/\*\*

\* The `readline` module provides an interface for reading data from a `Readable` stream (such as `process.stdin`) one line at a time.

\*

\* To use the promise-based APIs:

\*

\* ```js

\* import \* as readline from 'node:readline/promises';

\* ```

\*

\* To use the callback and sync APIs:

\*

\* ```js

\* import \* as readline from 'node:readline';

\* ```

\*

\* The following simple example illustrates the basic use of the `readline` module.

\*

\* ```js

\* import \* as readline from 'node:readline/promises';

\* import { stdin as input, stdout as output } from 'process';

\*

\* const rl = readline.createInterface({ input, output });

\*

\* const answer = await rl.question('What do you think of Node.js? ');

\*

\* console.log(`Thank you for your valuable feedback: ${answer}`);

\*

\* rl.close();

\* ```

\*

\* Once this code is invoked, the Node.js application will not terminate until the`readline.Interface` is closed because the interface waits for data to be

\* received on the `input` stream.

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

\*/

declare module 'readline' {

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

interface Key {

sequence?: string | undefined;

name?: string | undefined;

ctrl?: boolean | undefined;

meta?: boolean | undefined;

shift?: boolean | undefined;

}

/\*\*

\* Instances of the `readline.Interface` class are constructed using the`readline.createInterface()` method. Every instance is associated with a

\* single `input` `Readable` stream and a single `output` `Writable` stream.

\* The `output` stream is used to print prompts for user input that arrives on,

\* and is read from, the `input` stream.

\* @since v0.1.104

\*/

class Interface extends EventEmitter {

readonly terminal: boolean;

/\*\*

\* The current input data being processed by node.

\*

\* This can be used when collecting input from a TTY stream to retrieve the

\* current value that has been processed thus far, prior to the `line` event

\* being emitted. Once the `line` event has been emitted, this property will

\* be an empty string.

\*

\* Be aware that modifying the value during the instance runtime may have

\* unintended consequences if `rl.cursor` is not also controlled.

\*

\* \*\*If not using a TTY stream for input, use the `'line'` event.\*\*

\*

\* One possible use case would be as follows:

\*

\* ```js

\* const values = ['lorem ipsum', 'dolor sit amet'];

\* const rl = readline.createInterface(process.stdin);

\* const showResults = debounce(() => {

\* console.log(

\* '\n',

\* values.filter((val) => val.startsWith(rl.line)).join(' ')

\* );

\* }, 300);

\* process.stdin.on('keypress', (c, k) => {

\* showResults();

\* });

\* ```

\* @since v0.1.98

\*/

readonly line: string;

/\*\*

\* The cursor position relative to `rl.line`.

\*

\* This will track where the current cursor lands in the input string, when

\* reading input from a TTY stream. The position of cursor determines the

\* portion of the input string that will be modified as input is processed,

\* as well as the column where the terminal caret will be rendered.

\* @since v0.1.98

\*/

readonly cursor: number;

/\*\*

\* NOTE: According to the documentation:

\*

\* > Instances of the `readline.Interface` class are constructed using the

\* > `readline.createInterface()` method.

\*

\* @see https://nodejs.org/dist/latest-v10.x/docs/api/readline.html#readline\_class\_interface

\*/

protected constructor(input: NodeJS.ReadableStream, output?: NodeJS.WritableStream, completer?: Completer | AsyncCompleter, terminal?: boolean);

/\*\*

\* NOTE: According to the documentation:

\*

\* > Instances of the `readline.Interface` class are constructed using the

\* > `readline.createInterface()` method.

\*

\* @see https://nodejs.org/dist/latest-v10.x/docs/api/readline.html#readline\_class\_interface

\*/

protected constructor(options: ReadLineOptions);

/\*\*

\* The `rl.getPrompt()` method returns the current prompt used by `rl.prompt()`.

\* @since v15.3.0

\* @return the current prompt string

\*/

getPrompt(): string;

/\*\*

\* The `rl.setPrompt()` method sets the prompt that will be written to `output`whenever `rl.prompt()` is called.

\* @since v0.1.98

\*/

setPrompt(prompt: string): void;

/\*\*

\* The `rl.prompt()` method writes the `readline.Interface` instances configured`prompt` to a new line in `output` in order to provide a user with a new

\* location at which to provide input.

\*

\* When called, `rl.prompt()` will resume the `input` stream if it has been

\* paused.

\*

\* If the `readline.Interface` was created with `output` set to `null` or`undefined` the prompt is not written.

\* @since v0.1.98

\* @param preserveCursor If `true`, prevents the cursor placement from being reset to `0`.

\*/

prompt(preserveCursor?: boolean): void;

/\*\*

\* The `rl.question()` method displays the `query` by writing it to the `output`,

\* waits for user input to be provided on `input`, then invokes the `callback`function passing the provided input as the first argument.

\*

\* When called, `rl.question()` will resume the `input` stream if it has been

\* paused.

\*

\* If the `readline.Interface` was created with `output` set to `null` or`undefined` the `query` is not written.

\*

\* The `callback` function passed to `rl.question()` does not follow the typical

\* pattern of accepting an `Error` object or `null` as the first argument.

\* The `callback` is called with the provided answer as the only argument.

\*

\* Example usage:

\*

\* ```js

\* rl.question('What is your favorite food? ', (answer) => {

\* console.log(`Oh, so your favorite food is ${answer}`);

\* });

\* ```

\*

\* Using an `AbortController` to cancel a question.

\*

\* ```js

\* const ac = new AbortController();

\* const signal = ac.signal;

\*

\* rl.question('What is your favorite food? ', { signal }, (answer) => {

\* console.log(`Oh, so your favorite food is ${answer}`);

\* });

\*

\* signal.addEventListener('abort', () => {

\* console.log('The food question timed out');

\* }, { once: true });

\*

\* setTimeout(() => ac.abort(), 10000);

\* ```

\*

\* If this method is invoked as it's util.promisify()ed version, it returns a

\* Promise that fulfills with the answer. If the question is canceled using

\* an `AbortController` it will reject with an `AbortError`.

\*

\* ```js

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

\* const question = util.promisify(rl.question).bind(rl);

\*

\* async function questionExample() {

\* try {

\* const answer = await question('What is you favorite food? ');

\* console.log(`Oh, so your favorite food is ${answer}`);

\* } catch (err) {

\* console.error('Question rejected', err);

\* }

\* }

\* questionExample();

\* ```

\* @since v0.3.3

\* @param query A statement or query to write to `output`, prepended to the prompt.

\* @param callback A callback function that is invoked with the user's input in response to the `query`.

\*/

question(query: string, callback: (answer: string) => void): void;

question(query: string, options: Abortable, callback: (answer: string) => void): void;

/\*\*

\* The `rl.pause()` method pauses the `input` stream, allowing it to be resumed

\* later if necessary.

\*

\* Calling `rl.pause()` does not immediately pause other events (including`'line'`) from being emitted by the `readline.Interface` instance.

\* @since v0.3.4

\*/

pause(): this;

/\*\*

\* The `rl.resume()` method resumes the `input` stream if it has been paused.

\* @since v0.3.4

\*/

resume(): this;

/\*\*

\* The `rl.close()` method closes the `readline.Interface` instance and

\* relinquishes control over the `input` and `output` streams. When called,

\* the `'close'` event will be emitted.

\*

\* Calling `rl.close()` does not immediately stop other events (including `'line'`)

\* from being emitted by the `readline.Interface` instance.

\* @since v0.1.98

\*/

close(): void;

/\*\*

\* The `rl.write()` method will write either `data` or a key sequence identified

\* by `key` to the `output`. The `key` argument is supported only if `output` is

\* a `TTY` text terminal. See `TTY keybindings` for a list of key

\* combinations.

\*

\* If `key` is specified, `data` is ignored.

\*

\* When called, `rl.write()` will resume the `input` stream if it has been

\* paused.

\*

\* If the `readline.Interface` was created with `output` set to `null` or`undefined` the `data` and `key` are not written.

\*

\* ```js

\* rl.write('Delete this!');

\* // Simulate Ctrl+U to delete the line written previously

\* rl.write(null, { ctrl: true, name: 'u' });

\* ```

\*

\* The `rl.write()` method will write the data to the `readline` `Interface`'s`input`\_as if it were provided by the user\_.

\* @since v0.1.98

\*/

write(data: string | Buffer, key?: Key): void;

write(data: undefined | null | string | Buffer, key: Key): void;

/\*\*

\* Returns the real position of the cursor in relation to the input

\* prompt + string. Long input (wrapping) strings, as well as multiple

\* line prompts are included in the calculations.

\* @since v13.5.0, v12.16.0

\*/

getCursorPos(): CursorPos;

/\*\*

\* events.EventEmitter

\* 1. close

\* 2. line

\* 3. pause

\* 4. resume

\* 5. SIGCONT

\* 6. SIGINT

\* 7. SIGTSTP

\* 8. history

\*/

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

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

addListener(event: 'line', listener: (input: string) => void): this;

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

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

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

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

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

addListener(event: 'history', listener: (history: string[]) => void): this;

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

emit(event: 'close'): boolean;

emit(event: 'line', input: string): boolean;

emit(event: 'pause'): boolean;

emit(event: 'resume'): boolean;

emit(event: 'SIGCONT'): boolean;

emit(event: 'SIGINT'): boolean;

emit(event: 'SIGTSTP'): boolean;

emit(event: 'history', history: string[]): boolean;

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

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

on(event: 'line', listener: (input: string) => void): this;

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

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

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

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

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

on(event: 'history', listener: (history: string[]) => void): this;

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

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

once(event: 'line', listener: (input: string) => void): this;

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

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

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

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

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

once(event: 'history', listener: (history: string[]) => void): this;

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

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

prependListener(event: 'line', listener: (input: string) => void): this;

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

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

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

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

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

prependListener(event: 'history', listener: (history: string[]) => void): this;

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

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

prependOnceListener(event: 'line', listener: (input: string) => void): this;

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

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

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

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

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

prependOnceListener(event: 'history', listener: (history: string[]) => void): this;

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

}

type ReadLine = Interface; // type forwarded for backwards compatibility

type Completer = (line: string) => CompleterResult;

type AsyncCompleter = (line: string, callback: (err?: null | Error, result?: CompleterResult) => void) => void;

type CompleterResult = [string[], string];

interface ReadLineOptions {

input: NodeJS.ReadableStream;

output?: NodeJS.WritableStream | undefined;

completer?: Completer | AsyncCompleter | undefined;

terminal?: boolean | undefined;

/\*\*

\* Initial list of history lines. This option makes sense

\* only if `terminal` is set to `true` by the user or by an internal `output`

\* check, otherwise the history caching mechanism is not initialized at all.

\* @default []

\*/

history?: string[] | undefined;

historySize?: number | undefined;

prompt?: string | undefined;

crlfDelay?: number | undefined;

/\*\*

\* If `true`, when a new input line added

\* to the history list duplicates an older one, this removes the older line

\* from the list.

\* @default false

\*/

removeHistoryDuplicates?: boolean | undefined;

escapeCodeTimeout?: number | undefined;

tabSize?: number | undefined;

}

/\*\*

\* The `readline.createInterface()` method creates a new `readline.Interface`instance.

\*

\* ```js

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

\* const rl = readline.createInterface({

\* input: process.stdin,

\* output: process.stdout

\* });

\* ```

\*

\* Once the `readline.Interface` instance is created, the most common case is to

\* listen for the `'line'` event:

\*

\* ```js

\* rl.on('line', (line) => {

\* console.log(`Received: ${line}`);

\* });

\* ```

\*

\* If `terminal` is `true` for this instance then the `output` stream will get

\* the best compatibility if it defines an `output.columns` property and emits

\* a `'resize'` event on the `output` if or when the columns ever change

\* (`process.stdout` does this automatically when it is a TTY).

\*

\* When creating a `readline.Interface` using `stdin` as input, the program

\* will not terminate until it receives `EOF` (Ctrl+D on

\* Linux/macOS, Ctrl+Z followed by Return on

\* Windows).

\* If you want your application to exit without waiting for user input, you can `unref()` the standard input stream:

\*

\* ```js

\* process.stdin.unref();

\* ```

\* @since v0.1.98

\*/

function createInterface(input: NodeJS.ReadableStream, output?: NodeJS.WritableStream, completer?: Completer | AsyncCompleter, terminal?: boolean): Interface;

function createInterface(options: ReadLineOptions): Interface;

/\*\*

\* The `readline.emitKeypressEvents()` method causes the given `Readable` stream to begin emitting `'keypress'` events corresponding to received input.

\*

\* Optionally, `interface` specifies a `readline.Interface` instance for which

\* autocompletion is disabled when copy-pasted input is detected.

\*

\* If the `stream` is a `TTY`, then it must be in raw mode.

\*

\* This is automatically called by any readline instance on its `input` if the`input` is a terminal. Closing the `readline` instance does not stop

\* the `input` from emitting `'keypress'` events.

\*

\* ```js

\* readline.emitKeypressEvents(process.stdin);

\* if (process.stdin.isTTY)

\* process.stdin.setRawMode(true);

\* ```

\*

\* ## Example: Tiny CLI

\*

\* The following example illustrates the use of `readline.Interface` class to

\* implement a small command-line interface:

\*

\* ```js

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

\* const rl = readline.createInterface({

\* input: process.stdin,

\* output: process.stdout,

\* prompt: 'OHAI> '

\* });

\*

\* rl.prompt();

\*

\* rl.on('line', (line) => {

\* switch (line.trim()) {

\* case 'hello':

\* console.log('world!');

\* break;

\* default:

\* console.log(`Say what? I might have heard '${line.trim()}'`);

\* break;

\* }

\* rl.prompt();

\* }).on('close', () => {

\* console.log('Have a great day!');

\* process.exit(0);

\* });

\* ```

\*

\* ## Example: Read file stream line-by-Line

\*

\* A common use case for `readline` is to consume an input file one line at a

\* time. The easiest way to do so is leveraging the `fs.ReadStream` API as

\* well as a `for await...of` loop:

\*

\* ```js

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

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

\*

\* async function processLineByLine() {

\* const fileStream = fs.createReadStream('input.txt');

\*

\* const rl = readline.createInterface({

\* input: fileStream,

\* crlfDelay: Infinity

\* });

\* // Note: we use the crlfDelay option to recognize all instances of CR LF

\* // ('\r\n') in input.txt as a single line break.

\*

\* for await (const line of rl) {

\* // Each line in input.txt will be successively available here as `line`.

\* console.log(`Line from file: ${line}`);

\* }

\* }

\*

\* processLineByLine();

\* ```

\*

\* Alternatively, one could use the `'line'` event:

\*

\* ```js

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

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

\*

\* const rl = readline.createInterface({

\* input: fs.createReadStream('sample.txt'),

\* crlfDelay: Infinity

\* });

\*

\* rl.on('line', (line) => {

\* console.log(`Line from file: ${line}`);

\* });

\* ```

\*

\* Currently, `for await...of` loop can be a bit slower. If `async` / `await`flow and speed are both essential, a mixed approach can be applied:

\*

\* ```js

\* const { once } = require('events');

\* const { createReadStream } = require('fs');

\* const { createInterface } = require('readline');

\*

\* (async function processLineByLine() {

\* try {

\* const rl = createInterface({

\* input: createReadStream('big-file.txt'),

\* crlfDelay: Infinity

\* });

\*

\* rl.on('line', (line) => {

\* // Process the line.

\* });

\*

\* await once(rl, 'close');

\*

\* console.log('File processed.');

\* } catch (err) {

\* console.error(err);

\* }

\* })();

\* ```

\* @since v0.7.7

\*/

function emitKeypressEvents(stream: NodeJS.ReadableStream, readlineInterface?: Interface): void;

type Direction = -1 | 0 | 1;

interface CursorPos {

rows: number;

cols: number;

}

/\*\*

\* The `readline.clearLine()` method clears current line of given `TTY` stream

\* in a specified direction identified by `dir`.

\* @since v0.7.7

\* @param callback Invoked once the operation completes.

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

\*/

function clearLine(stream: NodeJS.WritableStream, dir: Direction, callback?: () => void): boolean;

/\*\*

\* The `readline.clearScreenDown()` method clears the given `TTY` stream from

\* the current position of the cursor down.

\* @since v0.7.7

\* @param callback Invoked once the operation completes.

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

\*/

function clearScreenDown(stream: NodeJS.WritableStream, callback?: () => void): boolean;

/\*\*

\* The `readline.cursorTo()` method moves cursor to the specified position in a

\* given `TTY` `stream`.

\* @since v0.7.7

\* @param callback Invoked once the operation completes.

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

\*/

function cursorTo(stream: NodeJS.WritableStream, x: number, y?: number, callback?: () => void): boolean;

/\*\*

\* The `readline.moveCursor()` method moves the cursor \_relative\_ to its current

\* position in a given `TTY` `stream`.

\*

\* ## Example: Tiny CLI

\*

\* The following example illustrates the use of `readline.Interface` class to

\* implement a small command-line interface:

\*

\* ```js

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

\* const rl = readline.createInterface({

\* input: process.stdin,

\* output: process.stdout,

\* prompt: 'OHAI> '

\* });

\*

\* rl.prompt();

\*

\* rl.on('line', (line) => {

\* switch (line.trim()) {

\* case 'hello':

\* console.log('world!');

\* break;

\* default:

\* console.log(`Say what? I might have heard '${line.trim()}'`);

\* break;

\* }

\* rl.prompt();

\* }).on('close', () => {

\* console.log('Have a great day!');

\* process.exit(0);

\* });

\* ```

\*

\* ## Example: Read file stream line-by-Line

\*

\* A common use case for `readline` is to consume an input file one line at a

\* time. The easiest way to do so is leveraging the `fs.ReadStream` API as

\* well as a `for await...of` loop:

\*

\* ```js

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

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

\*

\* async function processLineByLine() {

\* const fileStream = fs.createReadStream('input.txt');

\*

\* const rl = readline.createInterface({

\* input: fileStream,

\* crlfDelay: Infinity

\* });

\* // Note: we use the crlfDelay option to recognize all instances of CR LF

\* // ('\r\n') in input.txt as a single line break.

\*

\* for await (const line of rl) {

\* // Each line in input.txt will be successively available here as `line`.

\* console.log(`Line from file: ${line}`);

\* }

\* }

\*

\* processLineByLine();

\* ```

\*

\* Alternatively, one could use the `'line'` event:

\*

\* ```js

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

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

\*

\* const rl = readline.createInterface({

\* input: fs.createReadStream('sample.txt'),

\* crlfDelay: Infinity

\* });

\*

\* rl.on('line', (line) => {

\* console.log(`Line from file: ${line}`);

\* });

\* ```

\*

\* Currently, `for await...of` loop can be a bit slower. If `async` / `await`flow and speed are both essential, a mixed approach can be applied:

\*

\* ```js

\* const { once } = require('events');

\* const { createReadStream } = require('fs');

\* const { createInterface } = require('readline');

\*

\* (async function processLineByLine() {

\* try {

\* const rl = createInterface({

\* input: createReadStream('big-file.txt'),

\* crlfDelay: Infinity

\* });

\*

\* rl.on('line', (line) => {

\* // Process the line.

\* });

\*

\* await once(rl, 'close');

\*

\* console.log('File processed.');

\* } catch (err) {

\* console.error(err);

\* }

\* })();

\* ```

\* @since v0.7.7

\* @param callback Invoked once the operation completes.

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

\*/

function moveCursor(stream: NodeJS.WritableStream, dx: number, dy: number, callback?: () => void): boolean;

}

declare module 'node:readline' {

export \* from 'readline';

}