module Kernel

Index:

Array Float Integer String `(backquote) abort at_exit autoload binding block_given? callcc caller catch chomp chomp! chop chop! eval exec exit exit! fail fork format gets global_variables gsub gsub! iterator? lambda load local_variables loop open p print printf proc putc puts raise rand readline readlines require scan select set_trace_func singleton_method_added sleep split sprintf srand sub sub! syscall system test throw trace_var trap untrace_var

---

The `Kernel` module is included by class `Object`, so its methods are available in every Ruby object. The `Kernel` instance methods are documented in class `Object` beginning on page 351. This section documents the module methods. These methods are called without a receiver and thus can be called in functional form.

class methods

Array

Array(*arg*) -> *anArray*

Returns *arg* `.to_a`.

```
Array(1..5)                 »   [1, 2, 3, 4, 5]
```

Float

Float(*arg*) -> *aFloat*

Returns *arg* converted to a float. Numeric types are converted directly, `nil` is converted to `0.0`, and the rest are converted using *arg*.to_f.

```
Float(1)                    »   1.0
Float(nil)                  »   0.0
Float("123.456")            »   123.456
```

Integer

Integer(*arg*) -> *anInteger*

Converts *arg* to a `Fixnum` or `Bignum`. Numeric types are converted directly (with floating point numbers being truncated). If *arg* is a `String`, leading radix indicators (`0`, `0b`, and `0x`) are honored. This behavior is different from that of `String#to_i`.

```
Integer(123.999)                    »  123
Integer("0x1a")                     »  26
Integer(Time.new)                   »  1023599977
```

String

String(*arg*) -> *aString*

Converts *arg* to a `String` by calling its `to_s` method.

```
String(self)                        »  "main"
String(self.type)                   »  "Object"
String(123456)                      »  "123456"
```

` (backquote)

`cmd` -> *aString*

Returns the standard output of running *cmd* in a subshell. The built-in syntax `%x{...}` described on page 73 uses this method.

```
`date`                    »  "Sun Jun  9 00:19:37 CDT 2002\n"
`ls testdir`.split[1]     »  "main.rb"
```

abort

abort

Terminate execution immediately, effectively by calling `Kernel.exit(1)`.

at_exit

at_exit { block } -> *aProc*

Converts *block* to a `Proc` object (and therefore binds it at the point of call) and registers it for execution when the program exits. If multiple handlers are registered, they are executed in reverse order of registration.

```
def do_at_exit(str1)
  at_exit { print str1 }
end
at_exit { puts "cruel world" }
do_at_exit("goodbye ")
exit
```

produces:

```
goodbye cruel world
```

autoload

autoload(*aModule*, *aFile*) -> `nil`

Registers *aFile* to be loaded (using Kernel::require) the first time that *aModule* (which may be a String or a symbol) is accessed.

```
autoload :MyModule, "/usr/local/lib/modules/my_module.rb"
```

binding

binding -> *aBinding*

Returns a Binding object, describing the variable and method bindings at the point of call. This object can be used when calling eval to execute the evaluated command in this environment. Also see the description of Binding beginning on page 291.

```
def getBinding(param)
  return binding
end
b = getBinding("hello")
eval "param", b                          »    "hello"
```

block_given?

block_given? -> true or false

Returns true if yield would execute a block in the current context.

```
def try
  if block_given?
    yield
  else
    "no block"
  end
end
try                                      »    "no block"
try { "hello" }                          »    "hello"
try do
  "hello"
end
```

callcc

callcc {| cont | block } -> *anObject*

Generates a Continuation object, which it passes to the associated block. Performing a *cont* .call will cause the callcc to return (as will falling through the end of the block). The value returned by the callcc is the value of the block, or the value passed to *cont* .call. See Continuation on page 294 for more details. Also see Kernel::throw for an alternative mechanism for unwinding a call stack.

caller

caller(*[anInteger]*) -> *anArray*

Returns the current execution stack---an array containing strings in the form ``*file:line*'' or ``*file:line: in `method'*''. The optional *anInteger* parameter determines the number of initial stack entries to omit from the result.

```
def a(skip)
  caller(skip)
end
def b(skip)
  a(skip)
end
def c(skip)
  b(skip)
end
c(0) » ["prog:2:in `a'", "prog:5:in `b'", "prog:8:in `c'", "prog:10"]
c(1) » ["prog:5:in `b'", "prog:8:in `c'", "prog:11"]
c(2) » ["prog:8:in `c'", "prog:12"]
c(3) » ["prog:13"]
```

catch

catch(*symbol*) {| | block }

-> anObject

`catch` executes its block. If a `throw` is executed, Ruby searches up its stack for a `catch` block with a tag corresponding to the `throw`'s *symbol*. If found, that block is terminated, and `catch` returns the value given to `throw`. If `throw` is not called, the block terminates normally, and the value of `catch` is the value of the last expression evaluated. `catch` expressions may be nested, and the `throw` call need not be in lexical scope.

```
def routine(n)
  puts n
  throw :done if n <= 0
  routine(n-1)
end


catch(:done) { routine(3) }
```

produces:

```
3
2
1
0
```

chomp

chomp(*[aString]*) -> $_ or *aString*

Equivalent to `$_ = $_.chomp(aString)`. See String#chomp on page 367.

```
$_ = "now\n"
```

```
chomp                          »  "now"
$_                             »  "now"
chomp "ow"                     »  "n"
$_                             »  "n"
chomp "xxx"                    »  "n"
$_                             »  "n"
```

chomp!

chomp!(*[aString]*) -> $_ or `nil`

Equivalent to `$_.chomp!(aString)`. See String#chomp!

```
$_ = "now\n"
chomp!                         »  "now"
$_                             »  "now"
chomp! "x"                     »  nil
$_                             »  "now"
```

chop

chop -> *aString*

Equivalent to `($_.dup).chop!`, except `nil` is never returned. See String#chop! on page 367.

```
a  =  "now\r\n"
$_ = a
chop                 »  "now"
$_                   »  "now"
chop                 »  "no"
chop                 »  "n"
chop                 »  ""
chop                 »  ""
a                    »  "now\r\n"
```

chop!

chop! -> $_ or `nil`

Equivalent to `$_.chop!`.

```
a  = "now\r\n"
$_ = a
chop!                »  "now"
chop!                »  "no"
chop!                »  "n"
chop!                »  ""
```

```
chop!                          »  nil
$_                             »  ""
a                              »  ""
```

eval

eval(*aString [*, *aBinding [file [line]]]*) -> *anObject*

Evaluates the Ruby expression(s) in *aString*. If *aBinding* is given, the evaluation is performed in its context. The binding may be a `Binding` object or a `Proc` object. If the optional *file* and *line* parameters are present, they will be used when reporting syntax errors.

```
def getBinding(str)
  return binding
end
str = "hello"
eval "str + ' Fred'"                            » "hello Fred"
eval "str + ' Fred'", getBinding("bye")         » "bye Fred"
```

exec

exec(*command [*, *args]*)

Replaces the current process by running the given external command. If `exec` is given a single argument, that argument is taken as a line that is subject to shell expansion before being executed. If multiple arguments are given, the second and subsequent arguments are passed as parameters to *command* with no shell expansion. If the first argument is a two-element array, the first element is the command to be executed, and the second argument is used as the `argv[0]` value, which may show up in process listings. In MSDOS environments, the command is executed in a subshell; otherwise, one of the `exec(2)` system calls is used, so the running command may inherit some of the environment of the original program (including open file descriptors).

```
exec "echo *"       # echoes list of files in current directory
# never get here


exec "echo", "*"    # echoes an asterisk
# never get here
```

exit

exit(*anInteger*=0)

Initiates the termination of the Ruby script by raising the `SystemExit` exception. This exception may be caught. The optional parameter is used to return a status code to the invoking environment.

```
begin
  exit
  puts "never get here"
rescue SystemExit
  puts "rescued a SystemExit exception"
```

```
end
puts "after begin block"
```

produces:

```
rescued a SystemExit exception
after begin block
```

Just prior to termination, Ruby executes any `at_exit` functions and runs any object finalizers (see `ObjectSpace` beginning on page 430).

```
at_exit { puts "at_exit function" }
ObjectSpace.define_finalizer(self,  proc { puts "in finalizer" })
exit
```

produces:

```
at_exit function
```

exit!

exit!(*anInteger*=-1)

Similar to `Kernel::exit`, but exception handling, `at_exit` functions, and finalizers are bypassed.

fail

fail
fail(*aString*)
fail(*anException [*, *aString [anArray]]*)

Synonym for `Kernel::raise`.

fork

fork *[*{ block } *]* -> *aFixnum* or `nil`

Creates a subshell. If a block is specified, that block is run in the subshell, and the subshell terminates with a status of zero. Otherwise, the `fork` call returns twice, once in the parent, returning the process id of the child, and once in the child, returning `nil`. The child process can exit using `Kernel::exit!` to avoid running any `at_exit` functions. The parent process should use `Process::wait` to collect the termination statuses of its children; otherwise, the operating system may accumulate zombie processes.

```
fork do
  3.times {|i| puts "Child: #{i}" }
end
3.times {|i| puts "Parent: #{i}" }
Process.wait
```

produces:

```
Parent: 0
Child: 0
Parent: 1
Child: 1
Parent: 2
Child: 2
```

format

format(*aString [*, *anObject]**) -> *aString*

Synonym for Kernel::sprintf.

gets

gets(*aString*=$/) -> *aString* or nil

Returns (and assigns to $_) the next line from the list of files in ARGV (or $*), or from standard input if no files are present on the command line. Returns nil at end of file. The optional argument specifies the record separator. The separator is included with the contents of each record. A separator of nil reads the entire contents, and a zero-length separator reads the input one paragraph at a time, where paragraphs are divided by two consecutive newlines. If multiple filenames are present in ARGV, gets(nil) will read the contents one file at a time.

```
ARGV << "testfile"
print while gets
```

produces:

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

global_variables

global_variables -> *anArray*

Returns an array of the names of global variables.

```
global_variables.grep /std/     » ["$stdin", "$stderr", "$stdout"]
```

gsub

gsub(*pattern*, *replacement*) -> *aString*
gsub(*pattern*) {| | block }

-> *aString*

Equivalent to $_.gsub..., except that $_ receives the modified result.

```
$_ = "quick brown fox"
gsub /[aeiou]/, '*'                »  "q**ck br*wn f*x"
$_                                 »  "q**ck br*wn f*x"
```

gsub!

gsub!(*pattern*, *replacement*) -> *aString* or nil
gsub!(*pattern*) {| | block }

-> *aString* or nil

Equivalent to Kernel::gsub, except nil is returned if $_ is not modified.

```
$_ = "quick brown fox"
gsub! /cat/, '*'                »  nil
```

```
$_                                »  "quick brown fox"
```

iterator?

iterator? -> `true` or `false`

Synonym for `Kernel::block_given?`. The `iterator?` method will be removed in Ruby 1.8.

lambda

lambda {| | block } -> *aProc*

Synonym for `Kernel::proc`.

load

load(*aFileName*, *wrap*=`false`) -> `true`

Loads and executes the Ruby program in the file *aFileName*. If the filename does not resolve to an absolute path, the file is searched for in the library directories listed in `$:`. If the optional *wrap* parameter is `true`, the loaded script will be executed under an anonymous module, protecting the calling program's global namespace. Any local variables in the loaded file will not be propagated to the loading environment.

local_variables

local_variables -> *anArray*

Returns the names of the current local variables.

```
fred = 1
for i in 1..10
   # ...
end
local_variables                   »   ["fred", "i"]
```

loop

loop {| | block }

Repeatedly executes the block.

```
loop {
  print "Input: "
  break if !gets or $_ =~ /^qQ/
  # ...
}
```

open

open(*aString [*, *aMode [perm]]*) -> *anIO* or `nil`
open(*aString [*, *aMode [perm]]*) {| anIO | block }

-> `nil`

Creates an `IO` object connected to the given stream, file, or subprocess.

If *aString* does not start with a pipe character (``|''), treat it as the name of a file to open using the specified mode defaulting to ``r'' (see the table of valid modes on page 326). If a file is being created, its initial permissions may be set using the integer third parameter.

If a block is specified, it will be invoked with the `File` object as a parameter, and the file will be automatically closed when the block terminates. The call always returns `nil` in this case.

If *aString* starts with a pipe character, a subprocess is created, connected to the caller by a pair of pipes. The returned `IO` object may be used to write to the standard input and read from the standard output of this subprocess. If the command following the ``|'' is a single minus sign, Ruby forks, and this subprocess is connected to the parent. In the subprocess, the `open` call returns `nil`. If the command is not ``-'', the subprocess runs the command. If a block is associated with an `open("|-")` call, that block will be run twice---once in the parent and once in the child. The block parameter will be an `IO` object in the parent and `nil` in the child. The parent's `IO` object will be connected to the child's `$stdin` and `$stdout`. The subprocess will be terminated at the end of the block.

```
open("testfile") do |f|
  print f.gets
end
```

produces:

```
This is line one
```

Open a subprocess and read its output:

```
cmd = open("|date")
print cmd.gets
cmd.close
```

produces:

```
Sun Jun  9 00:19:39 CDT 2002
```

Open a subprocess running the same Ruby program:

```
f = open("|-", "w+")
if f == nil
  puts "in Child"
  exit
else
  puts "Got: #{f.gets}"
end
```

produces:

```
Got: in Child
```

Open a subprocess using a block to receive the I/O object:

```
open("|-") do |f|
  if f == nil
    puts "in Child"
  else
    puts "Got: #{f.gets}"
  end
```

```
end
```
produces:
```
Got: in Child
```

p

p(*[anObject]*$^{+}$) -> `nil`

For each object, directly writes *`anObject`*`.inspect` followed by the current output record separator to the program's standard output. `p` bypasses the Ruby I/O libraries.

```
p self
```
produces:
```
main
```

print

print(*[anObject]**) -> `nil`

Prints each object in turn to `$defout`. If the output field separator (`$,`) is not `nil`, its contents will appear between each field. If the output record separator (`$\`) is not `nil`, it will be appended to the output. If no arguments are given, prints `$_`. Objects that aren't strings will be converted by calling their `to_s` method.

```
print "cat", [1,2,3], 99, "\n"
$, = ", "
$\ = "\n"
print "cat", [1,2,3], 99
```
produces:
```
cat12399
cat, 1, 2, 3, 99
```

printf

printf(*anIO*, *aString [*, *anObject]**) -> `nil`
printf(*aString [*, *anObject]**) -> `nil`

Equivalent to:

anIO`.write sprintf(`*aString, anObject* `...)`
or
`$defout.write sprintf(`*aString, anObject* `...)`

proc

proc { block } -> *aProc*

Creates a new procedure object from the given block. Equivalent to `Proc.new`.

```
aProc = proc { "hello" }
aProc.call                    »     "hello"
```

putc

putc(*anInteger*) -> *anInteger*

Equivalent to `$defout.putc( anInteger )`.

puts

puts(*[args]**) -> `nil`

Equivalent to `$defout.puts( args )`.

raise

raise
raise(*aString*)
raise(*anException [*, *aString [anArray]]*)

With no arguments, raises the exception in `$!` or raises a `RuntimeError` if `$!` is `nil`. With a single `String` argument, raises a `RuntimeError` with the string as a message. Otherwise, the first parameter should be the name of an `Exception` class (or an object that returns an `Exception` when sent `exception`). The optional second parameter sets the message associated with the exception, and the third parameter is an array of callback information. Exceptions are caught by the `rescue` clause of `begin...end` blocks.

```
raise "Failed to create socket"
raise ArgumentError, "No parameters", caller
```

rand

rand(*max*=0) -> *aNumber*

Converts *max* to an integer using max_1 = max`.to_i.abs`. If the result is zero, returns a pseudorandom floating point number greater than or equal to 0.0 and less than 1.0. Otherwise, returns a pseudorandom integer greater than or equal to zero and less than max_1. `Kernel::srand` may be used to ensure repeatable sequences of random numbers between different runs of the program.

```
srand 1234                  » 0
[ rand,  rand ]             » [0.7408769294, 0.2145348572]
[ rand(10), rand(1000) ]    » [3, 323]
srand 1234                  » 1234
[ rand,  rand ]             » [0.7408769294, 0.2145348572]
```

readline

readline(*[aString*=`$/` *]*) -> *aString*

Equivalent to `Kernel::gets`, except `readline` raises `EOFError` at end of file.

readlines

readlines(*[aString*=`$/` *]*) -> *anArray*

Returns an array containing the lines returned by calling

`Kernel.gets(aString)` until the end of file.

require

require(*aString*) -> `true` or `false`

Ruby tries to load the library named *aString*, returning `true` if successful. If the filename does not resolve to an absolute path, it will be searched for in the directories listed in `$:`. If the file has the extension ``.rb", it is loaded as a source file; if the extension is ``.so", ``.o", or ``.dll",*[Or whatever the default shared library extension is on the current platform.]* Ruby loads the shared library as a Ruby extension. Otherwise, Ruby tries adding ``.rb", ``.so", and so on to the name. The name of the loaded feature is added to the array in `$"`. A feature will not be loaded if it already appears in `$"`. `require` returns `true` if the feature was successfully loaded.

```
require "my-library.rb"
require "db-driver"
```

scan

scan(*pattern*) -> *anArray*
scan(*pattern*) {| | block } -> $_

Equivalent to calling `$_.scan`. See `String#scan` on page 373.

select

select(*readArray [*, *writeArray [errorArray [timeout]]]*) -> *anArray* or `nil`

Performs a low-level `select` call, which waits for data to become available from input/output devices. The first three parameters are arrays of `IO` objects or `nil`. The last is a timeout in seconds, which should be an `Integer` or a `Float`. The call waits for data to become available for any of the `IO` objects in *readArray*, for buffers to have cleared sufficiently to enable writing to any of the devices in *writeArray*, or for an error to occur on the devices in *errorArray*. If one or more of these conditions are met, the call returns a three-element array containing arrays of the `IO` objects that were ready. Otherwise, if there is no change in status for *timeout* seconds, the call returns `nil`. If all parameters are `nil`, the current thread sleeps forever.

```
select( [$stdin], nil, nil, 1.5 )    » [[#<IO:0x401ba090>], [], []]
```

set_trace_func

set_trace_func(*aProc*) -> *aProc*
set_trace_func(`nil`) -> `nil`

Establishes *aProc* as the handler for tracing, or disables tracing if the parameter is `nil`. *aProc* takes up to six parameters: an event name, a filename, a line number, an object id, a binding, and the name of a class. *aProc* is invoked whenever an event occurs. Events are: `c-call` (call a C-language routine), `c-return` (return from a C-language routine), `call` (call a Ruby method), `class` (start a class or module definition), `end` (finish a class or module definition), `line` (execute code on a new line), `raise` (raise an

exception), and `return` (return from a Ruby method). Tracing is disabled within the context of *aProc*.

See the example starting on page 267 for more information.

singleton_method_added

singleton_method_added(*aFixnum*) -> `nil`

Invoked with a symbol id whenever a singleton method is added to a module or a class. The default implementation in `Kernel` ignores this, but subclasses may override the method to provide specialized functionality.

```
class Test
  def Test.singleton_method_added(id)
    puts "Added #{id.id2name} to Test"
  end
  def a() end
  def Test.b() end
end
def Test.c() end
```

produces:

```
Added singleton_method_added to Test
Added b to Test
Added c to Test
```

sleep

sleep(*[aNumeric]*) -> *aFixnum*

Suspends the current thread for *aNumber* seconds (which may be a `Float` with fractional seconds). Returns the actual number of seconds slept (rounded), which may be less than that asked for if the thread was interrupted by a `SIGALRM`, or if another thread calls `Thread#run`. An argument of zero causes `sleep` to sleep forever.

```
Time.new        »  Sun Jun 09 00:19:40 CDT 2002
sleep 1.2       »  1
Time.new        »  Sun Jun 09 00:19:41 CDT 2002
sleep 1.9       »  2
Time.new        »  Sun Jun 09 00:19:43 CDT 2002
```

split

split(*[pattern [limit]]*) -> *anArray*

Equivalent to `$_.split(`*`pattern, limit`*`)`. See `String#split` on page 374.

sprintf

sprintf(*aFormatString [, arguments]**) -> *aString*

Returns the string resulting from applying *aFormatString* to any additional arguments. Within the format string, any characters other than format sequences are copied to the result. A format sequence consists of a percent sign, followed by optional flags, width, and precision indicators, then

terminated with a field type character. The field type controls how the corresponding `sprintf` argument is to be interpreted, while the flags modify that interpretation. The flag characters are shown in Table 23.1 on page 424, and the field type characters are listed in Table 23.2.

The field width is an optional integer, followed optionally by a period and a precision. The width specifies the minimum number of characters that will be written to the result for this field. For numeric fields, the precision controls the number of decimal places displayed. For string fields, the precision determines the maximum number of characters to be copied from the string. (Thus, the format sequence `%10.10s` will always contribute exactly ten characters to the result.)

`sprintf` flag characters

Flag	Applies to	Meaning
␣ (space)	bdeEfgGioxXu	Leave a space at the start of positive numbers.
#	beEfgGoxX	Use an alternative format. For the conversions `o', `x', `X', and `b', prefix the result with ``0'', ``0x'', ``0X'', and ``0b'', respectively. For `e', `E', `f', `g', and 'G', force a decimal point to be added, even if no digits follow. For `g' and 'G', do not remove trailing zeros.
+	bdeEfgGioxXu	Add a leading plus sign to positive numbers.
-	all	Left-justify the result of this conversion.
0 (zero)	all	Pad with zeros, not spaces.
*	all	Use the next argument as the field width. If negative, left-justify the result. If the asterisk is followed by a number and a dollar sign, use the indicated argument as the width.

`sprintf` field types

Field	Conversion
b	Convert argument as a binary number.
c	Argument is the numeric code for a single character.
d	Convert argument as a decimal number.
E	Equivalent to `e', but uses an uppercase E to indicate the exponent.
e	Convert floating point argument into exponential notation with one digit before the decimal point. The precision determines the number of fractional digits (defaulting to six).
f	Convert floating point argument as `[␣-]ddd.ddd`, where the precision determines the number of digits after the decimal point.
G	Equivalent to `g', but use an uppercase `E' in exponent form.
g	Convert a floating point number using exponential form if the exponent is less than -4 or greater than or equal to the precision, or in `d.dddd` form otherwise.
i	Identical to `d'.

o	Convert argument as an octal number.
s	Argument is a string to be substituted. If the format sequence contains a precision, at most that many characters will be copied.
u	Treat argument as an unsigned decimal number.
X	Convert argument as a hexadecimal number using uppercase letters.
x	Convert argument as a hexadecimal number.

```
sprintf("%d %04x", 123, 123)              » "123␣007b"
sprintf("%08b '%4s'", 123, 123)           » "01111011␣'␣123'"
sprintf("%*2$s %d", "hello", 10)          » "␣␣␣␣␣␣hello␣10"
sprintf("%*2$s %d", "hello", -10)         » "hello␣␣␣␣␣␣␣-10"
sprintf("%+g:% g:%-g", 1.23, 1.23, 1.23)  » "+1.23:␣1.23:1.23"
```

srand

srand(*[aNumber]*) -> *oldSeed*

Seeds the pseudorandom number generator to the value of *aNumber*.`to_i.abs`. If *aNumber* is omitted or zero, seeds the generator using a combination of the time, the process id, and a sequence number. (This is also the behavior if `Kernel::rand` is called without previously calling `srand`, but without the sequence.) By setting the seed to a known value, scripts can be made deterministic during testing. The previous seed value is returned. Also see `Kernel::rand` on page 421.

sub

sub(*pattern*, *replacement*) -> $_
sub(*pattern*) { *block* } -> $_

Equivalent to `$_.sub(`*`args`*`)`, except that `$_` will be updated if substitution occurs.

sub!

sub!(*pattern*, *replacement*) -> $_ or `nil`
sub!(*pattern*) { *block* } -> $_ or `nil`

Equivalent to `$_.sub!(`*`args`*`)`.

syscall

syscall(*aFixnum [*, *args]**) -> *anInteger*

Calls the operating system function identified by *aFixnum*, passing in the arguments, which must be either `String` objects, or `Integer` objects that ultimately fit within a native `long`. Up to nine parameters may be passed (14 on the Atari-ST). The function identified by *Fixnum* is system dependent. On some Unix systems, the numbers may be obtained from a header file called `syscall.h`.

```
syscall 4, 1, "hello\n", 6   # '4' is write(2) on our box
```

produces:

```
hello
```

system

system(*aCmd [, args]**) -> `true` or `false`

Executes *aCmd* in a subshell, returning `true` if the command was found and ran successfully, `false` otherwise. A detailed error code is available in `$?`. The arguments are processed in the same way as for `Kernel::exec` on page 415.

```
system("echo *")
system("echo", "*")
```

produces:

```
config.h main.rb
*
```

test

test(*aCmd*, *file1 [*, *file2]*) -> *anObject*

Uses the integer *aCmd* to perform various tests on *file1* (Table 23.3 on page 426) or on *file1* and *file2* (Table 23.4).

File tests with a single argument

Integer	Description	Returns
?A	Last access time for *file1*	Time
?b	True if *file1* is a block device	`true` or `false`
?c	True if *file1* is a character device	`true` or `false`
?C	Last change time for *file1*	Time
?d	True if *file1* exists and is a directory	`true` or `false`
?e	True if *file1* exists	`true` or `false`
?f	True if *file1* exists and is a regular file	`true` or `false`
?g	True if *file1* has the `setgid` bit set (false under NT)	`true` or `false`
?G	True if *file1* exists and has a group ownership equal to the caller's group	`true` or `false`
?k	True if *file1* exists and has the sticky bit set	`true` or `false`
?l	True if *file1* exists and is a symbolic link	`true` or `false`
?M	Last modification time for *file1*	Time
?o	True if *file1* exists and is owned by the caller's effective uid	`true` or `false`
?O	True if *file1* exists and is owned by the caller's real uid	`true` or `false`
?p	True if *file1* exists and is a fifo	`true` or `false`

?r	True if file is readable by the effective uid/gid of the caller	`true` or `false`
?R	True if file is readable by the real uid/gid of the caller	`true` or `false`
?s	If *file1* has nonzero size, return the size, otherwise return `nil`	Integer or `nil`
?S	True if *file1* exists and is a socket	`true` or `false`
?u	True if *file1* has the setuid bit set	`true` or `false`
?w	True if *file1* exists and is writable by the effective uid/gid	`true` or `false`
?W	True if *file1* exists and is writable by the real uid/gid	`true` or `false`
?x	True if *file1* exists and is executable by the effective uid/gid	`true` or `false`
?X	True if *file1* exists and is executable by the real uid/gid	`true` or `false`
?z	True if *file1* exists and has a zero length	`true` or `false`

File tests with two arguments

Integer	Description
?-	True if *file1* is a hard link to *file2*
?=	True if the modification times of *file1* and *file2* are equal
?<	True if the modification time of *file1* is prior to that of *file2*
?>	True if the modification time of *file1* is after that of *file2*

throw

throw(*aSymbol [*, *anObject]*)

Transfers control to the end of the active `catch` block waiting for *aSymbol*. Raises `NameError` if there is no `catch` block for the symbol. The optional second parameter supplies a return value for the `catch` block, which otherwise defaults to `nil`. For examples, see `Kernel::catch` on page 413.

trace_var

trace_var(*aSymbol*, *aCmd*) -> `nil`
trace_var(*aSymbol*) {| val | block }

-> `nil`

Controls tracing of assignments to global variables. The parameter *aSymbol* identifies the variable (as either a string name or a symbol identifier). *cmd* (which may be a string or a `Proc` object) or block is executed whenever the variable is assigned. The block or `Proc` object receives the variable's new

value as a parameter. Also see Kernel::untrace_var .

```
trace_var :$_, proc {|v| puts "$_ is now '#{v}'" }
$_ = "hello"
$_ = ' there'
```

produces:

```
$_ is now 'hello'
$_ is now ' there'
```

trap

trap(*signal*, *cmd*) -> *anObject*
trap(*signal*) {| | block } -> *anObject*

Specifies the handling of signals. The first parameter is a signal name (a string such as ``SIGALRM'', ``SIGUSR1'', and so on) or a signal number. The characters ``SIG'' may be omitted from the signal name. The command or block specifies code to be run when the signal is raised. If the command is the string ``IGNORE'' or ``SIG_IGN'', the signal will be ignored. If the command is ``DEFAULT'' or ``SIG_DFL'', the operating system's default handler will be invoked. If the command is ``EXIT'', the script will be terminated by the signal. Otherwise, the given command or block will be run.

The special signal name ``EXIT'' or signal number zero will be invoked just prior to program termination.

`trap` returns the previous handler for the given signal.

```
trap 0, proc { puts "Terminating: #{$$}" }
trap("CLD") { puts "Child died" }
fork && Process.wait
```

produces:

```
Terminating: 1425
Child died
Terminating: 1424
```

untrace_var

untrace_var(*aSymbol [*, *aCmd]*) -> *anArray* or `nil`

Removes tracing for the specified command on the given global variable and returns `nil`. If no command is specified, removes all tracing for that variable and returns an array containing the commands actually removed.
