



Ruby

Die Skriptsprache mit Zukunft



Organisatorisches

incl. "Spielregeln" des Kurses



Wie lernt man eine Programmiersprache?



- Wie lernt man eine Programmiersprache wirksam?
 - Durch Lesen, Beispiele und Projekte, also **durch eigenes Tun!**
 - Das Praktikum: Unverzichtbar - und doch nur ein Einstieg.
- Warum Vorlesung?
 - Vorbereitung für das Praktikum
 - Schrittmacherfunktion
 - Vollständigkeit
 - Vergleiche mit anderen Programmiersprachen
 - Erklärungen: Konzepte herausarbeiten, Hintergründe verstehen
 - Fragen klären, Beispiele/Fallstudien diskutieren
 - Je aktiver Sie mitmachen, desto wirksamer wird's!
 - Gemeinsam geht's leichter und macht mehr Spaß.
 - Weil Sie in derselben Zeit alleine weniger verstehen und behalten ...
 - ... und weil Sie sich die Zeit vielleicht doch nicht nähmen ;-)



Das Konzept der Vorlesung



- Zunächst Breite, später Tiefe
 - Wir sprechen viele Themen an, um Ihnen viele Starthilfen zu geben
 - Vertiefen können Sie später nach Bedarf über Projektarbeiten
 - Effekte: Abbau von Hemmschwellen, Bereitstellung elementarer, aber schon funktionsfähiger Beispiele als Ausgangspunkte
- Hilfe zur Selbsthilfe
 - Vorlesung und Praktikum sollen Ihre Beschäftigung mit Ruby anregen und erleichtern.
 - Seien Sie neugierig! Testen Sie Ideen!
 - Seien Sie aktiv - durch "Absitzen" der Termine hat noch keiner Programmieren gelernt.
- Schrittmacher
 - Die Vorlesung "zieht" Sie in rascher Folge durch die Themen.
 - Das Praktikum erfordert regelmäßiges Mitmachen.
 - Für die Klausur zu wiederholen ist gut, für sie zu lernen ist zu spät!



Spielregeln



- **Leistungsnachweis**
 - Per Klausur (80%) und Übungen (20%), bestanden bei $\geq 50\%$
 - Erlaubte Hilfsmittel:
 - 2-seitige, handschriftliche Notizsammlung
(Idealerweise: Klausur am Rechner...)
- **Praktikum**
 - Mindestens 75% Anwesenheit (hier: 11 von 14)
 - Anreiz zum regelmäßigen Mitmachen:
Übungen werden - moderat - bepunktet (s.u.)
- **Aufteilung in die Praktikumsgruppen**
 - Zur Verfügung stehende Termine
 - Ggf.: Diskussion zu Überbelegungen/Nachmeldungen



Termine im WS 2005/06



Datum Montag	Vorlesung (Fr)	Gruppe A (Fr)	Gruppe B (Fr)
14.10.	Ja	Ja	Ja
21.10.	Ja	Ja	Ja
28.10.	Ja	Ja	Ja
4.11.	Ja	Ja	Ja
11.11.	Ja	Ja	Ja
18.11.	Nein	Nein	Nein
25.11.	Ja	Ja	Ja
2.12.	Ja	Ja	Ja
9.12.	Ja	Ja	Ja
16.12.	Ja	Ja	Ja
6.1.	Ja	Ja	Ja
13.1.	Ja	Ja	Ja
20.1.	Ja	Ja	Ja
27.1.	Ja	Ja	Ja
3.2.	Ja	Wh/Fragen	Wh/Fragen
10.2.	1. Klausurwoche		



Spielregeln



- Praktikumsablauf
 - Die ersten 5 Minuten: Abgabe der alten Aufgaben (letzte Chance)
 - Anschließend: Besprechung der Musterlösungen
 - Vorstellung der neuen Aufgabe(n)
 - Bearbeitung der neuen Aufgaben
- Anreiz-System
 - Mit jeder Abgabe können Sie 1 "sicheren" Punkt erwerben
2 Punkte bei schweren Aufgaben, manchmal Sonderpunkte
 - "Großzügige" Bewertung - auf's Mitmachen kommt es an.
 - Voraussetzungen
 - **Rechtzeitige Abgabe**
 - **Selbständige Bearbeitung**
 - Klare Kennzeichnung mit Name/MatNr
 - Faustregel:
 - 15-20 Punkte aus Praktikum ==> Klausur i.d.R. bestanden.



"Support"



- Web-Unterstützung: Homepage des Kurses nutzen!
 - Skripte (Kopien der Folien, PDF)
 - Praktikumsaufgaben
 - Aktuelle Mitteilungen
 - Linksammlung
- E-Mail nutzen
 - Für Fragen an den Dozenten oder aktuelle Anliegen
 - Option: Verteiler für alle Kursteilnehmer?
- Sprechstunde:
 - Di 13 - 14 Uhr und nach Vereinbarung

Besuch der Seite, sofern
Netzwerkanschluss vorhanden!



Nacharbeiten und Vertiefen - womit?



- Skript oder Präsentation?
 - Präsentationsfolien – stichwortartig, erfordern mündliche Ergänzung
 - Skript - zum Nachlesen. Siehe Literaturhinweise!
 - Kompromiss:
Einige Folien sind dicht beschrieben und zum Nachlesen gedacht.
- Hinweise zum Drucken
 - Folien werden inkrementell bereitgestellt
Vorbereitungen auch mit den Folien des WS2003 möglich
 - Mit Änderungen rechnen → möglichst spät drucken
 - 2 Versionen:
 - 2 Folien pro Seite, farbig - zum online-Lesen
 - 4 Folien pro Seite, s/w - zum Drucken
 - **PDFs zu Hause drucken, nicht in der FH (quotas!)**
- Nutzen Sie die Online-Version des ausgezeichneten "Pickaxe"-Buchs von Thomas & Hunt



Literaturhinweise (englisch)



- **David Thomas: Programming Ruby (2nd ed.).** (*the „Pickaxe book“*)
O'Reilly, 2005. ISBN 0-9745140-5-5. 564 Seiten, ca. 40 €.
 - Die erste nicht-japanische gute Dokumentation von Ruby
 - Erste Ausgabe auch on-line und inzwischen auch auf Deutsch erhältlich!
 - Mehrteiliger Aufbau: Umschau, Vertiefung, Systematik, Referenz
 - **Unverzichtbar als Referenzhandbuch! Ebenfalls Vorlesungs-Grundlage.**
- **Hal Fulton: The Ruby Way.** (*the „Coral book“*)
Sams Publishing, Indianapolis, 2002. ISBN 0-672-32083-5. 579 Seiten.
 - Zum Teil Leitfaden / Quelle von Beispielen in dieser Vorlesung
 - Didaktisch gut, vielseitig, zahlreiche gute Beispiele gerade für Informatiker.
 - Kein Referenz-Handbuch!
- **Yukihiro Matsumoto: Ruby in a Nutshell.** A Desktop Quick Reference
O'Reilly, 2002. ISBN 0-596-00214-9. 204 Seiten.
 - Vom Ruby-Erfinder selbst! Preiswert, kompakt, detailreich
 - Kein Lehrbuch, kaum Erklärungen / Beispiele, Referenzen etwas eigenwillig sortiert.
- **R. Feldt, L. Johnson, M. Neumann (Hrsg.): Ruby Developer's Guide.**
Syngress Publishing Inc., Rockland, MA, 2002. ISBN 1-928994-64-4. 693 Seiten.
 - Ein Buch für spätere Projektarbeiten. Recht aktuell, zeigt viele neue Module in Aktion.
 - Relativ teuer, Inhalt könnte kompakter ausfallen
 - Verfolgt kein didaktisches Konzept, sondern präsentiert eine Sammlung von kommentierten "case studies".



Literaturhinweise (deutsch)



- **David Thomas, Andrew Hunt: Programmieren mit Ruby.**
Addison-Wesley, 2002. ISBN 382731965-X. 682 Seiten, € 29,95
 - Die deutsche Ausgabe des "Klassikers". Empfohlen!
- **Armin Röhrli, Stefan Schmiedl, Clemens Wyss: Programmieren mit Ruby.**
dpunkt-Verlag GmbH, 2002. ISBN 389864151-1. 300 Seiten, € 36,-
 - Von einem engagierten und begeisterten Autorenteam, Organisatoren der EuRuKo03
 - Erfrischende Ideen, aktuell; weniger systematisch als Thomas/Hunt.
 - Keine Referenz; eher für erfahrene Entwickler/Umsteiger gedacht.
- **Klaus Zeppenfeld: Objektorientierte Programmiersprachen.**
Einführung und Vergleich von Java, C++, C# und Ruby.
Spektrum Akademischer Verlag, Oktober 2003. ISBN 382741449-0. 300 Seiten, € 29,95
 - Das Buch behandelt ein spannendes Thema für Entscheider, die eine Entwicklungsplattform festzulegen haben.
 - Der Autor ist Professor für Informatik an der FH Dortmund



Kursinhalte



- Teil 1: Ruby im Überblick ("*highlights*")
 - Für Teilnehmer wie Sie, die schon programmieren können
 - Ganz im Stil von Thomas/Hunt als auch Fulton
 - Viele Aspekte, selten erschöpfend behandelt, ergeben einen guten ersten Eindruck von Rubys Möglichkeiten und Vorteilen
 - Dauer: 2 - 3 Doppelstunden
- Teil 2: Ruby-Vertiefung
 - Konzeptionell wichtige Aspekte von Ruby eingehender besprochen
 - Dauer: 3 - 4 Doppelstunden
- Teil 3: Anwendungen
 - Eine breite Palette von Aufgaben im Entwickler-Alltag
 - Ruby-Kenntnisse werden angewendet und dabei weiter vertieft
 - Dauer: Restliche Zeit.



Kursinhalte - Teil 2: Vertiefung



- Umgang mit Textdaten
 - Der vielleicht häufigste Aufgabentyp für Skriptsprachen!
- Reguläre Ausdrücke
- Numerische Daten (Auswahl)
- Die Containerklassen "Array" und "Hash"
- OOP mit Ruby
- Die Betriebssystem-Umgebung:
 - Dateien, *environment*, Kommandozeile,
 - Shell-Kommandos, *system calls*
- Optional, je nach Fortschritt:
 - Thread-Programmierung in Ruby



Kursinhalte - Teil 3: Anwendungen



- Module, Namensräume, Mixins
- Extensions
 - Das Zusammenspiel mit Bibliotheken von C/C++
- Testen und Dokumentieren
 - RDoc und die Klasse "TestUnit"
- GUI-Entwicklung mit Ruby
 - FXRuby: Erste Schritte mit der portablen GUI-Bibliothek FOX
- Design patterns
 - Ruby's eingebaute Unterstützung:
Visitor, Singleton, Observer, Delegation
- Ruby und das Internet. Für Interessierte: Ruby on Rails
- Distributed Ruby
- BEACHTEN:
 - Dies ist eine Auswahlliste. Wir werden nicht alles davon schaffen!



Teil 1: Ruby im Überblick

Ein erstes Kennenlernen



Erste Schritte



Das „klassische“ erste Beispiel



- 5 Zeilen in „C“:

```
#include <stdio.h>
int main( int argc, char **argv ) {
    puts( "Hello, world!" );
    return( 0 );
}
```

- 1 Zeile in Ruby:

```
puts "Hello, world!"
```

→ "Hello, world!" an stdout

Reduktion auf das Wesentliche!

- **Fragen an Ruby:**

- Wie startet man so einen Einzeiler?
- Was ist denn daran objekt-orientiert??



Ruby-Interpreter: 3 Startoptionen



1. Quelldatei erzeugen & ausführen

- a) "hello.rb" mit Editor anlegen,

```
unix%> ruby hello.rb # Explizit
```

- oder -

- b) "#!"-Startzeile einfügen, Datei ausführbar machen, direkt aufrufen:

```
unix%> cat hello.rb
#!/usr/bin/env ruby
puts "Hello, world!"
unix%> chmod +x hello.rb
unix%> hello.rb
Hello, world!
```

TIPP:
Bei Windows unnötig, da
Ruby per Assoziation mit
Ext. ".rb" gestartet wird:

```
C:\temp> hello.rb
Hello, world!
```



Ruby-Interpreter: 3 Startoptionen



2. Per Kommandozeile und execute-Option:

Unix/Linux:

```
unix%> ruby -e "puts \"Hello, world!\""
Hello, world!
unix%> ruby -e 'puts "Hello, world!">'
Hello, world!
```

Windows:

```
c:\temp> ruby -e "puts \"Hello, world!\""
Hello, world!
c:\temp> ruby -e 'puts "Hello, world!">'
Hello, world!
```

- Mehrzeiler möglich: `ruby -e "..."` `-e "..."`
- Beliebt für *ad hoc*-Kommandos!



Ruby-Interpreter: 3 Startoptionen



Vorsicht bei Interpretation von Sonderzeichen:

```
unix%> ruby -e "puts \"My\t world\""
My      world
unix%> ruby -e 'puts "My\t world"'
My      world
unix%> ruby -e "puts 'My\t world'"
My\t world
```

```
c:\temp> ruby -e 'puts "My\t world"'
My      world
c:\temp> ruby -e "puts 'My\t world'"
My\t world
```

Analogie zum Verhalten der Unix-Shell !



3. Mit "Interactive Ruby" (irb):

```
unix%> irb
irb(main):001:0> puts "Hello, world!"
Hello, world! ← Erwarteter Output
==> nil ← Rückgabewert des Ausdrucks "puts ...!"
irb(main):002:0> exit
unix%>
```

```
c:\temp> irb
irb(main):001:0> puts "Hello, world!"
Hello, world!
==> nil
irb(main):002:0> exit
c:\temp>
```

Ganz analog
zu Unix!



- Ja!
 - Die OO ist implizit vorhanden
 - Sie wird hier nicht aufgezwungen
 - Ruby ist kein "Prinzipienreiter", sondern verabreicht den Entwicklern viel "*syntactic sugar*" - wie etwa hier.

- Ausführlicher:

```
$stdout.puts( "Hello, world!" );# Ausführlich ...
$stdout.puts "Hello, world!" # oder "versüßt"
```

- Erläuterungen:

- `$stdout` ist ein (vordefiniertes) Objekt der eingebauten Klasse "IO"
- `puts()` ist eine Methode dieses Objekts
- Notation: „\$“ kennzeichnet globale Variablen



- Wenn Objekte „einfach so“, also ohne Deklaration, erscheinen können - wie erfahre ich denn, zu welcher Klasse ein Objekt zählt?
- Allgemeiner:
Kann man die Klasse eines Objekts zur Laufzeit ermitteln?

- Natürlich!

```
puts $stdout.class  
→ IO          # Objekt "$stdout" zu Klasse "IO"
```

- Objekte geben über sich selbst Auskunft!
- Dazu stellt Ruby eine Reihe von Methoden bereit, die alle Objekte besitzen.
- Realisiert als Methoden der Klasse "Object".
"Object" ist gemeinsame Oberklasse aller Klassen.



- Ist wirklich alles in Ruby ein Objekt?
 - Wie steht's denn mit Strings und Zahlen?
- Probieren wir es aus, z.B. mit irb:

- Strings:

```
"abc".class → String  
'A'.class  → String
```

- Zahlen:

```
42.class → Fixnum  
3.14159.class → Float  
1234567890123.class → Bignum
```

- Und Programmcode?
 - Hier ist Ruby weniger radikal als etwa Smalltalk: Programmcode ist i.a. kein Objekt - es gibt aber die Klasse "Proc":

```
p = Proc.new {|name| puts "Hello, #{name}!"}  
p.class → Proc  
p.call "Dave" → Hello, Dave!
```



- Ein Wort zur Begriffsbildung
 - Ein Objekt ist ein Exemplar einer Klasse
 - Der Begriff "Instanz" ist ein Übersetzungsfehler - meiden!

engl. "*instance*" = Exemplar, Beispiel

"Script languages are cool. Take Ruby for instance!"

deutsch "Instanz" wie in "juristische Instanzen",
etwa "Landgericht", "Oberlandesgericht", ...



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 einem \$-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 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
```



- 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, wissenschaft. Notation
Math::PI    # 3.14159265358979, aus Modul "Math"
```



Variablen vs. Objekte

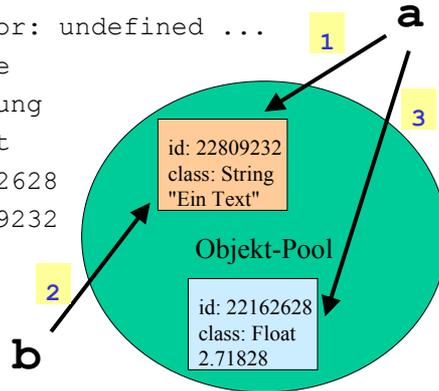


- Objekte sind "typisiert" - durch ihre Klassen & Meth.
- Variablen sind nur Verweise auf Objekte:

```

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



- 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*, ...



- 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
```



Offene Punkte vom letzten Mal



- Welche Art Sprache ist Ruby eigentlich?
 - OO-Sprache, „Skript“-Sprache, Hochsprache, *general purpose language*, Interpreter-Sprache (VM in Sicht!) auf C-Basis
 - Verdrängt Perl und PHP, konkurriert mit Python, lockt Entwickler von Java, C++, C# wegen der hohen Produktivität, ...
 - rein objekt-orientiert, aber auch mit Elementen prozeduraler, funktionaler und deklarativer Sprachen.
- 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).
```



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

```
?x          # 120 (ASCII-Code zum 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!



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, Perl's manchmal kryptische Notationen durch lesbarere Varianten zu ersetzen.



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

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



Zeichenketten (Strings)



- Generalisierte Notation:
 - Wie vermeidet man das "Escaping"? Wir definieren uns einen Begrenzer, der nicht im Nutzttext 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>
```



- 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



- `%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 Beispiele:



```

/^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'"

```



- *Here documents*:
 - 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

```



- *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!
```



- 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)
 - Unicode-Unterstützung frühestens ab Ruby 2.0



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)



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
```



- 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!"
```



- Ruby unterstützt Mengen nicht direkt (wie etwa Pascal)
 - 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
 - Ersatz für Operator "in"
 - Mengenoperationen: Vereinigung, Durchschnitt, Differenz
- Einzelheiten: The Ruby Way, p.148ff
- Siehe auch: Ruby Standard Library „Set“



- Vorbemerkungen:
 - Hashes sind die vielleicht wichtigsten Datentypen in Skriptsprachen.
 - Viele Entwickler 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.



- 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
```

- **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!
```



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 Regexp (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 Regexp im Vertiefungsteil.



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"
```



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 <<
- Betrachten Sie Operatoren als Kurzformen von Methoden!
- Sie können daher i.d.R. 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
- 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          unless x >= 5 then
    statement1         statement1
end                    end
```

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

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

```
x = if a>0 then b \    x = unless a<=0 then b \
     else c end        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
```
- 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`
- Was wird ausgegeben?
 - "Zweig 3" ! Warum??



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
  print "#{liste[i]} "
  i += 1
end
```
 - Version 2 (until)

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



Variationen zum Thema "Schleifen"



- Version 3 (*while* am Ende)

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

- Version 4 (*until* am Ende)

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

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

```
i=0
n=liste.size-1
loop do
  print "#{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?  
  print "#{x} "  
end
```

- Version 7a (Kurzform)

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

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

```
for x in liste do         # Etwas "Syntax-Zucker"  
  print "#{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| print "#{liste[i]} " }
```

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

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

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

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

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

```
liste.each_index { |i| print "#{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

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

und

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++



Einschub: „FAQ“



- Wie verhalten sich die Schleifenkerne mit "Test am Ende" im Fall "leere Liste"?
 - Sie werden tatsächlich einmal durchlaufen (=> whiletest.rb)! Ausgabe `print "#{liste[i]} "` ergibt aber keine Fehlermeldung:
 - `liste[i]` ist zwar `nil`, ausgegeben wird aber `nil.to_s` (hier "")!
- Hätte das Fehlen des "+" an folgender Stelle funktioniert?

```
def to_s          # Standardmethode to_s überladen
  "+@re.to_s+", "+@im.to_s+"
end
```

 - NEIN! (Syntaxfehler)
- Kann man sich eigene Operatoren "beliebig" ausdenken?
 - NEIN. Der Ruby-Parser erkennt sie nicht an. Wie sollte überhaupt die Bindungsstärke eigener Operatoren behandelt werden?
 - Siehe die Liste der als Methoden implementierten (und daher überladbaren) Operatoren.



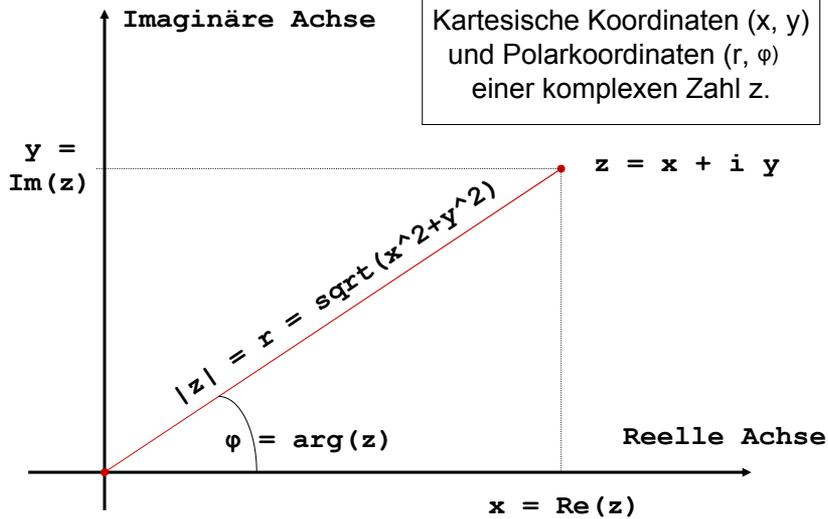
Einschub: „FAQ“ (2)



- Gibt es den Iterator `each_index` auch für Klasse `hash`?
 - Nein, aber es gibt `each_key` mit genau der gesuchten Wirkung!
 - Sie können ggf. `each_index` als Alias auf `each_key` hinzufügen.



Erinnerung: Die Gauß'sche Zahlenebene



03.11.2005

(c) 2003, 2005 H. Wertges, FB Informatik, FH Wiesbaden

85



OOP in Ruby: Definieren einer Klasse



```
class Complexnum          # Unterklasse von "Object"  
  def initialize(r=0.0, i=0.0) # Defaults: 0.0  
    @re = r  
    @im = i                # Attribute !  
  end  
end
```

```
a = Complexnum.new( 0.0, -1.0 )  
b = Complexnum.new( 2.0 )  
c = Complexnum.new  
puts a, b, c                # Verwendet Object#to_s
```

```
#<Complexnum:0x2a67a08>  
#<Complexnum:0x2a679a8>  
#<Complexnum:0x2a67990>
```

03.11.2005

(c) 2003, 2005 H. Wertges, FB Informatik, FH Wiesbaden

86



OOP in Ruby: Definieren einer Klasse



- Bemerkungen
 - Der Konstruktor "new" wird von Ruby bereitgestellt. Er wird normalerweise nicht überladen.
 - "new" ruft "initialize" auf, welches Gelegenheit gibt, das soeben angelegte Objekt zu füllen.
 - Ruby kennt keinen Destruktor - nicht mehr benötigte (referenzierte) Objekte werden automatisch von der GC entfernt.



OOP in Ruby: Überladen einer Methode



```
# Erweitern einer bereits vorhandenen (!) Klasse:  
#  
class Complexnum # Einfach nochmal "öffnen"..  
  def to_s      # Standardmethode to_s überladen  
    "(" + @re.to_s + ", " + @im.to_s + ")"  
  end  
end
```

```
# Erneut ausgeben:  
puts a, b, c      # Verwendet nun Complexnum#to_s
```

```
(0.0, -1.0)  
(2.0, 0.0)  
(0.0, 0.0)
```



```
class Complexnum
  def re      # Getter für @re
    @re
  end
  def im      # Getter für @im
    @im
  end
  def re=(v) # Setter für @re
    @re = v
  end
  def im=(v) # Setter für @im
    @im = v
  end
end
```

```
a.im      # -1.0
a.re = 3.0
puts a
```

```
(3.0, -1.0)
```



Entweder separat:

```
class Complexnum
  attr_reader :re, :im # Legt die Getter an
  attr_writer :re, :im # Legt die Setter an
end
```

oder gleich gemeinsam:

```
class Complexnum
  # Getter und Setter gleichzeitig anlegen:
  attr_accessor :re, :im
end
```

Kommentare:

- :re und :im sind Symbole



```
class Complexnum
  def abs      # Getter?
    # aus @re und @im berechnen ...
  end
  def arg      # Getter?
    # aus @re und @im berechnen ...
  end
  def abs=(v) # Setter?
    # @re und @im neu berechnen ...
  end
  def arg=(v) # Setter?
    # @re und @im neu berechnen ...
  end
end
```

- OO-Trend "**Uniform access principle**" (B. Meyer, 1997):
Die Grenzen von Methoden und Attributen verschwimmen! Neue Möglichkeiten für späteres "*refactoring*" ohne Konflikt mit Anwendern.
Hier: Von außen ist nicht erkennbar, ob unsere Klasse intern mit Kartesischen oder Polarkoordinaten arbeitet.



```
# Erneut Erweitern der Klasse:
#
class Complexnum
  def +(z)          # Komplexe Addition
    Complexnum.new(@re + z.re, @im + z.im)
  end

  def absq         # Betragsquadrat
    @re * @re + @im * @im
  end
end
```

```
puts a, b, a+b, a.absq
```

```
(3.0, -1.0)
(2.0, 0.0)
(5.0, -1.0)
10.0
```



```
# Erweitern der Klasse, initialize umdef.
#
class Complexnum
  @@zaehler = 0          # Klassenattribut!
  def initialize(r=0.0, i=0.0)
    @re = r; @im = i
    @@zaehler += 1
  end
  def Complexnum.counter # Klassenmethode!
    @@zaehler
  end
end
```

```
arr = [ Complexnum.new, Complexnum.new(1.0),
        Complexnum.new(1.0, 2.5), Complexnum.new ]
puts Complexnum.counter
```

```
4
```



- **Keine** Mehrfach-Vererbung!
 - Ruby betrachtet M. als problematisch / zu vermeiden.
- Statt dessen: "Mixins"
 - Matz: *"Single inheritance with implementation sharing"*
- 2 Varianten der Einfach-Vererbung:
 - 1) Normale Vererbung:

```
class MySubclass < MyParentClass
  ...
end
```

Default:

```
class MyClass < Object
  ...
end
```

Optional!



Verwendung von Methoden der Basisklasse: `super`

```
class Person
  def initialize(name, geb_datum)
    @name, @geb_datum = name, geb_datum
  end
  # Weitere Methoden ...
end

class Student < Person
  def initialize(name, geb_datum, matr_nr, st_gang)
    @matr_nr, @st_gang = matr_nr, st_gang
    super(name, geb_datum) # initialize() von "Person"
  end
end

a = Person.new("John Doe", "1950-01-01")
b = Student.new("Irgend Jemand", "1980-12-01",
               123456, "Allgemeine Informatik")
```



- Normale Methodensuche
 - Ruby sucht zunächst in der aktuellen Klasse nach dem passenden Methodennamen
 - Wird dort nichts gefunden, setzt Ruby die Suche in der nächsthöheren Basisklasse fort, usw.
 - Exception "NoMethodError", falls Suche erfolglos.
- Die Wirkung von "super"
 - "super" (verwendet wie ein Methodennamen) bewirkt, dass mit der Suche nach dem aktuellen Methodennamen in der direkten Basisklasse begonnen wird.
- Erinnerung: Abstraktionsmittel "*Ist-ein-Beziehung*"
 - Bsp.: Ein Cabriolet *ist ein* Auto. Ein Auto *ist ein* Fahrzeug.

(nach N. Josuttis, aus: B. Oesterreich, Objektorientierte Software-Entwicklung, Oldenbourg, 2001)

Konsequenz für die OOP: **Objekte einer abgeleiteten Klasse sollten stets auch Exemplare aller ihrer Basisklassen sein!**



- **Achtung - Designfehler!**

```
class Quadrat
  def initialize(a)
    @a = a
  end
  def flaeche
    @a * @a
  end
end
class Rechteck <Quadrat
  def initialize (a, b)
    @b = b
    super(a)
  end
  def flaeche
    @a * @b
  end
end
```

```
r = Rechteck.new(3, 4)
q = Quadrat(5)
r.flaeche      # 12 (ok)
q.flaeche      # 25 (ok)
# Funktioniert zwar ...

r.is_a? Quadrat # true
q.is_a? Rechteck # false
# Offenbar unsinnig!

# Flächenberechnung
# ferner redundant
# implementiert!
```



- **Korrekte Version:**

```
class Rechteck
  def initialize (a, b)
    @a, @b = a, b
  end
  def flaeche
    @a * @b
  end
end
class Quadrat < Rechteck
  def initialize(a)
    super(a, a)
  end
end
```

```
r = Rechteck.new(3, 4)
q = Quadrat(5)
r.flaeche      # 12 (ok)
q.flaeche      # 25 (ok)

r.is_a? Quadrat # false
q.is_a? Rechteck # true
# Viel besser!

# Methode "flaeche"
# komplett geerbt.
```



2) Singleton-Klassen:

```
a = "hallo"; b = a.dup
class <<a
  def to_s      # Überladen von to_s nur für a
    "Der Wert ist '#{self}'"
  end
  def zweifach  # Neue Methode, nur für a
    self + self
  end
end
```

```
a.to_s      # "Der Wert ist 'hallo'"
a.zweifach  # "hallohallo"
b.to_s      # "hallo"
```



2) Singleton-Klassen, alternative Notation:

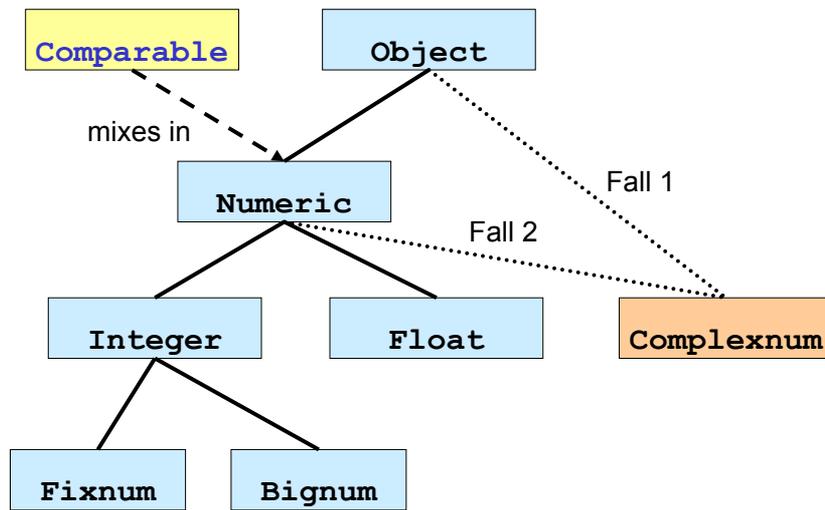
```
a = "hallo"
b = a.dup

def a.to_s      # Überladen von to_s nur für a
  "Der Wert ist '#{self}'"
end
def a.zweifach  # Neue Methode, nur für a
  self + self
end
```

```
a.to_s      # "Der Wert ist 'hallo'"
a.zweifach  # "hallohallo"
b.to_s      # "hallo"
```



OOP in Ruby: Vererbung, Klassenhierarchie



OOP in Ruby: Vererbung, "Selbstauskunft"



a = Complexnum.new(1.0, 2.0)			
Vergleichs-objekt	a.instance_of?	a.kind_of?, a.is_a? (1)	a.kind_of?, a.is_a? (2)
Numeric	false	false	true
Integer	false	false	false
Fixnum	false	false	false
Complexnum	true	true	true
Comparable	false	false	true



"Implementation sharing" in Aktion:

```
class Complexnum
  def <=>(z)          # Nur <=> implementieren
    if z.is_a? Complexnum # Konzept s.u.
      self.absq <=> z.absq
    else
      self.absq <=> z*z
    end
  end
end
```

```
puts a < b          # a, b: Complexnum-Objekte
False              # Operator '<' "geerbt"
```

– Implementieren von <=> erschließt automatisch:

```
<, >, <=, >=, ==, between?
```



- Anmerkungen
 - Das Beispiel ist nicht gerade sinnvoll, da komplexe Zahlen keine Ordnungsrelation besitzen, sondern nur ihre Beträge.
 - Es zeigt aber die generelle Wirkung:
 - Implementieren weniger zentraler Methoden, plus
 - "Mixing in" eines Moduls wie "Comparable", das Methoden enthält, die nur auf diesen Methoden basieren
- ==> Diverse Methoden werden auch für die neue Klasse quasi "geerbt", fast wie bei Mehrfachvererbung.



OOP in Ruby: Mixins, Bsp. "Enumerable"



- Weiteres wichtiges Beispiel: Enumerable.

Benötigt:

```
each
```

```
<=>
```

Stellt bereit:

a) nur **each**:

```
collect / map,  
each_with_index,  
find / detect,  
find_all / select,  
grep,  
include? / member?,  
reject,  
to_a / entries
```

b) plus **<=>**:

```
max, min, sort
```



OOP in Ruby: Mixins, Bsp. "Enumerable"



- Bemerkungen

– "Enumerable" wird später vertieft (Vorstellung der Methoden), ebenso das Modulkonzept.

– Die Selbstauskunfts-Methoden

```
kind_of? bzw. is_a? (alias)
```

akzeptieren auch "mixed in" Modulnamen, so als wären sie Elternklassen im Fall eines Mehrfachvererbungs-Modells:

```
class Complexnum < Numeric  
  ...  
end  
a = Complexnum.new(1.0, 2.0)  
a.is_a? Numeric           # true  
a.kind_of? Enumerable    # true !
```



Exceptions: Vorbemerkungen



- Sorgfältiger Umgang mit Laufzeitfehlern ist unverzichtbar!
 - Unter Produktionsbedingungen ist es durchaus normal, wenn mehr Code zu Fehlerbehandlungen als zur eigentlichen Lösung einer Aufgabe entsteht.
 - Dynamische Programmiersprachen wie Ruby können besonders flexibel auf Laufzeitfehler reagieren und ggf. Abbrüche vermeiden.
 - Ruby besitzt ausgereifte Werkzeuge für einen sauberen und effizienten Umgang mit Fehlersituationen.

- Merke:

Wer Fehlerbehandlung nicht beherrscht,
kann noch nicht programmieren!

Wer Fehlerbehandlung nicht praktiziert,
programmiert nicht professionell.



Exceptions: Konzept



- *Compile-time?*
 - Ruby ist ein *one-pass interpreter*, kann also auch aus *Pipes* lesen.
 - Übergebener Code wird zunächst sequenziell interpretiert,
 - Fehler in dieser Phase werden als Syntaxfehler gemeldet.
 - Anschließend wird der Code ausgeführt --> Laufzeitfehler.
- **Ausnahmefehler zur Laufzeit:**
 - Unterschiedlichste Gründe führen zu Laufzeitfehlern. Ruby verwaltet diese Gründe mittels einer Hierarchie von "*Exception*"-Klassen.
 - *Exceptions* werden vom Ort ihrer Ursache aus den "*calling stack*" hinauf ausgebreitet, bis dass eine Fehlerbehandlungs-Routine sich der speziellen Fehlerart annimmt.
 - Ist kein passender "*exception handler*" vorhanden, greift die Default-Fehlerbehandlung des Ruby-Interpreters ein, i.d.R. mit Abbruch.
- **Recovery**
 - Geschickte *exception handler* gestatten Fortsetzung des Programms!



Exceptions: Unterklassen der "Exception"-Klasse



- **fatal** (von Ruby intern genutzt)
- **Interrupt**
- **NoMemoryError**
- **SignalException**
 - Interrupt
- **ScriptError**
 - LoadError
 - NotImplementedError
 - SyntaxError
- **SystemExit**
- **SystemStackError**
- **StandardError**
 - ArgumentError
 - IOError
 - EOFError
 - IndexError
 - LocalJumpError
 - NameError
 - NoMethodError
 - RangeError
 - FloatDomainError
 - RegexpError
 - RuntimeError
 - SecurityError
 - SystemCallError
 - (systemabh. Ausnahmen, *Errno: :xxx*)
 - ThreadError
 - TypeError
 - ZeroDivisionError

Teil jedes Exception-Objekts:

- *message string*
- *stack backtrace*



Exceptions: raise / fail



Auslösen von Ausnahmefehlern mittels raise / fail

```
class Complexnum
  def <=>(z)          # Nur <=> implementieren
    if z.is_a? Complexnum
      self.absq <=> z.absq
    elsif z.is_a? Numeric
      self.absq <=> z*z
    else
      raise TypeError, "Wrong class: #{z.class}"
    end
  end
end
```

```
puts a < "x"          # Vergleich mit String
```

```
complexnum.rb:120:in '<=>': Wrong class: String (TypeError)
```



Exceptions: raise / fail



Aufruf-Varianten von `raise` (**fail** ist ein Alias von `raise`):

- Mit expliziter *exception*-Angabe und String

```
raise TypeError, "My error reason"
```

- Nur mit expliziter *exception*-Angabe

```
raise TypeError
```

Als Fehlertext wird ein Default genommen (Name der Fehlerklasse).

- Nur mit String (Fehlermeldung des Anwenders)

```
raise "My error reason"
```

Implizit wird `RuntimeError` ergänzt.

- Ohne Parameter

```
raise
```

Den (in `!` gespeicherten) letzten Ausnahmefehler erneut auslösen.

I.d.R. nur von *exception handler* verwendet. Default: `RuntimeError`.

- Mit expliziter *exception*-Angabe, String und *stack trace*-Kontrolle

```
raise TypeError, "My error reason", caller[0..2]
```



Exceptions: raise / fail



Was passiert beim Aufruf von `raise` bzw. `fail`?

- Ruby belegt ggf. das angegebene `Exception`-Objekt mit der Fehlermeldung des Anwenders und seiner *stack backtrace*-Variante.
- Ruby legt in der globalen Variablen `!` eine Referenz auf dieses Objekt an.
- Ruby sucht in der Umgebung des Fehlers nach einem passenden "*rescue*" (s.u.) und führt dieses ggf. aus.
- Falls nicht vorhanden, wandert Ruby den *calling stack* hinauf auf der Suche nach einem passenden "*rescue*".
- Falls kein "*rescue*" ihm zuvorkommt, fängt der Ruby-Interpreter selbst den Ausnahmefehler ab und bricht das laufende Programm ab.



Exceptions: rescue



Fehler abfangen mittels **rescue**: Ein Beispiel

```
while line=gets.chomp
  exit if line.upcase == "EXIT"
  begin          # Anfang der "rescue-Zone"
    eval line    # Hier entstehen Fehler!
  rescue SyntaxError, NameError => reason1
    puts "Compile error: " + reason1
    print "Try again: "
  rescue StandardError => reason2
    puts "Runtime error: " + reason2
    print "Try again: "
  end           # Ende der "rescue-Zone"
end
```

- Match-Suche ähnlich zu case / when: Stopp bei 1. Treffer
Grundlage des Vergleichs: `$.kind_of? (parameter)`



Exceptions: ensure



Aufräumen sicherstellen mittels **ensure**: Ein Beispiel

```
begin
  f = File.open("testfile")
  # verarbeite Dateiinhalt...
rescue
  # Behandle Fehler ...
else
  puts "Glückwunsch -- ohne Fehler!"
ensure
  f.close unless f.nil?
end
```

else-Zweig

Wird nur durchlaufen, wenn es keine Fehler gab; selten benötigt.

ensure-Zweig

Wird immer durchlaufen!



Exceptions: retry



Reparieren & nochmal versuchen mit **retry**: Ein Beispiel

```
fname = "nosuchfile"; retries = 0
begin
  f = File.open fname
  # main activity here ...
rescue => reason1
  retries += 1          # Avoid endless loops!!!
  raise if retries > 3 # 3 retries then give up
  puts "File handling error: "+reason1
  print "Retry with filename: "
  fname = gets.chomp
  raise if fname.nil? or fname.empty?
  retry
ensure
  f.close unless f.nil?
end
```



Exceptions: Ableiten eigener Exception-Klassen



Ein Beispiel:

```
# Definition:
class MyRuntimeError < RuntimeError
  attr :my_attr
  def initialize(some_value)
    @my_attr = some_value
  end
end

# Anwendung (Fehlerobjekt + Inhalt anlegen):
raise MyRuntimeError.new(err_param),
      "Strange error"

# Rettung (Zugriff auf Fehlerobjekt + Inhalt):
rescue MyRuntimeError => sample
  puts "Found strange parameter: "+sample.my_attr
  raise
```



rescue1:

Eingabe von Ruby-Code,
Abfangen von Compile- und Laufzeitfehlern

nosuchfile_ensure:

rescue, retry, ensure für Korrektur bei falschen Dateinamen

myruntimeerror:

Umgang mit eigenen Fehlerklassen

promptandget_catch_throw:

Nutzung von catch & throw beim Umgang mit Pflichtfeldern
und optionalen Eingaben.



Ein alternativer Mechanismus?

- Nicht nur Ausnahmefehler sind Anlässe, den normalen Verarbeitungsfluss zu unterbrechen.
- Normale Verarbeitung kann in bestimmten Situationen mit **catch & throw** geordnet und übersichtlich unterbrochen werden.
- Rückkehr aus tiefen Verschachtelungen ist manchmal einfacher per Ausnahmetechnik als durch lokale Fallunterscheidungen.
- **throw:**
 - Durch Aufruf von **throw** und Übergabe ("Werfen") eines Symbols oder Strings wird die Kontrolle an einen Fänger übergeben - wenn es einen gibt.
- **catch:**
 - Mittels **catch** werden Code-Blöcke markiert, aus denen herausgesprungen wird, wenn das vorgegebene Symbol "gefangen" wird.



Exceptions: catch & throw



... mit **catch**: Ein Beispiel

```
catch (:done) do
  while csvfile.gets
    throw :done unless fields = split(/,/,)
    csv_array << fields
  end
  csv_array.do_something # ...
end
```

- Methode `csv_array.do_something` wird nur ausgeführt, wenn alle `split`-Aufrufe erfolgreich waren.
- Eine fehlerhafte Eingabezeile sorgt also für einen geordneten Abbruch nicht nur der Eingabe, sondern auch der (von korrekter Eingabe abhängigen) Folgeaktionen.



Exceptions: catch & throw



... mit **catch**, kombiniert mit **begin/rescue**: Ein weiteres Beispiel

```
def promptAndGet(prompt)
  print prompt; answer = readline.chomp
  throw :quitRequested if answer.empty?
  res
end

begin
  name = promptAndGet("Name: ") # Pflicht
  email = promptAndGet("E-Mail: ") # Pflicht
  catch :quitRequested do # Optional
    alter = promptAndGet("Alter: ")
    beruf = promptAndGet("Beruf: ")
  end
  rescue NameError # Für throw aus Pflichtteilen!
    puts "Pflichtfeld ausgelassen!; exit 1"
  end
  # Weitere Verarbeitung ...
end
```



Tipps zum Abschluss des Einstiegs in Ruby

*"Everything is intuitive
once you understand it"*

(nach: The "Pickaxe Book", But it doesn't
work!, p.129f, und Hal Fulton, The Ruby Way,
Training Your Intuition, p. 47ff.)



Tipps ...



- Setter tut's nicht?
`setter = ...` belegt lokale Variable "setter"
`self.setter = ...` sichert den Aufruf von Methode "setter"!
- Parserfehler am Dateiende?
 - Ein "end" kann zwischendurch fehlen
- Komma vergessen?
 - Seltsame Effekte, falls Komma in Parameterliste fehlt...
- Öffnende Klammer direkt hinter Methodennamen!
 - `my_method(param),` nicht `my_method (err)`
- I/O-Pufferung durch Ruby!
 - Output-Reihenfolge unerwartet? Z.B. `$stderr.sync` nutzen!
- Zahlen funktionieren nicht?
 - Vielleicht sind sie Strings! Ruby wandelt Text beim Einlesen nicht automatisch in Zahlen. Ggf. nachhelfen mit `to_i` bzw. `to_f`!



Tipps ...



- Optionale Klammern

- Gleichwertig:

<code>foobar</code>	<code>foobar()</code>
<code>foobar(a, b, c)</code>	<code>foobar a, b, c</code>
<code>x y z</code>	<code>x(y(z))</code>
<code>my_meth({a=>1, b=>2})</code>	<code>my_meth(a=>1, b=>2)</code>
# Whitespace sinnvoll einsetzen:	
<code>x = y + z; x = y+z; x = y+ z</code>	
<code>x = y +z;</code>	<code>x = y(+z)</code>

- Blocks

- Nur an dafür vorgesehenen Stellen erlaubt, nicht beliebig wie in C.

- Retry mit 2 Bedeutungen - nicht verwechseln!

- a) in Iteratoren
- b) zur Ausnahmebehandlung



Tipps ...



- Unterschied zwischen do...end und {...} in Iteratoren

```

mymethod param1, foobar do ... end
mymethod(param1, foobar) do ... end # Gleichwertig
# Aber:
mymethod param1, foobar { ... }
mymethod param1, (foobar { ... }) # Gleichwertig

```

- Konvention: Einzeiler in {...}, mehrzeilige Blocks in do ... end

- Konstanten in Ruby

- Nicht veränderbar aus normalen Methoden heraus.
- Sonst dennoch veränderbar (also keine echten Konstanten):

```

$ irb
irb> puts Math::PI          # 3.14159265358979
irb> Math::PI = 3.0
warning: already initialized constant PI
irb> puts Math::PI          # 3.0

```



Tipps ...



- "Geek baggage"

- Characters

In Ruby echte Integers, nicht Bytes wie in C:

```
x = "Hello"; y=?A
print "x[0] = #{x[0]}\n" # "x[0] = 72\n"
print "y = #y\n"        # "y = 65\n"
y == "A" ? "yes" : "no"  # "no" (!)
```

- Kein "boolean"

Es gibt nur TrueClass mit (einzigem) Exemplar true sowie FalseClass mit false.

- Kein Operator ++ bzw. --

- Modulo-Operator: Vorsicht bei negativen Operanden!

```
5 % 3      # 2
-5 % 3     # 1
5 % -3     # -1
-5 % -3    # -2
```



Tipps ...



- "Geek baggage" (Forts.)

- Was ist "false" in Vergleichen?

```
r = (false) ? "Ja" : "Nein" # "Nein": Klarer Fall
i = 0; r = (i) ? "Ja" : "Nein" # "Ja", anders in C!
s = ""; r = (s) ? "Ja" : "Nein" # "Ja", anders in C!
r = (u) ? "Ja" : "Nein" # "Nein" (u uninitialized)
x = nil; r = (x) ? "Ja" : "Nein" # "Nein": nil -> false
```

- Variablen

sind untypisiert - anders: Klassen der Objektreferenzen werden nicht deklariert. Üblich aber: myvar = nil; ...

- "loop" vs. "while" oder "until":

loop ist eine Methode (aus Modul "Kernel"), while bzw. until sind reservierte Wörter / Kontrollstrukturen!

- Zuweisungsoperator = bindet stärker als or und and:

```
y = false; z = true
x = y or z # Ausdruck: true; x == false (!!)
```



- "Geek baggage" (Forts.)
 - Schleifenvariablen (und ihre Änderbarkeit)

```
for var in 1..10
  print "var = #{var}\n"
  if var > 5
    var += 2      # zulässig, aber unwirksam!
  end
end
```

- Post-test loops?

```
# Folgende puts-Anweisungen werden nicht ausgeführt
puts "in der Schleife..." while false
puts "noch immer in der Schleife..." until true
```



- "Geek baggage" (Forts.)
 - case-Statement / when

```
# Kein "drop through"!
# == und === sind verschieden! a === b und b === a auch!
case "Hello"
  when /Hell/
    puts "match!"      # Trifft zu!
  else
    puts "no match."  # Kein "drop through"
end
#
case /Hell/
  when "Hello"
    puts "match!"      # "Hello"===/Hell/ ist false
  else
    puts "no match."  # Trifft zu!
end
```