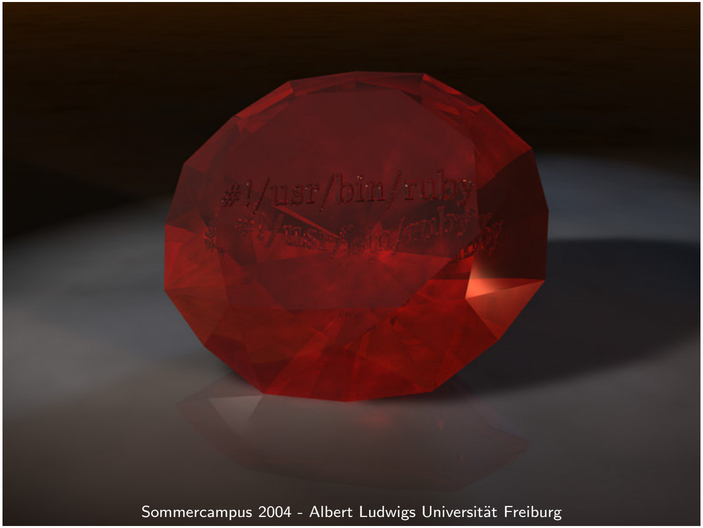
Ruby Course
— an immersive programming course —



Sommercampus 2004 - Albert Ludwigs Universität Freiburg

Brian Schröder
mail@brian-schroeder.de

Part I

Introduction

Simple hello world example

This is a must

```ruby
#!/usr/bin/ruby

puts 'Hello World'
```

```
Hello World
```

Function definition

Functions are defined using the def keyword

```ruby
#!/usr/bin/ruby

def hello(programmer)
  puts "Hello #{programmer}"
end

hello('Brian')
```

```
Hello Brian
```

In ruby everything is an object

Everything is an object, so get used to the ".method" notation.

```
(5.6).round                                      » 6
(5.6).class                                      » Float
(5.6).round.class                                » Fixnum

'a string'.length                                » 8
'a string'.class                                 » String
'tim tells'.gsub('t', 'j')                       » "jim jells"

'abc'.gsub('b', 'xxx').length                    » 5

['some', 'things', 'in', 'an', 'array'].length   » 5
['some', 'things', 'in', 'an', 'array'].reverse  » ["array", "an", "in", "things", "some"]

Float.class                                      » Class
Class.class                                      » Class
Object.class                                     » Class
```

Class definition and inheritance

Base Class

```ruby
class Person
  def initialize(name)
    @name = name
  end

  def greet
    "Hello, my name is #{@name}."
  end
end

brian = Person.new('Brian')
puts brian.greet
```

```
Hello, my name is Brian.
```

Sub Class

```ruby
class Matz < Person
  def initialize
    super('Yukihiro Matsumoto')
  end
end

puts Matz.new.greet
```

```
Hello, my name is Yukihiro Matsumoto.
```

All normal control structures are available

Ruby follows the principle of least surprise POLS

But if you already know some programming languages, there are sure some surprises here:

```ruby
def greet(*names)
  case names.length
  when 0
    "How sad, nobody wants to hear my talk."
  when 1
    "Hello #{name}. At least one wants to hear about ruby."
  when 2..5
    "Hello #{names.join(', ')}. Good that all of you are interested."
  when 6..10
    "#{names.length} students. Thats perfect. Welcome to ruby!"
  else
    "Wow #{names.length} students. We'll have to find a bigger room."
  end
end

puts greet('Ashraf', 'Ingo', 'Jens', 'Johannes', 'Marius', 'Robert',
           'Stefan', 'Thorsten', 'Tobias', 'Jet Loong')
```

```
10 students. Thats perfect. Welcome to ruby!
```

Basic syntax rules

Ruby syntax tries to omit “noise”

```
# Functions are defined by the def keyword (define function)
# Function arguments can have default values.
def multi_foo(count = 3)
  'foo ' * count
end                                                    » nil

# Brackets can be omitted, if the situation is not ambiguous
multi_foo(3)                                           » "foo foo foo "
puts 'hello world'                                     » nil

# Strings are written as
'Simple #{multi_foo(2)}'                               » "Simple #{multi_foo(2)}"
"Interpolated #{multi_foo}"                            » "Interpolated foo foo foo "

# Numbers
10                                                     » 10
0.5                                                    » 0.5
2e-4                                                   » 0.0002
0xFFFF                                                 » 65535
010                                                    » 8
```

Syntax: Variables, constants, methods, . . .

Variables / methods:	student, i, epsilon, last_time Variables and methods look alike. This is reasonable because a variable can be substituted by a method.
Constants:	OldPerson, PDF_KEY, R2D2 Constants can only be defined once.
Instance Variables:	@name, @last_time, @maximum Instance variables can only be accessed by the owning object.
Class Variables:	@@lookup_table, @@instance Class variables belong not to the instances but to the class They exist only once for the class, and are shared by all instances.
Global Variables:	$global, $1, $count Usage of global variables has been declared a capital crime by the school of good design.
Symbols:	:name, :age, :Class Symbols are unique identifiers, that we will encounter in various places.

Stylistic Rules

- Variables and methods should be written in snake_case
- Class Names should be written in CamelCase
- Constants should be written ALL_UPPERCASE

Tools

Editors: Theses Editors are available under windows and linux

xemacs	Good highlighting and auto-indentation. Can be expanded to do everything.
vim	Good highlighting and auto-indentation.
freeride	Complete ruby ide written in ruby.
...	and a lot more. For every 10 programmers you have 15 preferred editors.

Interpreter: Each ruby script you write should be prefixed by #!/usr/bin/ruby -w, to tell the system where the ruby interpreter is located. (The path may depend on the system.)

Ruby Shell: The interactive ruby shell `irb` can be used to try out parts of the code.

Ruby Documentation: Information about every class in ruby can be found using `ri`, the ruby interactive documentation system.

ri is ruby's fast helper

```
$ ri String#tr
————————————————————————————————————— String#tr
     str.tr(from_str, to_str) => new_str
————————————————————————————————————————————
     Returns a copy of str with the characters in from_str replaced by
     the corresponding characters in to_str. If to_str is shorter than
     from_str, it is padded with its last character. Both strings may
     use the c1–c2 notation to denote ranges of characters, and
     from_str may start with a ^, which denotes all characters except
     those listed.

        "hello".tr('aeiou', '*')           #=> "h*ll*"
        "hello".tr('^aeiou', '*')          #=> "*e**o"
        "hello".tr('el', 'ip')             #=> "hippo"
        "hello".tr('a-y', 'b-z')           #=> "ifmmp"
```

irb can be used to try out ideas

```
$ irb --simple-prompt
>> 'hal'.tr('za-y', 'ab-z')
=> "ibm"
>> class String
>>   def rot13
>>     self.tr('a-z', 'n-za-m')
>>   end
>> end
=> nil
>> a = 'geheimer text'
=> "geheimer text"
>> b = a.rot13
=> "trurvzre grkg"
>> b.rot13
=> "geheimer text"
```

Exercises: Tools

irb and numbers:

Open up `irb` and set the variables $a = 1$, $b = 2$.

- Calculate a/b. Calculate $1.0/2.0$. Calculate 10^{200}.
- Write `require 'complex'` into irb to load the "Complex" library
  Create a constant `I` set to `Complex.new(0, 1)` and calculate $(1 + 2i) \cdot (2 + 1i)$

First program, string interpolation:

Write a file `answer.rb` containing a function `answer(a,b)` that calculates $a \cdot b$ and returns the string "the answer is '#{result of $a \cdot b$}'.".
Create a file `answer` containing the following lines:

```
#!/usr/bin/ruby -w
require 'answer'
puts answer(6, 7)
```

Make the file executable and call it.

ri:

Use `ri` to find out how to make a string all uppercase, and try the function in `irb`.

Array

```
# Literal Array
['An', 'array', 'with', 5, 'entries'].join(' ')    » "An array with 5 entries"

# New Array
a = Array.new                                      » []
a << 'some' << 'things' << 'appended'              » ["some", "things", "appended"]
a[2]                                               » "appended"
a[0] = 3                                           » 3
a                                                  » [3, "things", "appended"]

# Default Values can be used ...
Array.new(10, 0)                                   » [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

# ... but beware of the reference
a = Array.new(2, 'Silke')                          » ["Silke", "Silke"]
a[0] << ' Amberg'                                  » "Silke Amberg"
a                                                  » ["Silke Amberg", "Silke Amberg"]
```

Basic Containers

Arrays can be used as queues, stacks, deques or simply as arrays.

```ruby
print 'Array as stack: '
stack = Array.new()
stack.push('a')
stack.push('b')
stack.push('c')
print stack.pop until stack.empty?

print "\n"
print 'Array as queue: '
queue = Array.new()
queue.push('a').push('b').push('c')
print queue.shift until queue.empty?
```

```
Array as stack: cba
Array as queue: abc
```

Basic Containers

Hashes are fast associative containers

```
# Literal Hash
h0 = { 'one' => 1, 'two' => 2, 'three' => 3} » {"three"=>3, "two"=>2, "one"=>1}
h0['one']                                    » 1

# Populating a hash
h1 = Hash.new                                » {}
h1['gemstone'] = 'ruby'                      » "ruby"
h1['fruit'] = 'banana'                       » "banana"
h1                                           » {"gemstone"=>"ruby", "fruit"=>"banana"}

# Often symbols are used as keys
h2 = {:june => 'perl', :july => 'ruby'}      » {:july=>"ruby", :june=>"perl"}
h2[:july]                                    » "ruby"

# But arbitrary keys are possible
a = ['Array', 1]                             » ["Array", 1]
b = ['Array', 2]                             » ["Array", 2]
h3 = { a => :a1, b => :a2 }                  » {["Array", 1]=>:a1, ["Array", 2]=>:a2}
h3[a]                                        » :a1
```

Blocks and iterators

A function can take a block as an argument.
A block is a piece of code, similar to an anonymous function, but it inherits the containing scope.

Using iterators

```
# A simple iterator, calling the block once for each entry in the array
['i', 'am', 'a', 'banana'].each do | entry | print entry, ' ' end
```

```
i am a banana
```

```
# Another commonly used iterator. The block is called in the scope where it was
# created.
fac = 1                                              » 1
1.upto(5) do | i | fac *= i end                      » 1
fac                                                  » 120

# The result of the block can be used by the caller
[1,2,3,4,5].map { | entry | entry * entry }          » [1, 4, 9, 16, 25]

# and more than one argument is allowed
(0..100).inject(0) { | result, entry | result + entry }   » 5050
```

Blocks and iterators

Block Syntax

Blocks can be enclosed by do | | … end.

```
[1,2,3,4,5].each do | e | puts e end
```

or by braces { | | … }

```
[1,2,3,4,5].map { | e | e * e }                » [1, 4, 9, 16, 25]
```

A convention is to

- use do | | … end wherever the side-effect is important
- and braces where the return value is important.

Writing iterators

```ruby
def f(count, &block)
  value = 1
  1.upto(count) do | i |
    value = value * i
    block.call(i, value)
  end
end

f(5) do | i, f_i | puts "f(#{i}) = #{f_i}" end
```

Blocks and iterators

Writing iterators

```ruby
def f(count, &block)
  value = 1
  1.upto(count) do | i |
    value = value * i
    block.call(i, value)
  end
end

f(5) do | i, f_i | puts "f(#{i}) = #{f_i}" end
```

```
f(1) = 1
f(2) = 2
f(3) = 6
f(4) = 24
f(5) = 120
```

Blocks and iterators

Saving the block

```ruby
class Repeater
  def initialize(&block)
    @block = block
    @count = 0
  end

  def repeat
    @count += 1
    @block.call(@count)
  end
end

repeater = Repeater.new do | count | puts "You called me #{count} times" end
3.times do repeater.repeat end
```

Saving the block

```ruby
class Repeater
  def initialize(&block)
    @block = block
    @count = 0
  end

  def repeat
    @count += 1
    @block.call(@count)
  end
end

repeater = Repeater.new do | count | puts "You called me #{count} times" end
3.times do repeater.repeat end
```

```
You called me 1 times
You called me 2 times
You called me 3 times
```

Exercises: Iterators

Refer to the exercise files for exact specification of the problems.

n_times

Write an iterator function `n_times(n)` that calls the given block n times.

Write an iterator class `Repeat` that is instantiated with a number and has a method `each` that takes a block and calls it as often as declared when creating the object.

Faculty

Write a one-liner in irb using `Range#inject` to calculate 20!. Generalize this into a function.

Maximum

Write a function to find the longest string in an array of strings.

find_it

Write a function `find_it` that takes an array of strings and a block. The block should take two parameters and return a boolean value.
The function should allow to implement `longest_string`, `shortest_string`, and other functions by changing the block.

Ruby assignments.

```
# Every assignment returns the assigned value
a = 4                                          » 4

# So assignments can be chained
a = b = 4                                      » 4
a + b                                          » 8

# and used in a test
file = File.open('../slides.tex')              » #<File:../slides.tex>
linecount = 0                                  » 0
linecount += 1 while (line = file.gets)        » nil

# Shortcuts
a += 2                                         » 6
a = a + 2                                      » 8
#...

# Parallel assignment
a, b = b, a                                    » [4, 8]

# Array splitting
array = [1, 2]                                 » [1, 2]
a, b = *array                                  » [1, 2]
```

Control Structures - Conditionals

Ruby has all standard control structures.
And you may even write them to the right of an expression.

```
if (1 + 1 == 2)
  "Like in school."
else
  "What a surprise!"
end                                               » "Like in school."

"Like in school." if (1 + 1 == 2)                 » "Like in school."
"Surprising!" unless (1 + 1 == 2)                 » nil

(1 + 1 == 2) ? 'Working' : 'Defect'               » "Working"

spam_probability = rand(100)                      » 64
case spam_probability
when 0...10 then "Lowest probability"
when 10...50 then "Low probability"
when 50...90 then "High probability"
when 90...100 then "Highest probability"
end                                               » "High probability"
```

Control Structures - True and False

Only *nil* and *false* are false, everything else is true.

```
def is_true(value)
  value ? true : false
end                                       » nil

is_true(false)                            » false
is_true(nil)                              » false
is_true(true)                             » true
is_true(1)                                » true
is_true(0)                                » true
is_true([0,1,2])                          » true
is_true('a'..'z')                         » true
is_true('')                               » true
is_true(:a_symbol)                        » true
```

Join the equal rights for zero movement!

Control Structures - Loops

Ruby has a variety of loop constructs, but don't forget the blocks!

```
i = 1                            » 1

while (i < 10)
  i *= 2
end                              » nil
i                                » 16

i *= 2 while (i < 100)           » nil
i                                » 128

begin
  i *= 2
end while (i < 100)              » nil
i                                » 256

i *= 2 until (i >= 1000)         » nil
i                                » 1024

```

```
loop do
  break i if (i >= 4000)
  i *= 2
end                              » 4096
i                                » 4096

4.times do i *= 2 end            » 4
i                                » 65536

r = []                           » []
for i in 0..7
  next if i % 2 == 0
  r << i
end                              » 0..7
r                                » [1, 3, 5, 7]

# Many things are easier with blocks:
(0..7).select { |i| i % 2 != 0 } » [1, 3, 5, 7]
```

Exercises: Control Structures

Fibonacci

Write functions that calculate the fibonacci numbers using different looping constructs

$$fib(i) = \begin{cases} 0 & i = 0 \\ 1 & i = 1 \\ fib(i-1) + fib(i-2) & otherwise \end{cases}$$

recursion:	Implement the function using recursion.
while:	Implement the function using a while loop.
for:	Implement the function using a for loop.
times:	Implement the function using the times construct.
loop:	Implement the function using the loop construct.

Iterator

Write a fibonacci iterator function.
That is a function that takes a number n and a block and calls the block with $fib(0), fib(1), \ldots fib(n)$

Generator

Write a fibonacci generator class.
That is: A class that has a next function which on each call returns the next fibonacci number.

Part II

A Real application

A Simple Chat Client

```ruby
#!/usr/bin/ruby -w
require 'socket'
require 'thread'

host = ARGV[0] || 'localhost'
port = ARGV[1] || 1111

socket = TCPSocket.new(host, port)

t = Thread.new do # Receiver thread
  while line = socket.gets
    puts "Received: #{line}"
  end
  socket.close
end

while line = $stdin.gets # Read input
  break if /^exit/ =~ line
  socket.puts line
end
socket.puts 'QUIT' # Request disconnect

t.join # Wait for receiver thread to finish
```

Problems:

- Code not extendable (what about adding a gui)
- No object orientation
- No exception handling
- No documentation

Missing features:

- Username choosing
- Listing of participating users
- ...

A Simple Chat Server

Problems:

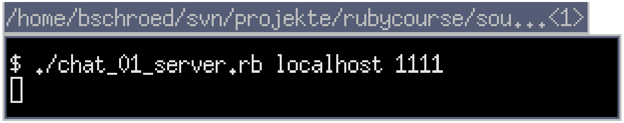
- Code not extendable (Everything in one function)
- No object orientation
- No exception handling
- No documentation

Missing features:

- Usernames
- Multiple channels
- ...

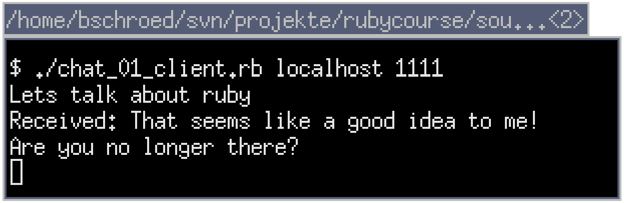
```ruby
#!/usr/bin/ruby -w
require 'socket' # TCP communication
require 'thread' # Multi Threading.

host, port = ARGV[0], ARGV[1]
semaphore = Mutex.new
server = TCPServer.new(host, port)
clients = []

while (socket = server.accept)
  semaphore.synchronize do clients << socket end
  swt = Thread.new(socket) do | the_socket |
    while line = the_socket.gets
      break if /^QUIT/ =~ line
      semaphore.synchronize do
        clients.each do | client |
          client.puts line if client != the_socket
        end
      end
    end
    semaphore.synchronize do clients.delete(socket) end
    socket.close
  end
end
```

A Simple Chat



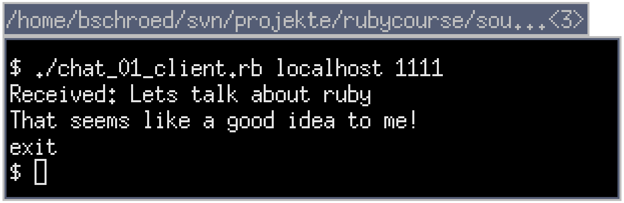
/home/bschroed/svn/projekte/rubycourse/sou...<1>

```
$ ./chat_01_server.rb localhost 1111
```



/home/bschroed/svn/projekte/rubycourse/sou...<2>

```
$ ./chat_01_client.rb localhost 1111
Lets talk about ruby
Received: That seems like a good idea to me!
Are you no longer there?
```



/home/bschroed/svn/projekte/rubycourse/sou...<3>

```
$ ./chat_01_client.rb localhost 1111
Received: Lets talk about ruby
That seems like a good idea to me!
exit
$
```

A Better Chat Client

```ruby
#!/usr/bin/ruby -w
require 'socket'
require 'thread'

class ChatClient
  def initialize(host, port)
    @socket = TCPSocket.new(host, port)
    @on_receive = nil
  end

  def on_receive(&on_receive)
    @on_receive = on_receive
  end

  def listen
    @listen_thread = Thread.new do
      while line = @socket.gets
        @on_receive.call(line) if @on_receive
      end
    end
  end

  def send(line)
    @socket.puts(line)
  end

  def close
    send('QUIT')
    @listen_thread.join
  end
end

host = ARGV[0] || 'localhost'
port = ARGV[1] || 1111
client = ChatClient.new(host, port)
client.on_receive do | line | puts "Received: #{line}" end
client.listen

# Input
while line = $stdin.gets
  break if /^exit/ =~ line
  client.send(line)
end

client.close
```

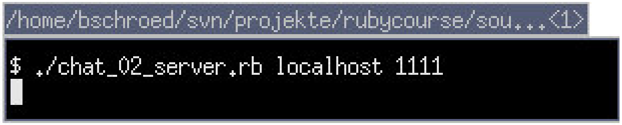
A Better Chat Server - Part I

```ruby
#!/usr/bin/ruby -w
require 'socket' # TCP communication
require 'thread' # Multi Threading

class ChatServer
  def initialize(host, port)
    @server = TCPServer.new(host, port)
    @semaphore = Mutex.new
    @clients = []
  end

  def serve
    while (socket = @server.accept)
      client = ClientThread.new(socket)
      client.on_received do | c, l |
        distribute(c, l)
      end
      add_client(client)
      client.listen
    end
  end

  def distribute(client, line)
    @semaphore.synchronize do
      @clients.each do | c |
        c.send(line) if c != client
      end
    end
  end

  def add_client(client)
    @semaphore.synchronize do
      @clients << client
    end
    client.on_terminate do | c |
      remove_client(c)
    end
  end

  def remove_client(client)
    @semaphore.synchronize do
      @clients.delete(client)
    end
  end
end
```

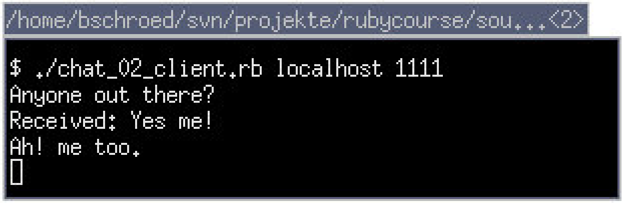
A Better Chat Server - Part II

```ruby
class ClientThread
  def initialize(socket)
    @socket = socket
    @on_received = @on_terminate = nil
  end

  def listen
    @listen_thread = Thread.new do
      while line = @socket.gets
        break if /^QUIT/ =~ line
        @on_received.call(self, line) if @on_received
      end
      @on_terminate.call(self) if @on_terminate
      @socket.close
    end
  end

  def send(line)
    @socket.puts(line)
  end

  def on_received(&on_received)
    @on_received = on_received
  end

  def on_terminate(&on_terminate)
    @on_terminate = on_terminate
  end
end

host, port = ARGV[0], ARGV[1]
cs = ChatServer.new(host, port)
cs.serve
```

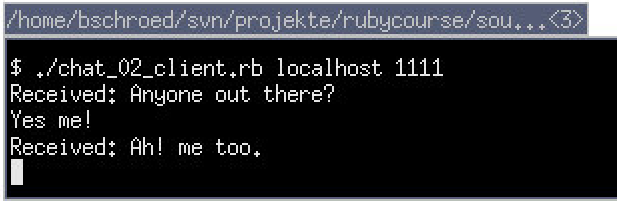
A Simple Chat



```
/home/bschroed/svn/projekte/rubycourse/sou...<1>
$ ./chat_02_server.rb localhost 1111
```



```
/home/bschroed/svn/projekte/rubycourse/sou...<2>
$ ./chat_02_client.rb localhost 1111
Anyone out there?
Received: Yes me!
Ah! me too.
```



```
/home/bschroed/svn/projekte/rubycourse/sou...<3>
$ ./chat_02_client.rb localhost 1111
Received: Anyone out there?
Yes me!
Received: Ah! me too.
```

Exception Handling

Catching Exceptions

```ruby
begin
  # Code
rescue ExceptionClass1 => exception1
  # Will be executed if code raises ExceptionClass1
rescue ExceptionClass2 => exception2
  # Will be executed if code raises ExceptionClass2
rescue
  # Rescues any exception
ensure
  # Code that will always be executed
end
```

Raising Exceptions

```ruby
# Re-raise current exception
raise
# Raise RuntimeError exception with message "Server down"
raise "Server down"
# Raise EServerDown exception with message "Server not responding"
raise EServerDown, "Server not responding"
```

Exception Handling

Exception trees

```ruby
class EChatException < Exception; end
class EInvalidServerException < EChatException; end
class EServerDiedException < EChatException; end
```

```ruby
def chat
  begin
    # ...
    raise EServerDiedException
    # ...
  rescue EInvalidServerException
    puts "Invalid server"
    raise
  rescue EServerDiedException
    puts "Server died"
    raise
  end
end
```

```ruby
begin
  #...
  chat
  #...
rescue EChatException => e
  puts "#{e} occurred."
  puts "Please contact your " +
    "system administrator ;)"
end
```

Exception Handling

Unhandled exception: No server is running.

```
client = ChatClient.new(host, port)
client.on_receive do | line | puts "Received: #{line}" end
client.listen
```

```
chat_02_client.rb:7:in ‘initialize’: Connection refused – connect(2) (Errno::ECONNREFUSED)
          from chat_02_client.rb:7:in ‘new’
          from chat_02_client.rb:7:in ‘initialize’
          from chat_02_client.rb:35:in ‘new’
          from chat_02_client.rb:35
```

Exception Handling

Example: Handled exception: No server is running.

```
begin
  client = ChatClient.new(host, port)
  client.on_line_received do | line | puts "Received: #{line}" end
  client.listen
rescue Errno::ECONNREFUSED => exception
  puts "No chat server running on #{host}:#{port}."
  puts "The error was: #{exception}."
  exit
rescue => exception
  puts "Could not connect to#{host}:#{port}."
  puts "The error was: #{exception}."
  exit
end
```

```
No chat server running on localhost:1111.
The error was: Connection refused – connect(2).
```

Example: Ensuring closing of a resource

```
file = File.open('/usr/share/dict/words')             » #<File:/usr/share/dict/words>
begin
  xys = file.select { | word | word[0..1] == 'xy' }
ensure
  file.close
end                                                    » ["xylem\n", "xylem's\n", "xylophone\n", "xy

xys = File.open('/usr/share/dict/words') { | file |
  file.select { | word | word[0..1] == 'xy' }
}                                                      » ["xylem\n", "xylem's\n", "xylophone\n", "xy
```

Exception Handling

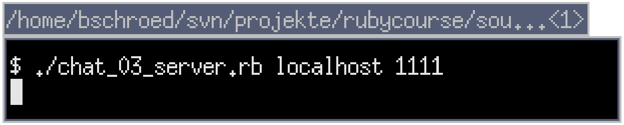
Example: Usage in the chat server (Old code)

```
def listen
  @listen_thread = Thread.new do
    while line = @socket.gets
      break if /^QUIT/ =~ line
      @on_received.call(self, line) if @on_received
    end
    @on_terminate.call(self) if @on_terminate
    @socket.close
  end
end
```

Example: Usage in the chat server (New code)

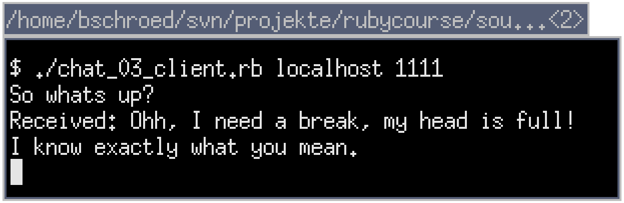
```ruby
def listen
  @listen_thread = Thread.new do
    begin
      while line = @socket.gets
        break if /^QUIT/ =~ line
        @on_received.call(self, line) if @on_received
      end
    ensure
      @on_terminate.call(self) if @on_terminate
      @socket.close
    end
  end
end
```

A Simple Chat



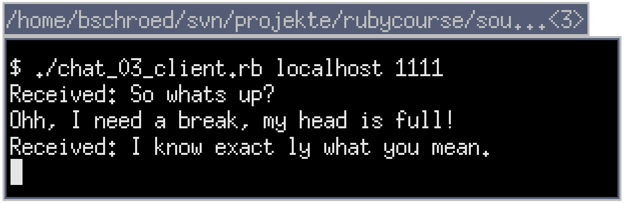
/home/bschroed/svn/projekte/rubycourse/sou...<1>

```
$ ./chat_03_server.rb localhost 1111
```



/home/bschroed/svn/projekte/rubycourse/sou...<2>

```
$ ./chat_03_client.rb localhost 1111
So whats up?
Received: Ohh, I need a break, my head is full!
I know exactly what you mean.
```



/home/bschroed/svn/projekte/rubycourse/sou...<3>

```
$ ./chat_03_client.rb localhost 1111
Received: So whats up?
Ohh, I need a break, my head is full!
Received: I know exact ly what you mean.
```

Exercises: Exception Handling

Handshake

Change the programs `chat_03_client.rb` and `chat_03_server.rb` to follow this protocol:

1. Client connects
2. Server sends "YASC: 0.1 Server"
3. Client sends "YASC: 0.1 Client"

Exception Raising

- Raise an ENoYASCServer exception in the client, if the server is not sending the correct greeting string.
- Raise an ENoYASCClient exception in the server, if the client is not sending the correct greeting string.

Exception Handling

- Terminate the client with a useful error message if a ENoYASCServer exception occurs.
- Close the client's socket and terminate client's-thread in the server if a ENoYASCClient exception occurs.

Part III

The dynamicity of ruby

Accessor Functions: Getting object properties

```ruby
class Cell
  def initialize
    @state = :empty
  end
end
```

```ruby
class Board
  def initialize(width, height)
    @width = width; @height = height
    @cells = Array.new(height) { Array.new(width) { Cell.new } }
  end
end
```

Access a property

```ruby
class Cell
  def state
    @state
  end
end
```

```ruby
cell = Cell.new    » #<Cell:... @state=:e...>
cell.state         » :empty
```

Calculated property

```ruby
class Board
  def size
    self.width * self.height
  end
end
```

Shortcut

```ruby
class Cell
  attr_reader :state
end
```

Accessor Functions: Setting object properties

```ruby
class Cell
  def initialize
    @state = :empty
  end
end
```

```ruby
class Board
  def initialize(width, height)
    @width = width; @height = height
    @cells = Array.new(height) { Array.new(width) { Cell.new } }
  end
end
```

Set a property

```ruby
class Cell
  def state=(state)
    @state = state
  end
end
```

```
cell = Cell.new             » #<Cell:... @state=:e...>
cell.state                  » :empty
cell.state = :king          » :king
cell.state                  » :king
```

Shortcut

```ruby
class Cell
  attr_writer :state
end
```

Shortcut for getter and setter

```ruby
class Cell
  attr_accessor :state
end
```

Accessor Functions - Array-like accessors

```ruby
class Cell
  def initialize
    @state = :empty
  end
end
```

```ruby
class Board
  def initialize(width, height)
    @width = width; @height = height
    @cells = Array.new(height) { Array.new(width) { Cell.new } }
  end
end
```

The method "[]" can be used to implement an array-like accessor.

```ruby
class Board
  def [](col, row)
    @cells[col][row]
  end
end
```

```ruby
board = Board.new(8, 8)   » #<Board:... @cells=[[...>
board[0, 0]               » #<Cell:... @state=:e...>
board[0, 0] = Cell.new()  » #<Cell:... @state=:e...>
```

The method "[]=" can be used as an array-like setter.

```ruby
class Board
  def []=(col, row, cell)
    @cells[col][row] = cell
  end
end
```

```ruby
board = Board.new(8, 8)        » #<Board:... @cells=[[...>
board[0, 0]                    » #<Cell:... @state=:e...>
board[0, 0] = Cell.new()       » #<Cell:... @state=:e...>
board[0, 0].state = :tower     » :tower
board[0, 0].state              » :tower
```

Exercise: Accessor Functions

PersonName

Create a class PersonName, that has the following attributes

Name	The name of the person.
Surname	The given name of the person.
Fullname	"#{surname} #{name}". Add also a fullname setter function, that splits (String::split) the fullname into surname and name.

Person

Create a class Person, that has the following attributes

Age	The person's age (in years).
Birthdate	The person's birthdate.
Name	A PersonName object.

- The person's constructor should allow to pass in name, surname and age. All optionally.
- The person's age and birth date should always be consistent. That means if I set the person's birth date, his age should change. And if I set a person's age, his birth date should change.

Ruby is Dynamic

Classes, functions, modules can be modified at runtime.

```ruby
class PersonShort < BasePerson
  attr_accessor :name, :surname
end
```

attr_accessor is not a special language construct, but a function, that creates getter and setter functions for each argument.

Ruby is Dynamic

You can extend existing classes

```
class Integer
  def fac
    raise "Faculty undefined for #{self}" if self < 0
    return (1..self).inject(1) { |result, i| result * i }
  end
end

puts (0..13).map { |i| i.fac }.join(', ')
```

```
1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, 39916800, 479001600, 6227020800
```

Exercises: Extension of existing classes

Fibonacci II

Extend Integer with a function fib that calculates the corresponding fibonacci number.

Shuffle

Extend Array with a method shuffle that creates a random permutation of the elements in the array.

```
[0,1,2,3,4,5].shuffle          » [4, 3, 2, 5, 1, 0]
[0,1,2,3,4,5].shuffle          » [2, 5, 1, 3, 0, 4]
[0,1,2,3,4,5].shuffle          » [3, 4, 1, 5, 2, 0]
```

Set

Extend Array with the set methods union and intersect. E.g.:

```
a1 = [1, 2, 3]
a2 = [2, 3, 4]
a3 = [{:c => 'a', :v => 1}, {:c => 'b', :v => 2}]
a4 = [{:c => 'b', :v => 2}, {:c => 'c', :v => 3}]
a1.intersect(a2)               » [2, 3]
a2.intersect(a1)               » [2, 3]
a1.intersect(a3)               » []
a3.intersect(a4)               » [{:v=>2, :c=>"b"}]
a1.union(a2).union(a3)         » [1, 2, 3, 2, 3, 4, {:v=>1, :c=>"a"}, {:v=>2, :c=>"b"}]
a1.intersect(a1.union(a2))     » [1, 2, 3]
```

Modules provide namespaces

```ruby
module AntGame
  class Ant
    attr_accessor :x, :y, :direction, :next_action

    def initialize(x, y)
      @x = x; @y = y
      @direction = :north
      @next_action = Actions::WAIT
    end
  end

  module Actions
    WAIT = :wait
    TURN_LEFT = :turn_left
    TURN_RIGHT = :turn_right
    GO = :go
  end
end

AntGame::Ant.new(4, 5)
include AntGame
Ant.new(1, 2)
```

Modules provide controlled multiple inheritance

```ruby
module Observable
  def register(event=nil, &callback)
    @observers ||= Hash.new
    @observers[event] ||= []
    @observers[event] << callback
    self
  end

  protected
  def signal_event(event = nil, *args)
    @observers ||= Hash.new
    @observers[event] ||= []
    @observers[event].each do | callback |
      callback.call(self, *args)
    end
  end
end
```

```ruby
class Observed
  include Observable

  def foo=(a_foo)
    signal_event(:changed, @foo, a_foo)
    @foo = a_foo
  end
end

observed = Observed.new
observed.register(:changed) do | o, old, new |
  puts "#{old} -> #{new}"
end

observed.foo = 'Yukihiro'
observed.foo = 'Yukihiro Matsumoto'
observed.foo = 'Matz'
```

```
 -> Yukihiro
Yukihiro -> Yukihiro Matsumoto
Yukihiro Matsumoto -> Matz
```

Exercises: Modules I

Tree

Create a class TreeItem that has the following attributes:

item	That contains the list item used.
left	The left child of this item.
right	The right child of this item.
each	A function that takes a block and calls the block for each item in the subtree.

Include the module `Enumerable` into the tree item. E.g.

```
root = TreeItem.new("root")                           » #<TreeItem:0x40293cec @item="root">
root.to_a.join(' | ')                                 » "root"
root.left = TreeItem.new("left")                      » #<TreeItem:0x403079f8 @item="left">
root.to_a.join(' | ')                                 » "root | left"
root.right = TreeItem.new("reft")                     » #<TreeItem:0x402eb5dc @item="reft">
root.to_a.join(' | ')                                 » "root | left | reft"
root.left.left = TreeItem.new("left-left")            » #<TreeItem:0x402e5178 @item="left-left">
root.to_a.join(' | ')                                 » "root | left | left-left | reft"
root.left.right = TreeItem.new("left-right")          » #<TreeItem:0x402dd5f4 @item="left-right">
root.to_a.join(' | ')                                 » "root | left | left-left | left-right | reft"
root.inject(0) { | r, e | r + 1 }                     » 5
```

Exercises: Modules I

Example Implementation

```
class TreeItem
  attr_accessor :left, :right, :item
  include Enumerable

  def initialize(item)
    self.item = item
  end

  def each(&block)
    block.call(self.item)
    left.each(&block) if left
    right.each(&block) if right
  end
end                                      » nil

root = TreeItem.new("root")              » #<TreeItem:0x40293cec @item="root">
root.to_a.join(' | ')                    » "root"
root.left = TreeItem.new("left")         » #<TreeItem:0x403079f8 @item="left">
root.to_a.join(' | ')                    » "root | left"
root.right = TreeItem.new("reft")        » #<TreeItem:0x402eb5dc @item="reft">
root.to_a.join(' | ')                    » "root | left | reft"
root.left.left = TreeItem.new("left-left") » #<TreeItem:0x402e5178 @item="left-left">
root.to_a.join(' | ')                    » "root | left | left-left | reft"
```

Exercises: Modules II

List

Create a class `ListItem` that has the following attributes/methods:

item	That contains the list item used.
previous	The predecessor in the list. When this property is set the old and new predecessor's `next` property should be updated.
next	The successor in the list. When this property is set the old and new successor's `previous` should be updated.
each	Takes a block and calls the block for each item in the list. This should be done by following `previous` to the beginning of the list and then returning each item in list order.
insert	Inserts an item after this item into the list.

Include the module `Enumerable` into the list item, such that the following constructs work. E.g.

```
one = ListItem.new("one")                            » #<ListItem:... @item="on...>
one.next = ListItem.new("two")                       » #<ListItem:... @item="tw...>
one.next.next = ListItem.new("three")                » #<ListItem:... @item="th...>
one.previous = ListItem.new("zero")                  » #<ListItem:... @next=#<L...>
one.inject('List:') { | r, v | r + ' ' + v }         » "List: zero one two three"

one.insert ListItem.new("one point five")            » #<ListItem:... @next=#<L...>
one.inject('List:') { | r, v | r + ' ' + v }         » "List: zero one one point five two three"
```

Part IV

Regular Expressions

Regular Expressions

- Any character except `\/^$|.+*?()[]\{\}`, matches itself.
- `^` matches the start of a line, `$` matches the end of a line.
- `.` matches any character.
- If `a`, `b` are regular expressions, then:
  - `ab` is also a regular expression, that matches the concatenated strings.
  - `a*` is a regular expression matching the hull of `a`.
  - `a+` is equivalent to `aa*`.
  - `a|b` matches either `a` or `b`.
  - Expressions can be grouped by brackets. E.g: `(a|b)c` matches $\{‘ac’, ‘bc’\}$, `a|bc` matches $\{‘a’, ‘bc’\}$.
- `[characters]` Matches a range of characters. Example: `[a-zA-Z0-9]` matches the alphanumeric characters.
- `[^characters]` Matches the negation of a range of characters. Example: `[^a-zA-Z0-9]` matches all non-alphanumeric characters.
- `+`, and `*` are greedy, `+?`, `*?` are the non-greedy versions .
- `(?=regexp)` and `(?!regexp)` is positive and negative lookahead.
- There exist a couple of shortcuts for character classes. E.g. `\w` = `[0-9A-Za-z_]`, `\W` = `[^0-9A-Za-z_]`, `\s` = `[ \t\n\r\f]`, `\S` = `[^ \t\n\r\f]`,

More information can be found at: http://www.regular-expressions.info/tutorial.html

Regular Expressions

Examples

```
# Simple regexps
/ruby/ =~ 'perls and rubys'           » 10
/ruby/ =~ 'complicated'               » nil
/b(an)*a/ =~ 'ba'                     » 0
/b(an)*a/ =~ 'some bananas'           » 5
/^b(an)*a/ =~ 'some bananas'          » nil
/[tj]im/ =~ 'tim'                     » 0
/[tj]im/ =~ 'jim'                     » 0
/[tj]im/ =~ 'vim'                     » nil

# Extracting matches
/(.*) (.*)/ =~ 'thats ruby'           » 0
[$1, $2]                              » ["thats", "ruby"]

# The OO way
re = /name: "(.*)"/                   » /name: "(.*)"/
mr = re.match('name: "brian"')        » #<MatchData:0x402c1fc0>
mr[1]                                 » "brian"
```

Some functions

```
def showRE(string, regexp)
  if regexp =~ string then "#{$`}<#{$&}>#{$'}" else "no match" end
end                                                   » nil

a = "The moon is made of cheese"                      » "The moon is made of cheese"
showRE(a, /\w+/)                                      » "<The> moon is made of cheese"
showRE(a, /\s.*\s/)                                   » "The< moon is made of >cheese"
showRE(a, /\s.*?\s/)                                  » "The< moon >is made of cheese"
showRE(a, /[aeiou]{2,99}/)                            » "The m<oo>n is made of cheese"
showRE(a, /mo?o/)                                     » "The <moo>n is made of cheese"

a = "rubys    are  brilliant  \t gemstones"           » "rubys    are  brilliant  \t gemstones"
a.gsub(/[aeiou]/, '*')                                » "r*bys    *r*  br*ll**nt  \t g*mst*n*s"
a.gsub!(/\s+/, ' ')                                   » "rubys are brilliant gemstones"
a.gsub(/(^|\s)\w/) { | match | match.upcase }         » "Rubys Are Brilliant Gemstones"
a.split(/ /)                                          » ["rubys", "are", "brilliant", "gemstones"]
a.scan(/[aeiou][^aeiou]/)                             » ["ub", "ar", "e ", "il", "an", "em", "on", "es"]
a.scan(/[aeiou](?=[^aeiou ])|
        [^aeiou ](?=[aeiou])/x).length                » 14

File.open('/usr/share/dict/words') { | words |
  words.select { | word | /a.*e.*i.*o.*u/ =~ word }
}[0..2].map{ | word | word.strip }                    » ["abstemious", "adventitious", "facetious"]
```

Exercises: Regular Expressions

Simple Match

Write a regular expression that matches lines, that begin with the string “USERNAME:”.

Character Classes

Write a function that extracts the tag names from a html document. E.g.

```
require 'open-uri.rb'                             » true
html = open("http://www.google.de/") { | f | f.read } » "<html><head><meta http-equiv=\"
tag_names(html)                                    » ["html", "head", "meta", "title", "style"
```

Extract Username

Write a regular expression that extracts the username from a string of the form “USERNAME: Brian”.

Extract Version Number

Include a function into the chat server that checks that the handshake string given by the chat client is correct, and returns the protocol version. If the string is not correct, raise an ENoYASCClient exception.

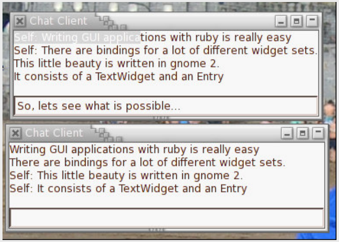
Part V

Application development

Ruby Gnome 2 Bindings

```ruby
Gtk.init

class ChatGUI < MainWindow
  def initialize(client)
    super('Chat Client')
    @client = client

    vbox = VBox.new
    self.add(vbox)

    @received = TextView.new()
    @received.editable = false

    @input = Entry.new
    @input.signal_connect(:activate) do send_line end

    vbox.pack_start(@received, true, true, 0)
    vbox.pack_start(@input, false, false, 0)

    @client.register(:line_received) do | c, line |
      @received.buffer.text += line
    end
    self.signal_connect(:destroy) do @client.close end
  end
```

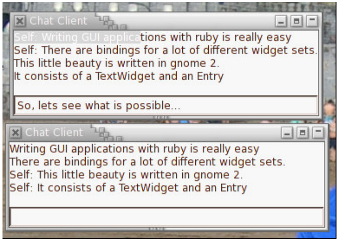
```ruby
  def send_line
    @client.send(@input.text)
    @received.buffer.text +=
      "Self: #{@input.text}\n"
    @input.text = ''
  end

  def start
    @client.listen
    self.show_all
    Gtk.main
  end
end
```



```
Self: Writing GUI applications with ruby is really easy
Self: There are bindings for a lot of different widget sets.
This little beauty is written in gnome 2.
It consists of a TextWidget and an Entry
So, lets see what is possible...
```

```
Writing GUI applications with ruby is really easy
There are bindings for a lot of different widget sets.
Self: This little beauty is written in gnome 2.
Self: It consists of a TextWidget and an Entry
```

```
require 'gtk2'
require '../../ants/observable'

class MainWindow < Gtk::Window
  include Gtk

  def initialize(title = nil)
    super()
    set_title("#{title}") if title
    signal_connect(:destroy) do Gtk.main_quit end
  end

  def quit
    destroy
    true
  end
end
```



Self: Writing GUI applications with ruby is really easy
Self: There are bindings for a lot of different widget sets.
This little beauty is written in gnome 2.
It consists of a TextWidget and an Entry

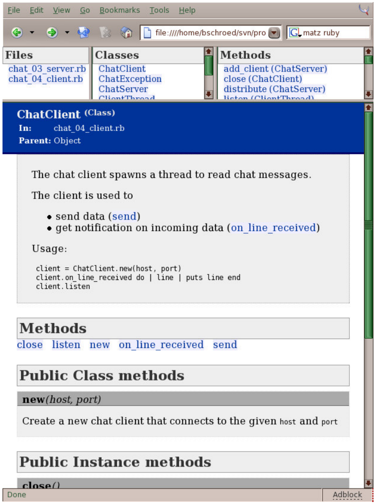
So, lets see what is possible...

Writing GUI applications with ruby is really easy
There are bindings for a lot of different widget sets.
Self: This little beauty is written in gnome 2.
Self: It consists of a TextWidget and an Entry

Documentation

The standard for documenting ruby programs is `rdoc`. From rdoc documentation the `ri` documentation and the standard library documentation is created. `rdoc` uses a wiki-like unobtrusive markup. E.g.

```ruby
# The chat client spawns a thread that
# receives incoming chat messages.
#
# The client is used to
# * send data (#send)
# * get notification on incoming data
#   (#on_line_received)
#
# Usage:
#  client = ChatClient.new(host, port)
#  client.on_line_received do | line | puts line end
#  client.listen
class ChatClient

  # Create a new chat client that connects to the
  # given +host+ and +port+
  def initialize(host, port)
    @socket = TCPSocket.new(host, port)
    @on_receive = nil
```



ChatClient (Class)
In: chat_04_client.rb
Parent: Object

The chat client spawns a thread to read chat messages.

The client is used to

- send data (send)
- get notification on incoming data (on_line_received)

Usage:

```
client = ChatClient.new(host, port)
client.on_line_received do | line | puts line end
client.listen
```

Methods

close listen new on_line_received send

Public Class methods

new(*host, port*)

Create a new chat client that connects to the given `host` and `port`

Public Instance methods

close()

Unit Testing

- Unit tests are small programs, that compare the behaviour of your program against specified behaviour.
- Unit tests are collected while developing an application/library.
- Unit tests save you from breaking something with one change which you did not take into account when applying the change.

Example

```ruby
#!/usr/bin/ruby -w

require 'faculty_1'
require 'test/unit'

class TC_Faculty < Test::Unit::TestCase

  @@faculties = [[0, 1], [1, 1], [2, 2], [3, 6], [4, 24], [6, 720], [13, 6227020800]]

  def test_faculty
    @@faculties.each do | i, i_fac |
      assert_equal(i_fac, i.fac,"#{i}.fac returned wrong value.")
    end
  end
end
```

Unit Testing - Examples

Library

```ruby
class Integer
  def fac
    (1..self).inject(1) { | r, v | r * v }
  end
end
```

Test

```ruby
  def test_faculty
    @@faculties.each do | i, i_fac |
      assert_equal(i_fac, i.fac,"#{i}.fac returned wrong value.")
    end
  end
```

Result of Testsuite

```
Loaded suite faculty_1_test_1
Started
.
Finished in 0.001138 seconds.

1 tests, 7 assertions, 0 failures, 0 errors
```

Unit Testing - Examples

Test

```
#!/usr/bin/ruby -w

require 'faculty_1'
require 'test/unit'

class TC_Faculty < Test::Unit::TestCase

  @@faculties = [[0, 1], [1, 1], [2, 2], [3, 6], [4, 24], [6, 720], [13, 6227020800]]

  def test_faculty
    @@faculties.each do | i, i_fac |
      assert_equal(i_fac, i.fac, "#{i}.fac returned wrong value.")
    end
  end

  def test_negative
    assert_raise(ENegativeNumber, '-1! should raise exception') do -1.fac end
    assert_raise(ENegativeNumber, '-10! should raise exception') do -10.fac end
    assert_raise(ENegativeNumber, '-111! should raise exception') do -111.fac end
  end
end
```

Unit Testing - Examples

Test

```ruby
def test_negative
  assert_raise(ENegativeNumber, '-1! should raise exception') do -1.fac end
  assert_raise(ENegativeNumber, '-10! should raise exception') do -10.fac end
  assert_raise(ENegativeNumber, '-111! should raise exception') do -111.fac end
end
```

Result of Testsuite

```
Loaded suite faculty_2_test_1
Started
.E
Finished in 0.001428 seconds.

  1) Error:
test_negative(TC_Faculty):
NameError: uninitialized constant TC_Faculty::ENegativeNumber
    faculty_2_test_1.rb:17:in `test_negative'

2 tests, 7 assertions, 0 failures, 1 errors
```

Unit Testing - Examples

Library

```
class ENegativeNumber < Exception; end

class Integer
  def fac
    raise ENegativeNumber if self < 0
    (1..self).inject(1) { | r, v | r * v }
  end
end
```

Unit Testing - Examples

Test

```ruby
  def test_negative
    assert_raise(ENegativeNumber, '-1! should raise exception') do -1.fac end
    assert_raise(ENegativeNumber, '-10! should raise exception') do -10.fac end
    assert_raise(ENegativeNumber, '-111! should raise exception') do -111.fac end
  end
```

Result of Testsuite

```
Loaded suite faculty_2_test_2
Started
..
Finished in 0.001925 seconds.

2 tests, 10 assertions, 0 failures, 0 errors
```

Literature

James Britt:
The ruby-doc.org ruby documentation project.
`http://www.ruby-doc.org/`

Chad Fowler:
Ruby Garden Wiki.
`http://www.rubygarden.org/ruby/`

ruby-lang.org editors <www-admin@ruby-lang.org>:
The Ruby Language.
`http://www.ruby-lang.org/`

Dave Thomas:
RDOC - Ruby Documentation System.
`http://www.ruby-doc.org/stdlib/libdoc/rdoc/rdoc/index.html`

Dave Thomas, Chad Fowler, and Andy Hunt:
Programming Ruby - The Pragmatic Programmer's Guide.
Addison Wesley Longman, Inc, 1st edition, 2001.
`http://www.ruby-doc.org/docs/ProgrammingRuby/`

Dave Thomas, Chad Fowler, and Andy Hunt:
Programming Ruby - The Pragmatic Programmer's Guide.
Addison Wesley Longman, Inc, 2nd edition, 2004.