineTransform<A, any>,

T2 extends PipelineTransform<T1, any>,

T3 extends PipelineTransform<T2, any>,

B extends PipelineDestination<T3, any>

>(source: A, transform1: T1, transform2: T2, transform3: T3, destination: B, options?: PipelineOptions): PipelinePromise<B>;

function \_\_promisify\_\_<

A extends PipelineSource<any>,

T1 extends PipelineTransform<A, any>,

T2 extends PipelineTransform<T1, any>,

T3 extends PipelineTransform<T2, any>,

T4 extends PipelineTransform<T3, any>,

B extends PipelineDestination<T4, any>

>(source: A, transform1: T1, transform2: T2, transform3: T3, transform4: T4, destination: B, options?: PipelineOptions): PipelinePromise<B>;

function \_\_promisify\_\_(streams: ReadonlyArray<NodeJS.ReadableStream | NodeJS.WritableStream | NodeJS.ReadWriteStream>, options?: PipelineOptions): Promise<void>;

function \_\_promisify\_\_(

stream1: NodeJS.ReadableStream,

stream2: NodeJS.ReadWriteStream | NodeJS.WritableStream,

...streams: Array<NodeJS.ReadWriteStream | NodeJS.WritableStream | PipelineOptions>

): Promise<void>;

}

interface Pipe {

close(): void;

hasRef(): boolean;

ref(): void;

unref(): void;

}

const promises: typeof streamPromises;

const consumers: typeof streamConsumers;

}

export = internal;

}

declare module 'node:stream' {

import stream = require('stream');

export = stream;

}