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
```
Functions are defined using the `def` keyword

```
#!/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.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(5.6).round</code></td>
<td>6</td>
</tr>
<tr>
<td><code>(5.6).class</code></td>
<td><code>Float</code></td>
</tr>
<tr>
<td><code>(5.6).round.class</code></td>
<td><code>Fixnum</code></td>
</tr>
<tr>
<td><code>'a string'.length</code></td>
<td>8</td>
</tr>
<tr>
<td><code>'a string'.class</code></td>
<td><code>String</code></td>
</tr>
<tr>
<td><code>'tim tells'.gsub('t', 'j')</code></td>
<td>&quot;jim jells&quot;</td>
</tr>
<tr>
<td><code>'abc'.gsub('b', 'xxx').length</code></td>
<td>5</td>
</tr>
<tr>
<td><code>['some', 'things', 'in', 'an', 'array'].length</code></td>
<td>5</td>
</tr>
<tr>
<td><code>['some', 'things', 'in', 'an', 'array'].reverse</code></td>
<td><code>['array', 'an', 'in', 'things', 'some']</code></td>
</tr>
<tr>
<td><code>Float.class</code></td>
<td><code>Class</code></td>
</tr>
<tr>
<td><code>Class.class</code></td>
<td><code>Class</code></td>
</tr>
<tr>
<td><code>Object.class</code></td>
<td><code>Class</code></td>
</tr>
</tbody>
</table>
**Class definition and inheritance**

**Base Class**

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

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

end
```

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

**Sub Class**

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

  puts Matz.new.greet
end
```

```ruby
puts brian.greet
```

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

```ruby
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
1 def greet(*names)
2   case names.length
3     when 0
4       "How sad, nobody wants to hear my talk."
5     when 1
6       "Hello #{name}. At least one wants to hear about ruby."
7     when 2..5
8       "Hello #{names.join(', ')}! Good that all of you are interested."
9     when 6..10
10      "#{names.length} students. That's perfect. Welcome to ruby!"
11     else
12       "Wow #{names.length} students. We'll have to find a bigger room."
13   end
14 end
15
16 puts greet('Ashraf', 'Ingo', 'Jens', 'Johannes', 'Marius', 'Robert',
17   'Stefan', 'Thorsten', 'Tobias', 'Jet Loong')
```

1 10 students. That's perfect. Welcome to ruby!
Basic syntax rules

Ruby syntax tries to omit “noise”

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

# 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
```

Brian Schröder Ruby Course 2004
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.
• 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.
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*l*l*
"hello".tr(’^aeiou’, ’*’) => "*e**o"
"hello".tr(’el’, ’ip’) => "hippo"
"hello".tr(’a-y’, ’b-z’) => "ifmmp"
$ irb --simple-prompt

```ruby
>> 'hal'.tr('za-y', 'ab-z')
=> "ibm"

>> class String
>>   def rot13
>>     self.tr('a-z', 'n-za-m')
>>   end
>> end

>> 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:

```ruby
#!/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`. 
Basic Containers

Array

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

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

3. **# Default Values can be used ...**
   ```ruby
   Array.new(10, 0)
   ```
   ```ruby
   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
   ```

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

```ruby
# 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'
h1['fruit'] = '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]
b = ['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, ' ' # entry
end

i am a banana

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

Blocks can be enclosed by `do | ... end`.

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

or by braces `{ | | ... }`

```ruby
1 [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.
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
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

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
```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.
# Every assignment returns the assigned value
1 a = 4

# So assignments can be chained
2 a = b = 4
3 a + b

# and used in a test
4 file = File.open('../slides.tex')
5 linecount = 0
6 linecount += 1 while (line = file.gets)

# Shortcuts
7 a += 2
8 a = a + 2

# Parallel assignment
9 a, b = b, a

# Array splitting
10 array = [1, 2]
11 a, b = *array
Ruby has all standard control structures. And you may even write them to the right of an expression.

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

"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.

```ruby
1  def is_true(value)
2      value ? true : false
3  end

4

5  is_true(false)  » false
6  is_true(nil)    » false
7  is_true(true)   » true
8  is_true(1)      » true
9  is_true(0)      » true
10 is_true([0,1,2]) » true
11 is_true('a..'z') » true
12 is_true('')     » true
13 is_true(:a_symbol) » true
```

Join the equal rights for zero movement!

Brian Schröder
Ruby Course 2004
Ruby has a variety of loop constructs, but don’t forget the blocks!

```ruby
1 i = 1
2
3 while (i < 10)
4   i *= 2
5 end
6 i
7
8 i *= 2 while (i < 100)
9 i
10
11 begin
12   i *= 2
13 end while (i < 100)
14 i
15
16 i *= 2 until (i >= 1000)
17 i
18
19 loop do
20   break i if (i >= 4000)
21   i *= 2
22 end
23 i
24
25 4.times do i *= 2 end
26 i
27
28 r = []
29 for i in 0..7
30   next if i % 2 == 0
31   r << i
32 end
33 r
34
35 # Many things are easier with blocks:
36 (0..7).select { |i| i % 2 != 0 }
37
```

Brian Schröder Ruby Course 2004
Exercises: Control Structures

Fibonacci

Write functions that calculate the fibonacci numbers using different looping constructs

\[
\text{fib}(i) = \begin{cases} 
0 & \text{if } i = 0 \\
1 & \text{if } i = 1 \\
\text{fib}(i - 1) + \text{fib}(i - 2) & \text{otherwise}
\end{cases}
\]

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 \( \text{fib}(0), \text{fib}(1), \ldots \text{fib}(n) \)

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
  end
  semaphore.synchronize do
    clients.delete(socket)
  end
  socket.close
end
```
A Simple Chat

$/home/bschoed/svn/projekte/rubycourse/sou...

$ ./chat_01_server.rb localhost 1111

$/home/bschoed/svn/projekte/rubycourse/sou...

$ ./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/bschoed/svn/projekte/rubycourse/sou...

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

Brian Schröder Ruby Course 2004
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
    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

$ ./chat_02_server.rb localhost 1111

$ ./chat_02_client.rb localhost 1111
Anyone out there?
Received: Yes me!
Ah! me too.

$ ./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
1  class EChatException < Exception; end
2  class EInvalidServerException < EChatException; end
3  class EServerDiedException < EChatException; end

5  def chat
6     begin
7         # ...
8         raise EServerDiedException
9         # ...
10        rescue EInvalidServerException
11         puts "Invalid server"
12         raise
13        rescue EServerDiedException
14         puts "Server died"
15         raise
16     end
17  end

19    begin
20       #...
21       chat
22       #...
23      rescue EChatException => e
24      puts "#{e} occurred."
25      puts "Please contact your " + "system administrator ;)"
26    end
```
Exception Handling

Unhandled exception: No server is running.

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

```
1 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.

```ruby
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

```ruby
file = File.open('/usr/share/dict/words')  #<File:/usr/share/dict/words>
beg
file = File.open('/usr/share/dict/words') { | file |
  file.select { | word | word[0..1] == 'xy' }
}
```
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 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
$ ./chat_03_server.rb localhost 1111

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

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

1 class Cell
2   def initialize
3     @state = :empty
4   end
5 end

7 class Board
8   def initialize(width, height)
9     @width = width; @height = height
10    @cells = Array.new(height) { Array.new(width) { Cell.new } }
11 end
12 end

Access a property

14 class Cell
15   def state
16     @state
17   end
18 end

20 cell = Cell.new  #<Cell::... @state=:e...>
21 cell.state        :empty

Calculated property

50 class Board
51   def size
52     self.width * self.height
53   end
54 end

Shortcut

34 class Cell
35   attr_reader :state
36 end
Accessor Functions: Setting object properties

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

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
```

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
# Accessor Functions - Array-like accessors

class Cell
  def initialize
    @state = :empty
  end
end

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
56 class Board
57   def [](col, row)
58     @cells[col][row]
59   end
60 end
```

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

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

```ruby
62 class Board
63   def []=(col, row, cell)
64     @cells[col][row] = cell
65   end
66 end
```

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

Brian Schröder Ruby Course 2004
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.
Classes, functions, modules can be modified at runtime.

```
25  class PersonShort < BasePerson
26    attr_accessor :name, :surname
27  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, 5040, 40320, 362880, 3628800, 39916800, 479001600, 6227020800
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.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>[0,1,2,3,4,5].shuffle</code></td>
<td><code>[4, 3, 2, 5, 1, 0]</code></td>
</tr>
<tr>
<td>2</td>
<td><code>[0,1,2,3,4,5].shuffle</code></td>
<td><code>[2, 5, 1, 3, 0, 4]</code></td>
</tr>
<tr>
<td>3</td>
<td><code>[0,1,2,3,4,5].shuffle</code></td>
<td><code>[3, 4, 1, 5, 2, 0]</code></td>
</tr>
</tbody>
</table>

Set

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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>a1 = [1, 2, 3]</code></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td><code>a2 = [2, 3, 4]</code></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td><code>a3 = [{'c' =&gt; 'a', 'v' =&gt; 1}, {'c' =&gt; 'b', 'v' =&gt; 2}]</code></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td><code>a4 = [{'c' =&gt; 'b', 'v' =&gt; 2}, {'c' =&gt; 'c', 'v' =&gt; 3}]</code></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td><code>a1.intersect(a2)</code></td>
<td><code>[2, 3]</code></td>
</tr>
<tr>
<td>6</td>
<td><code>a2.intersect(a1)</code></td>
<td><code>[2, 3]</code></td>
</tr>
<tr>
<td>7</td>
<td><code>a1.intersect(a3)</code></td>
<td><code>[]</code></td>
</tr>
<tr>
<td>8</td>
<td><code>a3.intersect(a4)</code></td>
<td><code>[{v=&gt;2, c=&gt;&quot;b&quot;}]</code></td>
</tr>
<tr>
<td>9</td>
<td><code>a1.union(a2).union(a3)</code></td>
<td><code>[1, 2, 3, 2, 3, 4, {v=&gt;1, c=&gt;&quot;a&quot;}, {v=&gt;2, c=&gt;&quot;b&quot;}]</code></td>
</tr>
<tr>
<td>10</td>
<td><code>a1.intersect(a1.union(a2))</code></td>
<td><code>[1, 2, 3]</code></td>
</tr>
</tbody>
</table>
```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

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

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
2 Yukihiro → Yukihiro Matsumoto
3 Yukihiro Matsumoto → Matz

Brian Schröder
Ruby Course 2004
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.

```ruby
1  root = TreeItem.new("root")
2  root.to_a.join(' | ')
3  root.left = TreeItem.new("left")
4  root.to_a.join(' | ')
5  root.right = TreeItem.new("reft")
6  root.to_a.join(' | ')
7  root.left.left = TreeItem.new("left-left")
8  root.to_a.join(' | ')
9  root.left.right = TreeItem.new("left-right")
10 root.to_a.join(' | ')
11 root.inject(0) { | r, e | r + 1 }
```

```ruby
» #<TreeItem:0x40293cec @item="root">
» "root"
» #<TreeItem:0x403079f8 @item="left">
» "root | left"
» #<TreeItem:0x402eb5dc @item="reft">
» "root | left | reft"
» #<TreeItem:0x402e5178 @item="left-left">
» "root | left | left-left | reft"
» #<TreeItem:0x402dd5f4 @item="left-right">
» "root | left | left-left | left-right | reft"
» 5
```
```ruby
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

root = TreeItem.new("root")
root.to_a.join(' | ')  # "root"
root.left = TreeItem.new("left")
root.to_a.join(' | ')  # "root / left"
root.right = TreeItem.new("reft")
root.to_a.join(' | ')  # "root / left / reft"
root.left.left = TreeItem.new("left-left")
root.to_a.join(' | ')  # "root / left / left-left / reft"
```

Brian Schröder Ruby Course 2004
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.

```ruby
1 one = ListItem.new("one")
2 one.next = ListItem.new("two")
3 one.next.next = ListItem.new("three")
4 one.previous = ListItem.new("zero")
5 one.inject('List:') { | r, v | r + ' ' + v }
6
7 one.insert ListItem.new("one point five")
8 one.inject('List:') { | r, v | r + ' ' + v }
```

```ruby
"List: zero one two three"

"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
# Simple regexps

1. `/ruby/ =~ 'perls and rubys'` » 10
2. `/ruby/ =~ 'complicated'` » nil
3. `/b(an)*a/ =~ 'ba'` » 0
4. `/b(an)*a/ =~ 'some bananas'` » 5
5. `/b(an)*a/ =~ 'some bananas'` » nil
6. `/^b(an)*a/ =~ 'some bananas'` » nil
7. `/[tj]im/ =~ 'tim'` » 0
8. `/[tj]im/ =~ 'jim'` » 0
9. `/[tj]im/ =~ 'vim'` » nil

# Extracting matches

10. `/(.*) (.*)/ =~ 'thats ruby'` » 0
11. `[$1, $2]` » ['thats', 'ruby']

# The OO way

12. `re = /name: "(.*)"/` » /name: "(.*)"/
13. `mr = re.match('name: "brian"')` » #<MatchData:0x402c1fc0>
14. `mr[1]` » "brian"
Some functions

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

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(/\s+/)
  => ["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.

1. `require 'open-uri.rb'`
2. `html = open("http://www.google.de/") { | f | f.read }` » `<html><head><meta http-equiv="`true
3. `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
Gtk.init

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

  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

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
```

Writing GUI applications with Ruby is really easy. 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.
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
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
    end
```

The chat client spawns a thread to read chat messages.

The chat client is used to
- send data (send)
- get notification on incoming data (on_line_received)

Usage:
```ruby
client = ChatClient.new(host, port)
client.on_line_received do | line | puts line end
client.listen
```
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
1 class Integer
2   def fac
3     (1..self).inject(1) { | r, v | r * v }
4   end
5 end
```

Test

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

Result of Testsuite

```
1 Loaded suite faculty_1_test_1
2 Started
3  
4 Finished in 0.001138 seconds.
5 1 tests, 7 assertions, 0 failures, 0 errors
```
#!/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

Brian Schröder Ruby Course 2004
Unit Testing - Examples

```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

```
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
class ENegativeNumber < Exception; end

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

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

### Result of Testsuite

```
1  Loaded suite faculty_2_test_2
2  Started
3   ..
4  Finished in 0.001925 seconds.
5
6  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.*

Dave Thomas, Chad Fowler, and Andy Hunt:
http://www.ruby-doc.org/docs/ProgrammingRuby/

Dave Thomas, Chad Fowler, and Andy Hunt: