"Advanced" Ruby

Weiterführende Themen in Ruby

Roland Kluge, 1. August 2013

Lambdas, Procs und Blocks I

- Ruby bietet Funktionen als Objekte
  - Blocks sind **keine** Objekte!
- Zwei "Ersatzkonzepte" dafür: **Procs** und **Lambdas**
  - **Lambda** überprüft Parameter und übergibt Kontrollfluss an aufrufende Methode
  - **Proc** nimmt als Defaultwert *Nil* and und beendet die aufrufende Methode
- Beispiel: Filterung der folgenden Liste nach Integern:

```
l = [:weezard, 42, "Trady Blix"]
```

Realisierung mittels Block

```
ints = l.select {|x|
  x.is_a? Integer
}
```

Realisierung mittels Proc

```
filter = Proc.new do |x|
  x.is_a? Integer
end

ints = l.select(&filter)
```

Realisierung mittels Lambda

```
filter = Proc.new do |x|
  x.is_a? Integer
end

ints = l.select(&filter)
```

Lambdas, Procs und Blocks II

- Jede Methode akzeptiert einen Block/Proc/Lambda und kann diesen mittels *yield* aufrufen

```ruby
def my_method(x)
  yield x
end

my_method(2) {|value| puts 2 * value}
# => 4
```

- Procs und Lambdas kann man auch als Paramter übergeben

```ruby
def my_method(x, f)
  f.call x
end

f = lambda {|value| puts 2 * value}
my_method(2, f)
# => 4
```

Lambdas, Procs und Blocks III

- Beispiele für Bibliotheksmethoden, die Blöcke/Procs nutzen
  - `["a", "b"].each {|elem| puts elem}`
  - `{"a" => 1, "z" => 26}.each { |k,v| puts k.to_s + ":" + v.to_s }`
  - `squared = (1..4).to_a.collect { |x| x**2 }`
  - `div_by_3 = (1..6).to_a.select { |x| x % 3 == 0}`
  - `as_string = (1..6).to_a.map(&:to_s)`
  - `as_sym = ["a","b"].map(&:to_sym)`

Objektorientierung in Ruby

- ***"Everything* in Ruby is an object"**
- Keine Interfaces (**Duck Typing**)
- Sichtbarkeitsniveaus: public/private
- Organisation von Klassen in **Modulen** (~Packages)

Klassen anlegen

- Syntax:

```
class <ClassNameCamelCase>
end
```

- *initialize* entspricht einem Konstruktor
- Implizite Memberdeklaration (*@name, @creator)*

```
class Language
  def initialize(name, creator)
    @name = name
    @creator = creator
  end

  def description
    puts "I'm #{@name} and I was created by #{@creator}!"
  end
end
```

Klassen nutzen

- Instantiierung über *ClassName.new*
  - Ruft *ClassName.initialize* auf
- Zugriff auf Member ist per default gesperrt (*creator*)

```
lang = Language.new("Ruby", "me")
lang.description
# I'm Ruby and I was created by me!
lang.creator
# NoMethodError: undefined method `creator' for
#<Language:0x000000014878c0>
```

```
class Language
  def initialize(name, creator)
    @name = name
    @creator = creator
  end

  def description
    puts "I'm #{@name} and I was created by #{@creator}!"
  end
end
```

Sichtbarkeit (*Scopes*)

```ruby
class Message
  @@message_count = 0

  def initialize(content)
    @content = content
    # Not @@message_count++
    @@message_count += 1
  end

  def print_content; puts @content; end
  def self.print_count; puts @@message_count; end
end

Message.print_count # => 0
m = Message.new("My message text")
Message.print_count # => 1
m.print_content # => "My message text"
```

Variablenscopes:
- $var: global (auch außerhalb)
- @var: Instanzvariable
- @@var: Klassenvariable

Methodenscopes:
- def call_me; end: Instanzmethode
- def self.call_me; end: statische Methode

Geheimnisprinzip

- Member sind **per default nicht sichtbar** (-> *NoMethodError*)
  - *attr_reader*: read-only
  - *attr_writer*: write-only
  - *attr_accessor*: read & write
- Für Methoden gibt es zwei Sichtbarkeitslevel
  - *public* und *private*
  - Alle Methoden **per default public**.
  - Sichtbarkeit gilt ab Schlüsselwort

```ruby
class Car
  attr_reader :serial_number
  attr_writer :driver
  attr_accessor :tank_filling_level
  def initialize(serial_number)
    @serial_number = serial_number;
  end
end

c.driver = "Janne"
```

```ruby
class A
  def public_by_default; end
  private
  def private_method; end
  def another_private_method; end
  public
  def public_again; end
end
```

Vererbung

- Keine Mehrfachvererbung, dafür aber *include* und *extend*
- Aufruf von Oberklasse-Methoden *super*

```
class Car
  attr_reader :driver
  def initialize(driver)
    @driver = driver;
  end
end

class Taxi < Car; end

taxi = Taxi.new("Harald")
taxi.driver # => "Harald"
```

Module importieren

- Kann "Namensraum" importieren mittels *include*
  - ähnlich dem *import.static* von Java oder *using namespace* von C++

```
class Angle
  include Math

  def initialize(radians)
    @radians = radians
  end

  def print_cosine
    # Without include: Math::cos(@radians)
    puts cos(@radians)
  end
end
```

Mixins

- Kann von Extern Methoden der Klasse hinzufügen

```
module Action
  def jump
    @distance = rand(4) + 2
    puts "I jumped forward #{@distance} feet!"
  end
end

class Rabbit
  include Action
end

class Cricket
  include Action
end

Rabbit.new.jump
Cricket.new.jump
```