# GUIDE

## Introduction

### First Steps

A handful of commands are enough to get started using `byebug`. The following session illustrates these commands. Take the following sample file:

```
#
# The n'th triangle number: triangle(n) = n*(n+1)/2 = 1 + 2 + ... + n
#
def triangle(n)
  tri = 0

  0.upto(n) { |i| tri += i }

  tri
end

t = triangle(3)
puts t
```

Let's debug it.

```
$ byebug /path/to/triangle.rb

[1, 10] in /path/to/triangle.rb
    1: #
    2: # The n'th triangle number: triangle(n) = n*(n+1)/2 = 1 + 2 + ... + n
    3: #
=>  4: def triangle(n)
    5:   tri = 0
    6:
    7:   0.upto(n) { |i| tri += i }
    8:
    9:   tri
   10: end
(byebug)
```

We are currently stopped before the first executable line of the program: line 4 of `triangle.rb`. If you are used to less dynamic languages and have used debuggers for more statically compiled languages like C, C++, or Java, it may seem odd to be stopped before a function definition but in Ruby line 4 is executed.

Byebug's prompt is `(byebug)`. If the program has died and you are in post-mortem debugging, `(byebug:post-mortem)` is used instead. If the program has terminated normally and the `--no-quit` option has been specified in the command line, the prompt will be `(byebug:ctrl)` instead. The commands available change depending on the program's state.

Byebug automatically lists 10 lines of code centered around the current line every time it is stopped. The current line is marked with `=>`. If the range would overflow the beggining or the end of the file, byebug will move it accordingly so that only actual real lines of code are displayed.

Now let us step through the program.

```
(byebug) step

[5, 14] in /path/to/triangle.rb
    5:   tri = 0
    6:
    7:   0.upto(n) { |i| tri += i }
    9:   end
   10:
   11:   tri
   12: end
   13:
=> 14: triangle(3)
(byebug) <RET> # hit enter

[1, 10] in /path/to/triangle.rb
    1: #
    2: # The n'th triangle number: triangle(n) = n*(n+1)/2 = 1 + 2 + ... + n
    3: #
    4: def triangle(n)
=>  5:   tri = 0
    6:
    7:   0.upto(n) { |i| tri += i }
    8:
    9:   tri
   10: end
(byebug) eval tri
nil
(byebug) step

[2, 11] in /path/to/triangle.rb
    2: # The n'th triangle number: triangle(n) = n*(n+1)/2 = 1 + 2 + ... + n
    3: #
    4: def triangle(n)
    5:   tri = 0
    6:
=>  7:   0.upto(n) { |i| tri += i }
    8:
    9:   tri
   10: end
   11:
(byebug) eval tri
0
```

The first `step` command runs the script one executable unit. The second command we entered was just hitting the return key: `byebug` remembers the last command you entered was `step` and runs it again.

One way to print the values of variables is `eval` (there are other ways). When we look at the value of `tri` the first time, we see it is `nil`. Again we are stopped *before* the assignment on line 5, and this variable hadn't been set previously. However after issuing another `step` command we see that the value is 0 as expected. If every time we stop we want to see the value of `tri` to see how things are going, there is a better way by setting a display expression:

```
(byebug) display tri
1: tri = 0
```

Now let us run the program until right before we return from the function. We'll want to see which lines get run, so we turn on *line tracing*. If we don't want whole paths to be displayed when tracing, we can turn on *basename*.

```
(byebug) set linetrace
linetrace is on
(byebug) set basename
basename is on
(byebug) finish 0
Tracing: triangle.rb:7   0.upto(n) { |i| tri += i }
1: tri = 0
Tracing: triangle.rb:7   0.upto(n) { |i| tri += i }
1: tri = 0
Tracing: triangle.rb:7   0.upto(n) { |i| tri += i }
1: tri = 1
Tracing: triangle.rb:7   0.upto(n) { |i| tri += i }
1: tri = 3
Tracing: triangle.rb:9   tri
1: tri = 6
1: tri = 6

[4, 13] in /home/davidr/Proyectos/byebug/triangle.rb
    4: def triangle(n)
    5:   tri = 0
    6:
    7:   0.upto(n) { |i| tri += i }
    8:
    9:   tri
=> 10: end
   11:
   12: t = triangle(3)
   13: puts t
(byebug) quit
Really quit? (y/n)
y
```

So far, so good. As you can see from the above, to get out of `byebug`, one can issue a `quit` command (or the abbreviation `q`). If you want to quit without being prompted, suffix the command with an exclamation mark, e.g., `q!`.

## Second Sample Session: Delving Deeper

In this section we'll introduce breakpoints, the call stack and restarting. Below we will debug a simple Ruby program to solve the classic Towers of Hanoi puzzle. It is augmented by the bane of programming: some command-parameter processing with error checking.

```
#
# Solves the classic Towers of Hanoi puzzle.
#
def hanoi(n, a, b, c)
  hanoi(n - 1, a, c, b) if n - 1 > 0

  puts "Move disk #{a} to #{b}"

  hanoi(n - 1, c, b, a) if n - 1 > 0
end

n_args = $ARGV.length

fail('*** Need number of disks or no parameter') if n_args > 1
```

Recall in the first section it was stated that before the `def` is run, the method it names is undefined. Let's check that out. First let's see what private methods we can call before running `def hanoi`.

```
$ byebug path/to/hanoi.rb

    1: #
    2: # Solves the classic Towers of Hanoi puzzle.
    3: #
    4: def hanoi(n, a, b, c)
    5:   hanoi(n - 1, a, c, b) if n - 1 > 0
    6:
    7:   puts "Move disk #{a} to #{b}"
    8:
    9:   hanoi(n - 1, c, b, a) if n - 1 > 0
   10: end
(byebug) private_methods
public
private
include
using
define_method
default_src_encoding
DelegateClass
Digest
timeout
initialize_copy
initialize_dup
initialize_clone
sprintf
format
Integer
Float
String
Array
Hash
warn
raise
fail
global_variables
__method__
__callee__
__dir__
eval
local_variables
iterator?
block_given?
catch
throw
loop
respond_to_missing?
trace_var
untrace_var
at_exit
syscall
open
printf
print
putc
puts
gets
readline
select
readlines
`
p
test
srand
rand
trap
load
require
require_relative
```

```
autoload
autoload?
proc
lambda
binding
caller
caller_locations
exec
fork
exit!
system
spawn
sleep
exit
abort
Rational
Complex
set_trace_func
gem_original_require
Pathname
pp
y
URI
rubygems_require
initialize
singleton_method_added
singleton_method_removed
singleton_method_undefined
method_missing
(byebug) private_methods.member?(:hanoi)
false
```

`private_methods` is not a byebug command but a Ruby feature. By default, when `byebug` doesn't understand a command, it will evaluate it as if it was a Ruby command. You can use any Ruby to inspect your program's state at the place it is stopped.

Now let's see what happens after stepping:

```
(byebug) step

[5, 14] in /path/to/hanoi.rb
    5:   hanoi(n - 1, a, c, b) if n - 1 > 0
    6:
    7:   puts "Move disk #{a} to #{b}"
    8:
    9:   hanoi(n - 1, c, b, a) if n - 1 > 0
   10: end
   11:
=> 12: n_args = $ARGV.length
   13:
   14: fail('*** Need number of disks or no parameter') if n_args > 1
(byebug) private_methods.member?(:hanoi)
true
(byebug)
```

Okay, lets go on and talk about program arguments.

```
(byebug) $ARGV
[]
```

Oops. We forgot to specify any parameters to this program. Let's try again. We can use the `restart` command here.

```
(byebug) restart 3
Re exec'ing:
  /path/to/bin/byebug /path/to/hanoi.rb 3
```

```
[1, 10] in /path/to/hanoi.rb
    1: #
    2: # Solves the classic Towers of Hanoi puzzle.
    3: #
=>  4: def hanoi(n, a, b, c)
    5:   hanoi(n - 1, a, c, b) if n - 1 > 0
    6:
    7:   puts "Move disk #{a} to #{b}"
    8:
    9:   hanoi(n - 1, c, b, a) if n - 1 > 0
   10: end
(byebug) break 5
Created breakpoint 1 at /path/to/hanoi.rb:5
(byebug) continue
Stopped by breakpoint 1 at /path/to/hanoi.rb:5

[1, 10] in /path/to/hanoi.rb
    1: #
    2: # Solves the classic Towers of Hanoi puzzle.
    3: #
    4: def hanoi(n, a, b, c)
=>  5:   hanoi(n - 1, a, c, b) if n - 1 > 0
    6:
    7:   puts "Move disk #{a} to #{b}"
    8:
    9:   hanoi(n - 1, c, b, a) if n - 1 > 0
   10: end
(byebug) display n
1: n = 3
(byebug) display a
2: a = :a
(byebug) display b
3: b = :b
(byebug) undisplay 3
(byebug) continue
Stopped by breakpoint 1 at /path/to/hanoi.rb:5
1: n = 2
2: a = :a
[1, 10] in /path/to/hanoi.rb
    1: #
    2: # Solves the classic Towers of Hanoi puzzle.
    3: #
    4: def hanoi(n, a, b, c)
=>  5:   hanoi(n - 1, a, c, b) if n - 1 > 0
    6:
    7:   puts "Move disk #{a} to #{b}"
    8:
    9:   hanoi(n - 1, c, b, a) if n - 1 > 0
   10: end

(byebug) c
Stopped by breakpoint 1 at /path/to/hanoi.rb:5
1: n = 1
2: a = :a

[1, 10] in /path/to/hanoi.rb
    1: #
    2: # Solves the classic Towers of Hanoi puzzle.
    3: #
    4: def hanoi(n, a, b, c)
=>  5:   hanoi(n - 1, a, c, b) if n - 1 > 0
    6:
    7:   puts "Move disk #{a} to #{b}"
    8:
    9:   hanoi(n - 1, c, b, a) if n - 1 > 0
   10: end
(byebug) set nofullpath
fullpath is off
(byebug) where
--> #0  Object.hanoi(n#Fixnum, a#Symbol, b#Symbol, c#Symbol) at .../shortpath/to/hanoi.rb:5
    #1  Object.hanoi(n#Fixnum, a#Symbol, b#Symbol, c#Symbol) at .../shortpath/to/hanoi.rb:5
```

```
    #2  <top (required)> at .../Proyectos/byebug/hanoi.rb:28
(byebug)
```

In the above we added new commands: `break` (see breakpoints), which indicates to stop just before that line of code is run, and `continue`, which resumes execution. To remove a display expression `undisplay` is used. If we give a display number, just that display expression is removed.

We also used a new command `where` (see backtrace) to show the callstack. In the above situation, starting from the bottom line we see we called the `hanoi` method from line 28 of the file `hanoi.rb` and the `hanoi` method called itself two more times at line 5.

In the callstack we show a *current frame* mark, the frame number, the method being called, the names of the parameters, the types those parameters *currently* have and the file-line position. Remember it's possible that when the program was called the parameters had different types, since the types of variables can change dynamically. You can alter the style of what to show in the trace (see callstyle).

Now let's move around the callstack.

```
(byebug) undisplay
Clear all expressions? (y/n) y
(byebug) n_args
NameError Exception: undefined local variable or method `n_args' for main:Object
(byebug) frame 2

[19, 28] in /path/to/hanoi.rb
   19:   begin
   20:     n = $ARGV[0].to_i
   21:   rescue ValueError
   22:     raise("** Expecting an integer, got: #{$ARGV[0]}")
   23:   end
   24: end
   25:
   26: fail('*** Number of disks should be between 1 and 100') if n < 1 || n > 100
   27:
=> 28: hanoi(n, :a, :b, :c)
(byebug) n_args
0
(byebug) eval n
3
(byebug) down 2

[1, 10] in /path/to/hanoi.rb
    1: #
    2: # Solves the classic Towers of Hanoi puzzle.
    3: #
    4: def hanoi(n, a, b, c)
=>  5:   hanoi(n - 1, a, c, b) if n - 1 > 0
    6:
    7:   puts "Move disk #{a} to #{b}"
    8:
    9:   hanoi(n - 1, c, b, a) if n - 1 > 0
   10: end
(byebug) eval n
2
```

Notice in the above to get the value of variable `n` we had to use a print command like `eval n`. If we entered just `n`, that would be taken to mean byebug command `next`. In the current scope, variable `n_args` is not defined. However I can change to the top-most frame by using the `frame 2` command. Notice that inside frame #2, the value of `n_args` can be shown. Also note that the value of variable `n` is different.

## Attaching to a running program with byebug

In the previous sessions we've been calling byebug right at the outset, but there is another mode of operation you might use. If there's a lot of code that needs to be run before the part you want to inspect, it might not be efficient or convenient to run

byebug from the outset.

In this section we'll show how to enter the code in the middle of your program, while delving more into byebug's operation. We will also use unit testing. Using unit tests will greatly reduce the amount of debugging needed, while at the same time, will increase the quality of your program.

What we'll do is take the `triangle` code from the first session and write a unit test for that. In a sense we did write a tiny test for the program which was basically the last line where we printed the value of `triangle(3)`. This test however wasn't automated: the expectation is that someone would look at the output and verify that what was printed is what was expected.

Before we can turn that into something that can be `required`, we probably want to remove that output. However I like to keep in that line so that when I look at the file, I have an example of how to run it. Therefore we will conditionally run this line if that file is invoked directly, but skip it if it is not. *NOTE: `byebug` resets `$0` to try to make things like this work.*

```
if __FILE__ == $PROGRAM_NAME
  t = triangle(3)
  puts t
end
```

Okay, we're now ready to write our unit test and we'll use the `minitest` framework for that. Here's the test code, it should be placed in the same directory as `triangle.rb`.

```
require 'minitest/autorun'
require_relative 'triangle.rb'

class TestTriangle < Minitest::Test
  def test_basic
    solutions = []

    0.upto(5) { |i| solutions << triangle(i) }

    assert_equal([0, 1, 3, 6, 10, 15], solutions, 'First 5 triangle numbers')
  end
end
```

Let's say we want to stop before the first statement in our test method, we'll add the following:

```
...
def test_basic
  byebug
  solutions = []
...
```

Now we run the program, requiring `byebug`

```
$ ruby -rbyebug test_triangle.rb
Run options: --seed 31679

# Running:

[2, 11] in test_triangle.rb
    2: require_relative 'triangle.rb'
    3:
    4: class TestTriangle < Minitest::Test
    5:   def test_basic
    6:     byebug
=>  7:     solutions = []
    8:
    9:     0.upto(5) { |i| solutions << triangle(i) }
   10:
   11:     assert_equal([0, 1, 3, 6, 10, 15], solutions, 'First 5 triangle numbers')
(byebug)
```

and we see that we are stopped at line 7 just before the initialization of the list `solutions`.

Now let's see where we are...

```
(byebug) set nofullpath
Displaying frame's full file names is off.
(byebug) bt
--> #0  TestTriangle.test_basic at .../Proyectos/byebug/test_triangle.rb:7
    #1  block (3 levels) in Minitest::Test.run at .../lib/minitest/test.rb:108
    #2  Minitest::Test.capture_exceptions at .../lib/minitest/test.rb:206
    #3  block (2 levels) in Minitest::Test.run at .../lib/minitest/test.rb:105
    #4  Minitest::Test.time_it at .../lib/minitest/test.rb:258
    #5  block in Minitest::Test.run at .../lib/minitest/test.rb:104
    #6  #<Class:Minitest::Runnable>.on_signal(name#String, action#Proc) at .../minitest-5.5.0/lib/minitest.rb:321
    #7  Minitest::Test.with_info_handler(&block#Proc) at .../lib/minitest/test.rb:278
    #8  Minitest::Test.run at .../lib/minitest/test.rb:103
    #9  #<Class:Minitest>.run_one_method(klass#Class, method_name#String) at .../minitest-5.5.0/lib/minitest.rb:768
    #10 #<Class:Minitest::Runnable>.run_one_method(klass#Class, method_name#String, reporter#Minitest::CompositeRepor
    #11 block (2 levels) in #<Class:Minitest::Runnable>.run(reporter#Minitest::CompositeReporter, options#Hash) at ..
    ├-- #12 Array.each at .../minitest-5.5.0/lib/minitest.rb:288
    #13 block in #<Class:Minitest::Runnable>.run(reporter#Minitest::CompositeReporter, options#Hash) at .../minitest-
    #14 #<Class:Minitest::Runnable>.on_signal(name#String, action#Proc) at .../minitest-5.5.0/lib/minitest.rb:321
    #15 #<Class:Minitest::Runnable>.with_info_handler(reporter#Minitest::CompositeReporter, &block#Proc) at .../minit
    #16 #<Class:Minitest::Runnable>.run(reporter#Minitest::CompositeReporter, options#Hash) at .../minitest-5.5.0/lib
    #17 block in #<Class:Minitest>.__run(reporter#Minitest::CompositeReporter, options#Hash) at .../minitest-5.5.0/li
    ├-- #18 Array.map at .../minitest-5.5.0/lib/minitest.rb:150
    #19 #<Class:Minitest>.__run(reporter#Minitest::CompositeReporter, options#Hash) at .../minitest-5.5.0/lib/minites
    #20 #<Class:Minitest>.run(args#Array) at .../minitest-5.5.0/lib/minitest.rb:127
    #21 block in #<Class:Minitest>.autorun at .../minitest-5.5.0/lib/minitest.rb:56
(byebug)
```

We get the same result as if we had run byebug from the outset.

## Debugging Oddities: How debugging Ruby may be different from other languages

If you are used to debugging in other languages like C, C++, Perl, Java or even Bash (see bashdb), there may be a number of things that seem or feel a little bit different and may confuse you. A number of these things aren't oddities of the debugger per se but differences in how Ruby works compared to those other languages. Because Ruby works a little differently from those other languages, writing a debugger has to also be a little different as well if it is to be useful. In this respect, using Byebug may help you understand Ruby better.

We've already seen one such difference: the fact that we stop on method definitions or `def`'s and that is because these are in fact executable statements. In other compiled languages this would not happen because that's already been done when you compile the program (or in Perl when it scans in the program). In this section we'll consider some other things that might throw off new users to Ruby who are familiar with other languages and debugging in them.

### Bouncing Around in Blocks (iterators)

When debugging languages with coroutines like Python and Ruby, a method call may not necessarily go to the first statement after the method header. It's possible that the call will continue after a `yield` statement from a prior call.

```
#
# Enumerator for primes
#
class SievePrime
  def initialize
    @odd_primes = []
  end

  def next_prime
    candidate = 2
    yield candidate
    not_prime = false
    candidate += 1
```

```
    loop do
      @odd_primes.each do |p|
        not_prime = (0 == (candidate % p))
        break if not_prime
      end

      unless not_prime
        @odd_primes << candidate
        yield candidate
      end

      candidate += 2
    end
  end
end

SievePrime.new.next_prime do |prime|
  puts prime
  break if prime > 10
end
```

```
$ byebug primes.rb
[1, 10] in /path/to/primes.rb
    1: #
    2: # Enumerator for primes
    3: #
=>  4: class SievePrime
    5:   def initialize
    6:     @odd_primes = []
    7:   end
    8:
    9:   def self.next_prime(&block)
   10:    candidate = 2
(byebug) set linetrace
line tracing is on.
(byebug) set basename
basename in on.
(byebug) step 9
Tracing: primes.rb:5   def initialize
Tracing: primes.rb:9   def next_prime
Tracing: primes.rb:31 SievePrime.new.next_prime do |prime|
Tracing: primes.rb:6     @odd_primes = []
Tracing: primes.rb:10     candidate = 2
Tracing: primes.rb:11     yield candidate
Tracing: primes.rb:32   puts prime
2
Tracing: primes.rb:33   break if prime > 10
Tracing: primes.rb:12     not_prime = false

[7, 16] in /path/to/primes.rb
    7:   end
    8:
    9:   def next_prime
   10:     candidate = 2
   11:     yield candidate
=> 12:     not_prime = false
   13:     candidate += 1
   14:
   15:     loop do
   16:       @odd_primes.each do |p|
   17:         not_prime = (0 == (candidate % p))
(byebug)
```

The loop between lines 31-34 gets interleaved between those of `SievePrime#next_prime`, lines 9-28 above.

**No Parameter Values in a Call Stack**

In traditional debuggers, in a call stack you can generally see the names of the parameters and the values that were passed in.

Ruby is a very dynamic language and it tries to be efficient within the confines of the language definition. Values generally aren't taken out of a variable or expression and pushed onto a stack. Instead a new scope is created and the parameters are given initial values. Parameter passing is by *reference* not by *value* as it is say Algol, C, or Perl. During the execution of a method, parameter values can change (and often do). In fact even the *class* of the object can change.

So at present, the name of the parameter is shown. The call-style setting (callstyle) can be used to set whether the name is shown or the name and the *current* class of the object.

#### Lines You Can Stop At

Consider the following little Ruby program.

```
'Yes it does' =~ /
(Yes) \s+
it  \s+
does
/ix
puts $1
```

The stopping points that Ruby records are the last two lines, lines 5 and 6.

Inside `byebug` you can get a list of stoppable lines for a file using the `info file` command.

## Threading support

Byebug supports debugging Ruby programs making use of multiple threads.

Let's consider the following sample program:

```
class Company
  def initialize(task)
    @tasks, @results = Queue.new, Queue.new

    @tasks.push(task)
  end

  def run
    manager = Thread.new { manager_routine }
    employee = Thread.new { employee_routine }

    sleep 6

    go_home(manager)
    go_home(employee)
  end

  #
  # An employee doing his thing
  #
  def employee_routine
    loop do
      if @tasks.empty?
        have_a_break(0.1)
      else
        work_hard(@tasks.pop)
      end
    end
  end

  #
  # A manager doing his thing
  #
  def manager_routine
```

```
    loop do
      if @results.empty?
        have_a_break(1)
      else
        show_off(@results.pop)
      end
    end
  end

  private

  def show_off(result)
    puts result
  end

  def work_hard(task)
    task ** task
  end

  def have_a_break(amount)
    sleep amount
  end

  def go_home(person)
    person.kill
  end
end

Company.new(10).run
```

The `Company` class simulates a real company. The company has a manager and an employee represented by 2 threads: they work concurrently to achieve the company's targets.

- The employee looks for tasks to complete. If there are tasks, it works hard to complete them. Otherwise he has a quick break.

```
#
# An employee doing his thing
#
def employee_routine
  loop do
    if @tasks.empty?
      have_a_break(0.1)
    else
      work_hard(@tasks.pop)
    end
  end
end
```

- The manager, on the other hand, sits there all day and sporadically checks whether there are any results to show off.

```
#
# A manager doing his thing
#
def manager_routine
  loop do
    if @results.empty?
      have_a_break(1)
    else
      show_off(@results.pop)
    end
  end
end
```

We do some abstractions easily readable in the code. Our tasks are just a `Queue` of numbers, so are our results. What our employer does when he works is some calculation with those numbers and what the manager does with the results is printing them to the screen.

We instantiate a new company with an initial task and after running that company we expect the result to be printed in the screen, but it is not. Lets debug our sample program:

```
[1, 10] in /path/to/company.rb
=>  1: class Company
    2:   def initialize(task)
    3:     @tasks, @results = Queue.new, Queue.new
    4:
    5:     @tasks.push(task)
    6:   end
    7:
    8:   def run
    9:     manager = Thread.new { manager_routine }
   10:     employee = Thread.new { employee_routine }
(byebug) l

[11, 20] in /path/to/company.rb
   11:
   12:     sleep 6
   13:
   14:     go_home(manager)
   15:     go_home(employee)
   16:   end
   17:
   18:   #
   19:   # An employee doing his thing
   20:   #

(byebug) c 12
Stopped by breakpoint 1 at /path/to/company.rb:12

[7, 16] in /path/to/company.rb
    7:
    8:   def run
    9:     manager = Thread.new { manager_routine }
   10:     employee = Thread.new { employee_routine }
   11:
=> 12:     sleep 6
   13:
   14:     go_home(manager)
   15:     go_home(employee)
   16:   end
(byebug) th l
+ 1 #<Thread:0x0000000192f328 run> /path/to/company.rb:12
  2 #<Thread:0x00000001ff9870@/path/to/company.rb:9 sleep>
  3 #<Thread:0x00000001ff80d8@/path/to/company.rb:10 sleep>
```

What we have done here is just start our program and advance to the point inmediately after our `employee` and `manager` threads have been created. We can then check that the threads are there using the `thread list` command. Now we want to debug both of this threads to check what's happening and look for the bug.

```
(byebug) th switch 3

[5, 14] in /path/to/company.rb
    5:     @tasks.push(task)
    6:   end
    7:
    8:   def run
    9:     manager = Thread.new { manager_routine }
=> 10:     employee = Thread.new { employee_routine }
   11:
   12:     sleep 6
   13:
```

```
   14:     go_home(manager)
(byebug) th stop 1; th stop 2
$ 1 #<Thread:0x00000001307310 sleep> /path/to/company.rb:12
$ 2 #<Thread:0x000000018bf438 sleep> /path/to/company.rb:9
(byebug) th l
$ 1 #<Thread:0x00000001307310 sleep> /path/to/company.rb:12
$ 2 #<Thread:0x000000018bf438@/path/to/company.rb:9 sleep> /path/to/company.rb:55
+ 3 #<Thread:0x00000001ff80d8@/path/to/company.rb:10 sleep> /path/to/company.rb:10
```

We have started by debugging the `employee` thread. To do that, we switch to that thread using the `thread switch 3` command. The thread number is the one specified by `thread list`, we know this is our worker thread because `thread list` specifies where the thread is defined in the file (and its current position if the thread is currently running, although this is only available since Ruby 2.2.1).

After that we stopped the main thread and the worker thread, using the command `thread stop`. We do this because we want to focus on the employee thread first and don't want the program to finish while we are debugging. Notice that stopped threads are marked with the "$" symbol whereas the current thread is marked with the "+" symbol.

```
(byebug) s

[17, 26] in /path/to/company.rb
   17:
   18:   #
   19:   # An employee doing his thing
   20:   #
   21:   def employee_routine
=> 22:     loop do
   23:       if @tasks.empty?
   24:         have_a_break(0.1)
   25:       else
   26:         work_hard(@tasks.pop)
(byebug) s

[18, 27] in /path/to/company.rb
   18:   #
   19:   # An employee doing his thing
   20:   #
   21:   def employee_routine
   22:     loop do
=> 23:       if @tasks.empty?
   24:         have_a_break(0.1)
   25:       else
   26:         work_hard(@tasks.pop)
   27:       end
(byebug) n

[21, 30] in /path/to/company.rb
   21:   def employee_routine
   22:     loop do
   23:       if @tasks.empty?
   24:         have_a_break(0.1)
   25:       else
=> 26:         work_hard(@tasks.pop)
   27:       end
   28:     end
   29:   end
   30:
(byebug) s

[49, 58] in /path/to/company.rb
   49:   def show_off(result)
   50:     puts result
   51:   end
   52:
   53:   def work_hard(task)
=> 54:     task ** task
   55:   end
   56:
```

```
   57:   def have_a_break(amount)
   58:     sleep amount
(byebug) s

[21, 30] in /path/to/company.rb
   21:   #
   22:   # An employee doing his thing
   23:   #
   24:   def employee_routine
   25:     loop do
=> 26:       if @tasks.empty?
   27:         have_a_break(0.1)
   28:       else
   29:         work_hard(@tasks.pop)
   30:       end
(byebug) n

[22, 31] in /path/to/company.rb
   22:   # An employee doing his thing
   23:   #
   24:   def employee_routine
   25:     loop do
   26:       if @tasks.empty?
=> 27:         have_a_break(0.1)
   28:       else
   29:         work_hard(@tasks.pop)
   30:       end
   31:     end
(byebug) n

[21, 30] in /path/to/company.rb
   21:   #
   22:   # An employee doing his thing
   23:   #
   24:   def employee_routine
   25:     loop do
=> 26:       if @tasks.empty?
   27:         have_a_break(0.1)
   28:       else
   29:         work_hard(@tasks.pop)
   30:       end
   31:     end
(byebug)
```

Everything seems fine in this thread. The first iteration the employee will do his job, and after that it will just check for new tasks and sleep. Let's debug the manager task now:

```
(byebug) th resume 2
  2 #<Thread:0x000000019892d8@/path/to/company.rb:12 run> /path/to/company.rb:12
(byebug) th switch 2
  2 #<Thread:0x000000019892d8@/path/to/company.rb:12 sleep> /path/to/company.rb:12

[7, 16] in /path/to/company.rb
    7:
    8:   #
    9:   # A CEO running his company
   10:   #
   11:   def run
=> 12:     manager = Thread.new { manager_routine }
   13:     employee = Thread.new { employee_routine }
   14:
   15:     sleep 6
   16:
(byebug)
```

We used the command `thread resume` to restart the manager's thread and then switch to it using `thread switch`. It's important to resume the thread's execution before switching to it, otherwise we'll get a hang because we cannot run a

sleeping thread.

Now we can investigate the problem in the employer's side:

```
(byebug) s
[30, 39] in /path/to/company.rb
   30:
   31:   #
   32:   # A manager doing his thing
   33:   #
   34:   def manager_routine
=> 35:     loop do
   36:       if @results.empty?
   37:         have_a_break(1)
   38:       else
   39:         show_off(@results.pop)
(byebug) s

[31, 40] in /path/to/company.rb
   31:   #
   32:   # A manager doing his thing
   33:   #
   34:   def manager_routine
   35:     loop do
=> 36:       if @results.empty?
   37:         have_a_break(1)
   38:       else
   39:         show_off(@results.pop)
   40:       end
(byebug) n

[32, 41] in /path/to/company.rb
   32:   # A manager doing his thing
   33:   #
   34:   def manager_routine
   35:     loop do
   36:       if @results.empty?
=> 37:         have_a_break(1)
   38:       else
   39:         show_off(@results.pop)
   40:       end
   41:     end
(byebug) n

[31, 40] in /path/to/company.rb
   31:   #
   32:   # A manager doing his thing
   33:   #
   34:   def manager_routine
   35:     loop do
=> 36:       if @results.empty?
   37:         have_a_break(1)
   38:       else
   39:         show_off(@results.pop)
   40:       end
(byebug)
```

Now we can see the problem, the `@results` variable is always empty! The employee forgot to leave the results in his manager's deck. We fix it by changing the line

```
work_hard(@tasks.pop)
```

in the `employee_routine` method with the line

```
@results << work_hard(@tasks.pop)
```

To be continued...

- More complex examples with objects, pretty printing and irb.
- Line tracing and non-interactive tracing.
- Post-mortem debugging.

## Getting in & out

### Starting byebug

There is a wrapper script called `byebug` which basically `require` 's the gem then loads `byebug` before its argument (the program to be debugged) is started. If you don't need to pass dash options to your program, which might be confused with byebug options, then you don't need to add the `--` . To get a brief list of options and descriptions, use the `--help` option.

```
$ byebug --help

  byebug 3.5.1

  Usage: byebug [options] <script.rb> -- <script.rb parameters>

    -d, --debug              Set $DEBUG=true
    -I, --include list       Add to paths to $LOAD_PATH
    -m, --[no-]post-mortem   Use post-mortem mode
    -q, --[no-]quit          Quit when script finishes
    -x, --[no-]rc            Run byebug initialization file
    -s, --[no-]stop          Stop when script is loaded
    -r, --require file       Require library before script
    -R, --remote [host:]port Remote debug [host:]port
    -t, --[no-]trace         Turn on line tracing
    -v, --version            Print program version
    -h, --help               Display this message
```

Many options appear as a long option name, such as `--help` and a short one letter option name, such as `-h` . The list of options is detailed below:

**-h | --help**

It causes `byebug` to print some basic help and exit.

**-v | --version**

It causes `byebug` to print its version number and exit.

**-d | --debug**

Sets `$DEBUG` to `true` . Compatible with Ruby's flag.

**-I | --include path**

Adds `path` to load path. `path` can be a single path or a colon separated path list.

**-m | --post-mortem**

If your program raises an exception that isn't caught you can enter byebug for inspection of what went wrong. You may also want to use this option in conjunction with `--no-stop` . See also Post-Mortem Debugging.

**--no-quit**

Keep inside `byebug` after your program terminates normally.

**--no-stop**

Normally `byebug` stops before executing the first statement. If instead you want it to start running initially and perhaps break it later in the execution, use this option.

**-r | --require lib**

Requires the library before executing the script. This option is compatible with Ruby's.

**-t | --trace**

Turns on line tracing. Running `byebug --trace <rubyscript>.rb` is pretty much like running `ruby -rtracer <rubyscript>.rb`. If all you want to do however is get a line trace, `tracer` is most likely faster than `byebug`.

```
$ time byebug --trace --no-stop hanoi.rb > /dev/null

real 0m0.743s
user 0m0.668s
sys  0m0.068s
$ time ruby -rtracer hanoi.rb > /dev/null

real 0m0.077s
user 0m0.072s
sys  0m0.004s
```

## Byebug default options

Byebug has many command-line options,; it seems that some people want to set them differently from the defaults. For example, some people may want `--no-quit` to be the default behavior. One could write a wrapper script or set a shell alias to handle this.

## Command Files

A command file is a file of lines that are `byebug` commands. Comments (lines starting with `#`) may also be included. An empty line in a command file does nothing; it does not mean to repeat the last command, as it would from the terminal.

When you start `byebug`, it automatically executes commands from its *init file*, called `.byebugrc`. During startup, `byebug` does the following:

- **Processes command line options and operands.** Reads the init file in your current directory, if any, and then checks your home directory. The home directory is the directory named in the `$HOME` or `$HOMEPATH` environment variable. Thus, you can have more than one init file, one generic in your home directory, and another, specific to the program you are debugging, in the directory where you invoke `byebug`.

You can also request the execution of a command file with the `source` command (see Source).

## Quitting byebug

To exit `byebug`, use the `quit` command (abbreviated to `q`). Normally, if you are in an interactive session, this command will prompt to ask if you really want to quit. If you want to quit without being prompted, enter `quit unconditionally` (abbreviated to `q!`).

Another way to terminate byebug is to use the `kill` command. This does the more forceful `kill -9`. It can be used in cases where `quit` doesn't work (I haven't seen those yet).

## Calling byebug from inside your program

Running a program from byebug adds a bit of overhead and slows it down a little. Furthermore, by necessity, debuggers change the operation of the program they are debugging. And this can lead to unexpected and unwanted differences. It has happened so often that the term Heisenbugs was coined to describe the situation where using a debugger (among other possibilities) changes the behavior of the program so that the bug doesn't manifest itself anymore.

There is another way to get into byebug which adds no overhead or slowdown until you reach the point at which you want to start debugging. However here you must change the script and make an explicit call to byebug. Because byebug isn't involved before the first call, there is no overhead and the script will run at the same speed as if there were no byebug.

To enter byebug this way, just drop `byebug` in whichever line you want to start debugging at. You also have to require byebug somehow. If using bundler, it will take care of that for you, otherwise you can use the ruby `-r` flag or add `require 'byebug'` in the line previous to the `byebug` call.

If speed is crucial, you may want to start and stop this around certain sections of code, using `Byebug.start` and `Byebug.stop`. Alternatively, instead of issuing an explicit `Byebug.stop` you can add a block to the `Byebug.start` and debugging is turned on for that block. If the block of code raises an uncaught exception that would cause the block to terminate, the `stop` will occur. See Byebug.start with a block.

When `byebug` is run, `.byebugrc` is read.

You may want to enter byebug at several points in the program where there is a problem you want to investigate. And since `byebug` is just a method call it's possible to enclose it in a conditional expression, for example

```
byebug if 'bar' == foo and 20 == iter_count
```

### Restarting Byebug

You can restart the program using `restart [program args]`. This is a re-exec - all byebug state is lost. If command arguments are passed, those are used. Otherwise program arguments from the last invocation are used.

You won't be able to restart your program in all cases. First, the program should have been invoked at the outset rather than having been called from inside your program or invoked as a result of post-mortem handling.

Also, since this relies on the OS `exec` call, this command is available only if your OS supports `exec`.

## Debugging remote programs

It is possible to set up debugging so that you can issue byebug commands from outside the process running the Ruby code. In fact, you might even be on a different computer than the one running the Ruby program.

To setup remote debugging, drop the following somewhere before the point in the program that you want to debug (In Rails, the `config/environments/development.rb` could be a good candidate).

```
require 'byebug/core'
Byebug.wait_connection = true
Byebug.start_server('localhost', <port>)
```

Once this piece gets executed, you can connect to the remote debugger from your local machine, by running: `byebug -R localhost:<port>`.

Next, at a place of program execution which gets run just before the code you want to debug, add a call to `byebug` as was done without remote execution:

```
# work, work, work...
byebug
some ruby code  # byebug will stop before this line is run
```

## Byebug Command Reference

### Command Syntax

Usually a command is put on a single line. There is no limit on how long it can be. It starts with a command name, which is followed by arguments whose meaning depends on the command name. For example, the command `step` accepts an argument which is the number of times to step, as in `step 5`. You can also use the `step` command with no arguments. Some commands do not allow any arguments.

Multiple commands can be put on a line by separating each with a semicolon `;`. You can disable the meaning of a semicolon to separate commands by escaping it with a backslash.

For example, you might want to enter the following code to compute the 5th Fibonacci number.

```
(byebug) fib1=0; fib2=1; 5.times {|temp| temp=fib1; fib1=fib2; fib2 += temp }
0
1
SyntaxError Exception: /home/davidr/Proyectos/sample_app/trace.rb:1: syntax
error, unexpected end-of-input, expecting '}'
 5.times { |temp| temp=fib1
                           ^
nil
1
SyntaxError Exception: /home/davidr/Proyectos/sample_app/trace.rb:1: syntax
error, unexpected tSTRING_DEND, expecting end-of-input
 fib2 += temp }
               ^
nil
(byebug) fib1=0\; fib2=1\; 5.times {|temp| temp=fib1\; fib1=fib2\; fib2 += temp }
5
(byebug) fib2
8
```

You might also consider using the irb or pry commands and then you won't have to escape semicolons.

A blank line as input (typing just `<RET>`) means to repeat the previous command.

Byebug uses readline, which handles line editing and retrieval of previous commands. Up arrow, for example, moves to the previous byebug command; down arrow moves to the next more recent command (provided you are not already at the last command). Command history is saved in file `.byebug_history`. A limit is put on the history size. You can see this with the `show history size` command. See history for history parameters.

## Command Output

In the command-line interface, when `byebug` is waiting for input it presents a prompt of the form `(byebug)`. If the program has terminated normally the prompt will be `(byebug:ctrl)` and in post-mortem debugging it will be `(byebug:post-mortem)`.

Whenever `byebug` gives an error message such as for an invalid command or an invalid location position, it will generally preface the message with `***`.

## Command Help

Once inside `byebug` you can always ask it for information on its commands using the `help` command. You can use `help` (abbreviated `h`) with no arguments to display a short list of named classes of commands

```
(byebug) help

  break      -- Sets breakpoints in the source code
  catch      -- Handles exception catchpoints
  condition  -- Sets conditions on breakpoints
  continue   -- Runs until program ends, hits a breakpoint or reaches a line
  delete     -- Deletes breakpoints
  disable    -- Disables breakpoints or displays
  display    -- Evaluates expressions every time the debugger stops
  down       -- Moves to a lower frame in the stack trace
  edit       -- Edits source files
  enable     -- Enables breakpoints or displays
```

```
  finish     -- Runs the program until frame returns
  frame      -- Moves to a frame in the call stack
  help       -- Helps you using byebug
  history    -- Shows byebug's history of commands
  info       -- Shows several informations about the program being debugged
  interrupt  -- Interrupts the program
  irb        -- Starts an IRB session
  kill       -- Sends a signal to the current process
  list       -- Lists lines of source code
  method     -- Shows methods of an object, class or module
  next       -- Runs one or more lines of code
  pry        -- Starts a Pry session
  quit       -- Exits byebug
  restart    -- Restarts the debugged program
  save       -- Saves current byebug session to a file
  set        -- Modifies byebug settings
  show       -- Shows byebug settings
  source     -- Restores a previously saved byebug session
  step       -- Steps into blocks or methods one or more times
  thread     -- Commands to manipulate threads
  tracevar   -- Enables tracing of a global variable
  undisplay  -- Stops displaying all or some expressions when program stops
  untracevar -- Stops tracing a global variable
  up         -- Moves to a higher frame in the stack trace
  var        -- Shows variables and its values
  where      -- Displays the backtrace
```

With a command name, `help` displays information on how to use the command.

```
(byebug) help list

  l[ist][[-=]][ nn-mm]

  Lists lines of source code

  Lists lines forward from current line or from the place where code was
  last listed. If "list-" is specified, lists backwards instead. If
  "list=" is specified, lists from current line regardless of where code
  was last listed. A line range can also be specified to list specific
  sections of code.
(byebug)
```

A number of commands, namely `info`, `set`, `show`, `enable` and `disable`, have many sub-parameters or *subcommands*. When you ask for help for one of these commands, you will get help for all of the subcommands that command offers. Sometimes you may want help only on a subcommand and to do this just follow the command with its subcommand name. For example, `help info breakpoints` will just give help about the `info breakpoints` command. Furthermore it will give longer help than the summary information that appears when you ask for help. You don't need to list the full subcommand name, just enough of the letters to make that subcommand distinct from others will do. For example, `help info b` is the same as `help info breakpoints`.

Some examples follow.

```
(byebug) help info
info[ subcommand]

Generic command for showing things about the program being debugged.

--
List of "info" subcommands:
--
info args        -- Argument variables of current stack frame
info breakpoints -- Status of user-settable breakpoints
info catch       -- Exceptions that can be caught in the current stack frame
info display     -- Expressions to display when program stops
info file        -- Info about a particular file read in
info files       -- File names and timestamps of files read in
```

```
info line        -- Line number and filename of current position in source file
info program     -- Execution status of the program
```

```
(byebug) help info breakpoints
Status of user-settable breakpoints.
Without argument, list info about all breakpoints.
With an integer argument, list info on that breakpoint.
```

```
(byebug) help info b
Status of user-settable breakpoints.
Without argument, list info about all breakpoints.
With an integer argument, list info on that breakpoint.
```

## Control Commands: quit, restart, source

### Quit

To exit `byebug`, type `quit` (abbreviated to `q`). Normally, if you are in an interactive session, this command will prompt you to confirm you really want to quit. If you want to quit without being prompted, enter `quit unconditionally` (abbreviated to `q!`).

### Restart

To restart the program, use the `restart|r` command. This is a re-exec - all `byebug` state is lost. If command arguments are passed, those are used. Otherwise program arguments from the last invocation are used.

You won't be able to restart your program in all cases. First, the program should have been invoked at the outset rather than having been called from inside your program or invoked as a result of post-mortem handling.

### Source

You can run `byebug` commands inside a file, using the command `source <file>`. The lines in a command file are executed sequentially. They are not printed as they are executed. If there is an error, execution proceeds to the next command in the file. For information about command files that get run automatically on startup see Command Files.

## Display Commands: display, undisplay

### Display

If you find that you want to print the value of an expression frequently (to see how it changes), you might want to add it to the *automatic display list** so that `byebug` evaluates it each time your program stops or after a line is printed if line tracing is enabled. Each expression added to the list is given a number to identify it; to remove an expression from the list, you specify that number. The automatic display looks like this:

```
(byebug) display n
1: n = 3
```

This display shows item numbers, expressions and their current values. If the expression is undefined or illegal the expression will be printed but no value will appear.

```
(byebug) display undefined_variable
2: undefined_variable =
(byebug) display 1/0
3: 1/0 =
```

If you use `display` with no argument, `byebug` will display the current values of the expressions in the list, just as it is done when your program stops. Using `info display` has the same effect.

#### Undisplay

To remove an item from the list, use `undisplay` followed by the number identifying the expression you want to remove. `undisplay` does not repeat if you press `<RET>` after using it (otherwise you would just get the error *No display number n*)

You can also temporarily disable or enable display expressions, so that the will not be printed but they won't be forgotten either, so you can toggle them again later. To do that, use `disable display` or `enable display` followed by the expression number.

### Evaluation of expressions: irb, pry

To examine and change data in your script you can just evaluate any Ruby code from `byebug` 's prompt. Any input that is not recognized as a command will be evaluated, so `byebug` essentially works as a REPL. If you want to evaluate something that conflicts with a `byebug` command, just use Ruby's `eval` . For example, if you want to print a variable called `n` , type `eval n` because typing just `n` will execute `byebug` 's command `next` .

Finally, if you need more advanced functionality from REPL's, you can enter `irb` or `pry` using `irb` or `pry` commands. The binding's environment will be set to the current state in the program. When you leave the repl and go back to `byebug` 's command prompt we show the file, line and text position of the program. If you issue a `list` without location information, the default location used is the current line rather than the current position that may have got updated via a prior `list` command.

```
$ byebug triangle.rb
[1, 10] in /path/to/triangle.rb
    1: # Compute the n'th triangle number, the hard way: triangle(n) == (n*(n+1))/2
=>  2: def triangle(n)
    3:   tri = 0
    4:   0.upto(n) do |i|
    5:     tri += i
    6:   end
    7:   tri
    8: end
    9:
   10: if __FILE__ == $0
(byebug) irb
irb(main):001:0> (0..6).inject { |sum, i| sum += i }
 => 21
irb(main):002:0> exit
(byebug)
```

### Printing variables: var

Byebug can print many different information about variables. Such as

- `var const <object>` . Show the constants of `<object>` . This is basically listing variables and their values in `<object>.constant` .
- `var instance <object>` . Show the instance variables of `<object>` . This is basically listing `<object>.instance_variables` .
- `var instance` . Show instance_variables of `self` .
- `var local` . Show local variables.
- `var global` . Show global variables.
- `var all` . Show local, global and instance and class variables of `self` .
- `method instance <object>` . Show methods of `<object>` . Basically this is the same as running `<object>.instance_methods(false)` .
- `method <class-or-module>` . Show methods of the class or module `<class-or-module>` . Basically this is the same as running `<class-or-module>.methods` .

### Examining Program Source Files: list

`byebug` can print parts of your script's source. When your script stops, `byebug` spontaneously lists the source code around the line where it stopped that line. It does that when you change the current stack frame as well. Implicitly there is a default

line location. Each time a list command is run that implicit location is updated, so that running several list commands in succession shows a contiguous block of program text.

If you don't need code context displayed every time, you can issue the `set noautolist` command. Now whenever you want code listed, you can explicitly issue the `list` command or its abbreviation `l`. Notice that when a second listing is displayed, we continue listing from the place we last left off. When the beginning or end of the file is reached, the line range to be shown is adjusted so "it doesn't overflow". You can set the `noautolist` option by default by dropping `set noautolist` in byebug's startup file `.byebugrc`.

If you want to set how many lines to be printed by default rather than use the initial number of lines, 10, use the `set listsize` command ([listsize()). To see the entire program in one shot, give an explicit starting and ending line number. You can print other portions of source files by giving explicit position as a parameter to the list command.

There are several ways to specify what part of the file you want to print. `list nnn` prints lines centered around line number `nnn` in the current source file. `l` prints more lines, following the last lines printed. `list -` prints lines just before the lines last printed. `list nnn-mmm` prints lines between `nnn` and `mmm` inclusive. `list =` prints lines centered around where the script is stopped. Repeating a `list` command with `RET` discards the argument, so it is equivalent to typing just `list`. This is more useful than listing the same lines again. An exception is made for an argument of `-`: that argument is preserved in repetition so that each repetition moves up in the source file.

## Editing Source files: edit

To edit a source file, use the `edit` command. The editor of your choice is invoked with the current line set to the active line in the program. Alternatively, you can give a line specification to specify what part of the file you want to edit.

You can customize `byebug` to use any editor you want by using the `EDITOR` environment variable. The only restriction is that your editor (say `ex`) recognizes the following command-line syntax:

```
ex +nnn file
```

The optional numeric value `+nnn` specifies the line number in the file where you want to start editing. For example, to configure `byebug` to use the `vi` editor, you could use these commands with the `sh` shell:

```
EDITOR=/usr/bin/vi
export EDITOR
byebug ...
```

or in the `csh` shell,

```
setenv EDITOR /usr/bin/vi
byebug ...
```

## The stack trace

When your script has stopped, one thing you'll probably want to know is where it stopped and some idea of how it got there.

Each time your script calls a method or enters a block, information about this action is saved. This information is what we call a *stack frame* or just a *frame*. The set of all frames at a certain point in the program's execution is called the *stack trace* or just the *stack*. Each frame contains a line number and the source-file name that the line refers to. If the frame is the beginning of a method it also contains the method name.

When your script is started, the stack has only one frame, that of the `main` method. This is called the *initial frame* or the *outermost frame*. Each time a method is called, a new frame is added to the stack trace. Each time a method returns, the frame for that method invocation is removed. If a method is recursive, there can be many frames for the same method. The frame for the method in which execution is actually occurring is called the *innermost frame*. This is the most recently created of all the stack frames that still exist.

Every time the debugger stops, one entry in the stack is selected as the current frame. Many byebug commands refer implicitly to the selected block. In particular, whenever you ask Byebug to list lines without giving a line number or location the value is found in the selected frame. There are special commands to select whichever frame you're interested in, such as `up`, `down` and `frame`.

After switching frames, when you issue a `list` command without any position information, the position used is the location in the frame that you just switched between, rather than a location that got updated via a prior `list` command.

Byebug assigns numbers to all existing stack frames, starting with zero for the *innermost frame*, one for the frame that called it, and so on upward. These numbers do not really exist in your script, they are assigned by Byebug to give you a way of designating stack frames in commands.

### Printing the Stack: `where` command

The command `where`, aliased to `bt` or `backtrace` prints the call stack., It shows one line per frame, for many frames, starting with the place that you are stopped at (frame zero), followed by its caller (frame one), and on up the stack. Each frame is numbered and can be referred to in the `frame` command. The position of the current frame is marked with `-->`.

The are some special frames generated for methods that are implemented in C. One such method is `each`. They are marked differently in the call stack to indicate that we cannot switch to those frames. This is because they have no source code in Ruby, so we can not debug them using Byebug.

```
(byebug) where
--> #0 Object.gcd(a#Fixnum, b#Fixnum) at line gcd.rb:6
    #1 at line gcd.rb:19
```

### Selecting a frame: `up`, `down` and `frame` commands

- `up <n>`: Move `n` frames up the stack, towards the outermost frame (higher frame numbers, frames that have existed longer). `n` defaults to one.
- `down <n>`: Move `n` frames down the stack, towards the *innermost frame* (lower frame numbers, frames that were created more recently). `n` defaults to one.
- `frame <n>`: Allows you to move to an arbitrary frame. `n` is the stack frame number or 0 if no frame number is given. `frame 0` will show the current and most recent stack frame. If a negative number is given, counting is from the other end of the stack frame, so `frame -1` shows the least-recent, outermost stack frame. Without an argument, `frame` prints the current stack frame.