Exceptions

Ausnahmebehandlung mit Ruby

© 2003, 2008 H. Werntges, FB Design Informatik Medien (DCSM), FH Wiesbaden

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.

© 2003, 2008 H. Werntges, FB Design Informatik Medien (DCSM), FH Wiesbaden

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!

© 2003, 2008 H. Werntges, FB Design Informatik Medien (DCSM), FH Wiesbaden

Exceptions: Unterklassen der Klasse "Exception"

- **`fatal`** (von Ruby intern genutzt)
- **`NoMemoryError`**
- **`ScriptError`**
  - **`LoadError`**
  - **`NotImplementedError`**
  - **`SyntaxError`**
- **`SignalException`**
  - **`Interrupt`**
- **`SystemExit`**
- **`SystemStackError`**

Teil jedes Exception-Objekts:

- *message string*
- *stack backtrace*

- **`StandardError`**
  - **`ArgumentError`**
  - **`IOError`**
    - **`EOFError`**
  - **`IndexError`**
  - **`LocalJumpError`**
  - **`NameError`**
    - **`NoMethodError`**
  - **`RangeEroor`**
    - **`FloatDomainError`**
  - **`RegexpError`**
  - **`RuntimeError`**
  - **`SecurityError`**
  - **`SystemCallError`**
    (systemabh. Ausnahmen, ***`Errno::xxx`***)
  - **`ThreadError`**
  - **`TypeError`**
  - **`ZeroDivisionError`**

© 2003, 2008 H. Werntges, FB Design Informatik Medien (DCSM), FH Wiesbaden

Exceptions: raise / fail

Auslösen von Ausnahmefehlern mittels raise / fail

```
class Complexnum
  def <=>(z)          # Nur <=> implementieren
    if z.respond_to? :absq # „Duck typing"-Stil
       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)
```


© 2003, 2008 H. Werntges, FB Design Informatik Medien (DCSM), FH Wiesbaden

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]
```


© 2003, 2008 H. Werntges, FB Design Informatik Medien (DCSM), FH Wiesbaden

Exceptions: raise / fail

Mehr zum *Caller Stack* und zu *Tracing*

– Die Schlüsselwörter `__FILE__`, `__LINE__` und (ab Ruby 1.9) `__method__` verraten stets, wo man im Quelltext ist:

Datei „tracing.rb“:

```
#!/usr/bin/env ruby
class MyTrace
  def meth1
    puts "In Datei #{__FILE__}, Zeile #{__LINE__}"
    self.meth2
  end
  def meth2
    fail "Zu tief verschachtelt" if caller.size > 10
    self.meth1          # Endlose Rekursion
  end
end
```

```
MyTrace.new.meth1
… (diverse Ausgaben)
```

– Ausführlicher: Online-Demo „tracing.rb“

© 2003, 2008 H. Werntges, FB Design Informatik Medien (DCSM), FH Wiesbaden

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 beendet das laufende Programm.

© 2003, 2008 H. Werntges, FB Design Informatik Medien (DCSM), FH Wiesbaden

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 case / when: Stopp bei 1. Treffer

  Grundlage des Vergleichs: `$!.kind_of?(parameter)`

© 2003, 2008 H. Werntges, FB Design Informatik Medien (DCSM), FH Wiesbaden

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!

© 2003, 2008 H. Werntges, FB Design Informatik Medien (DCSM), FH Wiesbaden

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
```


© 2003, 2008 H. Werntges, FB Design Informatik Medien (DCSM), FH Wiesbaden

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
```


© 2003, 2008 H. Werntges, FB Design Informatik Medien (DCSM), FH Wiesbaden

Exceptions: Demos

tracing:

Umgang mit dem Caller-Stack, Methode „backtrace“ der Fehlerobjekte und mit den Schlüsselwörtern __FILE__, __LINE__

rescue_eval:

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.

© 2003, 2008 H. Werntges, FB Design Informatik Medien (DCSM), FH Wiesbaden

Exceptions (?): catch & throw

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.

© 2003, 2008 H. Werntges, FB Design Informatik Medien (DCSM), FH Wiesbaden

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.

- Bem.:
  – Bitte nicht mit try … catch von Java verwechseln!

© 2003, 2008 H. Werntges, FB Design Informatik Medien (DCSM), FH Wiesbaden

Ausnahmebehandlung in anderen Skriptsprachen

- **Perl**
  - Keine Exception-Klassen
  - Abbruch mit Kommando „die“:

```
chdir "/usr/spool/news" or die "Can't cd to spool: $!\n";
```

  - Recovery mit eval-Blöcken möglich (und üblich):

```
# Ziel: Kein Abbruch beim Teilen durch Null
# Globale Var. $@ wird bei Fehler im eval-Block gesetzt
eval { $c = $a / $b }; warn $@ if $@;
```

- **Python**
  - Konzeptionell ähnlich zu Ruby: Fehlerklassen, Abfangen von Fehlern
  - Auslösen ebenfalls mittels **`raise`**
  - Abfangen mit Konstruktion **`try … except … finally`**
    statt **`begin … rescue … ensure … end`**
  - Kein Analogon zu **`catch … throw`**

© 2003, 2008 H. Werntges, FB Design Informatik Medien (DCSM), FH Wiesbaden

Exceptions: Ein Python-Beispiel

Einlesen einer (hoffentlich) ganzen Zahl:

```
#!/usr/bin/env python
while True:
  try:
    zahl = int(input("Eine ganze Zahl eingeben:"))
    print "Danke - Zahl", zahl, "angenommen."
    break # Verlässt die Endlosschleife bei Erfolg
  except NameError:
    print "Nur Ziffern eingeben"
  except TypeError:
    print "Nicht in eine ganze Zahl umwandelbar"
  except:
    print "Fehler in Eingabe"
```

Hinweis zur Python-Syntax:

- Doppelpunkte leiten Blöcke ein
- Blöcke müssen eingerückt werden
- Ende der Einrückung = Blockende

© 2003, 2008 H. Werntges, FB Design Informatik Medien (DCSM), FH Wiesbaden