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 Methodenname!
  - `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:

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

- Blöcke
  - 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 Blöcke 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 (bis 1.8.x) 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 uninitialisiert)
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 Wörte / Kontrollstrukturen!
  - Zuweisungsoperator = bindet stärker als `or` und `and`:

```
y = false; z = true
x = y or z    # Ausdruck, wie: (x = y) or z; x==false
x = y || z    # Ausdruck, wie:  x = (y||z) ; x==true
```

Tipps ...

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

```
for var in 1..10
  puts "var = #{var}"
  var += 2 if var > 5 # zulässig, aber unwirksam
end                   # für den Schleifenablauf!
```

  - *Post-test loops*?

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

```
# … diese hier werden einmal ausgeführt:
begin puts "in der Schleife..." end while false
begin puts "noch immer ..." end until true
```

Tipps ...

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

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