GIT & GITHUB

Arbeitsschritte

1. Anmelden bei GitHub.
2. In GitHub zum Repositorium naumann/Ruby-2016 wechseln.
3. Eine Kopie dieses Repositoriums erstellen (***fork***).
4. Auf dem eigenen Rechner eine lokale Kopie der persönlichen Repositoriumskopie (***fork***) erstellen (mit ***clone***).

Befehle

clone url	*git clone https://github.com/user/repo.git*
add file	
commit	
push	
pull remotename (***branchname***)	*git pull clone* ***https://github.com/user/repo.git***

GIT & GITHUB

Ressourcen

Software:

1. für Windows:
   https://windows.github.com/
2. für Mac OS:
   https://mac.github.com/

Tutorials:

1. http://www.youtube.com/watch?v=0fKg7e37bQE
2. http://www.youtube.com/watch?v=TPY8UwlTIc0
3. https://www.atlassian.com/git/tutorials

Skriptsprachen
* Ruby *

02 - Erste Schritte

Naumann
Wintersemester 2016

2.1 Klassen und Objekte

- RUBY ist eine durch und durch objektorientierte Programmiersprache:
  - Alles, was es innerhalb der Ruby-Universums gibt, ist ein Objekt.
  - Das Ergebnis jeder auf ein oder mehreren Objekten ausgeführten Operation ist selbst wieder ein Objekt.

- Klasse
  - Eine Klasse fasst Entitäten/Objekte zusammen, die bestimmte gemeinsame Eigenschaften aufweisen und *Methoden*, durch die auf die Eigenschaften dieser Objekte zugegriffen werden kann..
  - Die Objekte, die einer Klasse angehören, werden als *Instanzen* dieser Klasse bezeichnet.
  - Eigenschaften, die für alle Instanzen einer Klasse konstant sind, werden durch *Klassenvariablen*, alle anderen Eigenschaften durch *Instanzenvariablen* abgebildet.

2.1 Klassen und Objekte

- Instanzen/Objekte :

  - Erzeugung von Instanzen: Konstruktor ***new***
    Klassenname.new

  - Jede Instanz hat eine eindeutige Kennzeichnung (object ID).

  - Jede Instanz verfügt über einen eigenen Satz von Instanzenvariablen, durch die den Zustand des Objekts spezifizieren.

  - Für jede Klasse können Instanzenmethoden definiert werden.

Beispiel

Angenommen wir möchten eine mp3-Archivverwaltung in Ruby programmieren, so könnten wir eine Klasse Song definieren und jedes Lied (zunächst) durch Titel, Interpret und Länge beschreiben. Instanzen könnten dann wie folgt generiert werden:

```
song1 = Song.new("History", "The Riffles", 3.23)
song2 = Song.new ("Little Life", "Rachael Yamagata", 4.07)
...
```

2.1 Klassen und Objekte

- Methoden :
  - Die (*Instanzen*)*Methoden* beschreiben bestimmte, mit den Instanzen der Klasse ausführbare Operationen.
  - Sie ermöglichen den Zugriff auf die Instanzenvariablen eines Objektes und können seinen Zustand verändern.
  - Methoden werden aufgerufen, indem an ein Objekt eine Nachricht geschickt wird.

```
puts song1.länge     # Annahme: Es gibt eine Instanzenmethode länge
3.23                 # die als Wert die Länge eines Songs liefert.
```

  - Die Nachricht kann neben dem Methodennamen noch von der Methode geforderte Parameter enthalten.

```
objekt.methode parameter*
```

  - Neben den Instanzenmethoden gibt es auch sogenannte *Klassenmethoden*.

```
puts Song.gesamtlänge(song1, song2)
7.30
```

2.1 Klassen und Objekte

Um eine Methode auf eine Instanz einer Klasse anzuwenden, wird folgende Syntax verwendet:

empfänger.nachricht => Wert
Instanz.Methode

Der Wert selbst ist wieder ein Objekt, da das natürlich auch eine Nachricht geschickt werden kann:

empfänger.nachricht$_1$. (...) *nachricht*$_n$

```
puts “ein Versuch“.length
puts “Versuch“.index(“s“)
puts 42.even?
puts song1.play
=>
11
3
true
di da du...
```

2.1 Klassen und Objekte

- Namenskonventionen:
  - Konstanten, Modul- und Klassennamen beginnen mit einem Großbuchstaben.
  - Methodennamen sowie Bezeichner für Methodenparameter und lokale Variablen beginnen mit einem Kleinbuchstaben.
  - Globale Variablen beginnen mit dem Dollarzeichen $.
  - Instanzenvariablen erkennt man am initialen @-Zeichen.
  - Klassenvariablennamen beginnen mit @@.

2.1 Klassen und Objekte

 # Beispiel (2-1)

```
# Ruby-Kommentare beginnen mit ‚#‘.
# Der Wert eines Ausdrucks erscheint in diesen Beispielen hinter dem ‚=>‘.

1.class                 # => Fixnum
0.0.class               # => Float
true.class              # => TrueClass

# Angenommen, wir haben zuvor eine Klasse namens MeineErsteKlasse
# definiert:

MeineErsteKlasse.class  # => MeineErsteKlasse

42.even?                # => true
“Hello World“.length    # => 11
1.+ 2                   # => 3
```

2.2 Einige Beispiele

```
# Methoden werden mit def definiert.
# Struktur: def name (parameter*) anweisungen* end
def bieten(betrag)
  # Verkettung von Zeichenketten mit '+'
  result = "Ich biete " + betrag + "€!"
  return result
end

# gestern bei Sotheby ...
puts bieten(100)    # => Ich biete 100€!
puts bieten(150)    # => Ich biete 150€!
```

```
def bieten(betrag)
    # Stringinterpolation durch #{...}
    result = "Ich biete #{betrag}€!"
    return result
  end
  puts bieten(10000)# => Ich biete 10000€!
```

2.2 Einige Beispiele

```
def bieten(betrag)
    # In #{...} dürfen beliebig komplexe Formen stehen
    result = "Ich biete #{betrag.round}€!"
    return result
  end
  puts bieten(10.83) # => Ich biete 11€!
```

```
def bieten(betrag)
    # Da eine Methode den Wert der letzen Form zurückgibt,
    # ist eine explizite return-Form überflüssig
    "Ich biete #{betrag.round}€!"
  end
  puts bieten(10.23) # => Ich biete 10€!
```

2.2 Einige Beispiele

Zeichenketten

```
" ... "        # Stringinterpolation möglich; Steuerzeichen werden interpretiert
               puts "Ich esse:\n#{"drei".length} Hamburger!"
               => Ich esse:
                    4 Hamburger!

‘ ... ‘        # keine Stringinterpolation; Steuerzeichen bleiben wie sie erscheinen
               puts ‘Ich esse:\n#{"drei".length} Hamburger!‘
               => Ich esse:\n#{"drei".length} Hamburger!

string1 + string2 # Konkatenation
                  puts "Auto" + "bahn" =>
                  => Autobahn

string.split      # Zerlegung von Strings anhand von ‘white space‘-Zeichen
                  Ich esse gerne Katzen.“.split
                  => ["Ich", "esse", "gerne", „Katzen."]
string.split('.') # Parameter: Tokentrenner
                  "Y.M.C.A."
                  => ["Y", "M", "C", "A"]
```

2.3 Arrays und Hashes

Zu den wichtigsten Datenstrukturen in Ruby gehören Arrays und Hashes
Ein Array durch Verwendung eines Array-Literals [] oder durch Array.new
generiert und initialisiert werden:

 # Beispiel (2-2)

```
a = [ 1, 'cat', 3.14 ]   # Array mit drei Elementen
puts "Das erste Element ist #{a[0]}"   #=> Das erste Element ist 1
# Verändere das dritte Element:
a[2] = nil
puts "Der Array ist nun #{a.inspect}"  #=> Der Array ist nun [1, “cat“, nil]
# nil ist ein Objekt, das nichts repräsentiert.
a = [ 'ant', 'bee', 'cat', 'dog', 'elk' ]
a[0]  # => “ant“
a[3]  # => “dog“
# Eine andere Möglichkeit:
a = %w{ ant bee cat dog elk }
a[0] # => “ant“
a[3] # => “dog“
a[-1] # => “elk“
a[3][-1] # => “g“
a.include?(“dog“) # => true
```

2.3 Arrays und Hashes

Hashes sind ähnlich aufgebaut. Syntaktisch unterscheiden sie sich
(a) durch die Verwendung geschweifter Klammern {} und
(b) dadurch, das jeder Eintrag die Form hat: Schlüssel => Wert

 # Beispiel (2-3)

```
#
mein_song = {
    'Titel' => 'History',
    'Interpret' => 'The Riffles',
    'Länge' => 3.23,
    'Genre' => 'BritPop'
}

p mein_song['Titel']          # => “HIstory“
p mein_song['Länge']          # => 3.23
p mein_song['Album']          # => nil
```

```
mein_song = {
    'Titel' => 'History',
    'Interpret' => 'The Riffles',
    'Länge' => 3.23,
    'Genre' => 'BritPop'
    'Genre' => 'Pop'
}
```

2.4 Symbole

- *Symbole* sind konstante Bezeichner, die nicht vorab deklariert werden müssen und in jedem Fall eindeutig sind.
- Symbolbezeichner beginnen immer mit einem Doppelpunkt.
- Ihnen kann anders als Variablen/Konstanten kein Wert zugewiesen werden.
- Sie werden häufig als Schlüssel in Hashes verwendet:

```
mein_song = {
      :titel => 'History',
      :interpret => 'The Riffles',
      :länge => 3.23,
      :genre => 'BritPop'
}

 mein_song[:genre] # => "BritPop"

 mein_song[:länge] # => 3.23
```

2.4 Symbole

```
mein_song = {
    titel:          'History',
    interpret:   'The Riffles',
    länge:        3.23,
    genre:        'BritPop'
}
```

Auch bei der Verwendung dieser Kurzform erfolgt der Zugriff auf Einträge durch hash[:symbol] und nicht durch hash[symbol:].

Vorsicht:
Die Verwendung von Strings anstelle von Symbolen als Schlüssel kann zu Problemen führen!

2.5 Kontrollstrukturen

- Wie in anderen Programmiersprachen gehören bedingte Anweisungen (IF) und Schleifen (LOOPS) zu den wichtigsten Kontrollstrukturen.

- Anders als in JAVA, C, ... müssen Anweisungsblöcke nicht geklammert, Zeilenenden nicht mit ‘;‘ markiert werden.

- Das Ende eines Anweisungsblocks wird durch END markiert.

- Da die meisten RUBY-Formen einen Wert zurückgeben, können sie als Bedingungen verwendet werden.

- Wenn der Anweisungsblock einer IF- oder einer WHILE-Form nur aus einer Anweisung besteht, können IF/WHILE als Anweisungsmodifikatoren verwendet werden.

2.5 Kontrollstrukturen

```
if count > 10
  puts "Try again"
elsif tries == 3
  puts "You lose"
else
  puts "Enter a number"
end
```

```
while weight < 100 and num_pallets <= 30
  pallet = next_pallet()
  weight += pallet.weight
  num_pallets += 1
end
```

```
while line = gets
  puts line.downcase
end
```

2.5 Kontrollstrukturen

IF/WHILE als Anweisungsmodifikatoren

```
if strahlung > 3000
  puts "Gefährliche Strahlung!"
end
```

```
puts "Gefährliche Strahlung!" if strahlung > 3000
```

```
square = 2
while square < 1000
  square = square*square
end
```

```
square = 2
square = square * square while square < 1000
```

2.6 Reguläre Ausdrücke

- Reguläre Ausdrücke können als kompakte Bezeichnungen von Mengen von Zeichenketten aufgefasst werden.

- In RUBY werden reguläre Ausdrücke durch eine zwischen ein Paar von ‚/' (*slashes*) gesetzte Musterspezifikation gebildet: `/muster/`

Die zur Bildung von regulären Ausdrücken verwendete Syntax orientiert sich an den aus anderen Skriptsprachen vertrauten Konventionen:

```
a|b  # a oder b                          \s  # ein Whitespace-Zeichen
a*   # 0, 1, 2, ... Vorkommen von a      \d  # ein Zahlzeichen (0, ..., 9)
a+   # 1, 2, ... Vorkommen von a         .   # ein beliebiges Zeichen
```

- Der Matchoperator (`=~`) kann verwendet werden um zu überprüfen, ob sich in einer Zeichenkette ein durch einen regulären Ausdruck spezifiziertes Muster findet oder nicht. Es wird entweder die Startposition des gefundenen Ausdrucks oder `nil` zurückgegeben.

- Die Ersetzung von Teilstrings in einem String kann durch `sub` (`gsub`) plus regulärem Ausdruck vorgenommen werden.

2.6 Reguläre Ausdrücke

```
/\d\d:\d\d:\d\d/          # eine Zeitangabe wie 12:34:56
/Perl.*Python/            # Perl, beliebig viele andere Zeichen
                          # gefolgt von Python
/Perl Python/             # Perl, ein Leerzeichen, Python
/Perl *Python/            # Perl, beliebig viele Leerzeichen, Python
/Perl +Python/            # Perl, ein oder mehrere Leerzeichen, Python
/Perl\s+Python/           # Perl, ein oder mehrere whitespace-Zeichen, dann Python
/Ruby (Perl|Python)/      # Ruby, ein Leerzeichen gefolgt von Perl oder Python
```

```
if line =~ /Perl|Python/
  puts "Scripting language mentioned: #{line}"
end

line.sub(/Perl/, 'Ruby')          # ersetze das erste 'Perl' durch 'Ruby'
line.gsub(/Python/, 'Ruby')       # ersetze jedes 'Python' durch 'Ruby'
line.gsub(/Perl|Python/, 'Ruby') # ersetze jedes 'Perl' und 'Python'
```

2.7 Blöcke und Iteratoren

- Anweisungsblöcke können ähnlich wie Parameter an Methoden übergeben oder mit ihnen assoziiert werden.

- Ein Anweisungsblock besteht (in diesem Kontext) aus einer Folge von Anweisungen, die durch geschweifte Klammern { ... } oder do ... end begrenzt werden.

- Für einzeilige Blöcke werden meistens Klammern, für mehrzeilige dagegen do/end verwendet.

- Ein Block wird mit einer Methode assoziiert, indem er beim Aufruf der Methode dem Methodennamen und Parametern der Methode nachgestellt wird:

  methode argument* block

- Eine Methode kann einen Block ein- oder mehrmals durch eine yield-Anweisung ausführen.

2.7 Blöcke und Iteratoren

```
def call_block
  puts "Start der Methode"
  yield
  yield
  puts "Ende der Methode"
end

call_block { puts "Im Block" }
=>
  Start der Methode
  Im Block                # 1. yield
  Im Block                # 2. yield
  Ende der Methode
```

Der mit einem Methodenaufruf assoziierte Block kann prinzipiell beliebig komplex sein.

Es ist möglich durch yield auch Argumente an den Block zu übergeben:

```
def summe
  yield(3, 4)
  yield(-11, 2)
end

summe {|z1, z2| puts "Die Summe von #{z1} und #{z2}
ist #{z1 + z2}."}
# => Die Summe von 3 und 4 ist 7.
# => Die Summe von -11 und 2 ist -9.
```

2.7 Blöcke und Iteratoren

Iteratoren

```
autos = %w(bwm vw toyota mercedes opel )        # Erzeuge ein Array
autos.each {|autol puts auto}                   # Iteriere über die Elemente
# =>
bmw
vw
toyota
merceds
opel
```

```
[ 'cat', 'dog', 'horse' ].each {|namel print name, " " }
5.times {  print "*" }
3.upto(6) {|il  print i }
('a'..'e').each {|charl print char }
# =>
cat dog horse *****3456abcde
```

2.8 Diverse Kleinigkeiten

Variables				Constants and
Local	**Global**	**Instance**	**Class**	**Class Names**
name	$debug	@name	@@total	PI
fish_and_chips	$CUSTOMER	@point_1	@@symtab	FeetPerMile
x_axis	$_	@X	@@N	String
thx1138	$plan9	@_	@@x_pos	MyClass
_26	$Global	@plan9	@@SINGLE	JazzSong

✦ RUBY kennt viele Ein- und Ausgabemethoden: p, puts, gets, printf, ...

```
printf("Number: %5.2f,\nString: %s\n", 1.23, "hello")
# => Number:   1.23,
# => String: hello

while line = gets
  print line
end
```

2.9 Kommandozeilenargumente

Bei der Ausführung eines Ruby-Programms von der Kommandozeile aus, ist es möglich, Argumente an das aufgerufene Programm zu übergeben. Dazu gibt es zwei Möglichkeiten:

- ARGV bezeichnet den Array, in dem Ruby alle Argumente speichert, die beim Aufruf des Programms mit angegeben wurden. Jedes Programm kann auf die in diesem Array gespeicherten Objekte zugreifen.

```
puts “Die Datei erhielt #{ARGV.size} Argumente:
puts ARGV
```

test.rb

```
$ ruby test.rb Hallo System
Die Datei erhielt 2 Argumente:
Hallo
System
```

- In vielen Fällen handelt es sich bei den Argumenten um die Namen der Dateien, aus denen das Programm lesen bzw. in die das Programm schreiben will. ARGF bezeichnet ein besonderes
IO-Objekt, dass den Inhalt aller durch Kommandozeilenargumente bezeichneter Dateien enthält.