Ruby "basics"

Reservierte Schlüsselwörter / Identifier

- `BEGIN`
- `END`
- `alias`
- `and`
- `begin`
- `break`
- `case`
- `class`
- `def`
- `defined`
- `do`
- `else`
- `elsif`
- `end`
- `ensure`
- `false`
- `for`
- `if`
- `in`
- `module`
- `next`
- `nil`
- `not`
- `or`
- `redo`
- `rescue`
- `retry`
- `return`
- `self`
- `super`
- `then`
- `true`
- `undef`
- `unless`
- `until`
- `when`
- `while`
- `yield`

Namenskonventionen

- **Lokale** Variablen
  - Sie beginnen mit einem Kleinbuchstaben:

```
alpha = 45
_id = "Text"  # '_' wie Kleinbuchstabe!
some_name = "Name1" # underscore-Notation
otherName = "Name2" # CamelCase-Notation
self, nil, __FILE__ # Pseudovariablen!
```

- **Globale** Variablen
  - Sie beginnen mit einen $-Zeichen:

```
$stdout
$NOT_CONST
```

Namenskonventionen

- Objekt-Variablen (**Attribute**)
  - beginnen mit einem @-Zeichen:

```
@attr1
@NOT_CONST
```

- Klassen-Variablen (**Klassenattribute**)
  - beginnen mit zwei @-Zeichen:

```
@@class_attr
@@NOT_CONST
```

- **Konstanten**
  - beginnen mit einem Großbuchstaben

```
K6chip
Laenge
```

Anweisungen, Zuweisungen, Ausdrücke

- Vertraute Notation
  - Zuweisungen (*assignments*) in "C" und Ruby sind ähnlich:

```
a = true
b = x + 5
c = (a == true) ? "Wahr" : "Falsch"
d = meine_methode( b )
a = b = c = 0          # Mehrfach-Zuweisung
a, b = b, a            # Vertauschen ohne Hilfsvariable
```

- **Ausdrücke - typisch für Ruby:**
  - Anweisungen (*statements*) wie auch Zuweisungen sind fast immer Ausdrücke (*expressions*):

```
x = if a then b = 5 else nil  # x ist 5 oder nil
b                             # Auch ein Ausdruck!
a, b, c = x, (x+=1), (x+=1)   # Kombinierbar
```

Kommentare, eingebettete Dokumentation

- Kommentare im Programmcode

```
x = y + 5  # Dies ist ein Kommentar
# Dies ist eine ganze Kommentarzeile.
puts "# KEIN Kommentar!"
```

- Eingebettete Dokumentation
  - Mehrzeilige Textblöcke können mittels **=begin** und **=end** für den Interpreter ausgeblendet werden:

```
=begin
Dieses Programm ist geschrieben worden,
um ... usw. usw.

Version 1.0  YYYY--MM--DD  Autor
=end
```

Konstanten

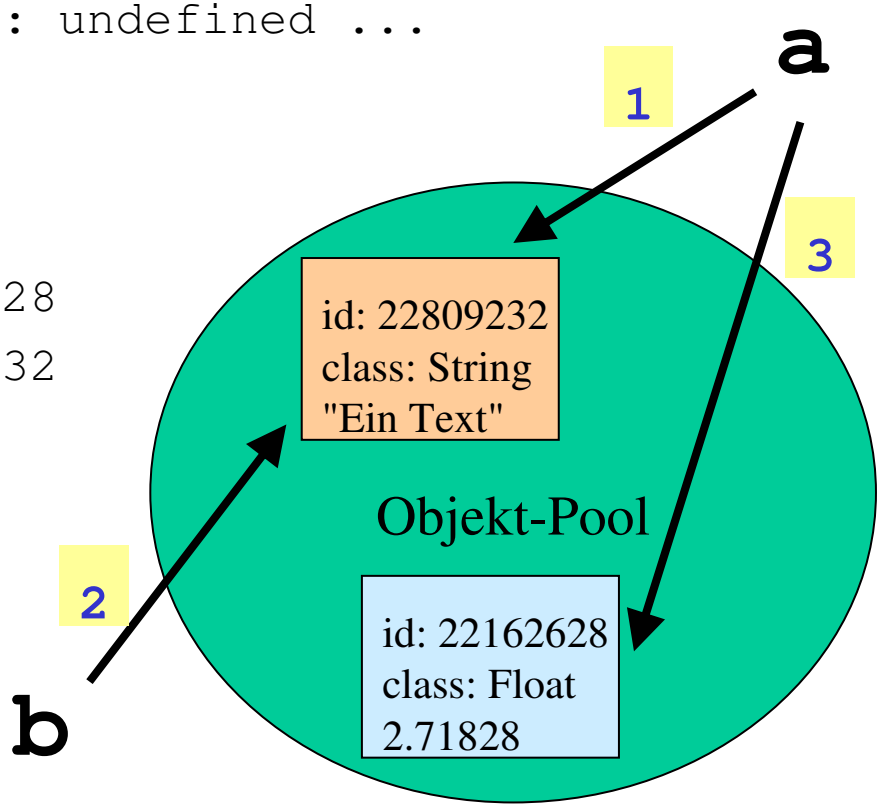
- Beispiele für numerische Konstanten:

```
123             # Integer
-124            # Integer, mit Vorzeichen
1_234_567       # Kurios: Auch zulässig für Integer!
0377            # Oktalzahl - Führende Null
0xBEEF          # Hexadezimalzahl
0xabef          # auch hex-Zahl
0b1010          # Dualzahl
3.14159         # Fließkommazahl
6.023e23        # Fließkommazahl, wissensch. Notation
Math::PI        # 3.14159265358979, aus Modul "Math"
```

Variablen vs. Objekte

- Objekte sind "typisiert" - durch ihre Klassen & Meth.
- Variablen sind nur Verweise auf Objekte:

```
1  a = "Ein Text"
   a.class        # --> String
   a.object_id    # --> 22809232
   b.class        # Name Error: undefined ...
   b = a          # Übernahme
2  a = 2.71828 # Neubelegung
3  a.class        # --> Float
   a.object_id    # --> 22162628
   b.object_id    # --> 22809232
```



Variablen vs. Objekte

- Analogie zum Unix *file system:*
  - Dateien – *inodes* - Objekte
  - Dateinamen - *links* - Variablen

```
Unix file system          Ruby objects
==============================================
# 2 Verweise auf gleichen inode / gleiches Objekt:

$ ln c d                  d = c

# 2 unterschiedliche inodes / Objekte entstehen:

$ cp a b                  b = a.dup
```

Variablen vs. Objekte

- Objekte
  - werden über interne Strukturen verwaltet und per ID referenziert.
  - benötigen / erhalten Speicherplatz (auf dem *heap*)
  - werden von Ruby automatisch verwaltet

    ```
    Allokation von Speicher bei Neuanlage oder Wachstum
    Freigabe wenn nicht mehr benötigt
    ```

  - existieren, solange sie in Benutzung sind, d.h. referenziert werden

    ```
    Beispiel: Variable verweist auf dieses Objekt
    ```

- ***Garbage collection***
  - Ruby erkennt Objekte, die nicht mehr referenziert werden, und entfernt sie dann automatisch aus dem Objekt-Pool.
  - Destruktor-Methoden werden daher nicht benötigt!
  - GC-Methode: *mark-and-sweep*

    ```
    Mit Methoden der Klasse "GC" kann man zur Not eingreifen
    ```

  - Wie performant? I.a. überwiegt der Nutzen erheblich: Effizienteres Coding, weniger Fehler, kein Ärger mit *memory leaks*, ...

Variablen vs. Objekte

- Gültigkeitsbereich (*scope*) von Variablen
  - Meist der Code-Block, in dem sie angelegt werden.

```
def my_method          # Methode anlegen
  a = 1.5              # Scope: method body
  begin                # Code block
    b = a * 2          # Scope: begin..end block
    b.class            # Float
    a += 1             # 2.5
    $c = a + 1         # Scope: Globale Variable!
  end                  # Ende Code block
  $c.class             # Float
  $c                   # 3.5
  b.class              # Name error: undefined ...
  a.class              # Float
  a                    # 2.5
end                    # Ende Methoden-Definition
```

Klärung eines irb-Befundes

- NilClass oder Fehlermeldung
  - Befund, z.B. in irb:

```
a.class       --> Fehlermeldung, aber
$a.class      --> NilClass. Warum?
```

  - Der Wert einer neuen Variablen ist für Ruby zunächst "nil", d.h. das (einzige) Objekt von NilClass.
  - **$a** ist immer eine Variable, **a** kann aber sowohl eine Variable als auch eine Methode sein.
  - Daher: Fehlermeldung bei Verwechslungsgefahr!

Datentypen?

- Der Begriff "Datentyp" ist in Ruby irreführend:
  - Anstelle der klassischen einfachen "Datentypen" für Zahlen, Zeichen und Zeichenketten (*strings*) besitzt Ruby Klassen.
  - Selbst Konstanten sind Objekte dieser Klassen!
- Einige Standardklassen
  - Fixnum

    Entspricht allen Integer-Typen, also etwa

    char, short, int (signed wie unsigned)
  - Bignum

    Setzt Fixnum fort wenn erforderlich

```
a = 2**29           # 536870912
a.class             # Fixnum
a *= 2; a.class     # Bignum
```

    Viel langsamer, aber mit beliebiger Genauigkeit!
  - Float

    Entspricht „double“ (gemäß der jeweiligen Hardware).

Zeichen

- Grundregel:
  - Zeichen in Ruby sind stets Integer (also: **`Fixnum`**)!
  - Spezielle Notationen für den Umgang mit Zeichen:

```
?x              # 120 (ASCII-Code des kleinen x)
?\n             # 10, das Newline-Zeichen
?\\             # 92, das Backslash-Zeichen
?\cd            # 4, Control-D
?\C-x           # 24, Control-X, alternative Notation
?\M-x           # 248, Meta-x (x OR 0x80)
?\M-\C-x        # 152, Meta-Control-x (24 + 128)
```

  - Umgang mit dieser Notation (Beispiele):

```
a = ?8 + ?\n - ?\C-A
b = ?A; a == b  # true (# 56 + 10 - 1 = 65)
```

  - All diese Zeichen lassen sich auch in Strings einbetten!

Einige wichtige Klassen

Array, Hash, String, Range

Zeichenketten (Strings)

Umgang mit Zeichenketten ist eine der wichtigsten Aufgaben jeder Skriptsprache.

- Ruby besitzt dazu eine Fülle an Möglichkeiten!
  - Sie werden insb. in Klasse "`String`" bereitgestellt
- Wer von C/C++ kommt,
  - findet hier eine Vielfalt neuer Konzepte und Ideen (was anfangs leicht verwirren kann).
- Wer Perl kennt,
  - findet viele "alte Bekannte" wieder
  - erhält Gelegenheit, Perls manchmal kryptische Notationen durch lesbarere Varianten zu ersetzen.
- Strings in Python:
  - Viele Ähnlichkeiten zu Ruby, insb. bei den Methoden
  - Aber: Python-Strings sind nicht veränderbar, Ruby-Strings wohl

Zeichenketten (Strings)

- Basisnotationen für Strings:
  - Hochkommata vs. Anführungszeichen als Begrenzer

```
s1 = "Ein String"                          # ok
s2 = 'Noch ein String'                     # auch ok
s3 = "Zeile 1\n\t#{s1}\n"; print s3
Zeile 1
        Ein String
s4 = 'Zeile 1\n\t#{s1}\n'; print s4
Zeile 1\n\t#{s1}\n
```

  - In " ... " werden \-Notationen wie \n und Ausdrücke der Art #{...} ausgewertet, in '...' nicht, analog zur Unix-Shell!

Nicht in Python

  - Ausnahme: \' wird auch in '...' beachtet:

```
'van\'t Hoff' --> "van't Hoff"
```

Unicode-Strings

- Ruby
  - Bis Ruby 1.8: String = Folge von Bytes, Hilfsbibliotheken für mehr
  - Ab Ruby 1.9: String = Folge von Zeichen, Zeichen != Byte; Unterstützung zahlreicher Zeichensatz-Codierungen incl. Unicode
  - Beispiel, Ruby 1.8:

```
$KCODE = "u"  # Unicode UTF-8
require "jcode"
s = "English Deutsch 日本語"   # Editor: UTF-8-Codierung
s.length      # 25
s.jlength     # 19
```

  - Beispiel, Ruby 1.9:

```
# -*- coding: utf-8 -*-
s = "English Deutsch 日本語"
s.size        # 19  (size und length sind Synonyme)
s.bytesize    # 25
s.encoding    # <Encoding: UTF-8>
```

Unicode-Strings

- Python
  - Unicode-Unterstützung ist seit Python 2.0 eingebaut
  - Python unterscheidet normale von Unicode-Strings
  - Beispiel, Python 2.5:

```
# -*- coding: utf-8 -*-
s = u"English Deutsch 日本語"
len(s)        # 19  (noch zu prüfen)
```

- Einbettung von Unicode-Zeichen
  - Python

```
u"F\u00FCnf \u20AC"          # Fünf €
u"F\u00fcnf \N{EURO SIGN}" # Fünf €
```

  - Ruby 1.9

```
"F\u00FCnf \u20AC"          # Fünf €
```

Zeichenketten: "*Here documents*"

- *Here documents*:
  - eine bequeme Möglichkeit, mehrzeilige Texte innerhalb des Programmcodes anzulegen:
  - Beispiel 1 (Ruby):

```
a = 123
print <<HERE   # Identifier
Double quoted \
here document.
Sum = #{a+1}
HERE
```

  - ergibt:

```
Double quoted here
document.
Sum = 124
```

- *Here documents*:
  - Python verwendet *triple quotes* als Begrenzer (""" oder '''). Diese werden auch oft als *documentation strings* verwendet.
  - Beispiel:

```
print """
usage: myprog [options]
  -h      Display this text
  -u url Check presence …
"""
```

  - ergibt:

```
usage: myprog [options]
  -h      Display this text
  -u url Check presence …
```

Zeichenketten: "*Here documents*"

- *Here documents* (Forts.):
  - eine bequeme Möglichkeit, mehrzeilige Texte innerhalb des Programmcodes anzulegen:
  - Beispiel 1:

```
a = 123
print <<HERE  # Identifier
Double quoted \
here document.
Sum = #{a+1}
HERE
```

  - ergibt:

```
Double quoted here document.
Sum = 124
```

Zeichenketten: "*Here documents*"

- *Here documents* (Forts.):
  - Beispiel 2:

```
a = 123
print <<-'THERE'     # String, single quoted
      This is single quoted.
      The above used #{a+1}
      THERE
```

  - ergibt:

```
      This is single quoted.
      The above used #{a+1}
```

  - Beachten:

    <<- (Minuszeichen beachten!)

    Ende-Sequenz darf eingerückt erscheinen.

    single / double quotation rules:

    Ableitung von der Notation des verwendeten Begrenzer-Strings!

Zeichenketten: Sonstiges

- Konkatenierung (Verkettung):

```
a = "abc" 'def' "g\n"       # Stringlisten
"abcdefg\n"
a << "hij"                  # Operator <<
"abcdefg\nhij"
a += "klm"                  # Operator +
"abcdefg\nhijklm"
```

- Interne Darstellung von Zeichenketten:
  - Sequenzen von 8-bit Bytes
  - Jedes Zeichen kann alle 256 Zustände annehmen;
    Wert 0 hat keine Sonderstellung (anders als in C)
  - Wesentliche Änderung wegen Unicode-Unterstützung ab Ruby 1.9!!

Zeichenketten (Strings)

- Generalisierte Notation:
  - Wie vermeidet man das "Escaping"? Wir definieren uns einen Begrenzer, der nicht im Nutztext steht!

```
# Seien  | bzw. / zwei neue Begrenzer:
s1 = %q|"Mach's richtig!", sagte der Chef.|
s2 = %Q/"Mach's richtig!", sagte #{name}./
```

  - %q|...| entspricht '...', %Q/.../ entspricht "...".
  - 0-9, A-Z, a-z sind offenbar keine zulässigen Begrenzer in Ruby 1.8
  - Sonderregeln für Klammern:

```
# Klammer-Zeichen werden gepaart!
s1 = %Q(Ein String)
s2 = %q[Noch ein String]
s3 = %Q{Zeile 1\n\t#{s1}\n}
s4 = %q<Zeile 1\n\t#{s1}\n>
```

Zeichenketten (Strings)

- Weitere Anwendungen der generalisierten Notation:
  - **%x** - Command output string ("back tick"-Notation)

```
# Ausführen folgender Shell-Kommandos:
`whoami`      # --> "user123\n"
`ls -l`       # --> Mehrzeiliger String
%x[ps -ef | grep acroread | wc -l]
```

  - %x[...] entspricht hier `...`, Klammerregel inklusive.

  - **%w** - Array aus Strings:

```
# Array von Strings - viel zu tippen!
a = ["Jan", "Feb", "Mär", "Apr", "Mai", "Jun", "Jul",
     "Aug", "Sep", "Okt", "Nov", "Dez"]
# Dasselbe, nur kürzer:
b = %w(Jan Feb Mär Apr Mai Jun Jul Aug Sep Okt Nov Dez)
a == b        # true
```

  - %w(...) entspricht hier ["...", ..., "..."], Klammerregel inklusive

Zeichenketten (Strings)

- **%r** - Reguläre Ausdrücke:

```
s =~ /Ruby/        # true, wenn s 'Ruby' enthält
s =~ /[rR]uby/   # true, wenn 'Ruby' oder 'ruby'
s =~ %r([rR]uby)# gleicher Fall
s =~ %r|[0-9]+| # beliebige Ziffernfolge
```

- Reguläre Ausdrücke ("regex"):
  - zum *pattern matching* (Erkennung von Textmustern)
  - gibt es schon seit Jahrzehnten
  - wurden dank Perl in den letzten ~10 Jahren populär
  - Ein ebenso mächtiges wie oftmals kryptisches Werkzeug
  - Absolut unverzichtbar für Informatiker!
  - Wir werden reguläre Ausdrücke im Vertiefungsteil näher behandeln. Hier nur einige Bespiele:

Exkurs: Beispiele für Reguläre Ausdrücke

```
/^Ruby/                    # Fängt Text mit "Ruby" an?
/Ende$/                    # Hört Text mit "Ende" auf?
/Ende\.$/                  # Hört Text mit "Ende." auf?
/[+-]?[0-9A-Fa-f]+/ # Hex-Zahl (Integer)?
# Parsen von Key/value-Paaren, klassisch:
   line =~ /\s*(\w+)\s*=\s*(.*?)$/
# Beachte nun $1, $2.
# Nun Ruby-style:
   pat = /\s*(\w+)\s*=\s*(.*?)$/
   matches = pat.match(line) # MatchData-Objekt
```

– Beispiele dazu (an der Tafel zu diskutieren):

```
1)  line = "a=b+c"
2)  line = "  a = b + c    "
3)  line = "Farbe = blau  # Kommentar"
4)  line = "x = puts'# Kommentar'"
```

Arrays

- Vorbemerkungen:
  - In C / C++ sind Arrays maschinennah als Speicherblöcke definiert:

```
void f()
{       // Statisch angelegte Arrays
  char buf[1024];    // 1024 * sizeof(char) Bytes
  int  primes[100]; // 100  * sizeof(int)  Bytes
  my_struct *p;
...
  p = malloc(len * sizeof(my_struct)) // Zur Laufzeit
...
  free(p)
}
```

  - Speicherverwaltung und Anpassung an Datentypen:
    Bleibt den Entwicklern überlassen.
  - Anpassungen zur Laufzeit?
    Sind Arrays einmal angelegt, lässt sich ihre Größe nicht verändern. "Workarounds" sind mit Mühe möglich.
  - Arrays aus verschiedenen Datentypen / Objekten unterschiedlicher Klassen ("Container")?
    In C nicht vorgesehen, keine leichte Übung in C++

Arrays

- Die Ruby-Klasse **`Array`**:
  - bündelt zahlreiche Eigenschaften von klassischen Arrays und Containerklassen
  - stellt viele Methoden für die bequeme Benutzung bereit
  - kümmert sich komplett um die Speicherverwaltung
  - verwaltet beliebig viele Objekte, auch aus verschiedenen Klassen
  - ist konzeptionell verwandt mit Perl-"Arrays"

- Tipp:
  - Denken Sie bei Ruby-Arrays NICHT an gleichnamige Dinge aus C/C++
  - Eine abstrakte, eher mathematische Sicht ist hilfreicher (etwa: n-Tupel, nicht: Menge)

- Python: **`list`**
  - Ferner: **`tuple`** (nicht veränderbar), **`set`**, **`frozenset`**

Arrays: Elementare Beispiele

- Beispiele:

```
[1, 2, 3]              # Array aus drei Fixnum-Objekten
[1, 2.0, "drei"]       # Mischfall: Fixnum,Float,String
[1, [2, 3], 4]         # Arrays in Array - na klar!
["eins", "zwei", "drei"] # 3 Strings
%w<eins zwei drei>     # dasselbe, nur kürzer
```

- ... für Zuweisungen:

```
a = []                 # Leeres Array
a = Array.new          # ebenfalls
a = [1, 2, 3]          # Vorbelegung mit Konstanten...
b = ["string", a]      # ... und auch mit Variablen
                       # --> ["string", [1, 2, 3]]
```

Arrays: Elementare Beispiele

- ... für Zugriff per Index:

```
x = a[0] + a[2]      # --> 4 (Index läuft ab 0)
a[1] = a[2] - 1      # Auch Zuweisungen, wie gewohnt.
```

- Neu (kleine Auswahl):

```
a[-1]          # --> 3 (neg. Index zählt vom Ende)
a[3, 2]        # Teil-Array: 2 Elemente, ab a[3]
a[1..4]        # ... und per "Bereich" adressiert
```

- **Ruby vs. Perl**
  - Perl führt Arrays als spezielle Datentypen, nicht als Objekte.
  - Die Notation ist leicht unterschiedlich:

```
@a = (1, 2, 3, 4, 5); # Perl
a[4]           # --> 5
a[8] = 7       # Dynamisches Hinzufügen - wie in Ruby
```

Exkurs / Vorgriff: Klassen und Methoden

- Eine Klasse anlegen – ganz einfach

```
class SayHi
end
```

```
c = SayHi.new          # Ein Exemplar anlegen
c.class                # "SayHi"
```

- Eine Methode anfügen

```
class SayHi
  def greet( name )
    puts "Hi, #{name}!"
  end
end
```

```
c.greet "folks"                # "Hi, folks!"
```

Mengen (sets)

- Ruby unterstützt Mengen nicht direkt (wie etwa Pascal oder Python)
  - Z.B. gibt es keinen Operator "in"
- Mengen lassen sich aber einfach auf der Basis der Array-Klasse aufbauen
- Benötigt:
  - Elemente dürfen nicht mehrfach vorkommen
  - Ihre Reihenfolge spielt keine Rolle
  - Ersatz für Operator "in"
  - Mengenoperationen: Vereinigung, Durchschnitt, Differenz
- Einzelheiten: The Ruby Way, p.148ff
- Siehe auch: Ruby Standard Library „Set“

Hashes

- Vorbemerkungen:
  - Hashes sind die vielleicht wichtigsten Datentypen in Skriptsprachen.
  - Viele Entwicker wechselten zu Skriptsprachen wegen Regulärer Ausdrücke - und wegen Hashes!
  - Synonyme: Assoziative Arrays, Maps, Dictionaries
  - Sie sind i.w. Mengen aus (*key, value*)-Paaren, wobei die "Schlüssel" eindeutig vergeben werden.
  - Alternative Sichtweise: Arrays mit generalisierten Indizes
  - (Bemerkungen zur Herkunft des Namens "hash")

- Die Ruby-Klasse **`Hash`**
  - Sie bietet zahlreiche Methoden zum bequemen Umgang mit Hashes (und davon ableitbaren Klassen).
  - Viele Parallelen zur Klasse **`Array`**
  - Als "keys" können Objekte beliebiger Klassen dienen. Bedingungen:
    (1) sie haben eine Methode "hash" implementiert,
    (2) der hash-Wert eines Objekts bleibt konstant.

Hashes: Elementare Beispiele

- Beispiele:

```
# Lookup-Tabelle für Quadratzahlen
q = {1=>1, 2=>4, 3=>9, 4=>16, 5=>25}
q[4]          # --> 16
q = {"eins"=>1, "zwei"=>4, "drei"=>9, "vier"=>16, }
q["vier"]     # --> 16
# Dynamisch hinzufügen
q["acht"]     # --> nil
q["acht"] = 64
q["acht"]     # --> 64
```

Zulässig, wird ignoriert

- **Ruby vs. Perl:**
  - Perl führt Hashes als spezielle Datentypen, nicht als Objekte.
  - Die Notation ist leicht unterschiedlich:

```
%q = (1=>1, 2=>4, 3=>9, 4=>16, 5=>25);   # Perl
q{4}          # --> 16
q{8} = 64     # Dynamisches Hinzufügen - ganz analog!
```

Hashes: Elementare Beispiele

- **Ruby vs. Python:**
  - Python kennt den Datentypen **`dict`**.
  - Die Notation ist wie bei Ruby, aber kompakter:

```
d = {1:1, 2:4, 3:9, 4:16, 5:25}   # Python
d[4]              # --> 16
d[8] = 64         # Dynamisches Hinzufügen - ganz analog!
```

- **Ruby-Hashes und Symbole:**
  - Hash keys in Ruby sind häufig Symbole: Sie sind selbst-dokumentierend, aber sparsamer als Strings:

```
wday_map1 = {"Mon" => "Monday", "Tue" => "Tuesday"}
wday_map2 = {:Mon => "Monday", :Tue => "Tuesday"}
```

  - Ab Ruby 1.9 gibt es eine noch besser lesbare Kurzform, falls die Keys Symbole sind:

```
wday_map3 = {Mon: "Monday", Tue: "Tuesday"}
# äquivalent zu wday_map2 !
```

Bereiche (*ranges*)

- Ruby behandelt Bereiche als eigene Klasse "Range"
  - Andere Sprachen behandeln sie als Listen oder als Mengen
- als Folgen (*sequences*)

```
1..10                          # ..  - Mit Endpunkt
'a'..'z'                       # auch für Strings
0...my_array.length            # ... - Ohne Endpunkt
testbereich = 5..20
testbereich.each { .... }      # Einfache Schleife
testbereich.to_a               # [5, 6, 7, ..., 20]
```

- in Bedingungen

```
while line=gets                # Ausgabe beginnt mit /start/
  print "Found: "+line if line=~/start/ .. line=~/end/
end                            # und stoppt wieder bei /end/
```

- als Intervalle

```
(1..10) === Math::PI           # true: 1 <= Pi < 10
('a'..'p') === 'z'             # false
```

Muster (patterns)

- Teil von Klasse Regex (für Reguläre Ausdrücke)!
  - Werden manchmal als eigene Datentypen behandelt.
  - Ruby tut dies nicht - es gibt keine eingebaute Klasse für *patterns*.
- Wir besprechen *patterns* zusammen mit Regex im Vertiefungsteil.

Namen, Symbole
Operatoren

Namen und Symbole

- Variablen, Konstanten, Methoden, Klassen, ... - Für alle diese Einheiten vergeben wir **Namen**:

```
local_var, $global_var, Math::PI
@instance_var, @@class_var
my_method, MyClass, MyModule
```

- Wenn man die mit einem bestimmten Namen identifizierte Einheit selbst als Argument behandeln will, verwendet man **Symbole** (Exemplare der Klasse **`Symbol`**):

```
:my_method, :instance_var
```

  - Durch Voranstellen eines ':' wird aus dem Namen "sein" Symbol.

- Umwandlungen:

```
my_var = "abc"
my_var.class            # String
my_sym = :my_var
my_sym.class            # Symbol
my_sym.id2name # "my_var"
```

Symbole

- Ruby besitzt die Klasse **Symbol**
  - Symbole sind „identifizierende, konstante Strings“
  - Nutzen Sie Symbole anstelle von Strings, wenn Sie Dinge nur per Name ansprechen wollen und auf die Veränderbarkeit von Strings keinen Wert legen:

```
conn = MyHttpConnector
                :host => 'www.fh-wiesbaden.de',
                :port => 80,
                :path => '/index.html,
result = conn.get    # usw.
```

  - Symbole werden prozessweit nur einmal angelegt, während String-Literale bei jedem Aufruf ein neues (teures) String-Objekt erzeugen
  - Umwandlungen sind einfach möglich:

```
:host.class   # Symbol
:host.to_s    # "host"      alternativ: :host.id2name
"host".to_sym # :host       alternativ: "host".intern
```

Operatoren

- Bindungsstärke der Operatoren (absteigende Reihenfolge)
  - Scope

```
::              # Math::PI
```

  - Index

```
[ ]             # a[3]
```

  - Potenzierung

```
**              # 2**5
```

  - Unitäre (Vz etc.)

```
+  -  !  ~      # Positiv, negativ, nicht, Bit-Kompl.
```

  - Punktrechnung

```
*  /  %         # Mal, geteilt, modulo (Rest)
```

  - Strichrechnung

```
+  -            # a + b, c - d
```

  - Bitweise Rechts- und Linksverschiebung (und abgeleitete Op.)

```
<<  >>          # 3 << 4 == 48
```

Operatoren

- Bindungsstärke der Operatoren (Forts.)
  - Bitweises und

```
&                 # 6 & 5 == 4  # true
```

  - Bitweises oder, exklusiv oder

```
|  ^              # 6 | 5 == 7; 6 ^ 5 == 3
```

  - Vergleiche

```
>  >=  <  <=      # 3 < 4
```

  - Gleichheiten

```
==  !=            # ist gleich/ungleich
=~  !~            # my_string =~ some_pattern
=== <=>           # Relations- und Vergleichsoperator
```

  - Boole'sches und

```
&&                # a && b
```

  - Boole'sches oder

```
||                # a || b
```

Operatoren

- Bindungsstärke der Operatoren (Forts.)
  - Bereichs-Operatoren

```
..   ...      # 1..10  a..z  nicht_volljaehrig = 0...18
```

  - Zuweisung

```
=             # Implizit auch += -= *= etc.
```

  - "3-Wege-Bedingung" (*ternary if*)

```
?:            # a = x < y ? 5 : -5
```

  - Boole'sche Negation

```
not           # b = not a  # Bindet schwächer als !
```

  - Boole'sches Und, Oder

```
and  or       # c = a or b # Binden schwächer als && ||
```

- Bemerkungen
  - Viele Operatoren dienen mehreren Zwecken, etwa + und <<
  - Die meisten **Operatoren sind Kurzformen von Methoden**!

```
x = 4 + 3  entspricht: x = 4.+(3)  # Methode '+' !
```

  - Diese können daher überladen / umdefiniert werden.

Umdefinieren von Operatoren

- Ein (etwas abwegiges) Beispiel mit Addition und Subtraktion:

```
a = 4 - 3                # 1
b = 4 + 3                # 7
b += 2                   # 9   Bisher ist alles normal.
```

```
class Fixnum             # Standardklasse modifizieren!
  alias plus  +          # alias: Synonyme für Methoden
  alias minus -          # Alte Ops "merken".
  def +(op)              # "+" überladen
    minus op             # auch: self.minus(op)
  end
  def -(op)              # "-" überladen
    plus op              # auch: self.plus(op)
  end
end
```

```
a = 4 - 3                # 7  !!  aber:   4.minus 3 == 1
b = 4 + 3                # 1  !!  analog: 4.plus 3  == 7
b += 2                   # -1  (+= implizit mitgeändert!)
```

Operatoren vs. Methoden

- Mittels Methoden implementiert:

```
[]   []=
**
!   ~   +   -   # +@,-@
*   /   %
+   -
>> <<
&
^   |
<= < > >=
<=> == === != =~ !~
```

  - rot: nicht überladbar
  - Unterscheide Methoden `+@, -@` von `+, -`

- Nicht überladbare Operatoren:

```
&&
||
..    ...
? :
= %= ~= /= -= +=
   |= &= >>= <<= *=
   &&= ||=
**=
defined?
not
or   and
if unless
   while until
begin / end
```

Ein kleines Beispielprogramm

(noch nicht objektorientiert)

Beispielprogramm: dm_euro.rb

- Umrechnung D-Mark ←→Euro
  - nach Hal Fulton's Beispiel "Fahrenheit-Celsius"
  - **Demo + Erläuterungen in der Vorlesung**
  - Quellcode auf dem Fileserver unter Aufgabe 1

- Bemerkungen (Stichworte)
  - Objekte sind implizit verwendet
  - Der leere String "" ergibt in Vergleichen nicht `false`!
  - `chomp!` erläutern: `chomp` vs. `chop`, Konvention zu "!"
  - Mehrfachzuweisung und Arrays, Methode "`split`"
  - Beispiel für einen regulären Ausdruck
  - `case`-Statement: Sehr mächtig in Ruby, kein *drop-through*
  - Keine Variablen-Deklarationen

```
b_euro != nil testet zur Laufzeit, welche Variable
  angelegt wurde / welcher Fall vorliegt.
```

Verzweigungen und Schleifen

Bedingte Verarbeitung - "if" und "unless"

```
if x < 5 then
      statement1
end
```

```
unless x >= 5 then
      statement1
end
```

```
if x < 5 then
      statement1
else
      statement2
end
```

```
unless x < 5 then
      statement2
else
      statement1
end
```

```
statement1 if y == 3
```

```
statement1 unless y != 3
```

```
x = if a>0 then b \
      else c end
```

```
x = unless a<=0 then b \
      else c end
```

```
if x < 3 then b elsif \
   x > 5 then c \
      else d end
```

Bem.:
then darf fehlen, wenn die Zeile ohnehin endet (hier die blau markierten Fälle).

Bedingte Verarbeitung: "case" und "when"

- Testbeispiel:

```
case "Dies ist eine
      Zeichenkette."
  when "ein Wert"
    puts "Zweig 1"
  when "anderer Wert"
    puts "Zweig 2"
  when /Zeichen/
    puts "Zweig 3"
  when "ist", "ein"
    puts "Zweig 4"
  else # optional
    puts "Zweig 5"
end
```

- Was wird ausgegeben?
  - "Zweig 3" ! Warum??

- Wie funktioniert's ?
  - Ruby verwendet hier den Operator ===. Je nach Objekt ist === anders definiert.
  - === ist *nicht* kommutativ !
  - Bequem: Auch Regex zulässig
  - "when" nimmt auch Listen an
  - Überladen von === ändert auch das Verhalten von case ... when (analog "+/+=")
- Der erste passende Zweig wird ausgeführt
  - Kein *drop through*, wie etwa bei switch/case in C/C++ !!
- Perl kennt kein case/when

Variationen zum Thema "Schleifen"

- Vorbereitung für alle Beispiele

```
liste = %w[alpha beta gamma delta epsilon]
```

- **Variationen von „Ausgabe der Liste“:**
  - Version 1 (while)

```
i=0
while i < liste.size do
  puts "#{liste[i]} "
  i += 1
end
```

  - Version 2 (until)

```
i=0
until i == liste.size do # Hier Unterschied
  puts "#{liste[i]} "
  i += 1
end
```

Variationen zum Thema "Schleifen"

– Version 3 (*while* am Ende)

```
i=0
begin
  puts "#{liste[i]} "
  i += 1
end while i < liste.size
```

– Version 4 (*until* am Ende)

```
i=0
begin
  puts "#{liste[i]} "
  i += 1
end until i >= liste.size  # analog
```

Variationen zum Thema "Schleifen"

- Version 5 (*loop*-Form)

```
i=0
n=liste.size-1
loop do                          # loop - eine Methode
  puts "#{liste[i]} "            # aus Klasse 'Kernel'
  i += 1
  break if i > n                 # break - analog zu C
end
```

- Version 6 (*loop*-Form, Variante)

```
i=0
n=liste.size-1
loop do
  puts "#{liste[i]} "
  i += 1
  break unless i <= n            # Hier Unterschied
end
```

Variationen zum Thema "Schleifen"

- Bisherige Nachteile:
  - Beschäftigung mit dem Aufbau der Liste (Index, Größe), obwohl nur "Iterieren" der Liste erforderlich ist; 5..7 Zeilen Code. Daher:
  - Version 7 ('*each*'-Iterator der Klasse **Array**)

```
liste.each do |x|                # Seltsam?
  puts "#{x} "
end
```

  - Version 7a (Kurzform)

```
liste.each { |x| puts "#{x} " }
                                 # Der Ruby-Normalfall!
```

  - Version 8 (*for .. in ..* - verwandelt Ruby in Version 7)

```
for x in liste do                # Etwas "Syntax-Zucker"
  puts "#{x} "                   # für die Perl-Fraktion
end
```

Variationen zum Thema "Schleifen"

- **Variationen zu Iteratoren:**
  - Version 9 ('*times*'-Iterator der Klasse **Fixnum**)

```
n=liste.size
n.times { |i| puts "#{liste[i]} " }
```

  - Version 10 ('*upto*'-Iterator der Klasse **Fixnum**)

```
n=liste.size-1
0.upto(n) { |i| puts "#{liste[i]} " }
```

  - Version 11 (*for und Range,* '*each*'-It. der Klasse **Range**)

```
# Wirkung von ... beachten:
for i in 0...liste.size { |i| puts "#{liste[i]} " }
```

  - Version 12 ('*each_index*'-Iterator der Klasse **Array**)

```
liste.each_index { |i| puts "#{liste[i]} " }
```

- Gemeinsamer Nachteil von Version 9-12:
  - Wieder Umgang mit Indizes notwendig!

Iteratoren und Blöcke - typisch für Ruby!

- Wie funktionieren Iterator-Methoden?
  - Man übergibt der Methode (neben Parametern) auch einen Codeblock:

```
do ... end    # bzw. {...}
```

  - Die Iteratormethode kann diesen Codeblock aufrufen:

```
yield         # Neues Schlüsselwort!
```

  - Optional benennt man Variablen, die innerhalb des Blocks gelten, und die von der Methode übergeben werden:

```
|...|         # Passend zu den Argumenten von yield
```

  - Auch Iteratormethoden können mit Parametern aufgerufen werden:

```
foo.my_iter(...) do |x, y|
  ...
end
```

Iteratoren und Blöcke - typisch für Ruby!

- Beispiel: `each_pair` - ein neuer Iterator von **Array**

```
class Array
  def each_pair( inc=2 )    # 2 = default increment
    i=0
    while i < size-1 do     # entspricht self.size
      yield self[i], self[i+1] # Führe Block aus!
      i += inc
    end
  end
end
```

```
a = %w| 1 eins 2 zwei 3 drei |
a.each_pair{|zahl, wort| puts "#{zahl}\t#{wort}"}
```

```
1 eins
2 zwei
3 drei
```

Iteratoren und Blöcke - typisch für Ruby!

- Beispiel: `each_pair` – Variante mit „step“

```
class Array
  def each_pair( inc=2 )    # 2 = default increment
    1.step(size,inc){|i| yield self[i-1], self[i]}
  end
end

a = %w| 1 eins 2 zwei 3 drei |
a.each_pair {|zahl, wort| puts "#{zahl}\t#{wort}"}
```

```
1 eins
2 zwei
3 drei
```

Schleifen & Iteratoren: break, next, redo

```
while line=gets
  next if line=~/^\s*#/  # skip comment lines
  break if line=~/^END/  # stop if END entered
  # Zum Nachdenken...
  redo if line.gsub!(/`(.*?)`/){ eval($1) }
  # OK, nun verarbeite die Zeile
  puts line              # Hier: Nur ausgeben
end
```

```
$ breakredo.rb
  # foo                 (next-Zweig)
3+4 # bar
3+4 # bar               Nur normale Ausgabe
`3+4 # bar`
7                       redo-Zweig / interpretiert
ENDE
$                       break-Zweig
```

Erläuterungen zum Beispiel "breakredo.rb"

- Spezielle globale Variablen:
  - $_ Aktuelle Ein/Ausgabezeile, Default an vielen Stellen
    Perl-Erbe, hier vermieden mittels „line“. Vgl. altes Beispiel
  - $1 Hier: Erster Match-Wert des regex
  - beide: Perl-Tradition!

- Sonstiges:
  - gsub! Ersetzen aller Treffer durch Wert von {...}
    gsub! ergibt "nil" (damit "false") falls "Kein Treffer"
  - eval "evaluate" - Dynamische Code-Ausführung,
    vgl. Perl

- Merke: "Taschenrechner"

Schleifen & Iteratoren: retry

```
puts "Schaffen Sie es, 10mal richtig zu rechnen?"
n = 0
(1..10).each do |i|
   a, b = rand(100), rand(100); c = a + b
   print "#{i}. #{a}+#{b} = "
   r = gets.to_i
   n += 1
   retry if r != c # Rechenfehler: ALLE nochmal
end
puts n==10 ? "ok" : "#{n} Aufgaben statt 10"
```

```
Schaffen Sie es, 10mal richtig zu rechnen?
1. 29+89 = 118     # ok
2. 27+39 = 66      # ok
3. 41+53 = 95      # FEHLER
1. 75+99 =         # usw., aber wieder von 1!
......
```

Schleifen & Iteratoren: Scoping von Variablen

```
for x in [1, 2, 3] do y = x + 1 end
[x, y]        # [3, 4]
```

x: äußere Ebene
y: äußere Ebene

```
[1, 2, 3].each {|x| y = x + 1}
[x, y]        # Fehler: x unbekannt
```

x: innere Ebene
y: innere Ebene

```
x = nil
[1, 2, 3].each {|x| y = x + 1}
[x, y]        # Fehler: y unbekannt
```

x: äußere Ebene
y: innere Ebene

```
x = y = nil
[1, 2, 3].each {|x| y = x + 1}
[x, y]        # [3, 4]
```

x: äußere Ebene
y: äußere Ebene

Fazit: Offenbar sind **for x in .. do ... end** und **..each do |x| ... end** doch nicht völlig äquivalent! for .. in .. do: vgl. while, until, ...

Schleifen & Iteratoren: Scoping von Variablen

- Erläuterungen, Bemerkungen
  - Offenbar sind

```
for x in .. do  ... end
```

und

```
obj.each do |x| ... end
```

    doch nicht völlig äquivalent!

  - Das Verhalten von **for .. in .. do** bez. Variablen-Scoping entspricht dem der anderen Schleifenkonstrukte wie **while** und **until:**
    Code in der Schleife ist Teil des aktuellen Blocks.

  - Iteratoren dagegen führen separate Code-Blöcke aus,
    mit lokalem Scoping!

  - Vergleich zu C/C++