Readline

Stability: 2 - Stable

Source Code: lib/readline.js

The `node: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:

```mjs
import * as readline from 'node:readline/promises';
```

```cjs
const readline = require('node:readline/promises');
```

COPY

To use the callback and sync APIs:

```mjs
import * as readline from 'node:readline';
```

```cjs
const readline = require('node:readline');
```

COPY

The following simple example illustrates the basic use of the `node:readline` module.

```mjs
import * as readline from 'node:readline/promises';
import { stdin as input, stdout as output } from 'node: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();
```

```cjs
const readline = require('node:readline');
const { stdin: input, stdout: output } = require('node:process');

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

```
rl.question('What do you think of Node.js? ', (answer) => {
  // TODO: Log the answer in a database
  console.log(`Thank you for your valuable feedback: ${answer}`);

  rl.close();
});
```

COPY

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.

Class: InterfaceConstructor

- Extends: <EventEmitter>

Instances of the `InterfaceConstructor` class are constructed using the `readlinePromises.createInterface()` or `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.

Event: 'close'

The `'close'` event is emitted when one of the following occur:

- The `rl.close()` method is called and the `InterfaceConstructor` instance has relinquished control over the `input` and `output` streams;
- The `input` stream receives its `'end'` event;
- The `input` stream receives `Ctrl` + `D` to signal end-of-transmission (EOT);
- The `input` stream receives `Ctrl` + `C` to signal `SIGINT` and there is no `'SIGINT'` event listener registered on the `InterfaceConstructor` instance.

The listener function is called without passing any arguments.

The `InterfaceConstructor` instance is finished once the `'close'` event is emitted.

Event: 'line'

The `'line'` event is emitted whenever the `input` stream receives an end-of-line input (`\n` , `\r` , or `\r\n`). This usually occurs when the user presses `Enter` or `Return` .

The `'line'` event is also emitted if new data has been read from a stream and that stream ends without a final end-of-line marker.

The listener function is called with a string containing the single line of received input.

```
rl.on('line', (input) => {
  console.log(`Received: ${input}`);
});
```

COPY

Event: 'history'

The `'history'` event is emitted whenever the history array has changed.

The listener function is called with an array containing the history array. It will reflect all changes, added lines and removed lines due to `historySize` and `removeHistoryDuplicates` .

The primary purpose is to allow a listener to persist the history. It is also possible for the listener to change the history object. This could be useful to prevent certain lines to be added to the history, like a password.

```
rl.on('history', (history) => {
  console.log(`Received: ${history}`);
});
```
COPY

Event: 'pause'

The `'pause'` event is emitted when one of the following occur:

- The `input` stream is paused.
- The `input` stream is not paused and receives the `'SIGCONT'` event. (See events `'SIGTSTP'` and `'SIGCONT'`.)

The listener function is called without passing any arguments.

```
rl.on('pause', () => {
  console.log('Readline paused.');
});
```
COPY

Event: 'resume'

The `'resume'` event is emitted whenever the `input` stream is resumed.

The listener function is called without passing any arguments.

```
rl.on('resume', () => {
  console.log('Readline resumed.');
});
```
COPY

Event: 'SIGCONT'

The `'SIGCONT'` event is emitted when a Node.js process previously moved into the background using `Ctrl` + `Z` (i.e. `SIGTSTP`) is then brought back to the foreground using `fg(1p)`.

If the `input` stream was paused *before* the `SIGTSTP` request, this event will not be emitted.

The listener function is invoked without passing any arguments.

```
rl.on('SIGCONT', () => {
  // `prompt` will automatically resume the stream
  rl.prompt();
});
```
COPY

The `'SIGCONT'` event is *not* supported on Windows.

Event: 'SIGINT'

The `'SIGINT'` event is emitted whenever the `input` stream receives a `Ctrl+C` input, known typically as `SIGINT`. If there are no `'SIGINT'` event listeners registered when the `input` stream receives a `SIGINT`, the `'pause'` event will be emitted.

The listener function is invoked without passing any arguments.

```
rl.on('SIGINT', () => {
  rl.question('Are you sure you want to exit? ', (answer) => {
    if (answer.match(/^y(es)?$/i)) rl.pause();
  });
});
```
COPY

Event: 'SIGTSTP'

The `'SIGTSTP'` event is emitted when the `input` stream receives a `Ctrl`+`Z` input, typically known as `SIGTSTP`. If there are no `'SIGTSTP'` event listeners registered when the `input` stream receives a `SIGTSTP`, the Node.js process will be sent to the background.

When the program is resumed using `fg(1p)`, the `'pause'` and `'SIGCONT'` events will be emitted. These can be used to resume the `input` stream.

The `'pause'` and `'SIGCONT'` events will not be emitted if the `input` was paused before the process was sent to the background.

The listener function is invoked without passing any arguments.

```
rl.on('SIGTSTP', () => {
  // This will override SIGTSTP and prevent the program from going to the
  // background.
  console.log('Caught SIGTSTP.');
});
```
COPY

The `'SIGTSTP'` event is *not* supported on Windows.

rl.close()

The `rl.close()` method closes the `InterfaceConstructor` 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 `InterfaceConstructor` instance.

rl.pause()

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 `InterfaceConstructor` instance.

rl.prompt([preserveCursor])

- `preserveCursor` `<boolean>` If `true`, prevents the cursor placement from being reset to `0`.

The `rl.prompt()` method writes the `InterfaceConstructor` 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 `InterfaceConstructor` was created with `output` set to `null` or `undefined` the prompt is not written.

rl.resume()

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

rl.setPrompt(prompt)

- `prompt` `<string>`

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

rl.getPrompt()

- Returns: `<string>` the current prompt string

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

rl.write(data[, key])

- `data` `<string>`
- `key` `<Object>`
  - `ctrl` `<boolean>` `true` to indicate the `Ctrl` key.
  - `meta` `<boolean>` `true` to indicate the `Meta` key.
  - `shift` `<boolean>` `true` to indicate the `Shift` key.
  - `name` `<string>` The name of the a key.

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 `InterfaceConstructor` was created with `output` set to `null` or `undefined` the `data` and `key` are not written.

```
rl.write('Delete this!');
// Simulate Ctrl+U to delete the line written previously
rl.write(null, { ctrl: true, name: 'u' });
```

COPY

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

rl[Symbol.asyncIterator]()

- Returns: `<AsyncIterator>`

Create an `AsyncIterator` object that iterates through each line in the input stream as a string. This method allows asynchronous iteration of `InterfaceConstructor` objects through `for await...of` loops.

Errors in the input stream are not forwarded.

If the loop is terminated with `break`, `throw`, or `return`, `rl.close()` will be called. In other words, iterating over a `InterfaceConstructor` will always consume the input stream fully.

Performance is not on par with the traditional `'line'` event API. Use `'line'` instead for performance-sensitive applications.

```
async function processLineByLine() {
  const rl = readline.createInterface({
    // ...
  });

  for await (const line of rl) {
    // Each line in the readline input will be successively available here as
    // `line`.
```

```
  }
}
```

COPY

`readline.createInterface()` will start to consume the input stream once invoked. Having asynchronous operations between interface creation and asynchronous iteration may result in missed lines.

rl.line

- `<string>`

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:

```
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();
});
```

COPY

rl.cursor

- `<number>` | `<undefined>`

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.

rl.getCursorPos()

- Returns: `<Object>`
  - `rows` `<number>` the row of the prompt the cursor currently lands on
  - `cols` `<number>` the screen column the cursor currently lands on

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.

Promises API

Stability: 1 - Experimental

Class: readlinePromises.Interface

- Extends: <readline.InterfaceConstructor>

Instances of the `readlinePromises.Interface` class are constructed using the `readlinePromises.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.

rl.question(query[, options])

- `query` <string> A statement or query to write to `output`, prepended to the prompt.
- `options` <Object>
  - `signal` <AbortSignal> Optionally allows the `question()` to be canceled using an `AbortSignal`.
- Returns: <Promise> A promise that is fulfilled with the user's input in response to the `query`.

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 `readlinePromises.Interface` was created with `output` set to `null` or `undefined` the `query` is not written.

If the question is called after `rl.close()`, it returns a rejected promise.

Example usage:

```
const answer = await rl.question('What is your favorite food? ');
console.log(`Oh, so your favorite food is ${answer}`);
```

COPY

Using an `AbortSignal` to cancel a question.

```
const signal = AbortSignal.timeout(10_000);

signal.addEventListener('abort', () => {
  console.log('The food question timed out');
}, { once: true });

const answer = await rl.question('What is your favorite food? ', { signal });
console.log(`Oh, so your favorite food is ${answer}`);
```

COPY

Class: readlinePromises.Readline

new readlinePromises.Readline(stream[, options])

- `stream` <stream.Writable> A TTY stream.
- `options` <Object>
  - `autoCommit` <boolean> If `true`, no need to call `rl.commit()`.

rl.clearLine(dir)

- `dir` <integer>
  - `-1`: to the left from cursor
  - `1`: to the right from cursor

    - `0` : the entire line
- Returns: this

The `rl.clearLine()` method adds to the internal list of pending action an action that clears current line of the associated `stream` in a specified direction identified by `dir` . Call `rl.commit()` to see the effect of this method, unless `autoCommit: true` was passed to the constructor.

rl.clearScreenDown()

- Returns: this

The `rl.clearScreenDown()` method adds to the internal list of pending action an action that clears the associated stream from the current position of the cursor down. Call `rl.commit()` to see the effect of this method, unless `autoCommit: true` was passed to the constructor.

rl.commit()

- Returns: `<Promise>`

The `rl.commit()` method sends all the pending actions to the associated `stream` and clears the internal list of pending actions.

rl.cursorTo(x[, y])

- `x` `<integer>`
- `y` `<integer>`
- Returns: this

The `rl.cursorTo()` method adds to the internal list of pending action an action that moves cursor to the specified position in the associated `stream` . Call `rl.commit()` to see the effect of this method, unless `autoCommit: true` was passed to the constructor.

rl.moveCursor(dx, dy)

- `dx` `<integer>`
- `dy` `<integer>`
- Returns: this

The `rl.moveCursor()` method adds to the internal list of pending action an action that moves the cursor *relative* to its current position in the associated `stream` . Call `rl.commit()` to see the effect of this method, unless `autoCommit: true` was passed to the constructor.

rl.rollback()

- Returns: this

The `rl.rollback` methods clears the internal list of pending actions without sending it to the associated `stream` .

readlinePromises.createInterface(options)

- `options` `<Object>`
    - `input` `<stream.Readable>` The Readable stream to listen to. This option is *required*.
    - `output` `<stream.Writable>` The Writable stream to write readline data to.
    - `completer` `<Function>` An optional function used for Tab autocompletion.
    - `terminal` `<boolean>` `true` if the `input` and `output` streams should be treated like a TTY, and have ANSI/VT100 escape codes written to it. **Default:** checking `isTTY` on the `output` stream upon instantiation.
    - `history` `<string[]>` 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:** `[]` .
    - `historySize` `<number>` Maximum number of history lines retained. To disable the history set this value to `0` . 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:** `30`.

- `removeHistoryDuplicates` <boolean> 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`.
- `prompt` <string> The prompt string to use. **Default:** `'> '`.
- `crlfDelay` <number> If the delay between `\r` and `\n` exceeds `crlfDelay` milliseconds, both `\r` and `\n` will be treated as separate end-of-line input. `crlfDelay` will be coerced to a number no less than `100`. It can be set to `Infinity`, in which case `\r` followed by `\n` will always be considered a single newline (which may be reasonable for reading files with `\r\n` line delimiter). **Default:** `100`.
- `escapeCodeTimeout` <number> The duration `readlinePromises` will wait for a character (when reading an ambiguous key sequence in milliseconds one that can both form a complete key sequence using the input read so far and can take additional input to complete a longer key sequence). **Default:** `500`.
- `tabSize` <integer> The number of spaces a tab is equal to (minimum 1). **Default:** `8`.

- Returns: <readlinePromises.Interface>

The `readlinePromises.createInterface()` method creates a new `readlinePromises.Interface` instance.

```
const readlinePromises = require('node:readline/promises');
const rl = readlinePromises.createInterface({
  input: process.stdin,
  output: process.stdout,
});
```

COPY

Once the `readlinePromises.Interface` instance is created, the most common case is to listen for the `'line'` event:

```
rl.on('line', (line) => {
  console.log(`Received: ${line}`);
});
```

COPY

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).

Use of the `completer` function

The `completer` function takes the current line entered by the user as an argument, and returns an `Array` with 2 entries:

- An `Array` with matching entries for the completion.
- The substring that was used for the matching.

For instance: `[[substr1, substr2, ...], originalsubstring]`.

```
function completer(line) {
  const completions = '.help .error .exit .quit .q'.split(' ');
  const hits = completions.filter((c) => c.startsWith(line));
  // Show all completions if none found
  return [hits.length ? hits : completions, line];
}
```

COPY

The `completer` function can also return a <Promise>, or be asynchronous:

```
async function completer(linePartial) {
  await someAsyncWork();
```

```
  return [['123'], linePartial];
}
```
COPY

Callback API

Class: readline.Interface

- Extends: <readline.InterfaceConstructor>

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.

rl.question(query[, options], callback)

- `query` <string> A statement or query to write to `output`, prepended to the prompt.
- `options` <Object>
  - `signal` <AbortSignal> Optionally allows the `question()` to be canceled using an `AbortController`.
- `callback` <Function> A callback function that is invoked with the user's input in response to the `query`.

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.

An error will be thrown if calling `rl.question()` after `rl.close()`.

Example usage:

```
rl.question('What is your favorite food? ', (answer) => {
  console.log(`Oh, so your favorite food is ${answer}`);
});
```
COPY

Using an `AbortController` to cancel a question.

```
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);
```
COPY

readline.clearLine(stream, dir[, callback])

- `stream` <stream.Writable>
- `dir` <number>
  - `-1` : to the left from cursor
  - `1` : to the right from cursor
  - `0` : the entire line
- `callback` <Function> Invoked once the operation completes.
- Returns: <boolean> `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` .

The `readline.clearLine()` method clears current line of given TTY stream in a specified direction identified by `dir` .

readline.clearScreenDown(stream[, callback])

- `stream` <stream.Writable>
- `callback` <Function> Invoked once the operation completes.
- Returns: <boolean> `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` .

The `readline.clearScreenDown()` method clears the given TTY stream from the current position of the cursor down.

readline.createInterface(options)

- `options` <Object>
  - `input` <stream.Readable> The Readable stream to listen to. This option is *required*.
  - `output` <stream.Writable> The Writable stream to write readline data to.
  - `completer` <Function> An optional function used for Tab autocompletion.
  - `terminal` <boolean> `true` if the `input` and `output` streams should be treated like a TTY, and have ANSI/VT100 escape codes written to it. **Default:** checking `isTTY` on the `output` stream upon instantiation.
  - `history` <string[]> 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:** `[]` .
  - `historySize` <number> Maximum number of history lines retained. To disable the history set this value to `0` . 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:** `30` .
  - `removeHistoryDuplicates` <boolean> 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` .
  - `prompt` <string> The prompt string to use. **Default:** `'> '` .
  - `crlfDelay` <number> If the delay between `\r` and `\n` exceeds `crlfDelay` milliseconds, both `\r` and `\n` will be treated as separate end-of-line input. `crlfDelay` will be coerced to a number no less than `100` . It can be set to `Infinity` , in which case `\r` followed by `\n` will always be considered a single newline (which may be reasonable for reading files with `\r\n` line delimiter). **Default:** `100` .
  - `escapeCodeTimeout` <number> The duration `readline` will wait for a character (when reading an ambiguous key sequence in milliseconds one that can both form a complete key sequence using the input read so far and can take additional input to complete a longer key sequence). **Default:** `500` .
  - `tabSize` <integer> The number of spaces a tab is equal to (minimum 1). **Default:** `8` .
  - `signal` <AbortSignal> Allows closing the interface using an AbortSignal. Aborting the signal will internally call `close` on the interface.
- Returns: <readline.Interface>

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

```
const readline = require('node:readline');
const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout,
});
```
COPY

Once the `readline.Interface` instance is created, the most common case is to listen for the `'line'` event:

```
rl.on('line', (line) => {
  console.log(`Received: ${line}`);
});
```
COPY

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 an EOF character . To exit without waiting for user input, call `process.stdin.unref()` .

Use of the `completer` function

The `completer` function takes the current line entered by the user as an argument, and returns an `Array` with 2 entries:

- An `Array` with matching entries for the completion.
- The substring that was used for the matching.

For instance: `[[substr1, substr2, ...], originalsubstring]` .

```
function completer(line) {
  const completions = '.help .error .exit .quit .q'.split(' ');
  const hits = completions.filter((c) => c.startsWith(line));
  // Show all completions if none found
  return [hits.length ? hits : completions, line];
}
```
COPY

The `completer` function can be called asynchronously if it accepts two arguments:

```
function completer(linePartial, callback) {
  callback(null, [['123'], linePartial]);
}
```
COPY

readline.cursorTo(stream, x[, y][, callback])

- `stream` `<stream.Writable>`
- `x` `<number>`
- `y` `<number>`
- `callback` `<Function>` Invoked once the operation completes.
- Returns: `<boolean>` `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` .

The `readline.cursorTo()` method moves cursor to the specified position in a given TTY `stream` .

readline.moveCursor(stream, dx, dy[, callback])

- `stream` <stream.Writable>
- `dx` <number>
- `dy` <number>
- `callback` <Function> Invoked once the operation completes.
- Returns: <boolean> `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`.

The `readline.moveCursor()` method moves the cursor *relative* to its current position in a given TTY `stream`.

readline.emitKeypressEvents(stream[, interface])

- `stream` <stream.Readable>
- `interface` <readline.InterfaceConstructor>

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.

```
readline.emitKeypressEvents(process.stdin);
if (process.stdin.isTTY)
  process.stdin.setRawMode(true);
```

COPY

Example: Tiny CLI

The following example illustrates the use of `readline.Interface` class to implement a small command-line interface:

```
const readline = require('node: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);
});
```

COPY

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:

```
const fs = require('node:fs');
const readline = require('node: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();
```

COPY

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

```
const fs = require('node:fs');
const readline = require('node:readline');

const rl = readline.createInterface({
  input: fs.createReadStream('sample.txt'),
  crlfDelay: Infinity,
});

rl.on('line', (line) => {
  console.log(`Line from file: ${line}`);
});
```

COPY

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:

```
const { once } = require('node:events');
const { createReadStream } = require('node:fs');
const { createInterface } = require('node: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);
  }
})();
```

COPY

TTY keybindings

Keybindings	Description	Notes
`Ctrl` + `Shift` + `Backspace`	Delete line left	Doesn't work on Linux, Mac and Windows
`Ctrl` + `Shift` + `Delete`	Delete line right	Doesn't work on Mac
`Ctrl` + `C`	Emit `SIGINT` or close the readline instance	
`Ctrl` + `H`	Delete left	
`Ctrl` + `D`	Delete right or close the readline instance in case the current line is empty / EOF	Doesn't work on Windows
`Ctrl` + `U`	Delete from the current position to the line start	
`Ctrl` + `K`	Delete from the current position to the end of line	
`Ctrl` + `Y`	Yank (Recall) the previously deleted text	Only works with text deleted by `Ctrl` + `U` or `Ctrl` + `K`
`Meta` + `Y`	Cycle among previously deleted texts	Only available when the last keystroke is `Ctrl` + `Y` or `Meta` + `Y`
`Ctrl` + `A`	Go to start of line	
`Ctrl` + `E`	Go to end of line	
`Ctrl` + `B`	Back one character	
`Ctrl` + `F`	Forward one character	
`Ctrl` + `L`	Clear screen	
`Ctrl` + `N`	Next history item	
`Ctrl` + `P`	Previous history item	

`Ctrl` + `-`	Undo previous change	Any keystroke that emits key code `0x1F` will do this action. In many terminals, for example `xterm`, this is bound to `Ctrl` + `-`.
`Ctrl` + `6`	Redo previous change	Many terminals don't have a default redo keystroke. We choose key code `0x1E` to perform redo. In `xterm`, it is bound to `Ctrl` + `6` by default.
`Ctrl` + `Z`	Moves running process into background. Type `fg` and press `Enter` to return.	Doesn't work on Windows
`Ctrl` + `W` or `Ctrl` + `Backspace`	Delete backward to a word boundary	`Ctrl` + `Backspace` Doesn't work on Linux, Mac and Windows
`Ctrl` + `Delete`	Delete forward to a word boundary	Doesn't work on Mac
`Ctrl` + `Left arrow` or `Meta` + `B`	Word left	`Ctrl` + `Left arrow` Doesn't work on Mac
`Ctrl` + `Right arrow` or `Meta` + `F`	Word right	`Ctrl` + `Right arrow` Doesn't work on Mac
`Meta` + `D` or `Meta` + `Delete`	Delete word right	`Meta` + `Delete` Doesn't work on windows
`Meta` + `Backspace`	Delete word left	Doesn't work on Mac