Class IO < Object

Subclasses: File

Class IO is the basis for all input and output in Ruby. An I/O stream may be *duplexed* (that is, bidirectional) and so may use more than one native operating system stream.

Many of the examples in this section use class File, the only standard subclass of IO. The two classes are closely associated.

As used in this section, *portname* may take any of the following forms:

- A plain string represents a filename suitable for the underlying operating system.
- A string starting with | indicates a subprocess. The remainder of the string following the | is invoked as a process with appropriate input/output channels connected to it.
- A string equal to |- will create another Ruby instance as a subprocess.

The IO class uses the Unix abstraction of *file descriptors* (fds), small integers that represent open files. Conventionally, standard input has an fd of 0, standard output an fd of 1, and standard error an fd of 2.

Ruby will convert pathnames between different operating system conventions if possible. For instance, on a Windows system the filename /gumby/ruby/test.rb will be opened as \gumby\ruby\test.rb. When specifying a Windows-style filename in a double-quoted Ruby string, remember to escape the backslashes.

```
"c:\\gumby\\ruby\\test.rb"
```

Our examples here will use the Unix-style forward slashes; File::SEPARATOR can be used to get the platform-specific separator character.

1.9

I/O ports may be opened in any one of several different modes, which are shown in this section as *mode*. This mode string must be one of the values listed in Table 27.7 on the next page. As of Ruby 1.9, the mode may also contain information on the external and internal encoding of the data associated with the port. If an external encoding is specified, Ruby assumes that the data it received from the operating system uses that encoding. If no internal encoding is given, strings read from the port will have this encoding. If an internal encoding is given, data will be transcoded from the external to the internal encoding, and strings will have that encoding. The reverse happens on output.

The file mode may optionally be specified as a Fixnum by *or*-ing together the flags described in Table 27.5 on page 514. (Yes, it is bad coupling that the IO class uses constants defined in a child.)

Mixes in

Enumerable:

all?, any?, collect, count, cycle, detect, drop, drop_while, each_cons, each_slice, each_with_index, entries, find, find_all, find_index, first, grep, group_by, include?, inject, map, max, max_by, member?, min, min_by, minmax, minmax_by, none?, one?, partition, reduce, reject, select, sort, sort_by, take, take_while, to_a, zip

Table 27.7. Mode Strings

Modes can be represented as an integer formed by or-ing together values from Table 27.7. They are more commonly represented as a string. Mode strings have the form "file-mode[:external-encoding[:internal-encoding]]". The file-mode portion is one of the options listed in the following table. The two encodings are the names (or aliases) of encodings supported by your interpreter. See Chapter 17 on page 264 for more information.

Mode	Meaning
r	Read-only, starts at beginning of file (default mode).
r+	Read/write, starts at beginning of file.
w	Write-only, truncates an existing file to zero length or creates a new file for writing.
w+	Read/write, truncates existing file to zero length or creates a new file for reading and writing.
a	Write-only, starts at end of file if file exists; otherwise, creates a new file for writing.
a+	Read/write, starts at end of file if file exists; otherwise, creates a new file for reading and writing.
b	Binary file mode (may appear with any of the key letters listed earlier). As of Ruby 1.9, this modifier should be supplied on all ports opened in binary mode (on Unix as well as on DOS/Windows). To read a file in binary mode and receive the data as a stream of bytes, use the modestring "rb:ascii-8bit".

1.9

Class methods

binread

IO.binread((name) ⟨ , *length* ⟨ , offset ⟩ ⟩) → *string*

1.9

Opens *name* with mode rb:ASCII-8BIT, reads *length* bytes starting at *offset*, and then closes the file. The bytes are returned in a string with ASCII-8BIT encoding. *offset* defaults to 0, and *length* defaults to the number of bytes between *offset* and the end of the file.

```
IO.binread("testfile", 20)       # =>  "This is line one\nThi"
IO.binread("testfile", 20, 20)   # =>  "s is line two\nThis i"
str = IO.binread("testfile")
str.encoding                     # =>  #<Encoding:ASCII-8BIT>
str1 = IO.read("testfile")
str1.encoding                    # =>  #<Encoding:UTF-8>
```

copy_stream

IO.copy_stream(*from*, *to* ⟨ , *max_length* ⟨ , offset ⟩ ⟩) → *integer*

1.9

Copies *from* to *to*. These may be specified as either filenames or as open IO streams. You may optionally specify a maximum length to copy and a byte offset to start the copy from. Returns the number of bytes copied.

```
IO.copy_stream("testfile", "newfile", 10, 10)
ip = File.open("/etc/passwd")
op = File.open("extract", "w")
op.puts "First 20 characters of /etc/passwd"
IO.copy_stream(ip, op, 20)
```

```
op.puts "\nEnd of extract"
op.close
puts File.readlines("extract")
```

produces:

```
First 20 characters of /etc/passwd
##
# User Database
#
End of extract
```

for_fd IO.for_fd(*int*, *mode*) → *io*

Synonym for IO.new.

foreach *io*.foreach(*portname*, *separator*=$/ ⟨ , *options* ⟩) {| *line* | *block* } → nil
io.foreach(*portname*, *limit* ⟨ , *options* ⟩) {| *line* | *block* } → nil
io.foreach(*portname*, *separator*, *limit* ⟨ , *options* ⟩) {| *line* | *block* } → nil

1.9 Executes the block for every line in the named I/O port, where lines are separated by *separator*. If *separator* is nil, the entire file is passed as a single string. If the *limit* argument is present and positive, at most that many characters will be returned in each iteration. If only the *limit* argument is given and that argument is negative, then encodings will be ignored while looking for the record separator, which increases performance.

```
IO.foreach("testfile") {|x| puts "GOT: #{x}" }
```

produces:

```
GOT: This is line one
GOT: This is line two
GOT: This is line three
GOT: And so on...
```

1.9 *options* is an optional hash used to pass parameters to the underlying open call used by read. It may contain one or more of

key	Value(s)
encoding:	The encoding for the string, either as "external" or "external:internal"
mode:	The mode string to be passed to open
open_args:	An array containing the arguments to be passed to open; other options are ignored if this one is present

```
IO.foreach("testfile", nil, mode: "rb", encoding: "ascii-8bit") do |content|
  puts content.encoding
end
IO.foreach("testfile", nil, open_args: ["r:iso-8859-1"]) do |content|
  puts content.encoding
end
```

produces:

```
ASCII-8BIT
ISO-8859-1
```

new IO.new(*int*, *mode*) → *io*

Returns a new IO object (a stream) for the given integer file descriptor and mode. See also IO#fileno and IO.for_fd.

```
a = IO.new(2, "w")      # '2' is standard error
STDERR.puts "Hello"
a.puts "World"
```

produces:

```
Hello
World
```

open IO.open(⟨ args ⟩+) → *io*
IO.open(⟨ args ⟩+) {| *io* | *block* } → *obj*

IO.open creates a new IO object, passing *args* to that object's initialize method. If no block is given, simply returns that object. If a block is given, passes the IO object to the block. When the block exits (even via exception or program termination), the *io* object will be closed. If the block is present, IO.open returns the value of the block. The rough implementation is as follows:

```
class IO
  def open(*args)
    file = return_value = self.new(*args)
    begin
       return_value = yield(file)
    ensure
       file.close
    end if block_given?
    return_value
  end
end
```

Note that subclasses of IO such as File can use open even though their constructors take different parameters. Calling File.open(...) will invoke File's constructor, not IO's.

```
IO.open(1, "w") do |io|
  io.puts "Writing to stdout"
end
```

produces:

```
Writing to stdout
```

```
File.open("testfile", mode: "r", encoding: "utf-8") do |f|
  puts f.read
end
```

produces:

```
This is line one
This is line two
This is line three
And so on...
```

pipe IO.pipe → [*read_file*, *write_file*]

Creates a pair of pipe endpoints (connected to each other) and returns them as a two-element array of IO objects. *write_file* is automatically placed into sync mode. Not available on all platforms.

In the following example, the two processes close the ends of the pipe that they are not using. This is not just a cosmetic nicety. The read end of a pipe will not generate an end-of-file condition if any writers have the pipe still open. In the case of the parent process, the rd.read will never return if it does not first issue a wr.close.

```
rd, wr = IO.pipe
if fork
  wr.close
  puts "Parent got: <#{rd.read}>"
  rd.close
  Process.wait
else
  rd.close
  puts "Sending message to parent"
  wr.write "Hi Dad"
  wr.close
end
```

produces:

```
Sending message to parent
Parent got: <Hi Dad>
```

popen IO.popen(*cmd*, *mode*="r") → *io*
IO.popen(*cmd*, *mode*="r") {| *io* | *block* } → *obj*

Runs the specified command string as a subprocess; the subprocess's standard input and output will be connected to the returned IO object. The parameter *cmd* may be a string or (in Ruby 1.9) an array of strings. In the latter case, the array is used as the argv parameter for the new process, and no special shell processing is performed on the strings. In addition, if the array starts with a hash, it will be used to set environment variables in the subprocess, and if it ends with a hash, the hash will be used to set execution options for the subprocess. See Kernel.spawn for details. If *cmd* is a string, it will be subject to shell expansion. If the *cmd* string starts with a minus sign (–) and the operating system supports fork(2), then the current Ruby process is forked. The default mode for the new file object is r, but *mode* may be set to any of the modes in Table 27.7 on page 547.

1.9

If a block is given, Ruby will run the command as a child connected to Ruby with a pipe. Ruby's end of the pipe will be passed as a parameter to the block. In this case, IO.popen returns the value of the block.

If a block is given with a *cmd_string* of "–", the block will be run in two separate processes: once in the parent and once in a child. The parent process will be passed the pipe object as a parameter to the block, the child version of the block will be passed nil, and the child's standard in and standard out will be connected to the parent through the pipe. Not available on all platforms. Also see the Open3 library on page 783 and Kernel#exec on page 568.

```
pipe = IO.popen("uname")
p(pipe.readlines)
puts "Parent is #{Process.pid}"
IO.popen("date") {|pipe| puts pipe.gets }
IO.popen("-") {|pipe| STDERR.puts "#{Process.pid} is here, pipe=#{pipe}" }
```

produces:

```
["Darwin\n"]
Parent is 84543
Mon Apr 13 13:26:27 CDT 2009
84543 is here, pipe=#<IO:0x0a2fd0>
84546 is here, pipe=
```

Here's an example that uses the Ruby 1.9 options to merge standard error and standard output into a single stream. Note that buffering means that the error output comes back ahead of the standard output.

```
pipe = IO.popen([ "bc", { STDERR => STDOUT }], "r+" )
pipe.puts '1 + 3; bad_function()'
pipe.close_write
puts pipe.readlines
```

produces:

```
Runtime error (func=(main), adr=8): Function bad_function not defined.
4
```

---

read IO.read(*portname*, ⟨ *length*=$/ ⟨ , *offset* ⟩ ⟩ ⟨ , *options* ⟩) → *string*

1.9

Opens the file, optionally seeks to the given offset, and then returns *length* bytes (defaulting to the rest of the file). read ensures the file is closed before returning.

options is an optional hash used to pass parameters to the underlying open call used by read. See IO.foreach for details.

```
IO.read("testfile")           # =>  "This is line one\nThis is line
                                   two\nThis is line three\nAnd so
                                   on...\n"
IO.read("testfile", 20)       # =>  "This is line one\nThi"
IO.read("testfile", 20, 10)   # =>  "ne one\nThis is line "
```

---

readlines IO.readlines(*portname*, *separator*=$/ ⟨ , *options* ⟩) → *array*
IO.readlines(*portname*, *limit* ⟨ , *options* ⟩) → *array*
IO.readlines(*portname*, *separator*, *limit* ⟨ , *options* ⟩) → *array*

1.9

Reads the entire file specified by *portname* as individual lines and returns those lines in an array. Lines are separated by *separator*. If *separator* is nil, the entire file is passed as a single string. If the *limit* argument is present and positive, at most that many characters will be returned in each iteration. If only the *limit* argument is given and that argument is negative, then encodings will be ignored while looking for the record separator, which increases performance. *options* is an optional hash used to pass parameters to the underlying open call used by read. See IO.foreach for details.

```
a = IO.readlines("testfile")
a[0]   # =>  "This is line one\n"
```

select IO.select(*read_array* ⟨ , *write_array* ⟨ , *error_array* ⟨ , *timeout* ⟩ ⟩ ⟩) → *array* or nil

See Kernel#select on page 576.

sysopen IO.sysopen(*path*, ⟨ *mode* ⟨ , *perm* ⟩ ⟩) → *int*

Opens the given path, returning the underlying file descriptor as a Fixnum.

```
IO.sysopen("testfile")   # =>   4
```

try_convert IO.try_convert(*obj*) → *an_io* or nil

1.9

If *obj* is not already an I/O object, attempt to convert it to one by calling its to_io method. Returns nil if no conversion could be made.

```
class SillyIOObject
  def to_io
    STDOUT
  end
end
IO.try_convert(SillyIOObject.new)   # =>   #<IO:<STDOUT>>
IO.try_convert("Shemp")             # =>   nil
```

Instance methods

<< *io* << *obj* → *io*

String Output—Writes *obj* to *io*. *obj* will be converted to a string using to_s.

```
STDOUT << "Hello " << "world!\n"
```

produces:

```
Hello world!
```

binmode *io*.binmode → *io*

1.9

Puts *io* into binary mode. It is more common to use the "b" modifier in the mode string to set binary mode when you open a file. Binary mode is required when reading or writing files containing bit sequences that are not valid in the encoding of the file. Once a stream is in binary mode, it cannot be reset to nonbinary mode.

binmode? *io*.binmode? → true or false

1.9

Returns true if *io* is in binary mode.

```
f = File.open("/etc/passwd")
f.binmode?   # =>   false
f = File.open("/etc/passwd", "rb:binary")
f.binmode?   # =>   true
```

bytes *io*.bytes → *enumerator*

1.9

Returns an enumerator that iterates over the bytes (not characters) in *io*, returning each as an integer. See also IO#getbyte.

```
file = File.open("testfile")
enum = file.bytes
enum.first(10)   # =>   [84, 104, 105, 115, 32, 105, 115, 32, 108, 105]
```

chars

io.chars → *enumerator*

1.9

Returns an enumerator that allows iteration over the characters in *io*.

```
file = File.open("testfile")
enum = file.chars
enum.first(7)   # =>   ["T", "h", "i", "s", " ", "i", "s"]
```

close

io.close → nil

Closes *io* and flushes any pending writes to the operating system. The stream is unavailable for any further data operations; an IOError is raised if such an attempt is made. I/O streams are automatically closed when they are claimed by the garbage collector.

close_on_exec?

io.close_on_exec? → true or false

1.9

Returns the state of the *close on exec* flag for *io*. Raises NotImplemented if not available.

close_on_exec=

io.close_on_exec = true or false → nil

1.9

Sets the *close on exec* flag for *io*. Raises NotImplemented if not available. I/O objects with this flag set will be closed across exec() calls.

close_read

io.close_read → nil

Closes the read end of a duplex I/O stream (in other words, one that contains both a read and a write stream, such as a pipe). Raises an IOError if the stream is not duplexed.

```
f = IO.popen("/bin/sh","r+")
f.close_read
f.readlines
```

produces:

```
prog.rb:3:in `readlines': not opened for reading (IOError)
from /tmp/prog.rb:3:in `<main>'
```

close_write

io.close_write → nil

Closes the write end of a duplex I/O stream (in other words, one that contains both a read and a write stream, such as a pipe). Will raise an IOError if the stream is not duplexed.

```
f = IO.popen("/bin/sh","r+")
f.close_write
f.print "nowhere"
```

produces:

```
prog.rb:3:in `write': not opened for writing (IOError)
from /tmp/prog.rb:3:in `print'
from /tmp/prog.rb:3:in `<main>'
```

closed?

io.closed? → true or false

Returns true if *io* is completely closed (for duplex streams, both reader and writer) and returns false otherwise.

```
f = File.new("testfile")
f.close         # =>   nil
f.closed?       # =>   true
f = IO.popen("/bin/sh","r+")
f.close_write   # =>   nil
f.closed?       # =>   false
f.close_read    # =>   nil
f.closed?       # =>   true
```

each

io.each(*separator*=$/) {| *line* | *block* } → *io*
io.each(*limit*) {| *line* | *block* } → *io*
io.each(*separator*, *limit*) {| *line* | *block* } → *io*
io.each(*args*..) → *enum*

1.9

Executes the block for every line in *io*, where lines are separated by *separator*. If *separator* is nil, the entire file is passed as a single string. If the *limit* argument is present and positive, at most that many characters will be returned in each iteration. If only the *limit* argument is given and that argument is negative, then encodings will be ignored while looking for the record separator, which increases performance.

Returns an enumerator if no block is given.

```
f = File.new("testfile")
f.each {|line| puts "#{f.lineno}: #{line}" }
```

produces:

```
1: This is line one
2: This is line two
3: This is line three
4: And so on...
```

each_byte

io.each_byte {| *byte* | *block* } → nil
io.each_byte → *enum*

1.9

Calls the given block once for each byte (a Fixnum in the range 0 to 255) in *io*, passing the byte as an argument. The stream must be opened for reading or an IOerror will be raised. Returns an enumerator if no block is given.

```
f = File.new("testfile")
checksum = 0
f.each_byte {|x| checksum ^= x }   # =>   #<File:testfile>
checksum                           # =>   12
```

each_char

io.each_char {| *char* | *block* } → nil
io.each_char → *enum*

1.9

Calls the given block passing it each character (a string of length 1) in *io*. The stream must be opened for reading or an IOerror will be raised. Returns an enumerator if no block is given.

```
f = File.new("testfile")
result = []
f.each_char {|ch| result << ch}   # =>   #<File:testfile>
result[0, 10]                     # =>   ["T", "h", "i", "s", " ", "i",
                                         "s", " ", "l", "i"]
```

each_line

io.each_line(...) {| *line* | *block* } → *io*

Synonym for IO#each.

eof

io.eof → true or false

Returns true if *io* is at end of file. The stream must be opened for reading or an IOError will be raised.

```
f = File.open("testfile")
f.eof   # =>   false
dummy = f.readlines
f.eof   # =>   true
```

eof?

io.eof? → true or false

Synonym for IO#eof.

external_encoding

io.external_encoding → *encoding*

1.9

Returns the encoding object representing the external encoding of this I/O object.

```
io = File.open("testfile", "r:utf-8:iso-8859-1")
io.external_encoding   # =>   #<Encoding:UTF-8>
io.internal_encoding   # =>   #<Encoding:ISO-8859-1>
```

fcntl

io.fcntl(*cmd*, *arg*) → *int*

Provides a mechanism for issuing low-level commands to control or query file-oriented I/O streams. Commands (which are integers), arguments, and the result are platform dependent. If *arg* is a number, its value is passed directly. If it is a string, it is interpreted as a binary sequence of bytes. On Unix platforms, see fcntl(2) for details. The Fcntl module provides symbolic names for the first argument (see page 753). Not implemented on all platforms.

fileno

io.fileno → *int*

Returns an integer representing the numeric file descriptor for *io*.

```
STDIN.fileno    # =>   0
STDOUT.fileno   # =>   1
```

flush

io.flush → *io*

Flushes any buffered data within *io* to the underlying operating system (note that this is Ruby internal buffering only; the OS may buffer the data as well).

```
STDOUT.print "no newline"
STDOUT.flush
```

produces:

```
no newline
```

fsync *io*.fsync → 0 or nil

Immediately writes all buffered data in *io* to disk. Returns nil if the underlying operating system does not support *fsync(2)*. Note that fsync differs from using IO#sync=. The latter ensures that data is flushed from Ruby's buffers but does not guarantee that the underlying operating system actually writes it to disk.

getbyte *io*.getbyte → *fixnum* or nil

1.9

Returns the next 8-bit byte (as opposed to an encoded character) from *IO* or returns nil at end of file. See also IO#bytes.

```
file = File.open("testfile")
file.getbyte   # =>   84
file.getbyte   # =>   104
```

getc *io*.getc → *string* or nil

1.9

Gets the next character from *io*. Returns nil if called at end of file.

```
f = File.new("testfile")
f.getc   # =>   "T"
f.getc   # =>   "h"
```

gets *io*.gets(*separator*=$/) → *string* or nil
io.gets(*limit*) → *string* or nil
io.gets(*separator*, *limit*) → *string* or nil

1.9

Reads the next "line" from the I/O stream; lines are separated by *separator*. A separator of nil reads the entire contents, and a zero-length separator reads the input a paragraph at a time (two or more successive newlines in the input separate paragraphs). If *separator* is nil, the entire file is passed as a single string. If the *limit* argument is present and positive, at most that many characters will be returned in each iteration. If only the *limit* argument is given and that argument is negative, then encodings will be ignored while looking for the record separator, which increases performance. The line read in will be returned and also assigned to $_ (although the setting of $_ is considered ugly—it may be removed in future). Returns nil if called at end of file.

```
file = File.new("testfile")
file.gets              # =>   "This is line one\n"
$_                     # =>   "This is line one\n"
file.gets(10)          # =>   "This is li"
file.gets("line")      # =>   "ne two\nThis is line"
file.gets("line", 4)   # =>   " thr"
```

internal_encoding *io*.internal_encoding → *encoding*

1.9

Returns the encoding object representing the internal encoding of this I/O object.

```
io = File.open("testfile", "r:utf-8:iso-8859-1")
io.external_encoding   # =>   #<Encoding:UTF-8>
io.internal_encoding   # =>   #<Encoding:ISO-8859-1>
```

ioctl

io.ioctl(*cmd*, *arg*) → *int*

Provides a mechanism for issuing low-level commands to control or query I/O devices. The command (which is an integer), arguments, and results are platform dependent. If *arg* is a number, its value is passed directly. If it is a string, it is interpreted as a binary sequence of bytes. On Unix platforms, see ioctl(2) for details. Not implemented on all platforms.

isatty

io.isatty → true or false

Returns true if *io* is associated with a terminal device (tty) and returns false otherwise.

```
File.new("testfile").isatty   # =>   false
File.new("/dev/tty").isatty   # =>   true
```

lineno

io.lineno → *int*

Returns the current line number in *io*. The stream must be opened for reading. lineno counts the number of times gets is called, rather than the number of newlines encountered. The two values will differ if gets is called with a separator other than newline. See also the $. variable.

```
f = File.new("testfile")
f.lineno   # =>   0
f.gets     # =>   "This is line one\n"
f.lineno   # =>   1
f.gets     # =>   "This is line two\n"
f.lineno   # =>   2
```

lineno=

io.lineno = *int* → *int*

Manually sets the current line number to the given value. $. is updated only on the next read.

```
f = File.new("testfile")
f.gets                     # =>   "This is line one\n"
$.                         # =>   1
f.lineno = 1000
f.lineno                   # =>   1000
$. # lineno of last read   # =>   1
f.gets                     # =>   "This is line two\n"
$. # lineno of last read   # =>   1001
```

lines

io.lines(*separator*=$/) → *enumerator*
io.lines(*limit*) → *enumerator*
io.lines(*separator*, *limit*) → *enumerator*

1.9

Returns an enumerator which allows iteration over the lines in *io*, where lines are terminated by *separator*. If *separator* is nil, the entire file is passed as a single string. If the *limit* argument is present and positive, at most that many characters will be returned in each iteration. If only the *limit* argument is given and that argument is negative, then encodings will be ignored while looking for the record separator, which increases performance.

pid

io.pid → *int*

Returns the process ID of a child process associated with *io*. This will be set by IO.popen.

```
pipe = IO.popen("-")
if pipe
  STDERR.puts "In parent, child pid is #{pipe.pid}"
else
  STDERR.puts "In child, pid is #{$$}"
end
```

produces:

```
In parent, child pid is 84605
In child, pid is 84605
```

pos *io*.pos → *int*

Returns the current offset (in bytes) of *io*.

```
f = File.new("testfile")
f.pos    # =>  0
f.gets   # =>  "This is line one\n"
f.pos    # =>  17
```

pos= *io*.pos = *int* → 0

Seeks to the given position (in bytes) in *io*.

```
f = File.new("testfile")
f.pos = 17
f.gets   # =>  "This is line two\n"
```

print *io*.print(⟨ *obj*=$_ ⟩*) → nil

Writes the given object(s) to *io*. The stream must be opened for writing. If the output record separator ($\) is not nil, it will be appended to the output. If no arguments are given, prints $_. Objects that aren't strings will be converted by calling their to_s method. Returns nil.

```
STDOUT.print("This is ", 100, " percent.\n")
```

produces:

```
This is 100 percent.
```

printf *io*.printf(*format* ⟨ , *obj* ⟩*) → nil

Formats and writes to *io*, converting parameters under control of the format string. See Kernel#sprintf on page 577 for details.

putc *io*.putc(*obj*) → *obj*

Writes the given character (the first byte from String or a Fixnum) on *io*. Note that this is not encoding safe, because the byte may be just part of a multibyte sequence.

```
STDOUT.putc "ABC"
STDOUT.putc 65
```

produces:

```
AA
```

puts *io*.puts(⟨ *obj* ⟩*) → nil

Writes the given objects to *io* as with IO#print. Writes a newline after any that do not already end with a newline sequence. If called with an array argument, writes each element on a new line. If called without arguments, outputs a single newline.

```
STDOUT.puts("this", "is", "a", "test")
```

produces:

```
this
is
a
test
```

read *io*.read(⟨ *int* ⟨ , *buffer* ⟩ ⟩) → *string* or nil

Reads at most *int* bytes from the I/O stream or to the end of file if *int* is omitted. Returns nil if called at end of file. If *buffer* (a String) is provided, it is resized accordingly, and input is read directly into it.

```
f = File.new("testfile")
f.read(16)        # =>   "This is line one"
str = "cat"
f.read(10, str)   # =>   "\nThis is l"
str               # =>   "\nThis is l"
```

readbyte *io*.getbyte → *fixnum*

1.9 Returns the next 8-byte byte (as opposed to an encoded character) from *IO*, raising EOFError at end of file. See also IO#bytes.

readchar *io*.readchar → *string*

Reads a character as with IO#getc but raises an EOFError on end of file.

readline *io*.readline(*separator*=$/) → *string* or nil
io.readline(*limit*) → *string* or nil
io.readline(*separator*, *limit*) → *string* or nil

1.9 Reads a line as with IO#gets, but raises an EOFError on end of file.

readlines *io*.readlines(*separator*=$/) → *array*
io.readlines(*limit*) → *array*
io.readlines(*separator*, *limit*) → *array*

1.9 Returns all of the lines in *io* as an array. Lines are separated by the optional *separator*. If *separator* is nil, the entire file is passed as a single string. If the *limit* argument is present and positive, at most that many characters will be returned in each iteration. If only the *limit* argument is given and that argument is negative, then encodings will be ignored while looking for the record separator, which increases performance.

```
f = File.new("testfile")
f.readlines           # =>  ["This is line one\n", "This is line two\n",
                           "This is line three\n", "And so on...\n"]
f = File.new("testfile")
f.readlines("line")   # =>  ["This is line", " one\nThis is line", "
                           two\nThis is line", " three\nAnd so on...\n"]
f = File.new("testfile")
f.readlines(10)       # =>  ["This is li", "ne one\n", "This is li", "ne
                           two\n", "This is li", "ne three\n", "And so
                           on.", "..\n"]
```

readpartial

io.readpartial(*limit*, *result*="") → *result*

1.9

Data read from files and devices is normally buffered. When reading line by line (for example using IO#gets), Ruby will read many lines at a time into an internal buffer and then return lines from that buffer. This buffering is normally transparent—Ruby will refill the buffer automatically when required. However, when reading from a device or pipe (as opposed to a file), you sometimes want to read whatever is in the buffer, reading from the device or pipe only if the buffer is empty when the read starts. This is what readpartial does. If any data is available in local buffers, it will be returned immediately. readpartial will read from the device or pipe (potentially blocking) only if the buffer is empty. Raises EOFError when it reached EOF. See also IO#read_nonblock.

The following example comes from the internal documentation, with thanks to the anonymous author:

```
r, w = IO.pipe           #               buffer          pipe content
w << "abc"               #               ""              "abc".
r.readpartial(4096)      #=> "abc"       ""              ""
r.readpartial(4096)      # blocks because buffer and pipe is empty.
r, w = IO.pipe           #               buffer          pipe content
w << "abc"               #               ""              "abc"
w.close                  #               ""              "abc" EOF
r.readpartial(4096)      #=> "abc"       ""              EOF
r.readpartial(4096)      # raises EOFError
r, w = IO.pipe           #               buffer          pipe content
w << "abc\ndef\n"        #               ""              "abc\ndef\n"
r.gets                   #=> "abc\n"     "def\n"         ""
w << "ghi\n"             #               "def\n"         "ghi\n"
r.readpartial(4096)      #=> "def\n"     ""              "ghi\n"
r.readpartial(4096)      #=> "ghi\n"     ""              ""
```

read_nonblock

io.readpartial(*limit*, *result*="") → *result*

1.9

Effectively the same as IO#readpartial, except in cases where no buffered data is available. In this case, it puts *io* into nonblocking mode before attempting to read data. This means that the call may return EAGAIN and EINTR errors, which should be handled by the caller.

reopen

io.reopen(*other_io*) → *io*
io.reopen(*path*, *mode*) → *io*

Reassociates *io* with the I/O stream given in *other_io* or to a new stream opened on *path*. This may dynamically change the actual class of this stream.

```
f1 = File.new("testfile")
f2 = File.new("testfile")
f2.readlines[0]   # =>   "This is line one\n"
f2.reopen(f1)     # =>   #<File:testfile>
f2.readlines[0]   # =>   "This is line one\n"
```

rewind

io.rewind → 0

Positions *io* to the beginning of input, resetting lineno to zero.

```
f = File.new("testfile")
f.readline   # =>   "This is line one\n"
f.rewind     # =>   0
f.lineno     # =>   0
f.readline   # =>   "This is line one\n"
```

seek

io.seek(*int*, *whence*=SEEK_SET) → 0

Seeks to a given offset *int* in the stream according to the value of *whence*.

IO::SEEK_CUR	Seeks to *int* plus current position
IO::SEEK_END	Seeks to *int* plus end of stream (you probably want a negative value for *int*)
IO::SEEK_SET	Seeks to the absolute location given by *int*

```
f = File.new("testfile")
f.seek(-13, IO::SEEK_END)   # =>   0
f.readline                  # =>   "And so on...\n"
```

set_encoding

io.set_encoding(*external*, *internal=external*) → *io*
io.set_encoding(*"external-name:internal-name"*) → *io*

Sets the external and internal encodings for *io*. In the first form, encodings can be specified by name (using strings) or as encoding objects. In the second form, the external and internal encoding names are separated by a colon in a string.

```
f = File.new("testfile")
f.internal_encoding                       # =>   nil
f.external_encoding                       # =>   #<Encoding:UTF-8>
f.set_encoding("ascii-8bit:iso-8859-1")   # =>   #<File:testfile>
f.internal_encoding                       # =>   #<Encoding:ISO-8859-1>
f.external_encoding                       # =>   #<Encoding:ASCII-8BIT>
```

stat

io.stat → *stat*

Returns status information for *io* as an object of type File::Stat.

```
f = File.new("testfile")
s = f.stat
"%o" % s.mode   # =>   "100644"
s.blksize       # =>   4096
s.atime         # =>   2009-04-13 13:26:28 -0500
```

sync *io*.sync → true or false

Returns the current "sync mode" of *io*. When sync mode is true, all output is immediately flushed to the underlying operating system and is not buffered by Ruby. See also IO#fsync.

sync= *io*.sync = *bool* → true or false

Sets the "sync mode" to true or false. When sync mode is true, all output is immediately flushed to the underlying operating system and is not buffered internally. Returns the new state. See also IO#fsync.

```
f = File.new("testfile")
f.sync = true
```

sysread *io*.sysread(*int* ⟨ , *buffer* ⟩) → *string*

Reads *int* bytes from *io* using a low-level read and returns them as a string. If *buffer* (a String) is provided, input is read directly in to it. Do not mix with other methods that read from *io*, or you may get unpredictable results. Raises SystemCallError on error and EOFError at end of file.

```
f = File.new("testfile")
f.sysread(16)        # =>   "This is line one"
str = "cat"
f.sysread(10, str)   # =>   "\nThis is l"
str                  # =>   "\nThis is l"
```

sysseek *io*.sysseek(*offset*, *whence*=SEEK_SET) → *int*

Seeks to a given *offset* in the stream according to the value of *whence* (see IO#seek for values of *whence*). Returns the new offset into the file.

```
f = File.new("testfile")
f.sysseek(-13, IO::SEEK_END)   # =>   53
f.sysread(10)                  # =>   "And so on."
```

syswrite *io*.syswrite(*string*) → *int*

Writes the given string to *io* using a low-level write. Returns the number of bytes written. Do not mix with other methods that write to *io*, or you may get unpredictable results. Raises SystemCallError on error.

```
f = File.new("out", "w")
f.syswrite("ABCDEF")   # =>   6
```

tell *io*.tell → *int*

Synonym for IO#pos.

to_i

io.to_i → *int*

Synonym for IO#fileno.

to_io

io.to_io → *io*

Returns *io*.

tty?

io.tty? → true or false

Synonym for IO#isatty.

ungetbyte

io.ungetbyte(*string* or *int*) → nil

1.9

Pushes back one or more bytes onto *io*, such that a subsequent buffered read will return them. Has no effect with unbuffered reads (such as IO#sysread).

```
f = File.new("testfile")   # =>  #<File:testfile>
c = f.getbyte              # =>  84
f.ungetbyte(c)             # =>  nil
f.getbyte                  # =>  84
f.ungetbyte("cat")         # =>  nil
f.getbyte                  # =>  99
f.getbyte                  # =>  97
```

ungetc

io.ungetc(*string*) → nil

Pushes back one or more characters onto *io*, such that a subsequent buffered read will return them. Has no effect with unbuffered reads (such as IO#sysread).

```
# encoding: utf-8
f = File.new("testfile")   # =>  #<File:testfile>
c = f.getc                 # =>  "T"
f.ungetc(c)                # =>  nil
f.getc                     # =>  "T"
f.ungetc("δog")            # =>  nil
f.getc                     # =>  "δ"
f.getc                     # =>  "o"
```

write

io.write(*string*) → *int*

Writes the given string to *io*. The stream must be opened for writing. If the argument is not a string, it will be converted to a string using to_s. Returns the number of bytes written.

```
count = STDOUT.write( "This is a test\n" )
puts "That was #{count} bytes of data"
```

produces:

```
This is a test
That was 15 bytes of data
```

write_nonblock

io.write_nonblock(*string*) → *int*

Writes the given string to *io* after setting *io* into nonblocking mode. The stream must be opened for writing. If the argument is not a string, it will be converted to a string using to_s. Returns the number of bytes written. Your application should expect to receive errors typical of nonblocking I/O (including EAGAIN and EINTR).