Musterlösungen

Aufgabe (1)

```
def letztes_zeichen (satz)
  satz[-1]
end
```

Aufgabe (2)

```
def satztyp(satz)
  if letztes_zeichen(satz) == "."
    puts "Aussagesatz"
  elsif letztes_zeichen(satz) == "?"
    puts "Fragesatz"
  elsif letztes_zeichen(satz) == "!"
    puts "Befehlssatz"
  else
    puts("unbekannter Satztyp")
  end
end
```

Musterlösungen

Aufgabe (3)

```
def tokisieren_string(text)
  text.split.each{|token| puts token}
end
```

Aufgabe (4)

```
def string_to_hash(string)
  ergebnis = Hash.new(0)
  string.split.each{|word| ergebnis[word] +=1}
  return ergebnis
end
```

Skriptsprachen
* Ruby *

03 - Klassen, Objekte und Variablen

Naumann

Sommersemester 2014

3.1 Klassen

- Anders als LISP/CLOS kennt Ruby keine Mehrfachvererbung: Zu jeder Klasse
  gibt es maximal eine Superklasse; d.h. die Klassenhierarchie kann als ein Baum dargestellt werden.

- Ab Ruby 1.9 bildet die Klasse Object die Wurzel der Klassenhierarchie.

- Neue Klassen lassen sich mit wenig Aufwand definieren:

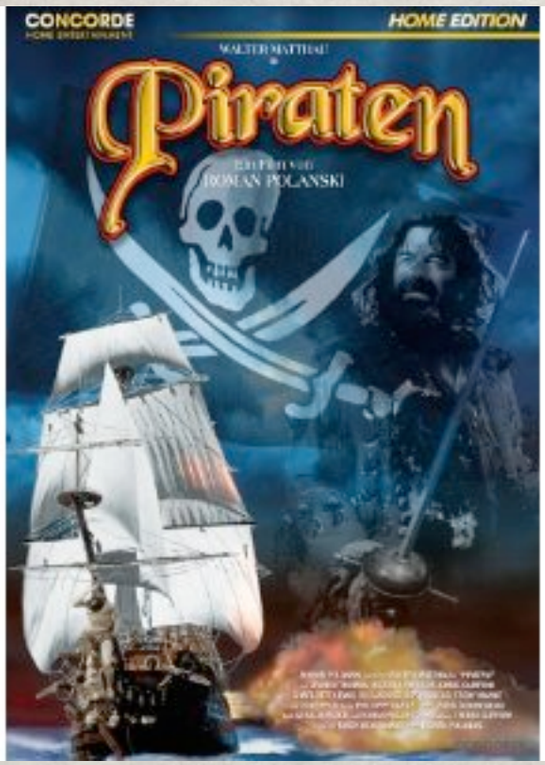
```
class Name    # Der Name einer Klasse muss mit einen Großbuchstaben beginnen
    ...       # Methoden werden in die Klassendefinition eingeschlossen
end           # Die Klassendefinition wird mit ‘end‘ abgeschlossen
```

3.1 Klassen

Beispielszenario

Angenommen, Sie verfügen über eine umfangreiche DVD-Sammlung und über mehrere Barkode-Scanner. Zusammen mit Freunden wollen Sie ihre Sammlung inventarisieren. Ihr System ist in der Lage, auf Grundlage der von einem Scanner gelieferten Daten eine CSV-Datei (Angaben durch Kommata voneinander getrennt) zu erzeugen, die pro DVD folgende Daten enthält:

< Datum, ISBN, Preis >.



“Datum“, “ISBN“, “Preis“
“2012-04-28“, “3-7742-1216-3“, “4.99“
“2012-04-28“, “3-453-86100-0“, “8.98“
“2012-04-28“, “0790745658“, “9.29“

3.1 Klassen

```
class MeineDVDs                    # minimale Klassendefinition
end

dvd1 = MeineDVDs.new               # 1. Instanz der Klasse MeineDVDs
dvd2 = MeineDVDs.new               # 2. Instanz dieser Klasse
                                   # Das Ergebnis sind 2 Objekte, die sich aber in Bezug
                                   # auf ihre Eigenschaften sich nicht unterscheiden.
```

Wenn Problem man das Datum als eine externe Angabe hier nicht berücksichtigt, müssen pro DVD zwei Angaben (ISBN und Preis) gespeichert werden. Dazu dienen Instanzenvariablen:

```
class MeineDVDs                    # erster (wenige erfolgreicher) Versuch
@isbn
@preis
end
```

Das Problem dieses Versuchs liegt darin, dass es nicht möglich ist, für Instanzen dieser Klasse individuelle Werte für die Instanzenvariablen festzulegen:

```
dvd1 = MeineDVDs.new(1234567890, 9.99)     # => 'initialize': wrong number of
                                           #      arguments (2 for 0) (ArgumentError)
```

3.1 Klassen

In Ruby gibt es eine generische Methode initialize, die es ermöglicht, genau dieses Problem zu lösen: Sie erlaubt es festzulegen, wievele Argumente bei der Generierung von Instanzen einer Klasse übergeben werden können/ müssen und wie/wo diese Werte gespeichert werden:

```
class MeineDVDs
  def initialize(isbn, preis)          # Die Methode initialize wird bei der Erzeugung neuer
      @isbn = isbn                     # Instanzen aufgerufen. Jedes Objekt hat zwei Eigen-
      @preis = Float(preis)            # schaften: eine ISBN-Nummer und einen Preis.
  end
end
```

! : isbn, preis - Parameter der initialize-Methode
   : @isbn, @preis - Instanzenvariablen

3.1 Klassen

```
dvd1 = MeineDVDs.new("isbn1", 3)
dvd2 = MeineDVDs.new("isbn2", 3.14)
dvd3 = MeineDVDs.new("isbn3", "5.67")

p dvd1
p dvd2
p dvd3
```

```
#<MeineDVDs:0x0a37f0 @isbn="isbn1", @preis=3.0>
#<MeineDVDs:0x0a3584 @isbn="isbn2", @preis=3.14>
#<MeineDVDs:0x0a3354 @isbn="isbn3", @preis=5.67>

puts dvd1
puts dvd2
puts dvd3

#<MeineDVDs:0x0a38cc>
#<MeineDVDs:0x0a3764>
#<MeineDVDs:0x0a36d8>
```

Der Unterschied zwischen puts und p besteht darin, dass puts einen String über die Standardausgabe des Programms (Bildschirm) ausgibt und alle anderen Argumente - sofern möglich - dazu in Strings konvertiert (to_s). Für Objekte, für die keine derartige Konvertierungs-methode zur Verfügung steht, wird nur eine kompakte, uniforme Repräsentation #<KlassenName ObjektID> ausgegeben.

3.1 Klassen

Es ist möglich und in den meisten Fällen auch sinnvoll, für selbstdefinierte Klassen eigene to_s-Methoden zu definieren.

Wie man an diesem Beispiel gut sieht, können (Klassen- und) Instanzenmethoden auf die Instanzenvariablen der Instanzen dieser Klasse zugreifen.

```
class MeineDVDs
  def initialize(isbn, preis)
    @isbn  = isbn
    @preis = Float(preis)
  end
  def to_s
    "ISBN: #{@isbn}, preis: #{@preis}"
  end
end

dvd1 = MeineDVDs.new("isbn1", 3)
dvd2 = MeineDVDs.new("isbn2", 3.14)
dvd3 = MeineDVDs.new("isbn3", "5.67")
puts dvd1
puts dvd2
puts dvd3

ISBN: isbn1, preis: 3.0
ISBN: isbn2, preis: 3.14
ISBN: isbn3, preis: 5.67
```

3.1 Klassen

Dadurch, dass wir in der Initialisierungsmethode für unsere Klasse MeineDVDs zwei Parameter spezifiziert haben, müssen bei der Generierung neuer Objekte auch immer zwei Argumente angegeben werden:

```
dvd4 = MeineDVDs.new("isbn3“)
... in `new': wrong number of arguments (1 for 2) (ArgumentError)
      from file.rb:104:in `<main>'
dvd5 = MeineDVDs.new("isbn11", 3, 9)
... :in `new': wrong number of arguments (3 for 2) (ArgumentError)
      from file.rb:104:in `<main>'
```

3.2 Objekte und Attribute

Der interne Zustand der Objekte der Klasse MeineDVDs ist zunächst vor allen Zugriffen von außen geschützt (Geheimnisprinzip): Kein anderes Objekt kann auf die Instanzenvariabeln eines Objekts zugreifen.

Eine komplette Abschottung ist aber oft nicht sinnvoll. Es gibt normalerweise bestimmte Eigenschaften von Objekten, die von außen zugänglich sein sollen. Diese Eigenschaften werden ***Attribute*** genannt.

```
class MeineDVDs
  def initialize(isbn, preis)
    @isbn  = isbn
    @preis = Float(preis)
  end
  def isbn
    @isbn
  end
  def preis
    @preis
  end
  # ..
end
dvd = MeineDVDs.new("isbn1", 12.34)
puts "ISBN   = #{dvd.isbn}"
puts "Preis  = #{dvd.preis}"
```

3.2 Objekte und Attribute

Da es häufig nötig ist, Zugriffsfunktionen für die Instanzenvariablen zu definieren, gibt es in RUBY eine Kurzform: Durch attr_reader, attr_writer und attr_accessor Deklarationen wird sichergestellt, das Lese-, Schreib- Lese/Schreibmethoden für Instanzenvariablen erzeugt werden.

Als Wert(e) dieser Schlüsselwörter werden Symbole verwendet, die die Namen für die Zugriffsmethoden festlegen.

```
class MeineDVDs

  attr_reader :isbn, :preis

  def initialize(isbn, preis)
    @isbn  = isbn
    @preis = Float(preis)
  end

  # ..
end
dvd = MeineDVDs.new("isbn1", 12.34)
puts "ISBN   = #{dvd.isbn}"
puts "Preis  = #{dvd.preis}"
```

3.2 Objekte und Attribute

```
class MeineDVDs

  attr_reader :isbn, :preis

  def initialize(isbn, preis)
    @isbn  = isbn
    @preis = Float(preis)
  end

  def preis=(neuer_preis)
    @preis = neuer_ preis
  end

  # ...
end
dvd = MeineDVDs.new("isbn1", 33.80)
puts "ISBN       = #{dvd.isbn}"
puts "Preis      = #{dvd.preis}"
dvd.preis = dvd.preis * 0.75       # Verkaufspreis
puts "Neuer Preis = #{dvd.preis}"
```

```
class MeineDVDs

  attr_reader   :isbn
  attr_accessor :preis

  def initialize(isbn, preis)
    @isbn  = isbn
    @preis = Float(preis)
  end
  # ...
end
dvd = MeineDVDs.new("isbn1", 33.80)
puts "ISBN       = #{dvd.isbn}"
puts "Preis      = #{dvd.preis}"
dvd.preis = dvd.preis * 0.75
puts "Neuer Preis = #{dvd.preis}"
```

3.2 Objekte und Attribute

Virtuelle Attribute

Die Zugriffsmethoden für die Attribute müssen sich nicht auf einfache Lese-/Schreibzugriffe beschränken:

```
class MeineDVDs

  attr_reader   :isbn
  attr_accessor :preis

  def initialize(isbn, preis)
    @isbn  = isbn
    @preis = Float(preis)
  end

  def preis_in_cents
    Integer(preis*100 + 0.5)
  end
  # ...
end
dvd = MeineDVDs.new("isbn1", 33.80)
puts "Preis              = #{dvd.preis}"              # => Preis              = 33.80
puts "Preis in Cents = #{dvd.preis_in_cents}"       # => Preis in Cents = 3380
```

3.2 Objekte und Attribute

```
class MeineDVDs

  attr_reader   :isbn
  attr_accessor :preis
  ...
  def preis_in_cents
    Integer(preis*100 + 0.5)
  end
  def preis_in_cents=(cents)
    @preis = cents / 100.0
  end
  # ...
end
dvd = MeineDVDs.new("isbn1", 33.80)
puts "Preis          = #{dvd.preis}"                  # => Preis          = 33.8
puts "Preis in Cents = #{dvd.preis_in_cents}"         # => Preis in Cents = 3380
dvd.preis_in_cents = 1234
puts "Preis          = #{dvd.preis}"                  # => Preis          = 12.34
puts "Preis in Cents = #{dvd.preis_in_cents}"         # => Preis in Cents = 1234
```

3.3 Ranges

Bereichsangaben in Ruby werden durch Angabe eines Start- und Endpunkts mit Hilfe der Operatoren .. bzw. ... gebildet. Der Unterschied zwischen beiden Operatoren liegt darin, dass im ersten Fall der Endpunkt im Bereich liegt, im zweiten Fall dagegen nicht.

```
1..10
'a'..'z'
0..."cat".length

# Bereiche können in Arrays und Enumeratoren
# konvertiert werden:

(1..10).to_a       # => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
('bar'..'bat').to_a  # => ["bar", "bas", "bat"]
enum = ('bar'..'bat').to_enum
enum.next          # => "bar"
enum.next          # => "bas"
```

3.3 Ranges

```
# Iteration über Bereiche

digits = 0..9
digits.include?(5)              # => true
digits.min                      # => 0
digits.max                      # => 9
digits.reject {|i| i < 5 }      # => [5, 6, 7, 8, 9]
digits.inject(:+)               # => 45
```

Um Bereichsangaben für selbstdefinierte Objekte bzw. Klassen verwenden zu können, reicht es aus, die `succ`-Methode und den `<=>`-Operator für diese Klasse zu definieren. `x <=> y` liefert den Wert `-1`, `0` oder `1` abhängig davon, ob das erste Argument kleiner, gleich oder größer als das zweite Argument ist.

3.3 Ranges

```
class PowerOfTwo
    attr_reader :value
    def initialize(value)
        @value = value
    end
    def <=>(other)
        @value <=> other.value
    end
    def succ
        PowerOfTwo.new(@value + @value)
    end
    def to_s
        @value.to_s
    end
end
p1 = PowerOfTwo.new(4)
p2 = PowerOfTwo.new(32)
puts (p1..p2).to_a

4
8
16
32
```

3.3 Ranges

Bereichsangaben können auch als Bedingungen verwendet werden. Sie wirken hier wie ein Schalter, der aktiviert wird, wenn der *Startpunkt* des Bereichs realisiert ist und deaktiviert wird, sobald der *Endpunkt* erscheint:

```
while line = gets
    puts line if line =~ /start/ .. line =~ /end/
end
```

Um zu überprüfen, ob ein Objekt innerhalb eines Bereiches liegt, kann man den `===`-Operator verwenden:

```
(1..10) === 5           # => true
(1..10) === 15          # => false
(1..10) === 3.14159     # => true
('a'..'j') === 'c'      # => true
('a'..'j') === 'z'      # => false
```

3.3 Ranges

```
car_age = gets.to_f # angenommen, der Wert ist 5.2
case car_age
when 0...1
    puts "Mmm.. new car smell"
when 1...3
    puts "Nice and new"
when 3...6
    puts "Reliable but slightly dinged"
when 6...10
    puts "Can be a struggle"
when 10...30
    puts "Clunker"
else
    puts "Vintage gem"
end
```

Reliable but slightly dinged

3.3 Ranges

```
car_age = gets.to_f # angenommen, der Wert ist 5.2
case car_age
when 0..0
    puts "Mmm.. new car smell"
when 1..2
    puts "Nice and new"
when 3..5
    puts "Reliable but slightly dinged"
when 6..9
    puts "Can be a struggle"
when 10..29
    puts "Clunker"
else
    puts "Vintage gem"
end
```

Vintage gem

3.3 Ranges