READ RUBY 1.9

Free ebook about the Ruby 1.9 programming language

ruby.runpaint.org

2011-01-28 20:02:10 +0000

CONTENTS

Language

API

LANGU

PROGRAMS

Every entity in Ruby is an object. Objects can remember things and communicate with each other by sending and receiving messages [Goldberg76, pp. 44–44] . A Ruby program describes how certain objects must communicate to achieve some definite ends. This chapter provides an overview of the structure and interpretation of Ruby programs.

Lexical Structure

Tokens[1] are "the mark or series of marks that denote one symbol or word in the language" [Fischer92, pp. 59–62] . A Ruby program consists of a combination of the following tokens: comments, literals, punctuation, identifiers, and keywords.

Comments

Comments are remarks which do not affect the meaning of a program. They are introduced with a number sign (`U+0023`) and continue until the end of the line: the text between # and the end of the line is ignored by the interpreter. Comments are not recognised inside of string/regexp literals—they are interpreted literally—however, regexps support an alternative form of embedded comment. There is no specific construct for multiline comments, but they may be approximated with embedded documentation.

Embedded Documentation

Embedded documentation is a portion of a source file that contains documentation intended for a postprocessor such as `rdoc`, and as such is

1. `Ripper` is a class in the standard library for parsing and analysing Ruby. The `Ripper.tokenize` method takes a string of code as an argument and returns an array of its constituent tokens. Experiment with this method in IRB (`irb -rripper`) to test your assumptions of how tokenization works.

ignored by the interpreter. It is introduced by a line beginning `=begin`, that is an equals sign (`U+003D`) followed by the string `begin`, and continues until a line beginning `=end` is encountered.

Both `=begin` and `=end` may be followed by arbitrary text, which is included in the embedded documentation, as long as it is preceded by a whitespace character. Conventionally, the text following `=begin` names the tool for which the documentation is intended.

Whitespace

Whitespace consists only of `U+0009`, `U+000B`–`U+000D`, and the space character (`U+0020`), i.e. ASCII whitespace other than the newline. Its primary role in Ruby syntax is to separate tokens and terminate statements. When whitespace is syntactically significant it is typically collapsed to a single space. The few areas of syntax where whitespace has different semantics are clearly labeled.

Newlines may function as whitespace, too, depending on the context in which they are used. See the Statement Terminators & Newlines for further details.

Literals

An *object literal* is a syntactical shortcut for the instantiation of a particular core object. Literals exist for `Arrays`, `Hashes`, `Numerics`, `Procs`, `Ranges`, `Regexps`, `Strings`, and `Symbols`.

Identifiers

An *identifier* is the name of a variable or method. It must not contain any US-ASCII character other than the alphanumerics (A—Z, a—z, 0–9) and the low line (`U+005F`), or begin with a US-ASCII digit. However, it may contain any other character legal in the source encoding.

Syntactical Structure

Expressions

An *expression*[2] is a syntactical construct that produces a value. Since every entity is an object, every expression evaluates to an object. It is either primary or compound.

Primary expressions comprise atomic[3] units such as variable references and numeric, string[4], and symbol, regexp[5], and keyword, literals.

Operators

Primary expressions can be combined with *operators* to produce *compound expressions*. An operator is a token with *precedence*, *associativity*, and *arity*, which operates upon one or more values (termed its *operands*).

Precedence dictates which of two different operations should be carried out first. It can be overridden by grouping sub-expressions that should be performed earlier with parentheses. When parenthetical groups contain other parenthetical groups, the innermost is given the highest precedence.

If both operators have the same precedence, the tie is broken by considering their associativity: left-associative expressions are evaluated left to right; right-associative expressions are evaluated right to left. If two operators have the same precedence and are both non-associative, they cannot be used in the same expression without parenthesising one or both.

---

2. We follow the lead of Aho, Sethi, & Ullman in defining *expression* by recursion rather than direct description [Aho86] .
3. As Turbak, Gifford, & Sheldon note, this characterisation is patently false even for toy languages since, in their example, “numerals can be broken down into digits”, however we, too, will “ignore this detail” [Turbak08, pp. 20–22] .
4. Double-quoted strings that interpolate other expressions are compound expressions.
5. Regexps that interpolate other expressions are also compound expressions.

An operator's arity is the number of arguments it takes. An arity of 1 makes an operator *unary*, 2, *binary*, and 3, *ternary*.

Operators	Arity	Associativity	Function
`!`, `~`, `+`	Unary	Right	NOT, bitwise complement, unary plus.
`**`	Binary	Right	Exponentiation
`-`	Unary	Right	Unary minus.
`*`, `/`, `%`	Binary	Left	Multiplication, division, modulus.
`+`, `-`	Binary	Left	Addition, subtraction.
`<<`, `>>`	Binary	Left	Left-shift or append, right-shift.
`&`	Binary	Left	Bitwise AND.
`|`, `^`	Binary	Left	Bitwise OR, Bitwise XOR.
`<`, `<=`, `>=`, `>`	Binary	Left	Inequalities.
`==`, `===`, `!=`, `=~`, `=~`, `!~`, `<=>`	Binary	None	Equality and comparison.
`&&`	Binary	Left	AND
`||`	Binary	Left	OR
`..`, `...`	Binary	None	`Range` constructor.
…?…:…	Ternary	Right	Conditional
`rescue`	Binary	Left	Exception handling.
`=`	Binary	Right	Assignment.
`**=`, `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&&=`, `&=`, `||=`, `|=`, `^=`	Binary	Right	Abbreviated assignment.
`defined?`	Unary	None	Variable tests.
`not`	Unary	Right	NOT
`and`, `or`	Binary	Left	AND, OR
`if`, `unless`, `while`, `until`	Binary	None	Statement modifiers.

Operators in descdending order of precedence

Keyword Literals

true

The `true` keyword returns the singleton instance of `TrueClass`. Its value is, by definition, true.

false

The `false` keyword returns the singleton instance of `FalseClass`. Its value is, by definition, false.

nil

The `nil` keyword returns the singleton instance of `NilClass`. Its value is represents the absence of a value. The `Kernel.nil?` predicate returns `true` if its value is `nil`; `false` otherwise.

self

`self` always evaluates to the current object. Outside of any class definition, i.e. at the top-level, the current object is an instance of `Object` called `main`. Inside a class definition, but outside of a method definition, the current object is an instance of `Class`. Within a method definition the current object is the instance of the containing class.

```
self #=> main
self.class #=> Object

class Classy
  self
end #=> Classy

class Classy
  self.class
end #=> Class

class Classy
```

```
  def methodical
    self
  end
end
Classy.new.methodical #=> #<Classy:0x90d2268>
```

__FILE__ / __LINE__

See Tracing.

__ENCODING__

Evaluates to an `Encoding` object representing the current source encoding, i.e. that of __FILE__.

Statements

A statement is an expression whose value is ignored[6]. In practice, this implies that a statement is executed for its side-effects, because an expression executed for neither its value nor effect is semantically meaningless.

Statement Terminators & Newlines

One statement must be separated from the next by a statement terminator. This may be a semicolon (`U+003B`) or newline. The latter is preferred because it leads to a natural separation: each statement on its own line. However, a newline does not terminate a statement if:

- It is immediately preceded by a reverse solidus (`U+005C`).
- It is preceded by an operator, with optional intervening whitespace.
- It is immediately preceded by a comma (`U+002C`) or full stop (`U+002E`) in a message expression or array/hash literal, with optional intervening whitespace.

---

6. Turbak, Gifford, & Sheldon offer a further example of distinguishing between statements and expressions by considering their context [Turbak08, pp. 472–476] .

- It is preceded by a left parenthesis, curly bracket, or square bracket, with optional intervening whitespace. (Allows the argument list of message expressions, and array or hash literals to span multiple lines).
- The first non-whitespace character on the following line is a full stop (`U+002E`). (Allows chained message expressions to span lines).
- It is preceded by one of the following keywords: `alias`, `and`, `begin`, `def`, `defined?`, `case`, `class`, `else`, `elsif`, `ensure`, `for`, `if`, `in`, `module`, `not`, `or`, `then`, `undef`, `unless`, `until`, `when`, or `while`.

__END__

If the interpreter encounters a line consisting solely of the token `__END__`, it ignores any lines that follow. However, they are made available to the program via a global, read-only `File` object named `DATA`.

```
DATA.lines.map{|l| l.split.first}
#=> ["Lovers", "part", "also", "a", "--"]
__END__
Lovers for the most
part are without hope: passion
also is just
a bridge, a means of connection
-- Marina Ivanovna Tsvetaeva, "The Poem of the End"
```

Interpretation

Interpreter

The statements and expressions that comprise a program are collectively known as its *source code*. To execute a program its source code must be provided to a Ruby *interpreter*: the program that executes Ruby source code. The reference implementation[7] of Ruby, or MRI, contains an interpreter called `ruby`, so when we speak of *executing the interpreter* we are referring to running this program.

7. There are several excellent alternative implementations such as JRuby and Rubinius. However, for the purposes of this book neither are recommended because, although JRuby is close, they are not yet compatible with Ruby 1.9.

The `ruby` program is invoked as `ruby` *`options file arguments`*, all of which are optional. If the `RUBYOPT` environment variable includes the `-W`, `-w`, `-v`, `-d`, `-I`, `-r`, and `-K` options, they are treated as if they were specified on the command line.

options is zero or more of the options tabulated below. They are passed to the interpreter. If the `-e` option is given, its argument is the Ruby code to execute. If *file* is given and it's the name of a file containing Ruby code, the file is executed. If *file* is `-` or omitted, the Ruby code to execute is read from standard input. If *arguments* are given, they are passed to the Ruby program as elements of `ARGV`.

Option	Description
`-0`*`n`*	Set the record separator to the character with ASCII code *n*, which is interpreted as up to three octal digits. If *n* is omitted, it is `0`; if *n* explicitly specified as `0`, the record separator is `"\n\n"`; if *n* is `777`, the record separator is `nil`.
`-a`	Auto-split mode: when used in conjunction with `-n` or `-p`, places `$F = $_.split` at the beginning of the loop body.
`-C` *`directory`* / `-X` *`directory`*	Change to the directory named *directory* before executing the program.
`-c`	Check syntax without executing the program: prints an error message if there are syntax errors; prints nothing if there aren't.
`--copyright`	Display the copyright notice then exits.
`-d` / `--debug`	Assign `true` to `$DEBUG` and `$VERBOSE`, which enables warnings.
`--disable-gems`	Don't load *rubygems* implicitly or add gem directories to the load path.
`-E` *`encoding`* / `--encoding` *`encoding`*	Set the default encoding. If *encoding* is a single encoding name, it is the default external encoding; if it is two encoding names separated by colons, the first encoding is the default external, and the second, the default internal.
`-e` *`string`*	Execute *string* as Ruby code. If this option is given multiple times, its arguments form successive lines of the same program.

Option	Description
`-F` *sep*	Set the input field separator (`$;`) to *sep*, where *sep* is a single character or regular expression without the `/…/` delimiters.
`-h` / `--help`	Display usage help then exit.
`-I` *directories*	Prepend *directories* to `$LOAD_PATH`. If *directories* contains multiple directories, they are separated by a colon on Unix-like systems; a semicolon on Windows systems. May be given multiple times.
`-i` *extension*	For each file named in `ARGV`, data written to the standard output stream will be written to that file. If *extension* is given, then before a file, *file*, is modified it is copied to *fileextension*.
`-K`*code*	Set the default external encoding and source encoding to *US-ASCII* if *code* is `a`, `A`, `n`, or `N`; *UTF-8* if *code* is `u` or `U`; *Shift-JIS* is *code* is `s` or `S`; or *EUC-JP* if `code` is `e` or `E`.
`-l`	Set `$\` to `$/` and remove `$/` from the end of input lines.
`-n`	Assume a `while gets; … ; end` loop around the Ruby program.
`-p`	Assume a `while gets; … ; print; end` loop around the Ruby program.
`-r`*library*	Require the library or gem named *library* before executing the program.
`-S`	Try to locate the specified program file relative to the `RUBYPATH` or `PATH` environment variables, before locating it normally.
`-s`	Remove any options following the program filename from `ARGV`, then create a global variable named after the option and assign to it the option's value. Options without values are assumed to be `true`.
`-T`*level*	Set the safe level to *level*, or `1` if *level* is omitted.
`-U`	Set the default internal encoding to UTF-8.
`-v` / `--verbose`	Enable warnings by setting `$VERBOSE` to `true`, then print the interpreter version number. If a program is specified, execute it.
`--version`	Display the interpreter version number then exits.
`-w`	Enable warnings by setting `$VERBOSE` to `true`, then execute the named program—if given—or read the program from standard input.

Option	Description
`-W`*`level`*	Set the warning level to *level*: silence all warnings if *level* is `0`; use standard warning level if *level* is `1`; otherwise, or if *level* is omitted, equivalent to `-w`.
`-x` *`directory`*	Strip each line of the program file that precede a line beginning `#!ruby`, change to the directory named *directory*—if given—then execute the program.

Source Files

Ruby programs are typically stored in plain text files with an `.rb` filename extension. As explained above, they are typically executed by supplying their filenames as arguments to the `ruby` interpreter, e.g. `ruby myfile.rb` would execute the source code saved in `myfile.rb`.

Shebang

The *shebang*[8] is a notation for informing a UNIX-like operating system of the interpreter with which a script should be executed. If present it must appear as the first line of a source file. It begins with a number sign (`U+0023`) making it a legal comment line, therefore ignored by the interpreter, which is followed by an exclamation mark (`U+0021`) then the path to the interpreter.

A typical shebang is `#!/usr/bin/env ruby` which uses `env` to avoid hardcoding the path to the interpreter.

If a script containing a shebang is executable, it may be executed by entering its filename in the shell. This allows the interpreter to be invoked implicitly, and is a common approach.

```
#!/usr/bin/env ruby
puts "The whole shebang"
```

8. A portmanteau of *sharp* and *bang* [Wall00, pp. 1001–1001] , colloquial terms for the number sign (`U+0023`) and the exclamation mark (`U+0021`), respectively.

```
run@paint:~ ➔  chmod +x shebang.rb
run@paint:~ ➔  ./shebang.rb
The whole shebang
```

Source Encoding

Source files are assumed to only contain US-ASCII characters, unless they have been explicitly declared to have a different encoding. This topic is explained fully in Source Encoding.

Warnings

Warnings— notices of deprecated, ambiguous, or otherwise problematic, code— are enabled when the interpreter is given the `-w` switch, e.g. `ruby -w myfile.rb`. For example, Ruby warns when a constant that has already been defined is assigned to.

Loading Features

A Ruby program may be entirely self-contained, in which case all the code it needs is stored in a single source file. Larger programs, however, are often partitioned such that each major component is stored in a separate file. Additionally, programs often reuse existing code written by third parties. In both cases, the program must load and execute these external files, which are collectively termed *features*.

A feature must be resolved to a filename: If it begins with `/` it is already an absolute filename; `../`, it is resolved relative to the current working directory; `~/`, it is resolved relative to the user's home directory; otherwise, it is resolved relative to a directory in the *load path*. The load path is an `Array`—named `$LOAD_PATH` or `$:`—of directories searched for a given feature. It is initialised by the interpreter, and modified with the `-I` option or by manipulating `$LOAD_PATH` directly.

require

`Kernel.require(`*`feature`*`)` attempts to load and execute *feature*. *feature* is a `String` that can be resolved to the name of an file with a supported

extension: either `.rb`—implying it contains Ruby source code—or one of the system's shared library extensions—implying it is a binary extension.

feature is resolved as explained above. This is repeated for each supported extension by first appending the extension to *feature*. Finally, *feature* is searched for in the *Gem path*[9]. When an existing filename is found, the remaining steps are skipped; if all fail, a `LoadError` is raised.

If the filename appears in an `Array` named `$LOADED_FEATURES`—alias: `$"`—it has already been required so `require` returns `false`; otherwise, the filename is loaded with `Kernel.load` at a safe level of `0`. If the filename was loaded successfully, it is appended to `$LOADED_FEATURES` and `true` is returned.

require_relative

`Kernel.require_relative` behaves like `require`, except it resolves the feature name relative to the file in which it is contained. `require_relative` *`path`* is equivalent to `require File.expand_path(File.dirname(__FILE__)) + `*`path`*.

load

`Kernel.load(`*`feature`*`)` resolves *feature* to a filename, which it then executes. Unlike *require*, *feature* can't omit the filename extension, must contain Ruby source code, is not searched for in the Gem path, is loaded even if appearing in `$LOADED_FEATURES`, and is loaded at the current safe level.

Once a file is loaded, its constants—therefore class and module definitions—method definitions, and global-, class-, and instance variables are imported into the loading environment. If the optional second parameter is `true`, the file is loaded into an anonymous module, to avoid polluting the caller's environment.

---

9. This is returned by `gem environment gempath`. Each directory in this path, has its `/gems/` sub-directory searched by appending `/lib/`*`feature`* to its name. If unsuccessful, it is repeated after having appended each supported extension—`.rb` first—to *feature*.

For example, consider a file, `a.rb`, that comprised the statement `@ivar = :i`. Located in the same directory is the following file:

```
load './a.rb' #=> true
@ivar #=> :i
@ivar = :j
load './a.rb' #=> true
@ivar #=> :i
```

IRB

Ruby is distributed with a program called `irb` which provides an interactive shell, or read-eval-print loop, for the interpreter. IRB works as follows:

1. You enter a statement of Ruby and press `Enter`.
2. That statement is evaluated and its value printed to the screen.
3. You go back to step 1.

This provides a superb environment for learning Ruby. As you read this book you can enter the examples in IRB and see for yourself how they work.

Evaluating Strings

Source code can be provided as an command-line argument to the interpreter if it is invoked with the `-e` switch, e.g. `ruby -e 'puts 1 + 2'` executes the code fragment and displays `3`.

The `Kernel.eval` method provides similar functionality from within a program. Pass it an arbitrary string of source code as an argument and it will return the result. Continuing with the above example, `eval 'puts 1 + 2'`, has the same result.

Tracing

Although Ruby does not include a debugger, she offers a variety of features to aid debugging. The keyword `__FILE__` evaluates to a `String` naming the

source file currently being executed. The strings `(eval)` and `(irb)` are returned when in an `eval` context and IRB session, respectively. The keyword `__LINE__` evaluates to a `Fixnum` specifying the line number in the current `__FILE__` being executed. Taken together, thee keywords can be used to produce error messages and warnings that identify the errant code. Indeed, `Kernel.eval`, `Object.instance_eval`, and `Module.class_eval`, accept a filename and line number as their final two arguments, which they use when reporting errors: by using `__FILE__` and `__LINE__` as these values, it becomes easier to trace dynamically generated code.

Similarly, `Kernel.__method__`, and its alias: `Kernel.__callee__`, return the name of the current method as a `Symbol`. If the current method was invoked via an alias, its original name is returned.

`Kernel.caller` returns a stack trace that culminates with the method that invoked the current method. The `Array` returned has one element per stack frame, organised in reverse chronological order. Each frame is represented as a `String`, which usually includes the filename, line number, and method name. If `caller` is given an argument, it specifies how many frames to drop from the beginning.

If a global constant named `SCRIPT_LINES__` is assigned a `Hash`, `Kernel.require`, `Kernel.require_relative`, and `Kernel.load`, append an entry to it for each file they load. The key is the filename as a `String`, and the value is that file's contents, also as a `String`.

Finally, `Kernel.set_trace_func(`*proc*`)` registers the given `Proc` to be called when an event occurs. It receives up to six arguments: the event name, a filename, a line number, an object ID, a binding, and a class name. If *proc* is `nil`, tracing is disabled.

Event	Description
c-call	A method written in C is invoked.
c-return	A method written in C returns.
call	A method written in Ruby is invoked.
class	A `Class` or `Module` is opened.
line	A new line of Ruby code is executed.
raise	An exception is raised.

Event	Description
return	A method written in Ruby returns.

VARIABLES

A *variable* is a named storage location that holds a value. It is named with an identifier, possibly preceded by a *sigil* (a symbol denoting the variable's scope).

When the name of a variable appears somewhere other than the left-hand side of an assignment expression it is a *variable reference* which evaluates to the variable's value.

A variable's *scope* "...is the portion of the program text in which the variable may be referenced." [Turbak08, pp. 334–335] . Its *lifetime* is "the duration, during a run of a program, during which a location is allocated as the result of a specific declaration." [Mitchell04, pp. 167–167] . When the scope of a variable "contains another declaration of the same name the inner declaration *carves out a hole* in the scope of the outer one" [Turbak08, pp. 336–338] . The outer's lifetime persists through the inner declaration, but its scope is hidden for the duration.

The specifics of variable references are explained in the following sections, along with complementary treatment of variable scope, initialization, and assignment.

Constants

A constant is a variable whose value, once assigned, is not expected to change. Its constancy is not enforced by Ruby, so repeated assignments are legal, but cause a warning to be issued.

Constants are, by definition, named with an identifier whose first character is an uppercase US-ASCII character (`A`–`Z`). It is a strong convention that constants qua constants are named entirely in US-ASCII uppercase letters with low lines to separate words, whereas constants used to name classes or modules are named in camel-case: title case with whitespace removed.

They only come into existence when they are assigned a value. Therefore a constant, unlike the other variables, is never in an uninitialized state. Referencing a constant that does not exist results in a `NameError`.

Constant	Class	Meaning
`DATA`	`IO`	If the source file contains the line `__END__`, the lines that follow are accessible by reading from `DATA`.
`FALSE`	`FalseClass`	`false`
`NIL`	`NilClass`	`nil`
`RUBY_COPYRIGHT`	`String`	Copyright statement for the interpreter, e.g. `ruby - Copyright (C) 1993-2010 Yukihiro Matsumoto.`
`RUBY_DESCRIPTION`	`String`	Version number and architecture of the interpreter, e.g. `ruby 1.9.2dev (2010-02-19 trunk 26715) [i686-linux].`
`RUBY_ENGINE`	`String`	The implementation of the interpreter, e.g. `ruby`. **`ruby`** MRI: official implementation. **`rbx`** Rubinius. **`macruby`** MacRuby. **`ironruby`** IronRuby. **`jruby`** JRuby.

Predefined global constants

Constant	Class	Meaning
		maglev MagLev
RUBY_PATCHLEVEL	Fixnum	Patch level of the interpreter, e.g. -1.
RUBY_PLATFORM	String	Platform for which the interpreter was built, e.g. i686-linux.
RUBY_RELEASE_DATE	String	Release date, for point releases, or build date, for trunk builds, of the interpreter, e.g. 2010-02-19.
RUBY_REVISION	Fixnum	Revision of the interpreter, e.g. 26715. (SVN revision number on MRI).
RUBY_VERSION	String	Version number of the interpreter, e.g. 1.9.2.
STDERR	IO	Standard error stream. Initial value of $stderr.
STDIN	IO	Standard input stream. Initial value of $stdin.
STDOUT	IO	Standard output stream. Initial value of $stdout.
SCRIPT_LINES__	Hash	If assigned a Hash, each subsequent file loaded or required will create an entry in the hash, whose key is the filename as a String, and value is the file's lines as an Array of Strings.
TOPLEVEL_BINDING	Binding	Represents the top-level execution environment, i.e. outside of any class, module, method, block, or other construct.
TRUE	TrueClass	true

References

A *constant reference* is an expression evaluating to the named constant. The simplest constant reference is a primary expression consisting solely of the constant's name, e.g. Constant.

A constant reference may also be qualified by prefixing the constant name with the "*scope operator*" [Thom09, pp. 336–339] : two consecutive colon

(U+003A) characters. This causes the constant to be looked up in the global scope[1].

A reference may be qualified further by preceding the scope operator with an expression evaluating to the `Class` or `Module` object in which the constant was defined.

```
module Math
  PLASTIC = cbrt(0.5 + Rational(1, 6) * sqrt(Rational(23, 3))) +
            cbrt(0.5 - Rational(1, 6) * sqrt(Rational(23, 3)))
end
Math::PLASTIC #=> 1.3247179572447458
```

Resolution Algorithm

To resolve an unqualified constant reference, *const*, made in a class or module named *container*:

1. Let *original-container* equal *container*.
2. If *const* is defined in *container*, we've successfully resolved it.
3. If *container* has an enclosing class or module, set *container* to its name and repeat step 2.
4. Set *container* to *original-container*.
5. In the order that they were included, inspect each module mixed-in to *container*. If one contains *const*, we've successfully resolved it.
6. If *container* has a superclass assign it to *container*; otherwise go to step 8.
7. If *const* is defined in *container*, we've successfully resolved it; otherwise, go to step 5.
8. Set *container* equal to `Object`, if it hasn't already been, and repeat step 5.
9. Set *container* to *original-container*
10. If *container* responds to `:const_missing`, we've resolved *const* to the value of *container*`.const_missing(`*const*`)`.
11. If *container* has a superclass, assign it to *container* and repeat step 10.
12. Raise a `NameError` exception: constant resolution failed.

---

1. That is, under `Object` or `Kernel`, with the former taking precedence. See Resolution Algorithm for more details.

This search path can be summarised as: `(Module.nesting + self.ancestors + Object.ancestors).uniq` (executed in the context of *original-container*).

Some notes on this procedure:

- In the search for a given constant no class or module is examined more than once: if a previously-seen location is suggested by the above algorithm above, it is skipped.
- Step 8 is needed in the case where *original-container* is a module: modules don't inherit from `Object`, so the preceding steps wouldn't have searched it.
- When a constant is referenced in an `*_eval` or `*_exec` block, *container* is set to that enclosing the block; not the container in whose context the block is evaluated.
- When *original-container*'s ancestors are searched for `const_missing`, they are *not* also sent `method_missing`.

Scope

We can restate the algorithm above to deduce a constant's scope, namely: it is established by its lexically enclosing class or module. Inner classes and modules inherit the scope of their parents, and constants initialized in the former cause a hole in their parent's scope.

A constant is accessible throughout its scope by unqualified reference, and may be accessed from an exterior context by qualifying its name with that of the defining class/module.

A constant defined in another context may be referred to unqualified or prefixed with the scope operator. However, it is illegal to define a constant in a method body as every invocation of the method would cause re-assignment, defeating the purpose of a constant.

A qualified constant name may be used as an lvalue, allowing constants to be defined in the context of a class or module from outside.

Missing Constants

As noted in Resolution Algorithm, before a search terminates a `:const_missing` message is sent. This allows classes and modules to create constants on-the-fly or otherwise influence the lookup process.

```
module Roman
  ARABIC = {'I' => 1,   'V' => 5,   'X' => 10, 'L' => 50,
            'C' => 100, 'D' => 500, 'M' => 1000          }
  def self.const_missing(name)
    digits = [*name.to_s.upcase.chars].reverse.map{|r| ARABIC[r] or super}
    digits.map.with_index {|d,i| digits[i.zero? ? i : i-1] > d ? -d : d}.reduce(:+)
  end
end
Roman::XLVII #=> 47
```

Reflection

The `Module#constants` method returns an `Array` of `Symbols` naming the constants defined in the receiver.

The value of a constant may be retrieved from a given class or module by supplying its name to the `Module#const_get` method, e.g. `Float.const_get :INFINITY #=> Infinity`. Similarly, the value of a constant may be set with `Module#const_set`, which takes two arguments: the name of the constant and its new value.

A predicate method, `Module#const_defined?`, exists for determining whether a class or module defines a given constant.

Both `#const_get` and `const_defined?` accept an optional second argument to control whether they look for inherited constants. By default this argument is `true`, but if it is set to `false` the method only considers constants defined directly in the receiver. Lastly, the private method `Module#remove_const` takes the name of a constant to remove from the receiver.

Local Variables

A local variable is named by an identifier whose first character is a lowercase US-ASCII character (`a`–`z`) or low line (`U+005F`). Conventionally, the name consists of lowercase words separated by low lines.

It is initialized if it appears on the left-hand side (before the equals sign (`U+003D`)) of an assignment expression, even if the expression does not actually execute. Variables of the latter sort have the value `nil`.

Attempting to use an uninitialized local variable causes the identifier to be interpreted as a message selector which is sent to the current implicit receiver. If such a method doesn't exist a `NameError` is raised.

Scope

The scope of a local variable is *static* in that it "can always be determined from the abstract syntax tree of a program" [Turbak08, pp. 334–335] . It is established by the block, method/class/module definition, or top-level program—hereafter a *scope-defining* construct— which lexically encloses its assignment. If a scope-defining construct itself contains a scope-defining construct, the local variables of the former are not visible in the latter: the inner construct "carves out a hole in the scope of the outer one" (ibid., 336–338).

```
def scope
  variable = 1
  3.times do
    variable += 1
  end
  variable
end
scope #=> 4
defined? variable #=> nil
```

However, this picture is complicated by blocks. A block both inherits the scope of its parent and defines its own scope: it maintains the lifetime of local variables initialized outside of it without causing a hole in their scope.

Therefore, if a block references, or assigns to, a local variable visible in its parent scope, it refers to the parent's variable. Otherwise, a variable initialized inside a block is not visible outside of it. If it is desired that variables defined within a block do not clash with those defined outside, block local variables are appropriate.

```
v = :out
1.times do
  v, w = :in, :in
  p [v, w]
end #=> [:in, :in]
v #=> :in
w #=> NameError
```

Reflection

The `Kernel.local_variables` method returns an `Array` of `Symbols`, each of which names a local variable defined in the current scope, in reverse chronological order.

Instance Variables

Instance variables are named by an identifier with a commercial at (`U+0040`) character as the sigil: `@`*`identifier`*. Conventionally, the name consists of lowercase words separated by low lines.

An instance variable is created by assigning it a value. It is not inherited from a superclass, so exists for a given object only if an instance method of that object has assigned it a value. An uninitialized instance variable has the value `nil`, but its use results in a warning.

Scope

An instance variable defined in the body of an instance method is accessible by every instance of that method's receiver. Its scope is a specific object. By corollary, an instance variable defined outside of an instance

method, such as in a class body, is not accessible from instance methods: it is, even if named identically, entirely separate.

The above describes the typical use for instance variables, namely storing an object's state, but they may also be defined inside a class or module body, outside of any method, in which case their scope is delimited by the class/module in which they were defined. The former are class instance variables; the latter module instance variables.

Reflection

The names of an object's instance variables are returned as an `Array` of `Symbols` by `Kernel#instance_variables` in the order they were assigned.

Their values may be retrieved and modified from another object with `Kernel#instance_variable_get(`*`ivar`*`)` and `Kernel#instance_variable_set(`*`ivar`*`, `*`value`*`)`, respectively, where *ivar* is the variable's name as a `Symbol`, including the `@` prefix. An instance variable may be removed with the private method`Kernel#remove_instance_variable(`*`ivar`*`)`. These methods should be used sparingly as they break encapsulation.

Class Variables

A class variable is named by an identifier whose sigil is two consecutive commercial at (`U+0040`) characters: `@@`*`identifier`*. Conventionally, the name consists of lowercase words separated by low lines.

Class variables must be initialized before use. Referencing an uninitialized class variable results in a `NameError`.

Scope

A class variable's scope is the body of the enclosing class or module. It is shared between all instances of the class and accessible from both the class body and its methods, but is nevertheless encapsulated: concealed, by default, from the class's users.

Unlike instance variables, class variables are inherited by child classes: if a class modifies a class variable defined in its superclass, the change is visible in both locations.

```
class Parent
  @@cvar = :parent
  def self.cvar
    @@cvar
  end
end
Child = Class.new Parent
Parent.cvar #=> :parent
Child.cvar  #=> :parent
```

A class variable defined at the top-level of a program is inherited by all classes. It behaves like a global variable.

```
@@cvar = :top
class Top
  def self.cvar
    @@cvar
  end
end
Top.cvar #=> :top
class Top
  @@cvar = :class
end
Top.cvar #=> :class
@@cvar   #=> :class
```

Ruby issues a warning—e.g., `class variable @@cvar of Top is overtaken by Object`—when a class variable is first defined in a child class then subsequently assigned to in a parent class, as shown below:

```
class Top
  @@cvar = :class
  def self.cvar
    @@cvar
  end
end
@@cvar = :top
```

```
Top.cvar #=> :top
# warning: class variable @@cvar
# of Top is overtaken by Object
```

However, a class variable is not shared between sibling classes, unless defined in their common parent, or those without a common parent.

```
class Parent
  @@parent = :strict
  def self.parent
    @@parent
  end
end
class Daughter < Parent
  @@parent = :unfair!
  @@me = :angelic
  def self.me; @@me end
end
class Son < Parent
  @@parent = :biased
  @@me = :unappreciated
  def self.me; @@me end
end
Parent.parent   #=> :biased
Daughter.parent #=> :biased
Son.parent      #=> :biased
Daughter.me     #=> :angelic
Son.me          #=> :unappreciated
```

These semantics have led Thomas et al. [Thom09, pp. 337–338] to explicitly advise against the use of class variables and Perrotta [Per10, pp. 129–129] to regard them as having "…a nasty habit of surprising you". An alternative mechanism for storing class state is class instance variables, which are not afflicted with either of the above problems, but cannot be referenced from instance methods.

Reflection

The names of a class's class variables are returned as an `Array` of `Symbols` by `Module#class_variables` in the order they were assigned. Their values

may be retrieved and modified from outside this class with `Module#class_variable_get(`*`cvar`*`)` and `Module#class_variable_set(`*`cvar`*`, `*`value`*`)`, respectively, where *cvar* is the variable's name as a `Symbol`, including the `@@` prefix. A class variable may be removed from a class with `Module#remove_class_variable(`*`cvar`*`)`. These methods should be used sparingly as they break encapsulation.

Global Variables

A global variable is named with a dollar sign (`U+0024`) sigil: `$`*`identifier`*. Conventionally, the name consists of lowercase words separated by low lines.

An uninitialised global variable has the value `nil`, although attempting to use such a variable results in a warning. The predefined global variables summarised in Predefined Global Variables are initialised automatically.

A global variable should not be defined with the same name as a predefined global variable, and indeed cannot if that variable is read-only.

Scope

A global variable is accessible in every scope. Once set it refers to the same object wherever it is referenced.

```
$omnipresent #=> nil
class Diety
  $omnipresent = :yes
  def preternatural?
    $omnipresent == :yes
  end
end
$omnipresent #=> :yes
Diety.new.preternatural? #=> true
```

Reflection

An `Array` of global variable names as `Symbol`s can be obtained with `Kernel.global_variables`

Tracing

`Kernel.trace_var(`*`global`*`)` accepts a `Symbol` naming a global variable and a block. Every time the named variable is assigned to, the block is called with the new value.

defined?

`defined?` is a unary operator which tests whether its operand is defined, and if so returns a description of it. `nil` is returned if the operand is an undefined variable or method, an expression which uses `yield` without an associated block, or an expression which uses `super` without a corresponding ancestor method. In all cases the test is conducted without evaluating the operand. Note that although a constant argument does not cause `:const_missing` to be called, when the argument is a message expression, `#respond_to_missing?` will be used to determine method existence.

```
defined? $a #=> nil
defined? 3 + 2 #=> "method"
var = defined? true or nil #=> "true"
defined? var #=> "local-variable"
defined? 1.times { :one } #=> "expression"
defined? yield #=> nil
```

Operand	Return Value of `defined?`
Defined local variable	`"local-variable"`
Defined global variable	`"global-variable"`
Defined constant	`"constant"`
Defined instance variable	`"instance-variable"`

The return values of the `defined?` operator.

Operand	Return Value of `defined?`
Defined class variable	`"class variable"`
`nil`	`"nil"`
`true`	`"true"`
`false`	`"false"`
`self`	`"self"`
Expression using `yield` correctly	`"yield"`
Expression using `super` correctly	`"super"`
Assignment expression	`"assignment"`
Message sending expression (doesn't check arity)	`"method"`
Any other legal expression	`"expression"`
Undefined variable or invalid use of `yield`/`super`	`nil`

Assignment

An assignment expression sets the value of one or more lvalues to their corresponding rvalues. Its general form is *`lvalues = rvalues`*: one or more lvalues, an equals sign (`U+003D`, then one or more rvalues.

Lvalues

An *lvalue* is a target of an assignment: an expression that can appear on the *l*eft-hand side of an assignment expression.

Variables

The name of any variable is a valid lvalue. An assignment expression involving a variable lvalue causes the variable to take the value of the corresponding rvalue. This operation occurs without any methods being invoked or messages sent: it is inbuilt and its semantics cannot be overridden.

Constants

The name of any constant is a valid lvalue. Assignment to a constant has the same semantics as assignment to a variable, with two caveats:

- If the constant is initialized prior to the assignment, a warning will be issued.
- Constant assignment is illegal in the body of a method.

Attributes

A message expression of the form *`receiver.selector`* is a valid lvalue unless the last character of *selector* is a question mark (`U+003F`) or exclamation mark (`U+0021`). If the last character of *selector* is not an equals sign (`U+003D`), it is set to be. Then, *receiver* is sent *selector* with the rvalue as its argument.

An important implication of the above is that the receiver defines the semantics of assignment. Typically, a selector ending with = will assign its argument to the corresponding instance variable, but it is free to do otherwise. However, the return value of an attribute assignment expression is always the rvalue; the value returned by the receiver is ignored.

Element Reference Lvalues

An element reference expression, i.e. *`receiver`*`[`*`expression`*`]` is a special case of attribute lvalues. It is equivalent to *`receiver`*`.[]=(`*`expression`*`,` *`rvalue`*`)`.

Rvalues

An *rvalue* is a value being assigned: appearing on the *r*ight-hand side of an assignment expression. Any expression is a valid rvalue. The value of the expression that assigned to the corresponding lvalue.

Simple Assignment

A simple assignment expression consists of a single lvalue and a single rvalue: *`lvalue = rvalue`*. It sets the value of *lvalue* to *rvalue*, then returns *rvalue*.

If the rvalue is an `Array`, the lvalue may be followed by a trailing comma to set the lvalue to the first element of the rvalue, discarding the remaining rvalues.

Abbreviated Assignment

An *abbreviated assignment* expression is a syntactical shortcut for an rvalue consisting of a binary operator whose operands are the lvalue and rvalue, respectively. It takes the following general form, where *operator* is one of thirteen predefined selectors enumerated in the table below: *`lvalue operator= rvalue`*. This is wholly equivalent to: *`lvalue = lvalue operator rvalue`*.

This equivalence means that the abbreviated form results in the lvalue being sent a message with the selector *operator* and the argument *rvalue*, then the result of this operation being assigned to *lvalue*.

The value returned by an abbreviated assignment expression is that of its expanded right-hand side.

The `||=` operator gives rise to a popular idiom Perrotta terms a “Nil Guard” [Per10, pp. 243–244] . Its purpose is to assign the rvalue to the lvalue iff the lvalue is false (`false` or `nil`). That is, to initialize the lvalue only if it isn’t already. Perrotta describes an instance variable employing this technique as a “Lazy Instance Variable” [Per10, pp. 243–244] .

```
var ||= Time.now
var #=> 2010-02-19 10:51:07 +0000
var ||= Time.now
var #=> 2010-02-19 10:51:07 +0000
```

Operator	Abbreviation	Expansion
+	*lvalue* += *rvalue*	*lvalue* = *lvalue* + *rvalue*
-	*lvalue* -= *rvalue*	*lvalue* = *lvalue* - *rvalue*
*	*lvalue* *= *rvalue*	*lvalue* = *lvalue* * *rvalue*
/	*lvalue* /= *rvalue*	*lvalue* = *lvalue* / *rvalue*
%	*lvalue* %= *rvalue*	*lvalue* = *lvalue* % *rvalue*
**	*lvalue* **= *rvalue*	*lvalue* = *lvalue* ** *rvalue*
&&	*lvalue* &&= *rvalue*	*lvalue* && *lvalue* = *rvalue*
||	*lvalue* ||= *rvalue*	*lvalue* || *lvalue* = *rvalue*
&	*lvalue* &= *rvalue*	*lvalue* = *lvalue* & *rvalue*
|	*lvalue* |= *rvalue*	*lvalue* = *lvalue* | *rvalue*
^	*lvalue* ^= *rvalue*	*lvalue* = *lvalue* ^ *rvalue*
<<	*lvalue* <<= *rvalue*	*lvalue* = *lvalue* << *rvalue*
>>	*lvalue* >>= *rvalue*	*lvalue* = *lvalue* >> *rvalue*

The abbreviated assignment operators

Parallel Assignment

A parallel assignment expression involves multiple lvalues and/or multiple rvalues. The values are separated by commas.

The parallel aspect of this operation is that all of the rvalues are evaluated, left to right, prior to assigning them. This allows the value of two or more variables to be swapped, as shown in the figure below.

Equal Number of Lvalues to Rvalues

When there are as many lvalues as there are rvalues, each lvalue is assigned the rvalue in the corresponding position on the right-hand side of the expression. That is, the n^{th} lvalue is assigned the n^{th} rvalue.

```
a, b, c = 1, 2, 3
a #=> 1
b #=> 2
c #=> 3

a, b = b, a
```

```
a #=> 2
a #=> 1
```

Splat Operator

Lvalues and rvalues may optionally be directly preceded by an asterisk (U+002A), termed a *splat*[2] hereafter.

Splatting an Lvalue

A maximum of one lvalue may be splatted in which case it is assigned an `Array` consisting of the remaining rvalues that lack corresponding lvalues. If the rightmost lvalue is splatted then it consumes all rvalues which have not already been paired with lvalues. If a splatted lvalue is followed by other lvalues, it consumes as many rvalues as possible while still allowing the following lvalues to receive their rvalues.

```
*a = 1
a #=> [1]

a, *b = 1, 2, 3, 4
a #=> 1
b #=> [2, 3, 4]

a, *b, c = 1, 2, 3, 4
a #=> 1
b #=> [2, 3]
c #=> 4
```

Empty Splat

An lvalue may consist of a sole asterisk (U+002A) without any associated identifier. It behaves as described above, but instead of assigning the corresponding rvalues to the splatted lvalue, it discards them.

---

2. "This may derive from the 'squashed bug' appearence of the asterisk on many early laser printers." [Raymond99, pp. 422–422]

```
a, *, b = *(1..5)
a #=> 1
b #=> 5
```

Splatting an Rvalue

When an rvalue is splatted it is converted to an `Array` with `Kernel.Array()`, the elements of which become rvalues in their own right.

```
a, b = *1
a #=> 1
b #=> nil

a, b = *[1, 2]
a #=> 1
b #=> 2

a, b, c = *(1..2), 3
a #=> 1
b #=> 2
c #=> 3
```

When given an object that does not respond to `:to_a`, `Kernel.Array()` returns an `Array` with that object as its sole element. Therefore, splatting such an object causes it to expand to itself, effectively a no-op.

However, for objects that do respond to `:to_a` with an `Array`, such as `Array`, `Hash`, `Range`, and `Enumerator`, splatting expands them into a list of their constituent elements.

One Lvalue, Many Rvalues

When multiple rvalues are assigned to a single lvalue, there is an implicit splat operator before the lvalue. Therefore, assigning *n* rvalues to a single lvalue is equivalent to: *lvalue* = [*rvalue*$_0$,…,*rvalue*$_n$]

If this behaviour is undesirable, the lvalue can be followed by a trailing comma, assigning *rvalue*$_0$ to *lvalue*, and discarding the remaining rvalues.

```
a = 1, 2, 3
a #=> [1, 2, 3]
colour, = :red, :green, :blue
colour #=> :red
```

Many Lvalues, One Rvalue

```
a, b, c = [1, 2, 3]
a #=> 1
b #=> 2
c #=> 3
```

If there are multiple lvalues and a single rvalue that responds to `:to_ary`[3] with an `Array`, the elements of this array become rvalues in their own right, replacing the original rvalue.

```
# coding: utf-8
one, two = [:ūnus, :duo, :trēs]
one #=> :ūnus
two #=> :duo

rvalue = Object.new
def rvalue.to_ary
  [:alpha, :beta]
end
a, b = rvalue
a #=> :alpha
b #=> :beta
```

If there are as many lvalues as there are elements in the splatted rvalue, assignment proceeds according to Equal Number of Lvalues to Rvalues; otherwise the following section, Unequal Number of Lvalues to Rvalues, applies.

---

3. Note that `:to_ary` is sent rather than the `:to_a` used by the splat operator. In the former case, the programmer did not directly request that conversion take place so the message for implicit conversion is sent; in the latter, his use of the splat operator constitutes an explicit conversion, so the more liberal protocol is followed.

```
x, y = 47
x #=> 47
y #=> nil
```

Unequal Number of Lvalues to Rvalues

When the lvalues outnumber the rvalues, assignment proceeds as with equal numbers of lvalues to rvalues, with the remaining lvalues being assigned `nil`. That is, for n rvalues and m lvalues, where $n < m$, $lvalue_0$–$lvalue_n$ are assigned $rvalue_0$–$rvalue_n$, and $lvalue_{n+1}$–$lvalue_m$ are assigned `nil`.

```
a, b = 1
a #=> 1
b #=> nil
a, b, c = :a, :b
a #=> :a
b #=> :b
c #=> nil
```

Conversely, when the rvalues outnumber the lvalues, assignment proceeds as with equal numbers of lvalues to rvalues, with the remaining rvalues being discarded.

```
a, b = :a, :b, :c
a #=> :a
b #=> :b
```

Sub-assignment

When a group of at least two lvalues are enclosed in parentheses, they are initially treated as a single lvalue in that, collectively, they are assigned a single rvalue. After all remaining lvalues have been paired with their corresponding rvalues, the rules of parallel assignment are applied again to each of these groups, recursively for each level of parentheticals.

```
a, (b, c), d = 1, 2, 3, 4
a #=> 1
b #=> 2
c #=> nil
```

```
d #=> 3

(a, b), c = [1, 2], 3, 4
a #=> 1
b #=> 2
c #=> 3

zero, (one, *rest) = 0, [*1..5, *6..10]
zero #=> 0
one #=> 1
rest #=> [2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Recall that if multiple lvalues are assigned a single rvalue that responds to `:to_ary`, the rvalue is assigned to the first lvalue, and the remaining lvalues are assigned `nil`. Therefore, sub-assignment is most useful when the corresponding rvalue is array-like, because it distributes the elements of potentially nested arrays on the right-hand side among lvalues.

Value of a Parallel Assignment Expression

A parallel assignment expression involving a single rvalue has that rvalue as its value. Otherwise, the value is an `Array` of the rvalues, including any that were discarded. In both cases, the rvalues are splatted as appropriate before being returned.

```
a, b, c = 1.5 #=> 1.5
(a, b), c = 1, *[2, 3] #=> [1, 2, 3]
```

MESSAGES

> …Smalltalk is not only NOT its syntax or the class library, it is not even about classes. I'm sorry that I long ago coined the term "objects" for this topic because it gets many people to focus on the lesser idea.
>
> The big idea is "messaging" -- that is what the kernal[*sic*] of Smalltalk/Squeak is all about (and it's something that was never quite completed in our Xerox PARC phase). The Japanese have a small word—*ma*—for "that which is in between"—perhaps the nearest English equivalent is "interstitial". The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be.
>
> —Kay98

To request that an object perform an operation we send it a *message*[1]. Imagine an envelope: on the front is the name of the object, inside is a sheet of paper naming the task to be performed and listing any additional information needed to perform it. The object is the message's *receiver*, the task name its *selector*, and the list its *arguments* [Budd87, pp. 16–16] . The selector is assumed[2] is to be the name of a method defined[3] upon the

---

1. Liu suggests envisaging messages as telegrams:

   > "Message" is too abstract a word. A better word is *telegram*. A "telegram" is tangible: I can touch it, I can see the information it carries, and I can picture the moment it arrives at the door of its addressee. It is not some vague electronic-sounding thing like a "message". Therefore, I encourage you to think of an old-fashioned telegram whenever you see the term "message".
   >
   > —Liu99, pp. 2–2

2. Due to the Method Missing protocol, the sending of the message may succeed even if this assumption is false.

receiver, so it is common[4] to describe sending a message as *invoking a method*. Accordingly, we shall use the two phrases interchangeably.

Message Expression Syntax

A message is sent with an expression, the simplest of which consists solely of the selector, e.g. `foo`. In this case, the receiver is *implicit*, so assumed to be `self`.

```
def hop
  :boing!
end
hop #=> :boing!
hop = :skip
hop #=> :skip
jump = :leap
jump #=> :leap
```

A message that invokes a private method must always be sent to the implicit receiver. By contrast, in the following scenarios the receiver must be specified explicitly:

1. The selector is also a keyword such as `class`. (An implicit receiver would create syntactical ambiguity).
2. The selector is that of a non-unary operator method.
3. The selector is used as an lvalue.

---

3. A message selector may consist of any `Symbol`, even those which are illegal when defining a method with the `def` keyword. Methods with corresponding names can be defined with `Object#define_method`, or such messages can be handled by the Method Missing protocol.
4. For instance, Flanagan & Matsumoto make the popular claim that "…methods are called "messages"" [Flan08, pp. 178–178] . However, in the definitions of Budd [Budd87, pp. 6–6] , Mitchell [Mitchell04, pp. 279–279] , and Klass & Schrefl [Klas95, pp. 13–13] , *message* refers to the *request* being made of an object; whereas Flanagan & Matsumoto use it to refer the means with which the receiver would respond to such a request. On this point we grudgingly conceede to Ruby's convention.

4. The message name is identical to a local variable. (This case can also be disambiguated by immediately following the selector with a pair of parentheses, i.e. *`selector`*`()`).

```
class Schneier
  private
  def blowfish
    "Teach a man to fish..."
  end
end
Schneier.new.blowfish #=> NoMethodError

class Schneier
  def backdoor
    self.blowfish
  end
end
Schneier.new.backdoor #=> NoMethodError

class Schneier
  def backdoor
    blowfish
  end
end
Schneier.new.backdoor #=> "Teach a man to fish..."
```

The receiver is made explicit by concatenating its name and the selector with a full stop, i.e. *`receiver`*`.`*`selector`*. *receiver* is an arbitrary expression, the value of which receives the message.

```
"esrever".reverse #=> "reverse"
(1 + 2).succ #=> 4
```

The value of a message expression is that of the method it invoked. If no corresponding method could be found, a `NoMethodError` exception is raised.

Arguments

A message may be accompanied by one or more *arguments*: values to be used in the resulting computations. They are supplied as a parenthesized list

of expressions separated by commas: *receiver*.*selector*(*arg*0,…,*arg*n). The number of arguments sent must agree with how the corresponding method was defined, otherwise an ArgumentError is raised.

```
cabin_dwellers = []
cabin_dwellers.push(:kaczynski, :roosevelt, :thoreau)
#=> [:kaczynski, :roosevelt, :thoreau]
cabin_dwellers.unshift(cabin_dwellers.pop())
#=> [:thoreau, :kaczynski, :roosevelt]
cabin_dwellers.insert(1, cabin_dwellers.pop)
#=> [:thoreau, :roosevelt, :kaczynski]
```

Enumerable Arguments

It is often the case that a message needs to be sent with multiple arguments, yet those arguments are stored in an Enumerable such as Array. If the message is sent with just the Enumerable argument it will receive a single argument (a reference to the Enumerable) rather than the constituent elements thereof. The solution is to prefix the Enumerable argument with an asterisk (U+002A), which in this context is termed a *splat operator*, thus expanding the Enumerable into its individual elements.

```
def one(argument)
  "One argument: #{argument}"
end

def three(first, second, third)
  "Three arguments: #{first}, #{second}, #{third}"
end

array = [1, 2, 3]
one(array) #=> "One argument: [1, 2, 3]"
three(array) #=> ArgumentError: wrong number of arguments (1 for 3)
three(*array) #=> "Three arguments: 1, 2, 3"
three(1, 2, 3) #=> "Three arguments: 1, 2, 3"
```

This technique works by sending :to_a to the prefixed object, so any object that responds to this message with an Array can be used in the same fashion.

Block Literals

A message expression followed by a block literal causes the block to be sent along with the message. A block literal may be enclosed in curly braces (`{...}`) or a `do...end` construct. These forms are semantically equivalent , however it is conventional to use the former for short blocks that fit on a single line, and the latter for multi-line blocks.

```
3.times {|number| print number ** 2, ' ' } #=> 3
# 0, 1, 4,

(:a..:z).select do |letter|
  letter > :r && letter < :y
end #=> [:s, :t, :u, :v, :w, :x]
```

Proc Arguments

Analogous to the situation described in Enumerable Arguments, is sending a `Proc` to a method expecting a block literal. The method cannot be fooled by simply including the `Proc` in the argument list: it sees an extra argument it wasn't expecting, and no block argument. However, if exactly one `Proc` argument is prefixed with an ampersand, and appears as the final argument in the list, it is automatically converted into an anonymous block. The method behaves as it should, unaware of your chicanery. This technique works by sending `#to_proc` to the object, and using the `Proc` returned as the block. By extension, any object that responds to `:to_proc` in this manner can be used in the same fashion.

```
title = ->(name) { %w{Mr Mrs Sir Dr}.sample + ' ' + name }
title.is_a?(Proc) #=> true
['Stephen Hawking', 'R. Feynman', 'Niels Bohr'].map(&title)
#=> ["Dr Stephen Hawking", "Mrs R. Feynman", "Sir Niels Bohr"]
```

Parentheses

Parentheses may be omitted from message expressions when doing so does not introduce syntactic ambiguity. They are rarely used when no arguments are involved, e.g. `'briefcase'.upcase` is equivalent to

`'briefcase'.upcase()`, with the former style being recommended. However, they serve to disambiguate a local variable reference from a message expression with an implicit receiver, so are sometimes necessary.

```
def ambig
  :uous
end
ambig #=> :uous
ambig() #=> :uous
ambig = :uity
ambig #=> :uity
ambig() #=> :uous
```

Parentheses may usually be omitted even when arguments are involved. If so, there must be whitespace between the selector name and the first argument.

```
"spending".sub(/spend/, 'sav') #=> "saving"
"spending".sub /spend/, 'sav'  #=> "saving"
";".gsub(/(.)/, '.' => 'full stop', ',' => 'comma',
                ';' => 'semicolon') #=> "semicolon"
";".gsub /(.)/, '.' => 'full stop', ',' => 'comma',
                ';' => 'semicolon' #=> "semicolon"
```

A common case when parentheses are *required* to disambiguate is nested message expressions where multiple arguments are involved, such as *abc*, *d*, where *a* and *b* are selectors. To which do the arguments, *c* and *d*, belong? Does the programmer mean *a*(*b*(*c*, *d*)) or *a*(*b*(*c*), *d*)? Ruby assumes the former. However, even if the programmer agrees with Ruby's interpretation, these forms of expression are seldom as clear to others. Therefore, it is recommended to employ parentheses in such cases, even if syntactically unnecessary, as an aid to legibility.

```
cubes = [1.0]
cubes.push cubes.first * 8, Math.log2 134217728, 64.0
#=> SyntaxError: syntax error, unexpected tINTEGER, expecting $end
#   cubes.push cubes.first * 8, Math.log2 134217728, 64.0
#                                                    ^
cubes.push cubes.first * 8, Math.log2(134217728), 64.0
#=> [1.0, 8.0, 27.0, 64.0]
```

When parentheses are used to enclose a message's arguments, the opening parenthesis must immediately follow the selector like so: *selector*(*arg*$_0$,…, *arg*$_n$). Whitespace between *selector* and `(` will result in a `SyntaxError`. This is due to a quirk in Ruby's syntax inasmuch as parentheses serve two distinct functions—grouping of expressions and associating an argument list with a message expression—so the above form is ambiguous.

Chaining

A message expression returns an object, which may in turn receive messages, i.e. *receiver*.*selector*$_0$.*selector*$_1$. *receiver* is sent *selector*$_0$ which returns an object; that object is sent *selector*$_1$. Message expressions can be arbitrarily *chained* in this fashion.

```
(100..110).map{|n| n / 5 }
#=> [20, 20, 20, 20, 20, 21, 21, 21, 21, 21, 22]
(100..110).map{|n| n / 5 }.uniq #=> [20, 21, 22]
(100..110).map{|n| n / 5 }.uniq.first(2) #=> [20, 21]
(100..110).map{|n| n / 5 }.uniq.first(2).reduce(:+) #=> 41
```

This technique relies on the composite receiver consistently returning an object responding to the selector; otherwise a `NoMethodError` will break the chain. It is unsuitable when the receiver may return a materially different object for certain arguments. For instance, `Array#select` always returns an `Array` even when it doesn't select any elements, allowing it to receive any `Array` selector. Conversely, methods such as `Array#[]`, which return either the specified element or `nil` if it does not exist, make awkward links.

```
" All propositions are of equal value. ".strip!.sub(/\.$/,'!')
#=> "All propositions are of equal value!"
"All propositions are of equal value.".strip!.sub(/\.$/,'!')
#=> NoMethodError: undefined method `sub' for nil:NilClass
```

Messages sent purely for their side effects should, and often do, return `self` so as to receive any message understood by their receiver.

```
" All propositions are of equal value. ".strip.sub(/\.$/,'!')
#=> "All propositions are of equal value!"
```

```
"All propositions are of equal value.".strip.sub(/\.$/,'!')
#=> "All propositions are of equal value!"
```

Dynamic Sending with Object#send

A message can also be sent to an object with `Object#send`, i.e. *`receiver`*`.send(`*`selector`*`, `*`arg`*$_0$ `,…,`*`arg`*$_n$ `)`. This allows the message selector to be determined dynamically, at runtime, as opposed to the syntax described previously which requires the selector to appear literally in the source file.

```
name = :size
"wool".name
#=> NoMethodError: undefined method `name' for "wool":String
"wool".send(name) #=> 4
33.send(:/, 3)    #=> 11
```

`Object#send` is aliased to `__send__`, so even if the former is overidden, the alias can be used in its place. A warning is issued if the user overrides `__send__`. `Object#public_send` performs the same function as `Object#send`, but raises a `NoMethodError` if the method is private or protected.

Operators

The following operators are implemented in terms of messages: `!`, `~`, unary `+`, `**`, unary `-`, `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `&`, `|`, `^`, `<`, `<=`, `>=`, `>`, `==`, `===`, `!=`, `=~`, `!~`, `<=>`. They are termed *operator methods* because their usual semantics can be overidden by defining a method with the corresponding name. In general, this name is identical to that of the operator, however unary plus and unary minus are named `+@` and `-@`, respectively. The syntax for invoking an operator method differs from the general rules set out above in the following ways:

1. Their receiver must always be explicit, i.e. an expression of the form *`operator`* or *`operator argument`* is illegal.
2. Unary operator methods may be invoked as *`operator`*`.`*`receiver`*
3. Binary operator methods may be invoked as *`receiver operator argument`* as long as they weren't defined to require multiple arguments.

4. An expression of the form *receiver*`.`*operator*`(`*arg*$_0$`,…,`*arg*$_n$`)`, where *arg* is an argument, is always legal. However, when *operator* is unary plus or unary minus, it must be given with an `@` suffix, e.g. `+@` or `-@`.

Further, in forms 2 and 3, the expression has the associativity, arity, and precedence outlined in Operators.

Conventions

Around certain selectors there exist conventions which are established by the core classes and standard library. Their adoption engenders APIs that are more coherent and feel familiar to Rubyists. They are, however, merely suggestions—explanations of traditions; not a list of requirements. Ignore as you see fit.

`#call`
: Accepts a variable-length argument list with which it invokes the receiver.

`#each`
: Yields the next element of a sequence when given a block; otherwise returns an enumerator.

`#each_`***attribute***
: Enumerates a collection by *attribute*. Yields the next element when given a block; otherwise returns an enumerator.

`#empty?`
: Returns `true` if the receiver doesn't have any content; `false` otherwise.

`#size`
`#length`
: Returns the receiver's magnitude as an `Integer`.

`_by`
: Selectors with a `_by` suffix typically imply that the method expects a block, the results of which constrain the computation.

`#to_`
: Converts the receiver into an object of the corresponding core type.

`try_convert`
: Class method that implicitly converts the argument to an instance of the receiver.

`#<=>`
: Returns `0` if the operands are equal, `1` if the first is greater than the second, `-1` if the first is less than the second, and `nil` if they are incomparable.

`#<`
: Returns `true` if the receiver is less than argument; `false` otherwise.

`#>`
: Returns `true` if the receiver is greater than argument; `false` otherwise.

`#==`
: Returns `true` if the receiver is equal to the argument; `false` otherwise.

`#[]`
: Indexes the receiver by the "key" supplied as the argument(s), returning the requested slice or `nil`/`[]` if no corresponding data was found.

`#[]=`
: The assignment counterpart to `#[]` the first argument(s) describe the slice, and the final argument its new value.

`#=~`
: Matches the receiver with the argument.

`#<<`
: For non-`Integer` receivers, appends the argument to the receiver then returns the mutated receiver.

`#+`
: Returns a new object comprising the concatenation of the operands.

`#rewind`
: Resets the receiver to its initial state.

`#*`
: Multiples the receiver by the numeric argument.

Tone

By convention, a selector with a question mark (?) suffix denotes that the message poses a polar question to the receiver. Such messages are referred to as *predicates*.

```
33.odd? #=> true
('a'..'z').include? 'd' #=> true
File.exists?(File.expand_path '~/.emacs') #=> false
:roland_barthes.is_a? Symbol #=> true
```

Etymology

This convention is shared with Scheme and probably derived from that of Lisp to name predicate functions with a `p` suffix, e.g. Lisp's `(evenp)` ⟹ Scheme's `(even?)` ⟹ Ruby's `Fixnum#even?`.

```
3.5.infinite? #=> nil

3.5.fdiv(0.0) #=> Infinity
3.5.fdiv(0.0).infinite? #=> 1

-10.quo(0.0) #=> -Infinity
-10.quo(0.0).infinite? #=> -1
```

Following similar convention, a selector with an exclamation mark (!) suffix (colloquially called a *bang*) is used "…to mark a method as special. It doesn't necessarily mean that it will be destructive or dangerous, but it means that it will require more attention than its alternative." [Brown09] . "Usually, the method without the exclamation mark returns a modified copy of the object it is invoked on, and the one with the exclamation mark is a mutator method that alters the object in place." [Flan08]

```
dungeon = 'oubliette'
dungeon.capitalize #=> "Oubliette"
dungeon #=> "oubliette"
dungeon.capitalize! #=> "Oubliette"
dungeon #=> "Oubliette"
```

Both suffixes are merely *conventional*, however; they do not guarantee that the receiver will respond in the aforementioned manner.

Plurality

Messages whose selectors are plural generally return an `Enumerable` collection of objects. For example, `String#bytes`, `String#chars`, and `String#codepoints` return `Enumerators` which yield each byte, character, or codepoint, respectively, in turn. Similarly, `Object#methods`, `Object#instance_methods`, `Object#instance_variables`, and similar methods, return `Arrays` of `Symbols`.

```
[*'Ruby'.bytes] #=> [82, 117, 98, 121]
```

Responding to Messages

An object is said to *respond to* a given selector if it expects to receive it as a message. The `Kernel#respond_to?(`*`selector`*`)` predicate returns `true` if a method is defined with the same name as the selector; `false` otherwise. In some situations, `Module#method_defined?` is an alternative, for the reasons outlined in its description.

```
Math::PI #=> 3.141592653589793
Math::PI.respond_to? :floor #=> true
Math::PI.floor #=> 3
Math::PI.methods.all?{|m| Math::PI.respond_to? m} #=> true
Math::PI.respond_to? :door #=> false
Math::PI.door #=> NoMethodError
```

However, by defining `method_missing` an object may respond to a message without having a corresponding method defined, making the behaviour of `Kernel#respond_to?` insufficient. The solution is `Kernel#respond_to_missing?`.

```
object = Object.new
def object.method_missing(selector)
  selector == :colour ? :red : super
end
object.respond_to? :colour #=> false
object.colour #=> :red
```

`Kernel#respond_to?` ignores private methods by default. If it is sent with a second argument of `true`, private methods are included in its search.

```
'uninitialized'.private_methods.include? :initialize #=> true
'uninitialized'.respond_to? :initialize #=> false
'uninitialized'.respond_to? :initialize, true #=> true
```

`Kernel#respond_to?` is often used to condition a message expression on the receiver understanding it. That is, before an object is told to perform an operation it is asked whether it is able. Normally this procedure is unnecessary because it is known, *a priori*, what messages the object will respond to; however, especially when employing metaprogramming techniques, it is sometimes necessary.

```
receiver.selector if receiver.respond_to?(:selector)
```

The following example suggests a simplification of this idiom by defining `Object#send?` which returns `nil` if the named method does not exist. In addition to the brevity it affords, this approach avoids the aforementioned weakness of `Kernel#respond_to?` with respect to messages handled by Missing Methods, by rescuing `NoMethodError` if it occurs.

```
class Object
  def send?(selector, *args, &block)
    begin
      send(selector, *args, &block)
    rescue NoMethodError
    end
  end
end
```

A closely related related scenario involves wanting to send a message unless the receiver is `nil`. It transpires when working with values which

shouldn't be `nil`, but might be. The general solution is one of the following constructs:

```
receiver ? receiver.selector : nil
```

```
receiver & & receiver.selector
```

Ruby on Rails defines for this purpose `Object#try(`*selector*`, `*arg*$_0$`,…,`*arg*$_n$` )`, which functions as `Object#send`, but returns `nil` when the receiver is `nil`, rather than raising a `NoMethodError`.

```
class Object
  alias_method :try, :__send__
end

class NilClass
  def try(*args)
    nil
  end
end
```

OBJECTS

An *object* is a compilation of data (attributes) and behaviour (methods) which encapsulate a specific *instance* of a class. The `String"hello"` and the `Integer 3` are both examples of objects, and instances of the `String` and `Integer` classes, respectively. They constitute data (*hello* and *3*, respectively) and behaviour relevant to that data.

Instantiation

There are five main ways to *instantiate*, or *create*, an object:

- Using a literal.
- Sending a constructor message to an existing object (usually a class).
- Cloning or duplicating an existing object.
- Loading a serialized, or Marshaled, object.

In addition, certain core objects always exist without being instantiated: they are created by the Ruby interpreter.

Constructors

.new

Sending the `:new` message to a class instantiates that class. For example, `Array.new` creates an `Array` object. A method that instantiates a class is a *constructor*: it allocates an object then initializes its state.

Allocation

`Class#allocate` allocates memory for a new object and returns a reference to it. It cannot be overridden. If invoked manually, it returns an uninitialized instance of the class.

Initialization

The newly allocated object is sent an `:initialize` message along with the arguments passed to `.new`. The `#initialize` method typically validates the constructor's arguments then assigns them to instance variables. It is a private method so cannot be called from outside the class.

The `.new` constructor ignores the return value of `#initialize`, so as to return the initialized object instead.

It is good practice for a class—particularly one with a superclass other than `Object`—that defines an `#initialize` method to call its parent's `#initialize` with super. This allows the superclass and any included modules to perform their own initialization routines.

Identity

Every object has a numeric identifier that is unique among other objects and constant for the object's lifetime. It is returned as a `Fixnum` by `Object#object_id`.

`ObjectSpace._id2ref` returns a reference to an object given its ID. For example, `ObjectSpace._id2ref([1, 2].object_id) = [1, 2]`.

An object's ID should not be confused with its hash code as returned by `Object#hash`. Logically identical objects should have the same hash code, yet will have different object IDs if they are pointed to by different references. For example, `[].object_id != [].object_id` yet `[].hash == [].hash`.

Class

The class of an object is returned by `Object#class`. To test whether an object is an instance of a given class, use the `Object#is_a?(`*`class`*`)` predicate, where *class* is a `Class` object.

Methods

The names of methods defined for an object are returned by `Module#methods` as an `Array` of `Symbol`s. For an object that is neither a class nor a module, the methods returned are the intersection of its instance and singleton methods; otherwise, they are its singleton methods only. To view only the receiver's singleton methods, use `Module#singleton_methods`. `Module#public_methods`, `Module#private_methods`, and `Module#protected_methods` only return the names of methods with the corresponding visibility. If a second argument of `false` is given to any of these methods, names of inherited methods are omitted from the `Array`.

```
Object.new.methods.grep /^[[:alpha:]]+\?/
#=> [:nil?, :eql?, :tainted?, :untrusted?, :frozen?, :equal?]
[].methods.grep /!/
#=> [:reverse!, :rotate!, :sort!, :sort_by!, :collect!, :map!,
#    :select!, :reject!, :slice!, :uniq!, :compact!, :flatten!,
#    :shuffle!, :!~, :!, :!=]
String.methods false #=> [:try_convert]
private_methods.grep /[[:upper:]]/
#=> [:Integer, :Float, :String, :Array, :Rational, :Complex]
Object.new.tap{|o| o.define_singleton_method(:s, ->{})}.methods(false)
#=> [:s]
```

Relations

Order

If instances of a class suggest an ordering relationship such that one instance is either less than, greater than, or equal to, another, they are said to be *comparable*. The class can define a method named `<=>` (the *spaceship operator*) expecting a single argument and following the `qsort(3)` convention of returning `-1` if `self` is less than the argument, `0` if they are equal, or `1` if `self` is greater.

The class then mixes in the `Comparable` module, which provides `#<`, `#==`, and `#>` methods implemented in terms of `#<=>`.

Equivalence

The means by which objects are compared for equality depend on the type of equality desired.

`Object#equal?` considers the receiver equal to the argument if the two objects are identical, i.e. their object IDs are equal. For example, `[].equal?([]) == false`. Classes are discouraged from overriding this method, so its semantics should not change.

`Object#==` is an alias of `Object#equal?`, but *is* normally overridden in subclasses to denote *logical* equivalence. For example, `Array#==` regards two `Array` objects equal if they contain the same number of elements and each element is == to its corresponding element. So, `[O.new] == [O.new]` is `true` if `O.new == O.new`.

`Object#!=` returns the inverse of `Object#==`, so it does not normally need to be defined explicitly. It can be, however.

`Object#eql?` is also an alias of `Object#equal?` that subclasses often override. It differs from `Object#==` in that it denotes *strict* logical comparison without performing type conversion. For example, consider `1` and `1.0`. The two are logically equivalent if converted to the same class, so `1 == 1.0`. However, `1.eql?(1.0) == false` because `Object#eql?` does not perform type conversion.

Flanagan & Matsumoto state that "If two objects are `eql?`, their `hash` methods must also return the same value." [Flan08, pp. 77–78] , going on to recommend that classes implement `#eql?` in terms of `#hash`.

State

Instance Variables

The state of an object is encapsulated in its instance variables, whose values are local to that particular object, hidden from others. An object will not respond to a message selecting one if its instance variables unless a

corresponding method has been defined, either explicitly, or implicitly with `attr`.

An instance variable is typically defined in an instance method. If defined in the context of a class it is a class instance variable.

Attributes

An attribute, *a*, is a pair [Flan08, pp. 94–95] of public methods—an accessor (*a*) and a writer (*a=*)—exposing a property of an object's state to other objects. They can be created automatically by supplying their names to `Module#attr_accessor` or `Module#attr_writer` inside a class definition. This assumes, as is typical, that an attribute corresponds to an instance variable of the same name, i.e. *a* returns *@a*; *a=* sets *@a*. If the attribute value is not backed by an instance variable, it should be exposed in the same way [Meyer00, pp. 55–57] by defining the methods manually.

The writer method behaves specially when used as an lvalue in an assignment expression.

Mutability

`Object#freeze` makes the receiver immutable: attempts to change its state elicit a `RuntimeError`. The `Object#frozen?` predicate returns `true` if the receiver is frozen. There is not a `#thaw` method, so this operation is not reversible. However, duplicating an object removes its frozen state.

Freezing works on objects not variables, so it is permissible to assign a new value to a "frozen variable".

References & Garbage Collection

Variables store references to objects. Assignment is, therefore, the copying of the reference on the right-hand side to the variable named on the left, leaving both sides referring to the same object. However, a reference may not be dereferenced; it is not a pointer.

An object is deemed *unreachable* when there are no references to it, or the sources of the remaining references are themselves unreachable. Such objects are subject to garbage collection: automatic destruction by Ruby. The lifetime of an object is determined by its longest living reference. If an object is intended to be temporal, its references must be likewise.

The garbage collector can be controlled manually via the `GC` module. It is disabled with `GC.disable`, then re-enabled with `GC.enable`—both methods return `true` if the garbage collector was already disabled; `false`, otherwise. The garbage collector normally runs periodically. `GC.start` initiates it immediately, while `GC.count` returns how many times it has run in the current process. For testing extension libraries it may be useful to run the garbage collector every time a new object is allocated. To do so, pass `true` to `GC.stress=`; to revert to normal behaviour, pass `false`. The current status of this flag is returned by `GC.stress`.

An object may register `Procs` that the garbage collector will invoke just prior to destroying the object. These are called *finalizers*, and are registered with `ObjectSpace.define_finalizer(`*obj*`, `*proc*`)`, where *proc* is a `Proc` that should be called when *obj* is about to be garbage collected. If multiple finalizers are attached to a single object, they are invoked in the order they were attached. The `Proc` is passed the object's ID as an `Integer` block parameter, but must not attempt to reference the object being destroyed. The finalizers associated with *obj* can be unregistered with `ObjectSpace.undefine_finalizer(`*obj*`)`.

Listing and Counting

The list of non-immediate objects currently defined is returned as an `Enumerator` by `ObjectSpace.each_object`. If given a `Class` or `Module` argument, it only returns objects with this class, module, or a subclass thereof. When a block is supplied, each object is yielded to it in turn.

`ObjectSpace.count_objects` returns a `Hash` whose keys are names of the interpreter's internal data types, and values the number of existing objects with the corresponding type.

Data Type	Ruby Class
T_ARRAY	Array
T_BIGNUM	Bignum
T_CLASS	Class
T_COMPLEX	Complex
T_DATA	Data
T_FALSE	FalseClass (false)
T_FILE	IO
T_FIXNUM	Fixnum
T_FLOAT	Float
T_HASH	Hash
T_ICLASS	N/A
T_MATCH	N/A
T_NIL	NilClass (nil)
T_NODE	N/A
T_MODULE	Module
T_OBJECT	Any class not specified in this table.
T_RATIONAL	Rational
T_REGEXP	Regexp
T_STRING	String
T_STRUCT	Struct
T_SYMBOL	Symbol
T_TRUE	TrueClass (true)
T_UNDEF	N/A
T_ZOMBIE	N/A

BasicObject

`BasicObject` is the root of the class hierarchy: from this all other classes ultimately inherit. Its superclass is `nil`. It defines the bare minimum of methods so as to be a "blank slate": "...useful as the superclass of delegating wrapper classes..." [Flan08, pp. 235–235] . (A thorough explanation of "blank slate" classes in Ruby is found in *Ruby Best Practices* [Brown09, pp. 57–62] .

Duplication

`Object#dup` creates a shallow copy of the receiver. A new instance of the receiver's class is allocated, tainted if the original object was, then populated with shallow copies of the receiver's instance variables. Neither singleton methods nor frozen state are duplicated[1]. Corresponding instance variables will refer to the same object because they are copied by reference; not referent.

If the duplicated object responds to `:initialize_dup`, it will be sent this message with the receiver as an argument. Its return value is ignored. A common use is to perform a deep copy[2] of instance variables by explicitly duplicating them. If an object may only be duplicated under certain circumstances, `#initialize_dup` may choose to raise an exception.

If an object doesn't respond to `:initialize_dup`, but does respond to `:initialize_copy`, it is sent the latter instead, with the same semantics as `:initialize_dup`.

Objects that shouldn't be duplicated can privatise their `#dup` method or define it to raise an informative exception [Flan08, pp. 243–245] .

Cloning

`Kernel#clone`, behaves like Kernel#dup except it also copies singleton methods and frozen state. Further, instead of sending `:initialize_dup` to its receiver it sends `:initialize_clone`, falling back to `:initialize_copy`, if the former isn't defined.

---

1. To copy these items use Object#clone instead.
2. Another way to perform a deep copy of an object is: `Marshal.load(Marshal.dump(`*`object`*`))`.

Marshaling

An object may be serialized as a binary `String` by supplying it to `Marshal.dump`. If an `IO` object is supplied as the second argument, the marshaled object is written to it. `Marshal.load` reverses this process by recreating the object from its marshaled form, which may be a string or `IO` object.

The Marshal data format is versioned with a major and minor number, which are stored in the first two bytes of marshaled data. `Marshal.load` raises a `TypeError` unless the data has the current major version and a minor version ≤ the current minor version.

Objects may control how they're dumped by responding to `:marshal_dump` with another object to be serialized in their place. If so, they must also respond to `:marshal_load`, which is sent to an allocated, uninitialized instance of their class with the recreated object as an argument. It is expected to initialize the state of the receiver from that of its argument. The return value is ignored.

Taint

Data derived from an external source is potentially unsafe, so should be explicitly validated before use. *Taint checking* is a security mechanism designed to aid this process. Objects derived from `IO` streams, environment variables[3], the command line, and user input are automatically marked as *tainted*. Further, any object ultimately derived, duplicated, or cloned from a tainted object is also tainted: the trait is contagious. The `Object#tainted?` predicate returns `true` if its receiver is tainted; `false`, otherwise. An object may be explicitly tainted with `Object#taint`. Once a tainted object is known to be safe, it can be untainted with `Object#untaint`.

3. The `PATH` environment variable is only tainted if one of its directories are world-writable.

Safe Levels

When Ruby is used in an untrusted environment, such as a CGI script on a public web server, a *safe level* can be set to prevent potentially dangerous methods from being invoked. The safe level is an `Integer` between `0`—no restrictions—and `4`—the most restrictive. It can be set when the interpreter is invoked by supplying a `-T`*`level`* argument, where *level* is the desired safe level; if *level* is omitted, it defaults to `1`. Otherwise, the safe-level is initially `0`, and can be set by assigning the appropriate `Integer` to the thread-local `$SAFE` variable. The value of `$SAFE` can't be lowered.

Level 1

At a safe level of `1` or more, potentially dangerous methods are prohibited from accepting tainted arguments:

- Certain methods of `Dir`, `IO`, `File`, and `FileTest` refuse to accept tainted arguments.
- Tainted arguments are forbidden by `Kernel.eval`, `Kernel.load`—unless the load is wrapped—`Kernel.require`, `Kernel.test`, and `Kernel.trap`.
- The `RUBYLIB` and `RUBYOPT` environment variables are ignored at startup.
- The following command-line options are prohibited: `-e`, `-i`, `-I`, `-r`, `-s`, `-S`, and `-x`.
- Methods that execute programs prohibit tainted arguments or executing a program relative to `PATH` when a directory in `PATH` is world-writable.
- Instruction sequences can't be compiled or disassembled: `VM::InstructionSequence.compile`, `VM::InstructionSequence.compile_option=`,`VM::InstructionSequence.disasm`, `VM::InstructionSequence.disassemble`, `VM::InstructionSequence#diasm`, `VM::InstructionSequence#disassemble`, `VM::InstructionSequence#eval`, `VM::InstructionSequence.new`, and `VM::InstructionSequence#to_a` are disabled.

Level 2

At a safe level of 2 or higher, the following additional constraints are imposed on the manipulation of files and processes:

- Directories can not be created, deleted, or changed: `Dir.chdir`, `Dir.chroot`, `Fir.mkdir`, and `Dir.rmdir` are disabled.
- File metadata can not be changed: `File.chmod`, `File#chmod`, `File.chown`, `File#chown`, `IO#ioctl`, `File.lchmod`, `File#lchown`, `File.umask`, and `File.utime` are disabled.
- File metadata can not be queried: `File.executable?`, `File.executable_real?`, `File.ftype`, `File.identical?`, `File.lstat`, `File#lstat`, `File.readable?`, `File.readable_real?`, `File.readlink`, `File.realpath`, `File.setuid?`, `File.stat`, `File::Stat.new`, `File.symlink?`, `Kernel.test`, `File.writable?`, and `File.writable_real?` are disabled.
- Files can not be deleted, renamed, or locked: `File.delete`, `File.flock`, `File#flock`, `File.rename`, `File.truncate`, `File#truncate`, `File.unlink` are disabled.
- Symbolic and hard links can not be created: `File.link` and `File.symlink` are disabled.
- Syscalls can not be made: `Kernel.syscall` is disabled.
- Process IDs can not be queried: `Process.getpgid`, `Process.getpgrp`, `Process::Sys.issetugid`, `Process.pid`, and `Process.ppid` are disabled.
- Process IDs can not be changed: `Process.setpgrp`, `Process.setpgid`, `Process.setsid`, `Process::Sys.setgid`, `Process::Sys.setrgid`, `Process::Sys.setegid`, `Process::Sys.setregid`, `Process::Sys.setresgid`, `Process.gid=`, `Process::GID.change_privilege`, `Process::GID.grant_privilege`, `Process.egid=`, `Process::GID.eid=`, `Process::GID.re_exchange`, `Process::GID.switch`, and `Process::UID.switch` are disabled.
- Processes can not be manipulated: `Process.daemon`, `Process.detach`, `Process.wait`, and `Process.waitall` are disabled.
- Processes can not be executed: `Kernel.\``, `Kernel.exec`, `Kernel.fork`, `Kernel.spawn`, and `Kernel.system` are disabled.

- Process priorities and limits can neither be queried nor set: `Process.getpriority`, `Process.getrlimit`, `Process.setpriority`, and `Process.setrlimit` are disabled.
- Signals can be neither sent nor trapped: `Process.kill` and `Process.trap` are disabled.
- Garbage collection can not be disabled: `GC.stress` is disabled.

Level 3

At a safe level of 3 or higher, objects other than those predefined in the global environment, are tainted and untrusted by default. Further, objects can neither be untainted or trusted because `Object#untaint` and `Object#trust` are disabled.

Level 4

At a safe level of 4, the following additional restrictions are imposed:

- `Object` and untainted `Arrays`, `Hashs`, and `Strings` can't be modified.
- Neither global variables nor environment variables can be modified.
- Instance variables in untainted objects can neither be accessed nor removed.
- Untainted files neither be closed nor reopened. Neither files nor pipes can be written to.
- Untainted objects can't be frozen, and those created at a lower safe level can't be modified either. No object can be tainted or untrusted.
- Method visibility can't be changed.
- In untainted classes and modules, modules can't be included, and methods can't be aliased, defined, redefined, undefined, or removed.
- Objects can't be queried for metadata such as method and variable lists.
- Threads can not be manipulated, terminated—unless the thread is the current thread—use `Thread.abort_on_exception=` or `Thread#abort_on_exception=`, moved between thread groups, or have thread-local variables.
- The interpreter can't be terminated with `Kernel.abort`, `Kernel.exit`, or `Kernel.exit!`.

- Files can not be loaded with `Kernel.autoload`, unwrapped-`Kernel.load`, or `Kernel.require`.
- Symbols can't be converted to object references.
- The pseudo-random number generator can't be seeded with `Kernel.srand` or `Random.srand`.
- `Kernel.eval` can be passed a tainted `String`; this was prohibited at safe level 1, but safe level 4 is so restrictive that it's allowed again.

Trust

New objects and running code are *trusted* unless the safe level is at least 3, in which case they are *untrusted*. Untrusted code is prohibited from modifying trusted objects, so at safe level 3 and 4 code will not be able to modify objects created at a lower safe level.

An object may be explicitly trusted with `Object#trust` when `$SAFE < 3`; `Object#untrust` does the converse when `$SAFE < 4`. The `Object#untrusted?` predicate returns `true` when the receiver is not trusted; `false`, otherwise.

Context

`BasicObject#instance_eval` takes a string or block which it evaluates in the receiver's context, setting `self` to the receiver. The evaluated code can access the object's instance variables, invoke its private and protected methods, and define methods on its singleton class.

`BasicObject#instance_exec` is similar, but accepts any number of arguments which it passes to the required block.

Conversion

Selectors whose names begin with *to_* are expected to return the receiver converted to an object of the indicated class. A conversion is either *implicit* or *explicit*, as explained below:

Implicit Conversion

A method may expect an argument of a particular class. If it receives an object of another class it wishes to automatically convert that object into one of the desired class.

For example, `Array.new` may be called with an Array as an argument, which is copied to produce a new Array. If the argument is not of class `Array`, `Array.new` sends `:to_ary` to the argument, implicitly converting it to an `Array`.

`Array.new` sent `:to_ary` because it is part of the implicit conversion protocol. By responding to this message with an `Array` objects are declaring that they may be used in place of an `Array`.

If the argument responds to `:to_a` it can also be automatically converted to an `Array`. However, `Array.new` does not send this message for it is part of the *explicit* conversion protocol; the sender of the message must send the argument `:to_a` himself if he requires the conversion.

Flanagan & Matsumoto suggest that objects should implement an implicit conversion protocol if they have "strong characteristics" of the target class [Flan08, pp. 80–80] .

try_convert

`Array`, `Hash`, `IO`, `Regexp`, and `String` define a class method named `try_convert` which uses the appropriate implicit conversion protocol to convert the argument to the receiver's class. If the argument does not respond to the appropriate implicit conversion message, `.try_convert` returns `nil`.

For example, `String.try_convert(`*`object`*`)` returns *`object`*`.to_str` if *object* responds to `:to_str`; `nil` otherwise.

Guidelines

From the discussion above we can derive the following guidelines:

1. A method may send implicit conversion messages to its arguments.
2. Objects responding to such messages are declaring that they may be used in this way.
3. A method must not send explicit conversion messages to its arguments.

Explicit Conversion

An object which can be represented as an object of another class may implement the relevant explicit conversion protocol. It is explicit because the user must explicitly send the conversion message to the object to effect the conversion; the message should never be sent automatically by another method.

If an object implements an implicit conversion protocol that corresponds with an explicit conversion protocol, it should implement the explicit protocol, too. The implicit protocol is a superset of the explicit protocol, so there will not exist a scenario where an object would need to implement the former without the latter. This can be easily achieved by aliasing the implicit method to the explicit method. For example, if an object responds to `:to_ary` but not to `:to_a` it should alias `:to_a` to `:to_ary`.

From the discussion above we can derive the following rules:

1. A method should not send explicit conversion messages to its arguments.
2. Objects responding to implicit conversion messages that have explicit counterparts should respond to the latter, too.

Summary

The conversion protocol is summarised in the table below. The *Target Class* column indicates the class of the object the conversion method should return. The *Implicit* column specifies the message, if any, that is part of the implicit conversion protocol for the target class. Likewise, the *Explicit* column specifies the message, if any, that is part of the explicit conversion protocol

for the target class. If a protocol is not defined for a class, it has a value of *N/A* is given.

Note that the absence of an implicit protocol for a class implies that it should not be converted implicitly. Methods expecting arguments of a non-convertible class should raise a `TypeError` if they receive arguments of an unsuitable type.

Target Class	Implicit	Explicit	Note
`Array`	`:to_ary`	`:to_a`	
`Complex`	`:to_c`	N/A	
`Enumerator`	N/A	`:to_enum`	
`Float`	`:to_f`	N/A	
`Hash`	`:to_hash`	N/A	
`Integer`	`:to_int`	`:to_i`	
`IO`	`:to_io`	N/A	Sent by `IO.try_convert`
`IO`	`:to_open`	N/A	Sent by `Kernel#open` to its first argument
`Proc`	`:to_proc`	N/A	
`Rational`	`:to_r`	N/A	
`Regexp`	`:to_regexp`	N/A	Sent by `Regexp.try_convert`
`String`	`:to_str`	`:to_s`	
`String`	`:to_path`	N/A	Sent by methods expecting a file path as an argument, such as the class methods of `File`
`Symbol`	`:to_sym`	N/A	

Converting to "Boolean"

It is rarely necessary to convert an object to a "Boolean" because Ruby automatically treats `false` and `nil` as false, and any other value as true. However, if a method wishes to return either `true` or `false` it may use the `!!`*`object`* idiom. This returns `false` if *object* is `false` or `nil`; `true` otherwise.

CLASSES

A *class*[1] is a *classification* of objects. It defines a set of methods, and can mint objects in its image.

Names

A class[2] is named with a constant, and this name can be retrieved as a `String` by `Class#name`. This does not apply to anonymous classes, of course, which have a name of `nil`. Conventionally class names use camel-case capitalization: the initial letter of each word is capitalized, and spaces between the words are removed. For example: `RubyProgrammingLanguage` or `NutsAndBolts`. This distinguishes a class from a constant *qua* constant, as the latter is named in uppercase.

Inheritance

A class *inherits* behaviour and certain state-class variables and constants-from another class called its *superclass*. The exception is `BasicObject`, which sits at the top of the class hierarchy. The default superclass is `Object`. Classes that inherit from a given class are its *subclasses*. A subclass is, therefore, a specialisation of its superclass.

---

1. The term *class* is roughly analogous to its biological definition where it denotes a taxonomic rank, however this analogy does not extend to subclasses. That is, a subclass of a class is termed a *subclass*; not an *order*.
2. Class names containing non-ASCII characters cannot be referred to from source files with a different source encoding. For example, a class name containing character the Greek small letter lamda (`U+03BB`) can only be referenced from source files using the UTF-8 source encoding.

Superclass

The superclass is a `Class` object. It is typically specified as a constant literal, but can be any expression evaluating to the same. Once a class has been created, its superclass cannot be changed. `Class#superclass` returns the receiver's superclass as a `Class` object, or `nil` if the receiver is `BasicObject`.

Ancestors

The *ancestors* of a class are the classes and modules it inherits from: its superclass and mixed-in modules, then their ancestors, and so on up until the root of the inheritance hierarchy. They are returned, in order, by `Module#ancestors` as an `Array` of `Class` objects.

Inheritance merely determines the initial behaviour of a class; the subclass can diverge by defining, redefining, or removing methods, or modifying state. It occurs because the method— and constant—lookup algorithms consider the superclass when unable to resolve a given name against the current class. It is worth stating explicitly that instance variables and class variables are *not* inherited. [Flan08, pp. 239–240] note a corollary: "If a subclass uses an instance variable with the same name as a variable used by one of its ancestors, it will overwrite the value of its ancestor's variable."

Class#inherited Hook

If a class defines a singleton method named `:inherited`, it is invoked when the class is inherited with the subclass as its argument.

Creation

class Keyword

The `class` *`name`* `<` *`superclass`*…`end` expression opens a class named *name*. If the constant *name* is already defined it must refer to an existing class, otherwise a `TypeError` is raised. If *name* was previously undefined, it is created to refer to a new `Class` object. The `<` *`superclass`* portion may be

omitted, in which case the *superclass* defaults to `Object`. *superclass* may be any expression that evaluates to a `Class` object. The class body, which may be empty, is the elliptical region in the expression. It introduces a new context in which `self` refers to the class.

```
class Dog
end
```

Reopening Classes

If `class` is used with the name of a pre-existing class that class is *re-opened*. If a method is defined in a re-opened class with the same name as a pre-existing method in the same class the old method is overwritten with the new. Classes can be made immutable, effectively preventing them from being reopened by freezing the class object. Frozen classes raise `RuntimeErrors` when methods are defined, or variables manipulated, in their context.

```
'hello'.length #=> 5
class String
  def length
    'How long is a piece of string?'
  end
end
'hello'.length #=> 'How long is a piece of string?'
```

Class.new

The *name* `= Class.new do…end` constructor may be used to similar effect. The principle difference being that existing classes are overwritten rather than reopened.

```
Dog = Class.new
```

Anonymous Classes

When a class is named with a constant it is accessible wherever that constant is in scope. If this behaviour is not desirable, a class can be made anonymous by assigning the value of `Class.new` to a local variable, thus

restricting the class to the local scope. Subsequently assigning this variable to a constant, names the class.

```
dog = Class.new
dog.class_eval do
  def bark
    :woof
  end
end
dog.new.bark #=> :woof
```

Structs

`Struct` is a class generator, particularly useful for classes that only need to wrap data. It it instantiated with `Struct.new(`*name*`, `*members*`)`. *name* is a `String` beginning with a capital letter: the name of the returned `Struct` is `Struct::`*name*. If *name* is omitted, an anonymous `Struct` is created. *members* are zero or more `Symbols` naming the `Struct`'s attribute members. When a block is supplied, it is evaluated in the context of the new `Struct`'s class allowing methods to be defined on it.

The `Struct` returned, *s*, is itself initialised with a *s*`.new(`*arguments*`)` method, or its alias *s*`[]`. *arguments* is a list of arbitrary objects that initialise *s*'s members in the order the members were specified. If there are fewer *arguments* than members, the remaining members are initialised to `nil`; if there are more, an `ArgumentError` is raised. A list of all members is returned by *s*`.members` as an `Array` of `Symbols`.

Corresponding accessor methods are defined on *s* for each member: the value of a member, *m*, is returned by `s#`*m* and set to *v* with *s*`#`*m*`=(`*v*`)`. Alternatively, *s*`#[`*m*`]` retrieves member *m*, while *s*`#`*m*`=(`*v*`)` sets it to *v*. In both `#[]` and `#[]=`, *m* can specify a member by its name as a `Symbol`, or its `0`-based `Integer` offset in the list of `Symbol` arguments passed to `Struct.new`. If the specified member does not exist, a `NameError` is raised in the former case, and an `IndexError`, in the second.

The values for all members are returned as an `Array` by *s*`#to_a` and its alias *s*`#values`. *s*`#size`, and its alias *s*, return the number of elements in this `Array`.

The values of specific members are returned by *s*#values_at(*location*), where *location* is one or more Integer indices or Ranges.

Members are enumerated by *s*#each_pair as a name-value pair, while *s*#each enumerates just their value. Both methods yield their enumerations to a block, returning an Enumerator when the block is omitted. As the presence of #each implies, *s* is also an Enumerable.

```
Element = Struct.new(:name, :symbol, :atomic_number, :mass)
as = Element.new(:Arsenic, :As, 33, 74.92)
as.mass #=> 74.92
```

A popular idiom is to create a class that inherits from a Struct: the Struct defines the simple attributes, and the class body adds behaviour/customisations.

```
class Element < Struct.new(:name, :symbol, :atomic_number)
  def initialize(*args)
    super
    @poisonous = false
  end

  def poisonous?
    @poisonous
  end

  def poisonous=(bool)
    @poisonous = !!bool
  end
end
thallium = Element.new(:Thallium, :Tl, 81)
thallium.symbol #=> :Tl
thallium.poisonous = true
thallium.poisonous? #=> true
```

Nesting

A class may be defined within the body of another class. The fully qualified name of the inner class is then *outer*::*inner*: the name of the enclosing class (*outer*) separated from that of the enclosed (*inner*) with the

scope operator. This *nesting* behaviour is primarily used for namespacing, with modules being an alternative. However, it does not affect inheritance: if the inner class is to inherit from the outer class, it must do so explicitly. The nesting of a class is returned as an `Array` of `Class` objects by `Module#nesting`, where the first element is the innermost class, and the last the outermost.

Context

`Class#class_eval` takes a string or block which it evaluates in the receiver's context, setting `self` to the receiver. The evaluated code can access the class's state, invoke its singleton methods, and define methods. `Class#instance_exec` is similar, but accepts any number of arguments which it passes to the required block.

Singleton Classes

Every object is associated with two classes: that with which it was instantiated, and an anonymous class specific to the object: its *singleton class*[3]. That a singleton class is unique to a particular object means that methods defined within it-singleton methods-are also unique to that object.

Further, an object's class-i.e. the one which instantiated it-is the superclass of its singleton class. Upon receiving a message an object asks his singleton class for a method, the singleton class searches its instance methods and included modules, then repeats the query to his superclass. This process continues, recursively, up the inheritance hierarchy until a suitable method is located. Therefore, singleton methods override all others because the singleton class is the first place searched.

However, the singleton classes of `Class` objects behave slightly differently. Consider two classes, *c* and *p*. Now, if the superclass of *c* is *p*, then the superclass of the singleton class of *c* is the singleton class of *p*. This seemingly

3. [Flan08, pp. 257–258] use the term *eigenclass*, instead, but the preferred nomenclature is now *singleton class*. See Feature #1082: add Object#singleton_class method for the background.

convoluted arrangement creates an inheritance hierarchy of singleton classes parallel to that of normal classes, allowing class methods to be inherited.

```
# coding: utf-8
class P
  def self.convoluted?
    true
  end
end
C = Class.new(P)
C.superclass == P                                    #=> true
C.singleton_class.superclass == P.singleton_class #=> true
C.convoluted?                                        #=> true
```

The singleton class is a curious hybrid between class and module because although it has a superclass, it cannot be instantiated. However, the latter shortcoming is surely a blessing, as without it class hierarchies would be plexiform. Regardless, the abstractionists will delight in the fact that a singleton class has its own singleton class, *ad infinitum*…

Instances of the `Integer`, `Float`, and `Symbol` classes are the only objects not to have a singleton class; attempting to open one causes a `TypeError` to be raised.

The `Kernel#singleton_class`[4] method returns the receiver's singleton class as a `Class` object. It is typically paired with `#class_eval` so as to operate within the context of the class.

State

A class may store its state in class variables, as discussed previously, however, due to the unpopular semantics of class variables, class instance variables may be used instead.

---

4. Prior to Ruby 1.9.2 the peculiar `class << object…end` construct— the `class` keyword followed by two less-than signs, an expression evaluating to an object, then the class body—was used to open the singleton class of *object*.

Class Instance Variables

An instance variable used within a class definition, outside of an instance method, is a *class instance variable*. It is not to be confused with a class variable. Both kinds of variables are associated with the class, as opposed to its instances. The primary advantage of class instance variables over class variables is that they don't exhibit the latter's awkward sharing semantics: class instance variables are not shared with subclasses. However, class instance variables cannot be referenced in instance methods—as in that context they are normal instance variables—so are not necessarily appropriate substitutes.

Accessor methods can be created for class instance variables by using `Module#attr_accessor` and `Module#attr_writer` inside the class's singleton class.

```
# encoding: utf-8
class King
  @numeral = Hash.new {|h,k| h[k] = 8543.chr('utf-8') }
  singleton_class.class_eval{ attr_accessor :numeral }

  def initialize(name)
    King.numeral[@name = name].succ!
  end

  def name
    "#@name #{King.numeral[@name]}".sub(/ I$/, '')
  end
end
%w{Henry Stephen Henry Richard John Henry Edward Edward
   Edward Richard Henry}.map {|name| King.new(name).name}.join(', ')
#=> Henry, Stephen, Henry Ⅱ, Richard, John, Henry Ⅲ, Edward,
#   Edward Ⅱ, Edward Ⅲ, Richard Ⅱ, Henry Ⅳ
```

Instances

`ObjectSpace.each_object(`*`class`*`)` returns an `Enumerator` of a *class*'s instances.

Methods

`Module#instance_methods` returns the names of non-private instance methods defined in its receiver and superclasses as an `Array` of `Symbols`. `Module#public_instance_methods`, `Module#private_instance_methods`, and `Module#protected_instance_methods` are identical, except they return only those instance methods with the corresponding visibility. If these methods are given an argument which is `false`, methods defined in the receiver's superclass are omitted. To query another type of object for the methods it defines, see Methods.

method_defined? Predicate

The `Module#method_defined?` predicate accepts a method name as argument and returns `true` if the named instance method is defined on the receiver; `false` otherwise. `Module#public_method_defined?`, `Module#private_method_defined?`, and `Module#protected_method_defined?` behave in a similar fashion, but also require the named method to have the corresponding visibility These predicates are clearly similar to `#respond_to?` but they differ as follows:

- They test the instance methods of a class or module; `#respond_to?` tests the methods defined on its receiver.
- They can only be used on classes or modules; `#respond_to?` with any object inheriting from `Object`.
- They don't consult `#respond_to_missing?`—whereas `#respond_to?` does—which means that they don't reflect methods defined with method_missing.
- They return `true` for methods unimplemented on the user's platform; `#respond_to?` behaves conversely.

```
String.instance_methods.include?(:try_convert) #=> false
String.method_defined?(:try_convert) #=> false
String.singleton_methods.include?(:try_convert) #=> true
String.respond_to?(:try_convert) #=> true

String.instance_methods.include?(:upcase) #=> true
String.method_defined?(:upcase) #=> true
```

```
String.singleton_methods.include?(:upcase) #=> false
String.respond_to?(:upcase) #=> false
```

Either approach is normally preferable to *object*.methods.include?(*selector*), which has all of the disadvantages of #method_defined, in addition to being more verbose and less efficient.

Missing Classes

When a constant is used without being defined the enclosing class is sent a :const_missing message with the constant name as a Symbol argument. This is similar to :method_missing, but for classes.

Enumeration

ObjectSpace.each_object(Class) enumerates all Class objects currently defined. Therefore, to enumerate the subclasses of a given class, this list must be filtered as shown in the figure below.

```
require 'tempfile'
class Class
  def children
    ObjectSpace.each_object(Class).select{|c| c < self}
  end
end
Delegator.children #=> [SimpleDelegator, Tempfile, #<Class:0x8a2edbc>]
```

Type

"In many object-oriented languages, class names are used…for the type of objects generated from the class." [Bruce02, pp. 20–20] . [Klas95, pp. 10–10] concur: "A class…defines…the type of [its] instances". Applying this notion to Ruby is problematic because while it is certainly possible for a method to dynamically *type check* its arguments with the Kernel#is_a?(*class*) predicate, this approach is both insufficient and unnecessary.

It is insufficient because an object's class is not indicative of its suitability for a specific role. Class-based type checking rests on the premise that all objects of a given type will respond to the same messages in the same fashion. However, Ruby's classes may be modified at will—allowing their methods to be redefined or removed—so two objects of the same class will not necessarily provide the same behaviour. Similarly, methods may be defined on, or removed from, individual objects, again breaking the assumption.

It is unnecessary because Ruby offers a superior, meritocratic alternative called "duck typing" ([Thom09, pp. 372–372]): if an object responds to the messages it would be sent in the course of a computation it constitutes suitable input. The yardstick is ability; not class.

An optimistic method simply assumes that its arguments are suitable; allowing them to raise a `NoMethodError` if sent a message they don't understand. This allows for particularly flexible APIs at the cost of potentially obscure error messages for nonsensical arguments. More typically, the `Kernel#respond_to?(`*`selector`*`)` predicate is used to determine the suitability of an object. For instance, a method may raise an `ArgumentError` unless its argument responds to `:<<`. A refinement may to send an argument the appropriate `:try_convert` message, raising an exception if `nil` is returned.

MODULES

A *module* is "a named group of methods, constants, and class variables" [Flan08] . It is, therefore, similar to a class except it can neither inherit behaviour nor be instantiated. Indeed, `Module` is the superclass of `Class`, the latter defining only three additional methods: `:new`, `:allocate`, and `:superclass`.

Creation

module Keyword

The `module` *name*...`end` expression opens a module named *name*. If the constant *name* is already defined it must refer to an existing module, otherwise a `TypeError` is raised. If *name* was previously undefined, it is created as a `Module` object. That both modules and classes are named with constants, means they share the same namespace: a given (qualified) constant cannot name a class and a module simultaneously. The module body, which may be empty, is the elliptical region in the expression. It introduces a new context in which `self` refers to the current module.

Reopening Modules

If `module` is used with the name of a pre-existing module that module is *re-opened*. If a method is defined in a re-opened module with the same name as a pre-existing method in the same module the old method is overwritten with the new.

Modules can be made immutable, effectively preventing them from being reopened by freezing the module object. Frozen modules raise `RuntimeErrors` when methods are defined, or variables manipulated, in their context.

Module.new

The `Module.new` constructor can be used to create a new anonymous module. It is usually passed a block argument which is evaluated in the context of the created module using `module_eval`.

`Kernel.load` can[1] use an anonymous module to prevent a source file from affecting the global namespace. The classes and modules the source file defines are created in the context of this anonymous namespace, making them inaccessible to the caller. The source file must explicitly specify the constants it wishes to share with the caller by assigning them to global variables.

A module is an object like any other, so by assigning an anonymous object to a local variable, for example, it only exists for as long as the variable does, and is invisible from disjoint scopes.

However, magic happens if you assign an anonymous module to a constant: the module takes on its name. Specifically, sending `:to_s`[2] to an anonymous module causes a search to be done for the first constant the module was assigned to: if one is found, that becomes the module's name.

This technique cannot be used to re-open modules. Assigning an anonymous module to a constant naming another module creates a new module with the same name, clobbering the old one.

```
indeterminate = Module.new do
  def size
  end
end
indeterminate.name #=> nil
quantities = [3, 7, 11, 47]
quantities.extend(indeterminate)
quantities.size #=> nil
```

---

1. And does, when supplied with a second argument that is true, i.e. `load file, true`.
2. ...or `:inspect`, `:name`, and any other method which uses `#to_s` internally.

```
Indeterminate = indeterminate
indeterminate.name #=> "Indeterminate"
```

Mixins

The primary use of modules is *mixins*: imbuing a given class with the instance methods, or *features*, of a given module. A class may have any number of modules mixed in.

Mixing a Module into a Class

Mixing-in a module to a class effectively enables multiple inheritance: appending to the class features from any number of sources. By contrast, classes may only inherit from a maximum of one other class.

Mixing a Module into a Module

It is less common to mixin a module to another module, but legal nevertheless. The result is simply that the features of the included module are copied to the other module.

Inclusion

`Module#include` takes one or more module names as arguments, then mixes them in to the enclosing class. Contrary to intuition, the named modules are mixed-in in the reverse order in which they are named. That is, `include A, B` appends the features of module `B`, then module `A`. Therefore, if `A` and `B` define an instance method with the same name, `B`'s copy will be overwritten by `A`'s.

Specifically, when used in the context of a class named *class* `#include` sends each of its arguments `:append_features` with *class* as an argument. The default behaviour is defined by `Module#append_features`, which adds the named modules to *class*'s method search path then sends the :included message to each. A corollary is that if a class defines its own

`#append_features` method, it must call `super` so as to invoke `Module#append_features`; otherwise, the module is not mixed in.

included Callback

When a module has been included in a class it is sent a message named `:included`, with the class name as an argument. This allows the module to perform initialization such as dynamically adding additional methods or variables to the named class. Be aware, however, that `:included` is sent *every* time `Module#include` is used, even if the module is already included in the class.

`:included` is sent *after* the module has been included, so is powerless to prevent the inclusion or otherwise condition it on some prerequisite. Its return value is ignored because `Module#include` always returns the class object in which it is used.

Class#include?

The `Class#include?` predicate returns `true` if the module given as its argument has been mixed-in to the receiver; `false` otherwise.

Class#included_modules

`Class#included_modules` returns an `Array` of `Modules` mixed-in to the receiver in the reverse order of their inclusion. As `Object` includes the `Kernel` module, this method will usually return `Kernel` as the last element.

Extension

Given a list of modules, `Kernel#extend` mixes them into to a specific object by including them in its receiver's singleton class. Specifically, it sends each of its arguments an `:extend_object` message with the receiver as an argument. The default behaviour of `:extend_object` is provided by `Module`, which includes the module into the receiver's singleton class, then sends the module an `:extended` message. As with `Module#append_features`, modules

which define their own `#extend_object` methods must employ `super` to actually effect the extension.

```
module Instances
  def instances
    ObjectSpace.each_object(self).to_a
  end
end
harmonic_sequence = 1.upto(10).map{|d| Rational(1,d)}
def pell(n)
  return n if [0, 1].include?(n)
  2 * pell(n-1) + pell(n-2)
end
approx_sqrt_2 = (1..20).map{|n| Rational(pell(n-1) + pell(n), pell(n))}
Rational.extend(Instances)
Rational.instances.size #=> 30
```

Extending self

An idiomatic application of `extend` is to use `extend self` within the context of a module. The effect is to mix the current module into its singleton class:

- Public instance methods can be invoked as instance methods in the context of the module, i.e. without an explicit receiver, and singleton methods of the module from anywhere.
- Private and protected instance methods can only be invoked from within the context of the module.

An example of `extend self` is provided in a figure below.

Namespacing

Modules can be used for *namespacing*: to combine a set of methods with a common purpose so their names do not clash with unrelated methods, and they are able to share data. Classes can be used in the same way.

Modules are preferable to classes in this respect when the namespace cannot be sensibly instantiated. The use of a module clarifies this aspect of the API.

Methods are defined in the module's singleton class, then invoked with the module name as the receiver: *module*.*selector*.

```
module CPU
  def self.processors
    File.read('/proc/cpuinfo').split(/\n\n/).map{|processor| Hash[
        processor.lines.reject{|line| line.end_with?(':')}.
                        map{|line| line.chomp.split(/\t+:\s?/)}
    ]}
  end
end

cpu = CPU.processors.first
cpu['model name'] + ' with a ' + cpu['cache size'] + ' cache'
#=> "Intel(R) Atom(TM) CPU N270   @ 1.60GHz with a 512 KB cache"
```

Brown [Brown09, pp. 133–138] suggests this technique can be useful in the following circumstances:

- You are solving a single, atomic task that involves lots of steps that would be better broken out into helper functions.
- You are wrapping some functions that don't rely on much common state between them, but are related to a common topic.
- The code is very general and can be used standalone or the code is very specific but doesn't relate directly to the object that it is meant to be used by.
- The problem you are solving is small enough where object orientation does more to get in the way than it does to help you.

This technique can be used in conjunction with extend self to create private helper methods: accessible to the module methods; inaccessible from outside the module.

```
module CPU
  extend self
  def processors
    File.read('/proc/cpuinfo').split(/\n\n/).map{|processor| Hash[
        processor.lines.reject{|line| line.end_with?(':')}.
                        map{|line| normalise *line.chomp.split(/\t+:\s?/)}
    ]}
  end

  private
  def normalise(key, value)
    key = key.gsub(/\s/,'_').to_sym
    return [key, value.split.map(&:to_sym)] if key == :flags
    [key] << case value
      when /^\d+$/      then value.to_i
      when /^\d+\.\d+$/ then value.to_f
      when 'no'         then false
      when 'yes'        then true
      else              value
    end
  end
end

cpu = CPU.processors.first
cpu[:model_name] + ' has the :mmx flag set' if cpu[:flags].include?(:mmx)
#=> "Intel(R) Atom(TM) CPU N270   @ 1.60GHz has the :mmx flag set"
"It doesn't have any known bugs" if cpu.keys.grep(/_bug/).map{|bug| cpu[bug]}.none?
#=> "It doesn't have any known bugs"
"Its faster than 500MHz" if cpu[:cpu_MHz] > 500
#=> "Its faster than 500MHz"
```

Nesting

A module may be defined within the body of another class or module. The fully qualified name of the inner module is *outer*::*inner*, i.e. the module names are concatenated with the scope operator.

Module Functions

A single module may be used both as a namespace and a mix-in. The first case requires the module's methods to be singleton methods on the `Module` object; the second requires them to be instance methods.

```
Math.cbrt 33 #=> 3.2075343299958265
include Math
cbrt 3 #=> 1.4422495703074085
```

One solution is `Module#module_function`, which is used to convert a mix-in module to a namespace module. In the context of a module, `module_function` is provided with one or more instance method names as arguments. Alternatively, `module_function` may be invoked without arguments, in which case it affects all instance methods defined subsequently within the same module. In both cases, each affected method is copied to the module's singleton class then the original method is made private. The duplication satisfies the requirement that methods are both instance and singleton methods. The visibility change forces classes which mix-in the module to invoke its methods in the "functional style", that is without an explicit receiver, so as not to confuse them with traditional instance methods [Flan08, pp. 251–252] .

```
module ISBN
  module_function
  # Convert ISBN-13 to ISBN-10
  def thirteen_to_ten(isbn)
    isbn, weight = isbn.to_s[3,9].chars, 11
    csum = 11 - isbn.reduce(0){|mem,c| mem + (c.to_i * weight -= 1)} % 11
    isbn.join + ({10 => 'X', 11 => '0'}[csum] || csum.to_s)
  end
end

isbns = {'9780596529260' => '0596529260',
         '9780596102432' => '0596102437',
         '9780596007591' => '0596007590'}
isbns.all?{|thirteen, ten| ISBN.thirteen_to_ten(thirteen) == ten} #=> true
include ISBN
isbns.all?{|thirteen, ten| thirteen_to_ten(thirteen) == ten} #=> true
```

A weakness of `module_function` is that it prevents a singleton method of the module from calling a private method of the same module.

```
module ISBN
  module_function
  # Convert ISBN-13 to ISBN-10
  def thirteen_to_ten(isbn)
    csum = checksum10(isbn = isbn.to_s[3,9])
    isbn + ({10 => 'X', 11 => '0'}[csum] || csum.to_s)
  end

  private
  def checksum10(isbn, weight=11)
    11 - isbn.chars.reduce(0){|mem,c| mem + (c.to_i * weight -= 1)} % 11
  end
end

isbns = {'9780596529260' => '0596529260',
         '9780596102432' => '0596102437',
         '9780596007591' => '0596007590'}
isbns.all?{|thirteen, ten| ISBN.thirteen_to_ten(thirteen) == ten}
#=> `thirteen_to_ten': undefined method `checksum10' for ISBN:Module (NoMethodError)
```

extend self does not exhibit this problem, but neither does it offer the granularity of `module_function`—it copies all instance methods into the singleton class—nor automatically privatise instance methods.

```
module ISBN
  extend self
  # Convert ISBN-13 to ISBN-10
  def thirteen_to_ten(isbn)
    csum = checksum10(isbn = isbn.to_s[3,9])
    isbn + ({10 => 'X', 11 => '0'}[csum] || csum.to_s)
  end

  private
  def checksum10(isbn, weight=11)
    11 - isbn.chars.reduce(0){|mem,c| mem + (c.to_i * weight -= 1)} % 11
  end
end

isbns = {'9780596529260' => '0596529260',
```

```
        '9780596102432' => '0596102437',
        '9780596007591' => '0596007590'}
isbns.all?{|thirteen, ten| ISBN.thirteen_to_ten(thirteen) == ten} #=> true
include ISBN
isbns.all?{|thirteen, ten| thirteen_to_ten(thirteen) == ten} #=> true
```

Context

Module Eval

`Module#module_eval` takes a string or block which it evaluates in the receiver's context, setting `self` to the receiver. The evaluated code can access the modules's state, invoke its singleton methods, and define methods.

Module Exec

`Module#module_exec` behaves as `Module#module_eval` except it requires a block, to which it passes any arguments it has received.

METHODS

"A *method* is a named block of parameterized code associated with one or more objects." ([Flan08, pp. 176–176] . It is the "code found in a class *for responding to a message* " ([Mitchell04, pp. 523–523] , emphasis mine.

Having received a message, an object *invokes* a method in response. *Invoke* describes the calling of a specific method, in contrast to *send*, which is used at a higher level to describe a request for an object to perform a certain operation.

Instance Methods

An *instance method* is a method defined, without an explicit receiver, in the context of a class. Instances of this class invoke the method when sent a message with its name as the selector. Therefore, an object's instance methods determine its behaviour.

Notation

It is conventional to employ the `Class#method` notation when referring to instance methods in documentation and prose. For example: `Integer#even?`.

Global Methods

Methods defined at the top-level of a program, outside of any `Class` or `Module` definitions, are private instance methods of `Object`. This enables programming in a *functional* style, without thinking in terms of objects.

Singleton Methods

A *singleton method* is a method defined in the context of an object's singleton class. It is commonly described as a method defined on a specific object, rather than on all instances of a certain class.

Notation

It is conventional to employ the *`Class.method`* notation when referring to singleton methods in documentation and prose. For example: `File.exists?`.

Class Methods

Singleton methods defined on `Class` objects are known as *class methods*. For example, `File.absolute_path` is a class method defined on the `File` class. Class methods are further distinguished from singleton methods in that "a method defined as a singleton method of a class object can also be called on subclasses of that class.": class methods are considered by the method lookup algorithm so, unlike other singleton methods, they are inherited [Black09, pp. 384–385] .

Class methods are typically *factory methods* in that they are constructors (*manufacturers*) of the class' instances.

Per-Object Behaviour

"In [the] prototype-based [programming] paradigm…there are no classes. Rather, new kinds of objects are formed more directly by composing concrete, full-fledged objects, which are often referred to as prototypes." [Taivalsaari96, pp. 1–1] . Taivalsaari continues: "…unlike in class-based languages in which the structure of an instance is dictated by its class, in prototype-based languages it is usually possible to add or remove methods and variables at the level of individual objects." (*ibid.*, pp. 8–10).

Singleton methods (along with Kernel#instance_variable_set) afford the same abilities to the ostensibly class-based Ruby. Whereas inheritance allows

a class to be created that specializes a more general class, singleton methods allow creation of an object that specializes on a more general object. (Ruby can also support Taivalsaari's observation that "…in prototype-based languages object creation usually takes place by copying…" (*ibid.*) through Kernel#clone).

The ability to specify per-object behaviour is an improvement Ruby made on her predecessors. For example, in the context of Smalltalk Budd remarked that "It is not possible to provide a method for an individual object; rather every object must be associated with some class, and the behaviour of the object in response to messages will be dictated by the methods associated with that class." [Budd87, pp. 5–9] .

Return Values

A method is invoked with an expression so always returns a value[1]: the last statement executed, or `nil` if the method body is empty. If this value is an `Array` the method effectively returns multiple values.

```
def elvis
  :to_sender
end

def elvis
  [:to_sender, :hound_dog]
end
elvis #=> [:to_sender, :hound_dog]
```

A `return` statement in the body of a method causes it to terminate prematurely, immediately passing control back to the caller. The value returned is that of `return`'s arguments, or `nil` if it has none. If multiple arguments are given they will be returned as an `Array`.

```
def elvis
  return :to_sender, :hound_dog
  # Not reached
```

---

1. A `def` expression also returns a value (`nil`), but this is distinct from that of invoking the method.

```
end
elvis #=> [:to_sender, :hound_dog]
```

This gives rise to two conventions:

- A method uses a `return` statement iff it may return before its last statement.
- A method intended to return a value has that value as its last statement

super

The `super` keyword is used in method definitions to invoke a method defined in the superclass, or an ancestor thereof (collectively hereafter: an *ancestor*), with the same name. An implication worth stating explicitly is that an instance method in a class may use `super` to invoke an instance method in an included module.

If called without arguments it invokes the ancestor method with the arguments received by the current method (this is known as the *implicit argument form of super*). Otherwise, it passes the arguments it has been given to the ancestor method. In either case, the arguments are sent as they currently exist: if they have been modified by the method, their modified forms are sent. To explicitly invoke the ancestor method without any arguments use `super()`.

```
class Chef
  def make(dish)
    puts "Chef: You don't want #{dish}! Try this:"
    :dish_of_the_day
  end
end

class Cook < Chef
  def make(dish)
    [super, :salad].join(' and ').tr('_', ' ')
  end
end
```

```
puts "Meal: " + Cook.new.make(:guinea_fowl_fricassee_with_foie_gras)
#=> Chef: You don't want guinea_fowl_fricassee_with_foie_gras! Try this:
# Meal: dish of the day and salad
```

A `NoMethodError` is raised if a corresponding method is not defined in an ancestor[2]. This can be avoided by conditioning the call to `super`, on `defined? super`, which returns `nil` in this case.

```
module Letter
  def letter
    l = ('A'..'Z').reject{|l| Object.const_defined?(l)}.first
    mod = Module.new do
      include Letter
      define_method(:letter, ->{ super(); l })
    end
    self.class.module_eval{ include Object.const_set(l, mod) }
    '->'
  end
end
class Alphabet
  include Letter
  def letter(n)
    n.times.map{super()}.join
  end
end
Alphabet.new.letter(6) #=> "->ABCDE"
Alphabet.ancestors
#=> [Alphabet, F, E, D, C, B, A, Letter, Object, Kernel, BasicObject]
```

Names

A valid method name is one of the following:

- An identifier that is optionally followed by an equals sign (`U+003D`), question mark (`U+003F`), or exclamation mark (`U+0021`).
- An operator method selector.
- The element reference (`[]`) selector.

---

2. `super` does obey the Method Missing protocol, however, so before raising the exception it will send `:method_missing` to each class it encounters.

- The element set (`[]=`) selector.

Abbreviated Assignment

Pseudo operators such as `+=` and `||=` are neither operator methods nor methods; they are abbreviated assignment operators.

Names that begin with an identifier conventionally begin with a lowercase[3] letter or low line (`U+005F`).

```
# coding: utf-8
def π
  Math::PI
end
π #=> 3.141592653589793
```

Operator Methods

An *operator method* is an operator definable as a method. By defining a method with the corresponding name an object can receive an operator message like core classes do. For example, the expression `3 + 2` is a syntactical shortcut for `3.+(2)`; by defining a method named `+` you can[4] redefine, or *overload*, the fundamentals of arithmetic.

Defining

A method consists of a name, an optional parameter list, and a body. Names have already been discussed, the parameter list is explained in

---

3. It is legal for a method name to begin with an uppercase letter, but then they may be confused with constants or class names. In fact, several core classes use this convention for precisely this reason: they provide a syntactical shortcut for constructing instances. For example, `Array()` is a method of `Kernel` which coerces its argument into an `Array` object.
4. …in the sense that you *can* learn Java: insanityAndScornFromYourPeers *will* result.

Arguments below. The *body* is a sequence of statements terminated with the `end` keyword.

If *`expression`*`.` is omitted, an instance method named *symbol* is defined on the enclosing class; otherwise a singleton method[5] named *symbol* is defined on *expression*.

```
class Dog
  def bark
    :woof
  end
end
Dog.new.bark #=> :woof
```

Therefore, `def self.`*`name`* defines a class method for the enclosing class. `def` *`class`*`.`*`name`* defines a method named *name* on the class named *class*.

```
class Dog
  def self.breed
    [new, new]
  end
end
Dog.breed #=> [#<Dog:0x95cb530>, #<Dog:0x95cb508>]
```

method_added Callback

When an instance method is defined, the receiver (i.e. the containing class) is sent a `:method_added` message with the new method's name as an argument. Similarly, when a singleton method is defined the receiver is sent `:singleton_method_added` instead.

Dynamic Method Definition

An instance method can be defined dynamically with `Module#define_method(`*`name`*`,` *`body`*`)`, where *name* is the method's name given as a `Symbol`, and *body* is its body given as a Proc, Method, UnboundMethod, or block literal. This allows methods to be defined at

5. Kernel#define_singleton_methodmay be used to the same end.

runtime, in contrast to `def` which requires the method name and body to appear literally in the source code.

```
class Conjure
  def self.conjure(name, lamb)
    define_method(name, lamb)
  end
end
# Define a new instance method with a lambda as its body
Conjure.conjure(:glark, ->{ (3..5).to_a * 2 })
Conjure.new.glark #=> [3, 4, 5, 3, 4, 5]
```

`Module#define_method` is a private method so must be called from within the class the method is being defined on. Alternatively, it can be invoked inside `class_eval` like so:

```
Array.class_eval do
  define_method(:second, ->{ self.[](1) })
end
[3, 4, 5].second #=> 4
```

`Kernel#define_singleton_method` is called with the same arguments as `Module#define_method` to define a singleton method on the receiver.

```
File.define_singleton_method(:match) do |file, pattern|
  File.read(file).match(pattern)
end
File.match('/etc/passwd',/root/) #=> #<MatchData "root">
```

Arguments

We have already discussed how messages may be sent along with arguments. A method expecting to receive these arguments must define their type and quantity via a *parameter list*.

When a method is defined with the `def` keyword, its parameter list follows its name, and is usually[6] enclosed in a set of parentheses (`U+0028`, `U+0029`). It specifies:

- The local variable names (hereafter: *parameters*) to which the corresponding argument will be aliased inside the method.
- Whether the arguments are required or optional.
- Whether a fixed or variable number of arguments are allowed.
- The default value, if any, of each parameter.
- Whether an argument is expected to be a block.

Each parameter name must be unique in the parameter list.

Parameters are *positional*: they describe the argument in the corresponding position of the argument list. The correspondence isn't one-to-one, as we will see below, but each parameter consumes as many arguments as it can, leaving those that remain for the following parameters.

Required Arguments

A *required argument must* be supplied by the caller: A method which requires exactly *n* arguments must receive them all, otherwise it raises an `ArgumentError`.

nil and false

`nil` and `false` are as valid an argument any other. If a method requires an argument and `nil` is supplied in its place, Ruby will not complain. Methods wishing to prohibit such values must do so themselves.

Required arguments are specified with a comma-separated list of identifiers. Each parameter represents a mandatory argument.

```
class Llama
  def laugh(how, volume)
```

---

6. The parentheses are actually optional, but their omission doesn't enhance readability so they are recommended.

```
    puts volume == :loudly ? how.upcase : how
  end
end
Llama.new.laugh
#=> ArgumentError: wrong number of arguments (0 for 2)
Llama.new.laugh(:snicker)
#=> ArgumentError: wrong number of arguments (1 for 2)
Llama.new.laugh(:chortle,:softly)
#=> chortle
Llama.new.laugh(:guffaw,:loudly)
#=> GUFFAW
Llama.new.laugh(:ho, :ho, :ho)
#=> ArgumentError: wrong number of arguments (3 for 2)
```

Optional Arguments and Default Values

An *optional argument may* be supplied by the caller. If it is not, the corresponding parameter will be assigned the value given as its default.

Optional arguments are specified with a comma-separated list of *name=value* pairs, where *name* is a parameter name and *value* is its default value. The *value* may be any Ruby expression, and is permitted to refer to previous parameters. The default value expression is evaluated each time the method is invoked, so may, for example, instantiate an object on every invocation.

```
class Llama
  def name(name='Larry')
    name + ' the llama (beast of burden)'
  end
end
Llama.new.name
#=> "Larry the llama (beast of burden)"
Llama.new.name('Lyle Jr.')
#=> "Lyle Jr. the llama (beast of burden)"
Llama.new.name('Lama', 'glama')
#=> ArgumentError: wrong number of arguments (2 for 1)
```

Optional and required arguments can be specified alongside each other in a reasonably natural fashion. However, all optional arguments must be

adjacent. For instance, it is a syntax error to both precede and follow a required argument with optional arguments.

```
def required_optional(a,b=1) end
def required_required_optional(a,b,c=1) end
def optional_required(a=1,b) end
def optional_optional_required(a=1,b=2,c) end
```

Variable-Length Argument Lists

A *rest parameter* (or *splat parameter*) consumes every argument that follows it while still allowing subsequent required parameters to receive their corresponding arguments. Put simply: it takes an arbitrary number—including zero—of arguments from its position onward. It is passed to the method as an `Array` containing one argument per element.

A rest parameter is specified by preceding a parameter name with an asterisk (`U+002A`). Only one rest parameter may appear in a parameter list, and any optional parameters must precede it.

```
def zero_or_more(*rest)
  rest.join(', ')
end
zero_or_more #=> ""
zero_or_more(1) #=> "1"
zero_or_more(1,2,3) #=> "1, 2, 3"
```

A rest parameter may be supplied as a sole asterisk, omitting the corresponding parameter name. The effect is to consume the corresponding arguments as an ordinary rest parameter, without assigning them to a local variable. The arguments are ignored. This allows methods to accept an arbitrary number of arguments, but discard, say, all but the last.

Thomas et al. suggest that this technique can used in conjunction with the implicit-argument form of super to define a method which accepts an arbitrary number of arguments then passes them all to its superclass [Thom09, pp. 138–139] .

```
class Parent
  def do_chores(*chores)
  end
end

class Child < Parent
  def do_chores(*)
    # our processing
    super
  end
end
```

Named Arguments

The parameter forms described above are positional in nature. An alternative approach, that can aid the readability of otherwise ungainly parameter lists, is *named arguments*, which allow the method to be invoked with a series of key-value pairs, arranged in an arbitrary order.

This style of argument passing is not supported explicitly, but can be ably approximated by defining methods that expect a `Hash` argument: the keys of which become the parameter names; and the values, the arguments. This technique is used by core methods such as `File.open` and `String#encode`.

```
class Chair
  def initialize(args)
    @legs = args[:legs] or raise ArgumentError
    @style = args[:style] || :victorian
    @height = args[:height] || :average
    @colour = args[:colour] || args[:color] || :brown
  end
end
Chair.new legs: 4, height: :tall
#=> #<Chair:0x8249908 @legs=4, @style=:victorian,
    #                 @height=:tall, @colour=:brown>
Chair.new(:color => :fuschia, :legs => 7)
#=> #<Chair:0x86958f4 @legs=7, @style=:victorian,
    #                 @height=:average, @colour=:fuschia>
Chair.new(height: :childs, color: :fuschia)
#=> ArgumentError
```

If a Hash literal is the final argument, other than a block, that a method expects, the curly braces which delimit it can be omitted, as long as there is white space between the selector and the first key. For example, *`selector`*`({ key: :value })` can be written as *`selector`* `key: :value`.

The advantages of this approach include:

- Arguments can be specified in any order.
- Arguments with default values can be omitted.
- If `Symbol`s are used for the `Hash` keys the invocation is particularly readable.
- Variable-length argument lists are supported.

```
class Chair
  DEFAULT_ARGS = {legs: 2, style: :victorian, height: :average, colour: :brown}
  def initialize(args)
    @attributes = DEFAULT_ARGS.merge args
  end
end
Chair.new legs: 4, height: :tall #=>
  #<Chair:0x9039af4 @attributes={
  #:legs=>4,       :style=>:victorian,
  #:height=>:tall, :colour=>:brown
  #};>
Chair.new(:color => :fuschia, :legs => 7) #=>
  ##<Chair:0x998a9dc @attributes={
  #:legs=>7, :style=>:victorian,
  #:height=>:average, :colour=>:fuschia
  #}>;
```

The primary shortcoming is that Ruby can not determine automatically whether an invalid number of arguments have been supplied; the programmer must validate the arguments instead. This is unlikely to be particularly significant, however, because a method expecting a variable number of arguments would otherwise use a rest parameter, which also preclude automatic validation.

Block Arguments

Any method may be sent a block argument. The block may be yielded to, allowing its return value to be captured, or objectified and assigned to a variable. A method may determine whether it has received a block with the `Kernel.block_given?` predicate.

Iterator Methods

Flanagan & Matsumoto insist [Flan08] insist on "[using] the term iterator…to mean any method that uses the `yield` statement", despite admitting that this doesn't make sense if the method doesn't actually *iterate* over the block it has been given. We will not perpetuate this confusion: an *iterator method* iterates over the block it has been given; a method which expects a block but does not iterate over it is simply *a method that expects a block*.

```
def m
  :m
end
m { 1 + 2 }
m do
  1 + 2
end
```

A block passed to a method in this way is not automatically called; the method must use the yield keyword to do so. An implication is that methods not expecting blocks will ignore them.

```
def m
  puts "This block returns: #{yield}" if block_given?
end
m { 1 + 2 } #=> This block returns: 3
m #=> nil
```

A method needing a reference to the block it was given, perhaps to pass to another method, is defined with a final parameter whose name is prefixed with an ampersand (`U+0026`). The method can access the block as a `Proc`

object named after the parameter (sans ampersand). It may invoke the block via its `Proc#call` method, or yield to it. In either case, the method is invoked in precisely the same way as before.

```
def m(&block)
  puts "This block returns: #{block.call}" if block_given?
end
m { 1 + 2 } #=> This block returns: 3
m #=> nil
```

The discussion above applies only to block *literals*; a method expecting a *reference* to a block, i.e. a `proc` or `lambda`, need not pay heed. Such a method employs precisely the same parameter list as in the previous sections.

```
def m(b)
  b.call
end
m ->{ "I am a \u{3bb}!" } #=> "I am a λ!"
```

Pass By Reference

Arguments are passed to methods by reference instead of value. If the method modifies an object it receives the caller's copy is modified, too.

```
def llama_sans_l(llama)
  llama.gsub!(/l/i,'')
end
llama = 'Larry'
llama_sans_l(llama) #=> 'arry'
llama #=> 'arry'
```

Alternatively, an argument may be duplicated to create a copy independent of the caller's.

```
def llama_sans_l(llama)
  llama.dup.gsub!(/l/i,'')
end
llama = 'Larry'
llama_sans_l(llama) #=> 'arry'
llama #=> 'Larry'
```

Arity

The arity of a method is the number of arguments it takes. If the method expects a fixed number of arguments, this number is its arity. If the method expects a variable number of arguments, its arity is the additive inverse of its parameter count. Methods implemented in C, i.e. most core methods, have an arity of `-1` if they accept a variable number of parameters. It follows, then, that an arity ≥ `0` indicates a fixed number of parameters; a negative value, a variable number. `Method` and `Proc` objects have `#arity` methods which return the arity for the method/proc it objectifies.

Classification by Arity

Methods with fixed arities can be classified as follows: A *unary method* expects exactly one operand (its receiver), a *binary method* requires two (its receiver and one argument), *ternary-method* requires exactly three (its receiver and two arguments), an *n-ary method* requires *n* operands (its receiver, and *n*-1 arguments).

Undefining

Undefining a method prevents the current class from responding to it. If the method was defined in a superclass, that copy is unaffected. For example, consider a `Rectangle` class which defines `:height` and `:width` methods. A `Square` class inherits from it, but it doesn't make sense for `Square` to have both `:height` *and* `:width` methods. `Square` can `undef :height`, preventing `Square#height` from being called without affecting `Rectangle#height`.

The `undef` statement takes one or more `Symbols`/identifiers as arguments, then undefines the corresponding instance methods. Undefining singleton methods requires `undef` to be used in the context of the corresponding singleton class.

Alternatively, `Module#undef_method` may be used with the same effect. Unlike `undef`, `undef_method` doesn't accept an identifier as an argument; it expects to receive the method name as a `Symbol` or `String`. However, `#undef_method` has the advantage of being a method, as opposed to `undef`

which is a keyword, which allows it to be used with Ruby's reflective capabilities.

```
def boo!
  "(goose)"
end
boo! #=> "(goose)"
undef :boo!
boo! #=> NoMethodError: undefined method `boo!' for main:Object
```

method_undefined Callback

When an instance method is undefined, the receiver (i.e. the containing class) is sent a `:method_undefined` message with the method's name as an argument. Similarly, when a singleton method is undefined the receiver is sent `:singleton_method_undefined` instead.

Removing

`Module#remove_method` *name* removes the method named *name* from the current class only. A *removed method* differs from an undefined method in that the former delegates the request to its superclass, whereas the latter doesn't.

```
class Parent
  def says
    "Tidy your room!"
  end
end

class Child < Parent
  def says
    "In a minute..."
  end
end

Child.new.says #=> "In a minute..."

class Child
```

```
  remove_method :says
end

Child.new.says #=> "Tidy your room!"
```

method_removed Callback

When an instance method is removed, the receiver (i.e. the containing class) is sent a `:method_removed` message with the method's name as an argument. Similarly, when a singleton method is removed the receiver is sent `:singleton_method_removed` instead.

Visibility

The *visibility* of a method specifies the contexts in which it may be invoked. A method is *public* by default unless defined outside of a class definition. Public methods are accessible to everyone. A method created outside of a class or module definition, or named `initialize`, is *private* by default. Private methods can only be invoked with an implicit receiver, therefore from their defining class or a subclass thereof. Instance methods may also be *protected*, which means that they may only be invoked by objects of their defining class or a subclass thereof.

```
class C
  # This method is public because it hasn't been specified
  # otherwise
  def pub
  end

  private
  # This method is private because it appears after the
  #'private' visibility specifier
  def pri
  end

  # This method is also private because the previous visibility
  # specifier has not been overridden
  def pri2
  end
```

```
  protected
  # This method is protected because it appears after the
  #'protected' visibility specifier
  def pro
  end

  public
  # This method is public because the protected visibility
  # specifier has been explicitly overridden. Typically this would
  # have been defined after 'pub', removing the need for a
  # visibility specifier
  def pub2
  end
end
```

The visibility of an instance method may be altered with `Module#public`, `Module#private`, and `Module#protected`—collectively *visibility specifiers*. A visibility specifier invoked without arguments affects every method subsequently defined in the same class definition until another visibility specifier is encountered. Alternatively, a visibility specifier may be given an argument list of method names—as `Symbols` or `Strings`—whose visibility it alters. The visibility of a class method may be altered with `Module#public_class_method` or `Module#private_class_method`, both of which require an argument list of method names.

```
class C
  # The three following methods are public because they haven't been
  # specified otherwise
  def pub
  end

  def pri
  end

  def pri2
  end

  # Both :pri and :pri2 are made private because their names are
  # given as arguments to the 'private' visibility specifier
  private :pri, :pri2
```

```
  # This method is also public; the preceding private keyword only
  # acts on its arguments
  def pro
  end
  # This method is made protected because its name is given to
  # the 'protected' visibility specifier
  protected 'pro'

  # This method is public because it hasn't been declared
  # otherwise; the previous 'protected' specifier only affects the
  # method it was called for
  def pub2
  end
end
```

Method visibility is a property of the binding between a method and a class. An implication is that even when a class is frozen, the visibility of the methods it defines may still be changed using `Object#send`.

Advisory Privacy

Method visibility is merely an advisory construct. Ruby does not *prohibit* the invocation of private methods; she ensures that they will not be called accidentally as follows:

- Standard method invocation syntax (`obj.method`) raises a `NoMethodError`, signaling that the programmer's intent is ill-advised. The caviller programmer must use a technique such as `Object#send` to explicitly ignore the privacy advice.
- The method introspection API (e.g. `Object#private_methods`, `Object#protected_methods`, and `Object#public_methods`) delineates methods by their visibility, allowing private and protected methods to be determined automatically.
- RDoc/`ri` only displays public methods by default.

Aliases

An *alias* of a method is an alternate name by which it can be referred. For a method, *m*, and its alias, *a*, invoking *m* is equivalent to invoking *a*.

The alias refers to a copy of the existing method's body. If the existing method is redefined after being aliased, the alias will continue to refer to the method's original definition.

Aliases are often used to provide synonyms for method names. For instance, `:size` may be aliased to `:length`. This allows the programmer to use method names which "read" more naturally in a given context.

An alias is created with the `alias` keyword from inside the class of the existing method. The syntax is `alias` *`new_name current_name`* , where both *new_name* and *current_name* are `Symbol` literals or identifiers. A method named *current_name* must already be defined. If a method named *new_name* already exists it is overwritten.

`Module#alias_method` *`new_name`*, *`current_name`* can be used to the same effect, the difference being that it uses standard method semantics to interpret identifier arguments: treating them as expressions; not literal method names. This allows methods to be aliased dynamically. For example, `alias new old` interprets its arguments as literal identifiers, whereas `alias_method` *`new`*, *`old`* sees them as variables, whose values are the method names. By implication, if the arguments to `Module#alias_method` *should* be interpreted as literal identifiers they must be supplied as `String` or `Symbol` literals, e.g. `alias_method :new, :old`.

Aliasing is also used to create a method which wraps the method of the same name by performing its own computations then calling the original method. For example, in the example below we wrap `String#to_i` such that it raises an exception if the string doesn't contain digits. (Normally, `String#to_i` returns `0` for such strings).

```
class String
  alias :old_to_i :to_i
  def to_i
```

```
    raise "No digits found" unless match(/\d/)
    old_to_i
  end
end
```

Lookup Algorithm

Evaluating a message expression requires the corresponding method definition be located in the receiver. The steps below illustrate the lookup algorithm for a message with a selector of *selector*, and arguments of *arguments*, where *class* is set the receiver's singleton class. When a method is found whose name is equal to *selector* the process terminates. It will always ultimately succeed because `BasicObject` defines a `:method_missing` method.

1. Search the instance methods of *class*.
2. Search the instance methods of each module included by *class*, in reverse order of inclusion.
3. If *class* has a superclass[7], assign its name to *class* then go to step one.
4. Prepend *selector* to *arguments*. Set *selector* to `:method_missing`, *class* back to the receiver's singleton class, and go to step one.

```
class Parent
  def method
    :superclass_instance # 4
  end
end
module M
  def method
    :module_instance     # 3
  end
end
class C < Parent
  include M
  def method
    :class_instance      # 2
  end
end
```

---

7. `BasicObject` doesn't have a superclass, so this step is guaranteed to terminate.

```
object = C.new
def object.method
  :object_singleton      # 1
end
```

Missing Methods

NoMethodError

The `NoMethodError` exception has an `#args` method which returns the arguments sent to the original method as an `Array`, and `#name` which returns the original method name as a `Symbol`. This information can be used to perform introspection on the caught exception and enhance error messages.

The exposition above shows that sending an object a message for which a corresponding method is not defined causes each object on the search path to be sent `:method_missing` with the original selector as the first argument, and the original arguments as the remainder. `BasicObject` defines `:method_missing` to provide the default behaviour of raising `NoMethodError` for non-existent methods. However, if another object defines `:method_missing` they can intervene, averting the exception and responding to the message themselves.

```
class BasicObject
  public :method_missing
end
b = BasicObject.new
b.method_missing :glark
#=> NoMethodError: undefined method `glark' for #<BasicObject:0x93ace0c>
```

Infinite Loops

Logic errors inside `method_missing` can easily lead to infinite loops, which can be troublesome to debug. The typical mistake is for a statement in the body of `method_missing` to send a non-existent message to the same object.

The object does not respond to that message, so the `method_missing` method is invoked a second time, and so the loop continues.

If `method_missing` is invoked with a message it does not wish to respond to, it should use the implicit-argument form of `super` to delegate to its parent. This gives the parent the option of responding to the message, or otherwise triggers the default behaviour.

Fowler describes the use of `method_missing` to "respond differently to an unknown message." as "Dynamic Reception" [Fowler08] . One use he suggests is to "convert what might otherwise be method parameters into the name of the method.", contrasting `find_by("firstname", "martin", "lastname", "fowler")` with the `method_missing`-supported `find_by_firstname_and_lastname("martin", "fowler")`[8]. He identifies a variation on this idea where a "a sequence of Dynamic Receptions" are chained together, such that each method returns an "Expression Builder", e.g. `find_by.firstname("martin").and.lastname("fowler)`.

Kernel#respond_to_missing?

A consequence of defining methods dynamically with `Kernel#method_missing` is that an object may respond to a given message, yet return `false` for `Kernel#respond_to?(`*`selector`*`)`.

Before `Kernel#respond_to?` returns `false` it tries to send itself a message named `:respond_to_missing?` with a first argument of the selector in question, and the second the value of *include_private*[9]. If `#respond_to_missing?` is defined and returns a true value, `#respond_to?` returns `true`; otherwise `#respond_to?` returns `false`.

Therefore, `#respond_to?` can be made to work with methods defined with `method_missing` by defining a `#respond_to_missing?` method which returns `true` when passed the name of such a method.

---

8. The example was derived from a feature of Ruby on Rails [Buck06] .
9. If `#respond_to?` is called with a second argument of `true`, *include_private* is `true` and private methods should be considered; otherwise it's `false` and they shouldn't.

Method Objects

An instance of the `Method` class represents a method bound to an object. This *method object* enables you to store a reference to a method in a variable, as you would any other object, query the method's metadata, and manipulate it. This is quite distinct from capturing the return value of a method.

Method objects can be created with `Kernel#method`: *`receiver`*`.method(`*`name`*`)`, where *name* is the method name as a `Symbol` or `String`. For example, `method(:eval)` returns a `Method` object for `Kernel#eval`. If the object does not `respond_to?` the given method a `NameError` will be raised[10].

`Kernel#public_method` works in the same way, but raises a `NameError` if the given method is private or protected.

Arity

`Method#arity` returns an `Integer` corresponding to the method's arity.

Calling

The method represented by a `Method` object can be invoked with `Method#call` or its alias `Method#[]`. The semantics are the same as for standard method invocation, however Flanagan & Matsumoto caution: "...invoking a method through a `Method` object is less efficient than invoking it directly." [Flan08]

Converting to a Proc.

A method object can be converted to a `Proc` by prefixing it with an ampersand (). Therefore it can be passed to a method expecting a block.

10. By implication, if the receiver's #respond_to_missing? returns `true` for the method in question, the method object will be created successfully; only when the underlying method needs to be called will Ruby establish the veracity of this claim.

Equality

`Method#==` returns true if both methods are bound to the same object and have the same body. The first requirement means that the objects must be identical in the sense of `Object.equal?`. The second encompasses methods defined with `Object#define_method` using the same `Proc`/block, aliases created with `alias`, and core method aliases.

Source Location

The filename and line number where a method was defined is returned as an `Array` by `Method#source_location`. If the method is core, i.e. implemented in C, it returns `nil`. This is primarily useful for extracting a method's signature and any preceding documentation.

Parameters

`Method#parameters` returns an `Array`, each element of which is a sub-`Array` of `Symbols` that describe the corresponding parameter expected by the method. The first `Symbol` is `:req` for a required parameter, `:opt` if it is optional, `:rest` if its of variable length, or `:block` if it corresponds to a block. The last `Symbol` is the name of the parameter. An empty `Array` is returned for method's expecting no arguments.

UnboundMethod Objects

An `UnboundMethod` object is a `Method` object disassociated from the object on which it was defined.

A `Method` object may converted to an `UnboundMethod` object with `Method#unbind`. Alternatively, they can be created with `Module#instance_method`. For example, `String.instance_method(:downcase)` creates an `UnboundMethod` object for `String#downcase`. `Module#public_instance_method` works in the same way, but raises a `NameError` if the given method is private or protected.

Before an `UnboundMethod` is invoked it must be re-associated with, or *bound* to, to an object which is a `Object#kind_of?` its original class. This is achieved by passing an object reference to `UnboundMethod#bind`.

With the exception of `#call`, for the reason described above, `UnboundMethod` objects support the same method's as `Method` objects.

Black [Black09, pp. 418–420] provides the following example (with minor adjustments for formating) of using `UnboundMethod` objects:

> The following question comes up periodically in Ruby forums:
>
> > Suppose I've got a class hierarchy where a method gets redefined:
> >
> > ```
> > class A
> >   def a_method
> >     puts "Definition in class A"
> >   end
> > end
> >
> > class B < A
> >   def a_method
> >   puts "Definition in class B (subclass of A)"
> >   end
> > end
> >
> > class C < B
> > end
> > ```
> >
> > And I've got an instance of the subclass (`c = C.new`). Is there any way to get that instance of the lowest class to respond to the message (`a_method`) by executing the version of the method in the class two classes up the chain?
>
> By default, of course, the instance doesn't do that; it executes the first matching method it finds as it traverses the method search path: `c.a_method`. The output is `Definition in class B (subclass of A)`. But you can force the issue through an unbind and bind operation:

`A.instance_method(:a_method).bind(c).call`. Here the output is `Definition in class A`.

CLOSURES

"A closure is a combination of a function and an environment." [Graham96, pp. 107–109] . The function is a parametrised block of executable code, and the "referencing environment" [Scott06, pp. 138–140] , or binding, is a reference to the lexical environment of the closure's creation site. The binding represents its variables as references, which are de-referenced in the environment the closure is called, every time it is called.

```
variable = :first
# Use a lambda literal to create a closure
# whose function is the block and environment
# includes  _variable_. Assign the closure
# to a variable.
closure = ->{ "variable = #{variable}" }
# Invoke the closure, using the value
# of _variable_ from its environment
closure.call #=> "variable = first"
# Assign a different value to a variable
# in the closure's environment
variable = :second
# Re-invoke the closure in the context
# of its current environment, where the
# value of _variable_ is :second
closure.call #=> "variable = second"
```

A closure is an instance of the `Proc` class, which provides methods for calling the closure and accessing its binding. The following example shows a closure being called with `Proc#[]` and an argument.

```
array=[*('a'..'c')] #=> ["a", "b", "c"]
element =->(i){ print "#{i}=#{array[i]} " }
# For each element of _array_, call the _element_
# closure with the index as an argument
array.each_index {|i| element[i] } #=> 0=a 1=b 2=c
```

Proc Literals

A block literal creates a `Proc` object which accepts the arguments provided in the optional parameter list and represents a sequence of zero or more statements. Unlike most other literals, block literals must not appear in the top-level context; they must terminate a message expression, an appropriate keyword expression, or lambda literal.

Semantics

A `Proc` has either `yield` semantics or invocation semantics. Its semantics determine how it handles unexpected arguments and control flow statements, such as `return`, appearing within the body of the closure. The differences are summarised in the following table, and elaborated below.

Difference	Invocation	`yield`
Extra arguments	Raise ArgumentError	Ignored
Omitted arguments	Raise ArgumentError	Assigned `nil`
`Array` arguments	Never exploded	Exploded if necessary
`return`	Returns from the lambda itself	Returns from the creation site method
`break`	Returns from the lambda itself	Returns from the call site method

A comparison of `Proc` semantics

#lambda? Predicate

`Proc#lambda?` is a predicate which returns `true` if the receiver has invocation semantics; `false` if it has `yield` semantics.

yield Semantics

Argument Passing

Interprets the arguments it receives with `yield` semantics.

`return`

Returns from the lexically enclosing method of the `Proc`'s *creation* site.

```
def inner &closure
  puts "\t\tmain -&gt; outer -&gt; inner: Calling closure &lt;call site&gt;"
  closure.call
  puts "\t\tmain -&gt; outer -&gt;  inner: Called closure &lt;/call site&gt;"
end
def outer
  puts "\tmain -&gt; outer: Invoking inner &lt;creation site&gt;"
  inner do
    puts "\t\t\tmain -&gt; outer -&gt; inner -&gt; closure: Return from closure"
    return
    puts "\t\t\tmain -&gt; outer -&gt; inner -&gt; closure: Returned from closure"
  end
  puts "\tmain -&gt; outer: Invoked outer &lt;/creation site&gt;"
end
puts "main: Invoking outer"
outer
puts "main: Invoked outer"
#=&gt; main: Invoking outer
#=&gt;   main -&gt; outer: Invoking inner &lt;creation site&gt;
#=&gt;      main -&gt; outer -&gt; inner: Calling closure &lt;call site&gt;
#=&gt;        main -&gt; outer -&gt; inner -&gt; closure: Return from closure
#=&gt; main: Invoked outer
```

If the `Proc` was not created within a method, e.g. at the top level, or the method has already returned, a `LocalJumpError` is raised.

```
Proc.new { p :alpha; return; p :beta }.call #=> :alpha
#=> in `block in <main>': unexpected return (LocalJumpError)
```

`break`

Returns from the lexically enclosing method of the `Proc`'s *call* site.

```
def inner &closure
  puts "\t\tmain -&gt; outer -&gt; inner: Calling closure &lt;call site&gt;"
  closure.call
  puts "\t\tmain -&gt; outer -&gt;  inner: Called closure &lt;/call site&gt;"
end
def outer
  puts "\tmain -&gt; outer: Invoking inner &lt;creation site&gt;"
  inner do
    puts "\t\t\tmain -&gt; outer -&gt; inner -&gt; closure: Breaking from closure"
    break
    puts "\t\t\tmain -&gt; outer -&gt; inner -&gt; closure: Broken from closure"
  end
  puts "\tmain -&gt; outer: Invoked outer &lt;/creation site&gt;"
end
puts "main: Invoking outer"
outer
puts "main: Invoked outer"
#=&gt; main: Invoking outer
#=&gt;   main -&gt; outer: Invoking inner &lt;creation site&gt;
#=&gt;     main -&gt; outer -&gt; inner: Calling closure &lt;call site&gt;
#=&gt;       main -&gt; outer -&gt; inner -&gt; closure: Breaking from closure
#=&gt;   main -&gt; outer: Invoked outer &lt;/creation site&gt;
#=&gt; main: Invoked outer
```

A `LocalJumpError` is raised if `break` is used from a block no longer in scope, e.g. at the top-level of a block created with `Proc.new` or `proc`.

```
Proc.new do
  p :alpha
  break
  p :beta
end.call
#=> :alpha
#=> in `block in <main>': break from proc-closure (LocalJumpError)
```

Invocation Semantics

Argument Passing

Interprets the arguments it receives according to the same rules as method invocation. This has the following implications:

- Superfluous arguments cause an `ArgumentError` to be raised.
- Omitted arguments cause an `ArgumentError` to be raised.
- `Array` arguments are *not* automatically exploded.

`return`
: Returns from the `Proc` as if it were a method.

```
hug = ->(who) do
  return "{#{who}}"
  :xxx # Not reached
end
name = :jess
p ["Hugging #{name}", hug.call(name), "Hugged #{name}"]
#=> ["Hugging jess", "{jess}", "Hugged jess"]
```

`break`
: Acts exactly like `return`.

```
hug = ->(who) do
  break "{#{who}}"
  :xxx # Not reached
end
name = :maria
p ["Hugging #{name}", hug.call(name), "Hugged #{name}"]
#=> ["Hugging maria", "{maria}", "Hugged maria"]
```

Control Flow Statements

Control flow statements other than `break` or `return` operate in the same way for both kinds of `Procs`.

`next`
: Returns its arguments to the yield statement or method that invoked the `Proc`.

`redo`
: Jump to the beginning of the `Proc`.

`redo`
: Jump to the beginning of the `Proc`.

`retry`
: Always raises a `LocalJumpError`.

`raise`
: Propagates the exception up the call stack: through any enclosing block, then to the invoking method.

Creation

Proc.new

`Proc.new` creates a `Proc` with `yield` semantics from the given block.

If the block is omitted, the block with which the lexically enclosing method was invoked is used in its place. If the method was not invoked with a block, or there is not an enclosing method, an `ArgumentError` is raised.

proc Keyword

The `proc` keyword is a synonym for Proc.new: it creates a `Proc` with `yield` semantics from the given block. Without a block argument an implicit block is assumed.

& Parameter

A method or lambda whose parameter list includes an identifier prefixed with an ampersand, assigns to the parameter a `Proc` with `yield` semantics representing the block literal that the method/lambda was sent. For more details, see Block Arguments, which includes an example.

lambda keyword

The `lambda` keyword creates a `Proc` with invocation semantics from the given block. Without a block argument an implicit block is assumed.

```
def intermittently wait, closure
  loop { sleep(wait.call) and rv = closure[Time.now] and return rv }
end
require 'prime'
prime_time = lambda do |t|
  t.strftime("%T:%6N") if [t.hour, t.min, t.sec, t.usec].all?(&:prime?)
end
intermittently lambda { Random.new.rand(3.0) }, prime_time #=> "11:29:47:435997"
```

Lambda Literal

A literal of the form `->(`*parameter*$_0$`..`*parameter*$_n$`) { `*statements*` }` instantiates a `Proc` object with invocation semantics. The optional parameter list takes the same form as that used in method definitions. It may be omitted entirely. *statements* is zero or more statements. For example, `-> { 42 }`, or `->(a, b) { a + b }`.

```
require 'resolv'
->(host) do
  Resolv.getaddress host
end.call 'runpaint.org' #=> "208.94.116.80"
resolv = ->(ips) { Hash[ ips.map {|ip| [ip, Resolv.getname(ip)]} ] }
resolv[%w{128.9.160.27 66.80.146.7 65.181.149.201 217.147.82.116}]
#=> {"128.9.160.27"  =>"www.rfc-editor.org", "66.80.146.7"   =>"forced.attrition.org",
#=>  "65.181.149.201"=>"ycombinator.com",    "217.147.82.116"=>"ns.irational.org"}
resolv.('128.30.52.45', '69.13.187.182')
#=> in `call': wrong number of arguments (2 for 1) (ArgumentError)
```

Calling

A `Proc` can be invoked in the following ways:

`Proc#call(`*arg*$_0$`,…,`*arg*$_n$`)`
: Also invoked with the syntax below.

proc`.(`*arg*$_0$`,…,`*arg*$_n$`)`
: A syntactical shortcut for `Proc#call`. The parentheses are mandatory, even if there are no arguments.

`Proc#yield(`*arg*`0,…,`*arg*`n)`
: An instance method with the selector `:yield`; distinct from the `yield` keyword.

`Proc#[`*arg*`0,…,`*arg*`n]`
: The square brackets are mandatory, even if there are no arguments.

`Proc#===` *arg*
: Allows `Procs` to be used in `case` expressions. It requires exactly one argument, so is unsuitable for a `Proc` with invocation semantics that has an arity other than 1.

```
person = ->(age) do
  "#{age}: (" + case age
    when (0..12)  then 'child'
    when (13..19) then 'teenager'
    when (20..59) then 'adult'
    else               'pensioner'
  end + ")"
end
person.call(19) #=> "19: (teenager)"
person.(101)    #=> "101: (pensioner)"
person.yield 8  #=> "8: (child)"
person[10]      #=> "10: (child)"
person === 26   #=> "26: (adult)"
```

Parameters

A `Proc` may be defined with a parameter list, which describes the arguments it accepts. The syntax of the parameter list content mostly mirrors that of method parameter lists, with the following differences:

- It is enclosed within a pair vertical lines (`|`), rather than parentheses, which are mandatory if a parameter list is specified.
- It is specified as the first element of the block associated with the `Proc`: after the opening curly bracket or the `do` keyword.
- If the lambda literal syntax is used, the vertical lines must be omitted and the parameter list must be specified within the parentheses following `->` à la method parameter lists; *not* in the block.

- The parameter list of a closure with `yield` semantics may include a trailing comma after the last parameter. This has no syntactical meaning, but serves to indicate that additional arguments are explicitly ignored.

```
->(a, b=2){ [a,b] }
lambda {|a,b=2| [a,b]}
Proc.new do  |a, b=2|
  [a, b]
end
```

Block-Local Variables

A closure may define *block-local variables*: local variables which are distinct from those with the same name in an outer lexical scope.

Block-local variables are defined in the parameter list after the non-block-local parameters, and before the closing vertical line. They are specified as a comma-separated list of identifiers, with a semicolon preceding the first: `|`*param*$_0$`,…,`*param*$_n$`;`*block-local*$_0$`,…,`*block-local*$_n$`|`. The semicolon is mandatory, even if the list of block-local variables is not preceded by any regular parameters.

In the case of lambda literals, block-local variables are specified in the same manner before the closing parentheses of the parameter list, i.e. `->(`*param*$_0$`,…,`*param*$_n$`;`*block-local*$_0$`,…,`*block-local*$_n$`){}`.

```
# `a` and `b` are normal parameters; `c` and `d` are block-local
Proc.new { |a, b=3.14;c, d| }
->(a, b=3.14;c, d){ }
```

If a variable *v* is defined block-local:

1. If *v* was defined in an outer scope, its value is saved.
2. Within the block *v* is assigned `nil`, then behaves as any other local variable.
3. Upon leaving the block, *v* is assigned the value it had originally in the outer scope.

```
v = :out
1.times do |;v,w| # Define `v` and `w` block-local
  v, w = :in, :in
  p [v, w]
end #=> [:in, :in]
# `v` preserves its value from before the block
v #=> :out
# `w` wasn't defined outside the block, so it still isn't
w #=> NameError: undefined local variable or method `w' for main:Object
```

By contrast, if *v* is not defined as block-local, it retains the value it was assigned inside the block, even after leaving the block scope. However, defining a variable, *w*, inside the block which did not exist in the outer scope, does not define it in the outer scope. In both examples, *w* is undefined upon leaving the block.

```
v = :out
1.times do
  v, w = :in, :in
  p [v, w]
end #=> [:in, :in]
v #=> :in
w #=> NameError
```

Binding

We have already introduced the concept of a binding as a reference to the closure's referencing environment. We have demonstrated that the binding is dynamic, resolving variables referenced within a closure relative to the environment in which it was called. An implication is that these variables must be defined in the closure itself or exist in the closure's environment prior to its creation: they can be modified or re-assigned subsequently, but they must have been assigned.

```
# Create a closure referencing an undefined _array_ variable
size = ->{ array.size }
# Define an _array_ variable in the same scope
array = [1,2,3]
# A NameError is raised because _array_ was not
```

```
# defined in the closure's binding
size.call #=> NameError: undefined local variable or method `array' for main:Object
```

A closure is "self-contained: they contain everything the procedure needs in order to be applied." [Friedman08, pp. 79–82] . Therefore, the binding must also "...hold all the information necessary to execute a method, such as the value of `self`, and the block, if any, that would be invoked by a yield." [Flan08, pp. 202–203] .

A closure's binding is encapsulated by a `Binding` object, which is obtained with `Proc#binding`. It can then be used to execute other code in the same environment with a method such as `eval`.

```
class Context
  @@of = :class
  def self.closure
    ->{ @@of = :method }
  end
end
eval "@@of" #=> uninitialized class variable @@of in Object (NameError)
eval "@@of", Context.closure.binding #=> :class
```

Kernel.binding

`Kernel.binding` returns a `Binding` object representing the referencing environment at the time the method is invoked. That is, it generalises the concept of bindings to any object.

```
class Context
  @@where = :class
  def self.context
    binding
  end
end
eval "@@where", Context.context #=> :class
```

Methods

A closure can be converted to a method with Module#define_method. Likewise, a Method object can be converted to a `Proc` with `Method#to_proc`.

However, `Method` objects are not closures: they do not have access to local variables in their parent scope. "The only binding retained by a `Method` object, therefore, is the value of `self`..." [Flan08, pp. 203–204] .

FLOW

Conditionals

A *conditional* is an expression evaluated as a truthbearer. The conditional is *false* if the expression's value is nil or false; otherwise it is *true*.

```
[0, "Erd\u0151s", :false, true, { key: :door }].all? #=> true

[nil, false].none? #=> true
```

Boolean Logic

The Boolean logic operators return either `true` or `false` by evaluating their operands as conditionals. For this reason, Boolean expressions are often themselves used as conditionals.

There are two forms of each Boolean operator—keyword (e.g. `and`) and punctuation (e.g. `&&`)—which differ only in precedence. The former have low precedence; the latter high.

AND Operator

The binary `and`/`&&` operators return `true` iff both operands are true. They perform short-circuit evaluation, so will only evaluate the second operand if the first is true.

```
class Integer
  def power_of_2?
    nonzero? && (self & (self - 1)).zero?
  end
end
(0..100).select(&:power_of_2?) #=> [1, 2, 4, 8, 16, 32, 64]
```

OR Operator

The binary `or`/`||` operators return `true` iff at least one operand is true. They perform short-circuit evaluation, so will only evaluate the second operand if the first is false.

```
def valid_isbn13?(isbn)
  isbn = isbn.to_s.scan(/\d/).map(&:to_i)
  isbn.size == 13 or return false
  check_digit = 10 - isbn.
                     first(12).
                     map.
                     with_index{|d,i| i.succ.odd? ? d : 3*d}.
                     reduce(:+).
                     modulo(10)
  (check_digit == isbn.last) or (check_digit == 10 and isbn.last.zero?)
end

valid_isbn13?('978-0-596-80948-5') #=> true
valid_isbn13?('9780596809484') #=> false
```

NOT Operator

The unary `not`/`!` operators return `true` iff the operand is false.

```
not true #=> false
not not nil #=> false
!!:false #=> true
3 > 2 and !3.even? and not 3.zero? #=> true
```

Flip Flops

When the `..` or `...` operators are used in a conditional they don't have their usual semantics as range literals; they create a *flip-flop* expression: a stateful, bitstable, Boolean test between two expressions. They take the form *`left..right`* , where *left* and *right* are both Boolean expressions. They are false until *left* is true, at which point they become true, remaining true until *right* is true, at which point they become false until *left* becomes true again. In this way they *flip-flop* between true and false.

When a flip-flop becomes true it tests *right* to determine whether its next state should be false. If *left* and *right* are separated by three consecutive periods instead of two, *right* is not tested until the expression is next evaluated.

Branching

Branching statements predicate the execution of block on a conditional. For example, an `if` statement executes its associated code block iff its conditional is true. Each code block of an branching statement is termed a *branch*, to describe the effect of the statement on program execution.

if Statement

The `if` statement comprises a conditional, zero or more statements (the branch), then, optionally, additional branches whose forms are explained subsequently. The branch is executed iff the conditional is true.

```
if true
  :verdadero
end #=> :verdadero

if false then :falso end #=> nil
```

The return value of an `if` statement is that of the executed branch, or `nil` if no branch was executed.

Postfix Form

A postfix `if` statement is a concise alternative when the branch consists of a sole statement. It comprises an expression, the `if` keyword, then a conditional. The expression is executed iff the conditional is true.

```
# coding: utf-8
balance = 250
puts "You owe £#{balance}" if balance > 0 #=> "You owe £250"
```

This syntax, as opposed to `if` *`conditional`*..., foregrounds the expression. This is of stylistic benefit if the conditional is normally true because it highlights the default case.

else Clause

Prior to an `if` statement's `end` keyword an `else` branch may appear. It is executed iff no preceding branch was executed, serving as the default branch.

```
if :cat > :dogs
  "Meow"
else
  "Woof!"
end #=> "Woof!"
```

elsif Clause

An `if` statement may contain any number of `elsif` branches between the `if` branch and before the `else` branch, if present. To execute they require all prior conditionals to be false and their conditional to be true.

```
# coding: utf-8
class Integer
  require 'prime'

  def square_free?
    prime_division.map(&:last).all?{|p| p == 1}
  end

  def möbius
    return if self < 1 # Postfix if statement
    if    not square_free? then                          0
    elsif prime_division.map(&:first).uniq.size.odd? then -1
    else                                                  1
    end
  end
end
(1..10).map(&:möbius) #=> [1, -1, -1, 0, -1, 1, -1, 0, 0, 1]
```

unless Statement

The `unless` statement executes its branch iff its conditional is false. It is equivalent to an `if` statement with the conditional inverted. It may be followed by an `else` branch, which executes iff the `unless` conditional is true. `elsif` clauses are prohibited.

```
string = '3 free frogs'
unless string =~ /^\d\./
  $stderr.puts "<#{string}>: must start with a digit followed by a period"
end
# <3 free frogs>: must start with a digit followed by a period
```

Postfix Form

The postfix form of the `unless` statement behaves as the postfix `if` statement, except the expression is executed iff the conditional is false.

```
def palindrome?(string)
  raise ArgumentError unless string = String.try_convert(string)
  string = string.scan(/\w/).join.downcase
  string == string.reverse
end

palindrome?("Go hang a salami I'm a lasagna hog.") #=> true
palindrome?("Level, madam, level!") #=> true
palindrome?("canon a 2 cancrizans") #=> false
palindrome?(['mad']) #=> ArgumentError
```

Ternary Operator

The ternary operator is a concise alternative to `if` *`conditional`*…`else`… when the conditional and branches are simple. It consists of three operands, the first of which is the conditional. If the conditional is true, the second expression is evaluated; otherwise the third expression is evaluated.

```
VOWELS = %w{a e i o u}
['d', 'e'].each do |letter|
  puts "#{letter} is a %s" % (VOWELS.include?(letter) ? "vowel" : "consonant")
```

```
end
#=> "d is a consonant"
#=> "e is a vowel"
```

case Statement

The case statement allows a single expression (hereafter: the *topic*) to be tested against other expressions without having to evaluate the topic each time. It begins with a `case` *`topic`* clause, where *topic* is an arbitrary expression.

when Clause

A branch is introduced by the `when` keyword followed by a comma-separated list of one or more expressions. This list is separated from the statements comprising the branch body by the `then` keyword or a statement terminator.

A *when clause* matches the topic if any of the expressions listed after `when` have case equality (*`expression === topic`*) with the topic. Therefore, `when` *`expression`* is equivalent to `if` *`expression === topic`* . By default, the `#===` message is equivalent to `#==`—they both test for equality—but certain core classes redefine `#===` to behave more usefully in this context, as shown in the following table.

Class (*a*)	Semantics of *`a === b`*
`Class`	*`b`*`.instance_of?(`*`a`*`)`
`Proc`	*`a`*`.call(`*`b`*`)`
`Range`	*`a`*`.include?(`*`b`*`)`
`Regexp`	*`a =~ b`*
`Symbol`	*`a == b`*`.to_sym`

The effect of the case equality operator on a receiver of class a and a single operand (b)

else Clause

A single `else` clause may appear between the final `when` clause and the `end` keyword which delimits the `case` statement.

```
class Integer
  def ordinal
    if    to_s =~ /^1\d$/ then 'th'
    elsif to_s =~ /1$/    then 'st'
    elsif to_s =~ /2$/    then 'nd'
    elsif to_s =~ /3$/    then 'rd'
    else                       'th'
    end
  end
end
[1,2,3,4].map{|n| "#{n}#{n.ordinal}"} #=> ["1st", "2nd", "3rd", "4th"]

class Integer
  def ordinal
    case to_s
      when /^1\d$/ then 'th'
      when /1$/    then 'st'
      when /2$/    then 'nd'
      when /3$/    then 'rd'
      else              'th'
    end
  end
end
[1,2,3,4].map{|n| "#{n}#{n.ordinal}"} #=> ["1st", "2nd", "3rd", "4th"]
```

Evaluation

A `case` expression is evaluated by evaluating each `when` branch in the order that they appear in the source file. If a branch matches the topic, it is executed, and its return value becomes that of the `case` statement; otherwise, the next branch is evaluated in the same manner. If none of the `when` branches match, and an `else` branch is present, the `case` statement's return value is that of the `else` branch; otherwise it is `nil`.

```
MAX_REDIRECTS = 5
redirects = 0
[200, 404, 408, 302, 500].map do |status_code|
  case status_code
    when 408, 504  then :timeout
    when 100...200 then :informational
    when 200...300 then :success
    when 300...400
      (redirects += 1) < MAX_REDIRECTS ? :redirection
                                       : (raise "Redirection limit exceeded")
    when 400...500 then :client_error
    when 500...600 then :server_error
    else                raise "Invalid status code: #{status_code}"
  end
end #=> [:success, :client_error, :timeout, :redirection, :server_error]
```

Looping

Looping constructs represent repetition. They comprise a block of code (hereafter: the *body*) and a *terminating condition* which governs the number of times the body will execute.

The following constructs are materially identical in that they can all be used to create any kind of loop, albeit when combined with control flow statements. The purpose of having many different approaches to looping, as opposed to a single, generic construct, is to allow common scenarios to be expressed concisely and simply, and in doing so lessen the chance of erroneous terminating conditions.

Count-Controlled Loops

The constructs that follow instrument an a priori number of repetitions.

Integer#times

`Integer#times` creates a loop which executes *n* times, where *n* is the integer's magnitude. For example, `10.times { … }` executes the block ten

times, passing into the block the number of the current iteration. When the block is omitted an `Enumerator` is returned.

```
def times_tables n
  max_width = (n * n).to_s.size
  n.times do |x|
    n.times do |y|
      printf "%#{max_width}d ", x.succ * y.succ
    end
    puts
  end
end
times_tables 10
#  1   2   3   4   5   6   7   8   9  10
#  2   4   6   8  10  12  14  16  18  20
#  3   6   9  12  15  18  21  24  27  30
#  4   8  12  16  20  24  28  32  36  40
#  5  10  15  20  25  30  35  40  45  50
#  6  12  18  24  30  36  42  48  54  60
#  7  14  21  28  35  42  49  56  63  70
#  8  16  24  32  40  48  56  64  72  80
#  9  18  27  36  45  54  63  72  81  90
# 10  20  30  40  50  60  70  80  90 100
```

Integer#upto

Similarly, `Integer#upto(`*`limit`*`)` counts from the value of the receiver *up* to the value of *limit*, executing the loop body each time. On each iteration the block is passed the current number in the progression. For, example, `10.upto(13)` executes the loop body four times, passing it 10, 11, 12, 13, then terminating. When the block is omitted an `Enumerator` is returned.

```
ten_times = []
10.upto(15) do |i|
  ten_times << i * 10
end #=> 10
ten_times #=> [100, 110, 120, 130, 140, 150]
```

Integer#downto

Conversely, `Integer#downto(`*`limit`*`)` counts from the value of the receiver *down* to the value of *limit*, executing the loop body each time. As with `Integer#upto`, the block is passed the number of the current progression. When the block is omitted an `Enumerator` is returned.

```
10.downto(1) do |count|
  print count, ' '
  puts "Blast off!" if count == 1
end #=> 10
# 10 9 8 7 6 5 4 3 2 1 Blast off!
```

Condition-Controlled Loops

A condition-controlled loop uses a conditional to determine when the loop should terminate. The conditional is tested prior to each repetition, its value determining whether to repeat or terminate the loop.

while Loops

The `while` loop executes its body as long as its conditional is true. If the conditional is initially false the body is never executed.

```
def farey(n)
  a, b, c, d = 0, 1,  1, n
  while c < n
    k = (n + b)/d
    a, b, c, d = c, d, k*c - a, k*d - b
    (sequence ||= []) << Rational(a, b)
  end
  sequence.unshift Rational(0, 1)
end
farey(4) #=> [(0/1), (1/4), (1/3), (1/2), (2/3), (3/4), (1/1)]
```

Postfix Form

`while` can also be used as a statement modifier in which case it executes its left-hand side while its right-hand side is true.

```
# (n.to_s(2).count('1') works just as well, of course)
def bits_set(n)
  bits_set = 0
  n &= n - 1 while n > 0 && bits_set += 1
  bits_set
end
[33, 736, 128].map{|n| bits_set(n)} #=> [2, 4, 1]
```

until Loops

The `until` loop executes its body as long as its conditional is false. If the conditional is initially true the body is never executed.

```
def look_and_say(seed=1, max_terms)
  [seed].tap do |terms|
    until terms.size >= max_terms or terms.last == terms[-2]
      last = terms.last.to_s.split(//)
      term = [[1,last.first]]
      last[1..-1].each do |e|
        term.last.last == e ? term.last[0] += 1 : term << [1, e]
      end
      terms << term.join.to_i
    end
  end
end
look_and_say(7) #=> [1, 11, 21, 1211, 111221, 312211, 13112221]
```

Postfix Form

`until` can also be used as a statement modifier in which case it executes its left-hand side until its right-hand side is true.

```
class Integer
  require 'prime'
  def next_prime
    n = succ
    n += 1 until n.prime?
    n
  end
end
1.upto(10).map {|n| {n => n.next_prime}}
```

```
#=> [{1=>2}, {2=>3},  {3=>5},  {4=>5},  {5=>7},
#    {6=>7}, {7=>11}, {8=>11}, {9=>11}, {10=>11}]
```

Infinite Loops

The `loop` keyword executes its body indefinitely. Unlike the constructs discussed previously, there is no explicit terminating condition. An infinite loop qua infinite loop is undesirable, so the body will typically include statements that conditionally terminate it. Such statements include the control flow statements, `return`, `throw`, `raise`, and `exit`. In addition, raising a `StopIteration` exception inside the loop body has the same effect. The principle of `loop`, therefore, is to repeat an operation until explicitly told to halt.

```
class Integer
  def happy?
    return false unless self > 0
    sad, sequence = [4, 16, 37, 58, 89, 145, 42, 20], [self]
    loop do
      sequence << sequence.last.to_s.split(//).map{|d| d.to_i ** 2}.reduce(:+)
      return true if sequence.last == 1
      return false if sequence.last(sad.size) == sad
    end
  end
end
(1..15).select(&:happy?) #=> [1, 7, 10, 13]
```

Control Flow Statements

break Statement

Within a loop the `break` statement transfers control to the first statement following the loop. (Within a proc or lambda it has different semantics). The value of a `break` statement, and therefore the enclosing loop, is that of its arguments—coerced into an `Array` if there is a plurality—, or `nil` in their absence.

```
(1..20).map{|i| i.to_s(2)}.each do |binary|
  break binary if binary.size > 4 && binary[-1] == '1'
end #=> "10001"
```

next Statement

The `next` statement ends the current iteration of the loop and begins the next. It raises a `LocalJump` exception when used outside of a loop or closure. Any arguments it is given are ignored.

```
ips = Hash.new {|h,k| h[k] = []}
IO.foreach('/etc/hosts').each do |line|
  next if line.start_with?('#')
  ip, *hosts = line.split
  ips[ip] += hosts
end
ips #=> {"127.0.0.1"=>["localhost"], "127.0.1.1"=>["paint", "read-ruby"],
    #    "::1"=>["localhost"]} #...
```

redo Statement

The `redo` statement restarts the current iteration of the loop, returning control to the first expression in the body. The loop conditional is not re-evaluated.

```
loop do
  puts "Password: "
  password = gets.chomp
  unless [/\d/, /[A-Z]/, /[a-z]/].all?{|pat| pat =~ password}
    $stderr.puts "Passwords must contain uppercase, lowercase, and digits"
    redo
  end
  unless password.size > 8
    $stderr.puts "Passwords must be at least 9 characters long"
    redo
  end
  puts "Thanks. Confirm password: "
  unless password == gets.chomp
    $stderr.puts "The passwords entered do not match!"
```

```
    redo
  end
  break
end
```

throw/catch Statements

The `catch` method defines a labelled block of code. The `throw` method exits a `catch` block with a given label. Taken together, they form a general-purpose control structure.

The label is normally given as a `Symbol` or `String`, however any object is permissible as long as the thrown object is *identical* to the caught object.

A `catch` block is defined with `catch` *`label block`*, where *label* is the label which this block should catch, and *block* is a block literal. The block is called immediately.

The syntax of `throw` is `throw` *`label`*, *`value`*, where *label* is the label of an enclosing `catch` block, and *value* is an optional expression which, if supplied, becomes the value of the corresponding `catch` block.

`throw` immediately causes the current execution path to terminate and a search to begin for the nearest `catch` block defined with the same label. The search proceeds out through the current lexical scope, then up through the call stack towards the top-level of the program, crossing method boundaries if necessary. If such a `catch` is found, it is exited, and execution resumes from the statement that follows it. Otherwise, if the search is unsuccessful, an `ArgumentError` exception is raised.

```
require 'net/http'

def follow(url)
  chain = []
  while url do
    url = catch(:redirect) do
      chain << url = URI.parse(url)
      resp = Net::HTTP.new(url.host).get url.path
      case resp.code.to_i
        when 200       then return chain.join(' => ')
```

```
        when 300...400 then throw :redirect, resp['location']
        else raise "#{url} => #{resp.code}: #{resp.message}"
      end
    end
  end
end
follow('http://w3c.org/html')
#=> "http://w3c.org/html => http://www.w3.org/html => http://www.w3.org/html/"
```

yield Statement

The `yield` statement causes control to be transferred to the associated block, then resume from the statement following `yield`. It effects a *temporary* transfer such that the yielding method retains control; whereas `return` cedes complete control of execution to its caller. This enables a method to `yield` to a block multiple times, as opposed to `return` which, by definition, may be executed by a method no more than once.

As explained in Block Literals, a block literal may be sent along with a message. Then, as explained in Block Arguments, the receiving method may access this block, which we will refer to as *the associated block*.

The return value of `yield` is that of the block yielded to.

Arguments

`yield` may be supplied with one or more arguments, which are passed to the associated block. It is this ability that enables internal iterators to be constructed.

`yield`'s arguments are married with the block's parameter list with *yield semantics*: semantics similar to those of methods, with the following differences:

- Superfluous arguments are silently discarded. A block wishing to collect a variable number of arguments into an `Array` must use a rest parameter.
- Parameters representing omitted arguments are assigned `nil`.
- `Array` arguments are automatically exploded.

- Block literals are prohibited as arguments. That is, a block cannot be yielded to another block. The workaround is to yield a `Proc` object representing the block to be yielded.

As with message expressions, the final argument to `yield` may be a `Hash` literal with or without its delimiting curly braces.

Iterators

Internal

An internal iterator is a method which accepts a block to which each element of a collection is yielded in turn. These are *internal*> iterators because they "push" elements of the collection to the block; whereas *external*> iterators, discussed below, are objects from which the elements are "pulled".

In fact, `Integer#times`, `Integer#upto`, and `Integer#downto`, which were introduced above, are themselves internal iterators. The collection over which they iterate is a sequence of integers.

The most common internal iterator is `#each` because its presence allows a collection class to mix-in the `Enumerable` module. It is defined on all core classes that can sensibly be iterated over, with the provisos noted in `#each`.

An object supporting internal iteration is sent the appropriate selector (e.g. `:each`) along with a block literal expecting at least one argument. Each time the block is called its argument will be the next element of the collection, where the semantics of *next* depend on the iterator's intent. The block is a closure, so flow control statements can be used in its body in the manner described in Closures.

`:each`, along with most internal iterators, can also be sent without a block in which case it returns an `Enumerator`.

Internal iterators are not limited to behaving like `:each`; Enumerables describes some of the specialised iterators the `Enumerable` module defines in terms of `:each`.

for

Objects responding to `:each` in the manner outlined above may also be iterated over with the `for` keyword as shown below. The only practical difference between a `for` loop and using `:each` directly is that the latter defines a new variable scope for the duration of the block.

```
string = ''
for element in 97..100
  string << element if element.odd?
end
string #=> "ac"
```

Custom Internal Iterators

Fundamentally, an internal iterator is just a method which yields a value to a block. The mechanics of methods accepting blocks are described in Block Arguments.

```
def intermittently(array)
  array.each do |element|
    sleep rand(10) * (element.size % array.size)
    yield element
  end
end

intermittently([191, 2726, 278, 12**10, 182729]) do |element|
  puts Time.now.strftime("%H:%M:%S") + " => #{element}"
end
# 00:29:33 => 191
# 00:29:33 => 2726
# 00:29:45 => 278
# 00:30:06 => 61917364224
# 00:30:34 => 182729
```

Begin / Exit Handlers

A *begin* or *exit* handler registers code to be executed at the beginning or end, respectively, of a program.

BEGIN Block

A `BEGIN` block, as opposed to a `begin` statement, is executed at the very beginning of a program. It consists of the `BEGIN` keyword at the top-level of a program, followed by statements delimited by curly braces, and defines its own variable scope. If multiple `BEGIN` blocks are present, they are executed in the order encountered by the interpreter.

```
puts 1
BEGIN { puts 2 }
puts 3
# Output:
# 2
# 1
# 3
```

END Block

An `END` block, as opposed to the `end` keyword, is executed at the very end of a program. They consist of the `END` keyword followed by statements delimited by curly braces. Unlike `BEGIN` blocks, they share local variable scope with surrounding code. A further difference is that the execution of an `END` block is governed by surrounding constructs: if it appears in the body of a conditional, for instance, its execution is dependent on that conditional being true. However, even if enclosed in a looping construct, an `END` block is still only executed once.

```
puts 1
END { puts 2 }
puts 3
# Output:
# 1
```

```
# 3
# 2
```

Kernel.at_exit

An alternative to `END` blocks is `Kernel.at_exit`. It accepts a block argument which it registers to execute at the end of program execution. If called multiple times the blocks are executed in reverse chronological order.

```
at_exit { puts 1 }
puts 2
at_exit { puts 3 }
at_exit { puts 4 } if false
# Output:
# 2
# 3
# 1
```

EXCEPTIONS

An *exception* is an object representing an abnormal condition that changes the typical flow of execution. A block may initiate, or *raise*, an exception to signal an abnormality which it is either unwilling or unable to handle locally. This suspends program execution and causes a corresponding *exception handler*—a block which has elected to resolve the abnormality—to be sought. If the search is successful, the handler is called, then execution resumes from the statement following that which raised the exception. Otherwise, the program terminates.

Exception Objects

Exceptions are instances of `Exception` or a subclass thereof. These subclasses primarily exist to increase granularity in `rescue` clauses, which match exceptions solely on their class. They rarely augment the behaviour of their parent.

Most core exceptions subclass `StandardError`, allowing them to rescued by a bare `rescue` clause. The remainder are not expected to be rescued in the normal course of a program, so must be mentioned explicitly if they are to be rescued. The principle, therefore, is that custom exception classes should inherit from `StandardError` if they are generally recoverable; and `Exception` otherwise.

A custom exception is created by subclassing an existing `Exception` class. Idiomatically this is: `CustomError = Class.new(ExceptionClass)`, where *CustomError* is the name of the new exception class and *ExceptionClass* is that of an existing exception class.

Exception objects are generally created implicitly with `raise`. An `Exception` class can be instantiated manually with its `.new` constructor, which accepts an optional `String` argument with which to set the exception's message.

Message

An exception's *message* is a human-readable `String` describing the nature of the encapsulated error. It is intended primarily for debugging purposes.

It is retrieved with the exception's `#message` accessor. It can be set with an argument to `raise` or the constructor of the exception class. The default message is the name of the exception's class.

Backtrace

The *backtrace* of an exception describes the call stack at the moment the exception was raised. It is represented as an `Array` which details the call stack in reverse chronological order: the first element describing the line where the exception was raised, the second, the caller of the method where the exception was raised, the third, the caller of that method, etc. Each element of the `Array` is a `String` detailing the filename and line number of the event.

An exception's backtrace is set automatically by `raise` using `Kernel.caller`. It can be created manually by supplying an appropriate `Array` as the third argument to `raise`, or `Exception#set_backtrace`.

The `Exception#backtrace` accessor returns the backtrace of the receiver.

raise

An exception is raised with the `Kernel.raise` method, or its alias `Kernel.fail`.

When `raise` is called without arguments it raises a `RuntimeError`. If called thusly inside a `rescue` clause, it re-raises the current exception.

If the sole argument to `raise` is an `Exception` object, that exception is raised. When the sole argument is a `String`, a `RuntimeError` is raised with that string as its message.

The first argument of `raise` can be an object responding to `:exception`, in which case the exception raised is the `Exception` object returned by that method. The `Exception` class itself defines such a method, allowing any of its subclasses to be used in this manner.

```
raise TypeError
#=> #<TypeError: TypeError>

raise "Quantum entanglement failed"
#=> <RuntimeError: Quantum entanglement failed>

obj = Object.new
def obj.exception
  NameError.new("We're supposed to be anonymous!")
end
raise obj
#=> #<NameError: We're supposed to be anonymous!>
```

`raise` accepts a `String` as an optional second argument, which becomes the exception's message.

The exception's backtrace may be set explicitly by providing an `Array` of `Strings` as `raise`'s third argument.

```
begin
  raise SystemExit.new(1), "Abnormal exit"
rescue SystemExit => e
  "Caught exit with status #{e.status}: #{e.message}"
end #=> "Caught exit with status 1: Abnormal exit"

def move
  raise LocalJumpError, "Try hopping or skipping", ["Buy Thriller DVD",
                                                    "[redacted]",
                                                    "Rise up, legs akimbo"]
rescue => e
  return "#{e.class}: <#{e.message}> from <#{e.backtrace.first}>??"
end

move
#=> "LocalJumpError: <Try hopping or skipping> from <Buy Thriller DVD>??"
```

Arguments *a*	Raises *b*	*c*	
			`RuntimeError` or current exception
`Exception`			*a*
`Exception`	`String`		*a* with *b* as the message
`Exception`	`String`	`Array`	*a* with *b* as the message, *c* as the backtrace
`String`			`RuntimeError` with *a* as the message
`respond_to?(:exception)`			`a.exception`
`respond_to?(:exception)`	`String`		`a.exception` with *b* as the message
`respond_to?(:exception)`	`String`	`Array`	`a.exception` with *b* as the message, *c* as the backtrace

A summary of the arguments accepted by `raise`

Propagation

Once raised, exceptions propagate "outward and upward" [Flan08, pp. 160–161] toward the first matching exception handler. Each block encountered by the search is examined for the presence of a `rescue` clause that matches the class of the current exception. If one is found, the exception is handled, and the search halted.

The outward propagation is the passage from the `raise` statement to the lexically enclosing block, to that block's lexically enclosing block, until arriving at the top-level of the program. If the entire program is contained in a single file this step can be visualised as the movement from `raise` toward the left margin.

If the outward search completes without the current exception being handled, the search continues up the call stack: to the caller of the current block, to the caller of the current block's caller, and so on until the call stack is exhausted. If at this point the exception has not been handled, it is printed to STDERR and the program is aborted.

Handling

An exception is handled with a rescue statement modifier or by attaching a rescue clause to a begin statement, a method definition, a class definition, or a module definition.

begin Statements

rescue is typically used in conjunction with a begin statement, which simply associates a block of statements with one or more rescue clauses. The rescue clauses may be followed by optional else and ensure clauses.

rescue

A rescue clause handles the exceptions described by its arguments, then returns control to the statement following the initiating raise.

Without arguments, rescue handles only StandardError exceptions and subclasses thereof. rescue may be followed by => *identifier* to assign the rescued exception to a local variable named *identifier*. Having done so, Exception instance methods such as #message and #class may be sent to it.

```
begin
  # 1. Raises a RuntimeError with 'Disaster' as the message
  raise "Disaster"
# RuntimeError is a subclass of StandardError, so this clause matches
# the exception raised above and assigns it to local variable e
rescue => e
  e.message
end #=> 'Disaster'
```

If arguments are provided to rescue they must be supplied as a comma-separated list of Exception classes to be handled. A rescue clause so written will handle exceptions of any of the named classes or their subclasses.

A construct that accepts a `rescue` clause accepts any number of them. They are tested against the current exception in the order that they appear in the source file, so should be arranged by decreasing specificity.

Postfix Form

`rescue` may be also used as a statement modifier. In this form it is not restricted to the aforementioned constructs; it may appear anywhere a statement can otherwise. Its syntax is *a* `rescue` *b*, where *a* and *b* are both expressions. It evaluates to *a* by default; *b* if *a* raises an exception of `StandardError` or subclass thereof.

Unlike the `rescue` clause, the `rescue` modifier does not permit specifying exception classes to match or a local variable to which the caught exception is aliased. However, regarding the last point, if the right-hand operand of `rescue` is the special variable `$!`, the statement will evaluate to the rescued exception.

```
[2, 0].map{|denominator| 1/denominator rescue nil}
#=> [0, nil]
```

$!

Within a `rescue` clause or on the right-hand side of a `rescue` statement modifier, the `$!` variable holds the caught exception. In other contexts it evaluates to `nil`.

else Clause

A group of `rescue` statements may be followed by an `else` clause, whose body is executed iff neither the `rescue` clauses handled an exception nor the preceding statements raised one. However, even in this case the use of flow control statements in the statements preceding the `rescue` clauses may cause the `else` clause to be skipped.

ensure Clause

A construct that accepts `rescue` clauses may include an `ensure` clause as its final clause. The statements in an `ensure` clause will always be executed, regardless of whether their previous sibling clauses raised an exception or employed a flow control statement such as `return`. Specifically, if the construct to which `ensure` is attached…

Exits normally
: The `else` clause, if any, is executed, followed by the `ensure` clause.

Executes `return`
: The `else` clause, if any, is skipped, and the `ensure` clause is executed.

Raises an exception
: A matching `rescue` clause, if any, is executed, followed by the `ensure` clause.

The return value of a construct containing `ensure` is that of the previously executed clause, unless the `ensure` clause explicitly returns a value with `return`. That is, unlike other constructs the return value of `ensure` is not necessarily its last statement executed. This is to allow `ensure` clauses to be attached to `def` statements, for example; otherwise the method being defined would never be able to return a value.

An `ensure` clause may cancel the propagation of an exception by raising an exception of its own or executing a control flow statement. This new transfer of control—be it to the end of the current block via `break` or `next`, the return of the current method via `return`, or the nearest matching `rescue` clause via `raise`—replaces that of the current exception, aborting its passage.

```
require 'yaml'
YAML_FILE = 'tasks.yaml'
begin
  tasks = ['Water plants', 'Plant water']
  # ...
  tasks << 'Practice pentatonic scales'
  raise RuntimeError unless tasks.include?('Run 5 miles')
rescue RuntimeError
```

```
  tasks << 'Run 5 miles'
ensure
  File.open(YAML_FILE,'w') {|f| YAML.dump(tasks, f)}
end
YAML.load(File.read YAML_FILE) #=> ["Water plants", "Plant water",
#                     "Practice pentatonic scales", "Run 5 miles"]
```

Class Hierarchy

`Exception`

- `NoMemoryError`
  The Ruby interpreter failed to allocate memory.

- `ScriptError`
  - `LoadError`
    Raised by `Kernel.require`, `Kernel.load`, and `Kernel.require_relative` when the named file cannot be opened; by `require_relative` when called from inside `eval` or `irb`; and by the `ruby` binary when a script is expected but no supplied.

  - `NotImplementedError`
    Raised by methods which rely on operating system functions that are not available to the current script. For example, methods depending on the `fsync()` or `fork()` system calls may raise this exception if the underlying operating system or Ruby runtime does not support them.

  - `SyntaxError`
    Raised by `require`, `require_relative`, `load`, and `eval` if the script they are interpreting is syntactically invalid.

- `SecurityError`
  Raised by methods that are prohibited by the current safe level.

`SignalException`

`Interrupt`

`SystemExit`

Raised by `exit` and processes which attempt to terminate the current script.

`SystemStackError`

Raised by Ruby when a stack overflow is detected.

`StandardError`

The parent class of most recoverable errors. Caught by a bare `rescue` statement modifier or clause.

`ArgumentError`

Raised by Ruby when a message is sent with an unexpected number of arguments. Often raised by methods themselves for similar reasons.

`EncodingError`

`Encoding::CompatibilityError`

Raised by `Encoding` and `String` methods when the source encoding is incompatiable with the target encoding.

`Encoding::ConverterNotFoundError`

Raised by transcoding methods when a named encoding does not correspond with a known converter.

`Encoding::UndefinedConversionError`

Raised by `Encoding` and `String` methods when a transcoding operation fails.

`Encoding::InvalidByteSequenceError`

Raised by `Encoding` and `String` methods when the string being transcoded contains a byte invalid for the either the source or target encoding.

`FiberError`

Raised by fibers when attempting to call/resume a dead fiber, attempting to yield from the root fiber, or calling a fiber across threads.

`IOError`
: Raised by methods performing input/output operations. For example, it is raised when attempting to read from an `IO` stream not opened for reading, or when attempting to operate on a closed stream.

  `EOFError`
  : Raised by `IO` methods when attempting to read past the end of a file.

`IndexError`
: Raised by some `Array` and `String` methods when attempting to access a subscript that is out of bounds; by some `Regexp` methods when attempting to access a capture group by an invalid name/subscript.

  `KeyError`
  : Raised by certain `Hash` methods when attempting to access a value with an undefined key.

  `StopIteration`
  : Raised by `Enumerators` when attempting to access an element past the end of the sequence. Caught by `loop`, causing the loop to terminate.

`LocalJumpError`
: Raised when attempting to return from a method that has already has already returned, call a `Proc` that has already returned, executing `retry` in a `Proc`, executing `next` outside of a loop or block, and by many methods which were not supplied with a required block argument.

`NameError`
: Raised when Ruby sees an identifier whose name neither corresponds to a local variable or method anywhere other than the left-hand side of the assignment operator; when an uninitialised constant is seen; when a class or instance variable is declared with an illegal name; and by `Object#method`, `Object#instance_method`, etc., when attempting to retrieve a `Method` object for a non-existent method.

`NoMethodError`
: Raised by Ruby when an attempt is made to invoke a non-existent method. `NameError` takes preference when an expression is ambiguous as to whether it represents a local variable reference or method call. For example, if there is neither method nor variable named `m`, the expression `m` would result in a `NameError`; whereas the expression *m()* would cause a `NoMethodError`. Responds to `#args` with an `Array` of the arguments sent to the method, and to `#name` with the method's name.

`RangeError`
: Raised by methods receiving numeric arguments that fall outside of their operating limits. For example, `Array#combination` and `Array#product` raise this exception when their arguments are too large to permute.

  `FloatDomainError`
  : Raised by many mathematical methods which receive NaN or Infinity (∞) as arguments.

`RegexpError`
: Raised by `Regexp.new` when passed a syntactically invalid regular expression, and when a `Regexp` literal includes characters invalid in its encoding.

`RuntimeError`
: The default exception instantiated by `raise`, so quite generic in usage. Core classes often raise this exception when a method has receieved valid arguments yet is unable to complete. Also, this is raised when attempting to modify frozen objects or modify an object whilst iterating it.

`SystemCallError`
: Raised when an attempt to call an operating system function fails, despite that function being implemented and expected to work. For example, it is raised by `Dir` when unable to create a directory, `exec` when unable to execute a command, and `File.link` when unable to create a symlink.

`Errno::*`
: There are many exception classes in the `Errno` namespace which subclass `SystemCallError`. They represent low-level, operating-system-specific exceptions, and as such it is often sufficient to rescue `SystemCallError` and ignore their specifics.

`ThreadError`
: Raised by `Thread` when a thread cannot be initialised, moved to another group, manipulated on account of being dead, or when a shared mutex is unexpectedly locked.

`TypeError`
: Common exception raised by methods who receive an argument of an unexpected type. For instance, it is raised by `Range` on attempts to iterate from `Floats`, `class` and `module` when the given constant is defined and not a `Class`, and by various classes unable to convert their argument to a specific type.

`ZeroDivisionError`
: Raised by mathematical operators when they are asked, explicitly or implicitly, to divided a number by `0`. Some methods raise this exception when a division by the `Float` `0.0`, while `Integer#/`, notably, returns `Float::INFINITY`.

CONCURRENCY

Most of the programs we have considered so far have been synchronous: they execute sequentially, from start to finish. `Kernel.fork` stood out as it allows an arbitrary block of code to be executed as a separate, concurrent process. This chapter explores two other techniques for writing concurrent[1] programs.

Threads

A program typically runs along a single thread of execution: each statement is executed sequentially and predictably. Programs that use only this thread—termed the *main thread*—are *single-threaded*; by contrast, a *multithreaded* program has multiple threads of execution, each of which is associated with a block of code. The operating system rapidly switches between these threads—interleaving their execution—to create the appearance[2] of parallelism.

Threads are represented as instances of the `Thread` class. `Thread.current` and `Thread.main` return `Thread` objects corresponding to the current and main threads, respectively.

Initialisation

Threads are created by passing a block to `Thread.new` which creates a new thread to run the block then returns immediately. Any arguments given are passed to the block as block paramaters. `Thread.start`, and its alias `Thread.fork`, behave identically except if their receiver is a subclass of `Thread` its `initialize` method is not called.

---

1. Or, at least, *seemingly* concurrent…
2. Sadly, Ruby does not actually execute threads concurrently even on multi-core CPUs because some extension libraries are not thread-safe: only one thread runs at any given time.

Termination

A given thread can be terminated with `Thread#kill`—which is aliased to `Thread#exit` and `Thread#terminate`. Similarly, `Thread.kill(`*thread*`)`, where *thread* is a `Thread` object, terminates the given thread. `Thread#kill!`—which is aliased to `Thread#exit!` and `Thread#terminate!`—behaves like `Thread#kill` except it bypasses any `ensure` clauses in the receiver. `Thread.exit` terminates the current thread, which it returns if already marked to be killed. Ruby exits when its main thread is terminated.

Other than `Thread.exit`, however, these methods have a significant flaw: they may terminate a thread at *any* point, potentially leaving resources in inconsistent states. Accordingly, JRuby's Charles Nutter states "there is no safe way to use `Thread#kill` or `Thread#raise`" [Nutter08] .

Status

A thread is always in one of five possible states—*runnable*, *sleeping*, *aborting*, *terminated normally*, or *terminated exceptionally*—which may be queried as shown below.

State	`Thread#status`	`Thread#alive?`	`Thread#stop?`
Runnable	`"run"`	`true`	`false`
Sleeping	`"sleep"`	`true`	`true`
Aborting	`"aborting"`	`false`	`true`
Terminated normally	`false`	`false`	`true`
Terminated exceptionally	`nil`	`false`	`true`

When a thread is created it is *runnable*. It enters the *sleeping* state while `Kernel.sleep`[3] or an `IO` method is blocking, after which it returns to *runnable*. A thread that calls `Thread.stop` also switches to *sleeping*, where it remains until awoken, then schedules execution of another thread. A *sleeping* thread may be made *runnable* with `Thread#wakeup` or `Thread#run`, unless it is sleeping due to blocking I/O. `Thread#run` then invokes the thread scheduler, possibly causing the thread to begin running; `Thread#wakeup` does not.

3. If `Kernel.sleep` is called without an argument, it blocks until the thread is awoken or terminated.

A thread that raises an exception, which it does not subsequently rescue, enters the *terminated exceptionally* state. Otherwise, after a thread has executed its final statement its state is *terminated normally*. When a thread is terminated explicitly, its state becomes *aborting*. Unless it was terminated with `Thread#kill!`—or an alias thereof—it then executes its `ensure` clause, the contents of which may cause the thread to change state again. Finally, when the thread actually terminates, its state will be *terminated normally*.

Variables

Threads are created with blocks, so standard scoping rules apply: they may access any variable in the scope of this block, and local variables that they define are not shared with other threads. An implication is that if a thread accesses a variable defined in its parent scope, it will share this variable with all other threads created in the same scope. We have already seen how this behaviour can be avoided by providing arguments to `Thread.new` to create block-local variables.

Some of the predefined global variables are *thread-local*—see the Predefined Global Variables table for a list—meaning that a thread has its own, private copy. An example is `$SAFE`, allowing potentially insecure code to be run in a thread with an elevated safe level. The safe level of a given thread can also be queried with `Thread#safe_level`.

Only Ruby can create thread-local global variables, but any thread may create *thread keys* [Black09, pp. 432–435] : thread-local variables, accessible to other threads via a `Hash`-like interface. `Thread#[`*`key`*`]=`*`value`* creates a thread key named *key*—which must be a `String` or `Symbol`—to *value*. `Thread#[`*`key`*`]` returns the value associated with *key*, or `nil` if there is none. The `Thread#key?(`*`key`*`)` predicate returns `true` if the receiver has defined a thread key named *key*; `false`, otherwise. The names of all keys defined by a given thread are returned by `Thread#keys` as an `Array` of `Symbols`.

Joining

Ruby runs until the main thread terminates, even if additional threads were created and are still running. To wait for a specific thread to terminate,

blocking until it does so, `Thread#value` may be used. It returns the value of the last statement executed by the thread. `Thread#join(`*seconds*`)` also blocks until its receiver terminates, but if more than *seconds* seconds elapse it gives up and returns `nil`; on success it returns the receiver. If *seconds* is omitted, there is no time limit.

Exceptions

A thread can raise an exception with the `raise` keyword, like any other piece of Ruby. `Thread#raise` accepts the same arguments as `raise`, but raises the exception in the thread represented by the receiver. However, see the warning in Termination regarding this method.

When an unhandled exception occurs in the main thread, Ruby prints the backtrace then exits. When another thread raises, but does not handle, an exception, its behaviour depends on an *abort on exception* flag.

If *abort on exception* is `false`, as it is by default, an unhandled exception causes a thread to terminate silently; when waited on—with either `Thread#join` or `Thread#value`—the exception is raised in the calling thread. If *abort on exception* is `true`, all threads behave like the main thread when encountering unhandled exceptions: they print its backtrace then cause the interpreter to exit.

The flag may be set globally with `Thread.abort_on_exception=` or per-thread with `Thread#abort_on_exception=`. These values may be retrieved with `Thread.abort_on_exception` and `Thread#abort_on_exception`, respectively. If `$DEBUG` is `true`—as it is when the interpreter is given the `-d` flag—threads behave as if `Thread.abort_on_exception` is also `true`.

Scheduling

Threads initially have a priority of zero, with a higher priority implying more favourable scheduling. `Thread#priority` returns the receiver's priority as an `Integer`; `Thread#priority=(`*priority*`)` sets the receiver's priority to *priority*. However, priorities are merely hints to the thread scheduler: they

may be entirely disregarded, as they are under Linux for non-privileged users.

There are, broadly, two approaches for allocating CPU cycles to threads: cooperative multitasking and preemptive multitasking. In the former, a context switch occurs when a thread explicitly yields control back to the CPU, or performs a blocking operation such as I/O. In the latter, each thread is run only for a certain amount of time before being interrupted so that another thread may run instead. Under a cooperative approach, a thread which neither yields nor performs blocking operations may *starve* the process's other threads from executing, monopolising the CPU. As a workaround, compute-bound threads may use `Thread.pass` to explicitly yield to the thread scheduler.

Groups

Ruby maintains an `Array` named `Thread.list` of all runnable or sleeping threads. It will always contain at least one entry: the main thread. When a thread terminates, it is removed from this list.

A thread is also a member of exactly one *thread group*, which provides finer-grained control over subsets of threads. The main thread is a member of the `ThreadGroup::Default` group, and new threads are members of their parent's group.

`Thread#group` returns the `ThreadGroup` containing the receiver. The members of a group are returned as an `Array` by `ThreadGroup#list`. A thread may be added to a different group with `ThreadGroup#add`, which first removes the thread from its original group. A group may be *enclosed*, preventing other threads from being explicitly added; however, threads created by existing members of the group will still be made members. `ThreadGroup#enclose` encloses its receiver. The `ThreadGroup#enclosed?` predicate returns `true` if its receiver is enclosed; `false` otherwise. Lastly, a new group may be created with `ThreadGroup.new`.

Synchronisation

When multiple threads need to access a shared, modifiable resource, they must ensure that no thread sees it an inconsistent state. Otherwise, a *race condition*—when a result is dependent on the order in which the threads finish—may occur, making the program non-deterministic.

For a trivial example, consider a thread that invokes `puts "foo"`. `Kernel.puts` first prints `"foo"`, then prints a newline. If another thread is scheduled in between these two operations, additional text could be written to STDOUT before the newline. Indeed, if the second thread invoked `puts "bar"`, the typically result is `"foobar\n\n"`.

Another example is the popular *"too much milk"* problem:

Time	You	Roommate
5:00	Arrive home	
5:05	Discover you've ran out of milk	
5:10	Leave for grocery shop	
5:15		Arrive home
5:20	Arrive at shop	Discover you've ran out of milk
5:25	Buy milk	Leave for grocery shop
5:30	Arrive home, put milk in fridge	
5:35		Arrive at shop
5:40		Buy milk
5:45		Arrive home, put milk in fridge…drat!

The solution, clearly, is that exactly one person—you or your roommate—buys milk. To do so, you must synchronise your actions:

1. If your roommate's left a note telling you that he's gone to get milk, just wait for him to return. Otherwise, leave a note for your roommate: "I'm buying milk", then go to step 2.
2. If there is no milk, go to the shop, buy some, then put the milk in the fridge.

3. Throw away the note you left.

We can generalise this solution to any shared, modifiable resource with a mutual exclusion—abbreviated as *mutex*—lock around the code—termed the *critical section*—which accesses the shared resource. A mutex works like so:

1. Before entering the critical section, request the mutex. If another thread already holds the mutex, wait until it releases it.
2. Perform the critical section: access the shared resource.
3. Release the mutex.

`Thread.exclusive` expects a block constituting a critical section of code, then uses a global mutex to ensure only one thread calls the block at any one time. More generally, a `Mutex` object represents a mutex which can be locked and unlocked independently of any other `Mutex` instance. A `Mutex` is created with `Mutex.new`. `Mutex#synchronize` is given a block representing a critical section, which it executes only if no other thread already holds this specific mutex. Both `Thread.exclusive` and `Mutex#synchronize` return the value of their block.

Alternatively, a lower-level interface is available. `Mutex#lock` obtains a lock with this mutex if possible; it blocks if this mutex is already locked by another thread; and raises a `ThreadError` if this mutex is already locked by the current thread. `Mutex#try_lock` behaves like `#lock`, except it never blocks: if this mutex is not locked, it's locked, and `true` is returned; otherwise, `false` is returned. The `Mutex#locked?` predicate returns `true` if this mutex is locked; `false`, otherwise. If a given mutex was locked by the current thread, it may be unlocked with `Mutex#unlock`. Lastly, `Mutex#sleep(`*`time`*`)` releases the current thread's lock on this mutex, sleeps for *time* seconds—or forever if *time* is `nil`—then locks this mutex. It returns the number of seconds that were slept.

Deadlock is a condition in which two or more threads are waiting for an event that can only be generated by these same threads. It has the following prerequisites, all of which must hold; breaking at least one of them, breaks the deadlock.

Mutual exclusion
: At least one thread must hold a resource that can not be shared. Requests are delayed until this resource is released.

Hold and wait
: A thread holds one resource while it waits for another.

No preemption
: Resources are only released voluntarily after completion; neither another thread nor the OS can force the thread to release the resource.

Circular wait
: Two or more threads form a circular chain where each waits for a resource that the next thread in the chain holds.

Fibers

Execution of blocks and methods always begins from their first statement. Each time, their local variables are initialised anew. If they need to retain state across calls, they must do so explicitly, using either global variables or variables defined in their enclosing scope. A *fiber*—a lightweight, semi-coroutine—provides an alternative approach. It is effectively a block whose execution can be suspended—passing control back to its caller. The caller may subsequently resume the fiber from the point at which it was suspended. A fiber, therefore, automatically maintains state across calls: its local variables are initialised only the first time it is resumed. Only one fiber may execute at any one time, so, like threads, they merely create the illusion of concurrency.

A `Fiber` object is created by passing a block to `Fiber.new`. The block is not called until the fiber is resumed.

Resuming a fiber that has not been resumed previously, executes the block from the beginning. If the block opts to pass control back to its caller, the fiber suspends itself, and execution jumps to the statement following that which resumed the fiber. If this fiber is resumed again, it will continue executing its block from where it left off last time. A fiber may repeat this process as often as it likes.

A fiber is resumed with `Fiber#resume`. It passes control back to its caller with `Fiber.yield`—which has no relation to the `yield` keyword. Any arguments supplied to `#resume` are passed to the fiber: if the fiber had not been resumed previously, they are passed in as block arguments; otherwise, they become the return value of the corresponding `Fiber.yield` invocation. Likewise, any arguments passed to `Fiber.yield` become the return value of the corresponding `Fiber#resume` invocation.

When the block exits, the fiber *dies*. Attempting to resume a dead fiber causes a `FiberError` to be raised. This exception will also be raised if a fiber is created in one thread but resumed from another.

API

NUMERICS

A *numeric*[1] is an object representing a number. It is an instance of one of Ruby's numeric classes:

Integer	Integers, or "counting numbers", e.g. 4.
Float	Floating-point numbers, i.e. numbers with digits after the decimal place, e.g. 3.14
Rational	Rational numbers, or "fractions", e.g. ⅔.
Complex	Complex numbers, i.e. those having both a real and an imaginary part.

Integers

A decimal integer literal consists of an optional `0d` prefix, then one or more decimal digits. If the `0d` prefix is omitted, the first digit must be non-zero. Hexadecimal, octal, and binary literals allow integers to be expressed in base sixteen, eight, or two, respectively. A hexadecimal literal begins with `0x`, and is followed by one or more hex digits (0–9, a–f). An octal literal begins with `0`, an optional `o`, then one or more octal digits (0–7). A binary literal begins with `0b`, and is followed by one or more binary digits (0–1). However, all `Integers` are stored as decimals so `0b10000`, `0x10`, and `020` have the same value: `16`. All forms ignore case and allow an optional sign (`+` or `-`) as their very first character. Consecutive digits may be separated by low lines to aid readability of large numbers, but they have no semantic meaning.

The class of an integer depends upon its magnitude: those representable natively, are `Fixnum` objects; otherwise they are `Bignums`. This distinction is largely irrelevant because Ruby handles the conversion implicitly and both respond identically to the same set of messages. Therefore, we shall refer to `Fixnum` and `Bignum` objects collectively by the name of their parent class, `Integer`. Quantities that exceed the limits of `Bignum`, have the value `Float::INFINITY`.

---

1. The name is due to the parent class of `Integer`, `Float`, `Rational`, and `Complex`: `Numeric`.

Immediates

`Symbol` and `Fixnum` literals are *immediate* values. They are stored as values rather than object references so are immutable and cannot have singleton methods defined on them.

Bases

`Integer` objects represent integers in base ten. They can be converted into another base, *b*, with `Integer#to_s(`*b*`)`, where $36 \geq b \geq 2$. Conversely, a `String` comprising an integer in base *b*, may be converted to the corresponding `Integer` with `String#to_i(`*b*`)`, where *b* has the same limits as before. `String#hex` and `String#oct` are equivalent to `String#to_i 16` and `String#to_i 8`, respectively. When converting an `Integer` into binary, octal, or hexadecimal, format strings provide greater control.

```
67.to_s 2            #=> "1000011"
134.to_s 16          #=> "86"
98765.to_s 30        #=> "3jm5"
'10111011101'.to_i 2 #=> 1501
'beef'.hex           #=> 48879
'%#X' % 36617        #=> "0X8F09"
```

Bit Twiddling

There are a collection of methods that treat `Integers` like bit fields, operating on the number's binary operation. The unary operator `Integer#~` inverts the bit pattern, while `#>>` and `#<<` shift it right or left, respectively, by the number of bits specified as an argument. The binary logical operators `#|`, `#&`, and `#^`, perform bitwise OR, AND or, XOR, respectively.

```
def next_power_of_2 n
  [1,2,4,8,16].inject(n - 1){|memo, x| memo |= memo >> x}.succ
end
[34, 67, 82720, 1024].map{|n| next_power_of_2 n} #=> [64, 128, 131072, 1024]

mask       =->(pos)          { 1 << pos }
set_bit    =->(f, pos)       { f | mask[pos]}
```

```
clear_bit  =->(f, pos)       { f & ~(mask[pos])}
modify_bit =->(f, pos, state){ f & ~mask[pos] | (-state & mask[pos])}
f = 0b1110001
f = set_bit[f, 3]; f.to_s(2)       #=> "1111001"
f = clear_bit[f, 5]; f.to_s(2)     #=> "1011001"
f = modify_bit[f, 0, 0]; f.to_s(2) #=> "1011000"
```

Floats

A `Float` object represents a double-precision floating-point number. It contains a decimal point, so can represent values such as 1.5. A floating-point literal consists of a decimal integer literal, a period, then one or more decimal digits, e.g. `3.14`.

A scientific notation literal consists of a coefficient (*a*: a decimal or `Float` literal), `e` or `E`, then an exponent (*e*: one or more decimal digits). Its value is $a \times 10^e$ represented as a `Float`.

Constants

The precision and limits of `Float` are specified by the following constants:

`DIG` = 15
: Precision in decimal digits

`EPSILON` = 2.220446049250313e-16
: Smallest value such that `Float::EPSILON + 1.0 != 1.0`

`INFINITY`
: Positive infinity: a value too large to be represented.

`MANT_DIG` = 53
: Number of digits in the mantissa.

`MAX` = 1.7976931348623157e308
: Largest possible representable `Float`

`MAX_10_EXP` = **308**
: Largest integer exponent, *x*, such that 10^x is a finite `Float`.

`MAX_EXP` = **1024**
: Largest integer exponent, *x*, such that `Float::RADIX`$^{x-1}$ is a finite `Float`.

`MIN` = **2.2250738585072014e-308**
: Smallest possible representable `Float`.

`MIN_10_EXP` = **-307**
: Smallest integer exponent, *x*, such that 10^x is a finite `Float`.

`MIN_EXP` = **-1021**
: Smallest integer exponent, *x*, such that `Float::RADIX`$^{x-1}$ is a finite `Float`.

`NAN`
: NaN: a value that is undefined.

`RADIX` = **2**
: The base in which the number is stored internally.

`ROUNDS` = **1**
: The rounding mode for floating-point operations:

-1
: Indeterminate

0
: Toward 0

1
: Nearest representable value

2
: Toward `+Float::INFINITY`

3
: Toward `-Float::INFINITY`

Precision & Accuracy

The inexactitude of `Floats` is a frequent source of confusion and illegitimate bug reports. To understand its cause, we must be broadly familiar with how computers store double-precision floating-point numbers.

On most platforms, `Floats` are stored in sixty-four bits. The first bit is the sign, *s*, which is zero for a positive number; one for a negative number. The next eleven are the positive exponent, *e*. To allow for negative exponents, from *e* is subtracted a constant, *b*, which has the value 1023. For example, an exponent of 18 is stored as 10000010001_2. The last fifty-two bits are the mantissa, *m*, which has one implied bit, so it has a precision of fifty-three bits. Therefore, a floating-point number has the following form:

$$(-1)^s \times 2^{e-b} \times 1.m$$

This system of encoding places an upper and lower limit on the range of values that can be represented, as evidenced by `Float::MAX` and `Float::MIN`, respectively.

```
Float::MAX + 1 == Float::MAX
Float::MAX * 2 == Float::INFINITY
Float::MIN - 60_000 == -60_000
```

> Floating point arithmetic adds a new spectrum of errors, all based on the fact that the machine can represent numbers only to a finite precision.
>
> —Kernighan78, pp. 115–116

Arithmetic with integers is exact, unless the result is outside of the range of representable values (underflow or overflow). However, floating-point arithmetic is inherently inexact. Unlike the real numbers, floating-point numbers are not continuous; there are "gaps" between any two numbers. Therefore, a number that cannot be represented exactly-such as an irrational, e.g. π and e, or non-terminating[2] rational, such as ⅓-must be approximated by one of the nearest representable values. Further, because the same number

of bits are used to represent all `Floats`, the smaller the exponent, the greater the density of representable numbers.

> As a wise programmer once said, "Floating point numbers are like sandpiles: every time you move one, you lose a little sand and you pick up a little dirt." And after a few computations, things can get pretty dirty.
>
> —Kernighan78, pp. 117–118

The poster child is 0.1. Regardless of how many bits are available for storage, it can never be represented exactly in base 2 of any finite precision because it would contain the sequence 1100 repeated ad infinitum. Therefore, the value stored is an approximation of the actual value. This becomes apparent when performing even trivial calculations:

```
0.1 * 3        #=> 0.30000000000000004
0.1 * 3 == 0.3 #=> false
```

This is often forgotten because when Ruby displays such fractions-in IRB, for example-it rounds them automatically. To reiterate, this behaviour is an inherent shortcoming of floating-point; it is neither the fault of Ruby nor your hardware.

> One of the first lessons that must be learned about floating point numbers is that tests for exact equality between two computed floating-point numbers are almost certain to fail.
>
> —Kernighan78, pp. 117–118

To conclude, some advice:

- When comparing two `Floats`, either round them beforehand, or compare the absolute value of their difference with an appropriate epsilon.

---

2. …non-terminating, that is, in base 2. Therefore, the only rationals that are representable exactly in binary are those whose denominators are powers of 2, e.g. ¾.

- When performing calculations, consider using `Rationals` rather than `Floats`, where possible, to avoid the unintuitive manner in which round-off errors propagate floating-point operations.
- Take special care when subtracting two values that are almost equal, adding two likewise values that have opposing signs, or performing either addition or subtraction with operands differing significantly in magnitude.

Rationals

A rational number is a number of the form n/d, where both variables are integers. The integer n is the *numerator*, and d the *denominator*. The denominator must not be zero.

Rationals are instances of the `Rational` class. They may be created with the constructor `Rational(`*n*`, `*d*`)`. When *d* is omitted, and *n* an integer, *d* has the implicit value of one. If one or both arguments are themselves `Rational` or `Float` objects, an equivalent rational is found and returned. If `Rational` is, instead, given a `String` representation of a rational, the numerator and denominator are derived from that. In all cases, if the numerator is an exact divisor of the denominator, the `Rational` represents this simplified quotient.

```
Rational(12)                                 #=> (12/1)
Rational(67, 31)                             #=> (67/31)
Rational('7/8')                              #=> (7/8)
Rational(500, 100)                           #=> (5/1)
Rational(3.5)                                #=> (7/2)
Rational(Rational(1, 2), Rational(1, 4)) #=> (2/1)
```

Using `Rationals` instead of `Floats` in calculations enables us to represent rational values exactly, rather than approximating them within the limits of floating-point arithmetic. This solves, for example, the `0.1 * 3` problem described above.

```
0.7 * 0.7                              #=> 0.48999999999999994
r = Rational(7, 10) * Rational(7, 10) #=> (49/100)
r.to_f                                 #=> 0.49
```

A `Rational` may also be created from a `String` or other numeric with `#to_r`. This creates a `Rational` mathematically equal to the receiver. Numerics also provide a method named `#rationalize`. For non-`Floats`, this behaves as `#to_r`; when the receiver is a `Float` it "returns a rational that approximates the float to the accuracy of the underlying floating-point representation." [Lispstd, pp. 12-62–12-62] .[3]. If `#rationalize` is given an argument, it is an epsilon which constrains the result such that:

(*receiver* – *epsilon*) <= *result* <= (*receiver* + *epsilon*)

```
1.7.to_r                     #=> (7656119366529843/4503599627370496)
1.7.to_r.to_f                #=> 1.7
1.7.rationalize              #=> (17/10)
1.7.rationalize.to_f         #=> 1.7
12345.6789.rationalize 0.1   #=> (37037/3)
'9/10'.to_r                  #=> (9/10)
```

The `Numeric#numerator` and `Numeric#denominator` methods return the numerator and denominator, respectively, of the receiver as if it were a `Rational`.

```
[41.numerator, 41.denominator] #=> [41, 1]
r = Rational(136, 153)
[r.numerator, r.denominator]   #=> [8, 9]
f = r.to_f
[f.numerator, f.denominator]   #=> [2001599834386887, 2251799813685248]
r = f.rationalize
[r.numerator, r.denominator]   #=> [8, 9]
```

Complex

A *complex number* is a number of the form $a + bi$, where a and b are real numbers, and i is the imaginary unit. a is termed the *real part* of the complex number, and b its *imaginary part*.

3. The provenance of `#rationalize` appears to be *CLISP*, which implemented the function of the same name from the ANSI Common LISP standard.

They are represented as instances of `Complex`. They can be created with the constructor `Complex(`*a*`, `*b*`)`, where, again, *a* is the real part, and *b* the imaginary. If *b* is omitted, it has the implicit value zero. Alternatively, `Complex()` may be given a sole `String` argument which represents a complex number.

```
Complex(42)                 #=> (42+0i)
Complex(7, 3)               #=> (7+3i)
Complex(Rational(3, 4), 4) #=> ((3/4)+4i)
Complex('3+8.2i')           #=> (3+8.2i)
```

The real part of a `Complex` object may be retrieved with the `Complex#real` accessor, while the imaginary part is available with `Complex#imaginary` or the synonymous `Complex#imag`.

Conjugation

> *conjugate complex numbers* Two numbers of type *a* + *bi* and *a* – *bi*, where *a* and *b* are real numbers.
>
> —James92, pp. 71–71

The conjugate of a complex number is its real part minus its imaginary part. It is itself a complex number. `Complex#conj`, or its alias `Complex#conjugate`, return the conjugate of the receiver as a `Complex`. The `#conj` and `#conjugate` methods of the other numerics return `self`.

```
Complex(13, Rational(1, 2)).conj #=> (13-(1/2)*i)
Complex(7, 6).conj.conj          #=> (7+6i)
```

Arg Function

> *amplitude of a complex number* The angle that the vector representing the complex number makes with the positive horizontal axis.
>
> —James92, pp. 11–11

The *arg*-also known as *angle*, *amplitude*, or *phase*-of a `Numeric` is computed with `#arg`, and its aliases `#angle` and `#phase`. For a non-`Complex` receiver, it is `0` if the receiver is positive; `Math::PI` otherwise.

The `Complex` implementations of these methods: return `Math::PI` if the receiver has a real part less than zero, and no imaginary part; but raise a `Math::DomainError` if both parts are zero. Otherwise, they compute the arg with the following formula:

```
2 * Math.atan(Rational(b, Math.sqrt(a**2 + b**2) + a))
```

```
(-1.7).arg            #=> 3.141592653589793
Rational(23, 32).arg  #=> 0
Complex(20, 9).arg    #=> 0.4228539261329407
Complex(-17, 3.5).arg #=> 2.938547436336328
```

Absolute Value

> *modulus of a complex number* The numerical length of the vector representing the complex number... The modulus of a complex number $a + bi$ is $\sqrt{(a^2 + b^2)}$, written $|a + bi|$. If the number is in the form $r(\cos \beta + i \sin \beta)$ with $r \geq 0$, the modulus is r.
>
> —James92, pp. 276–276

The *absolute* value-also known as the *modulus* or *magnitude*-of a `Complex` number is given by `#abs`, and its alias `#magnitude`, as a `Float`. For all other numerics, these methods return the receiver sans sign.

```
(-1.7).abs              #=> 1.7
Rational(23, 32).abs    #=> (23/32)
Complex(20, 9).abs      #=> 21.93171219946131
Complex(-17, 3.5).abs   #=> 17.356554957709783
```

Polar Form

> *polar form of a complex number* The form a complex number takes when it is expressed in polar coordinates. This form is $r(\cos \theta + i \sin \theta)$, where r and θ are polar coordinates of the point represented by

> the complex number. The number r is the *modulus* and the angle θ the *amplitude*, *angle*, or *phase*.
>
> —James92, pp. 321–321

`Numeric#polar` returns the receiver in *polar form*, that is an `Array` comprising its absolute value and its arg. Conversely, a `Complex` may be created from a given polar form with `Complex.polar(`*abs*`, `*arg*`=0)`.

```
(-1.7).polar                       #=> [1.7, 3.141592653589793]
Rational(23, 32).polar             #=> [(23/32), 0]
Complex(20, 9).polar               #=> [21.93171219946131, 0.4228539261329407]
Complex(-17, 3.5).polar            #=> [17.356554957709783, 2.938547436336328]
Complex.polar(*Complex(2,3).polar) #=> (2.0+3.0i)
Complex(-1).polar                  #=> [1, 3.141592653589793]
```

Rectangular Form

`Complex#rect`, and its alias `Complex#rectangular`, return a two-element `Array`, the first value of which is its receiver's real part; the second, its imaginary part. For all other numerics, these methods return `[self, 0]`.

The `Complex.rect` constructor, and its alias `Complex.rectangular`, create `Complex` objects from their rectangular forms. `Complex.rect(`*a*`, `*b*`)` is equivalent to `Complex(`*a*`, `*b*`)`. As before, if *b* is omitted it has the value `0`.

```
Complex(23.4, 5.6).rect #=> [23.4, 5.6]
35.to_c.rectangular     #=> [35, 0]
Math::PI.rect           #=> [3.141592653589793, 0]
Complex.rect(9, 7)      #=> (9+7i)
```

Basic Arithmetic

The arithmetic operations of addition, subtraction, multiplication, and division are performed with the binary operators `+`, `-`, `*`, and `/`, respectively. If the two operands are of different classes, Ruby will attempt to coerce them so they are not. The result of these operations will have the same class as the operands, or be one of the special values: `Float::INFINITY` or `Float::NAN`.

```
1 + 2                                       #=> 3
34.5 - 30                                   #=> 4.5
Rational(5,6) * 0.5                         #=> 0.4166666666666667
80 / 5                                      #=> 16
(1 + ((3.0 - 1) * 2)) * 7                   #=> 35.0
Complex(Rational(3, 4), 3) + Complex(4, 5) #=> ((19/4)+8i)
```

`Integer#/` performs integer division, i.e. it returns the integer quotient, discarding any decimal part. If this is undesirable, either convert the divisor to a `Numeric` other than `Integer`, such as `Float`, or use a more forgiving method, such as `Numeric#quo` or `Numeric#fdiv`. Both divide their receiver by their argument, returning the quotient: the former, in the most accurate form possible; the latter, as a `Float`.

```
5 / 2                  #=> 2
5.fdiv 2               #=> 2.5
5.quo 2                #=> (5/2)
33.quo 3.3             #=> 10.0
Rational(100, 3) / 0.3 #=> 111.11111111111113
Rational(100, 3) / 3   #=> (100/9)
```

`Integer#/` and `Numeric#quo` raise a `ZeroDivisionError` if their divisor is integer `0`; `Float#fdiv` and `Float#/` return `Float::INFINITY`, instead. Further, when one or both operands are not `Integers`, and both dividend and divisor are zero, `Float::NAN` is returned.

To find the largest integer dividing two given integers with no remainder, use `Numeric#gcd` *m*, which computes the GCD of the receiver and *m*. Conversely, to find the smallest positive rational number that is an integer multiple of both the receiver and *m*, compute the LCM with `Numeric#lcm` *m*. `Numeric#gcdlcm` *m* returns a two-element `Array`, with the arguments' GCD as the first element, and their LCM as the last.

```
456.gcd 320  #=> 8
29.gcd 19    #=> 1
4392.lcm 282 #=> 206424
10.gcdlcm 4  #=> [2, 20]
```

The `#divmod` method divides its receiver by its operand then returns a two element `Array`, containing the integer quotient and the remainder. If solely

interested in the remainder produced, `Numeric#remainder` divides its receiver by its argument, returning what remains as an `Integer`.

```
90.divmod 9            #=> [10, 0]
772.divmod 6           #=> [128, 4]
40_500.remainder 1_000 #=> 500
```

Conversion & Coercion

A numeric may be converted to an equivalent numeric of a different class via either the implicit or explicit conversion protocol, with the following caveats:

- A `RangeError` is raised when attempting to convert either `Float::NAN` or `Float::INFINITY` to an `Integer`.
- A `RangeError` is raised when attempting to convert a `Complex` with a non-zero imaginary part to any other numeric class.
- In general, converting to `Integer` is lossy; that is, the process does not round-trip.

The numeric coercion protocol enables two operands of different numeric classes to be converted into object's of the same class without any loss of accuracy. It is used by methods which expect their operands to have the same class. It is effected with `Numeric#coerce(`*o*`)`, which returns an `Array` of two elements, which represent *o* and the receiver, respectively, and have the same class.

```
23.coerce 567 #=> [567, 23]
23.coerce 5.67 #=> [5.67, 23.0]
Math::PI.coerce Rational(1, 2) #=> [0.5, 3.141592653589793]
Complex(2).coerce Float::INFINITY #=> [(Infinity+0i), (2+0i)]
```

Comparison & Equality

Conceptually, two `Numeric` objects are equal in terms of `#==` if, and only if, their coercion results in two values that are `#eql?`. In practice, this logic is hard-coded into each `#==` method, so `#cocerce` is not called.

```
3 == 3.0                           #=> true
a, b = 3.coerce(3.0)               #=> [3.0, 3.0]
a.eql? b                           #=> true

Rational(2, 3) == 2.3              #=> false
a, b = Rational(2, 3).coerce(2.3) #=> [2.3, 0.6666666666666666]
a.eql? b                           #=> false
```

Therefore:

```
23 == 23
10 != 10.00000000000001
34 != -34
Rational(56, 64) == Rational(7, 8)
Rational(3, 4) == 0.75
Complex(3) == Rational(3)
Complex(2, 10) != Rational(2, 10)
Rational(Rational(16.0, 1), Complex(8)) == 2
```

The operators #>, #>=, #<, #<=, provide the inequalities greater than, greater than or equal to, less than, and less than or equal to, respectively. Numerics compare precisely as you would expect. `Float::INFINITY` is greater than any numeric except itself and `Float::NAN`. Likewise, `-Float::INFINITY` is less than any numeric except itself and `Float::NAN`. `Float::NAN` is incomparable with every numeric, including itself.

```
34 < 43
109 >= 100.9
Rational(7,5) > Rational(1, 5)
p ((2**781) < Float::INFINITY) #http://redmine.ruby-lang.org/issues/show/3648
Float::NAN != Float::NAN
```

Rounding

Non-`Complex` numerics can be rounded to adjust their precision. This makes little sense for `Integers`, of course, so for `Integer` receivers these methods just return `self`. `Numeric#ceil` and `Numeric#floor` round their receiver up or down, respectively, to the nearest integer. `Numeric#truncate` converts its receiver into an `Integer` by removing any fractional part. This is

equivalent to rounding the receiver to zero digits of precision, therefore identical to `Numeric#round` with no arguments. When `#round` is supplied with a positive `Integer` argument, the receiver is rounded to that many decimal places. If the argument is $-i$, the receiver is rounded to the nearest 10^i.

```
n, r = 12345.67890, Rational(20, 7)
n.truncate    #=> 12345
n.floor       #=> 12345
n.ceil        #=> 12346
n.round(2)    #=> 12345.68
n.round(-3)   #=> 12000
r.truncate    #=> 2
r.ceil        #=> 3
r.round 3     #=> (2857/1000)
```

Predicates

The following predicates are available for testing numerics:

`Integer#even?`
: `true` if the receiver is zero or divisible by two; `false` otherwise.

`Float#finite?`
: `true` if the receiver is neither `Float::NAN` nor `±Float::INFINITY`; `false` otherwise.

`Float#infinite?`
: `nil` if the receiver is finite or NaN, `-1` for negative infinity, and `1` for positive infinity.

`#integer?`
: `true` for instances of `Fixnum` and `Bignum`; `false` for all other numerics.

`Float#nan?`
: `true` if the receiver is `Float::NAN`; `false`, otherwise.

`#nonzero?`
: `self` if the receiver is non-zero; `nil` otherwise.

`Integer#odd?`
: `true` if the receiver is not even; `false` otherwise.

`#real?`
: `true` for all numerics except instances of `Complex`.

`#zero?`
: `true` if the receiver is zero; `false` otherwise.

```
78690.even?                  #=> true
-23.5.finite?                #=> true
(-Float::INFINITY).finite?   #=> false
(1/0.0).infinite?            #=> 1
(23**7658).integer?          #=> true
(0/0.0).nan?                 #=> true
(45789 * 0).nonzero?         #=> nil
33.odd?                      #=> true
Rational(1, 2).real?         #=> true
Complex(Rational(1, 2)).real? #=> false
(0.005 * 0.004).zero?        #=> false
```

Moduluar Arithmetic

The `%` operator, and its alias `#modulo`, return their receiver modulo their operand, i.e. the remainder when dividing the former by the latter. If the divisor is `0`, `%` raises a `ZeroDivisionError`; if it is `0.0`, `Integer#%` raises `ZeroDivisionError` whereas `Float#%` returns `Float::NAN`.

```
5 % 10                 #=> 5
15 % 10                #=> 5
(-2..6).map{|n| n % 4} #=> [2, 3, 0, 1, 2, 3, 0, 1, 2]
```

Exponentiation

The binary `*` operator raises its receiver to a given power, e.g. `3 ** 2 == 9`. `Math.sqrt` and `Math.cbrt` return the square or cube root, respectively, of their argument. There is not a generic n^{th} root method, but it can be implemented as *x* `**` `Rational(1,` *n*`)`.

Finiteness

The special, signed value `Float::INFINITY` is returned by numeric methods that produce values too large for Ruby to represent. It can be tested for with the predicate `Float#finite?`, which returns `true` if the receiver is neither positive/negative infinity nor NaN; `false` otherwise. Similarly, `Float#infinite?` returns `nil` for finite values and NaN, `-1` for negative infinity, and `1` for positive infinity.

The special value `Float::NAN` represents NaN, i.e. an undefined or unrepresentable number. For example, it is returned by `0 / 0.0`, `0 * Float::INFINITY`, and any method with `Float::NAN` as an argument. `Float::NAN != Float::NAN`, so to test if a value is NaN, use the predicate `Float#nan?`, which returns `true` if it is; `false` otherwise.

Pseudo-Random Numbers

Instances of the `Random` class represent pseudo-random number generators (hereafter: PRNGs). Each encapsulates state, an `Integer` seed, and an algorithm for generating pseudo-random numbers based on that seed.

The seed may be supplied as an argument to `Random.new`. Otherwise it has the value of `Random.new_seed`, which is derived from the system PRNG-e.g. `/dev/urandom`-if available, or a combination of the current time, process ID, and a sequence number. For a given seed, the output of a PRNG is predictable. Therefore, by setting the seed to a known value before the PRNG is used, its output is made deterministic, thus testable. `Random#seed` returns the seed of its receiver.

```
prng = Random.new 42
prng.seed       #=> 42
Random.new.seed #=> 218115049506821704411283774120640050107
```

A pseudo-random number may be generated with `Random#rand`. If given no arguments, a pseudo-random `Float` between `0.0` and `1.0` is returned. A different upper limit may be chosen by supplying a non-zero, positive `Integer` or `Float` argument. The generated pseudo-random number has the

same class as the argument. When given a `Range` whose end-points respond to `#+` and `#-`, it returns a pseudo-random number from the values encompassed. `Random#bytes` *`len`* returns a pseudo-random ASCII-8BIT `String` of length *len*.

```
prng = Random.new 0xDEAF
prng.rand                #=> 0.9812733995904889
prng.rand 30             #=> 1
prng.rand Math::PI       #=> 0.21283178272607403
prng.rand(2**10..2**100) #=> 1018356302920191178791579529869
prng.bytes 5             #=> "\xA4.~\xAC\xAB"
```

Ruby maintains its own PRNG named `Random::DEFAULT`, whose seed was generated with `Random.new_seed`. It is used by methods that produce pseudo-random results such as `Array#sample` and `Kernel.rand`. The latter behaves as `Random#rand` when given no arguments or an argument of zero. Otherwise, it generates a pseudo-random `Integer` between zero and the absolute value of its `Integer` argument.

The seed of `Random::DEFAULT` may be set explicitly by calling `Kernel.srand` with an `Integer` argument. If the argument is omitted, the seed is set to a new value of `Random.new_seed`. In both cases, `srand` returns the previous seed.

```
srand 0xDEAF  #=> 158965985081249152526188226346818216914
rand          #=> 0.9812733995904889
rand 30       #=> 1
rand -45      #=> 25
rand Math::PI #=> 2
```

Trigonometry

The standard trigonometric functions are available as methods of the `Math` module. They measure angles in radians, and return `Floats`.

`Math.sin(`*`x`*`)`

The sine of angle *x*.

`Math.asin(`*x*`)`
: The principal value of the arc sine of *x*, i.e. the value whose sine is *x*.

`Math.sinh(`*x*`)`
: The hyperbolic sine of hyperbolic angle *x*.

`Math.asinh(`*x*`)`
: The inverse hyperbolic sine of *x*, i.e. the value whose hyperbolic sine is *x*.

`Math.cos(`*x*`)`
: The cosine of angle *x*.

`Math.acos(`*x*`)`
: The arc cosine of *x*, i.e. the value whose cosine is *x*.

`Math.cosh(`*x*`)`
: The hyperbolic cosine of hyperbolic angle *x*.

`Math.acosh(`*x*`)`
: The inverse hyperbolic cosine of *x*, i.e. the value whose hyperbolic cosine is *x*.

`Math.tan(`*x*`)`
: The tangent of angle *x*.

`Math.atan(`*x*`)`
: The arc tangent of *x*, i.e. the value whose tangent is *x*.

`Math.atan2(`*y*`, `*x*`)`
: The principal value of the arc tangent of *y*/*x*.

`Math.tanh(`*x*`)`
: The hyperbolic tangent of hyperbolic angle *x*.

`Math.atanh(`*x*`)`
: The inverse hyperbolic tangent of *x*, i.e. the value whose hyperbolic tangent is *x*.

`Math.hypot(`*x*`, `*y*`)`

The hypotenuse of a right-angled triangle with sides *x* and *y*, or the distance of point (*x*,*y*) from the origin.

Logarithms

The natural logarithm of a numeric, *n*, i.e. $\log_e$, is returned by `Math.log` *n*. A logarithm to base *b* can be found with `Math.log` *n*`,` *b*. Two shortcuts exist for common bases: `Math.log10` *n* and `Math.log2` *n*.

```
include Math
log E**7                            #=> 7.0
log 81, 9                           #=> 2.0
log10 10_000_000_000_000            #=> 13.0
log10 Float::RADIX**Float::MANT_DIG #=> 15.954589770191003
log2 Rational(1024, 16)             #=> 6.0
```

STRINGS

`Strings` are mutable sequences of characters with an associated encoding. They are generally created with literals, as explained below.

Literals

A *string literal* is either a double-quoted string literal or a single-quoted string literal.

Single-Quoted Strings

A single-quoted `String` is a string delimited by apostrophe (`U+0027`) characters or `%q`. Its contents are not subject to interpolation. The only recognised escape sequences are `\\` and `\`*`delimiter`*; any other escape is interpreted literally, i.e. as a reverse solidus followed by a single character.

Alternative Delimiters

A single-quoted string may also be delimited by arbitrary delimiters with the `%q`*`delimiter`*`…`*`delimiter`* construct, where *delimiter* is a single character. If *delimiter* appears in the string it must be escaped.

If the opening delimiter is `(`, `[`, `<`, or `{`, the closing delimiter must be the corresponding closing bracket. For example, if the opening delimiter is `[`, the closing delimiter must be `]`. When these paired delimiters are used, the same pair may appear inside the string as long as they are properly balanced.

```
%q&'\n& #=> "'\\n"
%q.
  s
  t
  r
  i
  n
```

```
  g
  . #=> "\n  s\n  t\n  r\n  i\n  n\n  g\n  "
%q<<b>bold</b>> #=> "<b>bold</b>"
```

Double-Quoted Strings

A double-quoted `String` is a `String` delimited with quotation marks (`U+0022`) characters or %Q. Its contents are subject to interpolation and character escapes.

String Interpolation

Interpolation is the embedding of the value of an expression within a string. The general form of the syntax is `#{`*`expression`*`}`. It is common for *expression* to be simply the name of a local variable which is to be substituted for its value.

The braces can be omitted if *expression* is the name of a global—, class—, or instance variable. However, in this case the variable name cannot be immediately followed by a character legal in an identifier, as ambiguity results.

Alternative Delimiters

A double-quoted string may also be delimited by arbitrary delimiters with the `%Q`*`delimiter`*`…`*`delimiter`* and `%`*`delimiter`*`…`*`delimiter`*constructs, where *delimiter* is a single character. If *delimiter* appears in the string it must be escaped.

If the opening delimiter is `(`, `[`, `<`, or `{`, the closing delimiter must be the corresponding closing bracket. For example, if the opening delimiter is `[`, the closing delimiter must be `]`. When these paired delimiters are used, the same pair may appear inside the string as long as they are properly balanced.

```
%&"& #=> '"'
%Q;\
 #{2**2}; #=> " 4"
%Q<<b>\u0062old</b>> #=> "<b>bold</b>"
```

Here Documents

Here documents extend the concept of alternative delimiters to allow an arbitrary *sequence* of characters as a delimiter. They begin with << followed immediately by an arbitrary identifier or string literal. Their content begins on the following line and continues until that same identifier/string is seen on a line by itself with no intervening whitespace. If there is a hyphen between the opening << and the delimiter, i.e. <<-*delimiter*, the closing delimiter may be preceded with whitespace. The final newline character before the closing delimiter is part of the here document's contents: a minimal here document is equivalent to `"\n"`.

If the delimiter is an identifier or double quoted string, the contents of the here document is interpreted with double-quoted string semantics. Otherwise, the here document's contents are interpreted literally: all escape sequences and interpolation constructs are ignored.

```
poem = <<-POEM
Me up at does

out of the floor
quietly Stare

a poisoned mouse

still who alive

is asking What
have i done that

You wouldn't have
-- Edward Estlin Cummings
    POEM
```

String Escapes

Escape sequences are character sequences prefixed with a reverse solidus (U+005C) that have a meaning other than their constituent characters when appearing in double-quoted `String`s.

Character Escapes

A character escape consists of a reverse solidus followed by a single character, *char*. If *char* is one of [abcefnrstuvxCM01234567], the escape has the meaning given in the table below. If *char* is a literal line terminator, both the reverse solidus and the line terminator are removed from the string. In all other cases `\`*char* evaluates to *char*.

```
puts "The worst\b\b\b\b\b\b best " #=> "The best "
puts "s p\sa\tc\ve\ s"
#=> s p a        c
#=>          e s
puts "no\
space" #=> 'nospace'
```

Byte Escapes

Both byte escapes force the string in which they are embedded to have ASCII-8BIT encoding if the bytes are invalid in the source encoding.

Octal Byte Escapes

A reverse solidus followed by an octal number between zero and 377_8 represents the given byte.

```
"\157\143\164\141\154" #=> "octal"
```

Hexadecimal Byte Escapes

A hexadecimal byte escape consists of `\x` followed by a hexadecimal number ≤ FF_{16}. It represents the given byte.

```
"\x68\x65\x78" #=> "hex"
```

Control Escapes

The escape sequence for control character`Ctrl-`*`char`* consists of a reverse solidus, either `c` or `C-`, then *char*, which may be a character or escape. (*char* must not be a control, Unicode, hexadecimal, or three-digit octal, escape). Its value is the character whose character code is *char* ∧ 0x9F.

```
"\C-Z".ord #=> 26
"\cx".ord #=> 24
```

Meta Character Escapes

The meta character escape `\M-`*`char`* represents the character whose character code is that of *char* ∨ 0x80, where *char* is a single character or escape. (*char* must not be a meta character, Unicode, hexadecimal, or three-digit octal, escape).

```
"\M-y".ord #=> 249
"\M-\C-B".ord #=> 130
```

Unicode Escapes

An arbitrary Unicode character may be embedded in a string by specifying its codepoint as four hexadecimal digits following `\u`.

The `\u{}` construct extends this ability to embedding multiple codepoints with the same escape. The curly braces delimit one or more hexadecimal codepoints separated by whitespace. This form does not restrict a codepoint to four digits.

```
# U+263A WHITE SMILING FACE (☺)
"\u263A" #=> "☺"

# U+1F090 DOMINO TILE VERTICAL-06-03 (?)
# U+1F091 DOMINO TILE VERTICAL-06-04 (?)
"\u{1f090 1F091}" #=> "??"
```

Both forms of the Unicode escape sequence force the string in which they are embedded to have UTF-8 encoding. Therefore they are illegal in a file that has both a non-UTF-8 source encoding and a string containing literal multibyte characters in that encoding.

Summary

Escape Sequence	Interpretation
`\a`	`U+0007`: The BEL character. Rings the console bell.
`\b`	`U+0008`: The Backspace character.
`\e`	`U+001B`: The ESC character.
`\f`	`U+000C`: The Form Feed character.
`\n`	`U+000A`: The Newline character.
`\r`	`U+000D`: The Carriage Return character.
`\s`	`U+0020`: The Space character.
`\t`	`U+0009`: The Tab character
\uhexhexhexhex	The Unicode codepoint specified by the four given hexadecimal digits.
\u{codepoints}	The Unicode codepoint(s) specified by codepoints.
`\v`	`U+000B`: The vertical tab character.
	The byte specified by the three given octal digits, whose combined value does not exceed 377_8.
\octaloctal	The byte specified by the two given octal digits.
\octal	The byte specified by the given octal digit.
\xhexhex	The byte specified by the two given hexadecimal digits.
\xhex	The byte specified by the given hexadecimal digit.
\cchar \C-char	The control character `Ctrl-`*`char`*.
\M-char	Meta character *char*.

`\U+000A`
`\U+000D`
`\` 〈`U+000D`, `U+000A`〉
	A backslash before a line terminator escapes it, removing both the line terminator and backslash from the string.

`\`*`char`	A backslash before any other character evaluates to the character itself.

Characters

Character is defined by the ISO/IEC as "A member of a set of elements used for the organisation, control, or representation of data." [Tr15285] . The *set* of which they are members is an encoding. Therefore, in Ruby, a character is a specific byte sequence interpreted according to a given encoding. An implication is that by changing the encoding of a `String`, one also changes what characters it contains.

A character is represented as a `String` of length 1, i.e. there is no explicit *Character* class. It can be created with the standard `String` literal syntax, e.g. `'l'`, or via a *character literal*. The latter comprises a question mark followed by a single character, i.e. `?`*char*, where *char* is a literal character, or a character escape that results in a single character, such as `\u`*hex*, `\n`, `\t`, or a byte escape. `?`*char* is entirely equivalent to `"`*char*`"`. To create a character for a given codepoint, see Codepoints.

`String#chars`, and its alias `String#each_char`, return an `Enumerator` of the receiver's characters. Both yield each character in turn if a block is given.

A `String` may be treated as an `Array` of characters with String#[]. However, Flanagan a.m.p; Matsumoto suggest that the aforementioned enumerators "may be more efficient" when processing a `String` character-by-character [Flan08, pp. 58–58] .

Bytes

Fundamentally a `String` is an array of bytes. An ordered sequence of numbers, each of which is in the range `0`–`255`. Bytes have no inherent meaning, so are typically used in conjunction with a scheme that ascribes them semantics or values. In the case of a binary data format, this scheme may be embodied in a program's algorithms, or be otherwise out-of-band. For textual data, this scheme is termed an encoding.

Byte-level access generally assumes, but does not enforce, that the `String`'s encoding is ASCII-8BIT. Explicitly manipulating byte sequences in a `String`

containing text is at best ill-advised. It breaks the abstraction of an encoding, and may result in `Strings` with invalid encodings and spurious exceptions being raised.

`String#bytes`, and its alias `String#each_byte`, return an `Enumerator` of their receiver's bytes, represented as `Fixnums`. A specific byte position may be assigned to with `String#setbyte(`*`index`*`, `*`byte`*`)`, where *index* is the zero-based offset of the byte to be changed, and *byte* is the new value as an `Integer`. A byte may be retrieved from a given offset with `String#getbyte(`*`index`*`)`. The length of a `String` in bytes is returned by `String#bytesize` as an `Integer`.

Codepoints

Unicode assigns each character in its repertoire a unique integer *codepoint*. This identifies a character, irrespective of its encoding. It is typically represented with the notation `U+`*`hex`*, where *hex* is the codepoint in uppercase hexadecimal digits. For example, `U+010E` is the codepoint for Ď (*Latin Capital Letter D with Carron*). In the UTF-8 encoding this character is represented by the byte sequence `"\xc4\x8e"`, in EUC-JP it is represented as `"\x8f\xaa\xb0"`, and in ISO-8859-2 it is simply `"\xcf"`. However, all three cases represent the same character, so all three consist of the codepoint 270.

The Unicode character escape allows a character with a given codepoint to be embedded into a `String`. Similarly, `Integer#chr(`*`encoding`*`)` interprets its receiver as a codepoint in the named encoding, and returns the corresponding character. If the codepoint does not exist in the given encoding, an `ArgumentError` is raised.

Conversely, `String#ord` returns the codepoint of the first character in its receiver as an `Integer`. More generally, `String#codepoints`, and its alias `String#each_codepoint`, return an `Enumerator` of their receiver's codepoints represented as `Integers`. If given a block, each codepoint is yielded to it in turn. All three methods will raise an `ArgumentError` if their receiver has an invalid encoding.

Iteration

A `String` can be iterated over by byte, character, codepoint, or line, using the methods summarised below. Each method returns an `Enumerator`, or yields each element in turn to a given block.

Method	Iterates
`String#bytes`	Bytes as `Fixnums`
`String#each_byte`	
`String#chars`	Characters as `Strings`
`String#each_char`	
`String#codepoints`	Codepoints as `Fixnums`
`String#each_codepoint`	
`String#lines`	Lines as `Strings`
`String#each_line`	

A sequence of consecutive characters may be enumerated with `String#next`, and its alias `String#succ`, which return a copy of the receiver with the codepoint of its last character incremented by one. However, ASCII alphanumeric characters are special-cased: *z*/*Z* is followed by *aa*/*AA*; and digits increment as integers. There are bang variants of both methods, which modify their receiver in-place. This behaviour forms the basis of `String#upto(`*e*`)`, which returns an `Enumerator` of the sequence beginning with its receiver and ending with *e*. If given a block, it yields each element in turn.

```
s = 'next'
s.succ! #=> 'nexu'
[*s.upto('neya')]
#=> ["nexu", "nexv", "nexw", "nexx", "nexy", "nexz", "neya"]
?(.upto(11.to_s).to_a
#=> ["(", ")", "*", "+", ",", "-", ".", "/", "0", "1", "2",
#    "3", "4", "5", "6", "7", "8", "9", "10", "11"]
```

Size

"How long is a piece of string." isn't always a rhetorical question, but in Ruby the answer depends on your unit of measurement. `String#length`, and its alias `String#size`, return the number of characters in their receiver. Therefore, they are dependent on the associated encoding: the length of a `String` may change simply by associating it with a different encoding. `String#bytesize`, however, returns the number of bytes contained in its receiver. It is unaffected by `String#force_encoding`.

A `String` is *empty* if it has a length of `0`. This can be tested with the predicate `String#empty?`, and effected with the destructive `String#clear`.

Equivalence

For two `Strings` to compare as equal in terms of `String#==` they must be byte-wise identical and associated with the same encoding. This last condition is dropped when both `Strings` consist entirely of ASCII characters and have an ASCII-compatible encoding. If the second operand responds to the implicit conversion protocol of `#to_str`, it is converted thus, then the result is tested for equivalence with the receiver. `String#eql?` behaves in the same manner, but does not attempt to convert its operand.

Comparison

`String#<=>` compares its receiver with its `String` argument in terms of their character codes. Neither operand is normalised. `String` mixes in `Comparable`, so gains `String#<`, `String#≤`, `String#>`, and `String#≥`, also.

The aforementioned methods are, by implication, case sensitive. `String#casecmp` provides a case-insensitive alternative for ASCII `Strings`; non-ASCII characters are compared as above. Alternatively, one must normalise the case of the two `Strings` themselves before comparing them.

```
# coding: utf-8
?a < ?A #=> false
```

```
'pre' < 'prefix' #=> true
?Β <=> ?β #=> -1
?α < ?β #=> true
'11' > '100' #=> true
?a.casecmp ?A #=> 0
'epsilON'.casecmp 'Epsilon' #=> 0
?Σ.casecmp ?ς #=> -1
```

Concatenation

`String#+` returns its receiver concatenated with its `String` argument, without modifying the former. No coercion is performed on the argument.

`String#<<`, and its alias `String#concat`, append their argument to their receiver, mutating the existing object rather than creating a new one, and return the receiver. The argument must be either an `Integer` codepoint, which is converted into the corresponding character before concatenation, or a `String`. In tight loops, therefore, `#<<` should be preferred to `#+=`, as the latter creates a new object each time.

```
str = 'con' #=> "con"
str + 'cat' #=> "concat"
str #=> "con"
str << 'cat' << 101 << 'nation' #=> "concatenation"
str #=> "concatenation"
```

Either approach requires at least one of the following conditions to hold:

- Both encodings are ASCII-compatible and one of the `Strings` is ASCII-only.
- One of the `Strings` is empty.
- The two encodings are compatible.

Repetition

`String#*` returns a new `String` comprising *n* copies of itself, where *n* is given by an `Integer` argument. The receiver is not modified.

```
(?a..?f).map.with_index(1){|l, i| l * i}.join
#=> abbcccddddeeeeeffffff
```

Substrings

`String#[]`, and its alias `String#slice`, provide access to specific portions of a `String`, allowing it to be treated as an array of characters.

An `Integer` argument, *i*, returns the i^{th} character, where the first character has the index `0`. If *i* is negative, it counts backward from the last character, so `String#[-2]` returns the penultimate character.

When two `Integer` arguments are given, `String#[i, l]`, the *l* characters are starting from index *i* are returned. *i* may be negative, with the same semantics as before, but *l* cannot be.

When the argument is a `Range`, `String#[i..j]`, a substring beginning at index *i* and ending at index *j* is returned.

A `String` argument is returned if contained in the receiver. Likewise, when a `Regexp` argument is given, whose pattern matches the receiver, the return value is the first matching substring. The latter may be accompanied by a second argument indicating the group of captured text to return: an `Integer` refers to the numbered group, a `Symbol` or `String` refers to the named group. However, if a group is specified but the pattern fails to match, an `IndexError` is raised.

If the receiver does not contain the given substring, `nil` is returned. If it does, `String#[]=` may also be used as an lvalue with any of the above forms. The result is that the substring returned is replaced in the receiver with the rvalue. For example, `'ab'[0] = ?b` returns `"bb"`.

```
str = 'In absentia'
str[4, 4] = str[4].succ
str[/t(?<ear>ia)/, :ear] = ?u
str #=> "In actu"
str[?c] = 'bsen'
str[-1] = str[0].downcase + str[3]
```

```
str #=> "In absentia"
str[3..-1] = 'casu'
str #=> "In casu"
```

`String#insert(`*i*`, `*s*`)` is equivalent to `String#[`*i*`] = `*s*. However, in addition to modifying its receiver, `#insert` returns the new `String`, as opposed to `#[]=` which returns *s*.

`String#slice!` accepts all the same combinations of arguments, but deletes the substring from the receiver.

`String#index` returns the first zero-based index of the matched substring, as opposed to the substring itself. The substring may be given as a `String` argument, and optionally followed by an `Integer` specifying the index to search from. If a `Regexp` argument is provided instead, the index of the start of the match is returned. By contrast, `String#rindex` accepts the same arguments but returns the index of the rightmost match. When no matching substring is found, `nil` is returned.

```
alphabet = [*?a..?z].join
alphabet.index ?e #=> 4
alphabet.index /[aeiou]/ #=> 0
alphabet.index '~' #=> nil
alphabet.index(/[[:xdigit:]]/) #=> 0
alphabet.rindex(/[[:xdigit:]]/) #=> 5
alphabet << alphabet
alphabet.index ?e, 10 #=> 30
```

`String#[]` can be used to test for the presence of a substring on account of its returning `nil` when the substring is not present. A slightly clearer approach is `String#include?`, which returns `true` if its `String` argument is contained within the receiver; `false` otherwise. `String#start_with?` and `String#end_with?` behave similarly, but require the substring to be located at the beginning or end, respectively, of the receiver.

```
uname = `uname -a`
#=> "Linux paint 2.6.32-23-generic #37-Ubuntu SMP Fri Jun 11 07:54:58 UTC 2010
# i686 GNU/Linux\n"
uname.start_with?('Linux') #=> true
uname.chomp.end_with?('Linux') #=> true
```

```
uname.include?('ubuntu') #=> false
uname.include?(2010.to_s) #=> true
uname.include?(?#) #=> true
```

Searching & Replacing

`String#sub(`*r*`, `*s*`)` replaces the first occurrence of a `Regexp`, *r*, with a `String`, *s*, then returns the new `String`. *s* may contain back-references to capture groups in the pattern, which are substituted for the corresponding match. In a single-quoted `String` a numbered group is referenced as `\`*d*, where *d* is the group number, while `\k<`*n*`>` references the capture group named *n*. In a double-quoted `String` the reverse solidus must be doubled, i.e. `\\`*d* and `\\k<`*n*`>`.

The first argument can be given as a `String` instead of a `Regexp`, in which case it is treated like a pattern with the metacharacters escaped. Therefore, this form is similar to `String#[`*r*`] = `*s*.

If the second argument is omitted, a block must be supplied. It is passed the matched text and must return the replacement `String`. If it is a `Hash` that has the match as a key, the replacement `String` is the corresponding value.

To replace all occurrences of a pattern use `String#gsub` instead, with the arguments described above. If `String#gsub` is called with neither a replacement `String` nor a block, it returns an `Enumerator`.

The `String#sub!` and `String#gsub!` variants behave identically to their unadorned equivalents except they modify the receiver in-place. If the substitution succeeded, the return value is the receiver; otherwise, it is `nil`.

```
alphabet = [*?a..?f].join
alphabet.sub(/U/, 'you?') #=> "abcdef"
alphabet.gsub(/[b-e]/){|match| "<U+#{'%.4X' % match.ord}>"}
#=> "a<U+0062><U+0063><U+0064><U+0065>f"
alphabet.sub!(?V, 'for Vendetta') #=> nil
'3 + -4'.sub(/(?<a>\d+) \+ -(?<b>\d+)/,
             "== \\k<a> - \\k<b>")
#=> "== 3 - 4"
subs = {?& => '&amp;', ?> => '&gt;'}
```

```
subs.default = '?'
'You & I > he & they!'.gsub(/[[:punct:]]/, subs)
#=> "You &amp; I &gt; he &amp; they?"
```

Splitting, Partitioning, & Scanning

`String#split` returns an `Array` of its receiver divided into substrings according to a delimiter. The default delimiter is given by `$;`, which is initially `nil`. This is equivalent to a single space character, i.e. `'Horses for courses'.split` returns `['Horses', 'for', 'courses']`. If a `String` or `Regexp` argument is given, that is used as the delimiter. The delimiter is omitted from the results.

`String#partition` also requires a `String` or `Regexp` argument specifying a delimiter. It always returns a three-element `Array` comprising the text preceding the first occurrence of the delimiter, the delimiter itself, then the text following the first occurrence of the delimiter. `String#rpartition` behaves similarly except it uses the last occurrence of the delimiter.

```
str = 'Out of memory'
str.split #=> ["Out", "of", "memory"]
str.split('o') #=> ["Out ", "f mem", "ry"]
str.split(/o\s?/i) #=> ["", "ut ", "f mem", "ry"]
str.partition('o') #=> ["Out ", "o", "f memory"]
str.rpartition('o') #=> ["Out of mem", "o", "ry"]
```

`String#scan` is almost the inverse of `#split`. It repeatedly matches its `Regexp` argument against the receiver, returning an `Array` of the results, i.e. whereas `#split` returned the text surrounding the matches; `#scan` returns the matches themselves. If the pattern contains capturing groups, each element of the `Array` is an `Array` of captures; otherwise it is a `String` containing the matched text. If the argument is given as a `String`, it is interpreted literally: regular expression metacharacters are ignored. If a block is supplied, it receives each element of the result `Array` in turn.

```
"3^3 = 27".scan(/\d/) #=> ["3", "3", "2", "7"]
cube = 'faces: 6; volume: a^3; area: 6a^2'.scan(/(\w+): ([^;]+)/).map do |k, v|
  [k.to_sym, v]
end
```

```
Hash[cube] #=> {:faces=>"6", :volume=>"a^3", :area=>"6a^2"}
'Cubes: 1^3 = 1, 2^3=8, 3^3 = 27  4^3 = ?, 5^3 = 125, 6^3 = 215'.
  scan(/(?<base>\d+)\^3\s*=\s*(?:(?<an>\d+)|(?<un>\?))/) do |base, answer|
    cube = base.to_i ** 3
    puts "%d^3 == %d" %  [base, cube] unless cube == answer.to_i
  end
# 4^3 == 64
# 6^3 == 216
```

Letter Case

`String#downcase` converts all characters in its receiver to lowercase, while `String#upcase` does the opposite. `String#swapcase` toggles the case of each character: lowercase characters are converted to uppercase, and vice versa. `String#capitalize` converts its receiver to lowercase then converts its first character to uppercase. All four methods have a corresponding bang method which modifies the receiver in-place. However, these methods only understand the capitalization of ASCII characters; any other character is left unchanged. Case-insensitive comparison was covered in the Comparison section.

```
str = 'cAsE-sEnSiTiVe'
str.downcase #=> "case-sensitive"
str.upcase #=> "CASE-SENSITIVE"
str.capitalize #=> "Case-sensitive"
str.swapcase #=> "CaSe-SeNsItIvE"
```

Whitespace

`String#chomp(`*s* `= $/)` returns a copy of its receiver with *s* deleted from the end. When *s* is omitted, it has the value of `$/`, which defaults to `"\n"`. `String#chop` returns its receiver minus the final character, regardless of its value. However, if a `String` ends with `"\r\n"`, both methods treat it specially: `#chop` deletes both characters at once, while `#chomp` does likewise when called with no argument and `$/ == ?\n`. Both methods have bang variants which modify their receiver in-place. When they truncate the receiver, they return it; otherwise, they return `nil`.

Removal of a trailing line terminator can be generalised to the removal of any trailing and/or preceding whitespace, i.e. any consecutive combination of `"\s"`, `"\t"`, `"\r"`, and `"\n"`. `String#lstrip` deletes whitespace from the left, `String#rstrip` deletes it from the right, and `String#strip` deletes it from both sides. All have bang variants which modify the receiver in-place.

`String#center(`*w*`)` returns a `String` of at least *w* characters in length. If its receiver is shorter than *w*, it pads it on either side with as many space characters as is necessary to achieve this goal. If a `String` is supplied for the optional second argument, it is used for padding instead of `"\s"`. `String#ljust` and `String#rjust` accept the same arguments but justify their receiver to the left or right, respectively.

```
str = " orient\tate\n"
str = str.lstrip             #=> "orient\tate\n"
str.chop!                    #=> "orient\tate"
str.chomp                    #=> "orient\tate"
str.chomp!('ate')            #=> "orient\t"
str = str.center(15, "\t")   #=> "\t\t\t\torient\t\t\t\t\t"
(str.strip << $/).capitalize #=> "Orient\n"
```

Converting to Numeric

`String#to_i(`*r* `= 10)` interprets the receiver as an integer in base *r*, returning the corresponding `Integer` object. Hexadecimal and octal numbers may be prefixed with `0x` or `0`, respectively. `String#oct` and `String#hex` are equivalent `#to_i(8)` and `#to_i(16)`, respectively.

A `Float` may be created from a `String` beginning with either form of `Float` literal, i.e. two groups of digits separated with a full stop or exponential notation, via `String#to_f`. A `Rational` may be instantiated from a `String` of the form *n*`/`*d*, where *n* is the numerator, and *d* the denominator, with `String#to_r`. If *d* is omitted, it has the value `1`. Lastly, `String#to_c` creates a `Complex` number. If the receiver is in the format recognised by `#to_f`, `#to_r`, or `#to_i(10)`, the number represented is the real part. If it is followed by a signed number with an `i` suffix, that is the imaginary part. If either part is omitted, it has the value `0`.

In all cases, the number may be prefixed with a sign and arbitrary whitespace, and followed by arbitrary characters. Digits may be separated with low lines. If a number cannot be extracted, it is assumed to be `0`.

```
' 34_89'.to_i         #=> 3489
'b_ee_f or lamb?'.hex #=> 48879
'tofu?!?'.to_i(8)     #=> 0
'-23.89'.to_f         #=> -23.89
'4.1e6'.to_f          #=> 4100000.0
'3/4.2'.to_r          #=> (3/4)
"\t2/3+2i\t".to_c     #=> ((2/3)+2i)
```

Checksums

`String#sum(`*b* `= 16)` calculates a rudimentary *b*-bit checksum of the receiver, which it returns as an `Integer`. If *b* ≤ `0`, this is simply the sum of the receiver's bytes; otherwise, it is equivalent to `String#bytes.reduce(&:+) & ((1 <<` *b*`) - 1)`.

`String#crypt(`*s*`)` is a thin wrapper around `crypt(3)`. The salt, *s*, is at least two characters from the alphabet `[a-zA-Z0-9./]`. The receiver is encrypted with *s* to produce a thirteen-character `String` from the aforementioned alphabet, which begins with the first two characters of *s*.

```
str = 'need-to-know'
str.sum         #=> 1176
str.sum 4       #=> 8
str.sum -10     #=> 1176
str.crypt('a.') #=> "a.YKZnbNk2rOs"
str.crypt('w8') #=> "w8Y/hB.q9SVxo"
```

Sets of Characters & Transliteration

`String#squeeze` returns a copy of its receiver without any runs of consecutive characters, i.e. it deletes all but the first character in each run. Optionally, one or more `String` arguments may be given, in which case only runs of characters appearing in the intersection of these arguments are collapsed. If an argument contains two characters separated by a hyphen

minus sign, it represents all characters in that range. An argument with a caret as the first character represents the negation of its contents. The bang variant modifies the receiver in-place, returning `nil` if no changes were made.

```
s = 'Aaa bbb cc dd e'
s.squeeze              #=> "Aa b c d e"
s.squeeze(?a, ?b)      #=> "Aaa bbb cc dd e"
s.squeeze('a-d')       #=> "Aa b c d e"
s.squeeze('a-e', '^b') #=> "Aa bbb c d e"
```

The same approach can be applied to counting and deleting characters. `String#count` and `String#delete` both accept the same forms of argument with the same semantics. The former returns a `Fixnum` indicating the specified character's frequency in its receiver, while the latter returns a copy of its receiver with the characters removed. `String#delete!` behaves as `#delete`, except the receiver is modified in-place and `nil` is returned if no changes were made.

```
s = 'Aaa bbb cc dd e'
s.count('Aa-e')        #=> 11
s.count(?a, ?b)        #=> 0
s.count('a-d')         #=> 9
s.delete!('b-e', '^c') #=> "Aaa  cc  "
s.count('a-e', '^b')   #=> 4
s.count('A-Za-z ')     #=> 9
```

`String#tr(`*t*`, `*f*`)` transliterates each character specified by its first argument with the corresponding character specified by its second. Both arguments may use the range construct introduced above, and *t* may use the caret construct. If *f* has more characters than *t*, the latter is padded with the last character of the former. `String#tr!` behaves in the same fashion, except it modifies the receiver in-place and returns `nil` if no changes were made. `String#tr_s` behaves as `#tr`, except it collapses consecutive runs of characters in the regions it affects.

```
str = 'stuffed shirt'
str.tr('st', 'S')      #=> "SSuffed ShirS"
str.tr('a-z', 'A-Z')   #=> "STUFFED SHIRT"
str.tr_s('a-z', 'A-Z') #=> "STUFED SHIRT"
str.tr('shirt', 'toy') #=> "tyuffed toyyy"
```

Debugging

`String#inspect` and `String#dump` return a copy of the receiver enclosed in quotation marks, with special characters escaped, in order to aid debugging. ASCII control characters are replaced with their corresponding escape sequences, if any, and other non-printing characters are substituted for the corresponding byte escape. Quotation marks are backslash escaped. In addition, `String#dump` substitutes non-ASCII characters with the corresponding `\u{`*`codepoint`*`}` escape. If the receiver is associated with an ASCII-incompatible encoding, *enc*, `#dump` appends `.force_encoding('`*`enc`*`')` to its result.

```
# coding: utf-8
str = "\"a\b\u{63}\t\x12\u{200}\""
str.inspect #=> "\"a\bc\t\u0012̀\""
str.dump    #=> "\"a\bc\t\x12\u{200}\""
```

Encoding

The encoding of a `String` literal is normally that of the source file in which it appears, with the caveats noted above for Unicode and byte escapes.

Forcing an Association

The encoding associated with a `String` may be changed independently of its contents. This is necessary if the contents are valid in one encoding, but associated with another. Transcoding would be inappropriate because it would alter the underlying bytes, which are already perfectly valid. The solution is the destructive method `String#force_encoding`, which associates its receiver with the encoding given as an argument. This operation will always succeed, even if the `String`'s contents are invalid in the new encoding.

Yui Naruse, a member of the Ruby core team, cautions against the use of this method: "`String#force_encoding` should be sparsely used since Strings have already had appropriate encodings assigned when those are created, or

read from files specifying the encoding.... If you need to use `String#force_encoding` in your library, you should reconsider your library design. You should not use this method thoughtlessly." [Harada09] .

Valid Encodings

A `String`'s encoding is *valid* if its constituent bytes are properly formed according to the encoding it is associated with. This check makes no claims as to whether the characters in the `String` exist in the encoding, for it inspects syntax rather than semantics.

In general, if you create a `String` via a literal, or consume a `String` as input that is validly encoded, manipulating it as a sequence of characters will ensure its encoding remains valid. However, associating a `String` with an improper encoding, treating it as byte array, or consuming garbage input, may all result in an invalid encoding. The `String#valid_encoding?` predicate returns `true` if the `String`'s encoding is valid; `false` otherwise.

ASCII Only

Many operations on `Strings` contain optimisations for `Strings` that are *ASCII-only*-containing 7-bit ASCII characters exclusively-regardless of the associated encoding. For example, two `Strings` with disparate encodings are compatible if they're both ASCII-only. In general, however, these optimisations are transparent so can be safely ignored. The `String#ascii_only?` predicate can be useful if you wish to perform your own optimisations along these lines.

Format Strings

A *format string* is a template specifying how a set of arguments should be interpolated into a new `String`. It contains arbitrary text-which is copied to the result unchanged-interspersed with *format sequences*-which describe how their corresponding argument should be converted before being substituted in their place.

`String#%` interprets its receiver as a format string, and its argument-a plurality are supplied as an `Array`-as the values to be interpolated, returning the expanded `String`. `Kernel#sprintf`, and its alias `Kernel#format`, interpret their first argument as a format string and subsequent arguments as the values to be interpolated.

A format sequence begins with a percent sign, then contains zero or more single-character *flags*, an optional minimal field width, an optional precision, and a mandatory conversion specifier, in that order. We will discuss each conversion specifier in turn, along with the flags they support, then conclude by explaining the other fields.

Textual Conversions

Firstly, the following text-based conversion specifiers are available:

Character
: Conversion specifier `c` interprets a numerical argument as a character code point, which it converts into the given character. An argument consisting of a one-character `String` is copied into the result unchanged.

Inspect
: Conversion specifier `p` is substituted for the result of sending `#inspect` to its argument.

String
: Conversion specifier `s` copies the argument into the result as a `String`.

If accompanied by a precision, both `p` and `s` copy at most *precision* characters into the result.

```
'%s %c %s' % ['Horses', 52, 'Courses']
#=> "Horses 4 Courses"
'%p =~ "%.5s..."' % [/[aeiou]/i, 'A slow, red fox']
#=> "/[aeiou]/i =~ \"A slo...\""
```

Numbers

The remaining conversion specifiers format numbers. All support a ␣ (a space character) flag, which prepends positive results with a space, a + flag which prepends a plus sign to positive results, and a `0` flag which pads fields with zeros instead of spaces. If ␣ and + are used together, the latter has precedence.

Converting Between Numerical Bases

Conversion specifiers can convert an argument into an integer in a given base. Negative results are usually represented in two's complement, prefixed with two full stops. However, if the ␣ flag is supplied they are given in their absolute form and prefixed with a hyphen-minus sign. When the `0` flag is given and the result is represented as two's complement, it is padded with the digit one fewer than the base. For example, negative octal numbers are padded with `7`s instead of `0`s.

Binary

Conversion specifiers `b` and `B` convert their argument to base `2`. The `#` flag causes `0b` to be prepended to the result of `b`, and `0B` to be prepended to the result of `B`, i.e. these conversion specifiers differ only in the case of their prefix.

Octal

The `o` conversion specifier converts its argument to base `8`. When this conversion specifier is used in conjunction with the `#` flag, and the result is positive, `0` is prepended to the result.

Decimal

Conversion specifiers `i`, `u`, and `d`, all of which are identical, convert their argument to base `10`. Negative numbers are prefixed with a hyphen-minus sign.

Hexadecimal

Both the `x` and `X` conversion specifiers convert their argument to base `16`. The case of the conversion specifier dictates the case of the characters in the result. The `#` flag prepends `0x` or `0X` to the result, as appropriate.

```
'2:%b, 8:%o, 16:%x' % [200, 800, 1600]
#=> "2:11001000, 8:1440, 16:640"
'%#B' % ('%d' % 032)
#=> "0B11010"
'% x; %#x; %#010X' % [-54].*(3)
#=> "-36; 0x..fca; 0X..FFFFCA"
```

Numerical Notation

The notation used for displaying numerical arguments can also be configured with conversion specifiers. The ␣ flag again causes positive results to have a single space prepended, but has no effect on negative results. The # flag forces the result to contain a radix point, even if no digits follow.

Exponential

The e conversion specifier represents the argument in exponential notation in the form: [-]α.βe±γ, where α is a single digit, β is the fractional part consisting of *precision* (default: 6) digits, and γ is the two-digit exponent.The E conversion specifier behaves identically, expect a capital E is used to introduce the exponent.

Fixed-point

The *f* conversion specifier represents its argument with a whole part, preceded with a hyphen-minus sign if negative, and a fractional part, separated by a radix point. There are always *precision* (default: 6) digits following the radix point.

Exponential or fixed-point

Both the g and G conversion specifiers render their argument in the notation most appropriate for its magnitude: exponential if the exponent is < -4 or ≥ *precision*; fixed-point, otherwise. There are at most *precision* (default: 6) digits in the result. Trailing zeros are omitted from the fractional part of the result unless the # flag is given. The radix point is omitted unless followed by at least one digit. The exponent is introduced with e if the g conversion specifier is used; or E if G is used instead.

Hexadecimal exponential

A conversion specifier of a converts the argument to base-16 exponential notation (using lowercase digits) in the form: [-]0xα.βp±γ, where α is a

single hex digit, β is one or more hex digits, whose quantity is capped at *precision*, and γ is the single-digit decimal exponent. If the `A` specifier is used instead, the lowercase characters in the result are converted to uppercase, i.e. it takes the form `[-]0X`α`.`β`P±`γ, where α and β use uppercase digits.

```
'%f' % Math::PI
#=> "3.141593"
'%e' % Math::E
#=> "2.718282e+00"
'%E' % -0.0000231
#=> "-2.310000E-05"
'%g %.G %g' % [9_000, 12e-8, -12000000000.5]
#=> "9000 1E-07 -1.2e+10"
'%.20a' % 678.19e100
#=> "0x1.8394d0b22721f0000000p+341"
```

Hash Interpolation

If the arguments are given as a `Hash`, rather than an `Array`, they may be referenced from a format sequence by name instead of position. When a conversion specifier is immediately preceeded by <*key*>, where *key* names a key of the `Hash`, the corresponding value in the `Hash` becomes the format sequence's argument. A format sequence with a conversion specifier of of the form {*name*} is equivalent to <*name*>s.

```
# coding: utf-8
%q@
Subject: Are You Our Missing Winner?
To: %{email}

Dear %{name},
Congratulations! You have been selected to receive a cash prize
of £%.2<prize>f!!! Your name was selected at random by our
supercomputer to be a millionaire!?! But hurry, you must claim
your fortune by **%.19<deadline>p** or it will be gifted
to the next name on our shortlist.

Hurry. Please act now.
@ % {name: 'A. Patsy', prize: 2_883_712.28271, email: 'mark@aol.com',
```

```
     deadline: Time.now.utc + (60+24*3600)}
#=>
#Subject: Are You Our Missing Winner?
#To: mark@aol.com
#
#Dear A. Patsy,
#Congratulations! You have been selected to receive a cash prize
#of £2883712.28!!! Your name was selected at random by our
#supercomputer to be a millionaire!?! But hurry, you must claim
#your fortune by **2010-06-18 22:23:52** or it will be gifted
#to the next name on our shortlist.
#
#Hurry. Please act now.
```

Field Width & Justification

The *width* component of a format sequence is an optional decimal digit string (with non-zero first digit) specifying the minimum width of the field. Alternatively, the value can be specified in relative or absolute terms.

If the result has fewer characters than this width, it is padded; if it has more characters, the field is expanded as needed. By default, ␣ (a space character) is used for padding, but the `0` flag causes an alternative character, usually a zero, to be used instead.

Fields are normally right justified, but a flag of - justifies left instead. In the former case, padding characters are prepended to the result, while in the latter they are appended. If a negative width is given, this flag is implied, and the field's width is this value with the sign ignored.

```
digits = [*(1..9)]
'%1s' % digits.join
#=> "123456789"
'%20s = %#x' % [digits.join, digits.reduce(:+)]
#=> "           123456789 = 0x2d"
'>%0-20X<' % [*digits.join.bytes].join
#=> ">6DEC6BD4A460909     <"
```

Precision

The optional *precision* component comprises a full stop followed by an optional integer. If said integer is omitted or negative, it has the value `0`. Alternatively, the value can be specified in relative or absolute terms. The precision gives:

- The minimum number of digits to appear for `b`, `B`, `d`, `i`, `o`, `u`, `x`, and `X` conversions.
- The number of digits to appear after the radix point for `a`, `A`, `e`, `E`, `f`, and `F` conversions.
- The maximum number of significant digits for `g` and `G` conversions.
- The maximum number of characters to be taken from a string for `s` and `p` conversions.

Relative & Absolute Arguments

Usually, an argument is mapped to a conversion specifier implicitly: the n^{th} argument corresponds to the n^{th} conversion specifier. However, this may be made explicit so as to support arguments in a different order to their conversion specifiers, and/or to use the same argument multiple times.

Arguments are indexed with integers, the first argument having an index of 1. To marry a conversion specifier with an argument at index *i*, the format sequence should begin with `%`*i*`$`. However, if one format sequence has this form, they all must: numbered and un-numbered conversion specifiers cannot be mixed in the same format string.

The precision and width of a field can be provided in a similar way. A width given as `*`*i*`$` or a precision given as `.*`*i*`$`, specifies that the value is supplied by the argument indexed by *i*. However, this construct must always be paired with numbered conversion specifiers, which as noted above, are an all-or-nothing deal.

Alternatively, a width given as a single asterisk, or a precision given as `.*`, denote that the n^{th} argument holds *their* value, rather than that of their

conversion specifier. These constructs can be intermixed freely with unnumbered conversion specifiers.

```
'%2$s %1$s' % [:first, :second]
#=> "second first"
'%0*d' % [10, 2]
#=> "0000000002"
'%*1$1$d%*2$2$d%*3$3$d%*4$4$d%*5$5$d' % [*1..5]
#=> "1 2  3   4    5"
'%.*2$1$f %.*3$1$f %.*4$1$f' % [Math::PI, 0, 2, 4]
#=> "3 3.14 3.1416"
```

Unpacking

`String#unpack` interprets the receiver as a sequence of bytes-ignoring its encoding-which it expands into an `Array` of values according to a given template. The template consists of single-character *directives* optionally followed by an integer count, *count*, or an asterisk. Whitespace preceding directives is ignored. In general, *count* specifies the number of times the corresponding directive should be applied; when *count* is `*`, the directive is applied as many times as possible. The inverse operation, i.e. packing an `Array` of values into a `String` according to a template, is performed with `Array#pack`.

The following directives create `Integers` in the result `Array`.

In the following table, the *C Type* column is the corresponding datatype in the C programming language. The *Byte Order* column indicates whether the field's bytes are big-endian, or *network*, i.e. most-significant bit first; little-endian, or *VAX*, i.e. least-significant bit first; or in the native order, i.e. that of the current architecture's. A *BER-compressed integer* is an unsigned integer in base 128 with as few digits as possible. The high bit is set on each byte except the LSB.

Directive	Interpretation	C Type	Byte order
C	8-bit unsigned integer	unsigned char	N/A
S	16-bit unsigned integer	uint16_t	Native
L	32-bit unsigned integer	uint32_t	Native
Q	64-bit unsigned integer	uint64_t	Native
c	8-bit signed integer	char	Native
s	16-bit signed integer	int16_t	Native
l	32-bit signed integer	int32_t	Native
l! / l_	Signed integer of sizeof(long) bytes	long	Native
q	64-bit signed integer	int64_t	Native
S_ / S!	Unsigned integer of sizeof(unsigned short) bytes	unisgned short	Native
I / I_ / I!	Unsigned integer of sizeof(unsigned int) bytes	unsigned int	Native
L! / L_	Unsigned integer of sizeof(unsigned long) bytes	unsigned long	Native
s! / s_	Signed integer of sizeof(short) bytes	short	Native
i / i! / i_	Signed integer of sizeof(int) bytes	int	Native
n	16-bit unsigned integer	unsigned short	Big endian
N	32-bit unsigned integer	unsigned long	Big endian
v	16-bit unsigned integer	unsigned short	Little endian
V	32-bit unsigned integer	unsigned long	Little endian
U	UTF-8 character (codepoint)	N/A	N/A
w	BER-compressed integer	N/A	Big-endian

Integer directives for `String#unpack`

```
'NO'.unpack ?s #=> [20302]
'%.4x' % 1234 #=> "04d2"
"\x04\xd2".unpack ?n #=> [1234]
"\xd2\x04".unpack ?S #=> [1234]
"\xd2\x04".unpack ?v #=> [1234]
```

```
str = "\x52\x65\x61\x64\x52\x75\x62\x79"
str.unpack ?q      #=> [8746682423038797138]
str.unpack('ll')   #=> [1684104530, 2036495698]
str.unpack('lss')  #=> [1684104530, 30034, 31074]
str.unpack('lscc') #=> [1684104530, 30034, 98, 121]
"\u1111".unpack ?U #=> [4369]
"\x7F\x85Q".unpack('w*') #=> [127, 721]
"\x7F\x00\x00\x01".unpack('C4').join(?.) "127.0.0.1"
```

Similarly, the following directives unpack to `Floats`. The *Byte Order* column indicates whether the field's bytes are big-endian, or *network*, i.e. most-significant bit first; little-endian, or *VAX*, i.e. least-significant bit first; or in the native order, i.e. that of the current architecture's.

Directive	Interpretation	Byte order
`D` / `d`	Double-precision float	Native
`F` / `f`	Single-precision float	Native
`E`	Double-precision float	Little-endian
`e`	Single-precision float	Little-endian
`G`	Double-precision float	Big-endian
`g`	Single-precision float	Big-endian

`Float` directives for `String#unpack`

```
"\xDB\x0FI@".unpack ?f      #=> [3.1415927410125732]
"\x18-DT\xFB!\t@".unpack ?d #=> [3.141592653589793]
sqrt = Math.sqrt 2          #=>  1.4142135623730951
[sqrt].pack(?f).unpack ?f  #=> [1.4142135381698608]
[sqrt].pack(?d).unpack ?d  #=> [1.4142135623730951]
```

And, finally, the `String` directives:

Directive	Interpretation
`A`	Binary string, padded with spaces if shorter than *count*
`a`	Binary string, padded with nulls if shorter than *count*
`Z`	As `a`, but null represented as `*`
`B`	Big-endian bit string
`b`	Little-endian bit string
`H`	Big-endian hex string
`h`	Little-endian hex string

Directive	Interpretation
u	UU-encoded string, as produced by uencode(1)
M	Quoted-printable-encoded string (MIME encoding)
m	Base64-encoded string of *count* characters; 0 omits line feeds
P	Pointer to a structure (fixed-length string)
p	Pointer to a null-terminated string

```
"\xba".unpack('h*') #=> ['ab']
"\xab".unpack('H*') #=> ['ab']
'feed me?'.unpack('a2a2a*') #=> ["fe", "ed", " me?"]
bin = 105.to_s 2    #=>  "1100101"
?K.unpack('b7')     #=> ["1101001"]
"\xD2".unpack('B7') #=> ["1101001"]
")4F5A9\"!2=6)Y\n".unpack ?u #=> ["Read Ruby"]
"Read Ruby=\n".unpack ?M #=> ["Read Ruby"]
"UmVhZCBSdWJ5\n".unpack ?m #=> ["Read Ruby"]

mem_ptr = ['Read Ruby'].pack ?P #=> "\xA0\x9C\x12\t"
mem_ptr.unpack('P')  #=> ["R"]
mem_ptr.unpack('P*') #=> ["Read"]
mem_ptr.unpack('P8') #=> ["Read Rub"]

null_str = ['Read Ruby'].pack(?p) #=> "\x90N/\t"
null_str.unpack ?p                #=> ["Read Ruby"]
```

There are also directives to specify which byte should be read next: `x` skips forward one byte, `X` skips backward one byte, and `@`*o* skips to the byte at offset *o*.

Symbols

A `Symbol` is an immutable immediate representing an identifier. Two symbols with the same content will always be represented by the same object, e.g. `:glark.object_id == :glark.object_id`, so symbol comparisons are extremely efficient. However, `Symbol` objects are not garbage collected, so are unsuitable for storing data from unbounded collections; use `String`s instead. `Symbol`s are used for unique identifiers, such as `Hash` keys, as message selectors and method names, and as a substitute for constants.

`Symbol` is discussed in this section because it shares many of its methods with `String`. In fact, these `Symbol` methods are often implemented by converting the receiver to a `String`, invoking the method upon it, then converting it back to a `Symbol`.

A `Symbol` literal consists of a colon (`U+003A`) followed by a symbol or `String` literal. A symbol is an identifier-optionally followed by `=`, `?`, or `!`-an operator method selector, `[]`, or `[]=`. The `%s`*`delimiter`*`…`*`delimiter`* construct enables symbols to be specified between arbitrary delimiters along the same lines as %q.

A `Symbol` may also be created from a `String` with `String#to_sym`, but see the remark below regarding invalid encoding.

```
# coding: utf-8
:roland_barthes #=> :roland_barthes
:'σὐμβολον' #=> :σὐμβολον
%s{A proposition shows its sense.} #=> :"A proposition shows its sense."
"\u22F0".to_sym #=> :⋰
```

Encoding

A `Symbol` is associated with an encoding, but it cannot be manipulated directly. `Symbol` literals adopt the source encoding of the file in which they are used. If they consist exclusively of ASCII characters, they have the encoding US-ASCII. If a `Symbol` is created from a `String`, either via the :*`string`* syntax or `String#to_sym`, they have the encoding of that `String`. Therefore, to create a `Symbol` with a given encoding, first create a `String` with that encoding, then convert that `String` to a `Symbol`.

The encoding of a `Symbol` must always be valid. Therefore, attempting to convert a `String` with invalid encoding to a `Symbol`, causes an `EncodingError` to be raised.

ENCODING

An encoding is a mapping between byte sequences and characters[1]. Each program source file, `String`, `Symbol`, `Regexp`, `File`, and `IO` object is, relatively independently, associated with its own encoding.

The process of converting data from one encoding to another is called *Transcoding*. It is quite distinct from re-associating an object with another encoding: transcoding translates the underlying bytes to their equivalent representation in the target encoding, while association changes the label attached to an object.

The encoding associated with a source file-the source encoding-is by default US-ASCII. If a source file contains characters outside of this encoding, it must specify which one, otherwise Ruby refuses to load it.

The encoding associated with Strings, Symbols, and Regexps, is by default the source encoding of the file in which they are contained. However, if their literals contain certain character escapes, their encoding changed implicitly. As with source files, this association can be overridden on a per-object basis.

Encoding Class

Ruby represents the encodings that she understands as instances of the `Encoding` class, defining each as a constant under the `Encoding` namespace. The constant is named after the upper-case encoding name, with low lines replacing hyphen-minus characters. Methods that accept encodings as arguments recognise both `Encoding` objects, e.g. `Encoding::UTF_8`, and their names, e.g. `"utf-8"`. The `Encoding` object associated with a `String`, `Symbol`, or `Regexp` is returned by their `#encoding` method.

1. In Unicode terminology, this encompasses both CESs and CEFs.

Source Encoding

The *source encoding* is the character encoding of a given source file. It is US-ASCII by default. A `SyntaxError` is raised when a source file contains one or more characters invalid in the source encoding.

A file's source encoding may be specified inline by means of a *coding comment*: a specially formatted comment that declares the encoding of the lines that follow. If omitted, the default source encoding is assumed. If a source file contains a shebang line, the coding comment must appear on the second line; otherwise it must appear on the first.

The coding comment is a US-ASCII string which begins with a number sign (`U+0023`) and contains[2] the string `coding` followed by an equals sign (`U+003D`) or colon (`U+003A`) then the name of the source encoding. The encoding name is one of those returned by `Encoding.name_list` written in a case insensitive fashion.

The source encoding of the currently executing code can be obtained with the __ENCODING__ keyword.

```
# coding: utf-8
__ENCODING__ #=> #<Encoding:UTF-8>
```

IO Streams

An `IO` object is associated with an *external encoding* and, optionally, an *internal encoding*. The former is the actual encoding of data in the stream; the latter is the desired encoding. Both encodings have default values, but may be set for a specific stream with `IO#set_encoding(`*`external`*`, `*`internal`*`=nil)`.

---

2. That the coding comment need only *contain* `coding` allow Ruby to recognise Vim and Emacs modelines which declare a file's encoding. For example, `# vim: set fileencoding=utf-8 :` informs both Vim and Ruby that the file is encoded in UTF-8.

The default external encoding is returned by `Encoding.default_external`, and may be set by assigning an encoding to `Encoding.default_external=`, or invoking the interpreter with a switch of the form `-E`*encoding*. Otherwise, Ruby attempts to derive it from the user's environment[3]. If she fails, or finds an encoding that she doesn't recognise, she sets the default external encoding to *US-ASCII* or *ASCII-8BIT*, respectively[4].

If an `IO` stream needs to be processed in an encoding different to its external encoding, it must be transcoded. The target of the transcoding—the desired encoding—is called the *internal encoding* of an `IO` object. The default internal encoding is `nil` so no transcoding occurs by default. It may be specified by assigning an encoding to `Encoding.default_internal=` or invoking the interpreter with a switch of the form `-E:`*encoding*. It may be set to *UTF-8* by invoking the interpreter with the `-U` switch. The salient point is that, unlike the external encoding, the internal encoding is never derived automatically: transcoding happens only when it is explicitly requested.

If the internal encoding is `nil`, or the internal and external encodings are equal, there is no transcoding needed: the stream is already encoded as desired. Otherwise, Ruby transcodes data read from a stream from the external to the internal encoding, and transcodes data written to the stream from the internal to the external encoding. The transcoding works exactly the same as String#encode, so the #encode options Hash may be merged with the IO options Hash, wherever the latter is accepted. For example, it can be supplied as the final argument of IO.new or `IO#set_encoding`.

---

3. Either by consulting relevant environment variables—e.g. `LANG`, `LC_CTYPE`, and `LC_ALL`—or, on Windows, by invoking the system's `nl_langinfo_codeset()` or `GetConsoleCP()` functions.
4. In fact, the derived encoding becomes the *locale charmap encoding*. Then the *locale encoding* is set to the locale charmap encoding, if the latter was derived successfully; *US-ASCII*, if it couldn't be derived at all; or *ASCII-8BIT*, if it was derived but is not supported by Ruby. Finally, the default external encoding is initialised to the locale encoding.

ASCII-8BIT

Ruby defines an encoding named `ASCII-8BIT`, with an alias of `BINARY`, which does not correspond to any known encoding. It is intended to be associated with binary data, such as the bytes that make up a PNG image, so has no restrictions on content. One byte always corresponds with one character. This allows a `String`, for instance, to be treated as bag of bytes rather than a sequence of characters. `ASCII-8BIT`, then, effectively corresponds to the absence of an encoding, so methods that expect an encoding name recognise `nil` as a synonym.

Compatibility

Methods of `String` and `Regexp` that take another such object as an argument require the encodings associated with the objects to be *compatible*. An encoding is always compatible with itself, so operations involving two objects associated with the same encoding are allowed. Likewise, two objects are compatible if they are both ASCII-only.

The compatibility of other combinations of encodings can be determined with `Encoding.compatible?`, which compares the encoding of its two arguments, which are either `Encoding` objects or objects associated with encodings. If they are compatible, the encoding which would result from their combination is returned; otherwise, `nil` results. Operating on objects with incompatible encodings causes an `Encoding::CompatibilityError` exception to be raised.

Transcoding

Transcoding a `String` converts its bytes to the equivalent byte sequences in a given encoding, with which it associates the result. It is typically performed with `String#encode`, which returns its receiver transcoded from a *source* encoding to a *target* encoding. `String#encode!` operates in the same manner, but transcodes the receiver in-place.

By default, *source* is the receiver's current encoding, and *target* is the default internal encoding. When called with one encoding argument, this becomes the *target* encoding. When called with two encoding arguments, the first is the *target*, the second is the *source*. This last form is mainly useful when the `String` is associated with ASCII-8BIT: it associates the `String` with *source*, then transcodes from *source* to *target*.

If a character in the `String` does not exist in the *target* encoding, or the `String` contains bytes invalid in its current encoding, an exception is raised. This behaviour can be changed by supplying an *options* `Hash` as the final argument, whose form is described in the table that follows.

Key	Values	Description
`:cr_newline`	`true` or `false`	Whether to convert `\n` to `\r`.
`:crlf_newline`	`true` or `false`	Whether to convert `\n` to `\r\n`.
`:invalid`	`:replace` or `nil`	A value of `:replace` causes characters invalid in the source encoding to be substituted for the replacement string. A value of `nil`, which is the default, causes an `Encoding::InvalidByteSequenceError` exception to be raised in this scenario.
`:replace`	`String`	The *replacement string* used by the `:invalid` or `:undef` options. By default, it is `U+FFFD` for Unicode encodings and `?` for others.
`:undef`	`:replace` or `nil`	A value of `:replace` causes characters invalid in the destination encoding to be substituted for the replacement string. A value of `nil`, which is the default, causes an `Encoding::UndefinedConversionError` exception to be raised in this scenario.

The keys and values that are recognised in the options `Hash` accepted by `String#encode` and `String#encode!`. The Key column names a key of the `Hash`, and the Values column specifies its possible values.

Key	Values	Description
`:universal_newline`	`true` or `false`	When true, `\r\n` and `\r` are converted to `\n`.
`:xml`	`:text` or `:attr`	Replaces `&` with `&`, `<` with `<`, `>` with `>`, and undefined characters with a hexadecimal entity of the form `&#x`*hex*`;`, where *hex* is a sequence of hexadecimal digits. In addition, when a value of `:attr` is supplied, `"` is replaced with `&quot;`.

Encoding::Converter

The `Encoding::Converter` class provides additional control over the transcoding process. `Encoding::Converter.new` takes a source encoding as its first argument, and a destination encoding as its second. Both may be given as encoding names or `Encoding` objects. An options Hash may be supplied as a third argument.

Conversion Path

Text is transcoded along a *conversion path*. Each step involves a source encoding and a destination encoding. In the simple case, the conversion path will have only one step: from the given source encoding to the given destination encoding. However, more complex transcoding requires intermediate stages, e.g. to transcode Big5 into ISO-8859-9, we must first transcode to UTF-8: Big5 to UTF-8, then UTF-8 to ISO-8859-9. The source and destination encodings that are currently in use are returned by `Encoding::Converter#source_encoding` and `Encoding::Converter#destination_encoding`, respectively, as `Encoding` objects.

The various newline conversion options and those which perform escaping are termed *decorators*, and also feature in the conversion path. If the destination encoding is ASCII-compatible, they appear as the final steps, i.e. after any encoding pairs. Otherwise, they appear before the final step.

`Encoding::Converter#convpath` returns an `Array` of steps in the conversion path. Steps which convert between two encodings are represented as an

`Array` of the respective `Encoding` objects. A steps which applies a decorator appears as a `String` naming the decorator.

`Encoding::Converter.new` may be invoked with an `Array` in this form as an argument. The instantiated converter then uses this conversion path rather than inferring one from its arguments.

Piecemeal Conversion

An `Encoding::Converter` object can perform piecemeal transcoding, by repeatedly calling `Encoding::Converter#convert` with the next fragment of input. The fragment is transcoded and returned, associated with the destination encoding. However, because each fragment is always assumed to be part of a larger source, it may legitimately end mid-character, i.e. prior to a character boundary. These trailing bytes are buffered internally, and the successfully transcoded characters are returned. Then, when `#convert` is called next, its argument is assumed to supply the remaining bytes. If an unambiguously invalid byte sequence is encountered, an exception is raised.

Conceptually, we can explain this process as follows. When an `Encoding::Converter` instance is created, an empty *pending* buffer is created. Each time it is called, `#convert` initialises two empty buffers of its own: *source* and *destination*. It copies its argument into *source*, which it then processes byte-by-byte:

1. The byte is appended to *pending* and removed from *source*. The next action depends on the contents of *pending*:
    1. If it constitutes a valid character in the destination encoding, it is transcoded and written to *destination*. *pending* is emptied.
    2. If it constitutes a byte sequence that could be valid in the source encoding, but currently isn't, it is left in *pending* in the hope that the next call to `#convert` will supply the remaining bytes.
    3. If it is invalid in the source encoding, regardless of subsequent input, an `Encoding::InvalidByteSequenceError` exception is raised.

    4. If its valid in the source encoding, but a corresponding character does not exist in the destination encoding, an `Encoding::UndefinedConversionError` exception is raised.
2. When the source buffer is empty, the destination buffer is returned, then emptied.

Thus, after `#convert` returns *destination*, *pending* may not be empty. An implication is that a call to `#convert` may raise an exception because, when combined with the contents of *pending*, its argument was invalid. i.e. an exception may be raised even if the argument is in itself valid. Therefore, when there is no more text to transcode, `Encoding::Converter#finish` should be called to signal that the contents of *pending* should be transcoded and returned. If *pending* isn't empty when `#finish` is called, this normally results in one of the aforementioned exceptions being raised, because if its contents constitute a valid character, it would have already been returned by `#convert`. However, if the destination encoding is a stateful encoding such as ISO/IEC 2022, there may legitimately be bytes left in *pending*, which `#finish` flushes out. The lesson is that `#finish` should always be called when there is no more text to transcode.

Primitive Conversion

`Encoding::Converter#convert` is built atop `Encoding::Converter#primitive_convert`, which provides even more control over the process. Unlike `#convert`, the source and destination buffers must be specified explicitly: the former as the first argument, the latter as the second. Both should be `Strings` holding, respectively, the text to be transcoded, and the `String` in which to store the result. If the source buffer is an empty `String` it may be given as `nil`, instead. Neither buffer can be frozen, as they are, respectively, depleted and replenished in the course of the operation.

Bytes are written from the source buffer to the destination buffer via the pending buffer, as with `#convert`. However, this time instead of exceptions being raised for erroneous input, a `Symbol` is returned, as explained subsequently, which describes the problem. Thus, the programmer may elect to resolve the error before calling `#primitive_convert` again to resume the conversion. Due to the use of the pending buffer, `Encoding::Converter#finish` should still be used, as described previously.

By default, the destination buffer is appended to. If an `Integer` offset is given as the third argument to `#primitive_convert`, it specifies the byte index after which the transcoded text should be written. An `ArgumentError` is raised if the offset is given and greater than the byte size of the destination buffer. If this argument is specified as `nil`, the default behaviour is followed.

An optional fourth argument, given as an `Integer`, specifies the maximum size in bytes of the destination buffer; by default this value is `nil` which denotes an absence of a limit. If this limit is non-`nil` and the size of the destination buffer reaches it, transcoding will stop and `:destination_buffer_full` will be returned.

An optional fifth argument specifies one or both of the following options as a `Hash` or a bitwise OR of the corresponding constants:

Conversion Options

`after_output: true`
`Encoding::Converter::AFTER_OUTPUT`
: After writing a character to the destination buffer, stop, and return `:after_output`.

`partial_input: true`
`Encoding::Converter::PARTIAL_INPUT`
: The source buffer is known to be incomplete, i.e. it ends outside of a character boundary. If this option is given and the last byte(s) of the source buffer don't correspond to a character in the destination encoding, `:source_buffer_empty` is returned. This indicates that the remainder of the source text should be assigned to the source buffer, and `#primitive_convert` called again.

The return value is one of the following `Symbols`:

Return Values of #primitive_convert

`:invalid_byte_sequence`
: The source buffer contains a byte sequence invalid in the destination encoding, regardless of any following bytes. Equivalent to the `Encoding::InvalidByteSequenceError` exception being raised.

`:incomplete_input`
: The source buffer ends prematurely, presumably prior to a character boundary, but is potentially valid if additional input is supplied. Nevertheless, this state is regarded as exceptional, equivalent to an `Encoding::InvalidByteSequenceError` being raised, because the `:partial_input` option is `false`. If, as expected, no more input is supplied, the result will end with an invalid byte sequence. Conversely, if `:partial_input` was `true`, the unexceptional `:source_buffer_empty` `Symbol` would be returned instead.

`:undefined_conversion`
: A character has been encountered in the source buffer which, although legal in the source encoding, has no equivalent in the destination encoding. Equivalent to an `Encoding::UndefinedConversionError` exception being raised.

`:after_output`
: If the `:after_output` option is given, after each character is converted this `Symbol` is returned.

`:destination_buffer_full`
: If a non-`nil` value has been given for the fourth argument, this `Symbol` indicates that the destination buffer has reached the given limit.

`:source_buffer_empty`
: The source buffer ends prematurely, presumably prior to a character boundary, and the `:partial_input` option has been given. The source buffer should be replenished and transcoding resumed.

`:finished`
: Conversion is finished, either naturally or because `Encoding::Converter#finish` has been called.

Error Context

When an error occurs during transcoding, it is often necessary to understand its context so as to recover. The exceptions raised by `#convert`, are augmented with accessors for gleaning this information. Additionally, `Encoding::Converter#primitive_errinfo` provides detailed information about the last error in the form of an `Array` with the following elements, in this order:

1. The `Symbol` that `#primitive_convert` would have returned in this situation, even if `#convert` was the actual method used. The six possible values were described above. If this element is `:after_output`, `:destination_buffer_full`, `:source_buffer_empty`, or `:finished`, all remaining elements are `nil`.
2. The source encoding as a `String`. This may not be the given source encoding if the conversion path has multiple steps. Equivalent to the `#source_encoding_name` methods of `Encoding::UndefinedConversionError` and `Encoding::InvalidByteSequenceError`.
3. The destination encoding as a `String`. This may not be the given destination encoding if the conversion path has multiple steps. Equivalent to the `#destination_encoding_name` methods of `Encoding::UndefinedConversionError` and `Encoding::InvalidByteSequenceError`.
4. The problematic bytes as a `String`: the bytes prior to the invalid byte for `:invalid_byte_sequence`, the errant character for `:undefined_conversion`, and the bytes read since the last character for `:incomplete_input`. Equivalent to `Encoding::UndefinedConversionError#error_char` and `Encoding::InvalidByteSequenceError#error_bytes`.
5. For `:invalid_byte_sequence`, this element holds the byte that rendered the sequence invalid. i.e. the preceding element held the bytes which could legitimately form a valid sequence, and this element holds the byte which invalidated it. This is equivalent to `Encoding::InvalidByteSequenceError#readagain_bytes`. For `:undefined_conversion` and `:incomplete_input`, this element is `""`.

So, error context can be ascertained, *ex post facto*, by using `#convert`, then rescuing and examining the exceptions it raises, or by using either transcoding method and inspecting elements of the `#primitive_errinfoArray`. A third approach is to use `#last_error`, which returns the exception `#convert` would have raised, or otherwise `nil`. This saves one from having to rescue exceptions or splice an `Array`, if his interest is purely the exceptional situations. It also suggests a useful idiom for dealing with errors: it is true the error indicates malformed input; otherwise, it relates to the configuration of the transcoder.

Recall, however, that the states `:incomplete_input` and `:invalid_byte_sequence` both indicate that an invalid byte sequence has been discovered-the former is simply optimistic that the next input will correct it-so both have `Encoding::InvalidByteSequenceError` as their exception. They can be distinguished with the `Encoding::InvalidByteSequenceError#incomplete_input?` predicate, which only returns `true` in the former case.

Recovery from an Invalid Byte Sequence

When an invalid byte sequence is encountered, the transcoder saves two sets of bytes: the *error bytes* and the *read-again bytes*. The former are always discarded: they are, independently of context, invalid in the destination encoding. The latter are the buffered bytes that followed. If this error is simply ignored, and the call to `#primitive_convert` repeated, the read-again bytes are returned to the source buffer, and considered again. If this new byte sequence is valid in the destination encoding, it will be transcoded as normal. Otherwise, this process will be repeated. The outcome is almost certainly undesirable. Invalid byte sequences are split, and some of their constituent bytes are interpreted as characters in their own right, and appended to the destination buffer.

```
source, dest = "a\u6543bc", ""
source.setbyte(2,0)
ec = Encoding::Converter.new('utf-8', 'ascii')
until source.empty?
  ec.primitive_convert(source, dest)
  p [source, dest, *ec.primitive_errinfo]
end
```

```
#=> ["\x83bc", "a", :invalid_byte_sequence, "UTF-8", "US-ASCII", "\xE6", "\x00"]
#=> ["bc", "a\x00", :invalid_byte_sequence, "UTF-8", "US-ASCII", "\x83", ""]
#=> ["", "a\x00bc", :finished, nil, nil, nil, nil]
```

The safer option is to drop the read-again bytes, too, with `Encoding::Converter#putback`. Optionally, the maximum number of bytes to put back can be given as an `Integer` argument. To contrast the two approaches, juxtapose the previous example with the following, taking care to note the final contents of *dest*.

```
source, dest = "a\u6543bc", ""
source.setbyte(2,0)
ec = Encoding::Converter.new('utf-8', 'ascii')
until source.empty?
  state = ec.primitive_convert(source, dest)
  p [source, dest, *ec.primitive_errinfo]
  ec.putback if state == :invalid_byte_sequence
end
#=> ["\x83bc", "a", :invalid_byte_sequence, "UTF-8", "US-ASCII", "\xE6", "\x00"]
#=> ["bc", "a", :invalid_byte_sequence, "UTF-8", "US-ASCII", "\x83", ""]
#=> ["", "abc", :finished, nil, nil, nil, nil]
```

Recovery from an Undefined Conversion Error

Again, this error can be ignored by simply repeating the call to `#primitive_convert`. The unknown character will be dropped, and transcoding will continue. A more refined approach is to substitute the unknown character for another: either a constant, such as `"?"`, or an approximation determined out-of-band. To support both scenarios, `Encoding::Converter#insert_output` accepts an arbitrary `String` argument, which it appends to the source buffer on the next call to `#primitive_convert`. Accordingly, the `String` will be transcoded into the target encoding and appended to the destination buffer.

```
source, dest = "a\ubeefbc", ""
ec = Encoding::Converter.new('utf-8', 'ascii')
until source.empty?
  if ec.primitive_convert(source, dest) == :undefined_conversion
    ec.insert_output("<U+%.4X>" % ec.last_error.error_char.ord)
  end
```

```
  p [source, dest, *ec.primitive_errinfo]
end
#=> ["bc", "a", :undefined_conversion, "UTF-8", "US-ASCII", "\xEB\xBB\xAF", ""]
#=> ["", "a<U+BEEF>bc", :finished, nil, nil, nil, nil]
```

REGEXPS

A Regexp represents a regular expression: a pattern that describes a String. If a String contains the pattern described by a given regular expression, it is said to *match*. Therefore, one use of regular expressions is validation: testing whether a String matches a pattern.

```
# Does a String contain a digit?
/\d/ =~ 'two: 2' #=> 5
# (Yes, starting at this fifth character)
/\d/ =~ 'Nope' #=> nil
# (No)

# Which Strings in an Array contain 'cat' case-insensitively?
%w{dogma verification wildcat dogfish medicate underdog Catholicism}.grep /cat/i
#=> ["verification", "wildcat", "medicate", "Catholicism"]
```

Another, introduced in the Strings chapter, is extracting the portions of a String that match a certain pattern.

```
# Extract sequences of consecutive digits
'one: 1, ten: 10, one-hundred: 100'.scan(/\d+/)
#=> ["1", "10", "100"]

# Separate a String into substrings separated by ', '
'Asia, Africa, North America, South America, Antarctica, Europe, Australia'.
  split(/, /)
#=> ["Asia", "Africa", "North America", "South America",
#    "Antarctica", "Europe", "Australia"]
```

Similarly, areas of the pattern can be designated as capturing, which effectively labels parts of the String so they can be referred to after the match.

```
time = /\A(?<hours>(0\d|1[0-9]|2[0-3])):(?<minutes>([1-5]\d|0\d))\Z/
match = time.match '11:30'
match[:hours] #=> "11"
match[:minutes] #=> "30"
```

Further examples of the utility of regular expressions can be found in the Strings chapter. In this chapter, we deal primarily with the syntax of patterns: how to construct a regular expression that matches precisely what is needed.

Literals

The literal is of the form `/`*`pattern`*`/`*`options`*: a pattern delimited by solidi, followed by zero or more single-character option specifiers. If the pattern is to contain either of these delimiters literally, they must be escaped with a reverse solidus. The construct `%r{`*`pattern`*`}`*`options* also constructs a `Regexp`, but in this form *pattern* may contain either solidi or reverse solidi literally, without having to escape them.

```
/pat/i =~ 'Pattern'
/1\/2/ =~ '1/2'
%r{1/2} =~ '1/2'
```

Options

The behaviour of a `Regexp` can be modified by following the literal with one or more of the following option specifiers:

Option	Effect
`e`	Associate the pattern with the EUC-JP encoding
`i`	Ignore case
`m`	Let `.` match newline characters.
`n`	Associate the pattern with the ASCII-8BIT encoding.
`o`	Only interpolate `#{…}` constructs the first time this literal is parsed.
`s`	Associate the pattern with the Windows-31J encoding.
`u`	Associate the pattern with the UTF-8 encoding.
`x`	Enable free-spacing mode.

```
3.times.map{|n| /#{n}/}       #=> [/0/, /1/, /2/]
3.times.map{|n| /#{n}/o}      #=> [/0/, /0/, /0/]
/case/i =~ 'Case'             #=> 0
/Case/i =~ 'cAsE'             #=> 0
/.../m =~ "a\nb"              #=> 0
```

```
/f r e e # comment
-spac ing/x =~ 'free-spacing' #=> 0
/EUC-JP/e.encoding            #=> #<Encoding:EUC-JP>
```

Options `i`, `m`, and `x`, may also be applied to a specific group rather than the pattern as a whole, with the syntax described in Grouping.

Matching

A `Regexp` may be matched against a `String` by supplying the latter as an argument to `Regexp#match` or `Regexp#=~`. `String#match` and `String#=~` behave in the same way, *mutatis mutandis*.

The `#match` methods return MatchData objects if the match succeeded, whereas the `#=~` methods return the character offset in the `String` where the match began. They all return `nil` if the `String` didn't match.

```
'23' =~ /a/                          #=> nil
/\A0x[[:xdigit:]]+\Z/i =~ '0XA55E'   #=> 0
/\D/ =~ '2 + 3'                      #=> 1
'33 + 41'.match /(\d+) [-+] (\d+)/   #=> #<MatchData "33 + 41" 1:"33" 2:"41">
/\./.match ""                        #=> nil
```

Metacharacters

In the context of a pattern, a character is either interpreted literally or as a *metacharacter*. A literal character matches itself, whereas a metacharacter has another meaning. To force a metacharacter to be interpreted literally it must be preceded by a reverse solidus (\).

Metacharacter	Meaning
^	Start of line anchor
$	End of line anchor
#	Introduces a comment if the `x` option is given
#{...}	Interpolates an expression
(...)	Encloses a group

Metacharacter	Meaning
*	Greedy quantifier: preceding atom may occur any number of times
+	Greedy quantifier: preceding atom occurs at least once
-	Separates a range inside a character class
.	Matches almost any character
?	Greedy quantifier: preceding atom occurs 0 or 1 times
[...]	Encloses a character class
/	Escapes the character that follows
{...}	Interval
|	Alternation

Escapes

In addition to to the metacharacter escapes already mentioned, a pattern may also contain String escapes, as well as those summarised in the following table:

Escape	Meaning
\1–\9	Back-reference to a numbered group
\A	Start of `String` anchor
\b	Word boundary outside of character class; backspace, otherwise.
\B	*Not* `\b`
\d	Decimal digit: 0–9
\D	*Not* `\d`
\g<*name*>	Sub-expression call for *name*
\G	Start of match or end of previous match
\h	Hexadecimal digit
\H	Not `\h`
\k<*name*>	Back-reference to group named or numbered *name*
\p{name}	A character with the Unicode property*name*
\P{name}	A character without the Unicode property*name*
\s	Whitespace
\S	*Not* `\s`
\w	Word character
\W	*Not* `\w`
\Z	End of `String` or before `String`-ending newline

Escape	Meaning
\z	Absolute end-of `String` anchor

Grouping

A balanced pair of parentheses are meta-characters which group and/or capture the characters they enclose. Grouping allows the enclosed to be treated as an atomic whole such that meta-characters directly following the closing parenthesis act on the whole.

It also allows the `i`, `m`, and `x`option specifiers to be applied to a specific group, rather than the pattern as a whole. The opening parenthesis is immediately followed by `?`*`options`*`:`, where *options* is one or more of the aforementioned option specifiers. If *options* is omitted, i.e. a group begins with `(?:`, it is non-capturing, as explained in the following section.

```
%w{CASE case CAse caSE cASe casE}.grep /(?i:ca)se/
#=> ["case", "CAse"]
%W{fin\n fi\n\n \n\n fin\r\n find}.grep /f.{2}(?m:.)\Z/
#=> ["fin\n", "fin\r\n", "find"]
/space(?x: m
           a
           n # or 'woman'...
      ) #\d+/ =~ 'spaceman #47'
#=> 0
/x(?xi:
 i)/ =~ 'xI' #=> 0
```

Capturing

Capturing is the extraction of specific parts of the text being matched so it can be referred to later, from within the pattern or the surrounding program. As explained above, a group captures unless its opening parenthesis is directly followed by `?`*`options`*`:`, where *options* is zero or more of the specifiers `i`, `m`, or `x`.

A capturing group can be referred to later in the pattern by means of a *back-reference*. The back-reference `\d`, where *d* is a decimal digit between one and nine, refers to the d^{th} capturing group. Captures can be referred to by name if the opening parenthesis is followed by `?<name>`. The back-reference `\k<name>` refers to the capturing group named *name*. If *name* is a decimal number, it refers to the corresponding numbered capture group, enabling numbered back-references to be used even when there are more than nine capture groups. However, named capture groups and numbered capture groups cannot both be used in the same pattern.

The text captured by a specific capture group can be retrieved after the match from the corresponding MatchData object. The text captured by a numbered group in the last match is also available in the global variable `$d`, where *d* is the group number between one and nine.

```
/(.)\1/ =~ 'UU' #=> 0
$1 #=> 'U'
%w{emaciate usurious enyzyme fists overdo unkind}.
  grep(/\A(?<vowel>[aeiou])(?<consonant>[^aeiou])\w+\k<vowel>\Z/)
#=> ["emaciate", "enyzyme", "overdo"]
pattern = %r{\A(?<scheme>[a-z]+)://(?<host>[^/]+)(?::\d+)}
match = pattern.match 'http://example.ORG:80/'
#=> #<MatchData "http://example.ORG:80" scheme:"http" host:"example.ORG">
match[:host] #=> "example.ORG"
$1 #= "http"
$~[:scheme] #=> "http"
```

If a `Regexp` literal is successfully matched against a `String` with `Regexp#=~`, a local variable will be initialized for each named group to hold its captured text.

```
phrase = 'Aut disce aut discede'
/(?<either>\w+) (?<learn>\w+) (?<_or>\k<either>) (?<leave>\w+)/i =~ phrase
[either, learn, _or, leave] #=> ["Aut", "disce", "aut", "discede"]
/\A(ab|ex) (?<one>uno) (?<learn>#{learn})/ =~ 'Ex uno disce omnes' #=> nil
learn #=> "disce"
defined?(one) #=> nil
'Aut...' =~ /Aut(?<ellipsis>\.{3})/ #=> 0
defined?(ellipsis) #=> nil
```

Quantifiers

A literal character, group, or character class may be followed by a *quantifier* meta-character which specifies how many consecutive occurrences are required at that point in the pattern. + requires at least one; ?, zero or one; and *, zero or more. Alternatively, an *interval* may be given, enclosed in a pair of curly brackets. An interval of the form {*n*} requires exactly *n* occurrences; {*n*,}, at least *n*; and {,*n*}, at most *n*. {*min*,*max*} requires between *min* and *max* occurrences.

```
%w{0bB0 0b1111010 0b(0|1)+ 0B1}.grep /\A0[bB](0|1)+\Z/
#=> ["0b1111010", "0B1"]
%w{+Infinity -Infinity Infinity NaN -NaN}.grep /\A(([+-]?Infinity)|NaN)\Z/
#=> ["+Infinity", "-Infinity", "Infinity", "NaN"]
%w{0x 0xfeed food 0xae!}.grep /\A0x[[:xdigit:]]*\Z/
#=> ["0x", "0xfeed"]
%w{0 01 08 o 065 0123 051171 082 0o0}.grep /\A0[0-7]{2,4}\Z/
#=> ["065", "0123"]
%w{NaN+ a4 00.0 3. 2\n.7 07 42 -Infinity +34.21 -0.54 1.23232 Infinity.2}.grep \
  /\A(NaN| # The string 'NaN', OR
      [-+]?( # An optional sign, then

        Infinity| # The string 'Infinity', OR

        ([1-9]\d+|\d) # A non-zero digit followed by one or more digits, OR
                      # a single digit by itself, then
          (\.\d{1,4})?  # optionally, a literal '.' followed by 1-4 digits
      )
     )\Z/x #=> ["42", "-Infinity", "+34.21", "-0.54"]
```

The above quantifiers are termed *greedy* because they consume as many characters as possible; in contrast, a *lazy* quantifier consumes as few characters as possible. The +, ?, and * quantifiers are made lazy when immediately followed by a question mark.

```
str = 'indolent poltroon!'
str.match /ro?/                #=> #<MatchData "ro">
str.match /ro??/               #=> #<MatchData "r">
str.match /n*t/                #=> #<MatchData "nt">
str.match /n*?t/               #=> #<MatchData "nt">
```

```
str.match /i.+t/                #=> #<MatchData "indolent polt">
str.gsub(/i.+?t/, 'gluttonous') #=> "gluttonous poltroon!"
```

Whether greedy or lazy quantifiers are used, Ruby tries all permutations of a pattern—a process known as *backtracking*—before declaring that it does not match the input. If a greedy quantifier is immediately followed by a plus sign—e.g., `*` becomes `*+`—it becomes *possessive*. Possessive quantifiers refuse to give up a partial match when backtracking. When they've found a match, they don't let it go, even if this causes the match as a whole to fail. This is primarily useful for performance reasons, as it avoids backtracking that is known, a priori, to be redundant. However, they may also change the results of a match that depends upon a greedy quantifier backtracking. Accordingly, possessive quantifiers are only applicable when the atom they quantify should not match the atom that follows.

```
 '123' =~ /^\d(2|4|6|8)?\d\d/                  #=> 0
'123' =~ /^\d(2|4|6|8)?+\d\d/                  #=> nil
'The "programmer" said...' =~ /"[[:graph:]]*"/  #=> 4
'The "programmer" said...' =~ /"[[:graph:]]*+"/ #=> nil
```

Character Classes

A character class specifies a set of characters, one of which must appear at that point in the pattern, enclosed within a pair of square brackets. The characters are specified literally, one after the other. A range of consecutive characters may be abbreviated with the notation *`start-end`*: the first character in the range separated from the last by a hyphen minus sign (`U+002D`). If the first character after the left square bracket is a circumflex accent (`U+005E`), the class is inverted: it matches any character except those listed. Inside a character class, therefore, the hyphen-minus sign and circumflex accent are metacharacters, so to be matched literally, the former must appear directly before the right square bracket, and the latter at any position other than the first. Alternatively, either can be escaped with a reverse solidus.

The following predefined character classes are also available. They are specified as `[:`*`name`*`:]`, and must appear within another character class. For

example, `[[:alpha:]]` represents alphabetical characters, and `[[:alpha:]2-4]` represents alphabetical characters and the digits *2*, *3*, and *4*.

`[[:alnum:]]`
: Characters with the Unicode properties *Alphabetic* or *Decimal Number*

`[[:alpha:]]`
: Characters with the Unicode property *Alphabetic*

`[[:blank:]]`
: Characters with the Unicode property *White Space*, excluding:
  - Characters with the Unicode properties *Line Separator* or *Paragraph Separator*
  - *Line Feed (LF)* (`U+000A`)
  - *Line Tabulation* (`U+000B`)
  - *Form Feed (FF)* (`U+000C`)
  - *Carriage Return (CR)* (`U+000D`)
  - *Next Line (NEL)* (`U+0085`)

`[[:cntrl:]]`
: Characters with the Unicode General Category *Cc* (*Control*)

`[[:digit:]]`
: Characters with the Unicode property *Decimal Number*

`[[:graph:]]`
: Any character except for characters...:
  - with the Unicode property *White Space*.
  - with the Unicode general categories *Cc* (*Control*), *Cs* (*Surrogate*), or *Cn* (*Unassigned*).

`[[:lower:]]`
: Characters with the Unicode property *Lowercase*

`[[:print:]]`
: Characters represented by `[[:graph:]]` or `[[:blank:]]`, excluding those represented by `[[:cntrl:]]`.

`[[:punct:]]`
: Characters with the Unicode property *Punctuation*

`[[:space:]]`
: Characters with the Unicode property *White Space*.

`[[:upper:]]`
: Characters with the Unicode property *Uppercase*

`[[:xdigit:]]`
: Digit allowed in a hexadecimal number (i.e., 0-9a-fA-F)

`[[:word:]]`
: Characters with…
  - the Unicode properties *Alphabetic* or *Decimal Number*.
  - the Unicode general categories *M* (*Mark*) or *Pc* (*Connector Punctuation*).

`[[:ascii:]]`
: A character in the ASCII character set, i.e. 〈U＋0000, U＋007F〉

```
# coding: utf-8
/\A[^1-9][xX][[:xdigit:]][[:space:]] = [[:digit:]]{2}\Z/ =~ '0xf = 15'
chars = (200..3000).map{|c| c.chr 'utf-8'}
chars.grep(/[[:digit:]]/).first(10)
#=> ["٭", "۱", "۲", "۳", "۴", "۵", "۶", "۷", "۸", "۹"]
chars.grep(/[[:upper:]]/).first(10)
#=> ["È", "É", "Ê", "Ë", "Ì", "Í", "Î", "Ï", "Ð", "Ñ"]
chars.grep(/[[:lower:]]/).first(10)
#=> ["ß", "à", "á", "â", "ã", "ä", "å", "æ", "ç", "è"]
```

Alternation

A vertical line is a meta-character specifying that either the expression to its right, or that to its left, must match. It is usually used inside a parenthetical.

```
%w{cat ls catls cats ca}.grep /\A(cat|ls)\Z/
#=> ["cat", "ls"]
```

```
(%w{cat ls(1) echo(1) cats} << '').grep /\A((cat|ls)\(\d+\)|echo|)\Z/
#=> ["ls(1)", ""]
%w{xx xy yX yy yz zz xixi xz}.grep /\A(x(?i:x|y)|y(?i:x|y)|z)\Z/
#=> ["xx", "xy", "yX", "yy"]
```

MatchData

A `MatchData` object encapsulates information about a match, providing access to the captures and matched text. It is returned by `Regexp#match` and `String#match`, and the `MatchData` object corresponding to the last match is available as `$~`.

`MatchData#regexp` returns the `Regexp` object used in this match, and `MatchData#string` returns a frozen copy of the `String` it was tested against. `MatchData#to_s` returns the portion of `MatchData#string` that matched `MatchData#regexp`.

`MatchData#[`*capture*`]` returns the text captured by the capture group, *capture*, given as a `Symbol`, for a named group, or an `Integer` for a numbered group. An argument of `0` returns the entire matched text, as does `$&`. `MatchData#names` returns an `Array` of `Symbols`, each of which is a name of a named capture group.

`MatchData#begin(`*capture*`)` and `MatchData#end(`*capture*`)` return the offset in the matched `String` of the begining and end, respectively, of the given capture. `Match#offset` returns a two-element `Array` containing the begining and ending offsets of the given capture. As above, *capture* is either a `Symbol` corresponding to a named capture, or an `Integer` corresponding to a numbered capture.

`MatchData#captures` and `MatchData#to_a` return an `Array` of the contents of each capture group. The first element of the former is the first captured group; the first element of the latter is the entire matched text. The last element, i.e. the contents of the group captured last, is also available as `$+`. `MatchData#values_at` takes one or more `Integer` arguments and returns an `Array` containing the elements of `MatchData#to_a` with the given indices. `MatchData#size`, and its alias `MatchData#length`, return the number of elements in `MatchData#to_a`.

`MatchData#pre_match`, and its alias `$`` ` ``, and `MatchData#post_match`, and its alias `$'`, return the portion of the portion of the `String` preceeding and following, respectively, this match.

Anchoring

A regular expression matches a `String` if the former is *contained* in the latter. For example, `/\d/` matches `"2"`, as well as `"2 by 4"` and `"DoB: 19/2/1922"`. Alternatively, a pattern may be *anchored* to a specific portion of the `String`. Whereas many of the metacharacters introduced so far match sequences of characters, anchors match positions. They are not recognised inside of character classes.

Anchor	Position in `String`
`^`	Start or after newline
`\A`	Start
`$`	End or before newline
`\Z`	End or before last newline
`\z`	End
`\b`	Word boundary
`\B`	Non-word boundary
`\G`	Point where last match finished

The *word boundary* referred to in the table above is a position before a word, where *word* is simply: `/[[:word:]]+/`. Therefore, `\b` matches a word character that is not preceded by a word character. This is quite different from `/[^[:word:]][[:word:]]/` because `\b` matches a position without consuming any characters. Indeed, the pathological pattern `/\b/` matches `"a"` at position 0, i.e. the non-word character need not even exist.

```
# coding: utf-8
str = "Hit him on the head\n" +
      "Hit him on the head with a 2×4\n"
str.scan(/^Hit/)     #=> ["Hit", "Hit"]
str.scan(/\AHit/)    #=> ["Hit"]
str =~/head$/        #=> 15
str.scan(/\d\Z/)     #=> ["4"]
str =~ /\d\z/        #=> nil
```

```
str.scan(/\b\d/)     #=> ["2", "4"]
str.scan(/\w+\B/)
#=> ["Hi", "hi", "o", "th", "hea", "Hi", "hi", "o", "th", "hea", "wit"]
```

Zero-Width Assertions

Inasmuch as anchors never consume any characters, they are a variety of a more general concept: the *zero-width assertion*. The latter require a given sub-expression to appear, or not appear, in the position preceding, or following, the assertion. That is, they vary across two axis as illustrated below.

Syntax	Name	Assertion
(?=*exp*)	Positive look-ahead	*exp* must follow
(?!*exp*)	Negative look-ahead	*exp* can't follow
(?<=*exp*)	Positive look-behind	*exp* must precede
(?<!*exp*)	Negative look-behind	*exp* can't precede

```
"foresight".sub(/(?!s)ight/, 'ee')          #=> "foresee"
"anterior".sub(/(?<!eleph)an(?=t)/, 'pos')  #=> "posterior"
%w{An abbess abjures an oblate
   for his absurd abacus}.grep /\A.b(?![four]).{4}(?!i?e)\z/
#=> ["abbess", "oblate", "absurd", "abacus"]
```

Readability

Longer patterns tend to be harder to comprehend, increasing the likelihood of errors. This can be avoided by adhering the following principles:

Named groups are self-documenting
: By using named groups instead of numbered groups, a capture describes its purpose.

Free-spacing mode allows complex patterns to be formatted clearly
: The x option causes literal whitespace and comments to be ignored, allowing a pattern to be commented and laid out over multiple lines.

Outside of free-spacing mode, comments are still useful
: The contents of a group beginning `(?#` are ignored, allowing comments to be interspersed with the pattern.

Long patterns can be built from smaller parts with interpolation
: When working with particularly complicated patterns, consider constructing independent regular expressions for each sub-expression, then interpolating them into one large pattern.

```
# coding: utf-8
# A naïve pattern to match a human name
title = /(?:(?<title>Mrs?|M(?:aster|s)|Dr|Sir)[[:blank:]])?/
name_part = /[[:upper:]][[:alpha:]-]+ # Double-barrelled names are allowed, but
                                      # names can't start or end with hyphens
             [[:alpha:]] # i.e. names must contain at least three characters
            /x
name_mid_part = /(?:
  (?:
    (?:[[:lower:]]{1,3})| # e.g. 'von', 'y', or 'de'
    (?:[[:upper:]]\.)   | # middle initial
    (?:#{name_part})
  )[[:blank:]])/x
name = /(?<name>#{name_part}[[:blank:]]   # A forename
         #{name_mid_part}* # Any number of middle names
         #{name_part} # A surname or family name
        )/x
suffix = /(?:[[:blank:]](?<suffix>[JS]r\.|[IVX]+(?# Roman numerals)))?/
full_name = /\A#{title}#{name}#{suffix}\Z/
['Mr Harvey Duchamp II', 'Dr Ludwig von Mises', 'William S. Burroughs',
 'Ms Henrietta Cartwright-Stevens', 'Paul Erdős', 'Anonymous', '2 by 4',
 'Master Elijah Humphrey Pennington Hargreeves Jr.'].grep full_name
#=> ["Mr Harvey Duchamp II", "Dr Ludwig von Mises", "William S. Burroughs",
#    "Ms Henrietta Cartwright-Stevens", "Paul Erdős",
#    "Master Elijah Humphrey Pennington Hargreeves Jr."]
```

Encoding

A `Regexp` is associated with an encoding which must be compatible with that of the `String` it is matched against. By default a `Regexp` has the same

encoding as the source file in which it is contained, with the following exceptions:

- An ASCII-only Regexp has the encoding US-ASCII when the source encoding is ASCII-compatible.
- An encoding option has been specified, in which case the corresponding encoding is used, as follows:

  `u`
  : UTF-8

  `e`
  : EUC-JP

  `s`
  : Windows-31J

  `n`
  : ASCII-8BIT

- The use of Unicode character escapes within a pattern force the `Regexp` to have UTF-8 encoding.

In addition to the single-letter encoding options described above, a `Regexp` can be encoded in any supporting encoding. This requires the `Regexp` to be constructed with `Regexp.new`, passing it the pattern as a `String` associated with the desired encoding. The same technique, *mutatis mutandis*, can be used with `String#force_encoding` to associate the `Regexp` with a different encoding.

```
# coding: utf-8
utf8 = /caf(e|é)/
utf8.encoding          #=> #<Encoding:UTF-8>
iso88591_a = Regexp.new('caf(e|é)'.encode('iso-8859-1'))
iso88591_b = Regexp.new(utf8.source.encode('iso-8859-1'))
iso88591_a == iso88591_b #=> true
iso88591_a.encoding      #=> #<Encoding:ISO-8859-1>
```

Fixed Encoding

The encoding of a `Regexp` is said to be *fixed* if its encoding and/or pattern is incompatible with ASCII. This is significant because a `Regexp` with a non-fixed encoding can match any `String` whose encoding is ASCII-compatible. The `Regexp#fixed_encoding?` predicate returns `true` if its receiver's encoding is fixed; `false` otherwise.

Character Properties

A generalisation of predefined character classes is the character property escape. The construct `\p{`*property*`}` represents characters with the property *property*; while the construct `\P{`*property*`}` represents its inverse. The encoding of a pattern dictates the property escapes it may use. In all encodings *property* may be the name of a predefined character class: Alnum, Alpha, ASCII, Blank, Cntrl, Digit, Graph, Lower, Print, Punct, Space, Upper, Word, and XDigit.

Further, in *Shift JIS* and *EUC-JP* encodings, the properties *Katakana* and *Hiragana* are available to match characters in the named script. In Unicode encodings, all properties are available and *property* is normalised by ignoring case[1], spaces, and low line characters. For example, in a Unicode pattern `\p{Lowercase_Letter}`, `\p{lowercase letter}`, and `\p{lowercaseletter}`, are all equivalent.

The majority of the remaining property names correspond to Unicode properties, but Ruby also defines the following:

Newline
: Comprises solely of `"\n"` (`U+000A`).

Any
: Any Unicode character: 〈`U+0000`, `U+10FFFF`〉.

1. As of Ruby 1.9.3, *property* is case-insensitive for all encodings if it's the name of a predefined character class.

Assigned

Equivalent to `/[\p{Any}\P{Cn}]/`, i.e. any character that has been assigned a codepoint.

General Categories

The Unicode general categories, specified either by abbreviation or long name, are all valid properties. They represent all characters assigned the given category. If *property* comprises only a single character, it represents all general categories whose abbreviations begin with that character. For example, `\p{Lu}` and `\p{Uppercase Letter}` are equivalent, while `\p{L}` represents characters from categories *Lu*, *Ll*, *Lt*, *Lm*, and *Lo*.

Abbreviation	Long Name	Description
Lu	Uppercase_Letter	Uppercase letter
Ll	Lowercase_Letter	Lowercase letter
Lt	Titlecase_Letter	Digraphic character, with first part uppercase
Lm	Modifier_Letter	Modifier letter
Lo	Other_Letter	Remaining letters, e.g. syllables and ideographs
Mn	Nonspacing_Mark	Non-spacing combining mark (zero advance width)
Mc	Spacing_Mark	Spacing, combining mark (positive advance width)
Me	Enclosing_Mark	Enclosing combining mark
Nd	Decimal_Number	Decimal digit
Nl	Letter_Number	Letter-like numeric character
No	Other_Number	Another type of numeric character
Pc	Connector_Punctuation	Connecting punctuation mark
Pd	Dash_Punctuation	Dash or hyphen punctuation mark
Ps	Open_Punctuation	Opening punctuation mark (of a pair)
Pe	Close_Punctuation	Closing punctuation mark (of a pair)
Pi	Initial_Punctuation	Initial quotation mark
Pf	Final_Punctuation	Final quotation mark
Po	Other_Punctuation	Another type of punctuation mark

Abbreviation	Long Name	Description
Sm	Math_Symbol	Mathematical symbol
Sc	Currency_Symbol	Currency sign
Sk	Modifier_Symbol	Non-letter-like modifier symbol
So	Other_Symbol	Another type of symbol
Zs	Space_Separator	Space character (of non-zero width)
Zl	Line_Separator	Line separator (`U+2028`)
Zp	Paragraph_Separator	Paragraph separator (`U+2029`)
Cc	Control	A C0 or C1 control code
Cf	Format	Format control character
Cs	Surrogate	Surrogate code point
Co	Private_Use	Private-use character
Cn	Unassigned	Reserved, unassigned code point or a non-character codepoint

Simple Properties

A Unicode simple-i.e. non-derived-property is any of the following, and represents all characters with that property:

ASCII Hex Digit
: "ASCII characters commonly used for the representation of hexadecimal numbers." [Uax44])

Bidi Control
: "Format control characters which have specific functions in the Unicode Bidirectional Algorithm." (*ibid.*)

Dash
: "Punctuation characters designated as dashes in the Unicode Standard, plus their compatibility equivalents." (*ibid.*)

Deprecated
: "Unicode characters whose use is strongly discouraged." (*ibid.*)

Diacritic
: "Characters that linguistically modify the meaning of another character to which they apply." (*ibid.*)

Extender

"Characters whose principal function is to extend the value or shape of a preceding alphabetic character." (*ibid.*)

Hex Digit

"Characters commonly used for the representation of hexadecimal numbers, plus their compatibility equivalents." (*ibid.*)

Hyphen

"Dashes which are used to mark connections between pieces of words, plus the Katakana middle dot." (*ibid.*)

IDS Binary Operator
IDS Trinary Operator

"Used in Ideographic Description Sequences." (*ibid.*)

Ideographic

"Characters considered to be CJKV (Chinese, Japanese, Korean, and Vietnamese) ideographs." (*ibid.*)

Join Control

"Format control characters which have specific functions for control of cursive joining and ligation." (*ibid.*)

Logical Order Exception

"There are a small number of characters that do not use logical order. These characters require special handling in most processing." (*ibid.*)

NonCharacter Code Point

"Code points permanently reserved for internal use." (*ibid.*)

Other Alphabetic

"Used in deriving the *Alphabetic* property." (*ibid.*)

Other Default Ignorable Code Point

"Used in deriving the *Default Ignorable Code Point* property." (*ibid.*)

Other Grapheme Extend

"Used in deriving the *Grapheme Extend* property." (*ibid.*)

Other ID Continue
"Used for backward compatibility of *ID Continue*." (*ibid.*)

Other ID Start
"Used for backward compatibility of *ID Start*." (*ibid.*)

Other Lowercase
"Used in deriving the *Lowercase* property." (*ibid.*)

Other Math
"Used in deriving the *Math* property." (*ibid.*)

Other Uppercase
"Used in deriving the *Uppercase* property." (*ibid.*)

Pattern Syntax
Pattern White Space
"Used for pattern syntax as described in UAX #31: Unicode Identifier and Pattern Syntax" (*ibid.*)

Quotation Mark
"Punctuation characters that function as quotation marks." (*ibid.*)

Radical
"Used in Ideographic Description Sequences." (*ibid.*)

STerm
"Sentence Terminal. Used in UAX #29: Unicode Text Segmentation" (*ibid.*)

Soft Dotted
"Characters with a "soft dot", like i or j. An accent placed on these characters causes the dot to disappear." (*ibid.*)

Terminal Punctuation
"Punctuation characters that generally mark the end of textual units." (*ibid.*)

Unified Ideograph
"Used in Ideographic Description Sequences." (*ibid.*)

Variation Selector
: "Indicates characters that are Variation Selectors. See UAX #37: Unicode Ideographic Variation Database for more details." (*ibid.*)

White Space
: "Separator characters and control characters which should be treated by programming languages as "white space" for the purpose of parsing elements." (*ibid.*)

Derived Properties

The Unicode derived properties are defined by reference to simple properties and general categories, and are all valid property names. They are as follows:

Math
: Equivalent to `/[\p{Sm}\p{Other Math}]/`.

Alphabetic
: Equivalent to `/[\p{L}\p{Nl}\p{Other Alphabetic}]/`.

Lowercase
: Equivalent to `/[\p{Ll}\p{Other Lowercase}]/`.

Uppercase
: Equivalent to `/[\p{Lu}\p{Other Uppercase}]/`.

Cased
: Equivalent to `/[\p{Lowercase}\p{Uppercase}\p{Lt}]/`.

Case Ignorable
: Characters with a *Word Break* category of *MidLetter* or *MidNumLet*, or general category of *Mn*, *Me*, *Cf*, *Lm*, or *Sk*.

Changes when Lowercased
: Those whose normalized forms are not stable under a `toLowercase` mapping.

Changes when Uppercased
: Those whose normalized forms are not stable under a `toLowercase` mapping.

Changes when Titlecased
: Those whose normalized forms are not stable under a `toTitlecase` mapping.

Changes when Casefolded
: Those whose normalized forms are not stable under case folding.

Changes when Casemapped
: Those whose normalized forms are not stable under case mapping.

ID Start
: Those which start an identifier. Equivalent to the union of `\p{L}`, `\p{Nl}`, `\p{Other ID Start}`, `\P{Pattern Syntax}`, `\P{Pattern White Space}`.

ID Continue
: Those that can continue an identifier. Equivalent to the union of `\p{ID Start}`, `\p{Mn}`, `\p{Mc}`, `\p{Nd}`, `\p{Pc}`, `\p{Other ID Continue}`, `\P{Pattern Syntax}`, `\P{Pattern White Space}`

XID Start
: *ID Start* modified for closure under NFKx.

XID Continue
: *ID Continue* modified for closure under NFKx.

Default Ignorable Code Point
: Equivalent to the union of `\p{Other Default Ignorable Code Point}`, `\p{Cf}`, `\p{Variation Selector}`, `\P{White Space}`, `[^\uFFF9-\uFFFB]`, `[^\u0600-\u0603]`, and `[^\u06DD\u070F]`.

Grapheme Extend
: Equivalent to the union of `\p{Me}`, `\p{Mn}`, and `\p{Other Grapheme Extend}`.

Grapheme Base

Equivalent to the union of `\p{Any}`, `\P{C}`, `\P{Zl}`, `\P{Zp}`, and `\P{Grapheme Extend}`.

Grapheme Link

Those with a canonical combining class of *Virama*.

Script

If the property name is a Unicode script value, or an alias thereof, it represents characters in that script. In the list below, names separated by a solidus are equivalent.

- *Arab* / *Arabic*
- *Armi* / *Imperial Aramaic*
- *Armn* / *Armenian*
- *Avst* / *Avestan*
- *Bali* / *Balinese*
- *Bamu* / *Bamum*
- *Beng* / *Bengali*
- *Bopo* / *Bopomofo*
- *Brai* / *Braille*
- *Bugi* / *Buginese*
- *Buhd* / *Buhid*
- *Cans* / *Canadian Aboriginal*
- *Cari* / *Carian*
- *Cham*
- *Cher* / *Cherokee*
- *Copt* / *Coptic* / *Qaac*
- *Cprt* / *Cypriot*
- *Cyrl* / *Cyrillic*
- *Deva* / *Devanagari*
- *Dsrt* / *Deseret*
- *Egyp* / *Egyptian Hieroglyphs*
- *Ethi* / *Ethiopic*
- *Geor* / *Georgian*
- *Glag* / *Glagolitic*
- *Goth* / *Gothic*

- *Grek* / *Greek*
- *Gujr* / *Gujarati*
- *Guru* / *Gurmukhi*
- *Hang* / *Hangul*
- *Hani* / *Han*
- *Hano* / *Hanunoo*
- *Hebr* / *Hebrew*
- *Hira* / *Hiragana*
- *Hrkt*: *Katakana* or *Hiragana*
- *Ital* / *Old Italic*
- *Java* / *Javanese*
- *Kali* / *Kayah Li*
- *Kana* / *Katakana*
- *Khar* / *Kharoshthi*
- *Khmr* / *Khmer*
- *Knda* / *Kannada*
- *Kthi* / *Kaithi*
- *Lana* / *Tai Tham*
- *Laoo* / *Lao*
- *Latn* / *Latin*
- *Lepc* / *Lepcha*
- *Limb* / *Limbu*
- *Linb* / *Linear B*
- *Lisu* / *Lisu*
- *Lyci* / *Lycian*
- *Lydi* / *Lydian*
- *Mlym* / *Malayalam*
- *Mong* / *Mongolian*
- *Mtei* / *Meetei Mayek*
- *Mymr* / *Myanmar*
- *Nkoo* / *Nko*
- *Ogam* / *Ogham*
- *Olck* / *Ol Chiki*
- *Orkh* / *Old Turkic*
- *Orya* / *Oriya*
- *Osma* / *Osmanya*
- *Phag* / *Phags Pa*
- *Phli* / *Inscriptional Pahlavi*

- *Phnx / Phoenician*
- *Prti / Inscriptional Parthian*
- *Rjng / Rejang*
- *Runr / Runic*
- *Samr / Samaritan*
- *Sarb / Old South Arabian*
- *Saur / Saurashtra*
- *Shaw / Shavian*
- *Sinh / Sinhala*
- *Sund / Sundanese*
- *Sylo / Syloti Nagri*
- *Syrc / Syriac*
- *Tagb / Tagbanwa*
- *Tale / Tai Le*
- *Talu / New Tai Lue*
- *Taml / Tamil*
- *Tavt / Tai Viet*
- *Telu / Telugu*
- *Tfng / Tifinagh*
- *Tglg / Tagalog*
- *Thaa / Thaana*
- *Thai*
- *Tibt / Tibetan*
- *Ugar / Ugaritic*
- *Vaii / Vai*
- *Xpeo / Old Persian*
- *Xsux / Cuneiform*
- *Yiii / Yi*
- *Zinh / Inherited / Qaai*
- *Zyyy / Common*
- *Zzzz / Unknown*

ENUMERABLES

The `Enumerable` module is a mix-in providing nearly fifty methods for searching, sorting, filtering, and transforming collections of objects. It is included by `Array`, `Hash`, `Range`, and `IO`, so knowledge of its API is widely applicable. Any class may mix-in `Enumerable` as long as it defines a method named `#each` which yields each element of the collection in turn. This chapter documents the `Enumerable` API, and explains how your own classes can take advantage of its capabilities.

Querying

`#include?(`*object*`)` and its alias `#member?` return `true` if the receiver contains *object*; `false` otherwise.

```
[/a/, :a, ?a].include? /a/ #=> true
[Dir.home, __FILE__].include? 47 #=> false
(?~ * 10).include? ?~ #=> true
1.0.step(2.0, 0.1).member? 1.6 #=> true
[[0,1]].include? 0 #=> false
{a:1}.include? :a #=> true
{a:1}.include? 1  #=> false
```

`#count` returns the number of elements contained in the receiver by iterating over them. If given an argument, it returns the number of times that appears in the receiver. Similarly, if given a block, it passes each element to it in turn, returning the number of times the block evaluated to true.

```
enu = [1, 1, 3, 5, 5, 5, 5, 6, 8, 10, 12, -12, 1, 2]
enu.count #=> 14
enu.count(1.0) #=> 3
enu.count(7) #=> 0
enu.count{|n| n.odd? and n > 0} #=> 8
enu.clear.count #=> 0
[%w{a b}, %w{c d}].count([?a, ?b]) #=> 1
```

`#all?` and `#none?` test whether their receiver contains no false and no true elements, respectively. `#any?` and `#one?` test whether their receiver has at least one and exactly one true element, respectively. They evaluate each element by passing it to the block, which is assumed to be `{|e| e}` if omitted. They return `true` as soon as they succeed—skipping any remaining elements—or `false` if they fail.

```
digits = 0..9
digits.all?{|d| d < 10} #=> true
digits.none?{|d| (d ** 2) > 10} #=> false
digits.one?{|d| d == 3} #=> true
digits.any?{|d| d == 3} #=> true
(1...Float::INFINITY).any?{|d| d % 10 == 2} #=> true
[5, Rational(8, 2), ''[1], :nil].all? #=> false
users = IO.foreach('/etc/passwd').map{|user| user.split(/:/)}
users.all?{|u| u.last.start_with?(?/)} and users.one?{|u| u.first == 'root'} #=> true
```

Filtering

The bulk of `Enumerable` methods are filters that compose a subset of the receiver's elements by passing them to a block.

`#grep(`*pattern*`)` selects the elements of the receiver which are `#===(`*pattern*`)`. As the name implies, `#grep` is often invoked with a `Regexp` argument, but any object that supports the case equality test is allowed. Matching elements are returned as an `Array`.

```
Enumerable.instance_methods.grep(/while/)
#=> [:take_while, :drop_while]
(1..100).grep(5..7)
#=> [5, 6, 7]
(1..1000).grep(->(e){ e.to_s(2) =~ /^110+11$/})
#=> [27, 51, 99, 195, 387, 771]
```

`#detect` returns the first element of the receiver for which its block is true. If the block is never true, and a `Proc` argument was given, it is called and its result returned; otherwise, `nil` is returned.

```
(?b..?r).detect{|e| e =~ /[aeiou]/} #=> "e"
(?b..?r).detect{|e| e =~ /[[:upper:]]/} #=> nil
[Rational(1,10), Rational(7,10), Rational(4,10)].find(->{ 10 }){|e| e > 10} #=> 10
```

`#find_index(`*value*`)` returns the 0-based index of the first element of the receiver equal to *value*. If a block is given instead of *value*, returns the index of the first element for which the block is not true. When no object matches, `nil` is returned.

```
(:aa..:zz).find_index(:az) #=> 25
stages = {baby: :infant, infant: :toddler, toddler: :preteen, teen: :adult}
stages.find_index{|from, to| from == :toddler} #=> 2
stages.find_index{|from, to| from == :adult} #=> nil
```

`#select` returns an `Array` of all elements for which the block is not false. Conversely, `#reject` returns an `Array` of elements for which the block *is* false.

```
(1...1000).select{|n| n.to_s(16).end_with?('ef')} #=> [239, 495, 751]
fruits = {raspberry: :red, grape: [:white, :black], banana: :yellow, orange: :orange}
fruits.select{|fruit, colour| fruits.key?(colour)} #=> {:orange => :orange}
(0..9).reject(&:even?) #=> [1, 3, 5, 7, 9]
```

`#first` returns the first element of the receiver, or `nil` if it is empty. If `#first` is given a numeric argument, *n*, they return an `Array` of, at most, the first *n* elements. Conversely, `#drop(`*n*`)` returns an `Array` containing all *but* the first *n* elements.

```
digits = 0..9
digits.first #=> 0
digits.first(3) #=> [0, 1, 2]
digits.drop(3) #=> [3, 4, 5, 6, 7, 8, 9]
digits.take(30)
#=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
digits.select{|e| e.even? and e.odd?}.first(2) #=> []
```

`#take_while` takes elements from the receiver while the block is true; when the block is false, it returns the collected elements as an `Array`. Conversely, `#drop_while` ignores each element of the receiver while the block is true; returning the remainder as an `Array`.

```
digits = 1..9
require 'prime'
square_free = ->(n){ n.prime_division.all?{|a,b| b == 1} }
digits.take_while(&square_free) #=> [1, 2, 3]
digits.drop_while(&square_free) #=> [4, 5, 6, 7, 8, 9]
```

Transforming

#map collects the results of running their block for each element in an Array, which they return. #flat_map is similar but flattens the result Array.

```
(?a..?f).map &:upcase
#=> ["A", "B", "C", "D", "E", "F"]
(10..20).map{|n| n**2 }
#=> [100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400]
{ 'google.com' => 'Google',
  'ruby-lang.org' => 'Ruby',
  'wikipedia.org' => 'Wikipedia'}.map do |host, anchor|
  "<a href=//#{host}/>#{anchor}</a>"
end
#=> ["<a href=//google.com/>Google</a>", "<a href=//ruby-lang.org/>Ruby</a>",
#=>  "<a href=//wikipedia.org/>Wikipedia</a>"]
[10..20, 20..30, 30..40, 40..50].collect{|r| r.to_a.sample}
#=> [16, 30, 36, 44]
shells = IO.foreach('/etc/passwd').flat_map{|l| l.chomp.split(/:/).values_at(0,-1)}
Hash[*shells]['kernoops'] #=> "/bin/false"
```

#partition returns an Array of Arrays: the first containing the elements for which the block was true, the second containing the remainder. #group_by generalises this approach by returning a Hash whose keys are the return values of the block, and values are the elements for which the block returned that value.

```
square, nonsquare = (1..25).partition{|n| Math.sqrt(n) == Math.sqrt(n).to_i}
#=> [[1, 4, 9, 16, 25],
#=>  [2, 3, 5, 6, 7, 8, 10, 11, 12, 13, 14, 15, 17, 18, 19, 20, 21, 22, 23, 24]]
(1..10).group_by{|n| 10.lcm n}
#=> {10=>[1, 2, 5, 10], 30=>[3, 6], 20=>[4], 70=>[7], 40=>[8], 90=>[9]}
String.instance_methods.group_by{|m| String.instance_method(m).arity}.map{|a,m| [a, m.size]}
#=> [[1, 42], [0, 71], [-1, 47], [2, 9]]
```

#chunk also groups elements by the value of its block, but chunks each *consecutive* group with the same block value, or *key*. A chunk is an Array whose first element is a key, and last element an Array of consecutive elements from the receiver which evaluated to it. The return value is an Enumerator of chunks.

```
 Math::PI.to_s.scan(/\d/).map(&:to_i).chunk(&:even?).map(&:last)
#=> [[3, 1], [4], [1, 5, 9], [2, 6], [5, 3, 5], [8], [9, 7, 9, 3]]
ROWS, COLS = 9, 17
grid = ROWS.times.map{ Array.new(COLS, '   ') }
(2..40).map{|n| Math.sin(n) ** 2}.chunk{|n| Math.log10(n).to_i}.each_with_index do |(log, ns
   ROWS.pred.downto(ROWS-ns.size){|row| grid[row][i] = " * "}
   (grid[ROWS] ||= []) << "%#2d " % log
end
grid.each{|row| puts row.join}
#             *                               *
#             *                               *
#             *                               *
#             *                               *
#             *                               *
#             *                               *
#             *                               *
#       *     *     *     *     *     *     *     *
# *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *
# 0 -1  0 -1  0 -1  0 -1  0 -4  0 -1  0 -1  0 -1  0
```

#slice_before(*pattern*) offers another approach for grouping elements into Arrays, or *slices*. It requires either an argument, *pattern*, or block. The first slice begins with the first element of the receiver. Starting with the second element, each element is either compared to *pattern* with the case equality operator (#===), or passed to the block. If the result is true, the element begins a new slice; otherwise, it continues the existing slice. Finally, an Enumerator of slices is returned.

```
# -*- coding: utf-8 -*-
 Math::PI.to_s.scan(/\d/).map(&:to_i).slice_before(&:even?).to_a
#=> [[3, 1], [4, 1, 5, 9], [2], [6, 5, 3, 5], [8, 9, 7, 9, 3]]
IO.foreach('/usr/share/dict/words').slice_before(/^[[:upper:]]$/).map{|w| w.last.chomp}[1..5
#=> ["Aztlan's", "Byzantium's", "Czerny's", "Düsseldorf", "Ezra's"]
(?a..?z).slice_before{|l| %w{a e i o u}.include? l}.to_a
```

```
#=> [["a", "b", "c", "d"], ["e", "f", "g", "h"], ["i", "j", "k", "l", "m", "n"],
#    ["o", "p", "q", "r", "s", "t"], ["u", "v", "w", "x", "y", "z"]]
```

`#zip` creates an `Array` for each element of the receiver, containing the element along with the corresponding element from each of its `Enumerable` arguments. If a block is given, each result `Array` is yielded to it; otherwise they are returned as an `Array` of `Arrays`.

```
[:a, :b, :c].zip #=> [[:a], [:b], [:c]]
[:a].zip([:b, :c], [:d, :e, :f]) #=> [[:a, :b, :d]]
[:a, :b, :c].zip([:d], [:e, :f]) #=> [[:a, :d, :e], [:b, nil, :f], [:c, nil, nil]]
digits = 1..3
sin, cos, tan = %w{sin cos tan}.map{|m| digits.map{|n| Math.send(m, n * Math::PI/180)}}
digits.zip(sin, cos, tan)
#=> [[1, 0.01745240643728351, 0.9998476951563913, 0.017455064928217585],
#=>  [2, 0.03489949670250097, 0.9993908270190958, 0.03492076949174773],
#=>  [3, 0.05233595624294383, 0.9986295347545738, 0.052407779283041196]]
```

`#inject(`*initial*`)` combines the elements of its receiver into a single value. When a block is given, it is called for each element, receiving an accumulator object and the element as arguments. When the block is omitted, and a `Symbol` argument supplied—`#inject(`*symbol*`)`—a block of the form `{|`*acc*`, `*el*`| `*acc*`.send(`*symbol*`, `*el*`)}` is assumed. The accumulator is initialised to the first element of the receiver, so iteration begins with the second element. The return value of `#inject` is the final value of the accumulator.

Iteration	Accumulator Value	Element
1	1 (first element)	2
2	2 (`1.send(:*, 2)`)	3
3	6 (`2.send(:*, 3)`)	

The evaluation of `[1, 2, 3].inject(:*)`

```
%w{a b c d}.reduce(:+) #=> "abcd"
(2..8).reverse_each.inject(:-) #=> -19
[[:a, 4], [:b, 12], [:c, 9]].reduce(1){|acc, el| acc * el.last} #=> 432
[{joe: 27}, {bob: 23}, {jim: 40}, {jolene: 18}].reduce :merge
#=> {:joe=>27, :bob=>23, :jim=>40, :jolene=>18}
Dir["#{Dir.home}/.*"].reduce do |newest, file|
  File.file?(file) && File.mtime(newest) < File.mtime(file) ? file : newest
end #=> "/home/run/.xsession-errors
```

Iteration

In addition to #each, the following iterators are also available. #each_with_index yields each element in turn along with its corresponding, 0-based index. #each_with_object takes an arbitrary object as argument, which it yields along with each element, then returns. #each_entry behaves like #each, but if the latter would have yielded multiple values at once, they are combined into an Array. #reverse_each yields each element of the receiver in reverse order.

```
$places = %w{Agartha Atlantis Avalon Camelot Eden
            El_Dorado Shangri-La Thule Utopia Valhalla}
$places.each_with_index.map{|place, index| "#{index}:#{place.size}"}
#=> ["0:7", "1:8", "2:6", "3:7", "4:4", "5:9", "6:10", "7:5", "8:6", "9:8"]
$places.reverse_each{|place| print place[0]}
# VUTSEECAAA
$places.each_with_object({}){|place,vowels| vowels[place] = place.downcase.count('aeiou')}
#=> {"Agartha"=>3, "Atlantis"=>3, "Avalon"=>3, "Camelot"=>3, "Eden"=>2, "El_Dorado"=>4,
#=>  "Shangri-La"=>3, "Thule"=>2, "Utopia"=>4, "Valhalla"=>3}
class Places
  def each
    $places.group_by{|place| place[0]}.each{|letter, names| yield letter, *names}
  end
end
Places.new.extend(Enumerable).each_entry{|places| print " #{places.first}:#{places[1..-1].si
# A:3 C:1 E:2 S:1 T:1 U:1 V:1
```

#each_slice(*size*) yields the elements of the receiver in slices with maximum size *size*. The first slice contains the first *size* elements, the second contains the next *size*, and so forth. #each_cons(*size*) is similar, but yields *consecutive* sub-Arrays of size *size*. The first contains the first *size* elements, the second element 1–*size*, and so forth.

```
plants = [:Hornworts, :Mosses, :Liverworts, :Conifers, :Cycads, :flowering_plants]
plants.each_slice(1).to_a
#=> [[:Hornworts], [:Mosses], [:Liverworts], [:Conifers], [:Cycads], [:flowering_plants]]
plants.each_cons(1).to_a
#=> [[:Hornworts], [:Mosses], [:Liverworts], [:Conifers], [:Cycads], [:flowering_plants]]
plants.each_slice(2).to_a
#=> [[:Hornworts, :Mosses], [:Liverworts, :Conifers], [:Cycads, :flowering_plants]]
```

```
plants.each_cons(2).to_a
#=> [[:Hornworts, :Mosses], [:Mosses, :Liverworts],      [:Liverworts, :Conifers],
#=>  [:Conifers, :Cycads],  [:Cycads, :flowering_plants]]
plants.each_slice(4).to_a.flatten.count(:Liverworts) #=> 1
plants.each_cons(4).to_a.flatten.count(:Liverworts) #=> 3
plants.each_slice(5).count #=> 2
plants.each_cons(5).count #=> 2
```

`#cycle(`*count*`)` yields each element of the receiver to the block, then recurses. After repeating this process *count* times, or forever if *times* is omitted, it returns `nil`.

```
(?a..?f).map &:upcase
#=> ["A", "B", "C", "D", "E", "F"]
(10..20).map{|n| n**2 }
#=> [100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400]
{ 'google.com' => 'Google',
  'ruby-lang.org' => 'Ruby',
  'wikipedia.org' => 'Wikipedia'}.map do |host, anchor|
  "<a href=//#{host}/>#{anchor}</a>"
end
#=> ["<a href=//google.com/>Google</a>", "<a href=//ruby-lang.org/>Ruby</a>",
#=>  "<a href=//wikipedia.org/>Wikipedia</a>"]
[10..20, 20..30, 30..40, 40..50].collect{|r| r.to_a.sample}
#=> [16, 30, 36, 44]
shells = IO.foreach('/etc/passwd').flat_map{|l| l.chomp.split(/:/).values_at(0,-1)}
Hash[*shells]['kernoops'] #=> "/bin/false"
```

Sorting

`#sort` sorts the elements of the receiver according to their `#<=>` methods, then returns them in an `Array`. Alternatively, if a block is supplied, it is passed two elements at once and expected to follow the `<=>` convention of returning `-1`, `0`, or `1`.

`#sort_by` also sorts the receiver, but does so using an arbitrary attribute of each element. Each element in turn is passed to the block, whose return value is the key to sort by. If the block returns an `Array`, a multi-level sort is performed: elements are sorted by its first element, then, if there's a tie, its second element, and so forth.

```
gods = %w{Aphrodite Demeter Ares Eros   Apollo   Hades  Dionysus
          Artemis   Zeus    Hera Hermes Poseidon Athena Hestia Hephaestus}
gods.sort #=>["Aphrodite", "Apollo", "Ares", "Artemis", "Athena", "Demeter", "Dionysus",
#=>           "Eros", "Hades", "Hephaestus", "Hera", "Hermes", "Hestia", "Poseidon", "Zeus"]
gods.sort{|a, b| a.length <=> b.length}
#=> ["Hera", "Zeus", "Ares", "Eros", "Hades", "Apollo", "Hermes", "Athena", "Hestia",
#=>  "Demeter", "Artemis", "Poseidon", "Dionysus", "Aphrodite", "Hephaestus"]
Hash[*gods.zip(%w{f f m m m m m f m f m m f f m}).flatten].sort_by(&:reverse).map(&:first)
#=> ["Aphrodite", "Artemis", "Athena", "Demeter", "Hera", "Hestia", "Apollo", "Ares",
#=>  "Dionysus", "Eros", "Hades", "Hephaestus", "Hermes", "Poseidon", "Zeus"]
class String
  def <=>(o)
    reverse.ord <=> o.reverse.ord
  end
end
gods.sort #=> ["Hera", "Hestia", "Athena", "Aphrodite", "Poseidon", "Apollo", "Demeter",
#=>            "Artemis", "Zeus", "Dionysus", "Hermes", "Hades", "Eros", "Ares", "Hephaestus
```

Minimums & Maximums

`#min` and `#max` compare the elements in the receiver with the elements' `#<=>` methods, and return the minimum and maximum element, respectively. If they are given a block, it is passed two elements at a time, and expected to follow the `<=>` convention of returning `-1`, `0`, or `1`. `#minmax` behaves identically, except it returns both minimum and maximum values as an `Array`.

```
[300, 92, 827_99, -45, 300, 1].minmax #=> [-45, 82799]
languages = %w{Ruby Haskell Scala Clojure Perl Python Lisp Smalltalk}
languages.min #=> "Clojure"
languages.max #=> "Smalltalk"
languages.min{ +1 } #=> "Ruby"
languages.minmax{|a,b| a.ord <=> b.ord } #=> ["Clojure", "Scala"]
languages.sort.minmax do |a,b|
  languages.count{|l| l.start_with?(a[0])} <=> languages.count{|l| l.start_with?(b[0])}
end #=> ["Clojure", "Perl"]
```

`#min_by` and `#max_by` pass each element to a block, and return those corresponding to the smallest and largest return value, respectively.

`#minmax_by` does likewise, except it returns an `Array` containing both the minimum and maximum values.

```
[300, 92, 827_99, -45, 300, 1].minmax_by(&:magnitude) #=> [1, 82799]
languages = %w{Ruby Haskell Scala Clojure Perl Python Lisp Smalltalk}
languages.min_by(&:size) #=> "Ruby"
languages.max_by{|l| l.codepoints.reduce(:+)} #=> "Smalltalk"
languages.minmax_by{|l| l.ord } #=> ["Clojure", "Scala"]
languages.sort.minmax_by {|l| languages.count{|e| l.start_with?(e[0])}}
#=> ["Clojure", "Perl"]
```

Enumerator

An `Enumerator` is an objectification of an enumeration. It mixes-in `Enumerable`, so also responds to the methods detailed above.

Instantiation

As described above, most `Enumerable` methods return `Enumerators` when their block is omitted. More generally, `Kernel#to_enum(`*method*`=:each` creates an `Enumerator` from its receiver's *method* method. If additional arguments are provided, they are passed to *method*.

`Enumerator.new` can take a block to which it passes an instance of `Enumerator::Yielder`. The block specifies an element to be enumerated by supplying it as an argument of `Enumerator::Yielder#yield`, or its alias `Enumerator::Yielder#<<`. `#yield` behaves lazily: blocking until the `Enumerator` requests a new element.

```
"Once bitten, twice shy".to_enum(:bytes).first(5)
#=> [79, 110, 99, 101, 32]
Enumerator.new do |yielder|
  (1..Float::INFINITY).each do |n|
    yielder << n if n.odd?
  end
end.first(5) #=> [1, 3, 5, 7, 9]
```

External Iterators

The `Enumerable` methods are termed *internal* iterators because they internalise the details of the enumeration. Each method invokes `#each` anew, restarting iteration from the first element of the collection. Conversely, *external* iterators are driven by the user, who must explicitly request each element he requires.

An `Enumerator` implements an external iterator with a `Fiber` that maintains a pointer, *p*, to the next element in the receiver. Initially, *p* points to the first element. `#next` returns the value of *p*, then advances it to the next element; `#peek` returns *p* without advancing it. `#rewind` resets *p* to point to the first element. Due to a limitation of `Fiber`s, these methods cannot be called across threads.

```
e = (1..3).to_enum
3.times{ print e.next } # 123
e.rewind
[e.peek, e.next, e.peek, e.peek, e.next] #=> [1, 1, 2, 2, 2]
```

After `#next` has returned the last element of the receiver, subsequent calls to `#next` or `#peek` cause `StopIteration` to be raised. This exception is rescued automatically by `Kernel.loop`, but if rescued manually, `StopIteration#result` holds the return value of the enumerated method.

```
e = [:rinse, :repeat].to_enum
e.next #=> :rinse
e.next #=> :repeat
begin
  e.next
rescue StopIteration => ex
  ex.result #=> [:rinse, :repeat]
end
e.rewind
loop{ p e.next }
# :rinse
# :repeat
```

Classes with Multiple Iteration Strategies

For `Enumerable` to be a sensible addition to a class, the collection must support a single, obvious means of iteration. If multiple approaches are plausible, the semantics of the `Enumerable` methods will be confusing.

For example, consider the `String` class. A `String` of binary data will probably be iterated by byte, a `String` containing a document may be iterated by line or paragraph, a hyphenation algorithm would iterate by character or Unicode codepoint. There is no single `#each` method that encompasses these approaches. There is no objective atomic unit that can form the basis of iteration. Accordingly, `String#each` does not exist. In its place are alternative iterators: `String#each_byte`, `String#each_char`, `String#each_codepoint`, and `String#each_line`.

The utility of this approach is seen when combining it with `Object#enum_for(`*`selector`*`)`. This method converts its receiver into an `Enumerator` object, which is an iterator implemented in terms of the receiver's *selector* method. An `Enumerator` has an `#each` method and mixes-in `Enumerable`. Hence, one can convert an object into `Enumerable` by specifying which form of iteration should be used by `#each`.

```
latin = "Tu ne cede malis, sed contra audentior ito"
char_string = latin.to_enum(:each_char)
vowels, consonants = char_string.partition{|char| char =~ /[aeiou]/}
vowels #=>
# ["u", "e", "e", "e", "a", "i", "e", "o", "a", "a", "u", "e", "i", "o", "i", "o"]
```

ARRAYS

An `Array` is a mutable, heterogeneous, ordered collection of objects, termed *elements*, indexed by an `Integer` subscript. Its mutability means that elements can be added to, and removed from, the collection at any point in its lifetime, causing the `Array` to expand or contract as needed. In keeping with her philosophy of duck-typing, Ruby does not require the elements to have a specific type: any permutation of objects may be stored in an `Array`. That each element is indexed by a unique `Integer`, enforces unambiguous order on the collection.

The first element of an `Array` has the index `0`, so the last element has an index one less than the total number of elements. Methods that accept `Array` indices as arguments, interpret negative indices as counting backward from the last element, i.e. `-2` refers to the penultimate element.

The number of elements an `Array` contains is returned by `Array#size`. When the size is `0`, `Array#empty?` returns `true`.

Instances of `Array` are `Enumerable`, so in addition to the methods described in this chapter, they also respond to the `Enumerable` methods.

Literals

An `Array` literal is a comma-separated list of expressions enclosed in square brackets (`[`, `]`). For example, `[1, :two, 'three']` creates a three-element `Array` object with `1` as the first element.

Alternative Delimiters

The `%w`*`delimiter`*`…`*`delimiter`* construct is a syntactical shortcut for instantiating an array of strings. The delimiters take the same form as those of %q. The text between them is split on whitespace, each substring becoming an element on the array. `%w` interprets its contents with single-quoted string

semantics; its counterpart, `%W`, behaves in the same way using double-quoted string semantics.

```
colours = %w{red green blue} #=> ["red", "green", "blue"]
%W{Kind of #{colours.last.capitalize}} #=> ["Kind", "of", "Blue"]
```

Array.new

`Arrays` are typically instantiated with the literal syntax explained above. `Array.new` offers more control over this process. When called with no arguments it is equivalent to `[]`. `Array.new(`*`size`*`, `*`o`*`)` creates a *size*-element `Array` with each value set to *o*. If *o* is omitted, it defaults to `nil`. When given an `Array` argument, *a*, it returns `a.to_ary`. Lastly, if a numeric argument, *size*, and block are provided, the block is called *size* times, with its return value becoming the corresponding `Array` value.

```
Array.new #=> []
Array.new 3 #=> [nil, nil, nil]
Array.new 5, :default
#=> [:default, :default, :default, :default, :default]
Array.new [42] #=> [42]
Array.new(10){|i| i**i}
#=> [1, 1, 4, 27, 256, 3125, 46656,
#    823543, 16777216, 387420489]
```

Lookup

The element at index *n* can be retrieved with `#[`*`n`*`]`. `#at(`*`n`*`)` and `#slice(`*`n`*`)` behave in the same manner. If the index *n* lies outside of the array, these methods return `nil`. The `#fetch` method also returns the element at a given index, however it differs from the aforementioned in how it reacts to non-existent indices. `#fetch(`*`n`*`)` raises an `IndexError`, or, if given a block, returns the value of the block when passed *n*. `#fetch(`*`n`*`, `*`default`*`)` returns *default*.

```
colours = %w{red green blue}
"#{colours[0]}, #{colours.at(1)}, and #{colours.slice(-1)}"
#=> "red, green, and blue"
colours[4] #=> nil
```

```
colours.fetch(-5, 'white') #=> "white"
colours.fetch(4)
#=> index 4 outside of array bounds: -3...3 (IndexError)
```

Whereas the methods above returned the element corresponding to a given index, `#index(`*e*`)`, or its alias `#find_index(`*e*`)` return the index corresponding to a given element. Specifically, `#index(`*e*`)` returns the index of the first element `#==` to `e`. If either method is provided a block instead of *e*, it calls the block with each element in turn, returning the index of the first element for which the block returns `true`. `#rindex` behaves identically to `#index` except it tests the elements in reverse order.

```
ruler = [*'Ramesses II'.chars]
ruler.index('e') #=> 3
ruler.rindex('e') #=> 6
ruler.index {|e| e =~ /[[:upper:]]/} #=> 0
```

To lookup multiple elements at once, `#[](`*start*`, `*length*`)` or `#[](`*range*`)`, or its alias, `#slice`, can be used. The first form returns an `Array` of at most *length* consecutive elements, beginning at index *start*. The second returns an `Array` of the elements at the indices covered by the given `Range`. Alternatively, `#values_at` accepts any combination of `Integer` and `Range` arguments, returning the elements at any of the given indices.

```
perfect = [6, 28, 496, 8128, 33550336]
perfect[0, 3] #=> [6, 28, 496]
perfect[2..4] #=> [496, 8128, 33550336]
perfect.values_at(-4, -2, 0) #=> [28, 8128, 6]
```

The element at index `0` can be retrieved with `#[0]` or `#first`. If `#first` is given an `Integer` argument, *n*, it is equivalent to `#[0...`*n*`]`: it returns the first *n* elements as an `Array`. Conversely, `#last` is equivalent to `#[-1]`, and `#last(`*n*`)` means `#[-`*n*`..-1]`.

```
alpha = [*(?a..?z)]
(alpha.first..alpha.last).to_a[0] #=> "a"
alpha.last 3 #=> ["x", "y", "z"]
```

Lastly, `#sample(`*n*`)` returns *n* random elements, where *n* is capped at the size of the receiver. If *n* is omitted, it defaults to `1`.

```
fractions = [Rational(1, 3), Rational(1, 2), Rational(4, 5)]
fractions.sample   #=> (1/2)
fractions.sample 5 #=> [(4/5), (1/2), (1/3)]
```

Insertion

`#insert(`*i*`, `*o*$_0$`,…,`*o*$_n$`)` inserts objects 0…*n* before the element with index *i* if *i* is positive. If *i* is negative, it counts from the end of the `Array`, and the objects are inserted after it.

```
a = []
a.insert 0, 0        #=> [0]
a.insert -1, 1, 2, 3 #=> [0, 1, 2, 3]
a.insert 0, -1       #=> [-1, 0, 1, 2, 3]
```

`#push(`*o*`)` and `#<< `*o* are equivalent to `#insert(-1, `*o*`)`, while `#unshift(`*o*`)` is equivalent to `#insert(0, `*o*`)`.

```
kubrick = []
kubrick << 'Lolita' #=> ["Lolita"]
kubrick.push('Dr. Strangelove',
             '2001',
             'A Clockwork Orange')
#=> ["Lolita", "Dr. Strangelove", "2001", "A Clockwork Orange"]
kubrick.unshift 'Spartacus'
#=> ["Spartacus", "Lolita", "Dr. Strangelove", "2001", "A Clockwork Orange"]
```

Replacement

To replace the object at index *n* with *o*, use `#[`*n*`]` as an lvalue, e.g. `#[`*n*`] = `*o*. If `#[]=` is given a pair of `Integers` or a `Range` for its arguments, the affected slice is replaced by the rvalue.

```
# encoding: utf-8
greek = %w{alpha beta gamma delta}
greek[0] = ?α          #=> "α"
greek[1, 2] = [?β, ?γ] #=> ["β", "γ"]
greek[-1] = ?δ         #=> "δ"
greek                  #=> ["α", "β", "γ", "δ"]
```

`#fill` replaces one sequence of elements with another. `#fill(`*o*`)` substitutes every element of the receiver for *o*. `#fill(`*o*`, `*start*`, `*length*`)` replaces index *start* through to index *length* with *o*. If *length* is omitted, it is assumed to be the length of the receiver. `#fill(`*o*`, `*range*`)` replaces the elements whose indices are covered by the `Range` *range* with *o*. If the *o* argument is omitted, and a block supplied instead, the selected elements are replaced with the value of the block when passed their index.

```
notes = Array.new(7)
notes.fill ?F
#=> ["F", "F", "F", "F", "F", "F", "F"]
notes.fill ?B, 6
#=> ["F", "F", "F", "F", "F", "F", "B"]
notes.fill ?A, 5, 1
#=> ["F", "F", "F", "F", "F", "A", "B"]
notes.fill ?G, 4...5
#=> ["F", "F", "F", "F", "G", "A", "B"]
notes.fill(0..2) do |i|
  case i
    when 0 then ?C
    when 1 then ?D
    when 2 then ?E
  end
end #=> ["C", "D", "E", "F", "G", "A", "B"]
```

Concatenation

`#concat(`*a*`)` and its alias, `#+ `*a*, append the elements of `Array` to the receiver.

`#join` concatenates the elements of the receiver with `$,`—the output record separator, which is `nil` by default—to produce a `String`. If given a `String` argument, that is used in place of `$,`

`#* `*string* is equivalent to `#join(`*string*`)`. `#* `*integer* is equivalent to concatenating *integer* copies of the receiver.

```
parents = %w{Michael Susan}
grandparents = %w{Thomas Isobel}
```

```
$, = ?\
(parents + grandparents).join #=> "Michael Susan Thomas Isobel"
parents * ', '                #=> "Michael, Susan"
parents * 2
#=> ["Michael", "Susan", "Michael", "Susan"]
```

Deletion

#delete_at(*n*) deletes and returns the element at index *n*. As special cases, #shift is equivalent to #delete_at(0), and #pop is equivalent to #delete_at(-1).

```
at = Array.instance_methods(false).grep(/at$/)
#=> [:at, :concat, :values_at, :delete_at]
at.delete_at 1 #=> :concat
at.shift       #=> :at
at.pop         #=> :delete_at
at             #=> [:values_at]
```

#delete(*e*) deletes all elements equal to *e*. If no deletions occurred, nil is returned. If a block is supplied, its value is returned in place of nil.

```
italians = %w{Abbati   Albertinelli Allori     Allori Altichiero
              Amigoni  Angelico     Anguissola Arcimboldo}
italians.delete('Allori') #=> "Allori"
italians.delete('Abbati') #=> "Abbati"
italians.delete('Allori') #=> nil
italians.delete('Azzolini') do |name|
  "#{name}: ???"
end                       #=> "Azzolini: ???"
italians
#=> ["Albertinelli", "Altichiero", "Amigoni",
#    "Angelico", "Anguissola", "Arcimboldo"]
```

#delete_if passes each element in turn to the associated block, deleting those for which the block evaluates to true. Its inverse is #keep_if, which deletes elements for which the block evaluates to false. #reject! and #select! behave like #delete_if! and #keep_if!, respectively, except they return the receiver if they made changes; nil otherwise.

```
digits = (0..9).to_a
digits.delete_if{|d| d > 7}
#=> [0, 1, 2, 3, 4, 5, 6, 7]
digits.keep_if(&:odd?)
#=> [1, 3, 5, 7]
digits.select!{|d| d**2 > d}
#=> [3, 5, 7]
digits.reject!(&:even?)
#=> nil
```

`#compact!` deletes all `nil` elements, while `#uniq!` deletes all duplicates. Both methods have non-bang variants which return a copy of their receiver with the elements removed.

```
a = 'James Joyce'.chars.map(&:upcase!)
#=> [nil, "A", "M", "E", "S", nil, nil, "O", "Y", "C", "E"]
a.compact!
#=> ["A", "M", "E", "S", "O", "Y", "C", "E"]
a.uniq
#=> ["A", "M", "E", "S", "O", "Y", "C"]
a.compact!
#=> nil
```

To remove every element from an `Array`, `#clear` can be used.

```
banned = ['Animal Farm', 'Areopagitica',
          'Brave New World', 'Fanny Hill']
banned.taint
banned.untrust
banned.clear      #=> []
banned.size       #=> 0
banned.tainted?   #=> true
banned.untrusted? #=> true
```

Arrays of Arrays

An *Array of Arrays* is an `Array` whose elements are `Arrays`. `#flatten` replaces every element that is an `Array` with its contents, recursively. If `#flatten` is given an `Integer` argument, *limit*, it recurses to a maximum depth of *limit*. `#flatten!` is identical except it modifies the receiver in-place.

```
pow = [[1,  [1,   [1]]],
       [2,  [4,   [8]]],
       [3,  [9,  [27]]],
       [4,  [16, [64]]]]
pow.flatten
#=> [1, 1, 1, 2, 4, 8, 3, 9, 27, 4, 16, 64]
pow.flatten(1)
#=> [1, [1, [1]], 2, [4, [8]], 3, [9, [27]], 4, [16, [64]]]
```

`#assoc(`*o*`)` assumes the receiver is an `Array` of `Arrays`, and returns the first `Array` whose first element is equal to *o*. `#rassoc(`*o*`)` does likewise, except it compares the second element of each `Array` with *o*. In both cases, `nil` is returned when no matching `Array` was found.

```
one = [[Rational(2, 3), Rational(1, 3)],
       [Rational(3, 5), Rational(2, 5)],
       [Rational(4, 7), Rational(3, 7)]]
one.assoc Rational(3, 5)  #=> [(3/5), (2/5)]
one.rassoc Rational(2, 7) #=> [(4/7), (3/7)]
one.assoc 1               #=> nil
```

Assuming the receiver is an `Array` of `Arrays`, where each row has the same number of columns, `#transpose` returns the result of transposing the rows and columns.

```
prime_fact = [[1, 1], [2, 1], [3, 1], [4, 2],
              [5, 1], [6, 2], [7, 1], [8, 3], [9, 2]]
prime_fact.transpose
#=> [[1, 2, 3, 4, 5, 6, 7, 8, 9],
#    [1, 1, 1, 2, 1, 2, 1, 3, 2]]
prime_fact.transpose.transpose == prime_fact
#=> true
```

Permutations & Combinations

`#permutation(`*size*`)` yields each permutation of the receiver of length *size*. If *size* is omitted, it is assumed to be that of the receiver. Similarly, `Array#combination` yields combinations of elements with the given length.

```
dna = [?A, ?C, ?G, ?T]
dna.permutation(2).to_a
#=> [["A", "C"], ["A", "G"], ["A", "T"], ["C", "A"], ["C", "G"],
#    ["C", "T"], ["G", "A"], ["G", "C"], ["G", "T"], ["T", "A"],
#    ["T", "C"], ["T", "G"]]
dna.combination(2).to_a
#=> [["A", "C"], ["A", "G"], ["A", "T"],
#    ["C", "G"], ["C", "T"], ["G", "T"]]
```

#product accepts any number of Array arguments, then returns an Array of Arrays comprising all combinations of picking an element from the receiver and each argument. If #product is given *n* arguments, each element of the Array it returns has *n* + 1 elements.

```
one_two = [1, 2]
ab = [?a, ?b]
one_two.product(ab, [:_])
#=> [[1, "a", :_], [1, "b", :_], [2, "a", :_], [2, "b", :_]]
```

Iteration

#each yields each element of the receiver in turn to the associated block. #each_with_index yields each element along with its index.

```
drops = [165, 168, 173, 180]
weight = 14.0
drops.each do |cm|
  puts "Use a #{cm} cm rope for a man of #{weight} stone"
  weight -= 0.5
end
# Use a 165 cm rope for a man of 14.0 stone
# Use a 168 cm rope for a man of 13.5 stone
# Use a 173 cm rope for a man of 13.0 stone
# Use a 180 cm rope for a man of 12.5 stone
drops.each_with_index.to_a
#=> [[165, 0], [168, 1], [173, 2], [180, 3]]
```

Set Operations

Set operations find the difference, intersection, and union of the receiver and another `Array`, *a*. `Array#-(`*a*`)` returns the difference between the receiver and *a*, i.e., the elements of the receiver less the elements of *a*. `Array#&(`*a*`)` returns the intersection of the receiver and *a*, i.e., elements common to both, without duplicates. `Array#|(`*a*`)` returns the union of the receiver and *a*, i.e. their concatenation, minus any duplicates.

```
male_names = %w{Alex Brian Chris Dave}
female_names = %w{Alex Bernice Chris Denise}
male_names - female_names
#=> ["Brian", "Dave"]
male_names & female_names
#=> ["Alex", "Chris"]
male_names | female_names
#=> ["Alex", "Brian", "Chris", "Dave", "Bernice", "Denise"]
```

Ordering

An `Array` is sorted with either `Array#sort` or `Array#sort_by`, both of which behave like their namesakes in the `Enumerable` module. The elements in an `Array` can be reversed with `Array#reverse`, or sorted in a pseudo-random order with `Array#shuffle`. All four methods order a copy of the receiver, which they return.

```
words = %w{zero one two three four five}
words.sort!
#=> ["five", "four", "one", "three", "two", "zero"]
words.reverse
#=> ["zero", "two", "three", "one", "four", "five"]
words.shuffle
#=> ["two", "three", "five", "one", "four", "zero"]
```

HASHES

A `Hash` represents a mutable, heterogeneous collection of associations between pairs of objects. The collection is indexed by the first element of the pair-its *key*-which uniquely identifies the second element of the pair-its *value*. It is ordered by insertion[1]. Uses include a dictionary, allowing values to be looked up by key; a dispatch table, where the values are `Procs` identified by their key; and a cache of unique values, taking advantage of the unique keys property.

Literals

A `Hash` literal consists of a comma-separated list of key-value pairs enclosed in curly braces (`{`, `}`). It creates a new `Hash` object with the specified contents. The key is separated from its value with `=>`. If the key is a `Symbol` literal, the colon with which it's prefixed may be made its suffix, and `=>` can be omitted. (This syntactical shortcut is a reason for preferring `Symbol` keys).

```
{lemon: :yellow, orange: :orange, apple: [:red, :green]}
#=> {:lemon=>:yellow, :orange=>:orange, :apple=>[:red, :green]}
{?a => :vowel,     ?b => :consonant, ?c => :consonant,
 ?d => :consonant, ?e => :vowel,     ?f => :consonant}
#=> {"a"=>:vowel, "b"=>:consonant, "c"=>:consonant,
     "d"=>:consonant, "e"=>:vowel, "f"=>:consonant}
{:london => :england, :londonderry => :ireland, london: :ontario}
#=> {:london=>:ontario, :londonderry=>:ireland}
```

Look-up

The element reference syntax, `Hash#[]`, returns the value for a given key, or the default value if they key doesn't exist. The `#fetch(`*key*`)` behaves

---

1. Hash tables do not normally preserve order of insertion, of course; a doubly-circularly linked list is used behind the scenes. Ilya Grigorik examined the consequences of this decision.

likewise, except when the key doesn't exist: with no other arguments, it raises an `IndexError`; with a second argument, it returns that; and with a block, it returns the block's value when given the key as argument.

`#values_at` accepts an arbitrary number of keys as arguments, returning an `Array` comprising their corresponding values. The default value is returned for non-existent keys.

`#assoc(`*key*`)` and `#rassoc(`*value*`)` return a key-value pair as a two element `Array`. The former, returns the pair whose key equals *key*; the latter, returns the first pair whose value equals *value*. Both return `nil` when no corresponding pair exists.

```
vitamins = {apricot: :a, ham: :b1, cabbage: :c, spinach: :k}
vitamins[:apricot] #=> :a
vitamins.fetch(:pizza, '???') #=> "???"
vitamins.fetch(:kale){|food| [:a, :c, :d, :e, :k].sample} #=> :c
vitamins.values_at(:spinach, :cheese, :ham) #=> [:k, nil, :b1]
vitamins.assoc(:cabbage) #=> [:cabbage, :c]
vitamins.rassoc(:c)      #=> [:cabbage, :c]
```

Default Value

Retrieving a key from a `Hash` returns the corresponding value. If they key does not exist in the `Hash`, a *default value* is returned. The default value is normally `nil`, but can be overridden.

`Hash.new` instantiates a `Hash` with a default value. With no arguments, it is equivalent to `{}`, so has a default value of `nil`. `Hash.new(`*value*`)` sets *value* as the default. If `Hash.new` is provided a block, the block is invoked whenever a default value is required. It receives the `Hash` and missing key as arguments, and is expected to return the corresponding value. For efficiency, the block can store this value in the `Hash`, so future look-ups for the key bypass the block.

The default value of an existing `Hash` can be modified. `Hash#default=` sets the default value to its argument. `Hash#default_proc=` expects its argument

to be a Proc, which is equivalent to the block given to Hash.new. #default and #default_proc return the current default value and Proc, respectively.

```
h = {a: 1, b: 2}
h[:c] #=> nil
h.default #=> nil
h = Hash.new(0)
h[:c] #=> 0
h[:c] = 3
h[:c] #=> 3
h.default = ??
h[:d] #=> "?"
require 'prime'
next_prime = Hash.new do |h, k|
  prime = k.succ
  prime += 1 until prime.prime?
  h[k] = prime
end
[12, 17, 48, 200].map{|n| next_prime[n]} #=> [13, 19, 53, 211]
next_prime #=> {12=>13, 17=>19, 48=>53, 200=>211}
```

Insertion

The element set syntax, #[*key*]= *value*, stores the given key-value pair in the receiver. #store(*key*, *value*) behaves identically. If a value already exists for *key*, it is overwritten; the predicates described in Keys can be used to detect when this would happen.

```
h = {}
h[:key] = :value #=> :value
h.store("Key", "Value") #=> "Value"
h["Key"] = "Ring" #=> "Ring"
h #=> {:key=>:value, "Key"=>"Ring"}
```

#[]= and #store associate a key object with a value object. If the key is a String, it is duplicated and frozen before insertion. If the key is another mutable object, it is possible to modify it after storing it in the Hash. However, this is inadvisable because the corresponding value will still be associated with the original key object, defeating attempts to perform look-ups with the modified key. This can be worked around with the #rehash described below,

but a better solution is to freeze mutable keys before insertion, or at least treat them as if you had.

`#rehash` re-indexes the receiver based on the current values of its keys. This is only necessary if the value of a key object has changed since it was inserted into the `Hash`. If `#rehash` is called while the `Hash` is being iterated, an `IndexError` is raised.

```
coords = {[0,0] => :origin}
treasure = [32,19]
coords[treasure] = :gold
treasure[-1] += 1
coords #=> {[0, 0]=>:origin, [32, 20]=>:gold}
coords[treasure] #=> nil
coords.rehash
coords[treasure] #=> :gold
```

Deletion

`#delete(`*key*`)` deletes *key* and its corresponding value, then returns the latter. It returns `nil` if the key didn't exist; or, if a block is given, the value of the block when invoked with *key* as an argument. `#shift` deletes and returns the first key-value pair from the receiver. If the `Hash` is empty, returns the default value for a key of `nil`.

`#delete_if` passes each key-value pair to its associated block in turn, deleting those for which the block is true. An `Enumerator` is returned if the block is omitted. `#reject!` is identical, expect it returns `nil` if no changes occurred. `#reject` is equivalent to invoking `#delete_if` on a copy of the receiver, then returning that copy.

The inverse of `#delete_if` is `#keep_if`, which deletes each key-value pair for which the block is false. An `Enumerator` is returned if the block is omitted. `#select!` is equivalent, except it returns `nil` if no changes occurred. `#select` returns a new `Hash` containing the key-value pairs for which its associated block returned true.

`#clear` deletes all of the receiver's key-value pairs, returning the empty `Hash`. The receiver's taintedness, trust, and default value are preserved. Similarly, `#replace(`*hash*`)` replaces all key-value pairs in the receiver with those in *hash*.

```
elements = {H: :hydrogen,  Li: :lithium, Na: :sodium, K: :potassium,
            Rb: :rubidium, Cs: :caesium, Fr: :francium}
elements.delete(:Fr) #=> :francium
elements.delete(:O){ false } #=> false
elements.shift #=> [:H, :hydrogen]
elements.delete_if{|sym, name| sym.size == 1}
#=> {:Li=>:lithium, :Na=>:sodium, :Rb=>:rubidium, :Cs=>:caesium}
elements.reject{|sym, name| name =~ /um/} #=> {}
elements.select!{|sym, name| name.to_s.end_with?(?m)} #=> nil
elements.replace({li: 'list item', p: 'paragraph', hr: 'horizontal rule'})
#=> {:li=>"list item", :p=>"paragraph", :hr=>"horizontal rule"}
```

Iteration

`Hash#each`, and its alias `#each_pair`, calls its associated block with each key-value pair in turn. `#each_key` and `#each_value` call the associated block with each key and value, respectively. Each method returns the receiver, or an `Enumerator` if the block is omitted.

```
music = {LP: :long_play, EP: :extended_play, CD: :compact_disc}
music.each.map{|k, v| "#{k} means '#{v}'"}
["LP means 'long_play'", "EP means 'extended_play'", "CD means 'compact_disc'"]
music.each_key{|acronym| print acronym}
# LPEPCD
#=> {:LP=>:long_play, :EP=>:extended_play, :CD=>:compact_disc}
```

Keys

`#key(`*value*`)` returns the first key whose value is *value*. If a `Hash` contains a given key, the `#key?(`*key*`)` predicate returns `true`; otherwise, it returns `false`. The `#has_key?`, `#member?`, and `#include?` methods behave identically.

`#keys` returns the receiver's keys as an `Array`. `#each_key` returns an `Enumerator` of the same. If `#each_key` is given a block, it yields each key in turn, then returns the receiver.

```
snomed = {T: :Topography, M: :Morphology, L: :Living_organisms, C: :Chemical,
          F: :Fever, J: :Occupation, D: :Diagnosis, P: :Procedure,
          A: :Physical,  S: :Social_context, G: :General}
snomed.key :General #=> :G
snomed.key? :D #=> true
snomed.include? :E #=> false
snomed.keys #=> [:T, :M, :L, :C, :F, :J, :D, :P, :A, :S, :G]
snomed.each_key.reject{|k| snomed[k] =~ /^#{k}/} #=> [:J, :A]
```

Values

The value `#value?(`*value*`)` predicate, and it alias, `#has_value?`, return `true` if one or more keys have a value of *value*.

`#values` returns the receiver's values as an `Array`, while `#each_value` returns an `Enumerator` of the same. If `#each_value` is given a block, it yields each value in turn, then returns the receiver. To retrieve the value(s) associated with one or more keys, see Look-up.

```
snomed = {T: :Topography, M: :Morphology, L: :Living_organisms, C: :Chemical,
          F: :Fever, J: :Occupation, D: :Diagnosis, P: :Procedure,
          A: :Physical,  S: :Social_context, G: :General}
snomed.value? :Diagnosis #=> true
snomed.value? :Imaginary #=> false
snomed.values #=> [:Topography, :Morphology, :Living_organisms, :Chemical, :Fever, :Occupati
#=> :Diagnosis, :Procedure, :Physical, :Social_context, :General]
snomed.each_value.reject{|v| v =~ /^#{snomed.key(v)}/} #=> [:Occupation, :Physical]
```

Transformations

`#flatten(`*depth* `= 1)` converts the receiver to an `Array`, on which it invokes `Array#flatten!`, and returns. The result is an `Array` whose first element is the first key, second element is the first value, third element is the

second key, and so on. The *depth* arguments controls how deep `Array#flatten!` recurses.

`#invert` returns a `Hash` whose keys are the receiver's values, and whose value's are the receiver's keys. As keys are unique, if the original `Hash` contained multiple keys with the same value, only the last[2] of these pairs is preserved.

```
demonyms = {Barbados: :Bajan, Spain: :Spaniard, Ireland: :Irish,
            Australia: [:Australian, :Aussie], Peru: :Peruvian}
demonyms.flatten
#=> [:Barbados, :Bajan, :Spain, :Spaniard, :Ireland, :Irish,
#=>  :Australia, [:Australian, :Aussie], :Peru, :Peruvian]
demonyms.flatten 2
#=> [:Barbados, :Bajan, :Spain, :Spaniard, :Ireland, :Irish,
#=>  :Australia, :Australian, :Aussie, :Peru, :Peruvian]
demonyms.invert
#=> {:Bajan=>:Barbados, :Spaniard=>:Spain, :Irish=>:Ireland,
#=>  [:Australian, :Aussie]=>:Australia, :Peruvian=>:Peru}
```

Merging

`#merge(`*hash*`)` returns a new `Hash` containing the key-value pairs of the receiver plus those of *hash*. If duplicate keys are encountered, the corresponding value from `hash` is used. Alternatively, if a block is supplied it is called for each duplicate key with three arguments—the key, the receiver's value, and *hash*'s value—and its return value becomes the value of the key. `#merge!` and its alias `#update` behave identically, except they modify the receiver in-place.

```
currencies = {ruble: :Russia, dollar: :Fiji, euro: :Malta, zloty: :Poland}
currencies.update(dollar: :Taiwan, euro: :Spain, franc: :Switzerland)
#=> {:ruble=>:Russia, :dollar=>:Taiwan, :euro=>:Spain,
#=>  :zloty=>:Poland, :franc=>:Switzerland}
```

---

2. Thomas claims that "If `hsh` has duplicate values, the result will contain only one of them as a key—which one is not predictable." This was true in Ruby 1.8, but now that a `Hash` iterates in order of insertion, this behaviour can be predicted.

```
currencies.merge(krona: :Sweden, won: :South_Korea, euro: :Italy){|currency, old, new| old}
#=> {:ruble=>:Russia, :dollar=>:Taiwan, :euro=>:Spain, :zloty=>:Poland,
#=>  :franc=>:Switzerland, :krona=>:Sweden, :won=>:South_Korea}
```

Size

The size of a Hash is the number of key-value pairs it contains. It is returned by #size and its alias #length. A Hash of size 0, is *empty*, and can be tested with the #empty? predicate.

```
h = {}
h.size   #=> 0
h.empty? #=> true
h[3] = 4
h[4] = 5
h[5] = nil
h.size   #=> 3
h.empty? #=> false
```

Sorting

#sort converts the receiver to an Array, whose elements are themselves Arrays of the form [*key*, *value*], sorts it with Array#sort, then returns the result.

```
old_capitals = {Scotland: [:Scone, :Edinburgh], Denmark: [:Roskilde, :Copenhagen],
                Finalnd: [:Turku, :Helsinki],   Sweden: [:Sigtuna, :Stockholm]}
old_capitals.sort
#=> [[:Denmark, [:Roskilde, :Copenhagen]], [:Finalnd, [:Turku, :Helsinki]],
#=>  [:Scotland, [:Scone, :Edinburgh]],    [:Sweden, [:Sigtuna, :Stockholm]]]
{}.sort #=> []
```

Equality

Two Hash objects are equal only if all the following conditions hold true:

- They contain the same number of keys.

- They contain the same keys.
- Each value in the receiver is equal, using #==, to the corresponding value in the argument Hash.

#== returns true if its argument is a Hash and equal to the receiver. If the argument is not a Hash but responds to #to_hash, it is sent #== with the receiver as the argument. Otherwise, Hash#== returns false.

```
{} == {} #=> true
{foo: :bar, "foo" => :bar} == {foo: :bar, "foo" => :bar} #=> true
{a: 1, b: 2} == {b: 2, a: 1} #=> true
{a: 1, b: 2} == {a: 1} #=> false
Hash.new(8) == {} #=> true
{key: :value} == [:key, :value] #=> false
a = {a: 1}
a[:b] = a
a == a.merge(b: a) #=> true
```

Coercion

An object that responds to :to_hash may be implicitly converted to a Hash. Hash[] is another approach. If given an object convertible to a Hash, it performs the conversion and returns the new Hash; otherwise, when given an even number of arguments, it interprets them as key-value pairs-the first argument being the first key, the second argument being its corresponding value, and so forth-with which it creates a new Hash. Hash.try_convert coerces its argument with :to_hash, if possible, or returns nil.

A Hash may be converted to an Array with Hash#to_a. Each element of the Array is itself an Array, whose first element is a key, and second element the corresponding value.

#to_s, and its alias #inspect, return a String of the following form for a Hash of size *n*, with *key* and *value* generated by #inspect:

{*key*$_0$ => *value*$_0$, …, *key*$_n$ => *value*$_n$}

Recursive Hash objects are represented as {...}, while {} stands for the empty Hash.

```
{}.to_hash #=> {}
Hash[*(1..6)] #=> {1=>2, 3=>4, 5=>6}
{a: :value, b: [:key]}.to_a #=> [[:a, :value], [:b, [:key]]]
Hash.try_convert(try: :again) #=> {:try=>:again}
Hash.try_convert([:element]) #=> nil
h = { Object.new => Rational(2, 3) }
h[:h] = h
h.to_s #=> "{#<Object:0x00000001388e38>=>(2/3), :h=>{...}}"
```

Identity

`Hash` keys are compared with `#eql?` by default, so two keys are equal if they have the same value. An *identity `Hash`* regards keys as equal only if they are the same object. This is another argument in favour of `Symbol` keys, because two `Symbol`s with the same value are the same object. When `String`s are used as keys, Ruby duplicates them before use. Therefore, in an identity `Hash`, `String` key look-ups won't work. A regular `Hash` is converted to an identity `Hash` with `#compare_by_identity`, which returns the receiver. The `#compare_by_identity?` predicate returns `true` if the receiver is an identity `Hash`; `false`, otherwise.

```
rat = {Rational(1,2) => 0.5, Rational(3,4) => 0.75}
rat[Rational(3,4)] = '3/4'
rat #=> {(1/2)=>0.5, (3/4)=>"3/4"}
rat.compare_by_identity[Rational(1,2)] = '1/2'
rat #=> {(1/2)=>0.5, (3/4)=>"3/4", (1/2)=>"1/2"}
rat.compare_by_identity? #=> true
feet = {cat: :paws, pigs: :hooves, bird: :claws}.compare_by_identity
feet[:pigs] = :cloven_hooves
feet #=> {:cat=>:paws, :pigs=>:cloven_hooves, :bird=>:claws}
```

RANGES

A Range represents an immutable sequence between two given values. The first of which is its start-point, and the second its end-point. For example, 0–9 is a range consisting of all single-digit integers, expressed by specifying the start-point and endpoint. Range is Enumerable, so in addition to the methods described below, it also supports the Enumerable API.

A Range is either *inclusive* or *exclusive*: the former includes the endpoint; the latter does not. The Range#exclude_end? predicate returns true if its receiver is exclusive; false otherwise.

Instantiation

A Range literal consists of two values separated by two or three full stops: the former range is inclusive; the latter exclusive. It returns a new Range object representing the given interval. To use the literal as a receiver, enclose it within parentheses; otherwise, the message is sent to the Range endpoint rather than the Range itself. However, in the conditional of a branching or looping statement, two or three consecutive full stops don't create a range; they constitute a Boolean flip-flop instead.

```
(0..9).to_a
#=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
(0...9).to_a
#=> [0, 1, 2, 3, 4, 5, 6, 7, 8]
?a..?z
```

Alternatively, a Range can be created with Range.new. The start-point is supplied as the first argument, and the endpoint as the second. If a third argument is given, and it is true, the Range is inclusive; otherwise, it is exclusive.

```
Range.new(0, 9).to_a
#=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Range.new(0, 9, true).to_a
```

```
#=> [0, 1, 2, 3, 4, 5, 6, 7, 8]
Range.new(?a, ?z)
```

Start-points & End-points

The start-point and end-point must be comparable with #<=>. That is, for a Range *start*..*end*, *start* <=> *end* must return -1, 0, or 1.

A range is *discrete* if it begins with a value that responds to #succ by returning the next element of the sequence. It is so called because it represents a finite set of values that can be iterated over. A non-discrete range is *continuous*. It represents an infinite set of values, therefore a TypeError is raised when attempting to iterate over it. For instance, Integer objects respond to #<=> and #succ, so a Range between Integer values is discrete. Conversely, Float responds to <=> but not to #succ, so represents a continuous range.

The start-point is returned by Range#begin, while the end-point is returned by Range#end. #first and #last behave identically to #begin and #end, respectively, when called without arguments. If a numeric argument is given, #first returns that many elements from the beginning of the Range as an Array, while #last returns an Array comprising that many from the end. However, in this form both #first and #last require the Range to be discrete; raising an ArgumentError if not.

```
Range.new(1.0, 3.0).begin #=> 1.0
(?a...?z).end #=> "z"
(:abc..:cba).first(10)
#=> [:abc, :abd, :abe, :abf, :abg, :abh, :abi, :abj, :abk, :abl]
(Rational(1,5)..1).first #=> (1/5)
```

Membership Testing

There are two principle ways to test whether a given value is a member of a Range. Range#cover?(*value*) uses a simple inequality. For a Range between *a* and *b*, #cover? tests $a \le value \le b$ if the receiver is inclusive; $a \le value < b$,

otherwise. If the test succeeds, `true` is returned; if it fails, `false` is returned instead.

`Range#include?(`*`value`*`)`, and its alias `#member?`, work in the same way when the end-points are numeric. Otherwise, they test for membership through iteration: each successive element of the `Range` is enumerated until either the element is equal—using `#==`—to *value*, or the end of the `Range` is reached. Accordingly, unless the start-point and end-point are numeric, `#include?` requires the receiver to be discrete. Again, `true` is returned if the test succeeds; `false` otherwise. `Range#===` is implemented in terms of `#include?`, so these restrictions also apply when using a `Range` in a `when` statement.

```
ages = 18..30
ages.include? 505 #=> false
ages.include? 25.0 #=> true
ages.include? 14 #=> false
(18...30).include? 30 #=> false
(Rational(1,10)..Rational(10, 1)).cover? 3 #=> true
(:above..:below).include? :angels #=> false
(:above..:below).cover? :angels #=> true
```

Iteration

`Range#each` yields successive elements of the receiver to a block if one is given; otherwise, returns them as an `Enumerator`. `#succ` is used to generate elements. `Range#step(`*`n`*`=1)` yields to a block each n^{th} element. If the start-point and end-point are numeric, elements are generated through addition; otherwise, `#succ` is used. As with `#each`, an `Enumerator` is returned when the block is omitted.

```
(-1..1).each {|n| print "%2d" % n}
# -1 0 1
(1.0..2.0).step(0.1).reduce(:+) #=> 16.5
```

Equality

Two Ranges are equal if they have the same start-point and end-point, and either both exclusive or both inclusive. Range#== compares the start-points and end-points with #==, while Range#eql? compares them using #eql?.

```
(5..6) == (5..6) #=> true
(:an..:other) == (:an...:other) #=> false
(-Float::INFINITY..Float::INFINITY).eql?(Range.new(0, Float::INFINITY)) #=> false
```

FILES & DIRECTORIES

Files

A file is an entity on a filesystem addressable by a path. It can be represented as an instance of the `File` class, which provides a high-level API for reading, writing to, and querying and manipulating the metadata of, files. `File` is a subclass of IO, so the latter's methods are available in addition to those defined by `File`.

Paths

A file's *path* is a `String` describing its location in the filesystem. It consists of one or more *components*, which in the latter case are separated with the *path separator*. On non-Windows platforms the path separator is the solidus character. Ruby expects paths to be given in the Unix-style, but automatically uses the correct path separator for the platform it is running on.

We use the term *path* to refer either to a `String`, whose contents is a path, or an object that responds to `:to_path` with such a `String`. A method that expects a path as an argument will accept either of these representations.

`File.path` returns the non-normalised path of its argument. If its argument is a `String`, it will be returned as-is, otherwise the result of sending the argument `:to_path` will be returned. The path associated with a `File` object is returned by `File#path`.

`File.dirname` returns the directory name of its argument: all components other than the last. Conversely, the last component of a filename is returned by `File.basename`. The filename extension, e.g. `.txt`, is returned by `File.extname`. If these methods are given a path without the requested

component, they return the empty `String`. `File.split` returns a two-element `Array` with the file's dirname as the first element, and its basename as the last. `File.join` performs the inverse operation: given a list of path components it joins them with the path separator into a `String`.

In the table that follows, each column indicates the output of the corresponding `File` class method for the path given in the first column.

path	dirname	basename	extname	split
/home/user/base64.rb	/home/user	base64.rb	.rb	["/home/user", "base64.rb"]
/etc/hosts	/etc	hosts		["/etc", "hosts"]
sort	.	sort		[".", "sort"]
~/.vimrc	~	.vimrc		["~", ".vimrc"]

Examples of how path components are interpreted by the named class methods of `File`

A *relative path* is a path given in terms of another path, as opposed to an *absolute path* which stands alone. On Unix-like systems, the former are paths that do not begin with a solidus, and the latter are the opposite.

A relative path may be expanded to an absolute path with `File.expand_path`. Absolute paths are returned as-is. Otherwise, they are assumed to be relative to the current directory. If an optional second argument is given, it names the directory the first argument is relative to. If the relative path begins with a tilde, it is interpreted as relative to the current user's home directory (as returned by `ENV['HOME']`). If the relative path begins with a tilde followed by a username, it is relative to the named user's home directory. `File.absolute_path` behaves in the same fashion except it doesn't treat tildes specially, i.e. it interprets a path of `~/glark` as `./~/glark`.

```
Dir.chdir('/etc') do
  File.expand_path '/etc/resolv.conf' #=> "/etc/resolv.conf"
  File.expand_path 'resolv.conf' #=> "/etc/resolv.conf"
  File.expand_path '../var/log/messages' #=> "/var/log/messages"
  File.expand_path './filesystems', '/proc' #=> "/proc/filesystems"
  File.expand_path '~/.bashrc' #=> "/home/run/.bashrc"
  ENV['HOME'] = '/tmp'
  File.expand_path '~/.bashrc' #=> "/tmp/.bashrc"
```

```
  File.expand_path '~root' #=> "/root"
end
```

Neither `File.absolute_path` nor `File.expand_path` resolve relative paths by traversing the filesystem: at most, they request the current working directory or a user's home directory from the environment. An implication is that they will quite happily return paths that do not exist on the filesystem. `File.realpath` is an alternative. It creates an absolute path from a relative path by interrogating the filesystem, and following symbolic links if necessary. It resolves relative to the current working directory, or the directory given by the optional second argument. An exception is raised if the absolute path it resolves does not exist. `File.realdirpath` accepts the same arguments, and performs the same operation, but allows the last component of the path to be non-existent.

```
Dir.chdir('/etc') do
  File.realpath '/etc/resolv.conf' #=> "/etc/resolv.conf"
  File.realpath 'resolv.conf' #=> "/etc/resolv.conf"
  File.readlink '/etc/alternatives/www-browser' #=> "/usr/bin/w3m"
  File.realpath '../etc/alternatives/../alternatives/ruby'
  # No such file or directory - /etc/alternatives/ruby (Errno::ENOENT)
  File.realdirpath '../etc/alternatives/../alternatives/ruby' #=> "/etc/alternatives/ruby"
end
```

Reading

Reading a file is the process of retrieving its contents. It can be achieved by passing a path argument to `File.read` which returns the contents as a `String`. An optional second argument specifies the number of bytes to read, and an optional third argument specifies the offset from which to begin reading. An options Hash may be supplied as the final argument. `File.binread` takes the same arguments but reads the file in binary mode.

```
File.read('/etc/timezone') #=> "Europe/London\n"
File.read('/etc/timezone', 6) #=> "Europe"
File.read('/etc/timezone', 6, 7) #=> "London"
```

Opening

Opening a file enables one to read from, write to, or otherwise manipulate and query, the resource. `Kernel.open` opens the file with the given path, returning a corresponding `File` object. It is typically used with a block, which receives the opened `File` object as an argument. Having executed the block, the file is automatically closed, even if the block raises an exception. The block-form has a return value equal to that of the block; otherwise a new `File` object is returned. Both forms accept an options Hash as their final argument.

```
# coding: utf-8
open('/tmp/file', mode: ?w) {|f| f.print "text\r\n"}
text = File.read('/tmp/file') #=> "text\r\n"
text.encoding == Encoding::UTF_8
open('/tmp/file', mode: ?a){|f| f << "more text"}
open('/tmp/file', textmode: true, external_encoding: 'ascii') do |f|
  text = f.read #=> "text\nmore text"
  text.encoding #=> Encoding::US_ASCII
end
```

Existence

It is often necessary to determine whether a given file exists, as methods that accept path arguments may raise exceptions otherwise. To this end Ruby provides the predicate `File.exists?`, and its alias `File.exist?`.

Deletion

Files may be deleted by supplying a list of their paths as arguments to `File.delete`, or its alias `File.unlink`. An `Integer` is returned indicating how many files were deleted. `File.truncate`, which is not available on all platforms, truncates a given file to a given number of bytes. It expects a path as its first argument, and the number of bytes as an `Integer` for its second. `File#truncate` behaves in the same fashion, but truncates its receiver to the size given as its sole argument.

Renaming

Files may be renamed by invoking `File.rename` with their current path as the first argument, and their desired path as the second.

Size

A file that exists has a size. The `File.size` method returns the size of its argument, and `File#size` returns that of its receiver. In both cases, the size is an `Integer` and in bytes. The `File.size?` predicate returns the size of the file named by its argument if it exists and has a non-zero size, or otherwise: `nil`. Similarly, `File.zero?` returns `true` if the file named by its argument exists and has a size of zero; `false` otherwise.

Comparison

The `File.identical?` predicate is used to determine whether the two files it is given as arguments are the same: returning `true` if they are, `false` otherwise. Two files are considered identical if their paths normalise to the same path, or if one or both are symbolic links with identical targets. It is not sufficient that two files merely contain the same content for this method to succeed.

File::Stat

`File::Stat` objects represents file metadata. They are normally created with `IO#stat`, `File#stat` or `File.stat(`*`file`*`)`. `File#lstat` and `File.lstat` are used to the same end, but they do not follow the last symbolic link, if any, in the file path; they return metadata for the link itself.

Attribute	Method	Returns	Kernel.test
Last access time	`File::Stat#atime` / `File.atime` / `File#atime`	`Time`	`?A`

Attributes of files

Attribute	Method	Returns	Kernel.test
Preferred block size for I/O	`File::Stat#blksize`	`Fixnum` or `nil`	
Number of blocks allocated	`File::Stat#blocks`	`Fixnum` or `nil`	
Inode change time	`File::Stat#ctime` / `File.ctime` / `File#ctime`	`Time`	`?C`
Device number of filesystem	`File::Stat#dev`	`Fixnum` or `nil`	
〃 (major part)	`File::Stat#dev_major`	`Fixnum` or `nil`	
〃 (minor part)	`File::Stat#dev_minor`	`Fixnum` or `nil`	
Type	`File::Stat#ftype`	`String`	
Owner's group ID	`File::Stat#gid`	`Fixnum`	
Inode number	`File::Stat#ino`	`Fixnum`	
Permission bits	`File::Stat#mode`	`Fixnum`	
Last modify time	`File::Stat#mtime` / `File.mtime` / `File#mtime`	`Time`	`?M`
Pathname	`File#path`	`String`	
Device ID	`File::Stat#rdev`	`Fixnum` or `nil`	
〃 (major part)	`File::Stat#rdev_major`	`Fixnum` or `nil`	
〃 (minor part)	`File::Stat#rdev_minor`	`Fixnum` or `nil`	
Size (bytes)	`File::Stat#size` / `File#size` / `File.size`	`Fixnum`	
Owner's user ID	`File::Stat#uid`	`Fixnum`	

Types

To determine whether a given file is of a given type, one may use the appropriate predicate method of the `File` class. For example, `File.directory?` determines whether its argument is a directory. Alternatively, `File::Stat#ftype` returns a `String` identifying the type of the represented file. In the case of a directory, `#ftype` returns `directory`. These two approaches are summarised in the table that follows.

In the table below, the *`File::Stat#ftype`* column contains the `String` that method returns for a file of the corresponding type. *Predicate* is a method that expects a path as argument, returning `true` iff the named file is of the corresponding type. The *Creation* column specifies, where possible, how a file of the corresponding type may be created in Ruby. Examples assume a Linux/Debian platform.

Description	`File::Stat#ftype`	Predicate	Example	Creation	Kernel.test
Block device	`blockSpecial`	`File.blockdev?`	`/dev/sda`		`?b`
Character device	`characterSpecial`	`File.chardev?`	`/dev/tty`		`?c`
Directory	`directory`	`File.directory?`	`/etc`	`Dir.mkdir`	`?d`
FIFO (named pipe)	`fifo`	`File.pipe?`		`system("mkfifo` *`name`*`")`	`?p`
Regular file	`file`	`File.file?`	`/etc/passwd`	`Kernel.open`	`?f`
Symbolic link	`link`	`File.symlink?`	`/dev/root`	`File.symlink`	`?l`
Socket	`socket`	`File.socket?`	`/dev/log`	Use the `socket` library	`?S`
Another type of file	`unknown`				

Types of file Ruby knows about

Permissions

The permission bits associated with a file may be retrieved from the corresponding `File::Stat` object. The fields of interest are described below. Be aware that on non-POSIX systems the semantics of file permissions are quite different, and are likely to be less granular. To compensate, on these disadvantaged platforms Ruby attempts to synthesize a numeric mode such that it is broadly similar to what POSIX requires. However, given the lack of granularity and other incompatibilities, these factitious modes are cumbersome and error-prone. It is therefore recommended that the

permission bits are treated as opaque, as far as possible, and the predicates described later are used instead.

Attribute	Method	Returns
Owner's group ID	`File::Stat#gid`	`Fixnum`
Permission bits	`File::Stat#mode`	`Fixnum`
Owner's user ID	`File::Stat#uid`	`Fixnum`

Permission and ownership attributes of `File::Stat` objects

The permissions of a file may be changed with `File.chmod` by providing the new permission bits as the first argument, and a list of files as the subsequent arguments. Each file listed has its permissions changed accordingly. The meaning of the permission bits is platform-specific, but on Unix-like systems they correspond to the numeric mode understood by `chmod(1)`. Some systems, such as the BSDs, support a variant of `chmod` that does not follow symbolic links, therefore acts upon the link rather than its target. If available, Ruby exposes this functionality via `File.lchmod`, which takes the same arguments as `File.chmod`. `File` also provides the instance methods `File#chmod` and `File#lchmod`, which require the permission bits to assign to their receiver as their sole argument.

The owner and group of a file may be changed-although the former requires root permissions-with `File.chown`. It takes a numeric user ID, and numeric group ID, as the first and second arguments, respectively, and a list of files as the subsequent arguments. The ownership of each file listed that the user has permission to change is adjusted accordingly. `File.lchown` functions similarly, but does not follow symlinks. Its availability mirrors that of `File.lchmod`. The instance method, `File#chown`, requires a numeric user ID, and numeric group ID, as its arguments, which it applies to its receiver.

The umask value of the current process is set by supplying `File.umask` with the new value as an `Integer` argument. It returns the previous umask value. When called without arguments, `File.umask` returns the current umask.

In the table below, methods take a filename as argument and return either `true` or `false` unless stated otherwise. The *`Kernel.test`* column indicates the corresponding command for use with `test`.

Predicate	Test	Kernel.test
`File.executable?`	Executable by our effective user ID?	`?x`
`File.executable_real?`	Executable by our real user ID?	`?X`
`File.grpowned?`	Owned by our effective group ID?	`?G`
`File.owned?`	Owned by our effective user ID?	`?o`
`File.readable?`	Readable by our effective user ID?	`?r`
`File.readable_real?`	Readable by our real user ID?	`?R`
`File.setgid?`	Setgid bit set?	`?g`
`File.setuid?`	Setuid bit set?	`?u`
`File.sticky?`	Sticky bit set?	`?k`
`File.world_readable?`	Readable by others? (Returns permission bits or `nil`)	
`File.world_writable?`	Writable by others? (Returns permission bits or `nil`)	
`File.writable?`	Writable by our effective user ID?	`?w`
`File.writable_real?`	Writable by our real user ID?	`?W`

Permissions predicate methods of the `File` class

Links

On platforms that support symlinks, `File.symlink` will create a link to the file named as its first argument, with the name given as its second. `File.link` takes the same arguments, but creates hard links instead. The `File.symlink?` predicate can be used to test whether its argument is a symlink, and `File.readlink` returns the target of a given link.

```
link, target = '/tmp/link', '/tmp/target'
File.unlink(link) if File.exist?(link)
open(target, mode: ?w) {}
File.symlink(target, link)
File.symlink?(target) #=> false
File.symlink?(link)   #=> true
File.readlink(link)   #=> "/tmp/target"
```

Locks

`File#flock(`*`operation`*`)` places or removes an advisory lock on the receiver's file, where *operation* is a logical OR of the constants below. A lock is either *exclusive* or *shared*: the former may only be held by a single process for a given file; the latter may be held by multiple processes. A single file can only have one type of lock.

Constant	Meaning
`File::LOCK_EX`	Place an exclusive lock on the file.
`File::LOCK_NB`	Don't block when locking.
`File::LOCK_SH`	Place a shared lock on the file.
`File::LOCK_UN`	Remove the lock on the file.

`#flock` blocks if attempting to lock a file that has an incompatible lock, e.g. if a file has an exlcusive lock, another process that attempted to place an exclusive lock on the same file would block until the first process released its lock. Alternatively, if *operation* includes `File::LOCK_NB` and `#flock` would have blocked, `false` is returned immediately. Otherwise, `#flock` returns `0`. When all file descriptors associated with a locked file have been closed, the lock is automatically released.

Filename Matching

`File.fnmatch?`, and its alias `File.fnmatch`, determine whether a given globbing pattern matches a given filename. The pattern is given as the first argument, and the pathname as the second. The optional third argument is a bitmask of flags, which are explained below in Globbing. If the pathname matches the pattern, `true` is returned; otherwise `false`.

Kernel.test

`Kernel.test` performs a given test on a given file. It accepts `2–3` arguments, the first of which is a single character command, and the remainder are the files on which to perform the test.

Command	Description	Returns	Arity
`?A`	Last access time	`Time`	1
`?b`	Block device?	`true` or `false`	1
`?c`	Character device?	`true` or `false`	1
`?C`	Last status change time	`Time`	1
`?d`	Directory?	`true` or `false`	1
`?e`	Exists?	`true` or `false`	1
`?f`	Exists and a regular file?	`true` or `false`	1
`?g`	Has the `setgid` bit set?	`true` or `false`	1
`?G`	Exists and owned by our group?	`true` or `false`	1
`?k`	Sticky bit set?	`true` or `false`	1
`?l`	Exists and a symlink?	`true` or `false`	1
`?M`	Last modification time	`Time`	1
`?o`	Owned by our effective user ID?	`true` or `false`	1
`?O`	Owned by our real user ID?	`true` or `false`	1
`?O`	Owned by our real user ID?	`true` or `false`	1
`?p`	A FIFO?	`true` or `false`	1
`?r`	Readable by our effective user/group ID?	`true` or `false`	1
`?R`	Readable by our real user/group ID?	`true` or `false`	1
`?s`	Returns the size if non-zero	`Integer` or `nil`	1

Commands supported by `Kernel.test`

Command	Description	Returns	Arity
`?S`	Socket?	`true` or `false`	1
`?u`	`setuid` bit set?	`true` or `false`	1
`?w`	Exists and writable by our effective user/group ID?	`true` or `false`	1
`?W`	Exists and writable by our real user/group ID?	`true` or `false`	1
`?x`	Exists and executable by our effective user/group ID?	`true` or `false`	1
`?X`	Exists and executable by our real user/group ID?	`true` or `false`	1
`?z`	Exists with a size of zero?	`true` or `false`	1
`?-`	Are both operands identical?	`true` or `false`	2
`?=`	Do both operands have the same modification time?	`true` or `false`	2
`?<`	Is the modification time of the first operand prior to that of the second?	`true` or `false`	2
`?>`	Is the modification time of the first operand after that of the second?	`true` or `false`	2

Directories

Directory streams are represented by the `Dir` class. `Dir` is neither a subclass of `File` nor `IO`. As with files, directories are identified by pathnames which may be absolute or relative. The pathname `.` refers to the current working directory, and `..` its parent. The pathname associated with a `Dir` object is returned by `Dir#path`, or its alias `Dir#to_path`.

Working Directory

A Ruby process has the notion of a *current working directory*, an absolute directory path from which relative paths are resolved. This is returned by `Dir.pwd`, or its alias `Dir.getwd`, as a `String`. It can be changed by invoking

`Dir.chdir` with the new working directory as an argument. If the argument is omitted, it defaults to `ENV['HOME']` or `ENV['LOGDIR']`, whichever is set.

If `Dir.chdir` is called with a block, it changes the working directory as before, but resets it to its original value at the end of the block. The block is passed the new directory as an argument, and its return value becomes that of `Dir.chdir`. These blocks may be nested, but the official documentation cautions that in multi-threaded programs a thread must not open a `Dir.chdir` block while another thread is inside one.

Home Directory

The home directory of the current user is returned by `Dir.home`. This is the value of `ENV['HOME']`, or the result of expanding ~. The home directory of an arbitrary user can be obtained by passing their username as an argument. An `ArgumentError` is raised if that user does not exist.

Instantiation

A `Dir` object can be instantiated by supplying `Dir.new` with a directory name argument. `Dir.open` behaves the same way, but if its supplied with a block, it yields the new `Dir` object to the block, then ensures it is closed when the block finishes. The return value of this form is that of the block.

Both forms assume the default filesystem encoding, but this can be overridden by supplying a second argument of `encoding:` *`encoding`*, where *encoding* is the name of an encoding as a `String` or an `Encoding` object.

If the block form is not used, the directory handle should be closed explicitly with `Dir#close`.

Entries

A directory's *entries* are the names of the files and other directories which it contains. On a Unix-like system the first two entries for every directory are `.` and `..`, which refer to the current directory and its parent directory, respectively.

`Dir.entries` returns the entries of the directory named by its argument as an `Array` of `Strings`, or raises a `SystemCallError` if it doesn't exist. `Dir.foreach` also accepts a directory name as an argument, but returns an `Enumerator` of its entries. If called with a block, `Dir.foreach` yields each entry in turn.

The entries of a directory represented by a `Dir` object, are returned by `Dir#each`. An `Enumerator` is returned if no block is given, otherwise each entry is yielded in turn. Indeed, `Dir` mixes-in the `Enumerable` module, so all of its methods are available for manipulating these entries.

Alternatively, a `Dir` instance can be treated like an `Enumerator` by calling `Dir#read` to return the next entry. `nil` is returned when no entries remain. `Dir#rewind` resets the stream such that the next call to `Dir#read` will return the first entry again.

Creation

A directory can be created by supplying its name as an argument to `Dir.mkdir`. An optional second argument specifies the permissions of the new directory. These are platform-specific permission bits, in the same form as `IO` and `File` accept them. They respect the current umask value, and are completely ignored on Windows.

Existence

The `Dir.exist?` predicate, and its alias `Dir.exists?`, return `true` if their argument is both an existing path and a directory; `false` otherwise.

Deletion

`Dir.rmdir`, and its aliases `Dir.delete` and `Dir.unlink`, delete the directory named by their argument. However, the directory must already be empty, otherwise an exception is raised.

Globbing

`Dir.glob` accepts a pattern, similar in form to a shell glob, and returns an `Array` of matching filenames. Characters in a pattern match themselves unless they take one of the following forms.

`*` (Asterisk)
: An asterisk matches zero or more characters. It does not match a leading full stop unless the FNM_DOTMATCH flag is given.

`?` (Question Mark)
: A question mark matches exactly one character. It does not match a leading full stop unless the FNM_DOTMATCH flag is given.

`**` (Double Asterisk)
: Two consecutive asterisks match zero or more directory components.

`[`*characters*`]` (Character Class)
: A sequence of characters enclosed in square brackets match any one of the specified characters. Two characters separated by a hyphen-minus sign (-) represent the inclusive range of characters between them. If the first character is a caret (^), the meaning of the class is inverted: it matches any character not specified.

{pat_0,…,pat_n} (Alternation)
: Curly brackets enclose one or more glob patterns separated by commas. Filenames matching any of the given patterns are matched.

`\` (Reverse Solidus)
: A reverse solidus escapes the metacharacter that follows it, removing its special significance. Matches itself if the FNM_NOESCAPE flag is given.

```
Dir.chdir('/tmp/animals') do
  Dir.entries(?.) #=> ["..", "."]
  Dir.mkdir('extinct')
  %w{ants hippopotamus elephants arachnids koalas accounts.csv
     accounts.csv.bak visitors.log rota-2010 extinct/velociraptor
     extinct/quagga extinct/aurochs extinct/dodo}.each{|file| open(file, ?w){} }
  Dir.glob('*ants') #=> ["elephants", "ants"]
```

```
  Dir.glob('rota-????') #=> ["rota-2010"]
  Dir.glob('a*s') #=> ["ants", "arachnids"]
  Dir.glob('**/a*s') #=> ["ants", "arachnids", "extinct/aurochs"]
  Dir.glob('extinct/*o[dc]*') #=> ["extinct/aurochs", "extinct/dodo", "extinct/velociraptor"
  Dir.glob('{*.*,rota-201[0-9]}')
  #=> ["accounts.csv", "accounts.csv.bak", "visitors.log", "rota-2010"]
end
```

The semantics of the match may be altered by supplying a bitmask of flags as the second argument. The flags are represented by the constants listed below. If multiple flags are given they should be combined with bitwise OR. For example, to specify that a pattern should treat `\` literally and ignore case, the second argument to `Dir.glob` should be `File::FNM_NOESCAPE | File::FNM_CASEFOLD`.

`File::FNM_NOESCAPE`
: Backslash matches itself. By default, a backlash is a metacharacter, escaping the metacharacter which follows it; this flag reduces it to a lowly literal character.

`File::FNM_PATHNAME`
: Prevent asterisk or question mark from matching a solidus. To match a literal solidus, the pattern must contain one when using this flag.

`File::FNM_DOTMATCH`
: Allow asterisk or question mark to match a leading full stop in a path component; by default leading full stops are only matched if specified literally.

`File::FNM_CASEFOLD`
: Ignore case when matching.

`Dir[]` takes a glob as its sole argument, returning matching filenames as an `Array`. It is equivalent to `Dir.glob` with a second argument of `0`.

Position & Seeking

Directory streams support seeking similarly to IO objects. The current position is returned by `Dir#tell`, or its alias `Dir#pos`, as an `Integer`. Initially,

this value is `0`, but is increased after every call to `Dir#read`. A specific position can be sought by supplying it as argument to `Dir#pos=` or `Dir#seek`. However, `Dir#read` increments the position in a seemingly arbitrary fashion, so the only sensible position to seek to is one previously returned by `Dir#pos`. `Dir#rewind` resets the position to `0`.

INPUT & OUTPUT

> Unless you're using artificial intelligence to model a solipsistic philosopher, your program needs some way to communicate with the outside world.
>
> —Wall00, pp. 20–22

The `IO` class provides an interface to input and output on the level of file descriptors, while its `File` subclass deals with files on a higher level of abstraction. Accordingly, `File`, which is discussed in Files & Directories, is often more appropriate, and easier to use.

> All input and output is done by reading or writing files, because all peripheral devices, even your terminal, are files in the file system. This means that a single interface handles all communication between a program and peripheral devices.
>
> —Kernighan84, pp. 201–202

In keeping with Unix's everything-is-a-file philosophy [Raymond03] , Ruby allows regular files, directories, block and character devices, symbolic links, hard links, sockets, unnamed pipes, and FIFOs (named pipes), to be treated conceptually the same. They can be read from and written to. Before operating on a file, we need to open it by providing the operating system with the pathname and asking permission to read it, write to it, or both. If our request is granted, we are provided with a integer *file descriptor* which we must use to refer to the file in future operations. We are allowed only to use the file in the manner, or access mode, we requested: it is illegal to write to a file opened for reading, or vice versa. With this file descriptor we can initialize an IO object, or *stream*[1], which provides an interface for accessing the corresponding file.

A stream containing binary data can have its binmode attribute set, which prevents it from being transcoded and associates it with the ASCII-8BIT encoding. Conversely, a stream containing textual data can have its textmode attribute set, which allows transcoding and newline normalisation.

Also, associated with a stream is a *byte offset*, or file position, which specifies where the next read or write should occur. "When a file is first opened, the file position is zero. Usually, as bytes in the file are read from or written to, byte-by-byte, the file position increases in kind. The file position may also be set manually to a given value, even a value beyond the end of the file." [Love07, pp. 9–10] . Attempting to read past the end of a file (EoF) will result in either a `nil` value or an exception being raised, depending on the method used. A file is appended to by writing to a position past its end. The size of a file may be reduced by truncating[2] it to a smaller size.

Finally, having used a file descriptor, we should *close* it to release it back to the operating system.

Standard Input, Output, & Error

> When it is started by the shell, a program inherits three open files, with file descriptors 0, 1, and 2, called the standard input, the standard output, and the standard error. All of these are by default connected to the terminal, so if a program only reads file descriptor 0 and writes file descriptors 1 and 2, it can do I/O without having to

---

1. Our using *stream* in this sense fits the spirit if not the letter of its existing meaning in I/O. Loosemore et al. [Loosemore07, pp. 220–221] , for example, differentiate strongly between operations performed on file descriptors and those performed on streams. However, their description of streams as providing a "higher-level interface, layered on top of the primitive file descriptor facilities." and their remark that "You can also initially open a connection as a file descriptor and then make a stream associated with that file descriptor.", so closely follows the model of `IO`, that our minor redefinition seems justified.
2. We use *truncate* in its POSIX sense, i.e. to refer to the behaviour of `truncate(2)`. On most file systems, this operation can be used to increase as well as decrease the size of a file, hence our clarification.

> open files. If the program opens any other files, they will have file descriptors 3, 4, etc.
>
> —Kernighan84, pp. 201–202

Ruby behaves in the same manner. The constants `STDIN`, `STDOUT`, and `STDERR`, are defined automatically to refer to the program's standard input, standard output, and standard error streams, respectively. The value of each constant is an `IO` object associated with the corresponding file descriptor. `IO#fileno` returns the associated file descriptor as an `Integer`.

```
STDIN.fileno  #=> 0
STDOUT.fileno #=> 1
STDERR.fileno #=> 2
```

Three global variables are also defined automatically-`$stdin`, `$stdout`, and `$stderr`-which initially hold the value of the corresponding constant. We have in fact been interacting with `$stdout` implicitly all along: `Kernel.puts` is equivalent to `$stdout.puts`. Similarly, `Kernel.warn` writes to `$stderr`. The reason for the existence of these global variables in addition to the aforementioned constants is that by assigning another `IO` object to one of the global variables, we can temporarily redirect it elsewhere. Then, if we want to restore the original behaviour, we can assign the corresponding constant to the variable.

```
puts "STDOUT" # Written to STDOUT
$stdout = STDERR
puts "STDERR" # Written to STDERR
$stdout = STDOUT
puts "STDOUT" # Written to STDOUT

# From the command-line:
# $ ruby stdout-redirection.rb
# STDOUT
# STDERR
# STDOUT
# $ ruby stdout-redirection.rb 2>/dev/null
# STDOUT
# STDOUT
```

Writing

To write to a stream, `IO#print` may be used. It takes any number of arguments, which it converts to `String`s before writing them to the receiver. `IO#puts` is similar, but ensures that what it writes ends with a newline. An idiomatic method for writing a single object to a stream is `IO#<<` because its selector is polymorphic. All three methods are implemented in terms of `IO#write`, as are most other methods that perform writing. `#write` takes a single argument, writes it to the stream, then returns the number of bytes written.

```
array = [3, 2, 1]
print array
# Writes "[3, 2, 1]" to $stdout
puts array
# Writes "3\n2\n1\n" to $stdout
$stderr.puts 2, 4, 6
# Writes "2\n4\n6\n" to $stderr
proverb = "Ab igne ignem capere"
sez = "--Cicero"
print proverb, sez
# Writes "Ab igne ignem capere--Cicero" to $stdout
puts proverb, sez
# Writes "Ab igne ignem capere\n--Cicero\n" to $stdout
```

`Kernel#p` provides output suitable for debugging. It accepts any number of arguments, which it converts with `#inspect`, then writes each of them to `$stdout` separated by `"\n"`.

```
killed_by = {
  'Augustus' => nil,
  'Tiberius' => nil ,
  'Caligula' => {who: 'Praetorian Guard'},
  'Claudius' => {who: 'wife'},
  'Nero'     => {who: 'himself'}
}
killed_by['Nero'][:who] = killed_by['Nero']
p killed_by.select{|*, killer| killer}
# Writes to $stdout:
# {"Caligula"=>{:who=>"Praetorian Guard"}, "Claudius"=>{:who=>"wife"}, "Nero"=>{:who=>{...}}
```

Reading

The simplest way to read from a file is by supplying its name to `IO.read`. If the file contains binary data, `IO.binread` should be used instead. Likewise, `IO#read` and `IO#binread` read from their receiver, instead.

If parsing a binary file format, for example, you might treat a file as a stream of bytes. To write a spell checker, you may think in terms of characters. And to analyse a log file you'd think in terms of lines. Its still the same stream, however we think of it, but working in the most appropriate *unit* leads to clearer code. The key methods for reading from a stream are summarised below. The methods in the *Fetch Next* column return the unit at the current file position, which they then advance. If called at the end of a stream, they return `nil`. By contrast, the methods listed in *Fetch All Remaining* return all the units from the current position through to the end of a stream. The *Enumerate* methods return an `Enumerator`, or if given a block, yield each unit of the stream in turn.

Unit	Fetch Next	Fetch All Remaining	Enumerate
Bytes	`IO#getbyte`	N/A	`IO#bytes`
Characters	`IO#getc`	`IO#read`	`IO#chars`
Codepoints	`N/A`	`N/A`	`IO#codepoints`
Lines	`IO#gets`	`IO#readlines`	`IO#lines`

The methods described above are all instance methods, so before use they require that an `IO` object is created. When the only reason for instantiating this object is to enumerate it, a more elegant approach is `IO.foreach(`*fn*`)`, which yields each line of the file named *fn* to a block, or returns an `Enumerator` if the block is omitted.

Access Mode

A file is opened with a particular *access mode* which specifies the type of action that may be performed on it. It is given as either a `String` or, less commonly, a bitmask, as shown in the table below. The default access mode is `r`.

In Access mode specifiers, the access mode is presented as both a *String* and the equivalent *Bitmask*. The *Read?* and *Write?* columns indicate whether the opened file can be read from or written to, respectively. *Truncate?* specifies that existing files will be truncated before use. *Creates?* specifies that non-existent files will be created before use. Finally, *Start Position* is the position in the file reading/writing starts from: *Beginning* is before the first byte; *End* is after the last.

String	Bitmask	Read?	Write?	Truncates?	Creates?	Start Position
`r`	`File::RDONLY`	✓	✖	✖	✖	Beginning
`r+`	`File::RDWR`	✓	✓	✖	✖	〃
`w`	`File::WRONLY | File::TRUNC | File::CREAT`	✖	✓	✓	✓	〃
`w+`	`File::RDWR | File::TRUNC | File::CREAT`	✓	✓	✓	✓	〃
`a`	`File::APPEND | File::WRONLY | File::CREAT`	✖	✓	✖	✓	End.
`a+`	`File::APPEND | File::RDWR | File::CREAT`	✓	✓	✖	✓	〃

Access mode specifiers

Binary & Text Mode

An `IO` stream may be configured to use binary mode or text mode. These mutually exclusive options determine what automatic modifications, if any, Ruby will make to data read from, and written to, the stream. They have no relationship to the access mode.

Binary mode is disabled by default. It must be enabled when reading a file with an ASCII-incompatible external encoding. When enabled it has the following effects:

- Unless an external encoding has been specified explicitly, it is set to ASCII-8BIT.
- Newline conversion is disabled.
- Transcoding is disabled unless both an internal and external encoding have been specified.

Binary mode may be enabled when opening a stream by either including the `b` modifier in the mode string, or supplying `binmode: true` for the options `Hash`. It may be enabled for an existing stream with `IO#binmode`, and queried with the `IO#binmode?` predicate.

Text mode defaults to on under Microsoft Windows, and off everywhere else. Reading a file in text mode causes `"\r\n"` to be replaced by `"\n"`, and other occurrences of `"\r"` to be replaced by `"\n"`. Writing a file in text mode causes `"\n"` to be replaced with `"\r\n"` under Windows; having no effect on other platforms.

Opening

`IO.sysopen` *opens* a given pathname and returns a corresponding file descriptor. (Use the File class for a higher-level interface to this operation). Optionally, it accepts an access mode as the second argument, and permission bits as the third.

If a file is being created, its initial permissions may be specified as an `Integer`; otherwise, this argument is ignored. On Unix-like systems, these permissions bits are interpreted in the same fashion that `chmod(1)` interprets octal mode arguments. For example, permissions of `0400` gives the user read permission, and no permissions to anybody else. See Permissions for further details.

Encoding String

`IO` methods that expect encoding names as arguments, often accept *encoding string*s, which allow one or both of the external encoding and internal encoding to be specified at once in one of the forms below.

Both *external* and *internal* are names of encodings. The *Inferred from BOM* column indicates that the external encoding is set to that specified by a BOM, if present, otherwise to the named encoding.

Form	External Encoding Value	Internal Encoding Inferred from BOM?	Value
external	*external*	✖	`Encoding.default_internal`
external:-			
BOM|*external*	*external*	✓	`Encoding.default_internal`
BOM|*external*:-			
external:*internal*	*external*	✖	*internal*
BOM|*external*:*internal*	*external*	✓	*internal*
:*internal*	Default external	✖	*internal*

The forms the value of the encoding string may take

The `BOM|` prefix deserves a fuller explanation. The UTF-8, UTF-16BE, UTF-16LE, UTF-32BE, and UTF-32LE encodings support the presence of a byte-order mark: a special character occuring at the start of the stream which indicates the byte order of the encoding. This can be used to distinguish between the big-endian and little-endian forms of UTF-16, for example. Prefixing one of the aforementioned encoding names with the case-insensitive string `BOM|` causes the named encoding to be used if the stream doesn't contain a BoM; otherwise, the BoM is stripped from the stream and the encoding that it specifies is used instead.

Initializing

An `IO` instance objectifies a given file descriptor. It may be initialized by supplying this file descriptor as the first argument to `IO.new`. Optionally, a mode string, or numeric access mode, can be provided as the second argument, however this must agree with the access mode already associated

with the file descriptor. An options Hash may be provided as the final argument.

`IO.open` accepts the same arguments, but also expects a block. It initializes an `IO` object as before, yields it to the block, then ensures the stream is closed when the block exits.

Mode String

The *mode string* is a concise way to specify options for opening a file. At its simplest, it consists of only the access mode as a `String`, e.g. a mode string of `"r"` opens a file in read-only mode. If the next character is `b`, it specifies binary mode; if it is `t` , it specifies text mode. Finally, the external and/or internal encodings may be specified as an encoding string.

In the table below, *mode* denotes one of the access modes given in the Access Mode table. The *Binary?* and *Text?* columns indicates whether the stream is in binary text mode, respectively. Both *internal* and *external* are names of encodings.

Form	Binary?	Text?	External Encoding	Internal Enco
mode	✖	✖	`Encoding.default_external`	`Encoding.default_`
mode`t`	✖	✓	〃	〃
mode`b`	✓	✖	`ASCII-8BIT`	〃
mode`:`*external*	✖	✖	*external*	〃
mode`t:`*external*	✖	✓	〃	〃
mode`b:`*external*	✓	✖	〃	〃
mode`:`*external*`:`*internal*	✖	✖	〃	*internal*
mode`t:`*external*`:`*internal*	✖	✓	〃	〃
mode`b:`*external*`:`*internal*	✓	✖	〃	〃

The forms a mode string may take

For example, `r:ascii` reads from the beginning of a file, tagging the data it reads as US-ASCII; `a+:ascii:utf-8` opens the file for reading and appending, transcoding from US-ASCII to UTF-8 when reading, and in the opposite direction when writing.

Options Hash

Alternatively, many methods that open files accept an options `Hash` as their final argument. Class methods of `IO` that accept this `Hash` recognise an `:open_args` key whose value is an `Array` of arguments for the underlying `open` method; if this key is supplied, all others are ignored. In addition to the keys listed below, those of the `String#encode` options `Hash` are also recognised.

Key	Value	Default value	Description
`:mode`	The mode string or the access mode as an `Integer`.	`r`	Whether to open the file in read-only, read-write, or write-only mode, and whether to truncate existing files or append to them.
`:textmode`	`true` or `false`	`false`	Whether to perform newline conversion.
`:binmode`	`true` or `false`	`false`	Whether to treat the file as a stream of bytes.
`:open_args`	`Array`	`[]`	Arguments for `Kernel.open`.
`:perm`	`Integer`	Platform-specific	The permissions the file

Options that may be specified as a `Hash` for certain methods that open files

Key	Value	Default value	Description
			should be created with.
`:external_encoding`	`Encoding` object or encoding name	`Encoding.default_external`	The external encoding to apply to the stream.
`:internal_encoding`	〃	`Encoding.default_internal`	The internal encoding to apply to the stream.
`:encoding`	Encoding name or two names separated by a colon.	*`default_external`*`:`*`default_internal`*	The external, or external and internal, encodings to apply to the stream.
`:autoclose`	`true` or `false`	`true`	Whether to automatically close the file descriptor when the `IO` object is finalised.

Open Flags

The access mode may also be given as an `Integer` formed by taking the bitwise OR of one or more *open flags*, each of which is represented by a constant in the `File` namespace. The available flags are explained below.

`File::APPEND`
: Opens the file in *append mode*. "If set, then all write operations write the data at the end of the file, extending it, regardless of the current file position. This is the only reliable way to append to a file. In append mode, you are guaranteed that the data you write will always go to the

current end of the file, regardless of other processes writing to the file. Conversely, if you simply set the file position to the end of file and write, then another process can extend the file after you set the file position but before you write, resulting in your data appearing someplace before the real end of file." [Loosemore07, pp. 335–335] .

`File::BINARY`
: Puts the stream in binary mode.

`File::CREAT`
: Creates the file if it doesn't already exist. If the file already exists, this has no effect unless `File::EXCL` is also given.

`File::DSYNC`
: "... specifies that only normal data be synchronized after each write operation, not metadata." [Love07, pp. 40–40] . On Linux, this is synonymous with `File::SYNC`.

`File::EXCL`
: When given with `File::CREAT`, an `Errno::EXIST` exception will be raised if the file already exists. "This is used to prevent race conditions on file creation." [Love07, pp. 24–26] .

`File:NOATIME`
: Do not update the access time (*atime*) of the file. "This is used by programs that do backups, so that backing a file up does not count as reading it. Only the owner of the file or the superuser may use this bit." [Loosemore07, pp. 335–335] .

`File:NOCTTY`
: "If the named file is a terminal device, don't make it the controlling terminal for the process." [Loosemore07, pp. 333–334] . (`IO#tty?`, or its alias `IO#isatty`, can be used to determine whether the stream is a TTY).

`File::NOFOLLOW`
: If the pathname is a symbolic link, an `Errno::ELOOP` exception will be raised. "Normally, the link is resolved, and the target file is opened. If other components in the given path are links, the call will still succeed. For example, if name is `/etc/ship/plank.txt`, the call will fail if

`plank.txt` is a symbolic link. It will succeed, however, if `etc` or `ship` is a symbolic link, so long as `plank.txt` is not." [Love07, pp. 24–26] .

`File:NONBLOCK`
: "If possible, the file will be opened in nonblocking mode. Neither the `open()` call, nor any other operation will cause the process to block (sleep) on the I/O. This behavior may be defined only for FIFOs." [Love07, pp. 24–26] .

`File:RDONLY`
: Opens the file for reading.

`File:RDWR`
: Opens the file for reading and writing.

`File::RSYNC`
: "...specifies the synchronization of read requests as well as write requests. It must be used with one of [`File::SYNC`] or [`File::DSYNC`]. As mentioned earlier, reads are already synchronized—they do not return until they have something to give the user, after all. The [`File::RSYNC`] flag stipulates that any side effects of a read operation be synchronized, too. This means that metadata updates resulting from a read must be written to disk before the call returns." [Love07, pp. 40–40] . On Linux, this is synonymous with `File::SYNC`.

`File:WRONLY`
: Opens the file for writing.

`File::SYNC`
: "The file will be opened for synchronous I/O. No write operation will complete until the data has been physically written to disk; normal read operations are already synchronous, so this flag has no effect on reads." [Love07, pp. 24–26] .

`File::TRUNC`
: "If the file exists, it is a regular file, and the given flags allow for writing, the file will be truncated to zero length. Use of [`File::TRUNC`] on a FIFO or terminal device is ignored. Use on other file types is undefined.

> Specifying [`File::TRUNC`] with [`File::RDONLY`] is also undefined, as you need write access to the file in order to truncate it." [Love07, pp. 24–26] .

Buffering

"Buffering writes provides a *huge* performance improvement, and consequently, any operating system even halfway deserving the mark "modern" implements delayed writes via buffers." [Love07, pp. 37–37] . Love explains the Linux kernel's approach-which is similar to that of other operating systems-to write buffering as follows:

> …when a user-space application issues a `write()` system call, the Linux kernel performs a few checks, and then simply copies the data into a buffer. Later, in the background, the kernel gathers up all of the "dirty" buffers, sorts them optimally, and writes them out to disk (a process known as *writeback*). This allows write calls to occur lightning fast, returning almost immediately. It also allows the kernel to defer writes to more idle periods, and batch many writes together.
>
> —Love07, pp. 37–37

Further, and for the same reasons, Ruby maintains her own I/O buffer. Therefore, before data is written to disk by Ruby, it passes first through her buffers, and then those of the underlying operating system. The latter are beyond the scope of this text, so we shall focus on the former from now on.

The `IO` methods for writing that we discussed above, are buffered-that is, they pass through Ruby's buffers. An exception is `IO#syswrite`, so shouldn't be used in conjunction with other methods that perform writing. To understand why, consider a scenario where a given stream has `?a` written to it with `IO#print`, then `?b` written to it with `IO#syswrite`, then `?c` written to it with `IO#print`. The first character written is buffered, the second is written directly, and the third is buffered. When the stream is closed, Ruby flushes her buffer, causing `"ac"` to be written. Therefore, the data is written out of order: `"bac"`.

Ruby's buffer can be flushed manually with `IO#flush`. This causes the contents of the buffer to be passed to the operating system, and the buffer to

be emptied. Alternatively, buffering can be disabled for a given stream by setting `IO#sync=` to `true`; by default, it is `false`. This value can be queried with `IO#sync`.

The operating system's buffer can be flushed with `IO#fsync`, which ensures both the data and metadata—e.g. file creation timestamps—associated with the stream are written to disk. A faster alternative is `IO#fdatasync` because it only flushes the data. Both methods wrap the eponymous system call, so if that is not available an `NotImplementedError` is raised.

Reads are buffered by Ruby, too, by reading more data than the user requests, and buffering the surplus. Accordingly, when a stream is read from then duplicated or reopened, its replica may return data out of order or report negative values for `IO#pos`. See [Ruby-core-28281] for more details.

Buffered reads make possible what Loosemore et al. term *unreading*: [Loosemore07, pp. 241–241]

> Using stream I/O, you can peek ahead at input by first reading it and then *unreading* it (also called *pushing it back* on the stream). Unreading a character makes it available to be input again from the stream, by the next call to [an] input function on that stream.
>
> —Loosemore07, pp. 241–241

Two methods are provided for this purpose: `IO#ungetc` and `IO#ungetbyte`. Both expect an argument-characters for the former; bytes for the latter-which they unread. Subsequent buffered reads from the stream will return the unread characters/bytes in reverse chronological order before reading new data from the stream.

Closing

Once an `IO` stream is finished with, it should be *closed*. This ensures that Ruby's write buffer is flushed, and the associated file-descriptor is released back to the operating system. A stream is closed for reading and writing with `IO#close`. Subsequent attempts to read from or write to a closed stream causes an `IOError` to be raised, so the `IO#closed?` predicate is available for

testing a stream. A duplex stream may also be closed just for writing or just for reading, using `IO#close_write` and `IO#close_read`, respectively.

```
$stdin.closed? #=> false
$stdin.close #=> nil
$stdin.gets #=> IOError
pipe = IO.popen(ENV['SHELL'],'r+')
pipe.close_write #=> nil
pipe.closed? #=> false
pipe.close_read #=> nil
pipe.closed? #=> true
```

When an `IO` object is finalised its *auto-close* flag determines whether the underlying file descriptor is closed automatically. By default, this flag has the value `true`, but it can be set explicitly with `IO#autoclose=`, and queried with `IO#autoclose?`.

On a Unix-based system a process created by `Kernel.exec`, `Kernel.fork`, or `IO.popen` inherits the file descriptors of its parent. Depending on the application, this may constitute an information leak in that the child is able to access data that he shouldn't have access to. If given a true argument, `IO#close_on_exec=` ensures that its receiver is closed before the new process is created; otherwise, it does not. `IO#close_on_exec?` returns the status of this flag as either `true` or `false`. On systems that don't support this feature, these methods raise `NotImplementedError`.

```
open('/tmp/f','w'){|f| f << 'sekrit'}
f = open('/tmp/f')
f.close_on_exec? #=> false
exec("ruby -e 'p IO.open(#{f.fileno}).read'")
# Prints "sekrit"

open('/tmp/f','w'){|f| f << 'sekrit'}
f = open('/tmp/f')
f.close_on_exec = true
f.close_on_exec? #=> true
exec("ruby -e 'p IO.open(#{f.fileno}).read'")
# Prints:
#-e:1:in `initialize': Bad file descriptor (Errno::EBADF)
#        from -e:1:in `open'
#        from -e:1:in `<main>'
```

Positions & Seeking

Beck et al. explain that an attribute of a file's structure is "the position of the read/ write pointer at which the next I/O operation will be carried out. This value is updated by every I/O operation and by the system calls `lseek` and `llseek`." ([Beck98, pp. 41–43] . Ruby exposes the current position in an I/O stream with `IO#pos` as the offset in bytes. To seek to a given position in a stream, one can assign the new, `Integer`, offset to `IO#pos=`. For more control over seeking, use `IO#seek(`*`pos`*`,` *`origin`*`)`, where *pos* is a byte offset specified as an `Integer`, and *origin* is one of the values given below.

`IO::SEEK_CUR`
: The current position is set to its current value plus *pos*, which can be negative, zero, or positive. If *pos* is zero the current position is unchanged.

`IO::SEEK_END`
: The current position is set to the length of the file plus *pos*, which can be negative, zero, or positive. If *pos* is zero the current position is set to the end of the file.

`IO::SEEK_SET`
: The current position is set to *pos*. If *pos* is zero the current position is set to the beginning of the file.

Ruby also keeps track of the current line number in a stream. The line-orientated reading methods increment the line number when they encounter the separator character. It initially has a value of zero, and its current value may be retrieved with `IO#lineno`, or set explicitly with `IO#lineno=`. The current position and line number for a given stream may be reset to zero with `IO#rewind`.

As described in the introduction to this chapter, a position past the end of a stream is termed *End of File*, or EoF. Depending on the method, reading from a stream at EoF causes either an `EoFError` exception to be raised or `nil` to be returned. If a stream is open for reading, this condition can be tested for with the `IO#eof?` predicate.

Pipes

A *pipe* is a unidirectional[3], or *half-duplex*, inter-process communication channel. It comprises two file descriptors: one of which is open for reading, and the other, writing. Data written to the write end of a pipe can be read from the read end on a first-in-first-out basis.

A pipe is created in Ruby using `IO.pipe`. It returns a two-element `Array` of `IO` objects, the first of which is the read end, and the second the write end. If `IO.pipe` is given a block, it is passed both `IO` objects as arguments, and ensures they are closed when the block exits. The write end of the pipe has its sync mode set automatically such that Ruby does not buffer writes; the read end will block until the write end has been closed for writing. However, as W. Richard Stevens notes "A pipe in a single process is next to useless." [Stevens05, pp. 428–429] ; the *Processes & Signals* chapter discusses how pipes are used to communicate between processes.

```
r, w = IO.pipe
w << 'secret'
w << ' message'
w.close_write
r.read #=> "secret message"
```

Asynchronous & Multiplexed

By default, I/O operations are *blocking* (*synchronous*): they do not return control to their caller until finished. This can be problematic given the performance disparity between CPUs and devices such as hard disks and routers. An I/O-bound program may spend the majority of its time idly waiting. Conversely, *non-blocking* (*asynchronous*) operations return immediately. They try to perform the operation as normal, but if doing so would require blocking, they signal an error and return.

---

3. Bi-directional pipes also exit in some systems. POSIX only requires unidirectional semantics, however, so it is there that we shall focus.

We have already seen a partial solution to this problem in the `File::NONBLOCK` flag taken by `Kernel.open`. `IO#read_nonblock(`*max*`)` builds upon this approach by setting the `NONBLOCK` flag for the file descriptor, then attempting to read at most *max* bytes without blocking. If an optional second argument is supplied, it is a `String` to which the received data is appended. This method works as follows:

1. If there is data in Ruby's read buffer, up to *max* bytes are returned.
2. If the stream can be read from without blocking, at most *max* bytes are read and returned.
3. If `IO#read` would have raised an exception when attempting to read from this stream, the same exception is raised.
4. Otherwise, either `Errno::EWOULDBLOCK` or `Errno::EAGAIN` is raised to indicate that the stream can not be read without blocking. These two exceptions are virtually synonymous[4], so Ruby mixes in `IO::WaitReadable` to both, allowing `rescue IO::WaitReadable` to handle either.
5. If EoF is reached, `EOFError` is raised.

The principle, then, is that if `#read_nonblock` raises `Errno::EWOULDBLOCK` or `Errno::EAGAIN`, the program can attend to other tasks before it retries the call. `IO#readpartial` is identical, except it does not set the `NONBLOCK` flag on the file descriptor: if blocking is unavoidable, it blocks.

`IO#write_nonblock(`*string*`)` is similar to `#read_nonblock`. It flushes the write buffer, then tries to write *string* to the receiver, raising `Errno::EAGAIN` or `Errno::EWOULDBLOCK` if the write would have blocked. It returns the number of bytes written, which should be compared to *string*`.bytesize` to detect partial writes. On Windows, some `IO` objects cannot be used in this way, so `#write_nonblock` raises `Errno::EBADF`.

Given a collection of `IO` objects, how does a program know which are available for writing or reading? `Kernel.select` is one answer. It accepts up to three `Arrays` of `IO` objects: those in the first `Array` are monitored for reading, the second, for writing, and the third, for errors. It returns a three-element `Array` of the same configuration containing only the `IO` objects that are ready for their respective operation. If a fourth argument is given, it is a

4. POSIX allows both to have the same value.

timeout: if none of the streams are ready in the given amount of seconds `Kernel.select` returns `nil`.

```
pipes = 5.times.map{ IO.pipe }
readers, writers = pipes.map(&:first), pipes.map(&:last)
Thread.new do
  writers.shuffle!
  while w = writers.pop
    sleep(w.to_i / writers.size) and w << "#{w.to_i}"
    w.close
  end
end.run
until readers.empty? do
  ready = select(readers, writers, [], 1) or abort "Got bored"
  ready.first.each do |box|
    begin
      print "#{box.readline} (wait: #{ready[1].size}) "
    rescue EOFError
      readers.delete(box)
    end
  end
end
# 12 (wait: 4) 10 (wait: 3) 4 (wait: 2) 8 (wait: 1) Got bored
```

Manipulating File Descriptors

`IO#fcntl(`*cmd*, *arg*`)` performs *cmd* on the receiver, where *cmd* is an platform-specific `Integer`. The `Fcntl` module in the standard library provides constants corresponding to some of the more common commands. If *cmd* requires an argument, it should be supplied as *arg*.

Command	Argument	Returns	Description
`F_DUPFD`	`Integer`	Positive `Integer`; `-1` on error	Find the lowest numbered available file descriptor greater than or equal to *arg* and make it a copy of the receiver's file descriptor. If *arg* is omitted, it is

`IO#fcntl` commands with corresponding constants defined in the `Fcntl` module

Command	Argument	Returns	Description
			assumed to be equal to the receiver's file descriptor.
F_GETFD	N/A	File descriptor flags	Retrieves the associated file descriptor flags. Currently, these are either 0 or FD_CLOEXEC. These flags may be set with F_SETFD.
F_GETFL	N/A	Integer	Returns the file status flags, i.e. a bitwise OR of O_APPEND, O_ASYNC, O_DIRECT, etc. O_ACCMODE is a bitmask for extracting the access mode from these flags.
F_GETLK	struct flock *	N/A	The argument describes a lock the caller wishes to place on the file. If this is possible, the l_type field of the struct is set to Fcntl::F_UNLCK; otherwise the struct is updated with details of the current lock holder.
F_SETFD	FD_CLOEXEC or 0	0; -1 on error	Sets the file descriptor flags to *arg*.When *arg* is FD_CLOEXEC, this is equivalent to #close_on_exec=true.
F_SETFL	Integer	0; -1 on error	Set the file status flags to *arg*
F_SETLK	struct flock *	0; -1 on error.	When the struct's l_type field has the value F_RDLCK or F_WRLCK, acquires the lock; when it has the value F_UNLCK, releases the lock.
F_SETLKW	struct flock *	0; -1 on error.	Behaves like F_SETLK, except when a conflicting lock is held this call blocks until the lock is released or a signal is caught.

```
require 'fcntl'
$stdout.fcntl(Fcntl::F_DUPFD) #=> 3
f = open('/tmp/file', ?w)
f.fcntl(Fcntl::F_SETFD, Fcntl::FD_CLOEXEC)
f.close_on_exec? #=> true
access_mode = f.fcntl(Fcntl::F_GETFL) & Fcntl::O_ACCMODE
access_mode == Fcntl::O_WRONLY #=> true
```

```
f.fcntl(Fcntl::F_SETFL, access_mode | Fcntl::O_APPEND)
f.fcntl(Fcntl::F_GETFL) & Fcntl::O_APPEND == Fcntl::O_APPEND #=> true
```

Unfortunately, the locking commands are awkward to use from Ruby because they require *arg* to be a binary `String` representing a specific C struct… If you insist, consult `man 2 fcntl` and `String#unpack`.

In a similar vein, `IO#ioctl(`*cmd*`, `*arg*`)` is an esoteric way to send commands to hardware devices. *cmd* is the `Integer` associated with the command, and *arg* is its optional argument. This is utterly unportable because commands and their values are typically specific to particular operating systems, drivers, and hardware. For example, on Linux `man 2 ioctl_list` provides a partial list of values for *cmd*. One of them is `CDROMEJECT`, which on my system has the value `0x00005309` and doesn't accept arguments. When sent to a CD-ROM drive, it causes the disc to eject. When *arg* is an integer it can be supplied as an `Integer`, otherwise `String#unpack` will be required to craft a `String` resembling the expected data structure. The example that follows animates the LEDs corresponding to my `Num Lock`, `Caps Lock`, and `Scroll Lock` keys. To have any chance of running it on another system, you will at least need to adjust the value of `KDSETLED`.

```
# Value of KDSETLED; see `man 2 ioctl_list` or `grep` /usr/include for
# the corresponding value on your system
KDSETLED = 0x4B32
def flash n
  @tty ||= open("/dev/console") # You'll probably need to be root
  @tty.ioctl(KDSETLED, n)
  sleep 0.1
end

at_exit { flash 0 }

loop do
  (0..6).each do |n|
    [n, ~n, n].each{|m| flash m }
  end
end
```

ARGV

The command-line arguments given to a program are available as elements of the `ARGV` `Array`. If `ARGV` is empty, no arguments were provided. Therefore, unlike C, `ARGV[0]` does not hold the program name, which is available as `$0` instead. This `Array` can be modified, so programs may remove an element after they have processed it.

```
#!/usr/bin/env ruby
# argv.rb
puts ARGV.size
puts ARGV.first
puts "#$0: #{ARGV}"

run@paint ➔  ruby -w argv.rb -n 3 "bags full"
3
-n
argv.rb: ["-n", "3", "bags full"]
run@paint ➔  ln -s argv.rb argv
run@paint ➔  chmod +x argv
run@paint ➔  ./argv
0
nil
"./argv: []"
run@paint ➔  ./argv -n 3 "bags full"
3
-n
./argv: ["-n", "3", "bags full"]
```

ARGF

`ARGF` is an `IO`-like stream abstracting the contents of a program's file arguments, representing the concatenation of their contents. An assumption, therefore, is that all arguments in `ARGV` are filenames; the remainder should have been removed.

Each file is read from in the order they were specified. The filename of the file currently being read is available as `ARGF.filename`, while `$.` holds the

number of the last line read. `ARGF.skip` advances to the next file, and is a no-op if no more files remain.

In-place mode allows the files in `ARGF` to be modified in-place, possibly after backing them up. It sets `$stdout` to the file being read from `ARGF`, so `Kernel` methods such as `puts` write their output to this file. It is enabled by either giving the interpreter an `-i` *`extension`* switch, or setting `ARGF.inplace_mode=` to *extension*. When the argument to `-i` is omitted, or *extension* is the empty `String`, no backups are made. Otherwise, before each file is modified it is copied to a filename formed by appending *extension* to it.

If `ARGV` is empty, `ARGF` refers to `$stdin` instead. The current filename is set to `-`.

```
#!/usr/bin/env ruby
# argf.rb
p ARGV.to_a
p ARGF.read

run@paint ➜   for l in "a" "b" "c"; do echo $l > $l; done
run@paint ➜   ruby argf.rb  # Hangs; blocking on STDIN
run@paint ➜   ruby argf.rb < a
[]
"a\n"
run@paint ➜   ruby argf.rb a b < c
["a", "b"]
"a\nb\n"
run@paint ➜   ruby argf.rb --help
["--help"]
/tmp/argf.rb:4:in `read': No such file or directory - --help (Errno::ENOENT)
  from /tmp/argf.rb:4:in `<main>'
```

PROCESSES

A *process* is an instance of a computer program. We use the term *current process* to refer to an instance of Ruby running your program. The semantics of processes are partly platform-specific. Ruby abstracts these differences where possible, but for the best experience use a UNIX-based system such as *GNU/Linux* or *Mac OS*.

Executing & Forking

A process is often created by *executing* a program: loading a specific binary program into memory. The program is identified by a filename which is either absolute or relative to a directory in the user's path[1] . Broadly, Ruby provides four approaches for executing programs, which are summarised in the table below. Alternatively, `Kernel.fork` creates a new process by duplicating the current process. The remainder of this section explains these methods in detail.

Method	Returns	Blocks Until Process Exits?
`Kernel.\``	The process's output	Yes
`Kernel.exec`	Nothing on success (replaces the current process with the new process)	N/A
`Kernel.system`	Whether the process executed successfully	Yes
`Kernel.spawn`	The process's PID	No

Backticks

A double-quoted string delimited with grave accents (`U+0060`) characters, or "backticks", executes its contents as an operating system command and returns the output. It runs the command via a shell, so wildcard expansion

1. That is, his *search path*: a list of directories which contain program files. On UNIX-based systems, this is the value of the environment variable `$PATH`.

and similar features are available. This is implemented with the `Kernel.`` method, which can be redefined to alter these semantics. Alternative delimiters can be used with the corresponding `%x`*`delimiter`*`…`*`delimiter`* construct, which follows the same rules as %Q.

```
`date` #=> "Tue Feb 16 04:01:10 GMT 2010\n"
file = '/etc/fstab'
%x<ls -all #{file}>.chomp
#=> "-rw-r--r-- 1 root root 753 2009-11-19 13:36 /etc/fstab"
```

Kernel.exec

`Kernel.exec` replaces the current process image with a new process image. Accordingly, it will not return if successful; if it fails, `SystemCallError` is raised. If a single argument is provided, it is a `String` containing a command line that should be executed by the shell—`/bin/sh` on Unix-like systems; the value of the `RUBYSHELL` or `COMPSEC` environment variables otherwise—so is subject to shell expansion.

```
exec("no-such-command")
#=> Errno::ENOENT
exec("ls /usr/share/dict/* | wc")
#     5       5     172
```

If multiple `String` arguments are given, the first is the name of a command in the user's path, and the remainder are the command's arguments. The command may be either a binary, or an executable script with a shebang. It is executed by a system call from the `exec(3)` family[2], so neither the command or arguments are subject to shell expansion. If the command name is given as an `Array` of the form [*`name`*, *`argv0`*], *name* is the command's name, and *argv0* is the filename associated with *name*[3].

---

2. A consequence is that open file descriptors are passed to the new process. To avoid this, use `IO#close_on_exec=` or the `:close_others` key in the options `Hash`
3. Changing *argv0* is useful because some programs, such as `ps(1)` and `top(1)`, will use it in place of the command name, while others, such as `csh(1)`, treat it specially.

```
exec("uname", "-s")
# Linux
```

In either of these forms, the new process's environment may be modified by providing a `Hash` of environment variables before the first argument. An environment variable is created for each `String` key, or unset if the corresponding value is `nil`. An options `Hash` may be supplied for the final argument.

```
script = "/tmp/script.rb"
open(script, ?w){|f| f << "#!/usr/local/bin/ruby\np ENV['GLARK']"}
File.chmod 0755, script
exec({'GLARK' => 'always'}, script)
# Prints "always"
```

Kernel.system

`Kernel.system` interprets its arguments in the same way as `exec`, but executes the command in a subshell then returns. Its return value is `true` if the command executed successfully, `false` if the command's exit status was non-zero, or `nil` if the command failed to execute.

```
system('git init') #=> true
system(*%w{gpg -r runrun@runpaint.org --encrypt file}) #=> true
system("cat glark") #=> false
# cat: glark: No such file or directory
system("xzxzxzxz") #=> nil
```

Kernel.spawn

`Kernel.spawn`, and its alias `Process.spawn`, also interpret their arguments in the same way as `exec`, but execute the command in a subshell then return without waiting for the command to complete. Their return value is an `Integer` holding the PID of the new process.

```
pid = spawn('cat /dev/urandom > /dev/null') #=> 24206
`ps --no-headers #{pid}`
#=> "24206 pts/1    S+     0:00 sh -c cat /dev/urandom > /dev/null\n"
system("kill #{pid}") #=> true
```

Kernel.fork

On UNIX-based systems[4], a new process may also be created by duplicating[5]—or *forking*—the current process. If this is successful, both processes continue to run as normal.

Ruby implements forking with `Kernel.fork`, and its alias `Process.fork`. These methods raise a `NotImplementedError` on platforms such as Microsoft Windows that don't implement the `fork(2)` system call. If given a block, that block is run in the subprocess, then the subprocess terminates with an exit status of `0`.

```
puts "Parent: #{Process.ppid} -> #{Process.pid}"
fork do
  puts "Child: #{Process.ppid} -> #{Process.pid}"
end

# Prints:
# Parent: 6653 -> 16743
# Child: 16743 -> 16745
```

If the block is omitted, `fork` returns the PID of the child to the current process, and `nil` in the child process.

```
print "\nIn #{(pid = fork) ? "parent (#{pid})" : "child (#{Process.pid})"}: "
2.times{|i| print "#{i} "}
# Prints:
#
# In parent (24225): 0 1
# In child (24225): 0 1
```

---

4. Typically, forking is how UNIX executes all programs: first it forks, then in the new process it uses the `execve(2)` system call—provided by Ruby as `Kernel.exec`—to replace this process with the result of executing the program.
5. The new process is not *identical* to the current process—an obvious difference is that the new process has a different PID—but the differences are minor; for details, consult your system manuals for `fork(2)`.

In both examples above, the child process may become a *zombie*; see Status for how to avoid this.

IO.popen

`IO.popen(`*cmd*`, `*mode*`='r')` executes a command, *cmd*, as a subprocess, opening a pipe to this subprocess's standard input and output streams, which it returns as an `IO` object. The default access mode of the pipe is `"r"`, but this may be overridden with the *mode* argument.

If *cmd* is a `String` it names a command in the user's path, and is subject to shell expansion. If it is a `"-"`, and the platform supports forking, the current process forks: an `IO` pipe connected to the child's standard input and output streams is returned to the parent; `nil` is returned to the child.

Otherwise, *cmd* is an `Array` of `Strings`, the first of which specifies the command name; the remainder, its arguments. The shell is bypassed, so none of these `Strings` are subject to shell expansion. If the first element of this `Array` is a `Hash`, it specifies the names and corresponding values of environment variables that should be set in the subprocess. An options `Hash` may be supplied as the last element of this `Array`.

If a block is supplied, Ruby's end of the pipe is passed to it as a parameter, then closed when the block exits. `$?` is set to the exit status of the subprocess, and the value of the block is returned.

When a block is supplied along with a *cmd* of `"-"`, Ruby forks, running the block in both processes. In the parent process the block is passed an `IO` pipe connected to the child's standard input and output streams; in the child process the block is passed `nil`.

`Kernel.open("|`*cmd*`", `*mode*`='r')` behaves like `IO.popen(`*cmd*`, `*mode*`='r')`, when *cmd* is a `String`. Likewise, `Kernel.open("|-", `*mode*`='r')` behaves like `IO.popen("-", `*mode*`='r')`

In all of these cases, the process ID of the subprocess may be retrieved with `IO#pid`.

Options Hash

`exec`, `system`, `spawn`, and `IO.popen` all accept an options `Hash` as their last argument. It may contain any of the following keys.

Key	Default Value	Description
`:unsetenv_others`	`false`	If `true` clears the environment variables not named in the *env* `Hash`
`:pgroup`	`nil`	If `true` or `0`, make a new process group; if an `Integer` join the process group with that ID; if `nil`, don't change the process group.
`:rlimit_`*resource*	`Process.getrlimit(`*resource*`)`	Where *resource* is a resource name recognised by `Process.setrlimit`/`Process.getrlimit`, sets that resource to the given value. If the value is an `Array`, its first element is the new soft limit, and its second is the new hard limit.
`:chdir`	`Dir.pwd`	The value is a `String` naming the directory to change to before invoking the command.
`:umask`	`File.umask`	The value is an `Integer` specifying the new value of the process's file creation mask, or *umask*.
`:in`	`STDIN`	Redirects the standard input stream to the given stream.
`:out`	`STDOUT`	Redirects the standard output stream to the given stream.
`:err`	`STDERR`	Redirects the standard error stream to the given stream.
`Integer`	N/A	When the key is an `Integer`, it is interpreted as a file descriptor to redirect to the given stream.
`IO`	N/A	When the key is an `IO` object, its file descriptor is redirected to the given stream.

Key	Default Value	Description
`Array`	N/A	Each element is a file descriptor specified in any of the formats listed above. They are all redirected to the given stream.
`:close_others`	`true` for `system` and `exec`; `false` otherwise.	If `true`, the process does not inherit its parent's file descriptors; otherwise it does.

The options that redirect an I/O stream may be given a value in any of the following formats.

`:in`
: Standard input

`:out`
: Standard output

`:err`
: Standard error

`String`
: File descriptor of `open(`*`string`*`, ?w)`

`[String]`
: As above, but with a mode of `File::RDONLY`

`[String, Integer]`
: As above, but with the open mode given by the second argument

`[String, Integer, Integer]`
: As above, but with the open permissions given by the third argument

`:close`
: Close this file descriptor

Terminating

The current process may be terminated with `Kernel.exit(`*status*`=1)`. If `exit` is used within a `begin` block, it raises a `SystemExit` exception which may be caught by a corresponding `rescue` clause. Otherwise, it terminates the process with `exit(2)`, using *status* as the exit status. If *status* is `true`, it has the value `0`; if `false` or omitted, it has the value `1`. Prior to termination, `exit` runs any `at_exit` functions or object finalisers. `Kernel.exit!` is identical to `exit` except it bypasses both `at_exit` functions and finalisers.

`Kernel.abort(`*msg*`=nil)` displays the optional message on the standard error stream then terminates the current process. It is equivalent to `$stderr.puts` *msg*—if *msg* is given—then `exit(false)`.

Environment

`ENV` provides `Hash`-like access to the current process's environment variables. `ENV[`*var*`]` retrieves the value of the environment variable named *var*; `ENV[`*var*`]=`*value* sets its value to *value*. In both cases *var* is a `String`. If *value* is `nil`, the environment variable is deleted.

Status

POSIX systems record the status of stopped and terminated processes as a 16-bit integer. The lower 8 bits are the process's *exit status*, i.e. the value returned to its parent; the higher bits are platform-dependent. The status is encapsulated by a `Process::Status` object.

`Kernel.`` ` `` and `Kernel.system` set the global variable `$?` to the `Process::Status` object corresponding to the command they executed. Methods such as `exec`, `fork`, and `spawn`, cannot set `$?` because they return before the command has terminated. To retrieve the `Process::Status` object associated with such processes, we must first *wait* for them to exit.

Waiting

`Process.wait` waits for *any* child process to exit, then returns its PID. `$?` is set to the corresponding `Process::Status` object. `Process.wait2` also waits for any child process to exit, but returns an `Array` comprising its PID and `Process::Status` object. Both methods raise a `SystemError` if there aren't any child processes.

`Process.waitall` waits for *all* child processes to exit, then returns an `Array` of `[`*`pid`*`, `*`status`*`]` pairs, where *pid* is the process's PID, and *status* its `Process:Status` object. If there are no child processes, an empty `Array` is returned.

`Process.waitpid(`*`pid`*`, `*`flags`*`=0)` waits on the child process described by *pid* to exit, then returns its PID. `Process.waitpid2(`*`pid`*`, `*`flags`*`=0)` does likewise, but returns an `Array` comprising the PID and corresponding `Process::Status` object. Both methods interpret *pid* as follows:

pid	Semantics
< −1	Any child whose process group ID equals the absolute value of *pid*
−1	Any child
0	Any child whose process group ID equals that of the current process.
> 0	The child with the PID of *pid*.

flags is either `0`, or a logical OR of the following constants. Some platforms may not support these flags.

`Process::WNOHANG`
: The method will not block if status is not immediately available for one of the specified child processes.

`Process::WUNTRACED`
: The status of any specified child processes that are stopped, but whose status has not been reported since they stopped, will also be reported.

A child process that is not waited on may become a *zombie*—remaining in the process table in case somebody wants to retrieve its status. `Process.detach(`*`pid`*`)` offers a solution by running a background thread to

monitor the status of *pid*, then reap it—i.e. remove the process from the process table—when terminated. It returns the `Thread` object.

Process::Status

If a process has exited, `Process::Status#exitstatus` returns its exit status as an `Integer` byte. If it exited because of a signal, `Process::Status#termsig` returns the signal number; otherwise it returns `nil`. If the process was stopped by a signal, `Process::Status#stopsig` returns the signal number; otherwise it returns `nil`. `Process::Status#pid` returns the PID of the corresponding process. In addition, the following predicates are defined:

`Process::Status#coredump?`
: Returns `true` if the process generated a coredump when it terminated; `false` otherwise.

`Process::Status#exited?`
: Returns `true` if the process terminated normally; `false` otherwise.

`Process::Status#signaled?`
: Returns `true` if the process terminated due to the receipt of a signal which was not caught; `false` otherwise.

`Process::Status#stopped?`
: Returns `true` if the process is currently stopped; `false` otherwise. Only meaningful if the `Process::WUNTRACED` flag was given to `waitpid` or `waitpid2`.

`Process::Status#success?`
: Returns `true` if the process exited normally, `false` if it exited abnormally, and `nil` if it hasn't exited.

A `Process::Status` object may also be treated as a collection of bits: `Process::Status#to_i` returns the status as an `Integer`; `Process::Status#&(`*n*`)` performs a logical AND of the bits with *n*; and `Process::Status#>>(`*n*`)` shifts the bits right by *n* places.

Daemons

A *daemon* is a process that runs in the background—rather than under the direct control of a user—to provide a service for other programs. `Process.daemon(`*`keep_dir`*`=false, `*`redirect`*`=false)` detaches the current process from its controlling terminal, then runs it in the background. The process's working directory is set to `/` unless *keep_dir* is `true`. If *redirect* is `true`, the process's standard input, output, and error streams are all redirected to `/dev/null`. If the process is successfully daemonised, this method returns `0`; otherwise an `Errno` exception is raised.

This method uses the `daemon(3)` syscall if its available, or forks then calls `Process.setssid`. On platforms with neither option available, a `NotImplementedError` is raised.

Scheduling Priorities

On platforms that support the `getpriority(2)` and `setpriority(2)` system calls, the scheduling priority associated with a process, process group, or user, may be obtained and set. A priority is an `Integer` between −20 and 19, which defaults to 0. Lower priorities cause more favourable scheduling, but only the superuser may decrease a priority.

`Process.getpriority(`*`which`*`, `*`who`*`)` returns the most favourable priority associated with the specified processes. `Process.setpriority(`*`which`*`, `*`who`*`, `*`prio`*`)` sets the priorities of all the specified processes to *prio*, returning `0`. *which* is a constant describing the type of priority, *who* is an `Integer` identifying an instance of that type, and *prio* is an `Integer`. If *who* is `0` it refers to the current instance of that type, as shown below.

which	***who***	***who* = 0**
`Process::PRIO_PROCESS` (process)	Process ID	Calling process
`Process::PRIO_PGRP` (process group)	Process group ID	Process group of calling process

which	*who*	*who* = 0
`Process::PRIO_USER` (user)	User ID	Real user ID of calling process

Resource Limits

On platforms that support the `getrlimit(2)` and `setrlimit(2)` system calls, `Process.getrlimit(`*resource*`)` and `Process.setrlimit(`*resource*`, `*soft*`, `*hard*`)` may be used to obtain and set, respectively, resource limits.

The resources supported by Ruby are listed below. Those supported by your platform have a corresponding constant defined named `Process::RLIMIT_`*name*, where *name* is the resource's name. Both `Process.setrlimit` and `Process.getrlimit` accept *resource* as a resource name—given as a `String` or `Symbol`—or the value of the corresponding constant.

Name	Platform	Description
`AS`	SUSv3, NetBSD, FreeBSD, OpenBSD	Limits the maximum size of the process's address space, in bytes. If the stack expands beyond this limit, the process is sent the `SIGSEGV` signal.
`CORE`	SUSv3	Limits the maximum size of core files, in bytes. If `0`, they are never created; otherwise, they're truncated to this size.
`CPU`	SUSv3	Limits the maximum CPU time the process can consume, in seconds. If a process overruns it is sent the `SIGXCPU` signal.
`DATA`	SUSv3	Limits the maximum size of process's data segment and heap, in bytes.
`FSIZE`	SUSv3	Limits the maximum file size a process may create, in bytes. If the process expands a file beyond this limit, it is sent the `SIGXFSZ` signal.
`MEMLOCK`	BSD, GNU/ Linux	Limits the maximum number of bytes a process[6] can lock into memory.

6. Processes with the `CAP_SYS_IPC` capability, effectively root processes, are exempt.

Name	Platform	Description
`MSGQUEUE`	GNU/Linux	Limits the maximum number of bytes allocated for POSIX message queues.
`NICE`	GNU/Linux	Limits the maximum number[7] to which a process can lower its *nice* value.
`NOFILE`	SUSv3	Limits the maximum number, less 1, of file descriptors the process may have open.
`NPROC`	BSD, GNU/Linux	Limits the maximum number of processes the user may have running at any given time.
`RSS`	BSD, GNU/Linux ~<= 2.4	Limits the resident set size (maximum number of pages the process may have resident in memory).
`RTPRIO`	GNU/Linux	Limits the maximum real-time priority level a process[8] may request.
`RTTIME`	GNU/Linux	Limits the CPU time a process scheduled under a real-time policy may consume without making a blocking system call, in microseconds.
`SBSIZE`	NetBSD, FreeBSD	Limits the maximum size of socket buffer usage for this user, in bytes.
`SIGPENDING`	GNU/Linux	Limits the maximum number of signals that may be queued for this user.
`STACK`	SUSv3	Limits the maximum size of a process's stack, in bytes. If a process's stack grows beyond this point, it is sent a `SIGSEGV` signal.

A resource has both a *soft limit* and a *hard limit*. The hard limit is a ceiling on the soft limit, so may only be set by privileged processes. Either limit may be set to `0` or `:INFINITY`/`Process::RLIM_INFINITY`, which disable the resource and remove all limits on the resource, respectively. A process inherits the limits of its parent.

`Process.getrlimit` returns an `Array` whose first element is the soft limit of *resource*, and its last, the hard limit of *resource*. `Process.setrlimit` sets the soft limit of *resource* to *soft*, and, if permitted, the hard limit of *resource* to *hard*; if *hard* is omitted, it has the value *soft*.

7. Since a *nice* value, *n*, may be negative, this value is interpreted as 20 – *n*.
8. Processes with the `CAP_SYS_NICE` capability, effectively root processes, are exempt.

IDs

On UNIX-like systems, users, processes, and groups are associated with various IDs. In order to explain how these can be manipulated via methods of the `Process` module, some background theory is necessary…

When a process is invoked it is allocated a process ID (hereafter: PID) by which it can be uniquely identified. The PID of the current process is returned by `Process.pid`. Every process, other than *init*, has a *parent*: a process which spawned it. The parent's PID is returned by `Process.ppid`. On Windows, this always returns `0`.

A user, identified by a user ID (hereafter: UID), is a member of at least one group, each of which is identified by a group ID (hereafter: GID). When a user is a member of multiple groups, one is designated his *primary group*[9], and the remainder his *supplementary groups*. A user's GID is the GID of his primary group. Every process is also associated with a UID and GID. When a user logs in, the UID and GID of his login shell are set to his UID and GID, respectively. Normally a process inherits the UID and GID of its parent, so the processes invoked by a user will also be associated with his UID and GID. The UID of the current process is returned by `Process.uid`, and may be set with `Process.uid=`. Likewise, the GID of the current process may be retrieved and set with `Process.gid` and `Process.gid=`, respectively.

Processes are associated with a list of supplemental groups in much the same way. A user's login shell is associated with his supplementary groups, which are then inherited by the processes he creates. `Process.groups` returns an `Array` of GIDs for the current process's supplementary groups. `Process.groups=` is given an `Array` of GIDs or group names, with which it sets the process's supplementary group IDs. The `Array`s of GIDs must not contain more than 32 elements. This limit can be increased, up to a maximum of 4096, with `Process.maxgroups=`. The current limit is returned by `Process.maxgroups`. `Process.initgroups(`*user*`, `*group*`)` initialises the list of supplementary groups from *user*'s, and adds to this set *group*. *user* is a username given as a `String`, and *group* is a GID given as an `Integer`.

---

9. This is the group listed alongside a user's username in `/etc/passwd`.

A process is also associated with an *effective user ID* (hereafter: EUID) and an *effective group ID* (hereafter: EGID). Initially, the EUID has the same value as the UID, and the EGID has the same value as the GID. However, if the mode of an executable file has its *setuid* (a contraction of *set user ID*) and/or *setgid* bit set, the EUID or EGID, respectively, of the process is that of the file's owner/group. Before the EUID/EGID are set, the current value is saved in order to determine what EUIDs/EGIDs the user may switch to. For example, if a file was owned by user *zach* and had its setuid bit set, when user *zoe* executed it the process would have a UID and GID corresponding to her UID and GID, an EGID corresponding to her GID, but an EUID with the value of *zach*'s UID. The EUID and EGID of the current process are returned by `Process.euid` and `Process.egid`, respectively, and set with `Process.euid=` and `Process.egid=`, respectively.

For the purposes of job control, a process is assigned a *process group*. This allows signals to be sent to a group of processes at once. A process group also has an ID (hereafter: PGID), which is initially that of its leader. The PGID is returned by `Process.getpgrp`, and the PGID of an arbitrary process may be retrieved with `Process.getpgid(`*PID*`)`. `Process.getpgid(0)` is equivalent to `Process.getpgrp`. The PGID of the current process may be set to its PID with `Process.setpgrp`, while `Process.setpgid(`*PID*`, `*PGID*`)` sets the PGID of the process with a PID of *PID* to *PGID*. If either value is `0`, it implies the value of this attribute for the current process, i.e. `Process.setpgid(0,0)` is equivalent to `Process.setpgrp`.

A *session* is a collection of process groups. The session ID (hereafter: SID) is the PID of the session leader. "Sessions arrange a logged-in user's activities, and associate that user with a *controlling terminal*, which is a specific tty device that handles the user's terminal I/O." [Love07, pp. 154–155] . `Process.setsid` "creates a new process group inside of a new session, and makes the invoking process the leader of both", returning the new SID of the calling process. [Love07, pp. 156–157] However, this method will raise `Errno::EPERM` if the calling process is already a process group leader.

Process::GID

The `Process::GID` module provides a higher-level interface to GIDs. `Process::GID.change_privilege(`*gid*`)` changes the GID, EGID, and saved

GID to *gid*, which it then returns. On error, an `Errno` exception is raised. This is incompatible with `Process.gid=`.

The `setresgid(2)` and `setregid(2)` system calls allow the GID and EGID to be exchanged with each other. On such systems, `Process::GID.re_exchangeable?` returns `true`, and `Process::GID.re_exchange` may be used to set the GID to the current EGID, the EGID to the current GID, and the saved GID to the new EGID. `Process::GID.grant_privilege(`*egid*`)`, and its alias `Process::GID.eid=(`*egid*`)`, set the EGID to *egid*, or raise an `Errno` exception. If the GID and EGID may be exchanged, these methods also set the saved GID to *egid*.

`Process::GID.switch` sets the EGID to the GID, returning the former. If given a block whose body does not modify these values, it ensures that the EGID is reset to its original value.

The `Process::GID.sid_available?` predicate returns `true` if the operating system supports saved GIDs; `false`, otherwise. In the latter case, Ruby saves the GID itself and tries to emulate this functionality.

Process::UID

The `Process::UID` module is the equivalent of `Process::GID` for UIDs. The methods it provides are identical except they manipulate UIDs/EUIDs rather than GIDs/EGIDs. The `setresuid(2)` or `setreuid(2)` system call is required for exchanging the UID with the EUID.

Process::Sys

The `Process::Sys` module, however, provides lower-level access to the system calls used for manipulating user and group IDs. The `Process::Sys.getegid`, `Process::Sys.geteuid`, `Process::Sys.geteuid`, `Process::Sys.getgid`, and `Process::Sys.getuid` methods are aliases of `Process.egid`, `Process.euid`, `Process.gid`, and `Process.uid`, respectively. The following methods are defined in terms of the system call with the same

name as the method. Each raises an `Errno` exception if the system call fails, or `NotImplementedError` if the system call does not exist.

`Process::Sys.issetugid`
: Returns `true` if the process is either setuid or setgid; `false`, otherwise.

`Process::Sys.setegid(`*gid*`)`
: Sets the EGID to *gid*.

`Process::Sys.seteuid(`*uid*`)`
: Sets the EUID to *uid*.

`Process::Sys.setgid(`*gid*`)`
: Sets the GID to *gid*.

`Process::Sys.setregid(`*gid*`, `*egid*`)`
: Sets the real GID to *gid*, and the EGID to *egid*.

`Process::Sys.setresgid(`*gid*`, `*egid*`, `*sgid*`)`
: Sets the real GID to *gid*, the EGID to *egid*, and the saved GID to *sgid*.

`Process::Sys.setresuid(`*uid*`, `*euid*`, `*suid*`)`
: Sets the real UID to *uid*, the EUID to *euid*, and the saved UID to *suid*.

`Process::Sys.setreuid(`*uid*`, `*euid*`)`
: Sets the real UID to *uid*, and the EUID to *euid*.

`Process::Sys.setrgid(`*gid*`)`
: Sets the real GID to *gid*.

`Process::Sys.setruid(`*uid*`)`
: Sets the real UID to *uid*.

`Process::Sys.setuid(`*uid*`)`
: Sets the UID to *uid*.

Signalling

A *signal* is a software interrupt that provides a "a mechanism for handling asynchronous events". [Love07, pp. 279–279] These events may be generated by the operating system or elsewhere in the program, and signals can be sent from one process to another. "The key point is not just that the events occur asynchronously…but also that the program handles the signals asynchronously." [*ibid.*]

A process may register *signal handlers* to trap specific signals. When the process receives a corresponding signal the handler is invoked; otherwise, the signal is ignored. The `SIGKILL` and `SIGSTOP` signals can neither be trapped nor ignored: they always kill and stop, respectively, the process to which they are sent.

A signal is identified by both a name and a number. The name is a portable way to refer to a signal and always begins with `SIG`; the number is, in theory, platform-specific. `Signal.list` returns a `Hash` of signals supported by your platform. The keys are signal names without the `SIG` prefix, as `Strings`; the values, signal numbers as `Integers`. Methods that expect a signal as an argument accept this name as a `String` or `Symbol`—with or without the prefix—or the number.

Name	Number	Description	Default Action
`SIGABRT` / `SIGIOT`	6	Sent by `abort(3)`.	Dump core
`SIGALRM`	14	Sent by `alarm(2)`.	Terminate
`SIGBUS`	7	Hardware or alignment error.	Dump core
`SIGCHLD` / `SIGCLD`	17	Child has terminated.	Ignore
`SIGCONT`	18	Process has continued after being stopped.	Ignore
`SIGEXIT`	0	Ruby is about to terminate; prior to `at_exit` functions.	Terminate

Signals Supported on Linux

Name	Number	Description	Default Action
SIGFPE	8	Arithmetic exception.	Dump core
SIGHUP	1	Process's controlling terminal was closed.	Terminate
SIGILL	4	Process tried to execute an illegal instruction.	Terminate
SIGINT	2	User generated the interrupt character (Ctrl + C).	Terminate
SIGIO	29	Asynchronous I/O event.	Terminate
SIGIO / SIGPOLL	29	Asynchronous I/O event.	Terminate
SIGKILL	9	Process termination (untrappable).	Terminate
SIGPIPE	13	Process wrote to a pipe without readers.	Terminate
SIGPROF	27	Profiling timer expired.	Terminate
SIGPWR	30	Power failure.	Terminate
SIGQUIT	3	User generated the quit character (Ctrl + \).	Dump core
SIGSEGV	11	Memory access violation.	Dump core
SIGSTOP	19	Suspends execution of the process.	Stop
SIGSYS	31	Process tried to execute an invalid system call.	Dump core
SIGTERM	15	Process termination (trappable).	Dump core
SIGTRAP	5	Break point encountered.	Dump core
SIGTSTP	20	User generated the suspend character (Ctrl + Z).	Stop
SIGTTIN	21	Background process read from controlling terminal.	Stop
SIGTTOU	22	Background process wrote to controlling terminal.	Stop
SIGURG	23	Urgent I/O pending.	Ignore
SIGUSR1	10	Process-defined signal.	Terminate
SIGUSR2	12	Process-defined signal.	Terminate
SIGVTALRM	26	Generated by setitimer(2) when called with ITIMER_VIRTUAL flag.	Terminate

Name	Number	Description	Default Action
SIGWINCH	28	Size of controlling terminal window changed.	Ignore
SIGXCPU	24	Processor resource limits were exceeded.	Dump core
SIGXFSZ	25	File resource limits were exceeded.	Dump core

Sending

`Process.kill(`*sig*`, `*pid*$_0$*`…`*pid*$_n$*`)` sends the signal, *sig*, to the processes with the given PIDs. If *sig* is a negative `Integer` or its first character is `-`, it signals process groups instead of processes. If *sig* is an invalid signal number, `Errno::EINVAL` or `RangeError` is raised; if it is an invalid `String` or `Symbol`, an `ArgumentError` is raised instead.

A PID of zero refers to the current process. If one or more PIDs are invalid, an `Errno::ESRCH` is raised; if you lack permission to signal them, an `Errno::EPERM` is raised. However, if the PID which caused an exception was preceded by legitimate PIDs, the latter may have already been sent *sig*.

Trapping

`Kernel.trap(`*sig*`, `*command*`)`, and its alias `Process.trap`, register a handler for signal *sig*. If *command* is a `Proc`, or a block is supplied and *command* omitted, the `Proc`/block is invoked on receipt of *sig* with the signal number as its sole argument. Otherwise, *command* must be one of the following:

`"IGNORE"`
`"SIG_IGN"`
`""`
`nil`
: Ignores the signal.

`"DEFAULT"`
`"SIG_DFL"`
: Invokes Ruby's default handler.

`"EXIT"`
: Terminates the program.

`"SYSTEM_DEFAULT"`
: Invokes the operating system's default handler.

A program may have at most one signal handler per signal. If *sig* already has a handler registered, the old handler is replaced with the new, and the old handler is returned; otherwise, `trap` returns `nil`.

Times

The current process times are returned by `Process.times` as a `Struct::Tms` object with the following methods:

`#utime`
: User time: CPU time, in seconds, spent executing instructions of the calling process.

`#stime`
: System time: CPU time, in seconds, spent in the system while executing tasks for the calling process.

`#cutime`
: User time of children: sum of `#utime` and `#cutime` values for all waited-for terminated child processes.

`#utime`
: System time of children: sum of `#stime` and `#cstime` values for all waited-for terminated children.

TIMES

A `Time` object represents a specific moment as a time and a date, stored with microsecond granularity. It is stored as the number of seconds since the *Unix epoch*, which is 1970-01-01 01:00:00, so on some platforms `Time` cannot represent dates before 1970 or after 2038.

Instantiation

`Time.now` creates a `Time` instance for the current time, with the resolution of the system clock. `Time.new` is identical when called without arguments; otherwise it must be given between one and seven arguments specifying the year, month, day, hour, minute, second, and offset from UTC, with which it initialises a `Time` object. All but the year argument are optional.

`Time.at(`*`seconds`*`, `*`microseconds`*`=0)` creates a `Time` object representing *seconds* seconds and *microseconds* microseconds from the Epoch. Both arguments may be any non-`Complex` numeric: if *microseconds* is a `Float` or `Rational`, the time has nanosecond granularity. A negative *seconds* represents a time before the Unix epoch, but this construction is operating-system specific. If `Time.at` is given a `Time` object as its sole argument, the argument is returned.

`Time.utc`, and its alias `Time.gm`, create a `Time` object for a given time in UTC. If ten arguments are given they are: second, minute, hour, day of month, year, day of week, isDST?, and the time zone abbreviation. Otherwise, between one and seven arguments are required: year, month, day of month, hour, minute, second, microseconds. In the second form, all but the first argument are optional; the latter may be omitted or `nil`. `Time.local`, and its alias `Time.mktime`, are identical to `Time.utc`, except they interpret their arguments in the local time zone.

The arguments listed above all correspond to attributes of `Time` objects, so are described in Attributes.

Attributes

A `Time` object has the following primary attributes. If a `Time` method expects an argument corresponding to a given attribute, the argument must satisfy the attribute's range; if the argument is omitted, the attribute's default value is assumed.

Attribute	Description	Range	Default	Accessor
Year	Year (possibly including century)	Positive `Integer`.	N/A	`Time#year`
Month	Month of year	`Integer` (1–12) or three-letter English abbreviation, e.g. *3* or *Feb* for February.	1	`Time#mon`, `Time#month`
Day	Day of month	`Integer`: 1–31.	1	`Time#day`, `Time#mday`
Week day	Day of week	`Integer`: 0–6, where Sunday is 0.	0	`Time#wday`
Year day	Day of year	`Integer`: 1–366.	1	`Time#yday`
Hour	Hour on 24-hour clock	`Integer`: 0–23.	0	`Time#hour`
Minute	Minute of hour	`Integer`: 0–59.	0	`Time#min`
Second	Second of minute	`Integer`: 0–60[1].	0	`Time#sec`
Microseconds	Microsecond[2] of second	Positive numeric less than 1000000000.	0	`Time#usec`, `Time#tv_usec`
Nanoseconds	Nanosecond[3] of second	`Integer`	0	`Time#nsec`, `Time#tv_nsec`
Zone	Time zone abbreviation	`String`, e.g. *UTC* or *CST*.	UTC	`Time#zone`

1.This range allows for leap seconds.
2.1 microsecond is 10^{-6} seconds
3.1 nanosecond is 10^{-9} seconds

Attribute	Description	Range	Default	Accessor
isDST?	Does the time occur in daylight saving time?	`true` (yes) or `false` (no).	`false`	`Time#isdst`, `Time#dst?`

Predicates

`Time` objects respond to the following predicates. They return `true` if the condition is true; `false`, otherwise.

`Time#dst?`
`Time#isdst`
: Occurs in DST

`Time#friday?`
: Occurs on a Friday

`Time#gmt?`
`Time#utc?`
: Occurs in UTC

`Time#monday?`
: Occurs on a Monday

`Time#saturday?`
: Occurs on a Saturday

`Time#sunday?`
: Occurs on a Sunday

`Time#thursday?`
: Occurs on a Thursday

`Time#tuesday?`
: Occurs on a Tuesday

`Time#wednesday?`
: Occurs on a Wednesday

Arithmetic

`Time#+(`*`numeric`*`)` adds *numeric* seconds to the receiver, and returns the result. Conversely, `Time#-(`*`numeric`*`)` subtracts *numeric* seconds from the receiver and returns the result. If `Time#-` is given a `Time` argument instead, it returns a new `Time` object representing the difference between them.

`Time` objects implement `Time#<=>` with the standard semantics, and mix-in the `Comparable` module. This allows inequalities to be tested between a `Time` object and a `Numeric` object—where the latter represents a number a seconds since the Unix epoch—or two `Time` objects.

Formatting

`Time#strftime` converts the receiver to a `String` by means of a user-supplied *format string*, analogous to Format Strings. A format string contains any number of *format directives* surrounded by arbitrary characters. It is returned with each format directive substituted for its payload.

A format directive has the form `%`*`fwc`*, where *f* is zero or more *flags*, *w* is an optional *field width*, and *c* a mandatory *conversion specifier*. Therefore, the simplest format directive has the form `%`*`c`*. The minimum field width is the minimum number of characters the directive will be substituted for; if fewer would have used, they are padded with either zeros or blanks, as described in the Conversion Specifiers table. The flags are described in the Flags table.

Specifier	Description	Range
`a`	Abbreviated name of day	*Sun—Sat*
`A`	Name of day	*Sunday—Saturday*
`b` / `h`	Abbreviated name of month	*Jan—Dec*
`B`	Name of month	*January—December*

Conversion Specifiers

Specifier	Description	Range
`c`	Date and time representation for current locale: *"%a %b %e %T %Y"*	
`%C`	Century, i.e. year ÷ 100, zero-padded	00–99
`d`	Day of the month, zero-padded	*01–31*
`D` / `x`	Date representation, without time, for current locale: *"%m/%d/%y"*	
`e`	Day of the month, blank-padded	*1–31*
`F`	ISO 8601 date: *"%Y-%m-%d"*	*1–31*
`g`	ISO 8601 week-numbering year, last two digits	*00–99*
`G`	ISO 8601 week-numbering year	*0–*
`H`	Hour of day on 24-hour clock, zero-padded	*00–23*
`I`	Hour of day on 12-hour clock, zero-padded	*01—12*
`j`	Day of year, zero-padded	*001–366*
`k`	Hour of day on 24-hour clock, blank-padded	*0–23*
`l`	Hour of day on 12-hour clock, blank-padded	*1–12*
`L`	Millisecond of second, zero-padded	*000–999*
`m`	Month of year, zero-padded	*01—12*
`M`	Minute of the hour, zero-padded	*00–59*
`n`	Newline character	*"\n"*
`N`	Fractional seconds, 9 digits by default	*000000000–999999999*
`p`	Meridian indicator, upper-case	*AM*, *PM*
`P`	Meridian indicator, lower-case	*am–pm*
`r`	12-hour-clock time: *"%I:%M:%S %p"*	*12:00:00 AM—11:59:59 PM*
`R`	24-hour-clock time: *"%H:%M"*	*00:00—23:59*
`s`	Number of seconds since the Unix epoch	*0–*
`S`	Second of minute, zero-padded	*00–60*
`t`	Tab character	*"\t"*
`T` / `X`	24-hour-clock time: *"%H:%M:%S"*	*00:00:00—23:59:59*
`u`	Day of week, *1* being Monday	*1–7*
`U`	Week number of year, starting with the first Sunday as the first day of week *01*, zero-padded	*00–53*
`v`	VMS date: *%e-%b-%Y"*	

Specifier	Description	Range
V	ISO 8601 week number of week-numbering year, zero-padded	*01–53*
w	Day of week, *0* being Sunday	*0–6*
W	Week number of year, starting with the first Monday as the first day of week *01*, zero-padded	*00–53*
y	Year without century, zero-padded	*00–99*
Y	Year including century	*0–*
z	Hour and minute offset from UTC	+00:01–+23:59
Z	Abbreviated name of time zone	

Flag	Description
-	Don't pad numerical output
_	Pad with blanks
0	Pad with zeros
^	Uppercase the output
#	Invert the case of the output
:	Use colons for %z

Flags

Coercion

`Time#to_a` coerces its receiver into a ten-element `Array` of its attributes in this order: *second*, *minute*, *hour*, *day of month*, *month*, *year*, *day of week*, *day of year*, *isDST?*, and *zone*. All elements are `Integers` except *isDST?*, which is `true` or `false`, and *zone*, which is a `String`.

A `Time` object may be coerced into a `Float` or `Rational` with `Time#to_f` and `Time#to_r`, respectively. `Time#to_i`, and its alias `Time#tv_sec`, converts the receiver into an `Integer` by truncating any fractional seconds, i.e. it is equivalent to `time.to_f.to_i`.

For a local time, `Time#to_s` is equivalent to calling `Time#strftime('%Y-%m-%d %H:%M:%S %z')`; for a UTC times, it is equivalent to `Time#strftime('%Y-%m-%d %H:%M:%S UTC')`. Alternatively, `Time#asctime`, and its alias `Time#ctime`, return a canonical `String` representation of their receiver, equivalent to `Time#strftime('%a %b %e %T %Y')`.

Zone Conversions

The time zone associated with a `Time` object is set during initialisation, and queried with `Time#zone`. It can be changed to UTC with `Time#getutc`, and its alias `Time#getgm`, which return a new `Time` object representing the receiver in UTC. `Time#utc`, and its alias `Time#gmtime`, do likewise but convert the receiver in-place.

Similarly, `Time#getlocal` returns a new `Time` object representing the receiver in the local time zone. `Time#localtime` is identical except it modifies the receiver in-place.

REFERE

ARRAY

`Array[](object, …) #=> Array`
Creates then returns a new `Array` comprising *object*(s).

`Array.new(`*`size=0`*`, `*`object=nil`*`) #=> Array`
`Array.new(array) #=> Array`
`Array.new(size) {|i| } #=> Array`
Creates and returns a new `Array`. The first form creates an `Array` with *size* elements each with the value *object*; without any arguments, an empty `Array` is created. If *array* is given, *array*`.to_ary` is returned. If a block is given, an `Array` of *size* elements is created by calling the block with each index and using its return value as the corresponding element.

`Array.try_convert(object) #=> Array or nil`
Returns *obj*`.to_ary` or `nil` if this fails.

`Array#&(array) #=> Array`
Returns the set intersection of the receiver and argument: a new `Array` comprising the elements common to both without duplicates.

`Array#*(object) #=> Array or String`
When *object* is numeric, concatenates *object*`.to_i` copies of the receiver to create a new `Array`. Otherwise, equivalent to `self.join(`*`object`*`)`.

`Array#+(array) #=> Array`
Concatenates the receiver with the argument to create a new `Array`.

`Array#-(array) #=> Array`
Returns a copy of the receiver less elements appearing in *array*.

`Array#<<(object) #=> Array`
Appends *object* to the receiver, which it returns.

`Array#==(object) #=> true or false`
Converts *object* to an `Array` with `#to_ary`, then returns `true` if both `Arrays`

are the same length and all of their elements are equal according to `#==`; `false`, otherwise.

`Array#[](index) #=> Object`
`Array#[](start, length) #=> Array or nil`
`Array#[](range) #=> Array or nil`
Returns the element at index *index*, the elements with index *start* through to *start* + *length*, or the elements with indices in the `Range` *range*. When no elements match the constraint, the first form returns `nil`; the others return `[]`. Aliased by `Array#slice`.

`Array#[]=(index, object) #=> Object`
`Array#[](start, length, object) #=> Array or nil`
`Array#[](range, object) #=> Array or nil`
With an `Integer` index, sets the element at index *index* to *object*, expanding the `Array` if necessary. Otherwise, replaces the elements with index *start* through to *start* + *length*, or indices within the given `Range`, with *object*. In these last two forms, if *object* is an `Array`, its elements are substituted for the matched elements in the receiver; if not, they are replaced by *object*.

`Array#|(array) #=> Array`
Returns the union of the receiver and the argument: a new `Array` comprising elements from both `Arrays` without duplicates.

`Array#assoc(object) #=> Array`
Where the receiver is an `Array` of `Arrays`, returns the first sub-`Array` whose first element is `#==` to *object*, or `nil` if no such element is found.

`Array#at(index) #=> Object or nil`
Returns the element with index *index*, or `nil` if the index is out of range.

`Array#clear() #=> Array`
Returns the receiver with all elements removed.

`Array#clear() #=> Array`
Removes all elements from the receiver, which it then returns.

`Array#combination(size)` *{|array| }* `#=> Array or Enumerator`
Generates all combinations of size *size* from the receiver's elements. Returns

an `Enumerator` if the block is omitted, otherwise yields each combination to the block, then returns the receiver.

`Array#collect!() {|object| } #=> Array`
Passes each element of the receiver to the block, replacing it with the value the block returns. Aliased to `Array#map!`.

`Array#compact() #=> Array`
Returns a copy of the receiver less any `nil` elements.

`Array#compact!() #=> Array or nil`
Returns `nil` if the receiver doesn't contain `nil` elements, otherwise removes all such elements then returns the receiver.

`Array#concat(array) #=> Array`
Appends the elements in the given `Array` to the receiver, which it then returns.

`Array#count(`*`object`*`) #=> Integer`
`Array#count() {|object| } #=> Integer`
With no argument, equivalent to `#size`. If *object* is given, returns the number of elements in the receiver that equal *object*. Otherwise, pass each element of the receiver to the block, returning the number of times it returns `true`.

`Array#cycle(`*`times`*`) {|`*`object`*`| } #=> Enumerator or nil`
Invokes the block with each element of the receiver in turn, then repeats *times* times or forever if *times* is omitted. If the receiver is empty, returns `nil`. If the block is omitted, returns an `Enumerator`.

`Array#delete(object) { } #=> Object or nil`
Deletes every element of the receiver that is equal to *object*. If the receiver didn't contain *object* returns the value of the block, if given, or `nil` otherwise.

`Array#delete_at(index) #=> Object or nil`
Deletes and returns the element of the receiver with the given index. Returns `nil` if the index is out of range.

`Array#delete_if() {|`*`object`*`| } #=> Array`
Passes each element of the receiver to the block, deleting those for which the

block is `true`, then returns the receiver. Returns an `Enumerator` if the block is omitted.

`Array#each() {|object| } #=> Array`
Passes each element to the block, deleting those for which the block is `true`, then returns the receiver. Returns an `Enumerator` if the block is omitted.

`Array#each_index() {|index| } #=> Array`
Passes the index of each element to the block, then returns the receiver. Returns an `Enumerator` if the block is omitted.

`Array#empty?() #=> true or false`
Returns `true` if the receiver contains no elements; `false`, otherwise.

`Array#eql?(object) #=> true or false`
Returns `true` if *object* is an `Array` whose elements are equal to the receiver in both number and content—according to `#eql?`; `false`, otherwise.

`Array#fetch(index) {|index| } #=> Object`
`Array#fetch(index, default) #=> Object`
Returns the element at index *index*. If the index is out of range, the first form returns the value of the block when given the index, or raises an `IndexError` if the block is omitted; the second form returns *default*.

`Array#fill(object) #=> Array`
`Array#fill() {|index| } #=> Array`
`Array#fill(object, start, length=nil) #=> Array`
`Array#fill(start, length=nil) {|index| } #=> Array`
`Array#fill(object, range) #=> Array`
`Array#fill(range) {|index| } #=> Array`
Sets elements of the receiver to the value of the block, if given, or *object*. The first two forms set all elements, the second two set elements starting at indices *start* through to *start* + *length*, and the remainder set elements with indices in the given `Range`. If *start* is `nil` it is equivalent to a value of `0`; if *length* is `nil` it is equivalent to the length of the receiver.

`Array#find_index(object) #=> Integer or nil`
`Array#find_index() {|object| } #=> Integer or nil`
Returns the index of the first element `#==` to *object* or, if a block is supplied,

the first element for which the block is `true`. Returns `nil` if no elements matched, or an `Enumerator` if both argument and block are omitted. Aliased by `Array#index`.

`Array#flatten(`*`level=-1`*`) #=> Array`
Returns an `Array` comprising each element, or, if the element is itself an `Array`, the result of calling `#flatten` on that element. The depth of recursion depends on the value of *level*: if negative, the method always recurses; if zero, there is no recursion; if positive, the method only recurses a maximum of *level* levels.

`Array#flatten!(`*`level=-1`*`) #=> Array or nil`
Returns `nil` if no element is itself an `Array`, otherwise invokes `Array#flatten(`*`level`*`)` to modify the receiver in-place.

`Array#frozen?() #=> true or false`
Returns `true` if the receiver is frozen; `false`, otherwise.

`Array#index(object) #=> Integer or nil`
`Array#index() {|`*`object`*`| } #=> Integer or nil`
Aliases `Array#find_index`.

`Array#insert(index, object, …) #=> Array`
Locates the element whose index is *index* then inserts *object*(s) before this element if *index* is positive, or after if *index* is negative. Returns the receiver.

`Array#join(`*`separator=$,`*`) #=> String`
Concatenates each element with *separator*, then returns a concatenation of the result.

`Array#last(`*`count=1`*`) #=> Object`
Returns the last *count* elements of the receiver as an `Array`. If *count* is `1`, returns the corresponding element, or `nil` if the receiver is empty.

`Array#length() #=> Integer`
Returns the number of elements in the receiver.

`Array#map!() {|object| } #=> Array`
Aliases `Array#collect!`.

`Array#pack(directives) #=> String`
Combines the elements of the receiver into a binary `String` by applying *directives*.

`Array#permutation(size) {|array| } #=> Enumerator or Array`
Generates all permutations with length *size* of the receiver's elements, then yields them to the block; if the block is omitted, returns them as an `Enumerator`.

`Array#pop(n=1) #=> Object or nil`
Removes then returns the last *n* elements from the receiver. Returns `nil` if the receiver is empty.

`Array#product(array, …) #=> Array`
Returns the Cartesian product of the receiver and its *argument*(s) as an `Array` of `Arrays`. Each inner `Array` contains one element from the receiver, and one from each argument *array*.

`Array#push(object, …) #=> Array`
Appends each argument to the receiver, which it then returns.

`Array#rassoc(object) #=> Array or nil`
Where the receiver is an `Array` of `Arrays`, returns the first sub-`Array` whose second element is `#==` to *object*, or `nil` if no such element is found.

`Array#reject!() {|object| } #=> Array or nil`
Behaves as `Array#delete_if` but returns `nil` if the receiver wasn't modified.

`Array#repeated_combination(length) {|array| } #=> Array`
Yields each combination of *length* elements—some of which may be repeated—or returns an `Enumerator` if the block is omitted.

`Array#repeated_permutation(length) {|array| } #=> Array`
Yields each permutation of *length* elements—some of which may be repeated—or returns an `Enumerator` if the block is omitted.

`Array#replace(array) #=> Array`
Substitutes the elements of *array* for the elements of the receiver, which is resized if necessary, then returns the receiver.

`Array#reverse() #=> Array`
Returns a copy of the receiver with the elements in reverse order.

`Array#reverse!() #=> Array`
Behaves as `Array#reverse` but modifies the receiver in-place.

`Array#reverse_each() {|object| } #=> Array`
Yields each element of the receiver to the block in reverse order, then returns `self`.

`Array#rindex(object) #=> Integer or nil`
`Array#rindex() {|object| } #=> Integer or nil`
Returns the index of the last element `#==` to *object* or, if a block is supplied, the last element for which the block is `true`. Returns `nil` if no elements matched, or an `Enumerator` if both argument and block are omitted.

`Array#rotate(n=1) #=> Array`
Returns a new `Array` comprising the element at index *n*, each consecutive element, then the element at index `0` through to the element at index *n* `- 1`.

`Array#rotate!(n=1) #=> Array`
Behaves as `Array#rotate!` but modifies the receiver in-place.

`Array#sample(n=1) #=> Array or nil`
Returns *n* elements of the receiver selected pseudo-randomly. If the receiver has fewer than *n* elements, returns them all; if the receiver is empty and *n* is omitted, returns `nil`.

`Array#shift(n=1) #=> Object or nil`
Deletes and returns the first *n* elements of the receiver, shifting the remaining elements down to fill the gap. Returns `nil` if the receiver is empty.

`Array#shuffle() #=> Array`
Returns a copy of the receiver ordered pseudo-randomly.

`Array#shuffle!() #=> Array`
Behaves as `Array#shuffle` but modifies the receiver in-place.

`Array#size() #=> Integer`
Aliases `Array#length`.

`Array#slice(index) #=> Object`
`Array#slice(start, length) #=> Array or nil`
`Array#slice(range) #=> Array or nil`
Aliases `Array#[]`.

`Array#slice!(index) #=> Object or nil`
`Array#slice!(start, length) #=> Array or nil`
`Array#slice!(range) #=> Array or nil`
Behaves as `Array#slice` but modifies the receiver in-place, returning the deleted element(s) or `nil` if no changes were made.

`Array#sort!() {|a, b| } #=> Array`
Sorts the receiver in-place, then returns `self`. If the block is omitted, elements are sorted according to `#<=>`; otherwise, passes two elements at a time to the block, which is expected to return `-1` if the first element is less than the second, `0` if they are equal, and `1` if the first element is greater than the second.

`Array#to_a() #=> Array`
Returns the receiver converted to an `Array`, using `#to_ary` if called on a subclass of `Array`.

`Array#to_ary() #=> Array`
Returns the receiver.

`Array#to_s() #=> String`
Returns the receiver represented in `Array` literal notation.

`Array#transpose() #=> Array`
When the receiver is an `Array` of `Arrays`, returns a new `Array` whose rows are the receiver's columns, and whose columns are the receiver's rows.

`Array#uniq() #=> Array`
Returns a copy of the receiver with duplicate—according to `#eql?` and `#hash`—elements removed.

`Array#uniq!() #=> Array`

Behaves as `Array#uniq` but modifies the receiver in-place and returns `nil` if there were no duplicate elements.

`Array#unshift(object, …) #=> Array`

Prepends *object*(s) to the receiver, moving the existing elements upward, then returns `self`.

`Array#values_at(indices, …) #=> Array`

Returns an `Array` comprising elements in the receiver with the given indices, where *indices* is an `Integer` index or a `Range` of the same.

BASICOBJECT

`BasicObject#!(object) #=> true or false`
Returns `true` if *object* is `false` or `nil`; `false`, otherwise.

`BasicObject#==(object) #=> true or false`
Returns `true` if *object* is the same object as the receiver; `false`, otherwise. Aliased by `BasicObject#equal?`.

`BasicObject#!=(object) #=> true or false`
Returns the inverse of `BasicObject#==`.

`BasicObject#equal?(object) #=> true or false`
Aliases `BasicObject#==`.

`BasicObject#instance_eval(code, file, line) #=> Object`
`BasicObject#instance_eval() {|object| } #=> Object`
Executes a given piece of code in the context of the receiver. In the first form, *code* is a `String` of Ruby, and *file* and *line* are the filename and line number, respectively, to be used in error messages. In the second, the block is passed the receiver as an argument, then invoked with `self` set to the receiver.

`BasicObject#instance_exec(argument, …) {|argument, …| } #=> Object`
Yields its arguments to the block, within which `self` is set to the receiver.

`BasicObject#__id__() #=> Fixnum`
Returns an identifier for the receiver which distinguishes it from all other active objects.

`BasicObject#__send__(name, argument, …, &block) #=> Object`
Sends a message named *name* to the receiver with the given *argument*(s) and *block*, returning the result.

`BasicObject#initialize(argument, …) #=> Object`
Hook called by `Class#new` on a newly allocated object, receiving any arguments passed to `Class#new`.

`BasicObject#method_missing(name, *argument*, …) #=> Object`
Called when the receiver is sent a message for which it has no method defined: *name* is the message selector as a `Symbol`, and *argument*(s) the argument(s) it was sent with.

`Kernel#singleton_method_added(name) #=> Object`
Hook invoked when a singleton method is added to the receiver, where *name* is the method's name as a `Symbol`.

`Kernel#singleton_method_removed(name) #=> Object`
Hook invoked when a singleton method is removed from the receiver, where *name* is the method's name as a `Symbol`.

`Kernel#singleton_method_undefined(name) #=> Object`
Hook invoked when a singleton method is undefined in the receiver, where *name* is the method's name as a `Symbol`.

BIGNUM

`Bignum#%(number) #=> Numeric`
Returns the result of the receiver modulo *number*. Aliased by `Bignum#modulo`.

`Bignum#&(number) #=> Numeric`
Returns the result of a bitwise AND between the receiver and *number*.

`Bignum#*(number) #=> Numeric`
Returns the result of multiplying *number* with the receiver.

`Bignum#**(number) #=> Numeric`
Returns the result of raising the receiver to the *number*th power.

`Bignum#+(number) #=> Numeric`
Returns the result of adding the receiver to *number*.

`Bignum#-(number) #=> Numeric`
Returns the result of subtracting *number* from the receiver.

`Bignum#-@() #=> Numeric`
Returns the receiver with a negative sign.

`Bignum#/(number) #=> Numeric`
Returns the result of dividing—using integer division—the receiver by *number*. Aliased by `Bignum#div`.

`Bignum#<(number) #=> true or false`
Returns `true` if the receiver is less than *number*; otherwise, `false`.

`Bignum#<<(number) #=> Numeric`
Returns the result of left-shifting *number* bits of the receiver.

`Bignum#<=(number) #=> true or false`
Returns `true` if the receiver is less than or equal to *number*; otherwise, `false`.

`Bignum#<=>(number) #=> -1, 0, 1`
Returns `-1` if the receiver is less than *number*, `0` if they are equal, and `1` if it is greater.

`Bignum#==(number) #=> true or false`
Returns `true` if the *number* is a `Numeric` with the same value as the receiver; `false`, otherwise. Aliased by `Bignum#===`.

`Bignum#===(number) #=> true or false`
Aliases `Bignum#==`.

`Bignum#>(number) #=> true or false`
Returns `true` if the receiver is greater than *number*; otherwise, `false`.

`Bignum#>=(number) #=> true or false`
Returns `true` if the receiver is greater than or equal to *number*; otherwise, `false`.

`Bignum#>>(number) #=> Numeric`
Returns the result of right-shifting *number* bits of the receiver and its sign.

`Bignum#[](bit) #=> 0 or 1`
Returns the bit^{th} bit of the receiver, where the 0^{th} bit is the least significant.

`Bignum#^(number) #=> Numeric`
Returns the result of a bitwise EXCLUSIVE OR between the receiver and *number*.

`Bignum#abs() #=> Bignum`
Returns the absolute value of the receiver. Aliased by `Bignum#magnitude`.

`Bignum#coerce(number) #=> Array`
Returns an `Array` whose first element is *number* as a `Bignum`, and last element is the receiver. If *number* is neither a `Fixnum` nor a `Bignum`, a `TypeError` is raised.

`Bignum#div(number) #=> Numeric`
Aliases `Bignum#/`.

`Bignum#divmod(number) #=> Array`
Divides the receiver by *number*, returning an `Array` whose first element is the quotient, and last element, the modulus. The quotient is rounded toward $-\infty$.

`Bignum#even?() #=> true or false`
Returns `true` if this number is even; otherwise, `false`.

`Bignum#eql?(number) #=> true or false`
Returns `true` if *number* is a `Bignum` with the same value as the receiver; `false`, otherwise.

`Bignum#fdiv(number) #=> Float`
Returns the result of dividing—using floating-point division—the receiver by *number*. Aliased by `Bignum#quo`.

`Bignum#magnitude() #=> Bignum`
Aliases `Bignum#abs`.

`Bignum#modulo(number) #=> Numeric`
Aliases `Bignum#%`.

`Bignum#odd?() #=> true or false`
Returns `true` if this number is odd; otherwise, `false`.

`Bignum#remainder(number) #=> Numeric`
Divides the receiver by *number*, returning the remainder.

`Bignum#size() #=> Integer`
Returns the number of bytes used to represent the receiver.

`Bignum#to_f() #=> Bignum`
Converts the receiver to a `Float`, or `Float::INFINITY` if its too big.

`Bignum#to_s(base=10) #=> Bignum`
Returns a `String` representation of the receiver in the given base, where *base* is between `2` and `36` inclusive.

`Bignum#|(number) #=> Numeric`
Returns the result of a bitwise OR between the receiver and *number*.

```
Bignum#~() #=> Numeric
```

Returns the result of inverting the receiver's bits.

BINDING

`Binding#eval(code, `*`filename`*`, `*`line`*`) #=> Object`

Evaluates in the context of the receiver the `String` of Ruby given as *code*, returning the result. If *filename* and/or *line* are given, they are the filename and line number, respectively, that will be used in error messages generated by *code*.

CLASS

`Class.inherited(class) #=> Object`
Hook that is fired when a subclass of the receiver class is created; *class* is the subclass as a `Class` object.

`Class.new(superclass=Object) { } #=> Class`
Creates and returns an anonymous class that inherits from *superclass*. If a block is supplied it is evaluated in the context of this class: within it, `self` is the new `Class` instance.

`Class#allocate() #=> Object`
Allocates memory for an instance of the receiver's class, then returns the new object. Invoked automatically by the interpreter when `#initialize` is called; cannot be overridden.

`Class#new(argument, …) #=> Object`
Creates a new instance of the receiver's class with `#allocate`, which it initialises with `#initialize(argument, …)`, then returns.

`Class#superclass() #=> Class or nil`
Returns the superclass of the receiver, or `nil` if the receiver is `BasicObject`.

COMPARABLE

`Comparable#<(object) #=> true or false`
Returns `true` if `#<=>(`*`object`*`)` is negative; `false`, otherwise.

`Comparable#<=(object) #=> true or false`
Returns `true` if `#<=>(`*`object`*`)` is negative or zero; `false`, otherwise.

`Comparable#==(object) #=> true or false`
Returns `true` if `#<=>(`*`object`*`)` is zero; `false`, otherwise.

`Comparable#>=(object) #=> true or false`
Returns `true` if `#<=>(`*`object`*`)` is zero or positive; `false`, otherwise.

`Comparable#>(object) #=> true or false`
Returns `true` if `#<=>(`*`object`*`)` is positive; `false`, otherwise.

`Comparable#between?(minimum, maximum) #=> true or false`
Returns `true` if the receiver is between *minimum* and *maximum*—i.e. `#<=>(`*`minimum`*`)` is zero or positive and `#<=>(`*`maximum`*`)` is zero or negative—or `false`, otherwise.

COMPLEX

`Complex.polar(magnitude, angle=0) #=> Complex`
Returns the `Complex` number represented by the polar coordinates *magnitude* and *angle*.

`Complex.rect(real, imaginary=0) #=> Complex`
Returns the `Complex` number with the real part, *real*, and the imaginary part, *imaginary*. Aliased by `Complex#rectangular`.

`Complex.rectangular(real, imaginary=0) #=> Complex`
Aliases `Complex#rect`.

`Complex#+(number) #=> Complex`
Returns the result of adding the receiver to *number*.

`Complex#-(number) #=> Complex`
Returns the result of subtracting *number* from the receiver.

`Complex#*(number) #=> Complex`
Returns the result of multiplying *number* with the receiver.

`Complex#/(number) #=> Complex`
Returns the result of dividing the receiver by *number*. Aliased by `Complex#quo`.

`Complex#**(number) #=> Complex`
Returns the result of raising the receiver to the $number^{th}$ power.

`Complex#-@() #=> Complex`
Returns the receiver with a negative sign.

`Complex#==(number) #=> true or false`
Returns `true` if the *number* is a `Numeric` with the same value as the receiver; `false`, otherwise.

`Complex#abs() #=> Complex`
Returns the absolute value of the receiver. Aliased by `Complex#magnitude`.

`Complex#abs2() #=> Complex`
Returns the square of the absolute value of the receiver.

`Complex#angle() #=> Float`
Returns the amplitude of the receiver, i.e. `atan2(`*`imaginary`*`, `*`real`*`)`, where *imaginary* is the receiver's imaginary part, and *real* its real part. Aliased by `Complex#arg` and `Complex#phase`.

`Complex#arg() #=> Float`
Aliases `Complex#angle`.

`Complex#conj() #=> Complex`
Returns the conjugate of the receiver: its real part minus its imaginary part. Aliased by `Complex#conjugate`.

`Complex#conjugate() #=> Complex`
Aliases `Complex#conj`.

`Complex#denominator() #=> Integer`
Returns the denominator of the receiver: the least common multiple of the denominators of both real and imaginary parts.

`Complex#eql?(number) #=> true or false`
Returns `true` if *number* is a `Complex` whose real and imaginary parts are `#eql?` to the receiver's real and imaginary parts, respectively; `false`, otherwise.

`Complex#fdiv(number) #=> Float`
Returns the result of dividing—using floating-point division—the receiver by *number*.

`Complex#imag() #=> Numeric`
Returns the imaginary part of the receiver. Aliased by `Complex#imaginary`.

`Complex#imaginary() #=> Numeric`
Aliases `Complex#imag`.

`Complex#magnitude() #=> Complex`
Aliases `Complex#abs`.

`Complex#numerator() #=> Complex`
Returns the numerator of the receiver.

`Complex#phase() #=> Float`
Aliases `Complex#angle`.

`Complex#polar() #=> Array`
Returns the receiver's polar coordinates as a two-element `Array`: the first element is `Complex#abs`, and the last element is `Complex#arg`.

`Complex#quo(number) #=> Complex`
Returns the result of dividing—after converting the real and imaginary parts of the receiver to `Rationals`—the receiver by *number*.

`Complex#rect() #=> Array`
Returns an `Array` whose first element is the receiver's real part, and last element is the receiver's imaginary part. Aliased by `Complex#rectangular`.

`Complex#rectangular() #=> Array`
Aliases `Complex#rect`.

`Complex#real() #=> Numeric`
Returns the real part of the receiver.

`Complex#real?() #=> false`
Returns `false`.

`Complex#to_f() #=> Float`
Returns the real part of the receiver as a `Float`; raises a `RangeError` if the imaginary part is non-zero.

`Complex#to_i() #=> Integer`
Returns the real part of the receiver as a `Integer`; raises a `RangeError` if the imaginary part is non-zero.

`Complex#to_r() #=> Rational`

Returns the real part of the receiver as a `Rational`; raises a `RangeError` if the imaginary part is non-zero.

DIR

```
  Dir[](array) #=> Array
Dir[](string, …) #=> Array
```
Passes its arguments to `Dir.glob`, returning the result.

```
  Dir.chdir(directory=ENV['HOME'] || ENV['LOGDIR']) {|directory| }
#=> Object or 0
```
Changes the current working directory of the process to *directory* and returns `0`. If a block is supplied, it receives the new working directory as an argument and ensures the original working directory is restored when the block exits; the return value is that of the block. However, it is an error for multiple threads to have these blocks open simultaneously. If *directory* does not exist the appropriate `Errno::` exception is raised.

```
  Dir.chroot(directory) #=> 0
```
Changes the root directory of the process—assuming it has the appropriate privileges—to *directory*. A `NotImplementedError` is raised on platforms without the `chroot(2)` system call.

```
  Dir.delete(directory) #=> 0
```
Deletes the empty directory *directory*, raising an `Errno::` exception on error. Aliased to `Dir.rmdir` and `Dir.unlink`.

```
  Dir.entries(directory) #=> Array
```
Returns the entries—filenames, `.`, and `..`—of *directory* as an `Array` of `Strings`. Raises an `Errno::` exception if the directory doesn't exist.

```
  Dir.exist?(directory) #=> true or false
```
Returns `true` if *directory* exists and is a directory; `false`, otherwise. Aliased by `Dir.exists?` and `File.directory?`.

```
  Dir.exists?(directory) #=> true or false
```
Aliases `Dir.exist?`.

```
Dir.foreach() {|entry| } #=> Enumerator or nil
```
Yields each entry—filenames, `.`, and `..`—of *directory* to the given block; returns an `Enumerator` if the block is omitted.

```
Dir.getwd() #=> String
```
Returns the canonical path of the current working directory for this process. Aliased by `Dir.pwd`.

```
Dir.glob(pattern, flags=0) {|filename| } #=> Array or false
```
Yields the filenames matching the glob pattern *pattern* to the block; returning an `Array` if the block is omitted. The syntax of *pattern* and valid *flags* are explained in Globbing.

```
Dir.home(user) #=> String
```
Returns the home directory of *user* or, if that argument's omitted, the current user.

```
Dir.mkdir(directory, permissions=0777) #=> 0
```
Creates a directory named *directory* with the permissions given in *permissions*. The permissions are ignored on Windows, and modified by the current process's umask.

```
Dir.new(directory, encoding: encoding) #=> Dir
```
Instantiates and returns a `Dir` object representing the directory named *directory*. The directory is assumed to have the same encoding as the local file system—i.e. `Encoding.find('filesystem')`—but an alternative encoding can be specified as *encoding*.

```
Dir.open(directory, encoding: encoding) {|dir| } #=> Dir or Object
```
Behaves as `Dir.new`, but if a block is supplied the new `Dir` object is yielded to it, then closed when the block exits. Returns the value of the block, if one was given, or the new `Dir` object.

```
Dir.pwd() #=> String
```
Aliases `Dir.getwd`.

```
Dir.rmdir(directory) #=> 0
```
Aliases `Dir.delete`.

`Dir.unlink(directory) #=> 0`
Aliases `Dir.delete`.

`Dir#close() #=> nil`
Closes the directory stream represented by the receiver.

`Dir#each() {|entry| } #=> Enumerator or Dir`
Yields each entry—filenames, `.`, and `..`—of the receiver's directory stream to the block, then returns `self`. Returns an `Enumerator` if the block is omitted.

`Dir#path() #=> String`
Returns the path of the directory stream represented by the receiver. Aliased by `Dir#to_path`.

`Dir#pos() #=> Integer`
Returns the current position in the directory stream represented by the receiver. Aliased by `Dir#tell`.

`Dir#pos=(position) #=> Integer`
Seeks the receiver's directory stream to the given position—which should have previously been returned by `Dir#pos`—then returns *position*.

`Dir#read() #=> String or nil`
Returns the next entry in the receiver's stream, then advances the stream's position. Returns `nil` after the last entry.

`Dir#rewind() #=> Dir`
Resets the position of the receiver's directory stream to its beginning, returning `self`.

`Dir#seek(position) #=> Integer`
Seeks the receiver's directory stream to the given position—which should have previously been returned by `Dir#pos`—then returns *self*.

`Dir#tell() #=> Integer`
Aliases `Dir#pos`.

`Dir#to_path() #=> String`
Aliases `Dir#path`.

ENCODING

`Encoding.aliases() #=> Hash`
Maps encoding aliases to their canonical names.

`Encoding.compatible?(a, b) #=> Encoding or nil`
Compares the encoding of its two arguments, which are either Encoding objects or objects associated with encodings. If they are compatible, returns the encoding which would result from their combination; otherwise, `nil`.

`Encoding.default_external() #=> Encoding`
Returns the default external encoding.

`Encoding.default_external=(encoding) #=> Encoding`
Sets the default external encoding to *encoding*, which may be an `Encoding` object or an encoding name.

`Encoding.default_internal() #=> Encoding or nil`
Returns the default internal encoding, or `nil` if there isn't one.

`Encoding.default_internal=(encoding) #=> Encoding`
Sets the default internal encoding to *encoding*, which may be an `Encoding` object, an encoding name, or `nil`.

`Encoding.find(encoding) #=> Encoding`
Returns the `Encoding` object representing the named encoding, raising an `ArgumentError` for invalid names. *encoding* may be a `String` or `Symbol` and may name an encoding alias. An *encoding* of *external* returns the default external encoding; *filesystem*, the filesystem encoding; *internal*, the default internal encoding; and *locale*, the locale encoding.

`Encoding.list() #=> Array`
Returns an `Array` of `Encoding` objects currently loaded. Aliases are excluded.

`Encoding.locale_charmap() #=> String`
Returns the name of the locale charmap encoding, if it could be derived from the environment, or `nil`.

`Encoding.name_list() #=> Array`
Returns the names of the currently loaded encodings as an `Array` of `Strings`. Aliases are included.

`Encoding#ascii_compatible?() #=> true or false`
Returns `true` if the receiver is ASCII-compatible; `false`, otherwise.

`Encoding#dummy?() #=> true or false`
Returns `true` if the receiver is a dummy encoding; `false`, otherwise.

`Encoding#name() #=> String`
Returns the name of this encoding.

`Encoding#names() #=> Array`
Returns the name of this encoding and those of its aliases as an `Array` of `Strings`.

`Encoding#replicate(name) #=> Encoding`
Returns a replica of the receiver named *name*; raises an `ArgumentError` if the given name is already in use.

ENCODING::CO

`Encoding::Converter.asciicompat_encoding(encoding) #=> Encoding or nil`
Returns the ASCII-compatible encoding corresponding to the given encoding, where *encoding* is either an `Encoding` object or an encoding name as a `String`. Returns `nil` if there isn't such an encoding.

`Encoding::Converter.new(source, destination, options) #=> Encoding::Converter`
`Encoding::Converter.new(conversion_path) #=> Encoding::Converter`
Instantiates an `Encoding::Converter` object for transcoding between a *source* and *destination* encoding, both of which may be either `Encoding` objects or the names of encodings as `Strings`. *options* is a transcoding options `Hash`. If *conversion_path* is given it should be an `Array` in the form returned by either `Encoding::Converter.search_convpath` or `Encoding::Converter#convpath`.

`Encoding::Converter.search_convpath(source, destination, options) #=> Array`
Returns the steps in the conversion path between a *source* and *destination* encoding, both of which may be either `Encoding` objects or the names of encodings as `Strings`. *options* is a transcoding options `Hash`. A step involving transcoding from one encoding to another is represented as `Array` containing two `Encoding` objects. A step involving a decorator is the decorator's name as a `String`.

`Encoding::Converter#convert(source) #=> String`
Transcodes the given `String` along the receiver's conversion path. *source* is assumed to be part of a larger `String`, so this method can be called repeatedly with the next chunk of input. Accordingly, after all input has been transcoded, `Encoding::Converter#finish` should be invoked.

`Encoding::Converter#convpath() #=> Array`
Returns the steps in the conversion path used by the receiver. A step involving transcoding from one encoding to another is represented as `Array`

containing two Encoding objects. A step involving a decorator is the decorator's name as a String.

`Encoding::Converter#destination_encoding() #=> Encoding`
Returns the Encoding to which the receiver transcodes.

`Encoding::Converter#finish() #=> String`
Signals that there is no more input to be transcoded, returning the final piece of the destination String.

`Encoding::Converter#insert_output(string) #=> nil`
Converts the given String into the destination encoding, then appends it to the output buffer.

`Encoding::Converter#last_error() #=> Exception or nil`
Returns an Exception corresponding to the last error the receiver encountered—i.e. Encoding::InvalidByteSequenceError or Encoding::UndefinedConversionError—or nil if no Exception occurred.

`Encoding::Converter#primitive_convert(source, destination, destination_offset=nil`
`#=> Symbol`
Converts the *source* String along the receiver's conversion path, appending the result to the *destination* String, and returning a Symbol indicating the state of the converter. If *destination_offset* is non-nil, it is an Integer specifying the byte position in *destination* where the result should be inserted. If *destination_size* is non-nil, it is an Integer specifying the maximum number of bytes to insert into *destination*. *options* is an options Hash. See Primitive Conversion for details.

`Encoding::Converter#primitive_errinfo() #=> Array`
Returns details of the last transcoding error the receiver encountered as an Array: the first element is the last Symbol returned by Encoding::Converter#primitive_convert, the next two are the names of the encodings in the current step in the conversion path as Strings, the next is the byte sequence which caused this error, and the last is the byte sequence which will be read again when Encoding::Converter#primitive_convert is next called. The last four of these elements are only meaningful when the

first is `:invalid_byte_sequence`, `:incomplete_input`, or `:undefined_conversion`.

`Encoding::Converter#putback(`*max*`) #=> String`

Puts the *read again* bytes from the last `Encoding::InvalidByteSequenceError` back into the input buffer so they will be transcoded again. If *max* is given, it is an `Integer` specifying the maximum number of bytes to put back.

`Encoding::Converter#replacement() #=> String`

Returns the receiver's replacement `String`.

`Encoding::Converter#replacement=(string) #=> String`

Sets the receiver's replacement `String` to *string*, which it then returns.

`Encoding::Converter#source_encoding() #=> Encoding`

Returns the `Encoding` from which the receiver transcodes.

ENUMERABLE

`Enumerable#all?() {|object| } #=> true or false`
Passes each element of the receiver to the block, returning `true` if the block is always `true`; `false`, otherwise. If the block is omitted, returns `true` if every element is neither `false` nor `nil`; `false`, otherwise.

`Enumerable#any?() {|object| } #=> true or false`
Passes each element of the receiver to the block, returning `true` as soon as the block is `true`; `false`, otherwise. If the block is omitted, returns `true` if at least one element is neither `false` nor `nil`; `false`, otherwise.

`Enumerable#chunk() {|object| } #=> Enumerator`
`Enumerable#chunk(initial_state) {|object, state| } #=> Enumerator`
Enumerates consecutive chunks of elements for which the block returns the same value. A chunk comprises the return value of the block, and an `Array` of corresponding elements. If the block returns `nil` or `:_separator` the corresponding element is dropped; if it returns `:_alone`, the element is the sole member of its chunk. If *initial_state* is given, it is duplicated for each iteration and passed to the block as a second argument: it can be used to maintain state.

`Enumerable#collect() {|object| } #=> Enumerator`
Passes each element of the receiver to the block, returning an `Array` of its results. Returns an `Enumerator` if the block is omitted. Aliased by `Enumerable#map`.

`Enumerable#collect_concat() {|object| } #=> Array or Enumerator`
Behaves like `Enumerable#collect`, but flattens the result `Array` before returning. Aliased by `Enumerable#flat_map`.

`Enumerable#count(object) #=> Integer`
`Enumerable#count() {|object| } #=> Integer`
Returns how many elements of the receiver equal *object* or for which the block is `true`. If both *object* and block are omitted, returns the total number of elements in the receiver.

`Enumerable#cycle(`*`times`*`) {|`*`object`*`| } #=> Enumerator or nil`
Invokes the block with each element of the receiver in turn, then repeats *times* times or forever if *times* is omitted. If the receiver is empty, returns `nil`. If the block is omitted, returns an `Enumerator`.

`Enumerable#detect(`*`default=->{ nil }`*`) {|`*`object`*`| } #=> Object, Enumerator, or nil`
Passes each element to the block, returning the first for which the block is `true`. If the block is never `true`, the result of calling the *default* `Proc` is returned. Aliased by `Enumerable#find`.

`Enumerable#drop(n) #=> Array`
Returns all but the first *n* elements.

`Enumerable#drop_while() {|`*`object`*`| } #=> Array or Enumerator`
Returns the first element for which the block is `false` along with all that follow, or an `Enumerator` if the block is omitted.

`Enumerable#each_entry() {|`*`object`*`| } #=> Enumerable or Enumerator`
Behaves like `#each` except if `#each` yielded multiple values at once, this method yields them as an `Array` rather than separate parameters. Returns an `Enumerator` if the block is omitted.

`Enumerable#each_cons(size) {|`*`object`*`| } #=> Enumerator or nil`
Yields consecutive sub-`Arrays` of size *size*: the first contains elements `0`–(*size*–1), the second, elements `1`–*size*, and so forth. An `Enumerator` is returned if the block is omitted.

`Enumerable#each_slice(size) {|`*`object`*`| } #=> Enumerator or nil`
Yields each group of *size* elements as `Arrays`: the first contains the first *size* elements, the second, the next *size* elements, and so forth. An `Enumerator` is returned if the block is omitted.

`Enumerable#each_with_index(`*`argument`*`, …) {|`*`object, index`*`| } #=> Enumerable or Enumerator`
Yields each element along with its index, or returns an `Enumerator` if the block is omitted. Any *argument*s are passed to `#each`.

`Enumerable#each_with_object(object) {|element, object| } #=> Object or Enumerator`
Yields each element along with *object*, then returns *object*. Returns an `Enumerator` if the block is omitted.

`Enumerable#entries(argument, …) #=> Array`
Returns the elements as an `Array`. Any *argument*s given are passed to `#each`. Aliased by `Enumerable#to_a`.

`Enumerable#find(default=->{ nil }) {|object| } #=> Object, Enumerator, or nil`
Aliases `Enumerable#detect`.

`Enumerable#find_all() {|object| } #=> Array or Enumerator`
Returns the elements for which the block is `true`, or returns an `Enumerator` if the block is omitted. Aliased by `Enumerable#select`.

`Enumerable#find_index() {|object| } #=> Integer, Enumerator, or nil`
Returns the index of the first element for which the block is `true`, or `nil` if it never is. Returns an `Enumerator` if the block is omitted.

`Enumerable#first() #=> Object or nil`
`Enumerable#first(n) #=> Array`
Returns the first element, or if an argument is given, the first *n* elements.

`Enumerable#flat_map() {|object| } #=> Array or Enumerator`
Aliases `Enumerable#collect_concat`.

`Enumerable#grep(pattern) {|object| } #=> Array`
Returns the elements which are equal with `#===` to *pattern*. If the block is given, each element is mapped through it before being appended to the result `Array`.

`Enumerable#group_by() {|object| } #=> Hash or Enumerator`
Returns a `Hash` mapping values returned by the block to `Arrays` of elements for which the block returned that value, or an `Enumerator` if the block is omitted.

`Enumerable#include?(object) #=> true or false`
Returns `true` as soon as an element is equal—in terms of `#==`—to *object*; `false` if there is no such element. Aliased by `Enumerable#member?`.

`Enumerable#inject(`*initial*`) {|accumulator, element| } #=> Object`
`Enumerable#inject(initial, selector) #=> Object`
`Enumerable#inject(selector) #=> Object`
Iterates over the receiver, accumulating a return value. The first form yields both an accumulator object and an element. On the first iteration the accumulator is initialised to *initial*, if given; or the first element, if not—in this case, the first element isn't yielded. Subsequently, the accumulator is assigned the value last returned by the block. The other forms are like the first but with an implicit block of `{|accumulator, element| accumulator.send(selector, element)}`. The return value is that of the block on the final iteration. Aliased by `Enumerable#reduce`.

`Enumerable#map() {|object| } #=> Array or Enumerator`
Aliases `Enumerable#collect`.

`Enumerable#max() {|a, b| } #=> Object`
Returns the element with the maximum value by passing each pair to the block, and expecting a return value congruous with that of `<=>`. If the block is omitted, compares elements with `#<=>`, instead.

`Enumerable#max_by() {|object| } #=> Object or Enumerator`
Returns the element for which the block returned the largest value, or an `Enumerator` if the block is omitted.

`Enumerable#member?(object) #=> true or false`
Aliases `Enumerable#include?`.

`Enumerable#min() {|a, b| } #=> Object`
Returns the element with the minimum value by passing each pair to the block, and expecting a return value accordant with that of `#<=>`. If the block is omitted, compares elements with `#<=>`, instead.

`Enumerable#min_by() {|object| } #=> Object or Enumerator`
Returns the element for which the block returned the smallest value, or an `Enumerator` if the block is omitted.

`Enumerable#minmax() {|a, b| } #=> Array`
Passes its arguments to both `Enumerable#min` and `Enumerable#max`, and returns their values.

`Enumerable#minmax_by() {|a, b| } #=> Array or Enumerator`
Passes its arguments to both `Enumerable#min_by` and `Enumerable#max_by`, and returns their values. Returns an `Enumerator` if the block is omitted.

`Enumerable#none?() {|object| } #=> true or false`
Passes each element of the receiver to the block, returning `true` if the block is never `true`; `false`, otherwise. If the block is omitted, returns `true` if every element is either `false` or `nil`; `false`, otherwise.

`Enumerable#one?() {|object| } #=> true or false`
Passes each element of the receiver to the block, returning `true` if the block is `true` exactly once; `false`, otherwise. If the block is omitted, returns `true` if exactly one element is neither `false` nor `nil`; `false`, otherwise.

`Enumerable#parition() {|object| } #=> Array or Enumerator`
Returns an `Array` whose first element is an `Array` of elements for which the block was `true`; and last element is an `Array` of the remainder.

`Enumerable#reduce(initial) {|accumulator, element| } #=> Object`
`Enumerable#reduce(initial, selector) #=> Object`
`Enumerable#reduce(selector) #=> Object`
Aliases `Enumerable#inject`.

`Enumerable#reject() {|object| } #=> Array or Enumerator`
Returns the elements for which the block is `false`, or an `Enumerator` if the block is omitted.

`Enumerable#reverse_each() {|object| } #=> Array or Enumerator`
Yields each element in reverse order, or returns an `Enumerator` if the block is omitted.

`Enumerable#select() {|object| } #=> Array or Enumerator`
Aliases `Enumerable#find_all`.

`Enumerable#slice_before(pattern) #=> Enumerator`
`Enumerable#slice_before() {|object| }) {|object, state| } #=> Enumerator`
`Enumerable#slice_before(initial_state) {|object, state| } #=> Enumerator`

Groups elements such that an element is a member of its predecessor's group unless the given condition is `true`, in which case it's a member of a new group. The first element is a member of the default group. In the first form, the condition is *pattern* having case-equality—i.e. `#===`—with the element; in the other forms, it is specified by the value of the block when passed the element. If *initial_state* is given, it can be used for maintaining state: it's duplicated for each iteration and passed to the block as the second argument.

`Enumerable#sort() {|a, b| } #=> Array`

Returns the elements sorted either by passing each pair to the block and expecting a return value accordant with that of `#<=>`, or comparing elements with their `#<=>` methods.

`Enumerable#sort_by() {|object| } #=> Array or Enumerator`

Maps each element through the block then sorts them on the value returned, or returns an `Enumerator` if the block is omitted.

`Enumerable#take(n) #=> Array`

Returns the first *n* elements.

`Enumerable#take_while() {|object| } #=> Array or Enumerator`

Collects elements until the block is `false`, then returns them. Returns an `Enumerator` if the block is omitted.

`Enumerable#to_a(argument, …) #=> Array`

Aliases `Enumerable#entries`.

`Enumerable#zip(object, …) {|array| } #=> Array or nil`

Creates an `Array` for each element, containing the element along with the corresponding element from each of its `Enumerable` arguments. If fewer arguments are given than there are elements in the receiver, the result `Arrays` are padded with `nils`. If a block is given, each result `Array` is yielded to it; otherwise they are returned as an `Array` of `Arrays`.

ENUMERATOR

`Enumerator.new(object, *selector=:each*, *argument*, …) #=> Enumerator`
`Enumerator.new() {|yielder| } #=> Enumerator`
The first form returns an `Enumerator` for *object* in terms of its method named *selector*; any arguments given are passed to this method. If a block is given instead, it is passed a new `Enumerator::Yielder` object which may be used to lazily append values to this `Enumerator`.

`Enumerator#each() {|*item| } #=> Object or Enumerator`
Yields each element of this enumeration—passing to the block as many parameters as the `Enumerator` supplied. Returns `self`, or an `Enumerator` if the block is omitted.

`Enumerator#each_with_index() {|(*item), index| } #=> Object or Enumerator`
Yields each element of this enumeration—passing to the block as many parameters as the `Enumerator` supplied—along with the corresponding index. Returns `self`, or an `Enumerator` if the block is omitted.

`Enumerator#each_with_object(object) {|(*item), object| } #=> Object or Enumerator`
Yields each element of this enumeration—passing to the block as many parameters as the `Enumerator` supplied—along with *object*. Returns *object*, or an `Enumerator` if the block is omitted. Aliased by `Enumerator#with_object`.

`Enumerator#feed(object) #=> nil`
Returns *object* to the receiver the next time it yields a value.

`Enumerator#next() #=> Object`
Returns the next element of this `Enumerator`, or raises `StopIteration` if the last has already been returned.

`Enumerator#next_values() #=> Array`
Returns the next element of this `Enumerator` as an `Array`, or raises

`StopIteration` if the last has already been returned. Returns `[]` if the receiver used a bare `yield`; or `[nil]` if it used `yield nil`.

`Enumerator#peek() #=> Object`
Returns the next element of this `Enumerator` without advancing the next-element pointer, or raises `StopIteration` if the last has already been returned.

`Enumerator#peek_values() #=> Array`
Returns the next element of this `Enumerator` as an `Array` without advancing the next-element pointer, or raises `StopIteration` if the last has already been returned. Returns `[]` if the receiver used a bare `yield`; or `[nil]` if it used `yield nil`.

`Enumerator#rewind() #=> Object`
Resets the `Enumerator` such that a subsequent call to `Enumerator#next` returns the first element again. If the enumerated object has a `#rewind` method, it is called. Returns `self`.

`Enumerator#with_index(`*`offset=0`*`) {|(*item), index| } #=> Object or Enumerator`
Behaves as `Enumerator#each_with_index`, except *offset* is added to each index.

`Enumerator#with_object(object) {|(*item), object| } #=> Object or Enumerator`
Aliases `Enumerator#each_with_object`.

EXCEPTION

`Exception.exception(`*`message='Exception'`*`) #=> Exception`
Returns a new `Exception` with the given message.

`Exception.new(`*`message='Exception'`*`) #=> Exception`
Returns a new `Exception` with the given message.

`Exception#backtrace() #=> Array`
Returns the backtrace as an `Array` of `Strings`.

`Exception#exception(`*`message`*`) #=> Exception`
Returns the receiver if no argument is given; otherwise, creates a new instance of the receiver's class with the given message.

`Exception#message() #=> Object`
Returns this `Exception`'s message.

`Exception#set_backtrace(array) #=> Array`
Sets this `Exception`'s backtrace to the given `Array` of `Strings`.

`Exception#to_s() #=> Object`
Returns the message or, if that's not set, the class name of this `Exception`.

FALSECLASS

```
FalseClass#&(object) #=> false
```
Performs a logical AND with the given argument.

```
FalseClass#^(object) #=> true or false
```
Performs exclusive OR: returns `false` if *object* is `nil` or `false`; `true`, otherwise.

```
FalseClass#|(object) #=> true or false
```
Performs logical OR: returns `false` if *object* is `nil` or `false`; `true`, otherwise.

FIBER

`Fiber.new() { } #=> Fiber`
Returns a new `Fiber` whose body is the given block.

`Fiber.yield(`*argument*`, …) #=> Object`
Suspends the current `Fiber`, returning control, and any *argument*(s), to where this `Fiber` was resumed from. Returns the arguments of the corresponding `Fiber#resume` call. The root `Fiber` cannot be yielded from.

`Fiber#resume(`*argument*`, …) #=> Object`
Resumes this `Fiber`. If there was a corresponding `Fiber.yield`, its arguments become this method's return value, and this method's arguments become `Fiber.yield`'s return value.

FILE

`File.absolute_path(filename, directory=Dir.pwd) #=> String`
Returns *filename* as an absolute path name relative to *directory*.

`File.atime(filename) #=> Time`
Returns the last access time for *filename*, or the epoch if has never been accessed.

`File.basename(filename, extension='') #=> String`
Returns the last component of *filename* with *extension* removed from the end. An *extension* of `.*` removes any extension present.

`File.blockdev?(filename) #=> true or false`
Returns `true` if *filename* is a block device; `false` if it isn't or this operating system does not support such devices.

`File.chardev?(filename) #=> true or false`
Returns `true` if *filename* is a character device; `false` if it isn't or this operating system does not support such devices.

`File.chmod(permission, filename, …) #=> Integer`
Sets the permission bits of each named file to the `Integer` *permission*, returning the number of files processed. On Unix-like systems, *permissions* is the bit-mask documented in `chmod(2)`.

`File.chown(owner, group, filename, …) #=> Integer`
Sets the owner and group of each named file to the `Integers` *owner* and *group*, respectively, returning the number of files processed. If either *owner* or *group* is `nil` or `-1`, it is ignored.

`File.ctime(filename) #=> Time`
Returns the time of last status change—i.e. the inode change time on Unix-like systems—for *filename*.

`File.delete(filename, …) #=> Integer`
Deletes each named file, returning the number deleted. Aliased by `File.unlink`.

`File.directory?(filename) #=> true or false`
Returns `true` if *filename* is a directory; otherwise, `false`.

`File.dirname(filename) #=> String`
Returns *filename* with the last component removed.

`File.executable?(filename) #=> true or false`
Returns `true` if *filename* is executable by the effective owner of the current process; otherwise, `false`.

`File.executable_real?(filename) #=> true or false`
Returns `true` if *filename* is executable by the real owner of the current process; otherwise, `false`.

`File.exist?(filename) #=> true or false`
Returns `true` if *filename* exists; otherwise, `false`. Aliased by `File.exists?`.

`File.exists?(filename) #=> true or false`
Aliases `File.exist?`.

`File.expand_path(filename, directory=Dir.pwd) #=> String`
Returns *filename* as an absolute path relative to *directory*. However, *directory* is ignored if *filename* begins with a tilde: if the tilde is followed by a user name, *filename* is expanded relative to that user's home directory; otherwise, *filename* is expanded relative to the current user's home directory.

`File.extname(filename) #=> String`
Returns the portion of the filename following the right-most full stop, including the full stop itself, or `""` if there is no full stop.

`File.file?(filename) #=> true or false`
Returns `true` if file is a regular—as opposed to a device, directory, pipe, or socket—file; otherwise, `false`.

`File.fnmatch(pattern, filename, flags) #=> true or false`
Returns `true` if *filename* matches the globbing pattern *pattern*; otherwise, `false`. The globbing syntax and permissible *flag* values are explained in Globbing. Aliased by `File.fnmatch?`.

`File.fnmatch?(pattern, filename, flags) #=> true or false`
Aliases `File.fnmatch`.

`File.ftype(filename) #=> String`
Returns the file type as one of the following `String`s: `"blockSpecial"`, `"characterSpecial"`, `"directory"`, `"fifo"`, `"link"`, `"socket"`, or `"unknown"`.

`File.grpowned?(filename) #=> true or false`
Returns `true` if the effective group ID of the current process is equal to the group ID of *filename*; otherwise, or on Windows, `false`.

`File.identical?(filename1, filename2) #=> true or false`
Returns `true` if *filename1* and *filename2* resolve to the same file; otherwise, `false`.

`File.join(component, …) #=> String`
Concatenates the given path components with `File::SEPARATOR`, returning the result.

`File.lchmod(permission, filename, …) #=> Integer`
Behaves as `File.chmod` except that if a *filename* is a symbolic link, it changes the permission of the link; not its target. Raises a `NotImplementedError` on platforms lacking the `lchmod(2)` system call.

`File.lchown(owner, group, filename, …) #=> Integer`
Behaves as `File.chown` except that if a *filename* is a symbolic link, it changes the owner and group of the link; not its target. Raises a `NotImplementedError` on platforms lacking the `lchown(2)` system call.

`File.link(filename, link) #=> 0`
Creates a hard link named *link* to an existing file named *filename*. If *link* exists prior to this method being invoked, an `Errno:: Exception` will be raised.

`File.lstat(filename) #=> File::Stat`
Behaves as `IO#stat` except that if *filename* is a symbolic link, it returns status information for the link; not its target.

`File.mtime(filename) #=> Time`
Returns the time of last modification—on Unix-like systems this equates to the file's contents being modified, or, if *filename* is a directory, the creation or deletion of files in that directory—for *filename*.

`File.new(filename, mode='r', permissions, options) #=> File`
`File.new(file_descriptor, mode='r', options) #=> File`
The first form opens a file named *filename*, which it returns as a `File` object. The *mode* may be either a given as a mode string or a logical OR of the file open flags. The permissions of the file are given by the `Integer` *permissions*, the meaning of which is platform dependent. An `IO` options `Hash` may supplied as *options*. Alternatively, a file descriptor may be given as the first argument in which case the arguments are passed to `IO.new` to instantiate a `File` object for the existing stream.

`File.owned?(filename) #=> true or false`
Returns `true` if the file named *filename* is owned by the effective user ID of the current process; otherwise, `false`.

`File.path(filename) #=> String or nil`
Returns this file's pat by invoking `#to_path`; if that method isn't defined, `nil` is returned instead.

`File.pipe?(filename) #=> true or false`
Returns `true` if the file named *filename* is a pipe; otherwise, or if the operating system doesn't support named pipes, `false`.

`File.readable?(filename) #=> true or false`
Returns `true` if the file named *filename* is readable by the effective user ID of the current process; otherwise, `false`.

`File.readable_real?(filename) #=> true or false`
Returns `true` if the file named *filename* is readable by the real user ID of the current process; otherwise, `false`.

`File.readlink(filename) #=> String`
Returns the target of the symbolic link named *filename*. Raises `NotImplementedError` if the operating system lacks the `readlink(2)` system call.

`File.realdirpath(filename, directory=Dir.pwd) #=> String`
Behaves as `File.realpath` but allows the last component to be nonexistent.

`File.realpath(filename, directory=Dir.pwd) #=> String`
Returns the canonical absolute pathname for the given path: all symbolic links are expanded, references to `/./` and `/../` are resolved relative to *directory*, and superfluous slashes are removed. All path components must exist.

`File.rename(filename, new) #=> 0`
Renames the file or directory named *filename* to *new*.

`File.setgid?(filename) #=> true or false`
Returns `true` if the file named *filename* has its set-group-ID bit set; otherwise, or if the operating system doesn't support *setgid* bits, `false`.

`File.setuid?(filename) #=> true or false`
Returns `true` if the file named *filename* has its set-user-ID bit set; otherwise, or if the operating system doesn't support *suid* bits, `false`.

`File.size(filename) #=> Integer`
Returns the size of the file named *filename* in bytes.

`File.size?(filename) #=> Integer or nil`
Returns `nil` if the file named *filename* has a size of `0`; otherwise, the size in bytes.

`File.socket?(filename) #=> true or false`
Returns `true` if the file named *filename* is a socket; otherwise, or if the operating system doesn't support sockets, `false`.

`File.split(filename) #=> Array`
Returns the directory name—i.e. `File.dirname`—and basename—i.e. File.basename—of the file named *file* as a two-element `Array`.

`File.stat(filename) #=> File::Stat`
Returns status information for the file named *filename*.

`File.sticky?(filename) #=> true or false`
Returns `true` if the file named *filename* has its sticky bit set; otherwise, or if the operating system doesn't support sticky bits, `false`.

`File.symlink(filename, new) #=> 0`
Creates a symbolic link from an existing file named *filename* to *new*. Raises `NotImplementedError` on operating systems that lack the `symlink(2)` system call.

`File.symlink?(filename) #=> true or false`
Returns `true` if there is a symbolic link named *filename*; otherwise, or if the operating system doesn't support symbolic links, `false`.

`File.truncate(filename, length) #=> 0`
Truncates the file named *filename* to *length* bytes in length. *length* may be greater or less than the file's current size; if greater, the file is extended with null (`"\0"`) bytes. Raises `NotImplementedError` on operating systems lacking the `truncate(2)` system call.

`File.umask(`*mask*`) #=> Integer`
Returns the file mode creation mask—*umask*—of the calling process. If an `Integer` *mask* is given, the umask is set to *mask* `& 0777`. The umask modifies the permissions of newly created files and directories by turning off the bits in the access mode that are on in the umask.

`File.unlink(filename, …) #=> Integer`
Aliases `File.delete`.

`File.utime(atime, mtime, filename, …) #=> Integer`
Sets the access times and last modification times for each named file to *atime* and *mtime*, respectively. The times may be given as `Time` objects or `Integer` seconds since the epoch. Returns the number of files changed. Raises `NotImplementedError` on non-Windows systems which lack both the `utimes(2)` and `utimensat(2)` system calls.

```
File.world_readable?(filename) #=> Integer or nil
```
Returns the permission bits of the file named *filename* if it is world readable; otherwise, `nil`.

```
File.world_writable?(filename) #=> Integer or nil
```
Returns the permission bits of the file named *filename* if it is world writable; otherwise, `nil`.

```
File.writable?(filename) #=> true or false
```
Returns `true` if the file named *filename* is writable by the effective user ID of the current process; otherwise, `false`.

```
File.writable_real?(filename) #=> true or false
```
Returns `true` if the file named *filename* is writable by the real user ID of the current process; otherwise, `false`.

```
File.zero?(filename) #=> true or false
```
Returns `true` if the file named *filename* has a size of `0`; otherwise, `false`.

FILE::STAT

`File::Stat#<=>(stat) #=> -1, 0, 1`
Compares the modification time of this file with that of the given `File::Stat` object, returning `-1` if the receiver is older, `0` if they are equal, and `1` if the receiver is younger.

`File::Stat#atime() #=> Time`
Returns the last access time for this file, or the epoch if has never been accessed.

`File::Stat#blksize() #=> Integer`
Returns the block size of this file's filesystem, or `nil` if the filesystem doesn't support this attribute.

`File::Stat#blockdev?() #=> true or false`
Returns `true` if this file is a block device; `false` if it isn't or this operating system does not support such devices.

`File::Stat#blocks() #=> Integer or nil`
Returns the number of file system blocks allocated for this file, or `nil` if the filesystem doesn't support this attribute.

`File::Stat#chardev?() #=> true or false`
Returns `true` if this file is a character device; `false` if it isn't or this operating system does not support such devices.

`File::Stat#ctime() #=> Time`
Returns the time of last status change—i.e. the inode change time on Unix-like systems—for this file.

`File::Stat#dev() #=> Integer`
Returns the device ID for this file.

`File::Stat#dev_major() #=> Integer or nil`
Returns the major part of `File::Stat#dev`; otherwise, or if the operating system doesn't support this attribute, `nil`.

`File::Stat#dev_minor() #=> Integer or nil`
Returns the minor part of `File::Stat#dev` on supported operating systems; otherwise, or if the operating system doesn't support this attribute, `nil`.

`File::Stat#directory?() #=> true or false`
Returns `true` if this file a directory; otherwise, `false`.

`File::Stat#executable?() #=> true or false`
Returns `true` if this file is executable by the effective owner of the current process; otherwise, or if the operating system does not have the concept of an *executable* file, `false`.

`File::Stat#executable_real?() #=> true or false`
Returns `true` if this file is executable by the real owner of the current process; otherwise, or if the operating system does not have the concept of an *executable* file, `false`.

`File::Stat#file?() #=> true or false`
Returns `true` if this file is a regular—as opposed to a device, directory, pipe, or socket—file; otherwise, `false`.

`File::Stat#ftype() #=> String`
Returns this file's type as one of the following `Strings`: `"blockSpecial"`, `"characterSpecial"`, `"directory"`, `"fifo"`, `"link"`, `"socket"`, or `"unknown"`.

`File::Stat#gid() #=> Integer`
Returns this group ID of this file's owner.

`File::Stat#grpowned?() #=> true or false`
Returns `true` if the effective group ID of the current process is equal to the group ID of this file; otherwise, or on Windows, `false`.

`File::Stat#ino() #=> Integer`
Returns the inode number for this file.

`File::Stat#mode() #=> Integer`
Returns the permission bits for this file.

`File::Stat#mtime() #=> Time`
Returns the time of last modification—on Unix-like systems this equates to the file's contents being modified, or, if this file is a directory, the creation or deletion of files in that directory—for this file.

`File::Stat#nlink() #=> Integer`
Returns the number of hard links to this file.

`File::Stat#owned?() #=> true or false`
Returns `true` if this file is owned by the effective user ID of the current process; otherwise, `false`.

`File::Stat#pipe?() #=> true or false`
Returns `true` if this file is a pipe; otherwise, or if the operating system doesn't support pipes, `false`.

`File::Stat#rdev() #=> Integer or nil`
Returns the device number that this special file represents, or `nil` if the operating system doesn't support this attribute.

`File::Stat#rdev_major() #=> Integer or nil`
Returns the major part of the device number that this special file represents, or `nil` if the operating system doesn't return this attribute.

`File::Stat#rdev_minor() #=> Integer or nil`
Returns the minor part of the device number that this special file represents, or `nil` if the operating system doesn't return this attribute.

`File::Stat#readable?() #=> true or false`
Returns `true` if this file is readable by the effective user ID of the current process; otherwise, `false`.

`File::Stat#readable_real?() #=> true or false`
Returns `true` if this file is readable by the real user ID of the current process; otherwise, `false`.

`File::Stat#setgid?() #=> true or false`
Returns `true` if this file has its set-group-ID bit set; otherwise, or if the operating system doesn't support setgid bits, `false`.

`File::Stat#setuid?() #=> true or false`
Returns `true` if this file has its set-user-ID bit set; otherwise, or if the operating system doesn't support setuid bits, `false`.

`File::Stat#size() #=> Integer`
Returns the size of this file in bytes.

`File::Stat#size?() #=> Integer or nil`
Returns `nil` if this file has a size of `0`; otherwise, the size in bytes.

`File::Stat#socket?() #=> true or false`
Returns `true` if this file is a socket; otherwise, or if the operating system doesn't support sockets, `false`.

`File::Stat#sticky?() #=> true or false`
Returns `true` if this file has its sticky bit set; otherwise, or if the operating system doesn't support sticky bits, `false`.

`File::Stat#symlink?() #=> true or false`
Returns `true` if this file is a symbolic link; otherwise, or if the operating system doesn't support symbolic links, `false`.

`File::Stat#uid() #=> Integer`
Returns the user ID of the file's owner.

`File::Stat#world_readable?() #=> Integer or nil`
Returns the permission bits of this file if it is world readable; otherwise, `nil`.

`File::Stat#world_writable?() #=> Integer or nil`
Returns the permission bits of this file if it is world writable; otherwise, `nil`.

`File::Stat#writable?() #=> true or false`
Returns `true` if this file is writable by the effective user ID of the current process; otherwise, `false`.

```
File::Stat#writable_real?() #=> true or false
```

Returns `true` if this file is writable by the real user ID of the current process; otherwise, `false`.

```
File::Stat#zero?() #=> true or false
```

Returns `true` if this file has a size of `0`; otherwise, `false`.

FILETEST

```
FileTest.blockdev?(filename) #=> true or false
```
Returns `true` if *filename* is a block device; `false` if it isn't or this operating system does not support such devices.

```
FileTest.chardev?(filename) #=> true or false
```
Returns `true` if *filename* is a character device; `false` if it isn't or this operating system does not support such devices.

```
FileTest.directory?(filename) #=> true or false
```
Returns `true` if *filename* is a directory; otherwise, `false`.

```
FileTest.executable?(filename) #=> true or false
```
Returns `true` if *filename* is executable by the effective owner of the current process; otherwise, `false`.

```
FileTest.executable_real?(filename) #=> true or false
```
Returns `true` if *filename* is executable by the real owner of the current process; otherwise, `false`.

```
FileTest.exist?(filename) #=> true or false
```
Returns `true` if *filename* exists; otherwise, `false`. Aliased by `FileTest.exists?`.

```
FileTest.exists?(filename) #=> true or false
```
Aliases `FileTest.exist?`.

```
FileTest.file?(filename) #=> true or false
```
Returns `true` if file is a regular—as opposed to a device, directory, pipe, or socket—file; otherwise, `false`.

```
FileTest.grpowned?(filename) #=> true or false
```
Returns `true` if the effective group ID of the current process is equal to the group ID of *filename*; otherwise, or on Windows, `false`.

```
FileTest.identical?(filename1, filename2) #=> true or false
```
Returns `true` if *filename1* and *filename2* resolve to the same file; otherwise, `false`.

```
FileTest.owned?(filename) #=> true or false
```
Returns `true` if the file named *filename* is owned by the effective user ID of the current process; otherwise, `false`.

```
FileTest.pipe?(filename) #=> true or false
```
Returns `true` if the file named *filename* is a pipe; otherwise, or if the operating system doesn't support named pipes, `false`.

```
FileTest.readable?(filename) #=> true or false
```
Returns `true` if the file named *filename* is readable by the effective user ID of the current process; otherwise, `false`.

```
FileTest.readable_real?(filename) #=> true or false
```
Returns `true` if the file named *filename* is readable by the real user ID of the current process; otherwise, `false`.

```
FileTest.setgid?(filename) #=> true or false
```
Returns `true` if the file named *filename* has its set-group-ID bit set; otherwise, or if the operating system doesn't support *setgid* bits, `false`.

```
FileTest.setuid?(filename) #=> true or false
```
Returns `true` if the file named *filename* has its set-user-ID bit set; otherwise, or if the operating system doesn't support *suid* bits, `false`.

```
FileTest.size(filename) #=> Integer
```
Returns the size of the file named *filename* in bytes.

```
FileTest.size?(filename) #=> Integer or nil
```
Returns `nil` if the file named *filename* has a size of `0`; otherwise, the size in bytes.

```
FileTest.socket?(filename) #=> true or false
```
Returns `true` if the file named *filename* is a socket; otherwise, or if the operating system doesn't support sockets, `false`.

```
FileTest.sticky?(filename) #=> true or false
```
Returns `true` if the file named *filename* has its sticky bit set; otherwise, or if the operating system doesn't support sticky bits, `false`.

```
FileTest.symlink?(filename) #=> true or false
```
Returns `true` if there is a symbolic link named *filename*; otherwise, or if the operating system doesn't support symbolic links, `false`.

```
FileTest.world_readable?(filename) #=> Integer or nil
```
Returns the permission bits of the file named *filename* if it is world readable; otherwise, `nil`.

```
FileTest.writable?(filename) #=> true or false
```
Returns `true` if the file named *filename* is writable by the effective user ID of the current process; otherwise, `false`.

```
FileTest.writable_real?(filename) #=> true or false
```
Returns `true` if the file named *filename* is writable by the real user ID of the current process; otherwise, `false`.

```
FileTest.zero?(filename) #=> true or false
```
Returns `true` if the file named *filename* has a size of `0`; otherwise, `false`.

```
FileTest.zero?(filename) #=> true or false
```
Returns `true` if the file named *filename* has a size of `0`; otherwise, `false`.

FIXNUM

`Fixnum#%(number) #=> Fixnum`
Returns the result of the receiver modulo *number*. Aliased by `Fixnum#modulo`.

`Fixnum#&(number) #=> Integer`
Returns the result of a bitwise AND between the receiver and *number*.

`Fixnum#*(number) #=> Numeric`
Returns the result of multiplying *number* with the receiver.

`Fixnum#**(number) #=> Numeric`
Returns the result of raising the receiver to the $number^{th}$ power.

`Fixnum#+(number) #=> Numeric`
Returns the result of adding the receiver to *number*.

`Fixnum#-(number) #=> Numeric`
Returns the result of subtracting *number* from the receiver.

`Fixnum#-@() #=> Fixnum`
Returns the receiver with a negative sign.

`Fixnum#/(number) #=> Integer`
Returns the result of dividing—using integer division—the receiver by *number*. Aliased by `Fixnum#div`.

`Fixnum#<(number) #=> true or false`
Returns `true` if the receiver is less than *number*; otherwise, `false`.

`Fixnum#<<(number) #=> Integer`
Returns the result of left-shifting *number* bits of the receiver.

`Fixnum#<=(number) #=> true or false`
Returns `true` if the receiver is less than or equal to *number*; otherwise, `false`.

`Fixnum#<=>(number) #=> -1, 0, 1`
Returns `-1` if the receiver is less than *number*, `0` if they are equal, and `1` if it is greater.

`Fixnum#==(number) #=> true or false`
Returns `true` if the *number* is a `Numeric` with the same value as the receiver; `false`, otherwise. Aliased by `Fixnum#===`.

`Fixnum#===(number) #=> true or false`
Aliases `Fixnum#==`.

`Fixnum#>(number) #=> true or false`
Returns `true` if the receiver is greater than *number*; otherwise, `false`.

`Fixnum#>=(number) #=> true or false`
Returns `true` if the receiver is greater than or equal to *number*; otherwise, `false`.

`Fixnum#>>(number) #=> Integer`
Returns the result of right-shifting *number* bits of the receiver and its sign.

`Fixnum#[](bit) #=> 0 or 1`
Returns the bit^{th} bit of the receiver, where the 0^{th} bit is the least significant.

`Fixnum#^(number) #=> Integer`
Returns the result of a bitwise EXCLUSIVE OR between the receiver and *number*.

`Fixnum#abs() #=> Fixnum`
Returns the absolute value of the receiver. Aliased by `Fixnum#magnitude`.

`Fixnum#div(number) #=> Fixnum`
Aliases `Fixnum#/`.

`Fixnum#divmod(number) #=> Array`
Divides the receiver by *number*, returning an `Array` whose first element is the quotient, and last element, the modulus. The quotient is rounded toward $-\infty$.

`Fixnum#even?() #=> true or false`
Returns `true` if this number is even; otherwise, `false`.

`Fixnum#fdiv(number) #=> Float`
Returns the result of dividing—using floating-point division—the receiver by *number*. Aliased by `Fixnum#quo`.

`Fixnum#magnitude() #=> Fixnum`
Aliases `Fixnum#abs`.

`Fixnum#modulo(number) #=> Numeric`
Aliases `Fixnum#%`.

`Fixnum#odd?() #=> true or false`
Returns `true` if this number is odd; otherwise, `false`.

`Fixnum#size() #=> Fixnum`
Returns the number of bytes used to represent the receiver.

`Fixnum#succ() #=> Integer`
Returns the receiver incremented by `1`.

`Fixnum#to_f() #=> Float`
Converts the receiver to a `Float`.

`Fixnum#to_s(base=10) #=> String`
Returns a `String` representation of the receiver in the given base, where *base* is between `2` and `36` inclusive.

`Fixnum#zero?() #=> true or false`
Returns `true` if the receiver is equal to zero; otherwise, `false`.

`Fixnum#|(number) #=> Integer`
Returns the result of a bitwise OR between the receiver and *number*.

`Fixnum#~() #=> Integer`
Returns the result of inverting the receiver's bits.

FLOAT

`Float#%(number) #=> Float`
Returns the result of the receiver modulo *number*. Aliased by `Float#modulo`.

`Float#*(number) #=> Float`
Returns the result of multiplying *number* with the receiver.

`Float#**(number) #=> Float`
Returns the result of raising the receiver to the $number^{th}$ power.

`Float#+(number) #=> Float`
Returns the result of adding the receiver to *number*.

`Float#-(number) #=> Float`
Returns the result of subtracting *number* from the receiver.

`Float#-@() #=> Float`
Returns the receiver with a negative sign.

`Float#/(number) #=> Integer`
Returns the result of dividing—using integer division—the receiver by *number*. Aliased by `Float#div`.

`Float#<(number) #=> true or false`
Returns `true` if the receiver is less than *number*; otherwise, `false`.

`Float#<=(number) #=> true or false`
Returns `true` if the receiver is less than or equal to *number*; otherwise, `false`.

`Float#<=>(number) #=> -1, 0, 1`
Returns `-1` if the receiver is less than *number*, `0` if they are equal, and `1` if it is greater.

`Float#==(number) #=> true or false`
Returns `true` if the *number* is a `Numeric` with the same value as the receiver; `false`, otherwise. Aliased by `Float#===`.

`Float#===(number) #=> true or false`
Aliases `Float#==`.

`Float#>(number) #=> true or false`
Returns `true` if the receiver is greater than *number*; otherwise, `false`.

`Float#>=(number) #=> true or false`
Returns `true` if the receiver is greater than or equal to *number*; otherwise, `false`.

`Float#abs() #=> Float`
Returns the absolute value of the receiver. Aliased by `Float#magnitude`.

`Float#angle() #=> Float`
Returns `0` if the receiver is positive; otherwise, `Math::PI`. Aliased by `Float#arg` and `Float#phase`.

`Float#arg() #=> Float`
Aliases `Float#angle`.

`Float#ceil() #=> Integer`
Returns the smallest `Integer` greater than or equal to the receiver.

`Float#coerce(number) #=> Array`
Returns an `Array` whose first element is *number* converted to a `Float` with `Kernel.Float`, and last element is the receiver.

`Float#denominator() #=> Integer`
Converts the receiver to a `Rational`, returning `Rational#denominator`. `Float::INFINITY` and `Float::NAN` both have a denominator of `1`.

`Float#divmod(number) #=> Array`
Divides the receiver by *number*, returning an `Array` whose first element is the quotient, and last element, the modulus. The quotient is rounded toward $-\infty$.

`Float#eql?(object) #=> true or false`
Returns `true` if the receiver is a `Float` with the same value as *object*; otherwise, `false`.

`Float#fdiv(number) #=> Float`
Returns the result of dividing—using floating-point division—the receiver by *number*. Aliased by `Float#quo`.

`Float#finite?() #=> true or false`
Returns `true` if the receiver is neither `Float::INFINITY` nor `Float::NAN`; otherwise, `false`.

`Float#floor() #=> Integer`
Returns the largest `Integer` less than or equal to the receiver.

`Float#infinite?() #=> nil, -1, or 1`
Returns `nil` if the receiver is neither `Float::INFINITY` or `Float::NAN`; `-1` if it is `-Float::INFINITY`; and `1` if it is `Float::INFINITY`.

`Float#magnitude() #=> Float`
Aliases `Float#abs`.

`Float#modulo(number) #=> Numeric`
Aliases `Float#%`.

`Float#nan?() #=> true or false`
Returns `true` if the receiver is `Float::NAN`; otherwise, `false`.

`Float#numerator() #=> Integer`
Converts the receiver to a `Rational`, returning `Rational#numerator`. `Float::INFINITY` and `Float::NAN` both have a numerator of their self.

`Float#phase() #=> Float`
Aliases `Float#angle`.

`Float#quo(number) #=> Float`
Aliases `Float#fdiv`.

`Float#rationalize(`*`epsilon`*`) #=> Rational`
Returns the simplest rational number differing from the receiver by no more than *epsilon*—if *epsilon* is omitted, it is calculated automatically. To do so, it assumes that the receiver is accurate only to the precision of Ruby's floating-point representation; as opposed to `Float#to_r`, say, which assumes the receiver is perfectly accurate.

`Float#round(`*`digits=0`*`) #=> Numeric`
Returns the receiver rounded to *digits* digits. If *digits* is `0`, the number is rounded to the nearest `Integer`.

`Float#to_f() #=> Float`
Returns the receiver.

`Float#to_i() #=> Integer`
Converts the receiver to an `Integer` by truncation, i.e. removing the fractional part. Aliased to `Float#to_int` and `Float#truncate`.

`Float#to_int() #=> Integer`
Aliases `Float#to_i`.

`Float#to_r() #=> Rational`
Converts the receiver to a `Rational` precisely.

`Float#to_s() #=> String`
Returns the receiver in fixed or exponential form, according to its magnitude. Returns `"Infinity"` for `Float::INFINITY`, `"-Infinity"` for `-Float::INFINITY`, and `"NaN"` for `Float::NAN`.

`Float#truncate() #=> Integer`
Aliases `Float#to_i`.

`Float#zero?() #=> true or false`
Returns `true` if the receiver is equal to `0.0`; otherwise, `false`.

GC

`GC.count() #=> Integer`
Returns the number of times the garbage collector has run in the current process.

`GC.disable() #=> true or false`
Disables garbage collection, returning `true` if garbage collection was already disabled; otherwise, `false`.

`GC.enable() #=> true or false`
Enables garbage collection, returning `true` if garbage collection was disabled; otherwise, `false`.

`GC.start() #=> nil`
Starts the garbage collector unless it has been explicitly disabled. Aliased to `GC#garbage_collect`.

`GC.stress() #=> true or false`
Returns `true` if the *stress* flag—see `GC.stress=` for details—is set; otherwise, `false`.

`GC.stress=(boolean) #=> true or false`
Sets the *stress* flag to *boolean*—which may be either `true` or `false`. In the first case, the garbage collector will run after every object allocation; in the second, and default, case the garbage collector will run as often as necessary.

`GC#garbage_collect() #=> nil`
Aliases `GC.start`.

HASH

`Hash[](object, …) #=> Hash`
Creates and returns a `Hash` such that the first two arguments comprise the first key-value pair, the second two arguments comprise the second key-value pair, and so forth—accordingly, there must be an even number of arguments given.

`Hash.new(`*object*`) #=> Hash`
`Hash.new() {|hash, key| } #=> Hash`
Creates and returns a `Hash` whose default value is `nil` if neither block nor argument are given, *object*, or the block. In the last case, the block is called every time a nonexistent key is requested, passing in the receiver and the given key.

`Hash.try_convert(object) #=> Hash or nil`
Converts *object* to a `Hash` with `#to_hash`, returning the result; if this is impossible, returns `nil`.

`Hash#==(object) #=> true or false`
Returns `true` if both the receiver and *object* have the same default value, the same number of keys, and the value of every key in the receiver is equal—according to `#==`—the value of the corresponding key in *object*. If *object* is not a `Hash`, it is converted using `#to_hash`, then the test repeated. In either the test or the conversion fails, `false` is returned.

`Hash#[](key) #=> Object`
Returns the value corresponding to the given key, or the default value if the key doesn't exist.

`Hash#[]=(key, value) #=> Object`
Sets the given key to the given value, returning the latter. Aliased by `Hash#store`.

`Hash#assoc(key) #=> Array or nil`
Returns an `Array` whose first element is *key*, and last element is the value

corresponding to that key. If *key* does not exist, the default value is ignored, and `nil` returned.

`Hash#clear() #=> Hash`
Returns the receiver with all of its key-value pairs removed.

`Hash#compare_by_identity() #=> Hash`
Converts the receiver to an identity `Hash`, which it returns.

`Hash#compare_by_identity?() #=> true or false`
Returns `true` if the receiver is an identity `Hash`; otherwise, `false`.

`Hash#default() #=> Object`
Returns the receiver's default value.

`Hash#default=(object) #=> Object`
Sets the default value to *object*, which it then returns.

`Hash#default_proc() #=> Proc or nil`
Returns the default `Proc`, or `nil` if there's not one.

`Hash#default_proc=(proc) #=> Proc`
Sets the default `Proc` to *proc*, which it then returns.

`Hash#delete(key) {|key| } #=> Object`
Deletes from the receiver the key, *key*, and its associated value, returning the value. If the block is given and *key* did not exist, the block is called with the key as an argument and its result returned; otherwise, `nil` is returned.

`Hash#delete_if() {|key, value| } #=> Hash or Enumerator`
Deletes each key-value pair for which the block is `true`, then returns the receiver. Returns an `Enumerator` if the block is omitted.

`Hash#each() {|key, value| } #=> Hash or Enumerator`
Yields each key-value pair to the block. Returns an `Enumerator` if the block is omitted. Aliased by `Hash#each_pair`.

`Hash#each_key() {|key| } #=> Hash or Enumerator`
Yields each key to the block. Returns an `Enumerator` if the block is omitted.

`Hash#each_pair() {|key, value| } #=> Hash or Enumerator`
Aliases `Hash#each`.

`Hash#each_value() {|value| } #=> Hash or Enumerator`
Yields each value to the block. Returns an `Enumerator` if the block is omitted.

`Hash#empty?() #=> true or false`
Returns `true` if the receiver contains no key-value pairs; otherwise, `false`.

`Hash#fetch(key, default) #=> Object`
`Hash#fetch(key) {|key| } #=> Object`
Returns the value associated with the key *key*. If the given key does not exist, and no other arguments are given, an `KeyError` is raised; if *default* is given, it will be returned; otherwise, the block is called with the key as a parameter, and its value returned. All forms ignore any default values.

`Hash#flatten(depth=1) #=> Array`
Returns the receiver converted to an `Array` then flattened with `Array#flatten(depth)`.

`Hash#has_key?(key) #=> true or false`
Returns `true` if the given key exists in the receiver; otherwise, `false`. Aliased by `Hash#include?` and `Hash#member?`.

`Hash#has_value?(value) #=> true or false`
Returns `true` if a key exists in the receiver with the given value; otherwise, `false`. Aliased by `Hash#value?`.

`Hash#include?(key) #=> true or false`
Aliases `Hash#has_key?`.

`Hash#index(key) #=> Object`
Deprecated; use the identical `Hash#key` instead.

`Hash#invert() #=> Hash`
Returns a new `Hash` whose keys are the receiver's values, and values, the receiver's keys—if the receiver has keys with duplicate values, the results are unspecified.

`Hash#key(value) #=> Object or nil`
Returns the first key associated with the given value, or `nil` if no such key exists.

`Hash#keys() #=> Array`
Returns the keys of the receiver.

`Hash#length() #=> Integer`
Returns the number of key-value pairs in the receiver. Aliased by `Hash#size`.

`Hash#member?(key) #=> true or false`
Aliases `Hash#has_key?`.

`Hash#merge(hash) {|key, old_value, new_value| } #=> Hash`
Returns a new `Hash` containing the key-value pairs of the receiver plus those of the given `Hash`. If the same key exists in both `Hash`es, it's associated with the value from *hash*. Alternatively, if a block is given it is called with each duplicate key, along with its value in the receiver and its value in *hash*: its return value becomes the value of the duplicate key.

`Hash#merge!(hash) {|key, old_value, new_value| } #=> Hash`
Behaves as `Hash#merge`, but modifies the receiver in-place. Aliased by `Hash#update`.

`Hash#rassoc(value) #=> Array or nil`
Returns a two-element `Array` comprising the first key whose value is *value*, and *value*. If there is no key associated with the given value, `nil` returned.

`Hash#rehash() #=> Hash`
Re-creates the receiver using the current hash values for each key. If called while the receiver is being iterated over, an `IndexError` is raised.

`Hash#reject() {|key, value| } #=> Hash or Enumerator`
Deletes each key-value pair for which the block is `true` from a copy of the receiver, then returns this copy. Returns an `Enumerator` if the block is omitted.

`Hash#reject() {|key, value| } #=> Hash or Enumerator`
Deletes each key-value pair for which the block is `true` from a copy of the receiver, then returns this copy. Returns an `Enumerator` if the block is omitted.

`Hash#reject!() {|key, value| } #=> Hash, Enumerator, or nil`
Deletes each key-value pair for which the block is `true`, then returns the receiver. Returns `nil` if the block was never `true`, or an `Enumerator` if the block is omitted.

`Hash#replace(hash) #=> Hash`
Replaces all key-value pairs in the receiver with those from *hash*, then returns the receiver.

`Hash#select() {|key, value| } #=> Hash`
Returns a new `Hash` containing the key-value pairs of the receiver for which the block is `true`.

`Hash#shift() #=> Array or Object`
Deletes and returns the oldest key-value pair—i.e. the first pair to be returned in an iteration—in the receiver. Returns the default value if the receiver is empty.

`Hash#size() #=> Integer`
Aliases `Hash#length`.

`Hash#sort() {|a, b| } #=> Array`
Converts the receiver to an `Array` of [*key*, *value*] `Array`ss, on which it invokes `Array#sort` with the block—if given.

`Hash#store(key, value) #=> Object`
Aliases `Hash#[]=`.

`Hash#to_a() #=> Array`
Returns the receiver as an `Array` of [*key*, *value*] `Array`ss.

`Hash#to_hash() #=> Hash`
Returns the receiver.

`Hash#to_s() #=> String`
Returns the receiver in the form {key_0 => $value_0$, …, key_n => $value_n$}, with both both *key* and *value* substituted for their `#inspect` output. An empty `Hash` is returned as `"{}"`, and a recursive `Hash` as `"{...}"`.

`Hash#update(hash) {|key, old_value, new_value| } #=> Hash`
Aliases `Hash#merge!`.

`Hash#value?(value) #=> true or false`
Aliases `Hash#has_value?`.

`Hash#values() #=> Array`
Returns the values of every key.

`Hash#values_at(key, …) #=> Array`
Returns the values associated with each given key, or the default value if the key does not exist.

INTEGER

`Integer#ceil() #=> Integer`
Returns the receiver. Aliased by `Integer#floor`, `Integer#round`, `Integer#to_i`, and `Integer#to_int`, and `Integer#truncate`.

`Integer#chr(`*encoding*`) #=> String`
Interprets the receiver as a codepoint in the *encoding*, returning the corresponding character. If *encoding* is not given, it is the default internal encoding—if set—or *US-ASCII*.

`Integer#denominator() #=> 1`
Returns `1`.

`Integer#downto(until) {|n| } #=> Integer or Enumerator`
Yields each `Integer` from the receiver down to, and including, the given `Integer`. Returns the receiver, or an `Enumerator` if the block is omitted.

`Integer#even?() #=> true or false`
Returns `true` if this number is even; otherwise, `false`.

`Integer#floor() #=> Integer`
Aliases `Integer#ceil`.

`Integer#gcd(number) #=> Integer`
Returns the greatest divisor of the receiver and *number*.

`Integer#gcdlcm(number) #=> Array`
Returns an `Array` whose first element is `Integer#gcd`, and second element is `Integer#lcm`.

`Integer#integer?() #=> true`
Returns `true`.

`Integer#lcm(number) #=> Integer`
Returns the lowest common multiple of the receiver and *number*.

`Integer#next() #=> Integer`
Returns the receiver incremented by `1`. Aliased by `Integer#succ`.

`Integer#numerator() #=> Integer`
Returns the receiver.

`Integer#odd?() #=> true or false`
Returns `true` if this number is odd; otherwise, `false`.

`Integer#ord() #=> Integer`
Returns the receiver.

`Integer#pred() #=> Integer`
Returns the receiver decremented by `1`.

`Integer#round() #=> Integer`
Aliases `Integer#ceil`.

`Integer#succ() #=> Integer`
Aliases `Integer#next`.

`Integer#times() {|n| } #=> Integer or Enumerator`
Yields each `Integer` from `0` up to, but not including, the receiver. Returns the receiver, or an `Enumerator` if the block is omitted.

`Integer#to_i() #=> Integer`
Aliases `Integer#ceil`.

`Integer#to_int() #=> Integer`
Aliases `Integer#ceil`.

`Integer#to_r() #=> Rational`
Returns a `Rational` whose numerator is the receiver, and whose denominator is one.

`Integer#truncate() #=> Integer`
Aliases `Integer#ceil`.

`Integer#upto(until) {|n| } #=> Integer or Enumerator`
Yields each `Integer` from the receiver up to, and including, the given `Integer`. Returns the receiver, or an `Enumerator` if the block is omitted.

`Integer#rationalize(epsilon) #=> Rational`
Converts the receiver to a `Rational`, ignoring its argument.

IO

`IO.binread(filename, *length*, *offset*) #=> String`
Opens the file named *filename*, reads from it in binary mode, then returns its contents as an ASCII-8BIT-encoded `String`. If *length* is given, a maximum of this many bytes are read; if *offset* is also given, reading starts from this byte.

`IO.copy_stream(source, destination, *length*, *offset*) #=> Integer`
Copies data from *source* to *destination*—both of which may be filenames or `IO` streams—returning the number of bytes copied. If *length* is given, a maximum of this many bytes are copied; if *offset* is also given, copying starts from this byte instead of the current file position of *from*.

`IO.for_fd(file_descriptor, mode) #=> IO`
Aliases `IO.new`.

`IO.foreach(filename, *separator=$/*, *limit*, *options*) {|line| } #=> nil`
Invokes the block with each line found in the file named *filename*. Lines are separated by *separator*, but if this value is `nil` the entire file is treated as a single `String`. If *limit* is given, it is the maximum number of characters to return for each line. If *options* is given it is an options `Hash` that may contain `:encoding`, `:mode`, and `:open_args` keys.

`IO.new(file_descriptor, mode) #=> IO`
Returns a new `IO` stream for the given file descriptor and access mode. Aliased to `IO.for_fd`.

`IO.open(argument, …) {|io| } #=> Object`
Instantiates an `IO` object by passing *argument*(s) to the class's constructor, then returning the `IO` object. If a block is given, the new `IO` object is passed to it as a parameter, then closed automatically when the block exits; the return value is that of the block.

`IO.pipe() #=> Array`
Creates a pipe, the ends of which it returns as an `Array` of `IO` objects. The first

element is the read end; the last element is the write end, which is in sync mode.

`IO.popen(command, mode="r") #=> Object`
Executes a command as a subprocess, opening a pipe to this subprocess's standard input and output streams, which it returns as an `IO` object. If *command* is a `String` it names a command in the user's path, and is subject to shell expansion. If it is a `"-"`, and the platform supports forking, the current process forks: an `IO` pipe connected to the child's standard input and output streams is returned to the parent; `nil` is returned to the child.

Otherwise, *command* is an `Array` of `Strings`, the first of which specifies the command name; the remainder, its arguments. The shell is bypassed, so none of these `Strings` are subject to shell expansion. If the first element of this `Array` is a `Hash`, it specifies the names and corresponding values of environment variables that should be set in the subprocess. An options `Hash` may be supplied as the last element of this `Array`.

If a block is supplied, Ruby's end of the pipe is passed to it as a parameter, then closed when the block exits. `$?` is set to the exit status of the subprocess, and the value of the block is returned.

When a block is supplied along with a *command* of `"-"`, Ruby forks, running the block in both processes. In the parent process the block is passed an `IO` pipe connected to the child's standard input and output streams; in the child process the block is passed `nil`.

`Kernel.open("|command", mode='r')` behaves like `IO.popen(command, mode='r')`, when *command* is a `String`. Likewise, `Kernel.open("|-", mode='r')` behaves like `IO.popen("-", mode='r')`

`IO.read(filename, length, offset=0, options) #=> String`
Opens the file named *filename*, then returns its contents from byte *offset* to the end of the file. If *length* is given, it is the maximum number of bytes to return. If *options* is given it is an options `Hash`.

`IO.readlines(filename, separator=$/, limit, options) #=> String`
Returns the lines contained in the file named *filename* as an `Array` of `Strings`. Lines are delimited by *separator*; if this value is `nil` the entire file is treated

as a single line. If *limit* is given, at most that many characters will be returned for each line. If *options* is given it is an options `Hash`.

`IO.select(read, write, error, timeout) #=> Array or nil`
Aliases `Kernel#select`.

`IO.sysopen(filename, mode, permissions) #=> Fixnum`
Opens the file named *filename* and returns its file descriptor.

`IO.try_convert(object) #=> IO or nil`
Returns *object* converted to an `IO` object by calling its `#to_io` method; or `nil` if this is impossible.

`IO#<<(object) #=> IO`
Converts *object* to a `String` with `#to_s`, writes it to the receiver, then returns `self`.

`IO#binmode() #=> IO`
Puts the receiver into binary mode.

`IO#bytes() {|byte| } #=> Enumerator or IO`
Returns an `Enumerator` of the receiver's bytes, each represented as a `Fixnum`. If a block is given, yields each byte to the block in turn, then returns the receiver.

`IO#chars() {|char| } #=> Enumerator or IO`
Returns an `Enumerator` of the receiver's characters, each represented as a `String`. If a block is given, yields each character to the block in turn, then returns the receiver.

`IO#close() #=> nil`
Closes the receiver's stream, flushing any pending writes to the operating system.

`IO#close_on_exec?() #=> true or false`
Returns `true` if the receiver's *close on exec* flag is set; `false` otherwise. Raises `NotImplementedError` if unavailable on this platform.

`IO#close_read() #=> nil`
Closes the read end of a duplex stream such as a pipe. Raises `IOError` if the receiver is not a duplex stream.

`IO#close_write() #=> nil`
Closes the write end of a duplex stream such as a pipe. Raises `IOError` if the receiver is not a duplex stream.

`IO#closed?() #=> true or false`
Returns `true` if the receiver is closed—for duplex streams, both ends must be closed; `false`, otherwise.

`IO#each(separator=$/, limit) {|line| } #=> IO or Enumerator`
Enumerates the lines in the receiver. If a block is given, each line is yielded to it in turn; otherwise, an `Enumerator` is returned. Lines are separated by *separator*; if this value is `nil`, the entire file is treated as a single line. If *length* is given it is the maximum number of characters to return for each line.

`IO#each_byte() {|byte| } #=> Enumerator or IO`
Aliases `IO#bytes`.

`IO#each_char() {|char| } #=> Enumerator or IO`
Aliases `IO#chars`.

`IO#eof() #=> true or false`
Returns `true` if the receiver is at end of file; `false` otherwise. If the receiver is not open for reading, and `IOError` is raised.

`IO#eof?() #=> true or false`
Aliases `IO#eof`.

`IO#external_encoding() #=> Encoding`
Returns the external encoding associated with the receiver.

`IO#fcntl(command, argument) #=> Integer`
Issues, via the `fcntl(2)` system call, the command *command* to the receiver's stream with an argument of *argument*. See Manipulating File Descriptors for more details.

`IO#fileno() #=> Integer`
Returns the file descriptor associated with the receiver. Aliased by `IO#to_i`.

`IO#flush() #=> IO`
Flushes Ruby's I/O buffers, returning `self`.

`IO#fsync() #=> 0 or nil`
Flushes the operating system's I/O buffers via the `fsync(2)` system call, returning `0`; returns `nil` if this system call is unimplemented.

`IO#getbyte() #=> Fixnum or nil`
Returns the next byte from the receiver, or `nil` at end of file.

`IO#getc() #=> String or nil`
Returns the next character from the receiver, or `nil` at end of file.

`IO#gets(`*`separator=$/`*`, `*`limit`*`) #=> String or nil`
Returns the next line from the receiver's stream, and assigns it to `$_`. Lines are delimited by *separator*: a value of `""` is equivalent to `"\n\n"`, while a `nil` value treats the entire file as a single line. If *limit* is given, at most that many characters are returned per line. Returns `nil` at end of file.

`IO#internal_encoding() #=> Encoding`
Returns the internal encoding associated with the receiver.

`IO#ioctl(command, argument) #=> Integer`
Issues, via the `ioctl(2)` system call, the command *command* to the receiver's stream with an argument of *argument*. See Manipulating File Descriptors for more details.

`IO#isatty() #=> true or false`
Returns `true` if the receiver is associated with a terminal device; `false` otherwise. Aliased to `IO#tty?`.

`IO#lineno() #=> Integer`
Returns the current line number read from the receiver. If the stream is not open for reading, an `IOError` is raised.

`IO#lineno=(line) #=> Integer`
Sets the current line number to the `Integer` *line*.

`IO#lines(`*`separator=$/`*`, `*`limit`*`) {|line| } #=> IO or Enumerator`
Aliases `IO#each`.

`IO#pid() #=> Integer or nil`
Returns the process ID associated with the receiver—as set by `IO.popen`—or `nil` if there isn't one.

`IO#pos() #=> Integer`
Returns the current byte offset of the receiver.

`IO#pos=(offset) #=> 0`
Seeks to the given `Integer` byte offset.

`IO#print(`*`object=$_`*`, …) #=> nil`
Converts the given objects with `#to_s`, then writes them to the receiver's stream. Unless `$\` is `nil`, writes it, too.

`IO#printf(format, `*`object=$_`*`, …) #=> nil`
Expands the format string, *format* and its given arguments with `Kernel.sprintf`, then writes the result to the receiver's stream.

`IO#putc(object) #=> Object`
Writes a single byte to its receiver's stream, then returns its argument. It interprets a `Numeric` argument as a character code, writing the least-significant byte of the character corresponding to its integer part. A non-numeric argument is converted to a `String`, then its least-significant byte written to the stream. Please note the term *byte*: before Ruby 1.9.3, this method would only ever write a single byte, even when given a multi-byte character; as of 1.9.3, it behaves correctly with multi-byte characters.

`IO#puts(`*`object=nil`*`, …) #=> nil`
Converts the given objects with `#to_s`, appends `"\n"` to each of which that do not already end with a newline, then writes them to the receiver's stream. If *object* responds to `#to_ary`, it is substituted for this method's return value—i.e., the elements of `Array` arguments are printed one per line.

`IO#read(`*`length`*`, `*`buffer`*`) #=> String or nil`
Reads from the current position in the receiver's stream through to the end, returning the result. If *length* is given, it is the maximum number of bytes to read. If a `String` *buffer* is given, the data is read into it. At the end of file, `nil` is returned.

`IO#readbyte() #=> Fixnum`
Returns the next byte from the receiver's stream, raising `EOFError` at end of file.

`IO#readchar() #=> String`
Returns the next character from the receiver's stream, raising `EOFError` at end of file.

`IO#readline(`*`separator=$/`*`, `*`limit`*`) #=> String or nil`
Reads the next line from the receiver's stream in the manner of `IO#gets`, raising `EOFError` at end of file.

`IO#readlines(`*`separator=$/`*`, `*`limit`*`) #=> Array`
Returns the lines from the receiver's stream as an `Array` of `Strings`. Lines are delimited by *separator*: a value of `""` is equivalent to `"\n\n"`, while a `nil` value treats the entire file as a single line. If *limit* is given, at most that many characters are returned per line.

`IO#readpartial(limit, `*`result=""`*`) #=> String`
Attempts to read at most *limit* bytes from the receiver's stream without blocking by returning buffered data before reading from the stream. If *result* is given it is a `String` to which the read data is appended. Raises `EOFError` at end of file.

`IO#read_nonblock(limit, `*`result=""`*`) #=> String`
Sets the `NONBLOCK` flag on the receiver's file descriptor, then attempts to read at most *limit* bytes from the receiver's stream without blocking. If there is buffered data, that is returned before trying to read from the stream. If the stream can be read from without blocking, it is read from. Otherwise, either `Errno::EWOULDBLOCK` or `Errno::EAGAIN` is raised to indicate that the stream can not be read without blocking. If *result* is given it is a `String` to which the read data is appended. Raises `EOFError` at end of file.

`IO#reopen(io) #=> IO`
`IO#reopen(filename, mode) #=> IO`
Re-associates the receiver with the given I/O stream, *io*, or a new stream for a file named *filename* that is opened with access mode *mode*. Due to the way I/O operations perform buffering, reopening a stream—especially one that has already been read from—can lead to unexpected behaviour. See Buffering and [Ruby-core-28281] for more details.

`IO#rewind() #=> 0`
Resets both the position of the receiver's stream and its line number to `0`.

`IO#seek(offset, whence=File::SEEK_SET) #=> 0`
Seeks to *offset* in the receiver's stream. See Positions & Seeking for an explanation of *whence*.

`IO#set_encoding(external, internal=external) #=> IO`
`IO#set_encoding(string) #=> IO`
Sets the external and internal encodings of the receiver's stream. Both *external* and *internal* may be `Encoding` objects or encoding names as `String`s. The *string* contains the name of the external encoding, a colon, then the name of the internal encoding; or, just one encoding name for both.

`IO#stat() #=> File::Stat`
Returns a `File::Stat` object for the receiver's stream.

`IO#sync() #=> true or false`
Returns `true` if the receiver is in *sync mode*; `false`, otherwise.

`IO#sync=(boolean) #=> true or false`
Sets the *sync mode* of the receiver to *boolean*—`true` or `false`—which it then returns.

`IO#sysread(limit, buffer) #=> String`
Reads at most *limit* bytes from the receiver's stream, bypassing Ruby's I/O buffer, returning them as a `String`. If the `String` *buffer* is given, it has the read data appended. Raises `SystemCallError` on error and `EOFError` at end of file.

`IO#sysseek(offset, whence=File::SEEK_SET) #=> Integer`
Behaves as `IO#seek` but bypasses Ruby's I/O buffer.

`IO#tell() #=> Integer`
Aliases `IO#pos`.

`IO#to_i() #=> Integer`
Aliases `IO#fileno`.

`IO#to_io() #=> IO`
Returns the receiver.

`IO#ungetbyte(byte) #=> nil`
Pushes back the given byte(s) onto the receiver's read buffer. *byte* may be a `String` or a single byte given as a `Fixnum`.

`IO#ungetc(character) #=> nil`
Pushes back the characters contained in the *character* `String` onto the receiver's read buffer.

`IO#write(object) #=> Integer`
Converts *object* to a `String` with `#to_s`, writes it to the receiver's stream, then returns the number of bytes written.

`IO#write_nonblock(object) #=> Integer`
Sets the `File::NONBLOCK` flag on the receiver's stream then behaves as `IO#write`. If a write would block, either `Errno::EWOULDBLOCK` or `Errno::EAGAIN`. If the platform doesn't support non-blocking writes for this type of `IO` object, `Errno::EBADF` is raised.

KERNEL

`Kernel.Array(object) #=> Array`
Converts *object* to an `Array` with either `#to_ary` or `#to_a`. If neither succeed, returns a new `Array` with *object* as its sole element. If *object* is `nil`, returns `[]`.

`Kernel.Complex(real, imaginary=0) #=> Complex`
`Kernel.Complex(string) #=> Complex`
Creates and returns a `Complex` number. The first form sets the real part to *real*, and the imaginary part to *imaginary*, both of which may be `Numerics` or `Strings`. The second form expects a `String` representation of a complex number, i.e. the `#to_s` form of a `Numeric`, `+` or `-`, the `#to_s` form of another `Numeric`, then `i`.

`Kernel.Float(object) #=> Float`
Returns the argument converted to a `Float`, either implicitly—if `Numeric`—or via `#to_f`. A `TypeError` is raised if *object* is `nil`.

`Kernel.Integer(object) #=> Integer`
Returns the argument converted to a `Fixnum` or `Bignum`, either implicitly—if `Numeric`—or via `#to_int` or `#to_i`. If *object* is a `String`, leading radix indications are understood: a `0` prefix implies octal, `0b`, binary, and `0x`, hexadecimal. A `TypeError` is raised if *object* is `nil`.

`Kernel.Rational(numerator, denominator=1) #=> Rational`
`Kernel.Rational(string) #=> Rational`
Creates and returns a `Rational` number. The first form sets the numerator to *numerator*, and the denominator to *denominator*, both of which may be `Numerics` or `Strings`. The second form expects a `String` representation of a rational number, i.e. the `#to_s` form of a `Numeric`, `/`, then the `#to_s` form of another `Numeric`.

`Kernel.String(object) #=> String`
Converts *object* to a `String` with `#to_s`, which it then returns.

`Kernel.__callee__() #=> Symbol or nil`
Returns the name of the current method, or `nil` if called outside of a method. Aliased to `Kernel.__method__`.

`Kernel.__method__() #=> Symbol or nil`
Aliases `Kernel.__callee__`.

`Kernel.`(command) #=> String`
Runs the `String` *command* in a subshell, then returns its standard output.

`Kernel.abort(`*message*`) #=> N/A`
Terminates the current process with an exit code of `1`. If the *message* `String` is given, it is written to standard error.

`Kernel.at_exit() { } #=> Proc`
Registers the given block to be executed just prior to program termination—if multiple blocks are registered in this way, they are executed in reverse chronological order.

`Kernel.autoload(constant, filename) #=> nil`
Causes the given filename to be required with `Kernel.require` the first time the constant named *constant* is accessed. The constant name is given as a `String` or `Symbol`, and resolved relative to the current scope.

`Kernel.autoload?(constant) #=> String or nil`
Returns the filename that will be autoloaded when the constant named *constant* is first accessed at the top-level, or `nil` if there is no file registered.

`Kernel.binding() #=> Binding`
Returns a `Binding` encapsulating the variable and method bindings at its call site.

`Kernel.block_given?() #=> true or false`
Returns `true` if the current method has been given a block argument; otherwise, `false`. Aliased by `Kernel.iterator?`.

`Kernel.caller(`*omit=0*`) #=> Array`
Returns the current execution stack as an `Array` of `Strings`, skipping the first *omit* frames. A frame has the form *`file`*`:`*`line`*`:in `` `location' ``. *file* is an

absolute filename, or, if there is no associated file, a parenthesised description of the location, e.g. `(irb)`. *line* is the line number. Lastly, *location* is normally the method name, possibly preceded by `block in` or `block(n levels) in .`

`Kernel.catch(`*object=Object.new*`) { } #=> Object`

Executes its block, expecting it to throw an object equal to *object*. If such a throw-clause is found, `catch` terminates its block, returning `throw`'s second argument. Otherwise, it returns the last expression of the block.

`Kernel.chomp(`*string*`) #=> String`

If `$_` ends with *string*, *string* is deleted; otherwise, this method is a no-op. Returns the new value of `$_`. This method is only defined when the `-n` or `-p` options are given to the interpreter.

`Kernel.chop() #=> String`

If `$_` ends with `\r\n`, both characters are removed; otherwise, just the last character is removed. Returns the new value of `$_`. This method is only defined when the `-n` or `-p` options are given to the interpreter.

`Kernel.eval(string,` *binding*, *file*, `line) #=> Object`

Evaluates *string* as Ruby code, then returns the result. If a `Binding` object is given as *binding*, the evaluation occurs in the binding's context. If a filename and line number are given as *file* and *line*, respectively, they are used in reporting errors emanating from the evaluation.

`Kernel.exec(`*environment={}*, `command,` *argument*, …, *options={}*`) #=> N/A`

Replaces the current process image with a new process image by executing *command* with the given *argument*s. *command* is subject to shell expansion only if no *argument*s are given. If *command* is given as an `Array`, its first element is the command to be executed, and its last element is that command's `argv[0]`. On Unix-like systems, this method uses a system call from the `exec(2)` family, so the new process inherits most of the current process's environment—including its file descriptors. If the *environment* `Hash` is present, it specifies environment variables for the new process: a `String` value sets the corresponding environment variable; a `nil` value clears it. *options* is an options `Hash`. If the command executes successfully, this method doesn't return; otherwise, a `SystemCallError` is raised.

`Kernel.exit(`*`status=1`*`) #=> N/A`
Exits the current process with a status of *status*, or raises `SystemExit` if called within an exception handler. A *status* of `true` is equivalent to `0`; `false`, `1`.

`Kernel.exit!(`*`status=1`*`) #=> N/A`
Behaves as `Kernel.exit`, but bypasses exception handlers, `Kernel.at_exit` blocks, and finalisers.

`Kernel.fail() #=> N/A`
`Kernel.fail(message) #=> N/A`
`Kernel.fail(exception, `*`message=""`*`, `*`backtrace`*`) #=> N/A`
The first form re-raises the exception in `$!`, or raises a new `RuntimeError` if `$!` is `nil`. If a `String` *message* is given, raises a `RuntimeError` with the given message. If *exception*—either an `Exception` class, or an object whose `#exception` method returns an `Exception`—it is raised with the given message. If *backtrace* is also given, it is an `Array` of `Strings` used for the exception's backtrace; otherwise, the backtrace is generated automatically. Aliased by `Kernel.raise`.

`Kernel.fork() `*`{ }`*` #=> Integer or nil`
Forks the current process to create a subprocess. If the block is specified, it is run in the subprocess; otherwise this method returns to the parent, the process ID of the child, and to the child, `nil`.

`Kernel.format(format, `*`argument`*`, …) #=> String`
Applies the format `String`, *format*, to the *argument*(s) to create a new `String`, which is returned. Aliased by `Kernel.sprintf`.

`Kernel.gets(`*`separator=$/`*`) #=> String or nil`
Returns the next line from `ARGV`, or `nil` at end of file. Lines are separated by, and include, *separator*. A *separator* of `nil` treats an entire file as a single line, while a *separator* of `""` is equivalent to `"\n\n"`. Each line read is assigned to `$_`.

`Kernel.global_variables() #=> Array`
Returns the names—as `Symbols`—of all defined global variables.

`Kernel.gsub(pattern, replacement) #=> String`
`Kernel.gsub(pattern) { } #=> String`

Behaves as `String#gsub` with an implicit receiver of `$_`. If substitution occurs, assigns the result back to `$_`. This method is only defined when the `-n` or `-p` options are given to the interpreter.

`Kernel.iterator?() #=> true or false`
Aliases `Kernel.block_given?`.

`Kernel.lambda() { } #=> Proc`
Creates a `Proc` with lambda semantics from the given block.

`Kernel.load(filename, wrap=false) #=> true`
Resolves *filename* relative to a directory in `$LOAD_PATH`, then loads and executes the Ruby code that it contains. If *wrap* is `true`, the code will be executed within an anonymous module, preventing it from modifying the global namespace.

`Kernel.local_variables() #=> Array`
Returns the names of the current local variables as an `Array` of `Symbols`.

`Kernel.loop() { } #=> Object`
Executes the given block repeatedly. If the block raises a `StopIteration` exception, the exception is rescued automatically and the loop terminated.

`Kernel.open(filename, mode='r', permissions) {|io| } #=> Object`
Opens a file named *filename*, which it returns as a `File` object. The *mode* may be either a given as a mode string or a logical OR of the file open flags. The permissions of the file are given by the `Integer` *permissions*, the meaning of which is platform dependent. If a block is given, the new `File` object will be passed to it, then closed when the block exits; the value of the block is returned to the caller. If *filename* begins with a pipe character (`|`), a subprocess is created instead. A pair of pipes connected to the standard input and output of this process are returned as an `IO` object. If the *filename* is `|-`, the interpreter forks, and `nil` is returned to the child; otherwise, the pipe character should be followed by the name of a command that is to be run in the subprocess. When a block is also given, it is run in both the parent and child process: in the former, its passed an `IO` object connected to the child's standard input and output; in the latter, its passed `nil`. When the block exits, the child process is terminated.

`Kernel.p(argument, …) #=> Object`
Writes to standard output the `#inspect` output for each *argument*, concatenated with the current output record separator.

`Kernel.print(argument=$_, …) #=> Object`
Writes to standard output the `#to_s` output of each *argument*, concatenated with the output field separator, and terminated with the output record separator.

`Kernel.printf(io=STDOUT, format, argument, …) #=> nil`
Writes to *io* the result of passing the remaining arguments to `Kernel.sprintf`.

`Kernel.proc() { } #=> Proc`
Creates and returns a `Proc` with proc semantics for the given block.

`Kernel.putc(argument) #=> Object`
Behaves as `STDOUT.putc(argument)`. See `IO#putc` for details.

`Kernel.puts(argument, …) #=> Object`
Invokes `IO#puts` on `STDOUT` with the given *argument*s.

`Kernel.raise() #=> N/A`
`Kernel.raise(message) #=> N/A`
`Kernel.raise(exception, message="", backtrace) #=> N/A`
Aliases `Kernel.fail`.

`Kernel.rand(max=0) #=> Numeric`
Generates a pseudo-random number between `0` and the absolute, integer value of *max*. If *max* is `0`, it is assumed to be `1.0`, and the return value is a `Float`; otherwise, an `Integer` is returned.

`Kernel.readline(separator=$/) #=> String or nil`
Returns the next line from `ARGV`, or raises `EOFError` at end of file. Lines are separated by, and include, *separator*. A *separator* of `nil` treats an entire file as a single line, while a *separator* of `""` is equivalent to `"\n\n"`. Each line read is assigned to `$_`.

`Kernel.readlines(`*`separator=$/`*`) #=> Array`
Returns the lines from `ARGV` as an `Array` of `Strings`. Lines are separated by, and include, *separator*. A *separator* of `nil` treats an entire file as a single line, while a *separator* of `""` is equivalent to `"\n\n"`.

`Kernel.require(feature) #=> true or false`
Resolves *feature* to an absolute path, then loads and executes the Ruby code or extension which it contains. If *feature* begins with `~`, `../`, or `/` it is resolved relative to the current working directory; otherwise, against a directory in `$LOAD_PATH`. If the filename does not end with a file extension, `.rb` and the default shared library extensions are appended to it in turn. If *feature* cannot be loaded, a `LoadError` is raised. Otherwise, it is searched for in the `$LOADED_FEATURES Array`: if present, it is not loaded again, so `false` is returned; if not present, it is appended, and `true` is returned.

`Kernel.require_relative(feature) #=> true or false`
Equivalent to `Kernel.require`, except *feature* is resolved relative to the path of the current source file.

`Kernel.select(read_array, `*`write_array`*`, `*`error_array`*`, `*`timeout`*`) #=> Array or nil`
Waits for any of the given `IO` objects to become ready, then returns those which are. The first three arguments are `Arrays` of `IO` objects: those in *read_array* are checked for whether they can be read from without blocking; those in *write_array*, for whether they be written to without blocking; and those in *error_array*, for whether an error occurs on the associated device. When at least one `IO` stream becomes ready, an `Array` of `Arrays` is returned: the first element is the streams ready for reading, the second, the streams ready for writing, and the third, the streams encountering an error. If *timeout* is given, and there is no change in status for this many seconds, `nil` is returned; otherwise, there is no timeout.

`Kernel.set_trace_func(proc) #=> Proc or nil`
Enables tracing of the current process by invoking the given `Proc` on every event, passing the details to *proc* as, at most, six arguments: the event name, the filename, the line number, the object ID, the binding, and the name of the class. If *proc* is `nil`, tracing is disabled. See Tracing for further details.

`Kernel.sleep(`*`seconds=0`*`) #=> Fixnum`
Suspends the current thread for *seconds* seconds, returning the actual number of seconds slept. *seconds* may be an `Integer`, or a `Float` specifying fractional seconds; if it is `0`, the thread sleeps forever.

`Kernel.spawn(`*`environment={}`*`, command, `*`argument`*`, …, `*`options={}`*`) #=> Fixnum`
Executes *command* with the given *argument*s in a subshell, returning immediately with its PID. *command* is subject to shell expansion only if no *argument*s are given. If *command* is given as an `Array`, its first element is the command to be executed, and its last element is that command's `argv[0]`. On Unix-like systems, this method uses a system call from the `exec(2)` family, so the new process inherits most of the current process's environment—including its file descriptors. If the *environment* `Hash` is present, it specifies environment variables for the new process: a `String` value sets the corresponding environment variable; a `nil` value clears it. *options* is an options `Hash`. A `SystemCallError` is raised on failure.

`Kernel.sprintf(format_string, `*`argument`*`, …) #=> String`
Expands the format string by interpolating the given *argument*s, then returns the result. See Format Strings for further details.

`Kernel.srand(`*`seed`*`) #=> Integer`
Converts *seed* to an `Integer`, uses it to seed the pseudo-random number generator, then returns the previous seed. If *seed* is omitted, it is derived from a combination of the current time, the PID, and a sequence number.

`Kernel.sub(pattern, replacement) #=> String`
`Kernel.sub(pattern) { } #=> String`
Behaves as `String#sub` with an implicit receiver of `$_`. If substitution occurs, assigns the result back to `$_`. This method is only defined when the `-n` or `-p` options are given to the interpreter.

`Kernel.syscall(number, `*`argument`*`, …) #=> Integer`
Performs the system call identified by *number*, passing in the given arguments. The arguments must be `String`s or `Integer`s that fit within a native `long`.

`Kernel.system(`*`environment={}`*`, command, `*`argument`*`, …, `*`options={}`*`)`
`#=> true, false, or nil`
Executes *command* with the given *argument*s in a subshell, returning `true` if it ran successfully, `false` if it exited with a non-zero status, and `nil` if it failed to execute. *command* is subject to shell expansion only if no *argument*s are given. If *command* is given as an `Array`, its first element is the command to be executed, and its last element is that command's `argv[0]`. On Unix-like systems, this method uses a system call from the `exec(2)` family, so the new process inherits most of the current process's environment—including its file descriptors. If the *environment* `Hash` is present, it specifies environment variables for the new process: a `String` value sets the corresponding environment variable; a `nil` value clears it. *options* is an options `Hash`. A `SystemCallError` is raised on failure.

`Kernel.test(command, file1, `*`file2`*`) #=> Object`
Performs the test given by the `Integer` *command* on the named files. See Kernel.test for details.

`Kernel.throw(symbol, `*`object`*`) #=> N/A`
Jumps to the end of the enclosing `catch` block expecting *symbol*, or raises a `NameError` if there is no such block. If *object* is given, it is returned by the corresponding `catch` block.

`Kernel.trace_var(name, `*`command`*`) #=> nil`
`Kernel.trace_var(name) {|value| } #=> nil`
Traces explicit assignments to the global variable named *name*. If *command* is a `Proc`, or if a block is supplied, the `Proc` or block is invoked on each assignment with the variable's new value as a parameter. Otherwise, if *command* is a `String`, it is evaluated as Ruby code on each assignment.

`Kernel.trap(signal, proc) #=> Object`
`Kernel.trap(signal) { } #=> Object`
Aliases `Signal.trap`.

`Kernel.untrace_var(name, `*`command`*`) #=> Array or nil`
Disables tracing for the global variable named *name*. If *command* is given, only tracing for that command is disabled, and `nil` is returned; otherwise, all tracing is disabled, and an `Array` of the disabled commands is returned.

`Kernel.warn(message) #=> nil`
Writes the given message to the standard error stream, unless `$VERBOSE` is `nil`.

`Kernel#!~(object) #=> true or false`
Returns `true` if the receiver does not match, as per `#=~`, *object*; otherwise, `false`.

`Kernel#<=>(object) #=> 0 or nil`
Returns `0` if the receiver equals, as per `#==`, *object*; otherwise, `nil`.

`Kernel#===(object) #=> true or false`
Aliases `BasicObject#==`. Usually overridden to test for case equality.

`Kernel#=~(object) #=> nil`
Returns `nil`. Usually overridden to match the receiver against a `Regexp`.

`Kernel#class() #=> Class`
Returns the `Class` of the receiver.

`Kernel#clone() #=> Object`
Returns a shallow copy of the receiver: its instance variables are copied by reference rather than value, and its tainted and frozen state is preserved.

`Kernel#define_singleton_method(name, body) #=> Proc`
`Kernel#define_singleton_method(name) { } #=> Proc`
Defines on the receiver a singleton method named with the `Symbol` *name*. The method body may be given as a `Proc`, `Method`, `UnboundMethod`, or literal block. In the last case, the block is evaluated via `BasicObject#instance_eval`.

`Kernel#display(stream=$>) #=> nil`
Uses `IO#write` to write the receiver to the `IO` *stream*.

`Kernel#dup() #=> Object`
Returns a shallow copy of the receiver: its instance variables are copied by reference rather than value, and its tainted state is preserved.

`Kernel#enum_for(name=:each, argument, …) #=> Enumerator`
Returns an `Enumerator` that will traverse the receiver using its method named

name. Any *argument*s are passed directly to this method. Aliased by `Kernel#to_enum`.

`Kernel#eql?(object) #=> true or false`
Aliases `BasicObject#==`.

`Kernel#extend(module, …) #=> Object`
Mixes-in each given `Module` to the receiver's singleton class, returning the receiver.

`Kernel#freeze() #=> Object`
Freezes then returns the receiver.

`Kernel#frozen?() #=> true or false`
Returns `true` if the receiver is frozen; otherwise, `false`.

`Kernel#hash() #=> Fixnum`
Returns the unique hash value for the receiver.

`Kernel#__id__() #=> Fixnum`
Aliases `BasicObject#object_id`.

`Kernel#initialize_clone(object) #=> Object`
Callback invoked by both `Kernel#clone`, expected to copy additional state from the receiver to the new object, *object*. By default, invokes `Kernel#initialize_copy`, passing in *object*.

`Kernel#initialize_dup(object) #=> Object`
Callback invoked by both `Kernel#dup`, expected to copy additional state from the receiver to the new object, *object*. By default, invokes `Kernel#initialize_copy`, passing in *object*.

`Kernel#inspect() #=> String`
Returns a human-readable representation of the receiver that is suitable for debugging purposes.

`Kernel#instance_of?(class) #=> true or false`
Returns `true` if the receiver is an instance of the given `Class` object; otherwise, `false`.

`Kernel#instance_variable_defined?(name) #=> true or false`
Returns `true` if the receiver defines an instance variable named *name*; otherwise, `false`. *name* is a `Symbol` of the form `:@identifier`..

`Kernel#instance_variable_get(name) #=> Object`
Returns the value of the receiver's instance variable named *name*, raising a `NameError` if it is undefined. *name* is a `Symbol` of the form `:@identifier`..

`Kernel#instance_variable_set(name, object) #=> Object`
Assigns *object* to the receiver's instance variable named *name*, returning *object*. *name* is a `Symbol` of the form `:@identifier`..

`Kernel#instance_variables() #=> Array`
Returns the names of instance variables defined in the receiver as an `Array` of `Symbols`.

`Kernel#is_a?(class) #=> true or false`
Returns `true` if the given `Class` or `Module` is an ancestor of the receiver's class; otherwise, `false`. Aliased by `Kernel#kind_of?`.

`Kernel#kind_of?(class) #=> true or false`
Aliases `Kernel#is_a?`.

`Kernel#method(name) #=> Method`
Returns an objectification of the receiver's method named *name*, where *name* is a `Symbol`. Raises a `NameError` if the named method does not exist.

`Kernel#methods(all_public=true) #=> Array`
Returns the names of both public and protected methods to which the receiver responds, as an `Array` of `Symbols`. If *all_public* is `false`, equivalent to `Kernel#singleton_methods`.

`Kernel#nil?() #=> false`
Returns `false`; overridden by `NilClass#nil?`.

`Kernel#object_id() #=> Fixnum`
Aliases `BasicObject#__id__`.

`Kernel#private_methods() #=> Array`
Returns the names of the private methods to which the receiver responds, as an `Array` of `Symbol`s.

`Kernel#protected_methods() #=> Array`
Returns the names of the protected methods to which the receiver responds, as an `Array` of `Symbol`s.

`Kernel#public_method(name) #=> Method`
Returns an objectification of the receiver's public method named *name*, where *name* is a `Symbol`. Raises a `NameError` if the named method does not exist or isn't public.

`Kernel#public_methods(inherited=true) #=> Array`
Returns the names of the public methods to which the receiver responds, as an `Array` of `Symbol`s. If *inherited* is `false`, inherited methods are omitted.

`Kernel#public_send(name, argument, …) #=> Object`
Invokes the receiver's public method named *name* with the given *argument*s, returning the method's value.

`Kernel#respond_to?(name, private=false) #=> true or false`
Returns `true` if the receiver responds to a public or protected method named *name*; `false`, otherwise. If *private* is `true`, considers private methods, too.

`Kernel#respond_to_missing?(name, private=false) #=> false`
Hook called by `Kernel#respond_to?` when the receiver doesn't define a method named *name*. If an object responds to a message via `BasicObject#method_missing` it is supposed to override this method to return `true` when given the selector.

`Kernel#send(name, argument, …, &block) #=> Object`
Aliases `BasicObject#__send__`.

`Kernel#singleton_class() #=> Class`
Returns the receiver's singleton class, creating it if necessary.

`Kernel#singleton_methods(from_modules=true) #=> Array`
Returns the names of the receiver's singleton methods, as an `Array` of

Symbols. If *from_modules* is false, the list excludes methods defined in Modules mixed-in to the receiver's singleton class.

Kernel#taint() #=> Object
Taints and returns the receiver.

Kernel#tainted?() #=> true or false
Returns true if the receiver is tainted; otherwise, false.

Kernel#tap() {|object| } #=> Object
Yields the receiver to the block then returns the receiver.

Kernel#to_enum(*name=:each*, *argument*, …) #=> Enumerator
Aliases Kernel#enum_for.

Kernel#to_s() #=> String
Returns a String containing the receiver's class and object ID. For the top-level object, returns "main".

Kernel#trust() #=> Object
Trusts and returns the receiver.

Kernel#untaint() #=> Object
Un-taints and returns the receiver.

Kernel#untrust() #=> Object
Un-trusts and returns the receiver.

Kernel#untrusted?() #=> Object
Returns true if the receiver is not trusted; otherwise, false.

Kernel.gem(name, *version*, …) #=> true or false
Adds the directories holding the gem named *name* to $LOAD_PATH. By default, the latest version of the gem is added; other versions can be specified by supplying one or more *version* predicates, all of which must be satisfied. A predicate has the form *operator number*. *operator* is one of = (this version only), != (any version but this), > (a higher version than this), < (a lower version than this), >= (at least this version), <= (at most this version), and ~> (at least this version, but less than this version after incrementing its

penultimate digit by `1`). *number* is up to three integers separated by periods which correspond to the major version, the minor version, and the patch level, respectively; omitted parts default to `0`.

`Kernel#initialize_copy(object) #=> Object`

Callback invoked by both `Kernel#initialize_clone` and `Kernel#initialize_dup`, expected to copy additional state from the receiver to the new object, *object*.

`Kernel#remove_instance_variable(name) #=> Object`

Removes the receiver's instance variable named *name*, returning its old value. *name* is a `Symbol` of the form `:@`*`identifier`*.

MARSHAL

`Marshal.dump(object, io, limit=-1) #=> IO or String`
Serialises *object* and all of its descendants, writing the result to the `IO` stream, *io*, if specified, or returning it as a `String`. If *limit* is positive, it specifies the maximum depth of descendant objects to serialise; if it is negative, there is no limit.

`Marshal.load(source, proc) #=> Object`
De-serialises the data in *source* to a Ruby object. *source* is either an `IO` stream from which the data is read, or an object responding to `#to_str`. If *proc* is given as a `Proc`, it is invoked with each object as it is de-serialised. Aliased by `Marshal.restore`.

`Marshal.restore(source, proc) #=> Object`
Aliases `Marshal.load`.

MATCHDATA

```
  MatchData#[](group) #=> String
MatchData#[](start, length) #=> Array
MatchData#[](range) #=> Array
```

The first form returns the text corresponding to the given group: a `Fixnum` specifies a numbered group, and a `Symbol` specifies a named group. If *group* is `0`, the entire matched string is returned. The second and third forms return the text corresponding to the *length* consecutive capture groups starting from *start*, or those in positions specified by the given `Range`.

```
  MatchData#begin(group) #=> Integer
```

Returns the character offset in the original string where the given capture group began. *group* may be a group's position, as a `Fixnum`, or name, as a `Symbol`.

```
  MatchData#captures() #=> Array
```

Returns the text corresponding to each capture group as an `Array` of `Strings`.

```
  MatchData#end(group) #=> Integer
```

Returns the character offset in the original string where the given capture group ended. *group* may be a group's position, as a `Fixnum`, or name, as a `Symbol`.

```
  MatchData#length() #=> Integer
```

Returns the number of captured groups, i.e. the number of elements in `MatchData#captures`. Aliased by `MatchData#size`.

```
  MatchData#names() #=> Array
```

Returns the names of each named capture group as an `Array` of `Strings`.

```
  MatchData#offset(group) #=> Array
```

Returns the character offsets in the original string where the given capture group began and ended. *group* may be a group's position, as a `Fixnum`, or name, as a `Symbol`.

`MatchData#post_match() #=> String`
Returns the portion of the original string which follows this match. Equivalent to `$'`.

`MatchData#pre_match() #=> String`
Returns the portion of the original string which precedes this match. Equivalent to `` $` ``.

`MatchData#regexp() #=> Regexp`
Returns the regular expression used in this match.

`MatchData#size() #=> Integer`
Aliases `MatchData#length`.

`MatchData#string() #=> String`
Returns a frozen copy of the original string used in this match.

`MatchData#to_a() #=> Array`
Returns the text corresponding to each capture group as an `Array` of `Strings`, with the full string matched as the first element.

`MatchData#to_s() #=> String`
Returns the matched string.

`MatchData#values_at(index, …) #=> Array`
Returns the text corresponding to each numbered group whose *index* is given.

MATH

```
Math.acos(x) #=> Float
```
Returns the arc cosine of angle *x*, which is given in radians.

```
Math.acosh(x) #=> Float
```
Returns the inverse hyperbolic cosine of angle *x*, which is given in radians.

```
Math.asin(x) #=> Float
```
Returns the arc sine of angle *x*, which is given in radians.

```
Math.asinh(x) #=> Float
```
Returns the inverse hyperbolic sine of angle *x*, which is given in radians.

```
Math.atan(x) #=> Float
```
Returns the arc tangent of angle *x*, which is given in radians.

```
Math.atanh(x) #=> Float
```
Returns the inverse hyperbolic tangent of angle *x*, which is given in radians.

```
Math.atan2(y, x) #=> Float
```
Returns the principal value of the arc tangent of *y*/*x*.

```
Math.cbrt(n) #=> Float
```
Returns the cube root of the given `Numeric`.

```
Math.cos(x) #=> Float
```
Returns the cosine of angle *x*, which is given in radians.

```
Math.erf(x) #=> Float
```
Returns the error function of *x*.

```
Math.erfc(x) #=> Float
```
Returns the complementary error function of *x*.

```
Math.exp(x) #=> Float
```
Returns `Math::E ** x`.

`Math.frexp(n) #=> Array`
Returns an `Array` whose first element is a `Float` which, when multiplied by `2` raised to the power of the last element, equals *n*.

`Math.gamma(x) #=> Float`
Returns the result of the Gamma function for *x*.

`Math.hypot(x, y) #=> Float`
Returns the hypotenuse of the right-angled triangle with sides *x* and *y*.

`Math.ldexp(base, exponent) #=> Float`
Returns the product of the `Float` *base* and `2` raised to the `Integer` *exponent*.

`Math.lgamma(x) #=> Array`
Returns a two-element `Array` whose first element is the natural logarithm of the absolute value of the Gamma function for *x*, and last element is `-1` if the Gamma function returned a negative number, or `1`, otherwise.

`Math.log(n, base=Math::E) #=> Float`
Returns the natural logarithm of the `Numeric` *n* in the base *base*.

`Math.log10(n) #=> Float`
Returns the base-10 logarithm of the given `Numeric`.

`Math.log2(n) #=> Float`
Returns the base-2 logarithm of the given `Numeric`.

`Math.sin(x) #=> Float`
Returns the sine of angle *x*, which is given in radians.

`Math.sinh(x) #=> Float`
Returns the hyperbolic sine of angle *x*, which is given in radians.

`Math.sqrt(x) #=> Float`
Returns the non-negative square root of *x*, raising `Math::DomainError` if *x* is negative.

`Math.tan(x) #=> Float`
Returns the tangent of angle *x*, which is given in radians.

```
Math.tanh(x) #=> Float
```

Returns the hyperbolic tangent of angle *x*, which is given in radians.

METHOD

`Method#[](`*argument*`, …) #=> Object`

Invokes the objectified method with the given arguments, returning the result. Aliased by `Method#call`.

`Method#==(object) #=> true or false`

Returns `true` if *object* is a `Method` object representing the same method as objectified by the receiver, or an alias thereof; otherwise, `false`. Aliased by `Method#eql?`.

`Method#arity() #=> Fixnum`

Returns the arity of the objectified method: if it accepts a fixed number of arguments, this number is returned; for methods implemented in Ruby that accept a variable number of arguments, the negative of this number less one is returned; for methods implemented in C that accept a variable number of arguments, `-1` is returned.

`Method#call(`*argument*`, …) #=> Object`

Aliases `Method#[]`.

`Method#eql?(object) #=> true or false`

Aliases `Method#==`.

`Method#name() #=> Symbol`

Returns the name of the objectified method, i.e. its selector.

`Method#owner() #=> Module`

Returns the `Class` or `Module` in which the objectified method is defined.

`Method#parameters() #=> Array`

Returns an `Array`, each element of which describes a parameter accepted by the objectified method as an `Array` of the form `[`*type*`, `*name*`]`. *type* is `:req` if the parameter is required, `:opt` if the parameter is optional, `:rest` if the parameter accepts a variable number of arguments, or `:block` if the parameter expects a block literal. *name* is the parameter's name as a `Symbol`.

`Method#receiver() #=> Object`
Returns the object to which the objectified method is bound.

`Method#source_location() #=> Array or nil`
Returns the absolute filename and line number where the objectified method was defined, or `nil` if it is implemented in C.

`Method#to_proc() #=> Proc`
Returns a `Proc` corresponding to the objectified method.

`Method#unbind() #=> UnboundMethod`
Returns the objectified method detached from its receiver.

MODULE

`Module.constants(`*`include_ancestors`*`) #=> Array`
Returns the names of top-level constants as an `Array` of `Symbols`. If *include_ancestors* is `true`, the receiver's ancestors are also searched; otherwise, they're not.

`Module.nesting() #=> Array`
Returns the enclosing `Module`, the `Module` which encloses that, and so on, as an `Array` of `Modules`.

`Module.new() {|`*`module`*`| } #=> Module`
Creates and returns a new, anonymous `Module`. If the block is supplied, it is passed this object, then evaluated in the context of the new `Module`.

`Module#<(module) #=> true, false, or nil`
Returns `true` if *module* is included in, or a subclass of, the receiver `Module` or one of its ancestors. Returns `false` if the two `Modules` are related in another way, or `nil` if they're not related at all.

`Module#<=(module) #=> true, false, or nil`
Returns `true` if *module* is included in, a subclass of, or equal to, the receiver `Module` or one of its ancestors. Returns `false` if the two `Modules` are related in another way, or `nil` if they're not related at all.

`Module#>(module) #=> true, false, or nil`
Returns `true` if *module* is included by, or subclasses, the receiver `Module` or one of its ancestors. Returns `false` if the two `Modules` are related in another way, or `nil` if they're not related at all.

`Module#>=(module) #=> true, false, or nil`
Returns `true` if *module* is included by, subclasses, or is equal to, the receiver `Module` or one of its ancestors. Returns `false` if the two `Modules` are related in another way, or `nil` if they're not related at all.

`Module#<=>(module) #=> -1, 0, 1`
Returns `-1` if the receiver includes *module*, `0` if the two `Modules` are equal, or `1`, otherwise.

`Module#===(object) #=> true or false`
Returns `true` if *object* is either an instance or descendant of the receiver; otherwise, `false`.

`Module#ancestors() #=> Array`
Returns an `Array` comprising the receiver and each of the `Modules` it includes.

`Module#autoload(name, filename) #=> nil`
Arranges for `require(`*`filename`*`)` to be invoked, in the top-level context, the first time that the `Module` named *name* is referenced within the namespace of the receiver. *name* may be either a `String` or `Symbol`.

`Module#autoload?(name) #=> String or nil`
Returns the filename that will be automatically loaded when the `Module` named *name* is referenced within the namespace of the receiver, or `nil` if there is no such file. *name* may be either a `String` or `Symbol`.

`Module#class_eval(ruby, `*`filename`*`, `*`line`*`) #=> Object`
`Module#class_eval() {|module| } #=> Object`
Evaluates either a `String` of Ruby, *ruby*, or the given block, in the context of the receiver. The `String` *filename* and `Fixnum` *line* are used as the filename and line number, respectively, reported in error messages. The block is passed the receiver as its argument. Aliased by `Module#module_eval`.

`Module#class_exec(`*`argument`*`, …) {|*arguments| } #=> Object`
Evaluates the given block in the context of the receiver, passing in its arguments. Aliased by `Module#module_exec`.

`Module#class_variable_defined?(name) #=> true or false`
Returns `true` if a class variable named *name* is defined in the receiver; otherwise, `false`. *name* is a `Symbol` of the form `:@@`*`identifier`*.

`Module#class_variable_get(name) #=> Object`
Returns the value of the class variable named *name* which is defined in the receiver. *name* is a `Symbol` of the form `:@@`*`identifier`*.

`Module#class_variable_set(name, value) #=> Object`
Assigns *value* to the class variable named *name* which is defined in the receiver. *name* is a `Symbol` of the form `:@@identifier`.

`Module#class_variables() #=> Array`
Returns the names of class variables defined in the receiver as an `Array` of `Symbols`.

`Module#const_defined?(name, include_ancestors=true) #=> true or false`
Returns `true` if a constant named *name* (a `Symbol`) is defined in the receiver; otherwise, `false`. If *include_ancestors* is `true`, the receiver's ancestors are also searched.

`Module#const_get(name) #=> Object`
Returns the value of the constant named *name* (a `Symbol`) which is defined in the receiver.

`Module#const_missing(name) #=> Object`
Hook method invoked when an undefined constant named *name* (a `Symbol`) is referenced, and expected to either return the corresponding value or delegate to its parent with `super`.

`Module#const_set(name, value) #=> Object`
Assigns *value* to the receiver's constant named *name* (a `Symbol`), creating it if necessary.

`Module#constants(include_ancestors=true) #=> Array`
Returns the names of constants accessible from the receiver as an `Array` of `Symbols`. If *include_ancestors* is `true`, the receiver's ancestors are also searched.

`Module#include?(module) #=> true or false`
Returns `true` if *module* is a `Module` included in the receiver or its ancestors; otherwise, `false`.

`Module#included_modules() #=> Array`
Returns the `Modules` included in the receiver and its ancestors.

`Module#instance_method(name) #=> UnboundMethod`
Returns an objectification of the receiver's instance method named *name*, where *name* is a `Symbol`.

`Module#instance_methods(include_ancestors=true) #=> Array`
Returns the names of the receiver's non-private instance methods as an `Array` of `Symbols`. When *include_ancestors* is `true`, the ancestors of the receiver are included in the search; otherwise, they're not.

`Module#method_defined?(name) #=> true or false`
Returns `true` if a public or protected instance method named *name* is defined in the receiver or its ancestors; otherwise, `false`.

`Module#module_eval(ruby, filename, line) #=> Object`
`Module#module_eval() {|module| } #=> Object`
Aliases `Module#class_eval`.

`Module#module_exec(argument, …) {|*arguments| } #=> Object`
Aliases `Module#class_exec`.

`Module#name() #=> String`
Returns the name of the receiver.

`Module#private_class_method(name, …) #=> Object`
Makes private each class method with one of the given names.

`Module#private_instance_methods(include_ancestors=true) #=> Array`
Returns the names of the receiver's private instance methods as an `Array` of `Symbols`. When *include_parents* is `true`, the ancestors of the receiver are included in the search; otherwise, they're not.

`Module#private_method_defined?(name) #=> true or false`
Returns `true` if a private instance method named *name* is defined in the receiver or its ancestors; otherwise, `false`.

`Module#protected_instance_methods(include_ancestors=true) #=> Array`
Returns the names of the receiver's protected instance methods as an `Array`

of Symbols. When *include_parents* is true, the ancestors of the receiver are included in the search; otherwise, they're not.

Module#protected_method_defined?(name) #=> true or false
Returns true if a protected instance method named *name* is defined in the receiver or its ancestors; otherwise, false.

Module#public_class_method(*name*, …) #=> Object
Makes public each class method with one of the given names.

Module#public_instance_method(name) #=> UnboundMethod
Returns an objectification of the receiver's public instance method named *name*, where *name* is a Symbol.

Module#public_instance_methods(*include_ancestors=true*) #=> Array
Returns the names of the receiver's public instance methods as an Array of Symbols. When *include_parents* is true, the ancestors of the receiver are included in the search; otherwise, they're not.

Module#public_method_defined?(name) #=> true or false
Returns true if a public instance method named *name* is defined in the receiver or its ancestors; otherwise, false.

Module#remove_class_variable(name) #=> Object
Undefines the receiver's class variable named *name*, where *name* is a Symbol of the form :@@*identifier*.

Module#alias_method(alias, name) #=> Module
Defines an alias named *alias* for the receiver's method named *name*.

Module#append_features(name) #=> Module
Hook method invoked when the receiver is included in a module named *name*. In order for Ruby to import the constants, methods, and class variables of *name* into the receiver, this method should call super.

Module#attr(name, …) #=> nil
For each given *name* creates an instance variable named @*name*, and an instance method named *name* which returns the value of this variable. *name* is a Symbol. Aliased by Module#attr_reader.

`Module#attr_accessor(name, …) #=> nil`
For each given *name* creates an instance variable named `@name`, an instance method named *name* which returns the value of this variable, and an instance method named *name*= which assigns its argument to this variable. *name* is a `Symbol`.

`Module#attr_reader(name, …) #=> nil`
Aliases `Module#attr`.

`Module#attr_writer(name, …) #=> nil`
For each given *name* creates an instance variable named `@name`, and an instance method named *name*= which assigns its argument to this variable. *name* is a `Symbol`.

`Module#define_method(name, body) #=> Proc`
`Module#define_method(name) { } #=> Proc`
Defines in the receiver an instance method named *name* with a body of *body* or the given block. *name* is a `Symbol` or `String`; *body* is a `Proc`, `Method`, or `UnboundMethod`.

`Module#extend_object(object) #=> Object`
Hook invoked by `Object#extend` on the `Module` with which *object* is being extended. Should delegate to its parent with `super`. Aliased to `Module#extended`.

`Module#extended(object) #=> Object`
Aliases `Module#extend_object`.

`Module#include(module, …) #=> Module`
Mixes in each of the given `Modules` to the receiver by iterating over its arguments in reverse order, invoking `Module#append_features`, then invoking `Module#included`.

`Module#method_added(name) #=> Object`
Hook method invoked when a method is defined on the receiver. *name* is the new method's name as a `Symbol`.

`Module#method_removed(name) #=> Object`
Hook method invoked when a method is removed from the receiver. *name* is the removed method's name as a `Symbol`.

`Module#method_undefined(name) #=> Object`
Hook method invoked when a method is undefined from the receiver. *name* is the undefined method's name as a `Symbol`.

`Module#module_function(`*name*`, …) #=> Module`
Copies the receiver's instance method named *name* to the receiver's singleton class, then makes the instance method private. Repeats this process for each *name* specified. If called without arguments, applies this process to each instance method defined subsequently in the same scope.

`Module#private(`*name*`, …) #=> Module`
Makes each named instance method private, where *name* is a `Symbol`. If no arguments are given, sets the visibility of each instance method subsequently defined in the same scope to private.

`Module#protected(`*name*`, …) #=> Module`
Makes each named instance method protected, where *name* is a `Symbol`. If no arguments are given, sets the visibility of each instance method subsequently defined in the same scope to protected.

`Module#public(`*name*`, …) #=> Module`
Makes each named instance method public, where *name* is a `Symbol`. If no arguments are given, sets the visibility of each instance method subsequently defined in the same scope to public.

`Module#remove_const(name) #=> Object`
Removes from the receiver the definition of the constant named *name* (a `Symbol`), returning its value.

`Module#remove_method(name) #=> Module`
Removes from the receiver the definition of the method named *name* (a `Symbol`).

`Module#undef_method(name) #=> Module`
Undefines each named method from the receiver, where *name* is a `Symbol`.

MUTEX

`Mutex.new() #=> Mutex`
Creates and returns a new `Mutex`.

`Mutex#lock() #=> Mutex`
Tries to place a lock on this mutex. If already locked by another thread, blocks until the lock has been removed; if locked by the current thread, raises a `ThreadError`.

`Mutex#locked?() #=> true or false`
Returns `true` if this mutex is locked; otherwise, `false`.

`Mutex#sleep(`*`duration=nil`*`) #=> Integer`
Unlocks this mutex, sleeps for *duration* seconds, re-locks this mutex, then returns the number of seconds slept. If *duration* is `nil`, sleeps forever.

`Mutex#synchronize() { } #=> Object`
Locks this mutex, yields to the block, releases the lock, then returns the value of the block.

`Mutex#try_lock() #=> true or false`
Tries to place a lock on this mutex without blocking: if it is unlocked, locks it, then returns `true`; otherwise, returns `false`.

`Mutex#unlock() #=> Mutex`
Releases the current thread's lock on this mutex.

NILCLASS

`NilClass#&(object) #=> false`
Performs a logical AND between the receiver and *object*.

`NilClass#^(object) #=> true or false`
Performs an exclusive OR between the receiver and *object*: returns `false` if *object* is `false` or `nil`; otherwise, `true`.

`NilClass#|(object) #=> true or false`
Performs a logical OR between the receiver and *object*: returns `false` if *object* is `false` or `nil`; otherwise, `true`.

`NilClass#nil?() #=> true`
Returns `true`.

`NilClass#rationalize() #=> Rational`
Returns a new `Rational` object whose numerator is `0`, and denominator is `1`.

`NilClass#to_a() #=> Array`
Returns `[]`.

`NilClass#to_c() #=> Complex`
Returns a new `Complex` object whose real part and imaginary part are both `0`.

`NilClass#to_f() #=> Float`
Returns `0.0`.

`NilClass#to_i() #=> Integer`
Returns `0`.

`NilClass#to_r() #=> Rational`
Returns a new `Rational` object whose numerator is `0`, and denominator is `1`.

`NilClass#to_s() #=> String`
Returns `""`.

NUMERIC

`Numeric#%(number) #=> Numeric`
Returns the result of the receiver modulo *number*. Aliased by `Numeric#modulo`.

`Numeric#+@() #=> Numeric`
Returns the receiver with a positive sign.

`Numeric#-@() #=> Numeric`
Returns the receiver with a negative sign.

`Numeric#<=>(number) #=> 0 or nil`
Returns `0` if the receiver is equal to *number*; otherwise, `nil`.

`Numeric#abs() #=> Numeric`
Returns the absolute value of the receiver. Aliased by `Numeric#magnitude`.

`Numeric#abs2() #=> Numeric`
Returns the square of the absolute value of the receiver.

`Numeric#angle() #=> Numeric`
Returns `Math::PI` if the receiver is negative; otherwise, `0`. Aliased by `Numeric#arg` and `Numeric#phase`.

`Numeric#arg() #=> Numeric`
Aliases `Numeric#angle`.

`Numeric#ceil() #=> Integer`
Converts the receiver to a `Float` then returns `Float#ceil`.

`Numeric#coerce(object) #=> Array`
Returns an `Array` containing *object* and the receiver. If *object* isn't a `Numeric`, both elements of the `Array` are converted to `Floats`.

`Numeric#conj() #=> Numeric`
Returns the receiver. Aliased by `Numeric#conjugate`.

`Numeric#conjugate() #=> Numeric`
Aliases `Numeric#conj`.

`Numeric#denominator() #=> Integer`
Converts the receiver to a `Rational` then returns `Rational#denominator`.

`Numeric#div(number) #=> Numeric`
Divides the receiver by *number*, using `#/`, then converts the result to an `Integer`.

`Numeric#divmod(number) #=> Array`
Divides the receiver by *number*, returning an `Array` whose first element is the quotient, and last element, the modulus. The quotient is rounded toward $-\infty$.

`Numeric#eql?(number) #=> true or false`
Returns `true` if the receiver and *number* have both the same type and value; otherwise, `false`.

`Numeric#fdiv(number) #=> Float`
Returns the result of dividing—using floating-point division—the receiver by *number*. Aliased by `Numeric#quo`.

`Numeric#floor() #=> Integer`
Converts the receiver to a `Float` then returns `Float#round`.

`Numeric#i() #=> Complex`
Returns a new `Complex` whose real part is `0`, and imaginary part is the receiver.

`Numeric#image() #=> 0`
Returns `0`. Aliased by `Numeric#imaginary`.

`Numeric#imaginary() #=> 0`
Aliases `Numeric#image`.

`Numeric#integer?() #=> true or false`
Returns `true` if the receiver is an `Integer`, or a subclass thereof; otherwise, `false`.

`Numeric#magnitude() #=> Numeric`
Aliases `Numeric#abs`.

`Numeric#modulo(number) #=> Numeric`
Aliases `Numeric#%`.

`Numeric#nonzero?() #=> Numeric or nil`
Returns the receiver if non-zero; otherwise, `nil`.

`Numeric#numerator() #=> Integer`
Converts the receiver to a `Rational` then returns `Rational#numerator`.

`Numeric#phase() #=> Numeric`
Aliases `Numeric#angle`.

`Numeric#polar() #=> Array`
Returns an `Array` whose first element is the absolute value of the receiver, using `#abs`, and last element is the arg of the receiver, using `#arg`.

`Numeric#quo(number) #=> Numeric`
Converts the receiver to a `Rational` then divides it, using `Rational#/`, by *number*, returning the result.

`Numeric#real() #=> Numeric`
Returns the receiver.

`Numeric#real?() #=> true`
Returns `true`.

`Numeric#rect() #=> Array`
Returns an `Array` whose first element is the receiver, and last element `0`. Aliased by `Numeric#rectangular`.

`Numeric#rectangular() #=> Array`
Aliases `Numeric#rect`.

`Numeric#remainder(number) #=> Numeric`
Computes the modulo of the receiver and *number* using `#modulo`. Returns the modulo minus *number* if the receiver and *number* have different signs; otherwise, the modulo.

`Numeric#round() #=> Integer`
Converts the receiver to a `Float`then returns `Float#round`.

`Numeric#step(stop, step) {|n| } #=> Numeric or Enumerator`
Yields each number from the receiver to the `Numeric` *stop*, incrementing by *step* with `#+`. If *step* is positive, counts up from the receiver until `#>` than *end*; otherwise, counts down until `#<` than *end*. Returns an `Enumerator` if the block is omitted.

`Numeric#to_c() #=> Complex`
Returns a new `Complex` number whose real part is the receiver, and imaginary part is `0`.

`Numeric#to_int() #=> Integer`
Converts the receiver to an `Integer` using `#to_i`.

`Numeric#truncate() #=> Integer`
Converts the receiver to a `Float` then returns `Float#truncate`.

`Numeric#zero?() #=> true or false`
Returns `true` if the receiver has a zero value; otherwise, `false`.

OBJECT

Defines no methods of its own, but mixes in the `Kernel` `Module`.

OBJECTSPACE

`ObjectSpace._id2ref(id) #=> Object`
Converts an `Integer` object ID, *id*, to the corresponding `Object`.

`ObjectSpace.count_objects() #=> Hash`
Returns a `Hash` mapping each internal object type to the number of objects having that type. See Listing and Counting for details.

`ObjectSpace.define_finalizer(object, proc) #=> Array`
Arranges for the `Proc`, *proc*, to be invoked just prior to *object* being garbage collected. *proc* is passed *object*'s object ID, a `Fixnum`, as an argument.

`ObjectSpace.each_object(`*module*`) {|`*object*`| } #=> Integer or Enumerator`
Yields each living, non-immediate object in the current process. If a `Class` is given for *module*, only objects with that class, or a subclass thereof, are selected; if a `Module` is given, only objects that include that module are selected. If the block is given, objects are yielded to it, then their count returned; otherwise, an `Enumerator` is returned.

`ObjectSpace.garbage_collect() #=> nil`
Starts the garbage collector.

`ObjectSpace.undefine_finalizer(object) #=> Object`
Removes any finalizers that were defined for *object*, then returns its argument.

PROC

`Proc.new() { } #=> Proc`
Creates and returns a new `Proc` object for the given block. If the block is omitted, must be called within a method that has a block parameter: the block passed to the method becomes the body of the `Proc`.

`Proc#[](`*argument*`, …) #=> Object`
Invokes the block associated with the receiver, passing in any *argument*s as block parameters. Returns the value of this block. Aliased by `Proc#call` and `Proc#yield`.

`Proc#==(object) #=> true or false`
Returns `true` if *object* is a `Proc` identical to the receiver; otherwise, `false`.

`Proc#===(object) #=> Object`
Returns the result of invoking the receiver with *object* as an argument.

`Proc#arity() #=> Integer`
Returns the arity of the receiver. A receiver that requires exactly *n* arguments has an arity of *n*. If it also accepts optional arguments, its arity is `-(`*n* `+ 1)`.

`Proc#binding() #=> Binding`
Returns the binding associated with the receiver.

`Proc#call(`*argument*`, …) #=> Object`
Aliases `Proc#[]`.

`Proc#curry() #=> Proc`
Returns a curried version of the receiver. If a curried `Proc` receives as many arguments as it expects, it behaves as the receiver. If it is called with fewer arguments than it expects, it returns a new curried `Proc`, *p*, with each argument it did receive already assigned to the corresponding block parameter. Therefore, *p* expects the remaining arguments, which it will assign to the remaining block parameters.

`Proc#lambda?() #=> true or false`
Returns `true` if the receiver has lambda semantics; otherwise, `false`.

`Proc#parameters() #=> Array`
Returns an `Array`, each element of which describes a parameter accepted by the receiver as an `Array` of the form `[`*`type`*`,` *`name`*`]`. *type* is `:req` if the parameter is required, `:opt` if the parameter is optional, `:rest` if the parameter accepts a variable number of arguments, or `:block` if the parameter expects a block literal. *name* is the parameter's name as a `Symbol`.

`Proc#source_location() #=> Array or nil`
Returns the filename and line number of the source file in which the receiver was defined, or `nil` if it was defined in C.

`Proc#to_proc() #=> Proc`
Returns the receiver.

`Proc#to_s() #=> String`
Returns a `String` specifying the object ID and source location of the receiver.

`Proc#yield(`*`argument`*`, …) #=> Object`
Aliases `Proc#[]`.

PROCESS

`Process.abort(`*`message`*`) #=> N/A`
Aliases `Kernel.abort`.

`Process.daemon(`*`keep_directory=false`*`, `*`keep_stdio_open=false`*`) #=> 0`
Detaches the current process from its controlling terminal and runs it in the background. The working directory of the process is changed to `/` unless *keep_directory* is *true*; otherwise, the working directory is unchanged. The standard output, input, and error streams are redirected to `/dev/null` unless *keep_stdio_open* is `true`. Uses the `daemon(3)` function, if available, or forks then calls `Process.setssid`. If neither approach is supported by the current platform, a `NotImplementedError` is raised.

`Process.detach(pid) #=> Thread`
Creates and returns a `Thread` that monitors the process with given PID, and reaps it when it terminates.

`Process.egid() #=> Integer`
Returns the effective group ID for the current process.

`Process.egid=(egid) #=> Integer`
Sets the effective group ID for the current process to *egid*, which it then returns. Raises `NotImplementedError` on platforms lacking `setresgid(2)`, `setregid(2)`, `setegid(3)`, and `setgid(2)`.

`Process.euid() #=> Integer`
Returns the effective user ID for the current process.

`Process.euid=(euid) #=> Integer`
Sets the effective user ID for the current process to *euid*, which it then returns. Raises `NotImplementedError` on platforms lacking `setresuid(2)`, `setreuid(2)`, `seteuid(3)`, and `setuid(2)`.

`Process.exec(`*`environment={}`*`, command, `*`argument`*`, …, `*`options={}`*`) #=> N/A`
Aliases `Kernel.exec`.

`Process.exit(`*`status=1`*`) #=> N/A`
Aliases `Kernel.exit`.

`Process.exit!(`*`status=1`*`) #=> N/A`
Aliases `Kernel.exit!`.

`Process.fork() { } #=> Integer or nil`
Aliases `Kernel.fork`.

`Process.getpgid(pid) #=> Integer`
Returns the process group ID for the process with the given PID. Raises `NotImplementedError` on platforms lacking `getpgid(2)`.

`Process.getpgrp() #=> Integer`
Returns the process group ID for the current process. Raises `NotImplementedError` on platforms lacking both `getpgrp(2)` and `getpgid(2)`.

`Process.getpriority(which, who) #=> Integer`
Returns the scheduling priority of the process, process group or user, as indicated by *which* and *who*. *which* is one of the following constants: `Process::PRIO_PROCESS`, `Process::PRIO_PGRP`, and `Process::PRIO_USER`. In the first case, *who* is a PID; in the second, a PGID; and in the third, a UID. If *who* is `0` it refers to the current process, the process group of the current process, or the real UID of the current process, respectively. Raises `NotImplementedError` on platforms lacking `getpriority(2)`.

`Process.getrlimit(resource) #=> Array`
Returns the soft and hard limit for the resource identified by *resource*. See Resource Limits for details. Raises `NotImplementedError` on platforms lacking `getrlimit(2)`.

`Process.gid() #=> Integer`
Returns the group ID for the current process.

`Process.gid=(gid) #=> Integer`
Sets the group ID for the current process to *gid*, which it then returns. Raises `NotImplementedError` on platforms lacking `setresgid(2)`, `setregid(2)`, `setrgid(3)`, and `setgid(2)`.

`Process.groups() #=> Array`
Returns the supplementary group IDs of the current process as an `Array` of `Integers`. Raises `NotImplementedError` on platforms without `getgroups(2)`.

`Process.groups=(groups) #=> Array`
Sets the supplementary group IDs of the current process to the given `Array`. *groups* may specify groups by GID, as a `Fixnum`, or name, as a `String`. Returns the new value of `Process.groups`. Raises `NotImplementedError` on platforms without `setgroups(2)`.

`Process.initgroups(user, group) #=> Array`
Initialises the group access list with all groups of which the named user is a member, plus the GID, *group*. Returns the new value of `Process.groups`. Raises `NotImplementedError` on platforms without `initgroups(2)`.

`Process.kill(signal, pid, …) #=> Integer`
Sends the signal identified by *signal* to each process identified by the PID *pid*, where *signal* is the name (as a `Symbol` or `String`) or number (as a `Fixnum`) of a POSIX signal. If *signal* is negative or its name begins with `-`, its process group is killed instead.

`Process.maxgroups() #=> Integer`
Returns the maximum number of supplementary groups handled by `Process.groups` and `Process.groups=`.

`Process.maxgroups=(max) #=> Integer`
Sets the maximum number of supplementary groups handled by `Process.groups` and `Process.groups=` to the lowest value of *max* and `4096`.

`Process.pid() #=> Integer`
Returns the process ID for the current process.

`Process.pid() #=> Integer`
Returns the process ID for the current process.

`Process.ppid() #=> Integer`
Returns the process ID for the current process's parent, or `0` on Windows.

`Process.setpgid(pid, pgid) #=> 0`
Sets the process group ID of the process identified by the given PID to *pgid*. A *pid* of `0` refers to the current process; a *pgid* of `0` sets the process group of the process specified by *pid* to its PID. Raises an `NotImplementedError` on platforms lacking `setpgid(2)`.

`Process.setgrp() #=> 0`
Sets the process group of the current process to its PID. Raises an `NotImplementedError` on platforms lacking both `setpgid(2)` and `setpgrp(3)`.

`Process.setpriority(which, who, priority) #=> 0`
Sets the scheduling priority of the process, process group or user, as indicated by *which* and *who*, to *priority*. *which* is one of the following constants: `Process::PRIO_PROCESS`, `Process::PRIO_PGRP`, and `Process::PRIO_USER`. In the first case, *who* is a PID; in the second, a PGID; and in the third, a UID. If *who* is `0` it refers to the current process, the process group of the current process, or the real UID of the current process, respectively. Raises `NotImplementedError` on platforms lacking `getpriority(2)`.

`Process.setrlimit(which, soft, hard=soft) #=> 0`
Sets the soft and hard limits of the resource identified by *which* to *soft* and *hard*, respectively. Raises `NotImplementedError` on platforms lacking `setrlimit(2)`. See Resource Limits for details.

`Process.setsid() #=> Integer`
Creates a new session with the current process as its sole member—i.e. makes the current process a process group leader. If the current process is already a process group leader, does nothing. Returns the session ID. Raises `NotImplementedError` on systems lacking either `setsid(2)` or both `setpgrp(3)` and `TIOCNOTTY`.

`Process.spawn(environment={}, command, argument, …, options={}) #=> Fixnum`
Aliases `Kernel.spawn`.

`Process.times() #=> Struct::Tms`
Returns the user and system CPU times for the current process. Raises `NotImplementedError` on systems lacking `times(2)`.

`Process.uid() #=> Integer`
Returns the user ID of the current process.

`Process.uid=(uid) #=> Integer`
Sets the user ID of the current process to the given `Integer`, which it then returns. Raises `NotImplementedError` on platforms lacking `setresuid(2)`, `setreuid(2)`, `setruid(3)`, and `setuid(2)`.

`Process.wait(`*`pid=-1`*`, `*`flags=0`*`) #=> Integer`
Suspends the calling process until one of its children exit. A *pid* less than `-1` refers to the child process identified by the absolute value of *pid*; a *pid* of `-1` refers to any child; a *pid* of `0` refers to any child whose PGID is that of the current process; and a positive *pid* refers to the child with a PID of *pid*. *flags* is a either `0` or a logical OR of `Process::WNOHANG` (don't block unless there is a matching child process) and `Process::WUNTRACED` (return stopped children that haven't previously been reported). Raises a `SystemError` if there are no child processes. Aliased by `Process.waitpid`.

`Process.waitall() #=> Array`
Waits for all child processes to terminate. Returns an `Array` whose elements have the form `[`*`pid`*`, `*`status`*`]`, where *pid* is the PID of a child, and *status* the corresponding `Process::Status` object.

`Process.wait2(`*`pid=-1`*`, `*`flags=0`*`) #=> Array`
Behaves as `Process.wait` but returns an `Array` containing the PID of the child process and the corresponding `Process::Status` object. Aliased by `Process.waitpid2`.

`Process.waitpid(`*`pid=-1`*`, `*`flags=0`*`) #=> Integer`
Aliases `Process.wait`.

`Process.waitpid2(`*`pid=-1`*`, `*`flags=0`*`) #=> Array`
Aliases `Process.wait2`.

PROCESS::GID

```
Process::GID.change_privilege(gid) #=> Integer
```
Sets the real, effective, and saved group IDs of the current process to the given `Integer`, which it then returns.

```
Process::GID.eid() #=> Integer
```
Aliases `Process.egid`.

```
Process::GID.eid=(eid) #=> Integer
```
Sets the effective group ID of the current process to the given `Integer`. If possible, also sets the saved group ID to the same value. Returns *eid*. Aliased by `Process::GID.grant_privilege`.

```
Process::GID.grant_privilege(eid) #=> Integer
```
Aliases `Process::GID.eid=`.

```
Process::GID.re_exchange() #=> Integer
```
Swaps the real and effective group IDs. Sets the saved group ID to the new effective group ID, which it then returns.

```
Process::GID.re_exchangeable?() #=> true or false
```
Returns `true` if this platform supports exchanging the real and effective group IDs; otherwise, `false`.

```
Process::GID.rid() #=> Integer
```
Aliases `Process.gid`.

```
Process::GID.sid_available?() #=> true or false
```
Returns `true` if this platform supports saved group IDs; otherwise, `false`.

```
Process::GID.switch() { } #=> Object
```
Exchanges the effective and real group IDs of the current process, then returns the new effective group ID. If a block is given, the IDs are restored to their original values once it has been yielded to; the return value is that of the block.

PROCESS::STAT

`Process::Status#==(object) #=> true or false`
Returns `true` if *object* is a `Process::Status` object whose `Integer` values are equal; otherwise, `false`.

`Process::Status#&(number) #=> Fixnum`
Performs a logical AND between the receiver's `Integer` value and the given `Integer`.

`Process::Status#>>(number) #=> Fixnum`
Right shifts the bits in the receiver by *number* places.

`Process::Status#coredump?() #=> true or false`
Returns `true` if the associated process generated a coredump; otherwise, `false`.

`Process::Status#exited?() #=> true or false`
Returns `true` if the associated process exited normally; otherwise, `false`.

`Process::Status#exitstatus() #=> Fixnum`
Returns least significant byte of the receiver's `Integer` value, or `nil` if the associated process exited abnormally.

`Process::Status#pid() #=> Fixnum`
Returns the PID of the associated process.

`Process::Status#signaled?() #=> true or false`
Returns `true` if the associated process terminated because of an uncaught signal; otherwise, `false`.

`Process::Status#stopped?() #=> true or false`
Returns `true` if the associated process is stopped; otherwise, `false`.

`Process::Status#success?() #=> true, false, or nil`
Returns `true` if the associated process exited successfully; `false` if it exited abnormally; and `nil` if it didn't exit.

`Process::Status#stopsig() #=> Fixnum or nil`
Returns the number of the signal that caused the associated process to stop, or `nil` if it isn't stopped.

`Process::Status#termsig() #=> Fixnum or nil`
Returns the number of the signal that caused the associated process to terminate, or `nil` if it didn't terminate due to an uncaught signal.

`Process::Status#to_i() #=> Fixnum`
Returns the `Integer` value of this status.

`Process::Status#to_s() #=> String`
Returns the `Integer` value of this status as a `String`.

PROCESS::SYS

```
Process.getegid() #=> Integer
```
Aliases `Process.egid`.

```
Process.geteuid() #=> Integer
```
Aliases `Process.euid`.

```
Process.getgid() #=> Integer
```
Aliases `Process.gid`.

```
Process.getuid() #=> Integer
```
Aliases `Process.uid`.

```
Process.issetugid() #=> true or false
```
Returns `true` if the current process environment or memory address space is considered tainted by UID or GID changes; otherwise, `false`. Raises `NotImplementedError` on platforms that lack `issetugid(2)`.

```
Process.setegid(gid) #=> nil
```
Sets the effective group ID of the current process to the given `Integer`. Raises `NotImplementedError` on platforms that lack `setegid(2)`.

```
Process.seteuid(uid) #=> nil
```
Sets the effective user ID of the current process to the given `Integer`. Raises `NotImplementedError` on platforms that lack `seteuid(2)`.

```
Process.setregid(real, effective) #=> nil
```
Sets the real and effective group IDs of the current process to *real* and *effective*, respectively. Raises `NotImplementedError` on platforms that lack `setregid(2)`.

```
Process.setresgid(real, effective, saved) #=> nil
```
Sets the real, effective, and saved group IDs of the current process to *real*, *effective*, and *saved* respectively. Raises `NotImplementedError` on platforms that lack `setresgid(2)`.

```
Process.setresuid(real, effective, saved) #=> nil
```
Sets the real, effective, and saved user IDs of the current process to *real*, *effective*, and *saved* respectively. Raises `NotImplementedError` on platforms that lack `setresuid(2)`.

```
Process.setreuid(real, effective) #=> nil
```
Sets the real and effective user IDs of the current process to *real* and *effective*, respectively. Raises `NotImplementedError` on platforms that lack `setreuid(2)`.

```
Process.setrgid(real) #=> nil
```
Sets the real group ID of the current process to the given `Integer`. Raises `NotImplementedError` on platforms that lack `setrgid(2)`.

```
Process.setruid(real) #=> nil
```
Sets the real user ID of the current process to the given `Integer`. Raises `NotImplementedError` on platforms that lack `setruid(2)`.

PROCESS::UID

```
Process::UID.change_privilege(uid) #=> Integer
```
Sets the real, effective, and saved user IDs of the current process to the given `Integer`, which it then returns.

```
Process::UID.eid() #=> Integer
```
Aliases `Process.euid`.

```
Process::UID.eid=(eid) #=> Integer
```
Sets the effective user ID of the current process to the given `Integer`. If possible, also sets the saved user ID to the same value. Returns *eid*. Aliased by `Process::UID.grant_privilege`.

```
Process::UID.grant_privilege(eid) #=> Integer
```
Aliases `Process::UID.eid=`.

```
Process::UID.re_exchange() #=> Integer
```
Swaps the real and effective user IDs. Sets the saved user ID to the new effective user ID, which it then returns.

```
Process::UID.re_exchangeable?() #=> true or false
```
Returns `true` if this platform supports exchanging the real and effective user IDs; otherwise, `false`.

```
Process::UID.rid() #=> Integer
```
Aliases `Process.uid`.

```
Process::UID.sid_available?() #=> true or false
```
Returns `true` if this platform supports saved user IDs; otherwise, `false`.

```
Process::UID.switch() { } #=> Object
```
Exchanges the effective and real user IDs of the current process, then returns the new effective user ID. If a block is given, the IDs are restored to their original values once it has been yielded to; the return value is that of the block.

RANGE

`Range.new(begin, end, exclusive) #=> Range`
Creates and returns a `Range` of *begin* to *end*, inclusive. If *exclusive* is `true`, *end* is omitted.

`Range#==(object) #=> true or false`
Returns `true` is a `Range` whose beginning and end values are `#==` to the corresponding values in the receiver, and they are both either exclusive or inclusive; otherwise, `false`.

`Range#===(object) #=> true or false`
Returns the value of `Range#include?` when passed *object*.

`Range#begin() #=> Object`
Returns the beginning value of the receiver.

`Range#cover?(object) #=> true or false`
Returns `true` if *object* is between—using the comparison operators—the beginning and end value of the receiver. If the receiver is inclusive, *object* may equal the end value; otherwise, it may not.

`Range#each() {|value| } #=> Range`
Generates successive elements of the receiver with `#succ`, yielding them to the block. If the block is omitted, an `Enumerator` is returned.

`Range#end() #=> Object`
Returns the end value of the receiver.

`Range#eql?(object) #=> true or false`
Returns `true` is a `Range` whose beginning and end values are `#eql?` to the corresponding values in the receiver, and they are both either exclusive or inclusive; otherwise, `false`.

`Range#exclude_end?() #=> true or false`
Return `true` if the receiver is exclusive; otherwise, `false`.

`Range#first(`*n=1*`) #=> Object or Array`

Returns the first *n* elements of the receiver as an `Array`. If *n* is omitted, returns the first element itself.

`Range#include?(object) #=> true or false`

If the endpoints of the receiver are `Numeric`, behaves as `Range#cover?`. Otherwise, uses `Enumerable#include?` to determine whether *object* is an element of the receiver, returning `true` if it is; otherwise, `false`. Aliased by `Range#member?`.

`Range#last(`*n=1*`) #=> Object or Array`

Returns the last *n* elements of the receiver as an `Array`. If *n* is omitted, returns the last element itself.

`Range#max() {|a,b| } #=> Object or nil`

Returns the largest element of the receiver, or `nil` if the beginning value exceeds the end value. If a block is given it is used to find the maximum value: it is passed each pair of elements in turn, and expected to behave like `#<=>`.

`Range#member?(object) #=> true or false`

Aliases `Range#include?`.

`Range#min() {|a,b| } #=> Object or nil`

Returns the smallest element of the receiver, or `nil` if the beginning value exceeds the end value. If a block is given it is used to find the maximum value: it is passed each pair of elements in turn, and expected to behave like `#<=>`.

`Range#step(`*n=1*`) {|value| } #=> Range`

Generates successive elements of the receiver with `#succ`, or if the endpoints are numeric, `#+`, yielding every n^{th} to the block. If the block is omitted, an `Enumerator` is returned.

RATIONAL

`Rational#*(number) #=> Numeric`
Returns the result of multiplying *number* with the receiver.

`Rational#**(number) #=> Numeric`
Returns the result of raising the receiver to the $number^{th}$ power.

`Rational#+(number) #=> Numeric`
Returns the result of adding the receiver to *number*.

`Rational#-(number) #=> Numeric`
Returns the result of subtracting *number* from the receiver.

`Rational#/(number) #=> Numeric`
Returns the result of dividing—using rational division if possible; otherwise, floating-point division—the receiver by *number*. Aliased by `Numeric#div` and `Rational#quo`.

`Rational#<=>(number) #=> -1, 0, 1`
Returns `-1` if the receiver is less than *number*, `0` if they are equal, and `1` if it is greater.

`Rational#==(number) #=> true or false`
Returns `true` if the *number* is a `Numeric` with the same value as the receiver; `false`, otherwise. If *number* is a `Float`, the receiver is coerced into a `Float` prior to the comparison.

`Rational#ceil(precision=0) #=> Integer or Rational`
Returns the smallest `Integer` greater than or equal to the receiver. If *precision* is given, returns the receiver rounded toward positive infinity: if *precision* is positive, it specifies the number of digits following the decimal point; otherwise, it specifies the number of digits preceding the decimal point.

`Rational#coerce(object) #=> Array`
Returns an `Array` whose first element is *object*, and second, the receiver—both

coerced to `Numeric`s of the same class. If *object* is an `Integer`, it is converted to a `Rational`; if it is a `Float`, both are converted to `Float`s; if it is a `Complex` without an imaginary part, it is converted to a `Rational` whose numerator equals the real part, and whose denominator is `1`; otherwise, a `TypeError` is raised.

`Rational#denominator() #=> Numeric`
Returns the denominator of the receiver.

`Rational#fdiv(number) #=> Float`
Returns the result of dividing—using floating-point division—the receiver by *number*. Aliased by `Rational#quo`.

`Rational#floor(`*`precision=0`*`) #=> Integer or Rational`
Returns the largest `Integer` less than or equal to the receiver. If *precision* is given, returns the receiver rounded toward negative infinity: if *precision* is positive, it specifies the number of digits following the decimal point; otherwise, it specifies the number of digits preceding the decimal point.

`Rational#numerator() #=> Numeric`
Returns the numerator of the receiver.

`Rational#quo(number) #=> Numeric`
Aliases `Rational#/`.

`Rational#rationalize(`*`epsilon`*`) #=> Rational`
Returns the simplest rational number differing from the receiver by no more than the absolute value of *epsilon*. If *epsilon* is omitted, returns the receiver.

`Rational#round(`*`precision=0`*`) #=> Integer or Rational`
Returns the `Integer` nearest to the receiver: rounding upwards if there's a tie. If *precision* is positive, it specifies the number of digits following the decimal point; otherwise, it specifies the number of digits preceding the decimal point.

`Rational#to_f() #=> Float`
Converts the receiver to a `Float`.

`Rational#to_i() #=> Integer`
Returns the receiver truncated to an `Integer`.

`Rational#to_r() #=> Rational`
Returns the receiver.

`Rational#to_s() #=> String`
Returns a `String` comprising the numerator and denominator of the receiver, separated by a solidus.

`Rational#truncate(`*`precision=0`*`) #=> Integer or Rational`
Returns the receiver truncated toward zero. If *precision* is given, the result is a `Rational`; otherwise, it is an `Integer`. When *precision* is negative, it specifies the number of digits preceding the decimal point; otherwise, it specifies the number of digits following the decimal point.

REGEXP

`Regexp.compile(pattern, options, encoding) #=> Regexp`
`Regexp.compile(regexp) #=> Regexp`
Creates and returns a new `Regexp`. *pattern* is a regular expression as a `String`. If *options* is a `Fixnum` it is the bitwise-OR of one or more of the following constants: `Regexp::EXTENDED`, `Regexp::IGNORECASE`, and `Regexp::MULTILINE`; if it is `nil`, it is equivalent to `Regexp::IGNORECASE`. The `Regexp` will have the encoding of *pattern*, unless *encoding* is `"n"` or `"N"`, in which case it will have the ASCII-8BIT encoding. In the second form, a new `Regexp` is created from the given `Regexp`, inheriting its options. Aliased by `Regexp.new`.

`Regexp.escape(string) #=> String`
Returns the given `String` with all `Regexp` metacharacters escaped. Aliased by `Regexp.quote`.

`Regexp.last_match(capture) #=> MatchData`
Returns the `MatchData` object representing the last successful match; equivalent to `$~`. If *capture* is given, returns the text corresponding to the specified capture group: a `Fixnum` specifies a numbered group, and a `Symbol` specifies a named group.

`Regexp.new(pattern, options, encoding) #=> Regexp`
`Regexp.new(regexp) #=> Regexp`
Aliases `Regexp.compile`.

`Regexp.quote(string) #=> String`
Aliases `Regexp.escape`.

`Regexp.try_convert(object) #=> Regexp or nil`
Returns *object* if it's a `Regexp`, otherwise tries to convert it to one with `#to_regexp`. If this approach fails, returns `nil`.

`Regexp.union(pattern, …) #=> Regexp`
`Regexp.union(patterns) #=> Regexp`
Returns a `Regexp` that matches any of the given *patterns*. If *pattern* is a

`String`, its metacharacters are escaped; if it's a `Regexp`, it corresponds to a group matching the same pattern with the same options. If no *pattern*s are given, `/(?!)/`—a `Regexp` that will never match—is returned. In the second form, the *pattern*s are given as an `Array`.

`Regexp#==(regexp) #=> true or false`
Returns `true` if *regexp* is a `Regexp` with the same pattern, encoding, and *case-fold* setting as the receiver; otherwise, `false`. Aliased by `Regexp#eql?`.

`Regexp#===(object) #=> true or false`
Returns `true` if *object* is a `String` that matches the receiver; otherwise, `false`.

`Regexp#=~(string) #=> Integer or nil`
If the given `String` matches the receiver, sets `$~` to the corresponding `MatchData` object, then returns the offset in *string* where the match begins; otherwise, returns `nil`.

`Regexp#~() #=> Integer or nil`
Behaves as `Regexp#=~`, but matches the receiver against `$_`.

`Regexp#casefold?() #=> true or false`
Returns `true` if the receiver is case-insensitive; otherwise, `false`.

`Regexp#encoding() #=> Encoding`
Returns the `Encoding` associated with the receiver.

`Regexp#eql?(regexp) #=> true or false`
Aliases `Regexp#==`.

`Regexp#fixed_encoding?() #=> true or false`
Returns `true` if the receiver contains non-ASCII characters; otherwise, `false`.

`Regexp#match(string, offset=0) #=> MatchData or nil`
`Regexp#match(string, offset=0) {|matchdata| } #=> Object or nil`
Matches the receiver against the `String` *string*, starting from the *offset*th character. If the match fails, `nil` is returned. Otherwise, the first form returns the `MatchData` object, and the second form yields the `MatchData` object to the block then returns the block's value.

`Regexp#named_captures() #=> Hash`

Assuming each named capture group has an integer index, with the first being `1`, returns a `Hash` associating each unique name with an `Array` of indices corresponding to the groups that capture it.

`Regexp#names() #=> Array`

Returns the unique names corresponding to each named capture group as an `Array` of `Strings`.

`Regexp#options() #=> Integer`

Returns the bitwise-OR of the options with which the receiver was created.

`Regexp#source() #=> String`

Returns the receiver's pattern.

`Regexp#to_s() #=> String`

Returns the receiver as a `String`, i.e. the pattern and the associated objects in a form that re-compiles to a semantically identical `Regexp`.

SIGNAL

```
Signal.list() #=> Hash
```

Returns a `Hash` whose keys are signal names, and values the corresponding signal numbers.

```
Signal.trap(signal, command) #=> Object
Signal.trap(signal) {|signal_number| } #=> Object
```

Registers a signal handler for the signal identified by *signal*, where *signal* is a signal name—as a `String` or `Symbol`—or a signal number. If *signal* is a `Proc`, or a block is given, they are called with the signal number as their argument. If *signal* is `nil`, `""`, or `"IGNORE"`, the signal is ignored. If *signal* is `"DEFAULT"`, the operating system's default handler will be used. If *signal* is `"EXIT"`, the signal will terminate the interpreter. Returns the previous handler for this signal.

STRING

`String.new(`*object*`) #=> String`
Initialises and returns a new `String`. If *object* is given, uses `String#replace` to substitute *object* for the receiver.

`String.try_convert(object) #=> String or nil`
Attempts to convert *object* to a `String`—if it isn't one already, uses `#to_str`—which it returns. Returns `nil` if the conversion failed.

`String#%(object) #=> String`
Formats the receiver as with `Kernel.sprintf`, using *object* as the value(s) to interpolate. If *object* is an `Array`, its values are interpolated, instead.

`String#*(n) #=> String`
Returns a new `String` comprising *n* copies of the receiver, where *n* is a `Numeric` truncated to an `Integer`.

`String#+(string) #=> String`
Returns a new `String` comprising the receiver concatenated with the given `String`.

`String#<<(object) #=> String`
Concatenates the receiver with *object* in-place, returning the new receiver. If *object* is a `Fixnum` it is interpreted as a codepoint in the receiver's encoding, and converted accordingly. Aliased by `String#concat`.

`String#<=>(object) #=> -1, 0, 1, or nil`
Returns `-1` if the receiver is less than, `0` if it is equal to, and `1` if it is greater than *object*. If *object* responds to both `#to_str` and `<=>`, the `#<=>` method of *object* is used to compare it with the receiver, then the negation of the result is returned; otherwise, `nil` is returned.

`String#==(object) #=> true or false`
Returns `true` if *object* is a `String` with the same `Encoding`, length, and content as the receiver. If *object* responds to `#to_str`, returns the result of

calling *object*'s #== with the receiver as the argument. In all other cases, returns false.

```
String#=~(object) #=> Integer or nil
```

Returns the result of calling *object*'s #=~ method with the receiver as the argument. If *object* is a Regexp, #=~ is not actually called, for performance reasons, but the semantics are identical. If *object* is a String, raises a TypeError.

```
String#[](offset, length) #=> String or nil
String#[](range) #=> String or nil
String#[](regexp, group) #=> String or nil
String#[](string) #=> String or nil
```

The first form returns the portion of the receiver which begins at the given Fixnum *offset* and extends to either the last character or, if a Fixnum *length* is given, the character *length* characters after *offset*. The second form selects the characters from the receiver whose offsets are covered by the given Range, returning them as a String. In both cases, negative offsets count from the end of the receiver. The third form matches the receiver against the given Regexp, returning either the text that matched or, if *group* identifies a capturing group by either a Symbol name or Fixnum number, the text captured by that group. In the last form, the given String is returned if it occurs in the receiver. In all cases, nil is returned if there was no matching sub-String. Aliased by String#slice.

```
String#[]=(offset, length, replacement) #=> String or nil
String#[]=(range, replacement) #=> String or nil
String#[]=(regexp, group, replacement) #=> String or nil
String#[]=(string, replacement) #=> String or nil
```

Replaces a sub-String of the receiver with the given *replacement* String, which it then returns. The first form replaces the portion of the receiver which begins at the given Fixnum *offset* and extends to either the last character or, if a Fixnum *length* is given, the character *length* characters after *offset*. If this sub-String doesn't exist, an IndexError is raised. The second form replaces the characters from the receiver whose offsets are covered by the given Range, raising a RangeError if there is no such sub-String. In both cases, negative offsets count from the end of the receiver. The third form matches the receiver against the given Regexp, replacing either the text that

matched or, if *group* identifies a capturing group by either a `Symbol` name or `Fixnum` number, the text captured by that group. If the `Regexp` didn't match, an `IndexError` is raised. In the last form, replaces the first occurrence of the `String` *string* in the receiver, raising an `IndexError` if it never occurs.

`String#ascii_only?() #=> true or false`
Returns `true` if the receiver has an ASCII-compatible encoding and contains no ASCII characters; otherwise, `false`.

`String#bytes() {|byte| } #=> String or Enumerator`
Yields each byte in the receiver as a `Fixnum`, returning the receiver. If the block is omitted, returns an `Enumerator`. Aliased by `String#each_byte`.

`String#bytesize() #=> Integer`
Returns the number of bytes contained within the receiver.

`String#capitalize() #=> String`
Returns a copy of the receiver with the first character converted to uppercase, and the remainder to lowercase. However, this only affects ASCII-characters; others remain as they are.

`String#capitalize!() #=> String or nil`
Behaves as `String#capitalize` except the receiver is converted in-place. Returns the receiver, or `nil` if it wasn't modified.

`String#casecmp(object) #=> -1, 0, 1, or nil`
Returns `-1` if the receiver is less than, `0` if it is equal to, and `1` if it is greater than *object*. Differences in case of ASCII characters are ignored. Returns `nil` if the encoding of the receiver is incompatible with that of *object*. Raises a `TypeError` if *object* can't be converted to a `String`.

`String#center(length, padding=" ") #=> String`
Returns a new `String` of length *length* with the receiver in the middle, surrounded either side with *padding*. If *length* is less than or equal the length of the receiver, returns the receiver.

`String#chars() {|character| } #=> String or Enumerator`
Yields each character of the receiver as a `String`, returning the receiver. If the block is omitted, returns an `Enumerator`. Aliased by `String#each_char`.

`String#chr() #=> String`
Returns the first character of the receiver, or if the receiver is empty, an empty `String`.

`String#clear() #=> String`
Deletes the contents of the receiver, returning the new receiver.

`String#chomp(remove=$/) #=> String`
Returns a copy of the receiver with the given `String` *remove* deleted from the end. If *remove* has the value `"\n"`, removes the longest of the following, instead: `"\n"`, `"\r"`, and `"\r\n"`.

`String#chomp!(remove=$/) #=> String or nil`
Behaves as `String#chomp`, but modifies the receiver in-place. Returns the receiver, or `nil` if it wasn't modified.

`String#chop() #=> String`
Returns the receiver with the last character removed. If it ends with `"\r\n"`, both characters are removed. If the receiver is empty, it is returned.

`String#chop!() #=> String or nil`
Behaves as `String#chop`, but returns `nil` if the receiver was empty.

`String#codepoints() {|codepoints| } #=> String or Enumerator`
Yields each codepoint of the receiver as a `Fixnum`, returning the receiver. If the block is omitted, returns an `Enumerator`. Aliased by `String#each_codepoint`.

`String#concat(object) #=> String`
Aliases `String#<<`.

`String#count(string, …) #=> Fixnum`
Returns the number of characters contained in the receiver which are specified as arguments. Each argument specifies a set of characters as a `String`: if their first character is a circumflex accent (`"^"`), their contents are negated; if they comprise two characters separated by a hyphen minus sign (`"-"`), they represent the range of characters between the two given. Counts the characters contained in the intersection of these sets.

`String#crypt(salt) #=> String`

Returns a one-way cryptographic hash of the receiver using `crypt(3)`. The *salt* should be at least two characters long and only contain ASCII letters, numbers, `"."` and `/`. Please do not, under any circumstances, use this method to perform cryptography; the algorithm used by `crypt(3)` is a variation of DES which is known to be insecure.

`String#delete(string, …) #=> String`

Returns a copy of the receiver with the characters specified as arguments removed. Each argument specifies a set of characters as a `String`: if their first character is a circumflex accent (`"^"`), their contents are negated; if they comprise two characters separated by a hyphen minus sign (`"-"`), they represent the range of characters between the two given. Deletes the characters appearing in the intersection of these sets.

`String#downcase() #=> String`

Returns a copy of the receiver with its uppercase ASCII characters converted to lowercase.

`String#downcase!() #=> String or nil`

Behaves as `String#downcase`, but modifies the receiver in-place, then returns it. Returns `nil` if no modifications were made.

`String#dump() #=> String`

Returns a copy of the receiver with non-printable ASCII characters and non-ASCII characters replaced by character escapes. The `String` returned will evaluate to the receiver.

`String#each_byte() {|byte| } #=> String or Enumerator`

Aliases `String#bytes`.

`String#each_char() {|character| } #=> String or Enumerator`

Aliases `String#chars`.

`String#each_codepoint() {|codepoints| } #=> String or Enumerator`

Aliases `String#codepoints`.

`String#each_line() {|line| } #=> String or Enumerator`

Aliases `String#lines`.

`String#empty?() #=> true or false`
Returns `true` if the receiver has a size of `0`; otherwise, `false`.

`String#encode(`*`options`*`) #=> String`
`String#encode(target_encoding, `*`source_encoding`*`, `*`options`*`) #=> String`
Transcodes the receiver from one encoding to another, returning the result. The first form transcodes from the receiver's current encoding to the default internal encoding. The second form transcodes from *source_encoding*, or the receiver's current encoding if that argument is omitted, to *destination_encoding*. The first form replaces invalid byte sequences and undefined characters with `"?"`; the others raise `Encoding::InvalidByteSequence` and `Encoding::UndefinedConversionError`, respectively. *options* is an `Encoding` options `Hash`.

`String#encode!(`*`options`*`) #=> String`
`String#encode!(target_encoding, `*`source_encoding`*`, `*`options`*`) #=> String`
Behaves as `String#encode`, but modifies the receiver in-place before returning it.

`String#encoding() #=> Encoding`
Returns the receiver's encoding.

`String#end_with?(suffix, …) #=> true or false`
Returns `true` if the receiver ends with any of the given `String`s; otherwise, `false`.

`String#eql?(object) #=> true or false`
Returns `true` if *object* is a `String` with the same length and content as the receiver; otherwise, `false`.

`String#force_encoding(encoding) #=> String`
Associates the receiver with the given encoding, then returns the receiver. *encoding* may be either an `Encoding` object or an `Encoding` name as a `String`.

`String#getbyte(offset) #=> Integer or nil`
Returns the byte at the given offset in the receiver, or `nil` if there is no such byte. If *offset* is negative it counts from the end of the receiver.

`String#gsub(pattern, replacement) #=> String`
`String#gsub(pattern) {|matchdata| } #=> String or Enumerator`
Replaces all occurrences of *pattern* with *replacement* or the value of the block. *pattern* is either a `Regexp` or a `String`; in the latter case, metacharacters are ignored. If *replacement* is a `String` it may contain references to capture groups as either `\`*`digit`* or `\k<`*`name`*`>`; otherwise, it is a `Hash` whose keys are `String`s containing the text captured by a group, and values the `String`s that they should be replaced with. In the second form, the block is invoked on each match with the corresponding `MatchData` object as a parameter. Returns the result or, if both *replacement* and block are omitted, an `Enumerator`.

`String#gsub!(pattern, replacement) #=> String or nil`
`String#gsub!(pattern) {|matchdata| } #=> String, nil, or Enumerator`
Behaves as `String#gsub` except the receiver is modified in-place. Returns the receiver if it was changed; otherwise, `nil`.

`String#hex() #=> Integer`
Interprets the leading characters of the receiver as a hexadecimal integer, with an optional sign and `0x` prefix, returning the `Integer` equivalent. Returns `0` if no such number was found.

`String#include?(string) #=> true or false`
Returns `true` if the given `String` is contained by the receiver; otherwise, `false`.

`String#index(needle, `*`offset`*`) #=> Integer or nil`
Searches the receiver for the given sub-`String` or `Regexp`, returning the character offset where the first occurrence begins, or `nil` if the search failed. If *offset* is given it is the `Fixnum` character offset in the receiver from which the search begins.

`String#insert(index, string) #=> String`
Inserts the `String` *string* into the receiver before the character at the `Fixnum` *offset*. A negative *offset* counts from the end of the receiver, inserting *string* after this character.

`String#intern() #=> Symbol`
Returns the receiver converted to a `Symbol`.

`String#length() #=> Fixnum`
Returns the number of characters contained in the receiver. Aliased by `String#size`.

`String#lines(separator=$/) {|line| } #=> String or Enumerator`
Yields each line of the receiver that is separated by *separator*, returning an `Enumerator` if the block is omitted. A *separator* of `""` is equivalent to one of `"\n\n"`.

`String#ljust(width, padding=" ") #=> String`
Returns a copy of the receiver *width* characters long, left-justified using *padding* if necessary.

`String#lstrip() #=> String`
Returns a copy of the receiver with leading ASCII-whitespace characters removed.

`String#lstrip!() #=> String or nil`
Behaves as `String#lstrip` but modifies the receiver in-place. Returns the new receiver, or `nil` if it was't changed.

`String#match(pattern) {|matchdata| } #=> MatchData or nil`
Matches the receiver against *pattern*. If successful, returns the corresponding `MatchData` object or, if the block is given, yields it to the block then returns the block's value; otherwise, returns `nil`. *pattern* may be a `String`, in which case its compiled into a `Regexp`, a `Regexp`, or an object which can be converted to one of the aforementioned types.

`String#next() #=> String`
Returns the `String` which succeeds the receiver. Starting with the last alphanumeric character, or the last character if there are no alphanumerics, increments it as follows: a digit produces the successive digit, a letter produces the successive letter, preserving case, and any other character produces the character with the successive codepoint. If the character incremented was the last of its type, e.g. `"z"` or `"9"`, the character to its left is incremented, *ad infinitum*; if there is no character to the left, the new character is appended to the receiver. However, if the receiver matches `/(?<a>\d+)(?<b>)[^\d](?<c>\d+)$/`, whereas the last character of *b* would

normally be incremented, the last character of *a* is, instead—i.e. the two sequences of digits are treated like a number containing a decimal point. Aliased by `String#succ`.

`String#next!() #=> String`

Behaves as `String#next`, but modifies the receiver in-place before returning it. Aliased by `String#succ!`.

`String#oct() #=> Integer`

Interprets the leading characters of the receiver as octal digits prefixed by an optional sign, and returns the `Integer` corresponding to their value.

`String#ord() #=> Integer`

Returns the codepoint of the first character in the receiver.

`String#partition(pattern) #=> Array`

Searches the receiver for the first occurrence of *pattern*. If successful, returns an `Array` whose first element is the portion of the receiver before the match, second element is the portion of the receiver which matched, and last element is the portion of the receiver after the match. Otherwise, returns an `Array` with the receiver as first element, and two empty `String`s as the last two. *pattern* may be either a `String` or `Regexp`.

`String#replace(string) #=> String`

Changes the receiver's contents, encoding, and taintedness to the respective values of the given `String`. Returns the new receiver.

`String#reverse() #=> String`

Returns a copy of the receiver with the characters reversed.

`String#reverse!() #=> String`

Behaves as `String#reverse` but modifies the receiver in-place.

`String#rindex(needle, offset) #=> Integer or nil`

Searches the receiver for the given sub-`String` or `Regexp`, returning the character offset where the last occurrence begins, or `nil` if the search failed. If *offset* is given it is the `Fixnum` character offset in the receiver at which the search ends.

`String#rjust(width, padding= " ") #=> String`
Returns a copy of the receiver *width* characters long, right-justified using *padding* if necessary.

`String#rpartition(pattern) #=> Array`
Searches the receiver for the last occurrence of *pattern*. If successful, returns an `Array` whose first element is the portion of the receiver before the match, second element is the portion of the receiver which matched, and last element is the portion of the receiver after the match. Otherwise, returns an `Array` whose first two elements are empty `String`s, and last element is the receiver. *pattern* may be either a `String` or `Regexp`.

`String#rstrip() #=> String`
Returns a copy of the receiver with trailing ASCII-whitespace characters removed.

`String#rstrip!() #=> String or nil`
Behaves as `String#rstrip` but modifies the receiver in-place. Returns the new receiver, or `nil` if it was't changed.

`String#scan(pattern) {|match| } #=> Array or String`
Searches the receiver for *pattern* yielding the sub-`String` matched or, if the *pattern* contains capturing groups, the text matched by each group as an `Array` of `String`s. If the block is omitted, the values that would have been yielded are returned as an `Array`. *pattern* may be a `Regexp` or a `String`; in the latter case, metacharacters that it contains are ignored.

`String#setbyte(n, byte) #=> Integer`
Replaces the n^{th} byte of the receiver with the `Fixnum` *byte*, returning *byte*. A negative *n* counts from the end of the receiver, and if *n* falls outside the receiver an `IndexError` is raised.

`String#size() #=> Fixnum`
Aliases `String#length`.

`String#slice(offset, length) #=> String or nil`
`String#slice(range) #=> String or nil`
`String#slice(regexp, group) #=> String or nil`

`String#slice(string) #=> String or nil`
Aliases `String#[]`.

`String#slice!(offset, length) #=> String or nil`
`String#slice!(range) #=> String or nil`
`String#slice!(regexp, group) #=> String or nil`
`String#slice!(string) #=> String or nil`
Behaves as `String#[]`, but deletes and returns the matching portion of the receiver. If the receiver wasn't modified, `nil` is returned.

`String#split(pattern=$;, limit) #=> Array`
Divides the receiver into an `Array` of `String` fields, each a run of consecutive characters up to, but excluding, *pattern*. A delimiter of `nil` or `" "`, splits on consecutive whitespace. Any other `String` *pattern* is interpreted literally. When *pattern* is a `Regexp`, the delimiter is the matching text, however text matched by a capturing group is included in the result as its own field. If *pattern* matches `""`, each character of the receiver is an field. Unless *limit* is negative, trailing empty fields are dropped. If *limit* is positive, it specifies the maximum number of elements in the result; if `1`, the receiver is the sole element in the result.

`String#squeeze(set, …) #=> String`
Returns a copy of the receiver in which runs of the same character are replaced by one of that character. If arguments are given, only runs of the characters they specify are collapsed in this way. Each argument specifies a set of characters as a `String`: if their first character is a circumflex accent (`"^"`), their contents are negated; if they comprise two characters separated by a hyphen minus sign (`"-"`), they represent the range of characters between the two given.

`String#squeeze!(set, …) #=> String or nil`
Behaves as `String#squeeze`, but modifies the receiver in-place. Returns the receiver if actually modified; otherwise, `nil`.

`String#start_with?(prefix, …) #=> true or false`
Returns `true` if the receiver begins with any of the given `Strings`; otherwise, `false`.

`String#strip() #=> String`
Returns a copy of the receiver with leading whitespace, trailing whitespace, and trailing `"\0"` characters removed.

`String#strip!() #=> String or nil`
Behaves as `String#strip`, but modifies the receiver in-place. Returns the receiver if actually modified; otherwise, `nil`.

`String#sub(pattern, replacement) #=> String`
`String#sub(pattern) {|matchdata| } #=> String or Enumerator`
Behaves as `String#gsub`, except only the first match is replaced.

`String#sub!(pattern, replacement) #=> String or nil`
`String#sub!(pattern) {|matchdata| } #=> String or Enumerator`
Behaves as `String#sub`, but modifies the receiver in-place. Returns the receiver if actually modified; otherwise, `nil`.

`String#succ() #=> String`
Aliases `String#next`.

`String#succ!() #=> String`
Aliases `String#next!`.

`String#sum(bits=16) #=> Integer`
Calculates a checksum of the receiver by computing `byte % (2 ** bits - 1)` for each byte of the receiver, then summing the result.

`String#swapcase() #=> String`
Returns a copy of the receiver with uppercase ASCII characters converted to lowercase, and lowercase ASCII characters converted to uppercase.

`String#swapcase!() #=> String or nil`
Behaves as `String#swapcase`, but modifies the receiver in-place. Returns the receiver if actually modified; otherwise, `nil`.

`String#to_c() #=> Complex`
Assumes the leading characters of the receiver represent a complex number comprising a numeric literal then, optionally, a solidus followed by another numeric literal, an optional sign, then the letter *i*. Interprets the portion

before the solidus as the real part of the complex number, and the portion between the solidus and *i* as the imaginary part. If the imaginary part is not specified, `0` is assumed. Returns a `Complex` number with the values extracted, or `Complex(0, 0)` on failure.

`String#to_f() #=> Float`

Assumes the leading characters of the receiver represent a floating-point number comprising an integer literal then, optionally, a full stop followed by another integer literal. Interprets the first portion as the whole part of the `Float`, and the last portion as its fractional part. Assumes a fractional part of `0` if the latter portion is omitted. Returns a `Float` with the value extracted.

`String#to_i(`*`base=10`*`) #=> Integer`

Assumes the leading characters of the receiver constitute an integer in the given base, where *base* is between `2` and `36`. Ignores leading whitespace, but honours a leading sign. If *base* is `0`, infers the base by looking for a prefix: `0b` implies binary, `0o` and `0` imply octal, `0d` implies decimal, and `0x` implies hexadecimal. Returns this value, or `0` if no value could be found, as an `Integer`.

`String#to_r() #=> Rational`

Assumes the leading characters of the receiver represent a rational number comprising a numeric literal then, optionally, a solidus followed by another numeric literal. These characters may be enclosed by parentheses. Interprets the first portion as the numerator, and the last portion as the denominator. If either part couldn't be read they have the values `0` and `1`, respectively. Returns a new `Rational` with the value extracted.

`String#to_s() #=> String`

Returns the receiver. Aliased by `String#to_str`.

`String#to_str() #=> String`

Aliases `String#to_s`.

`String#to_sym() #=> Symbol`

Returns the receiver as a `Symbol`.

`String#tr(from, to) #=> String`

Returns a copy of the receiver with the characters in the *from* `String`

translated to the corresponding characters in the *to* `String`. Either argument may specify ranges of characters by separating the beginning and end points with `-`. If *from* begins with `^` it represents the characters not listed. When *to* is shorter than *from*, its last character is repeated to redress the difference.

`String#tr!(from, to) #=> String or nil`
Behaves as `String#tr`, but modifies the receiver in-place. Returns the receiver if actually modified; otherwise, `nil`.

`String#tr_s(from, to) #=> String`
Translates the receiver with `String#tr` then collapses runs of identical characters in the translated regions.

`String#tr_s!(from, to) #=> String or nil`
Behaves as `String#tr_s`, but modifies the receiver in-place. Returns the receiver if actually modified; otherwise, `nil`.

`String#unpack(format) #=> Array`
Separates the receiver into an `Array` of sub-`String`s according to the given template. For more details see Unpacking.

`String#upcase() #=> String`
Returns a copy of the receiver with lowercase ASCII characters converted to uppercase.

`String#upcase!(from, to) #=> String or nil`
Behaves as `String#upcase`, but modifies the receiver in-place. Returns the receiver if actually modified; otherwise, `nil`.

`String#upto(max) {|string| } #=> String or Enumerator`
Generates, using `String#succ`, each `String` from the receiver to the `String` *max*, inclusive, yielding each to the block. Returns an `Enumerator` if the block is omitted.

`String#valid_encoding?() #=> true or false`
Returns `true` if the contents of the receiver is valid according to its encoding; otherwise, `false`.

STRUCT

`Struct.new(`*name*`, `*member*`, …) { } #=> Struct::Example`
Initialises and returns a new `Class` inheriting from `Struct`. It is named `Struct::`*name*, or is anonymous if *name* is omitted. An accessor method is defined for each *member* `Symbol`. If a block is given, it is evaluated in the context of the new `Struct`'s class. In the descriptions that follow, `Struct::Example` is a `Class` returned by this method.

`Struct::Example.new(value, …) #=> Struct::Example`
Instantiates and returns an instance of the receiver. Each *value* is assigned to the corresponding member; an omitted *value* is `nil`. An `ArgumentError` is raised if more *value*s are given than there are members. Aliased by `Struct::Example[]`.

`Struct::Example[](value, …) #=> Struct::Example`
Aliases `Struct::Example.new`.

`Struct::Example.members() #=> Array`
Returns the names of the receiver's members as an `Array` of `Symbol`s.

`Struct::Example#==(object) #=> true or false`
Returns `true` if *object* was generated by `Struct.new` and has the same number of members as the receiver, each of which have the same names and equal values according to `#==`; otherwise, `false`.

`Struct::Example#[](position) #=> Object`
`Struct::Example#[](name) #=> Object`
Returns the value of the given member, identified either by its `Fixnum` *position* or `Symbol` *name*. If there isn't such a member, the first form raises an `IndexError`, and the second form raises a `NameError`.

`Struct::Example#[]=(position, object) #=> Object`
`Struct::Example#[]=(name, object) #=> Object`
Sets the value of the given member to *object*. A member is identified either

by its Fixnum *position* or Symbol *name*. If there isn't such a member, the first form raises an IndexError, and the second form raises a NameError.

Struct::Example#each() {|value| } #=> Struct::Example or Enumerator
Yields the value of each member in turn, returning the receiver. If the block is omitted, an Enumerator is returned.

Struct::Example#each_pair() {|member, value| } #=> Struct::Example or Enumerator
Yields the name of each Symbol member, in turn, along with its value, returning the receiver. If the block is omitted, an Enumerator is returned.

Struct::Example#length() #=> Integer
Returns the number of members in the receiver. Aliased by Struct::Example#size.

Struct::Example#members() #=> Array
Returns the names of the receiver's members as an Array of Symbols.

Struct::Example#size() #=> Integer
Aliases Struct::Example#length.

Struct::Example#to_a() #=> Array
Returns the values of the receiver's members. Aliased by Struct::Example#values.

Struct::Example#values() #=> Array
Aliases Struct::Example#to_a.

Struct::Example#values_at(position, …) #=> Array
Returns the values corresponding to the given members, which are specified as Fixnum *position*s or Ranges of the same.

STRUCT::TMS

`Struct::Tms#cstime() #=> Float`

Returns the number of seconds of system CPU time consumed by waited-for, terminated child processes, i.e. the sum of their `Struct::Tms#stime` and `Struct::Tms#cstime` values. Returns `0.0` on Windows.

`Struct::Tms#cutime() #=> Float`

Returns the number of seconds of user CPU time consumed by waited-for, terminated child processes, i.e. the sum of their `Struct::Tms#utime` and `Struct::Tms#cutime` values. Returns `0.0` on Windows.

`Struct::Tms#stime() #=> Float`

Returns the number of seconds of system CPU time consumed by the calling process.

`Struct::Tms#utime() #=> Float`

Returns the number of seconds of user CPU time consumed by the calling process.

SYMBOL

`Symbol.all_symbols() #=> Array`
Returns the names of all `Symbol`s currently defined.

`Symbol#<=>(object) #=> -1, 0, 1, or nil`
Converts both the receiver and *object* to `String`s then compares them with `String#<=>`.

`Symbol#==(object) #=> true or false`
Returns `true` if *object* is a `Symbol` with the same `Symbol#object_id` as the receiver; otherwise, `false`. Aliased by `Symbol#===`.

`Symbol#===(object) #=> true or false`
Aliases `Symbol#==`.

`Symbol#=~(pattern) #=> Integer or nil`
Converts the receiver to a `String`, then returns the value of `String#=~` with the same argument. Aliased by `Symbol#match`.

`Symbol#[](offset, `*`length`*`) #=> String or nil`
`Symbol#[](range) #=> String or nil`
`Symbol#[](regexp, `*`group`*`) #=> String or nil`
`Symbol#[](string) #=> String or nil`
Converts the receiver to a `String` then returns the result of `String#[]` for the same arguments. Aliased by `Symbol#slice`.

`Symbol#capitalize() #=> Symbol`
Converts the receiver to a `String`, capitalises it with `String#capitalize`, then returns the result as a `Symbol`.

`Symbol#casecmp(object) #=> -1, 0, 1, or nil`
Returns `nil` unless *object* is a `Symbol`. Otherwise, converts the receiver and *object* to `String`s, then returns the value of `String#casecmp` when given the converted *object* as argument.

`Symbol#downcase() #=> Symbol`
Converts the receiver to a `String`, changes it to lowercase with `String#downcase`, then returns the result as a `Symbol`.

`Symbol#empty?() #=> Symbol`
Converts the receiver to a `String`, then returns the result of `String#empty?`.

`Symbol#encoding() #=> Encoding`
Returns the encoding of the receiver.

`Symbol#id2name() #=> String`
Returns the receiver as a `String`. Aliased by `Symbol#to_s`.

`Symbol#inspect() #=> String`
Returns a `String` which evaluates to the receiver.

`Symbol#intern() #=> Symbol`
Returns the receiver.

`Symbol#length() #=> Integer`
Converts the receiver to a `String`, then returns the result of `String#length`. Aliased by `Symbol#size`.

`Symbol#match(pattern) #=> Integer or nil`
Aliases `Symbol#=~`.

`Symbol#next() #=> Symbol`
Converts the receiver to a `String`, then returns the result of `String#next` as a `Symbol`. Aliased by `Symbol#succ`.

`Symbol#size() #=> Integer`
Aliases `Symbol#length`.

`Symbol#slice(offset, length) #=> String or nil`
`Symbol#slice(range) #=> String or nil`
`Symbol#slice(regexp, group) #=> String or nil`
`Symbol#slice(string) #=> String or nil`
Aliases `Symbol#[]`.

`Symbol#succ() #=> Symbol`
Aliases `Symbol#next`.

`Symbol#swapcase() #=> Symbol`
Converts the receiver to a `String`, adjusts its letter case with `String#swapcase`, then returns the result as a `Symbol`.

`Symbol#to_proc() #=> Proc`
Converts the receiver to a `Proc` of the form `{|object| object.send(`*symbol*`)}`, where *symbol* is the receiver. In other words, interprets the receiver as a name of a method to call on each object passed to the block.

`Symbol#to_s() #=> String`
Aliases `Symbol#id2name`.

`Symbol#to_sym() #=> Symbol`
Returns the receiver.

`Symbol#upcase() #=> Symbol`
Converts the receiver to a `String`, adjusts its letter case with `String#upcase`, then returns the result as a `Symbol`.

BIBLIOGRAPHY

Books

Aho86

Compilers, Principles, Techniques, and Tools; Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman; 1986; Addison-Wesley Publishing Company

Beck98

Linux Kernel Internals; Michael Beck, Harold Böhme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, and Dirk Verworner; 1998; Addison-Wesley Publishing Company

Black09

The Well-Grounded Rubyist; David A. Black; 2009; Manning Publications

Brown09

Ruby Best Practices; Gregory T. Brown; 2009; O'Reilly Media

Bruce02

Foundations of Object-Oriented Languages: Types and Semantics; Kim B. Bruce; 2002; MIT Press

Budd87

A Little Smalltalk; Timothy A. Budd; 1987; Addison Wesley

Fischer92

The Anatomy of Programming Languages; Alice E. Fischer and Frances S. Grodzinsky; 1992; Prentice-Hall

Flano8

The Ruby Programming Language; David Flanagan and Yukihiro Matsumoto; 2008; O'Reilly Media

Friedmano8

Essentials of Programming Languages; Daniel P. Friedman and Mitchell Wand; 2008; MIT Press

Goldberg76

Smalltalk-72 Instruction Manual; Adele Goldberg and Alan Kay; 1976; Xerox Corporation

Graham96

ANSI Common Lisp; Paul Graham; 1996; Prentice Hall

James92

Mathematics Dictionary; Robert C. James; 1992; Chapman & Hall

Kernighan78

The Elements of Programming Style; Brian W. Kernighan and P. J. Plauger; 1978; Mcgraw-Hill

Kernighan84

The UNIX Programming Environment; Brian W. Kernighan and Rob Pike; 1984; Prentice Hall

Klas95

Metaclasses and Their Application; Wolfgang Klas and Michael Schrefl; 1995; Springer

LispStd

Programming Language—Common Lisp; American National Standard for Information Systems; 1993;

Liu99

Smalltalk, Objects, and Design; Chamond Liu; 1999; toExcel

Loosemore07

The GNU C Library Reference Manual; Sandra Loosemore, Richard M. Stallman, Roland McGrath, and Ulrich Drepper; 2007; The Free Software Foundation

Love07

Linux System Programming; Robert Love; 2007; O'Reilly Media

Meyer00

Object-oriented Software Construction; Bertrand Meyer; 2000; Prentice Hall

Mitchell04

Concepts in Programming Languages; John C. Mitchell; 2004; Cambridge Univeristy Press

Per10

Metaprogramming Ruby; Paolo Perrotta; 2010; Pragmatic Bookshelf

Raymond99

The New Hacker's Dictionary; Eric Raymond; 1999; The MIT Press

Raymond03

The Art of UNIX Programming; Eric Raymond; 2003; Addison-Wesley

Scott06

Programming Language Pragmatics; Michael L. Scott; 2006; Morgan Kaufmann

Stevens05

Advanced Programming in the UNIX® Environment; W. Stevens and Stephen Rago; 2005; Addison Wesley Professional

Thom06

Programming Ruby; Dave Thomas, Chad Fowler, and Andy Hunt; 2009; Pragmatic Bookshelf

Turbako8

Design Concepts in Programming Languages; Franklyn A. Turbak, David K. Gifford, and Mark A. Sheldon; 2008; MIT Press

Walloo

Programming Perl; Larry Wall, Tom Christiansen, and Jon Orwant; 2000; O'Reilly & Associates, Inc.

Articles

Buckо6

Under the hood: ActiveRecord::Base.find, Part 3; Jamis Buck; 2006;

Fowlero8

Dynamic Reception; Martin Fowler; 2008;

Haradaо9

The Design and Implementation of Ruby M17N; Yui Naruse; 2009;

Kay98

Prototypes vs Classes; Alan Kay; 1998;

Nuttero8

Ruby's Thread#raise, Thread#kill, timeout.rb, and net/protocol.rb libraries are broken; Charles Nutter; 2008;

Taivalsaari96

Journal of Object-Oriented Programming: Classes vs. Prototypes — Some Philosophical and Historical Observations; Antero Taivalsaari; 1996; Springer Verlag

ruby-core:28281

[ruby-core:28281] [Bug:trunk] add explicit constraints for WONTFIX IO bug; Yusuke Endoh; 2010;

Tr15285

An operational model for characters and glyphs; ISO/IEC; 1998; ISO/IEC

Uax44

Unicode Standard Annex #44: Unicode Character Database; Mark Davis and Ken Whistler; 2009; Unicode Consortium

PREDEFINED GLOBAL VARIABLES

Ruby predefines certain global variables automatically. They are summarised in the following table.

Global Variable	English Name	Value	Meaning
`$*`	`$ARGV`	`Array`	Read-only alias of `ARGV`
`$$`	`$PID` / `$PROCESS_ID`	`Fixnum`	Process ID of the current Ruby process. Read-only.
`$?`	`$CHILD_STATUS`	`nil` or `Process::Status`	Exit status of the last terminated process. Read-only.
`$-d`			Whether Ruby was invoked with the `-d` or `--debug` switches.
`$DEBUG`		`true` or `false`	
`$"`			Absolute filenames of files loaded with `Kernel.load`, `Kernel.require`, or `Kernel.require_relative`. Read-only.
`$LOADED_FEATURES`		`Array` of `Strings`	
`$:`			Absolute paths to directories searched by `Kernel.load` or `Kernel.require`. Read-only, but the
`$LOAD_PATH`		`Array` of `Strings`	
`$-I`			

Predefined global variables

Global Variable	English Name	Value	Meaning
			contents of the Array can be modified.
$0			Filename of the Ruby script being executed.
$PROGRAM_NAME		Strings	Equal to - if the program was read from STDIN, or -e if ruby was supplied with the -e switch.
$SAFE		Fixnums	Current safe level. May be set from the command-line with the -T option.
$-w			true if the -v, -w, or --verbose switches were supplied on the command line; nil if the -W0 switch was supplied; false, otherwise. Assigning nil to this variable suppresses all warnings.
$-v			
$VERBOSE		true, false, or nil	
$!	$ERROR_INFO	Exception	Inside a rescue clause or after the rescue modifier holds the current exception.
$@	$ERROR_POSITION	Array of Strings	Inside a rescue clause or after the rescue modifier holds the stack trace of the current exception; equivalent to $!.backtrace.
$_	$LAST_READ_LINE	String	Last String read by Kernel.gets or Kernel.readline. Method-local.
$<	$DEFAULT_INPUT	ARGF	Read-only alias for ARGF.
$stdin		IO	Standard input stream.

Global Variable	English Name	Value	Meaning
$>	$DEFAULT_OUTPUT	IOs	Standard output stream.
$stdout			
$stderr		IO	Standard error stream.
$FILENAME		String	Filename of the file currently being read from ARGF; - if ARGF is empty or reading from STDIN. Read-only.
$.	$NR / $INPUT_LINE_NUMBER	Fixnum	Number of the last line read from the current input file in ARGF.
$/	$RS / $INPUT_RECORD_SEPARATOR	String	Input record separator. Default value is "\n". Can be set with the -0 switch.
$-0			
$\	$ORS / $OUTPUT_RECORD_SEPARATOR	String or nil	Appended to Kernel.print output if non-nil. Default value is nil, or $/ if the -l switch is given.
$,	$OFS / $OUTPUT_FIELD_SEPARATOR	nil or String	Separator printed between the arguments of Kernel.print and the default separator of Array.join. Equal to nil by default.
$;	$FS / $FIELD_SEPARATOR	String or nil	Default field separator of String#split. Default value is nil or the argument to the -F switch.
$-F			
$F		Array of Strings	If the -a and -n/-p switches were given, holds the return value of String#split for the current input line.

Global Variable	English Name	Value	Meaning
$~	$MATCH_INFO	MatchData or nil	MatchData from the last regexp match. Method-local.
$&	$MATCH	String	Text matched by last regexp match. Read-only. Method-local.
$`	$PREMATCH	String	Text preceding the match of the last regexp match. Read-only. Method-local.
$'	$POSTMATCH	String	Text following the match of the last regexp match. Read-only. Method-local.
$+	$LAST_PAREN_MATCH	String	Text enclosed in the last successfully matched group of the last regexp match. Read-only. Method-local.
$-a		true or false	true if the -a switch was given; false otherwise. Read-only.
$-i		true or false	Argument of the -i switch, if given; otherwise nil.
$-l		true or false	true if the -l switch was given; false otherwise. Read-only.
$-p		true or false	true if the -p switch was given; false otherwise. Read-only.
$-W		Fixnum	Current verbosity level: 0 if the -W0 switch was given; 2 if the -w, -v, or --verbose switches were given; 1 otherwise. Read-only.

Global Variable	`English` Name	Value	Meaning
`$1`–`$`*n*		`String`	Text matched by the n^{th} capturing group in the last pattern match; `nil` if the match failed or there were fewer than *n* groups. Read-only.

GLOSSARY

#[]

Conventionally, indexes the receiver by the "key" supplied as the argument(s), returning the requested slice or `nil`/`[]` if no corresponding data was found. It is a special case of the message expression syntax because it sends a selector named `:[]` to the receiver, passing in the contents of `[…]` as arguments, i.e. *`receiver`*`[`*`argument`*$_0$ `,…,`*`argument`*$_n$ `]` is equivalent to *`receiver`*`.[](`*`argument`*$_0$ `,…,`*`argument`*$_n$ `)`

The simplest example is `Array#[]`, which returns the element stored at the given `Integer` index. Similarly, `Hash#[]` returns the value corresponding to the given key object. In this role, `:[]` acts as an interface to a pre-computed lookup table.

However, this abstraction conveniently extends to virtual slices, where the values are computed dynamically, shielding the user from these unnecessary details: he need not concern himself with *how* the data are derived, merely that they satisfy the key. For example, `Array#[]` also accepts a `Range` argument, for which it returns a sub-`Array` of elements whose indexes are members of the `Range`. This is computationally quite a different operation to the case with `Integer` arguments, yet the API makes no distinction. A clearer case is `Dir.[]` which interprets its arguments as shell globs against the current working directory, returning an `Array` of the matching entries. In this way `Dir` behaves as if it maintains a `Hash`-style mapping from glob to files.

As alluded to with `Array#[`*`range`*`]`, it is common for `:[]` to accept arguments of wildly different types and try to produce a sensible result. For example, `String#[]` accepts either an `Integer`, a pair of `Integers`, a `Range`, a `Regexp`, a `Regexp` and an `Integer`, or a `String`.

```
formats = %w{MP3 CD Cassette 8-Track Record}

def i_buy(format)
```

```
  "I buy my music on #{format.downcase}"
end

method(:i_buy)[ formats[2] ]
#=> "I buy my music on cassette"
method(:i_buy)[ formats[-2..-1].join(' or ' ) ]
#=> "I buy my music on 8-track or record"
```

#[]=

Conventionally, the assignment counterpart to #[]: the first argument(s) describe the slice, and the final argument its new value. It is expected that *obj*[*key*] = *value* then *obj*[*key*] == *value*. For this equivalency to hold, #[]= may need to be more restrictive in the arguments it accepts than its counterpart because some key forms make unsuitable assignment targets. For example, Method#[] calls the objectified method with the supplied arguments, but it is unclear what it would mean to assign to such a slice, so Method#[]= is not defined.

For example, String#[*regexp*]= *value* assigns *value* to the portion of the String matched by *regexp*; Array#[*range*]= *value* replaces the elements in *range* with *value*.

```
tower = 'Tower of Pisa'
tower[/\w+$/] = 'Babel'
tower #=> "Tower of Babel"
tower[-5..-1] = 'London'
tower #=> "Tower of London"
tower = tower.split(//)
tower[0..4] = %w{C i t y}
tower.join #=> "City of London"
```

!

Right-associative unary operator which, conventionally, performs a Boolean NOT operation. Definable as a method with a selector of !.

~

Right-associative unary operator which, conventionally, performs a bitwise complement. Definable as a method with a selector of `~`.

+ (unary)

Right-associative unary operator which, conventionally, gives its operand a positive sign. Definable as a method with a selector of `+@`.

See also:

- `+`
- `-`

**

Right-associative binary operator which, conventionally, performs exponentiation. Definable as a method with a selector of `**`.

- (unary)

Right-associative unary operator which, conventionally, gives its operand a negative sign. Definable as a method with a selector of `-@`.

*

Left-associative binary operator which, conventionally, multiples the receiver by a sole numeric argument, returning the result. For example: `Fixnum#*`, `String#*`, and `Array#*`.

```
45 * 2    #=> 90
'45' * 2  #=> "4545"
[45] * 2  #=> [45, 45]
```

/

Left-associative binary operator which, conventionally, divides its operands. Definable as a method with a selector of `/`.

%

Left-associative binary operator which, conventionally, returns the modulus of numeric operands. Definable as a method with a selector of `%`.

+ (binary)

Left-associative binary operator which, conventionally, sums or concatenates the operands, which should be of the same class, returning a new object. Definable as a method with a selector of `+`. Whereas `<<` appends the argument to the receiver, `#+` combines the operands into a new object. For example, `Fixnum#+`, `String#+`, `Array#+`.

Addition in this manner is not necessarily commutative when the operands are non-numeric. For example, `'a' + 'b' != 'b' + 'a'`.

```
4 + 64     #=> 68
'4' + '64' #=> "464"
[4] + [64] #=> [4, 64]
```

See also:

- `+`

-

Left-associative binary operator which, conventionally, performs subtraction. Definable as a method with a selector of `-`.

See also:

- -

<<

Left-associative binary operator which, conventionally, appends the second operand to the first, then returns the mutated receiver. When the receiver is an `Integer` it is shifted left the amount of places specified by the argument. For example, `Array#`, `String#`, `IO#`, and `Enumerator::Yielder#`. Definable as a method with the selector <<.

```
4 << 64   #=> 73786976294838206464
'4' << 64 #=> "4@"
[4] << 64 #=> [4, 64]
```

See also:

- >>

>>

Left-associative binary operator which, conventionally, performs a rightwards bitwise shift. Definable as a method with the selector >>.

See also:

- <<

&

Left-associative binary operator which, conventionally, performs bitwise AND. Definable as a method with the selector `&`.

|

Left-associative binary operator which, conventionally, performs bitwise OR. Definable as a method with the selector `|`.

^

Left-associative binary operator which, conventionally, performs bitwise XOR. Definable as a method with the selector `^`.

<

Left-associative binary operator which, conventionally, determines whether the receiver is *less than* the argument. Normally supplied by the `Comparable` module which implements it in terms of `<=>` . `Class#` creatively uses this selector to test whether the receiver is a *kind of* the argument because it mirrors the syntax for defining a class with a superclass, i.e. `class` *`name superclass`*…`end`. Definable as a method with the selector `<`.

```
42 < 43    #=> true
10 < 1     #=> false
'a' < 'aa' #=> true
'z' < 'Z'  #=> false
```

<=

Left-associative binary operator which, conventionally, returns `true` if its receiver is less than or equal to its argument; `false` otherwise. Provided by the `Comparable` module, or definable as a method with the selector `<=`.

>=

Left-associative binary operator which, conventionally, returns `true` if its receiver is greater than or equal to its argument; `false` otherwise. Provided by the `Comparable` module, or definable as a method with the selector `>=`.

>

Left-associative binary operator which, conventionally, determines whether the receiver is *greater than* the argument. Provided by the `Comparable` module, or definable as a method with the selector `>`. `Class#` provides symmetry to `Class#` by testing whether the argument is *kind of* the receiver.

```
42 > 43    #=> false
10 > 1     #=> true
'a' > 'aa' #=> false
'z' > 'Z'  #=> true
```

==

Non-associative binary operator which, conventionally, determines whether the operands are equivalent. Provided by the `Comparable` module, or definable as a method with the selector `==`.

```
1 == 1   #=> true
1 == 1.0 #=> true
1 == :one #=> false
```

===

Non-associative binary operator which, conventionally, performs case equality. Definable as a method with the selector `===`.

!=

Non-associative binary operator which, conventionally, returns `true` if `==` returns `false`; `false` otherwise. Definable as a method with the selector `!=`.

=~

Non-associative binary operator which, conventially, matches the receiver with the argument. One of the operands is typically a `Regexp`. This operator is commutative, i.e. (*a* `=~` *b*) `==` (*b* `=~` *a*). For example, `String#=~`, `Regexp#=~`, and `Symbol#=~`. Definable as a method with the selector `=~`.

```
"Hieronymus Bosch" =~ /Ron/i    #=> 3
"Jheronimus van Aken" =~ /Ron/i #=> 3
/hero/ =~ "Jheronimus van Aken" #=> 1
/heretic/ =~ "Hieronymus Bosch" #=> nil
```

!~

Non-associative binary operator which, conventionally, returns `true` if `=~` returns `nil`; `false` otherwise. Definable as a method with the selector `!~`.

<=>

Non-associative binary operator which, conventionally, compares the operands. Colloquially: the *spaceship operator*. Definable as a method with the selector `<=>`.

```
2 <=> 2    #=> 0
2 <=> 1    #=> 1
2 <=> 3    #=> -1
2 <=> :two #=> nil
```

&&

Left-associative binary operator that performs Boolean AND.

||

Left-associative binary operator that performs Boolean OR.

..

Non-associative binary operator that creates an inclusive Range from its operands.

See also:

- ...

...

Non-associative binary operator that creates an exclusive Range from its operands.

See also:

- ..

…?…:…

Right-associative ternary operator that returns its second operand if its first is true, otherwise, it returns its third.

=

Right-associative binary operator that performs assignment.

operator=

Right-associative binary operator that performs abbreviated assignment, where *operator* is one of the following operators:

- **
- *
- /
- %
- + (binary)
- -
- <<
- >>
- &&
- &
- ||
- |
- ^

not

Right-associative, low-precedence, unary operator that performs Boolean NOT.

and

Left-associative, low-precedence, Boolean operator that performs Boolean AND.

or

Left-associative, low-precedence, Boolean operator that performs Boolean OR.

arity

An operator's *arity* is the number of operands on which it operates, including the receiver. The + operator, for example, has an arity of 2. Consequently, an operator can be classified as follows:

Unary operator
: An arity of 1.

Binary operator
: An arity of 2.

Ternary operator
: An arity of 3.

A few operators are both unary and binary: they can either be used with one operand or two. +, again, serves to illustrate: in its unary form it changes the sign of its operand; in its binary form it performs summation.

associativity

```
10 - 9 - 8 #=> -7
10 - (9 - 8) #=> 9
```

If one operator expression is followed by another with the same precedence, the operator's *associativity* determines their order of evaluation. *Left-associative* operators are evaluated from left to right; *right-associative* operators are evaluated in the opposite direction. *Non-associative* operators are ambiguous: parentheses must be employed to specify their order of evaluation.

_by

A selector with a `_by` suffix typically implies that the method expects a block, the results of which constrain the computation. For example, `Enumerable#group_by`, `Enumerable#sort_by`, and `Enumerable#minmax_by`.

```
(1..10).sort_by {|n| -n}
#=> [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
(1..10).sort_by {|n| n % 10}
#=> [10, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

#call

Conventionally, an instance method that accepts a variable-length argument list with which it invokes its receiver. The return value is the result of the invocation. `Method` and `Proc` objects both behave in this fashion.

This selector is notable because it instruments the following syntax: *receiver*`.call(`*arg*`0`…*arg*` n`) is equivalent to *receiver*`.(`*arg*`0`…*arg*` n`). That is, the selector's name can be omitted from the message expression.

```
class Girl
  def initialize(name)
    @name = name.capitalize
  end

  def call(*sweet_nothings)
    "#@name: #{sweet_nothings.sample}..."
  end
end

jessica = Girl.new :jessica
jessica.call('Your eyes are a blue million miles',
             'For all eternity')
#=> "Jessica: Your eyes are a blue million miles..."
rose = Girl.new :rose
rose.('I would walk 10, 000 miles',
      'Superlatives cannot express')
#=> "Rose: Superlatives cannot express..."
```

default external encoding

The default value for the external encoding of new `IO` streams. See IO Streams for details.

default internal encoding

The default value for the internal encoding of new `IO` streams. See IO Streams for details.

#each

Classes mixing-in the Enumerable module are expected, by default, to respond to `#each` by yielding the next element of the sequence. Conventionally, it returns an `Enumerator` when the block is omitted.

```
(-2..2).each #=> #<Enumerator: -2..2:each>
(-2..2).each {|n| print " <#{n}> "}
# <-2>  <-1>  <0>  <1>  <2>
```

#each_*attribute*

Conventionally, a message selector of the form `:each_`*`attribute`* request their receiver enumerate the collection in terms of *attribute*, that is yielding each successive *attribute*; as opposed to `#each` which presumably yields a different sort of attribute. For example, `String#each_codepoint` interprets the string as a collection of codepoints, yielding each in turn. `String#each_char`, however, interprets the string as a collection of characters, so will yield a different sequence of objects. An object need only respond to such messages if it already responds to `:each`, and can sensibly be enumerated in another fashion. It is common to provide an alias for a selector of this form named with the plural of *attribute*, e.g. `String#each_byte` is aliased to `String#bytes`; `IO#each_line` is aliased to `IO#lines`.

```
# coding: utf-8
"Böhm-Bawerk".each_line.to_a
#=> ["Böhm-Bawerk"]
"Böhm-Bawerk".each_char.to_a
#=> ["B", "ö", "h", "m", "-", "B", "a", "w", "e", "r", "k"]
"Böhm-Bawerk".each_byte.to_a
#=> [66, 195, 182, 104, 109, 45, 66, 97, 119, 101, 114, 107]
"Böhm-Bawerk".each_codepoint.to_a
#=> [66, 246, 104, 109, 45, 66, 97, 119, 101, 114, 107]
```

Element Reference

See: `#[]`

#empty?

The `:empty?` predicate is defined by `Array`, `Hash`, `Set`, `SortedSet`, `String`, and `Symbol`. It returns `true` if the receiver doesn't have any content; `false` otherwise.

external encoding

The encoding of the data in an `IO` stream. See IO Streams for details.

internal encoding

The encoding to which data in an `IO` stream should be automatically transcoded to. See IO Streams for details.

#length

See: `#size`

precedence

A compound expression might itself contain compound expressions, each of which comprises an operator and its operands. For example, `1 - 2 * 3`. This expression is potentially ambiguous: is the intent to subtract 2 from 1, then multiply the result by 3, or multiply 2 and 3, then subtract the result from 1? In evaluating such an expression Ruby must apply *precedence rules* so as to determine the order in which the operators should be performed. *Precedence* is a relative order defined over the operators such that each operator has lower, equal, or higher precedence than another. For a given statement, the higher an operator's precedence, the earlier it is evaluated. `*` has a higher precedence than `-`, as in mathematics, so the example above evaluates to −5.

```
4 * 3 + 1 ** 2     #=> 13
4 * (3 + 1) ** 2   #=> 64
(4 * (3 + 1)) ** 2 #=> 256
```

The default order of precedence may be overridden by grouping sub-expressions that should be performed earlier with parentheses, again mirroring the rule in mathematics. When parenthetical groups contain other parenthetical groups, the innermost is given the highest precedence.

#rewind

`Enumerator`'s respond to this selector by resetting their state to the initial element. If the object being enumerated responds to `:rewind`, it is sent the message instead. `IO` objects, such as `Dir` and `File`, respond in a similar way by re-position the stream to the beginning.

#size

Conventially, `#size` and `#length`, which are typically aliases of each other, return an `Integer` representing the magnitude of the receiver. For example, `Array#size` returns the number of elements the receiver contains; `File#size` returns the receiver's size in bytes.

```
4.size #=> 4
[:one, :two, :three, :four].size #=> 4
File.open('/tmp/four','w'){ print '1' * 4}
File.new('/tmp/four').size #=> 4
'five'.size #=> 4
```

Spaceship Operator

See: <=>

#to_

Conventionally, a message selector with a `#to_` prefix converts the receiver into an object of the corresponding core type.

```
10.to_s    #=> "10"                  # String
10.to_f    #=> 10.0                  # Float
10.to_r    #=> (10/1)                # Rational
10.to_c    #=> (10+0i)               # Complex
10.to_enum #=> #<Enumerator: 10:each> # Enumerator
```

try_convert

Conventionally, a class method which implicitly converts the argument to the receiver.

```
Array.try_convert [:violin] #=> [:violin]
Array.try_convert :violin   #=> nil
```