PM1/PT Ruby: Methodenblöcke ausführen

Ziel

- **Ü-7-c-1:** Wir wollen den Vertrag zwischen ***each*** und ***Enumerable*** nutzen, um für die Klasse ***Telefonbuch*** die Iteratormethoden ***inject, select, collect, max, sort*** etc… nutzen zu können.

- Bei solchen Aufgabenstellungen wird man wenn möglich versuchen, ***each*** für die eigene Klasse auf ein ***each*** einer intern enthaltenen Klasse zurückzuführen. In unserem Fall ist das der Hash, der in der Variable ***@eintraege*** referenziert wird.

```
class Telefonbuch

  def initialize(gebiet,vorwahl)
    @gebiet = gebiet
    @vorwahl = vorwahl
    @eintraege= {}
  end

end
```

Iterieren über das Telefonbuch

- Wenn wir über einen Hash mit der Methode ***each*** iterieren, dann brauchen wir
  - **zwei Blockvariablen**, die erste für den ***key*** die zweite für den ***value*** und
  - einen **Block**, der die Anweisungen enthält, die beim Iterieren ausgeführt werden. Der Block kann beliebig komplexe Anweisungen enthalten.

- Wenn wir das ***each*** auf unserem Telefonbuch aufrufen, wollen wir
  - einen Block übergeben können,
  - mit dem wir Auswertungen auf dem Telefonbuch formulieren können: z.B.: alle Namen oder Nummern einsammeln, Namen und / oder Nummern ausgeben, alle Einträge für den Buchstaben ***"A"*** finden etc…

```
tb = Telefonbuch.new("HH", "040")
tb.telefon_nummer_eintragen("Garnicht,
    Erna" , "04055555")
tb.telefon_nummer_eintragen("Albers,
    Bruno" , "04011111")
# Ausgeben
tb.each {|key,val| puts key }
tb.each {|key,val| puts val }
# Namen oder Nummern einsammeln
namen = []
tb.each {|key,val| namen << key }
p namen
nummern = []
tb.each {|key,val| nummern << val }
p nummern
# Namen, die mit "A" beginnen
a_namen = []
tb.each {|key,val|
    if key[0,1] == "A"
      a_namen << key
    end
}
p a_namen
```

Basisiterator für das Telefonbuch

- Dazu müssen wir den **Block**, der unserem Telefonbuch mit dem ***each*** übergeben wird, beim Aufruf von ***each*** an den Hash ***@eintraege*** übergeben.
- Das **Auszeichnen** eines Methodenparameters als **Block** erreichen wir, indem wir dem Parameter ein **&** voranstellen.
- Die Übergabe des Blocks als aktueller Parameter an die ***each***-Methode des Hashes ***@eintraege*** erfolgt dann über den Namen des Blockparameters.
- Rufen wir jetzt ***each*** auf dem Telefonbuch auf, dann wird der Block mit seinen Parametern an ***@eintraege*** weitergereicht.

```
class Telefonbuch

  def each(&block)
    @eintraege.each(&block)
  end

end
```

Aufruf von *each* auf einem Telefonbuch liefert dann die gewünschten Ergebnisse

```ruby
tb = Telefonbuch.new("HH", "040")
tb.telefon_nummer_eintragen("Garnicht,
    Erna" , "04055555")
tb.telefon_nummer_eintragen("Albers,
    Bruno" , "04011111")
# Ausgeben
tb.each {|key,val| puts key }
tb.each {|key,val| puts val }
# Namen oder Nummern einsammeln
namen = []
tb.each {|key,val| namen << key }
p namen
nummern = []
tb.each {|key,val| nummern << val }
p nummern
# Namen, die mit  ?A, "A" beginnen
a_namen = []
tb.each {|key,val|
    if key[0,1]== "A"
        a_namen<< key
    end
}
p a_namen
```

```ruby
def each(&block)
    @eintraege.each {|x,y|
            block.call(x,y)}
end
```

```
Garnicht, Erna
Albers, Bruno
04055555
04011111
```

```
["Garnicht, Erna", "Albers, Bruno"]
```

```
["04055555", "04011111"]
```

```
["Albers, Bruno"]
```

Wie werden Blockparameter ausgewertet?

- Wenn wir den Blockparameter an ***@eintraege*** weiterreichen, stellt sich die Frage, wie dieser Block ausgewertet wird.
- Ein Block wird ausgewertet, indem die Methode ***call*** auf dem Block aufgerufen wird.
- Wir wollen die Auswertung des Blocks bereits im Telefonbuch vornehmen. Dazu iterieren wir über die ***@eintraege*** und rufen den übergebenen Block mit den Blockvariablen (***x,y***) auf.
- Hier sprechen wir den Block ohne das ***&*** an, da wir nur die Anweisungen im Block ohne die Blockvariablen meinen.

```
class Telefonbuch

  def each(&block)
    @eintraege.each {|x,y|
         block.call(x,y)}
  end

end
```

Implizites Adressieren von Blöcken mit *yield*

- Eine zweite Variante, Blöcke auszuwerten, ist das Verwenden von ***yield***.
- In Ruby kann jeder Methode beim Aufruf optional ein Block übergeben werden.
- Der Block muss in der Parameterliste nicht explizit über ***&block*** referenziert werden. In der Definition von ***each*** fehlt jetzt der Blockparameter.
- Um den **impliziten Block** auszuführen, wird in ***each,*** die Methode ***yield*** aufgerufen.
- ***yield*** müssen beim Aufruf die aktuellen Werte für die Blockparameter übergeben werden.

```
class Telefonbuch

  # Aufruf des impliziten Blocks der
  # Methode each
  # Da der each Block zwei
  # Blockvariablen hat, die den key und
  # val Variablen beim Iterieren über
  # @eintraege entsprechen, iterieren
  # wir mit each über @eintraege und
  # übergeben key,val beim Aufruf von
  # yield

  def each()
    @eintraege.each {|key,val|
                    yield(key,val)}
  end

end
```

Parameterübergabe an *yield*

```
tb = Telefonbuch.new("HH", "040")
...
# Aufruf von each mit einem Block
# und den Blockvariablen key, val
tb.each {|key,val| puts key }
```

```
class Telefonbuch
  def each()
    @eintraege.each {|key, val|
                    yield(key, val)}
  end
end
```

- Der Block ***{|x,y| puts key }*** wird in der Definition von ***each*** des Telefonbuchs mit der Anweisung ***yield(key,val***) ausgeführt.
- Das Iterieren über den internen Hash liefert uns die aktuellen Parameter (***key,val***) für ***yield***.
- Also wird für jedes Paar in ***@einträge*** der Block von ***tb.each*** ausgeführt.

block_given?(): Sicheres Ausführen von Blöcken

- Die Methoden ***max, min*** und ***sort*** von ***Enumerable*** können mit und ohne Block aufgerufen werden.
- Das Verhalten der Methoden ändert sich dabei jeweils.
- Eine Methode kann mit ***block_given?()*** prüfen, ob beim Aufruf ein Block übergeben wurde.
- Ein Aufruf von ***yield***, ohne dass ein Block übergeben wurde, erzeugt einen Fehler.
- Die Methode ***ausfuehren_fuer(x)*** prüft daher vor dem Ausführen das Vorhandensein eines Blocks.

```ruby
def ausfuehren_fuer(x)
  if block_given?()
    yield(x)
  else
    x
  end
end

#def ausfuehren_fuer(x)
#   return yield(x)
#end

p ausfuehren_fuer(5)
p ausfuehren_fuer(5){|x| x**3 }
```

Ein weiteres Beispiel: Basisiterator für ein Intervall gerader Zahlen

- Die Klasse ***IntervallGeradeZahlen*** definiert ein geschlossenes Intervall mit geraden Zahlen. Bei der Erzeugung stellen wir sicher, dass untere (***lower***) und obere (***upper***) Grenze gerade sind.
- Wir schreiben den Basisiterator ***each*** für dieses Intervall.
- Hier nutzen wir die Methode ***step*** von ***Range***, um in Zweierschritten in dem Bereich ***lower*** und ***upper*** alle Elemente des Intervalls gerader Zahlen zu berechnen.

```
class IntervallGeradeZahlen

  def initialize(lower,upper)
    if (lower %2 != 0 || upper %2 != 0)
      raise ArgumentError, "Grenzen des
      Intervalls keine gerade Zahlen"
    end
    @lower = lower
    @upper = upper
  end

  def each()
    (@lower..@upper).step(2) {|elem|
        yield(elem)
    }
  end
end
```

Iterieren über das Intervall der geraden Zahlen

- Wir können jetzt ***each*** verwenden
  1. um die geraden Zahlen des Intervalls auszugeben.
  2. um die Summe über die Zahlen des Intervalls zu bilden
  3. f(x) = x-1 zu berechnen, die das Intervall der geraden Zahlen auf ein Array mit ungeraden Zahlen abbildet.
  4. alle gerade Zahlen des Intervalls < 8 zu bestimmen.
  5. etc...

- **Merken Sie was?**

```
igz =
   IntervallGeradeZahlen.new(0,12)
igz.each{|x| puts x}

sum = 0
igz.each {|x| sum += x}
puts "sum #{sum}"

z_ary = []
igz.each {|x| z_ary << (x-1)}
puts "z_ary #{z_ary.inspect}"

z_ary = []
igz.each {|x|  x < 8 ? z_ary << x :
   nil }
puts  "z_ary #{z_ary.inspect}"
```

Iterieren über das Intervall der geraden Zahlen

- Wir können jetzt ***each*** verwenden
  1. um die geraden Zahlen des Intervalls auszugeben.
  2. um die Summe über die Zahlen des Intervalls zu bilden
  3. f(x) = x-1 zu berechnen, die das Intervall der geraden Zahlen auf ein Array mit ungeraden Zahlen abbildet.
  4. alle gerade Zahlen des Intervalls < 8 zu bestimmen.
  5. etc...

- **Merken Sie was?** Wir implementieren unter 2. – 4. Methoden, die wir bereits aus ***Enumerable*** kennen.

- Also warum jetzt nicht den **Vertrag** dicht machen und ***Enumerable*** in die Pflicht nehmen?

```
igz =
   IntervallGeradeZahlen.new(0,12)
igz.each{|x| puts x}

sum = 0
igz.each {|x| sum += x}
puts "sum #{sum}"

z_ary = []
igz.each {|x| z_ary<<(x-1)}
puts "z_ary #{z_ary.inspect}"

z_ary = []
igz.each {|x|  x < 8 ? z_ary << x :
   nil }
puts  "z_ary #{z_ary.inspect}"
```

Den Vertragspartners *Enumerable* einbinden

- Nachdem unsere Klassen ihren Teil des Vertrags erfüllt haben – den Basisiterator ***each*** zu implementieren – wollen wir jetzt die Leistungen des Vertragspartners ***Enumerable*** in Anspruch nehmen.
- Dazu binden wir die ***Enumerable*** mit Hilfe von ***include*** in unsere Klassen ein.
- **That's it:** Jetzt können wir die Berechnungen für das Intervall gerader Zahlen der vorausgehenden Seite mit Methoden von ***Enumerable*** kompakter aufschreiben.

```
class IntervallGeradeZahlen
  include Enumerable
  ... # Rest bleibt gleich
end

# Umschreiben mit Methoden von
# Enumerable
sum = igz.reduce {|sum,x| sum + x}
puts "sum #{sum}"

z_ary = igz.map {|x| x - 1}
puts "z_ary #{z_ary.inspect}"

z_ary = igz.select {|x| x < 8}
puts  "z_ary #{z_ary.inspect}"
```

Übungen

- **Ü-7-c-2:** Binden Sie bitte ***Enumerable*** in die Klasse ***Telefonbuch*** ein! Berechnen bitte Sie für das Telefonbuch:
  1. den längsten Namen (***max***)
  2. den größten Namen (***max***)
  3. ein Array der Namen (***map***)
  4. ein Array der Nummern (***map***)
  5. alle Einträge, deren Nummern eine 1 enthalten (***select***)
  6. alle Einträge, deren Nummer nicht mit der Vorwahl beginnen (***select***)
  7. ein Array mit 2-elementigen Arrays, in der die Nummern nach Vorwahl und Durchwahl zerlegt sind. (***map***)
  8. ein Array mit 2-elementigen Arrays, in dem der Name nach Nachname und Vorname zerlegt ist. (***map***)
  9. eine Zeichenkette aller Namen (***reduce***)

Ü-7-c-3: Implementieren Sie bitte die Methoden ***map, reduce, select, max, min*** nur unter Verwendung von ***each***! Beachten Sie dass, ***max*** und ***min*** mit und ohne Block aufgerufen werden können.

Verbesserung der Lösung für Intervalle gerader Zahlen

- Wenn wir das Intervall gerader Zahlen wie gezeigt modellieren, dann müssen wir auch für ungerade Zahlen eine eigenes Intervall definieren. Wollen wir diese Art von Intervall verallgemeinern, so müssten wir für jedes Intervall, das Vielfache einer ganzen Zahl enthält, eine separate Klasse erschaffen.

- Für jede dieser Intervallklassen müssten wir dann die Funktionalität von Intervallen in Ruby neu implementieren. Das ist sehr arbeits- und wartungsaufwändig.

- Daher wollen wir Klassen schaffen, die gerade, ungerade und Vielfache ganzer Zahlen modellieren.

- Diese Klassen wollen wir so präparieren, dass sie als Elemente in einem Intervall teilnehmen können.

- Dazu müssen wir den Vertrag zwischen einem Intervall (***Range***) und den enthaltenen Elementen verstehen und für unsere Klassen umsetzen.

- **Wo wollen wir hin?** Zum Beispiel Intervalle für gerade Zahlen und ungerade Zahlen wie folgt aufschreiben können und dann als Intervalle nutzen können.

  (GeradeZahl(0)..GeradeZahl(12))
  (UngeradeZahl(1)..UngeradeZahl(13))
  oder allgemeiner
  (VielfacheVon(3,0)..VielfacheVon(3,12))

Der Vertrag zwischen *Range* und enthaltenen Elementen

- Grundsätzlich können Objekte jeder Klasse die Intervallgrenzen eines Intervalls bilden.

- ***Range*** fordert von diesen Klassen, dass sie
  1. Die Methode **succ** implementieren, die zu einem Objekt den Nachfolger berechnet
  2. Die Methode **<=>** implementieren, um Objekte miteinander vergleichen zu können.

- Wir definieren eine Klasse ***GeradeZahl***, die nur die Erzeugung gerader Zahlen zulässt.

- Wir merken uns die ganze Zahl, mit der die gerade Zahl erzeugt wurde in ***@num***.

- **Nachfolger** einer geraden Zahl (Methode ***succ***) ist eine gerade Zahl, deren Wert um 2 größer als ***@num*** der ursprüngliche Zahl.

- Der **Vergleich** (<=>) zweier gerader Zahlen lässt sich auf den Vergleich deren ***@num*** Werte zurückführen.

Präparieren von *GeradeZahl* als Intervallgrenze

- Wir definieren eine Klasse ***GeradeZahl***, die nur die Erzeugung gerader Zahlen zulässt.
- Wir merken uns die ganze Zahl, mit der die gerade Zahl erzeugt wurde in ***@num***.
- **Nachfolger** einer geraden Zahl (Methode ***succ***) ist eine gerade Zahl, deren Wert um 2 größer als ***@num*** der ursprüngliche Zahl.
- Der **Vergleich** (<=>) zweier gerader Zahlen lässt sich auf den Vergleich deren ***@num*** Werte zurückführen.

```
class GeradeZahl
  attr_reader :num
  def initialize(num)
    raise ArgumentError, "#{num}
   ungerade Zahl" if (num%2 != 0)
    @num = num
  end

  def succ()
     GeradeZahl.new(@num+2)
  end

  def <=>(other)
    @num <=> other.num
  end

  def to_s()
    "#{num}"
  end
end
```

Präparieren von *GeradeZahl* als Intervallgrenze

- **That's it:** Jetzt können wir mit ***GeradeZahl*** Intervalle definieren und wie gewohnt mit den Intervallen rechnen.

```
i_gerade = GeradeZahl.new(-4)..GeradeZahl.new(12)
i_gerade.each {|x| puts x }
p("include?(GeradeZahl.new(0)) #{i_gerade.include?(GeradeZahl.new(0))}")
p "first #{i_gerade.first()}"
p "end #{i_gerade.end()}"
p "collect #{i_gerade.collect {|x| x.num}.inspect()} "
p "inject #{i_gerade.inject(0) {|acc,x| acc + x.num}}"
p "step(5)"
i_gerade.step(5) {|x| puts x }
```

```
i_gerade =
    GeradeZahl.new(-4)..GeradeZahl.new(12)
i_gerade.each {|x| puts x }
```

```
-4
-2
0
2
4
6
8
10
12
```

```
P "include?(GeradeZahl.new(0))
   #{i_gerade.include?(GeradeZahl.new(0))}"
p "first #{i_gerade.first()}"
p "end #{i_gerade.end}"
p "map #{i_gerade.collect {|x|
   x.num}.inspect} "
p "reduce #{i_gerade.inject(0) {|acc,x| acc
   + x.num}}"
p "step(5)"
i_gerade.step(5) {|x| puts x }
```

```
"include?(GeradeZahl.new(0))
   true"

"first -4"
"end 12"
"collect [-4, -2, 0, 2, 4, 6, 8,
   10, 12] "
"inject 36"

"step(5)"
-4
6
```

Übungen

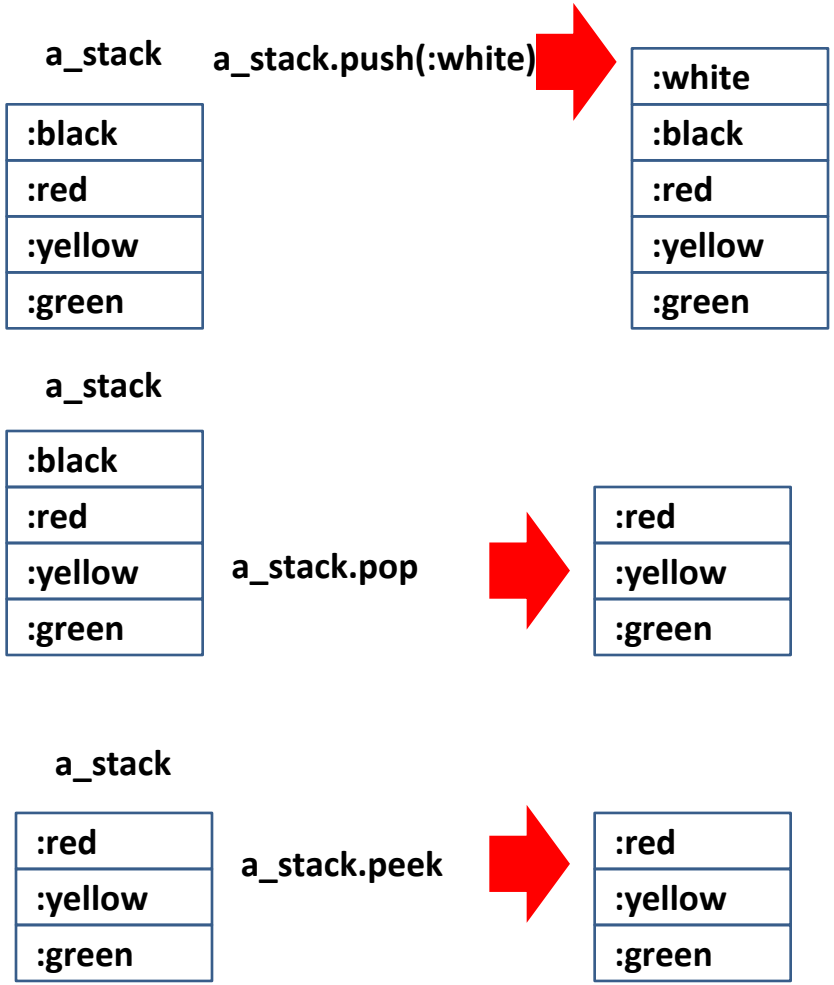
- **Ü-7-c-3:** Implementieren Sie bitte die Klassen ***UngeradeZahl*** und ***VielfachesVon***, so dass sie Objekte dieser Klassen als Intervallgrenzen nutzen können! ***VielfachesVon*** wird beim Erzeugen der Teiler und ein Startwert mitgegeben.

- **Ü-7-c-3:** Testen Sie bitte Ihre Implementierungen mit einem ähnlichen Script wie das für ***GeradeZahlen***!

Ein letztes Beispiel: Der Objektstapel (*Stack*)

- Ein ***Stack*** ist eine Objektsammlung, in der Sie Elemente aufeinanderstapeln können und Elemente wieder vom Stapel nehmen können. Sie können immer nur das oberste Element vom Stapel nehmen oder lesen. Ein ***Stack*** ist also eine Datenstruktur, die die Reihenfolge der Elemente beibehält und die Elemente in der umgekehrten Reihenfolge, in der diese auf den Stapel gelegt wurden, wieder ausgibt.

  Elemente werden mit ***push(elem)*** auf den Stapel gelegt. Elemente werden mit ***pop()*** vom Stapel genommen. ***pop()*** liefert das oberste Element zurück. Um das oberste Element nur zu lesen, es aber nicht vom Stapel zu entfernen hat ein Stapel die Methode ***peek()***. Um nachzuschauen, ob ein Stapel leer ist, hat der Stapel die Methode ***empty?()***

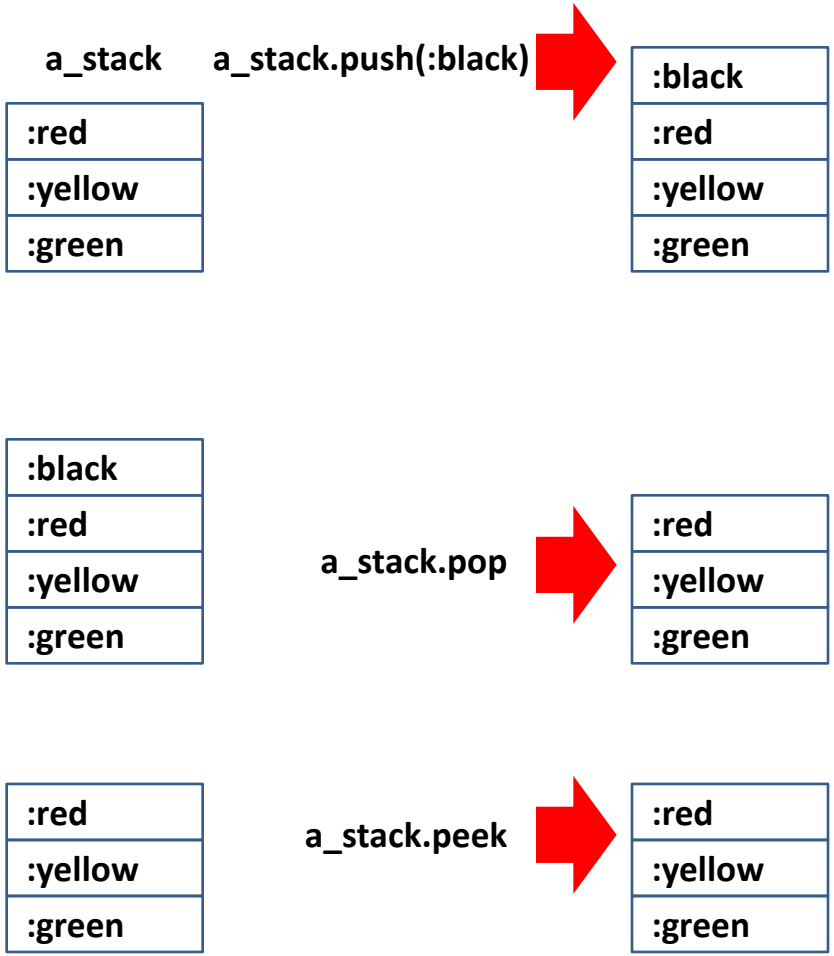


Stack - eine Objektsammlung mit besonderem Verhalten

- **Ü-7-c-4**: Schreiben Sie bitte eine Klasse für die Datenstruktur ***Stack*** mit den Methoden ***push(elem), pop(), peek()*** und ***empty?()***! Verwenden Sie für die Implementierung ein ***Array***.

- **Ü-7-c-5**: Erzeugen Sie bitte den nebenstehenden ***Stack a_stack*** mit drei Elementen. Verwenden Sie dazu nur die Methode ***push.***

- **Ü-7-c-6**: Schreiben Sie bitte den Basisiterator ***each*** für Ihre Klasse ***Stack***. Stellen Sie die Vertragsbeziehung zu ***Enumerable*** her und zeigen Sie, wie Ihre ***Stack***-Objekte von diesem Vertrag profitieren.



Zusammenfassung

- Jeder Methode kann in Ruby beim Aufruf ein **Block** übergeben werden.
- Blöcke enthalten Anweisungen, die in der Methode durch ***block.call(…)*** oder ***yield(…)*** **ausgeführt** werden können. Die Anzahl der **Parameter**, die in ***call*** oder ***yield*** übergeben werden müssen, ergeben sich aus der Anzahl der **Blockparameter**.
- Das **Vorhandensein** eines Blocks lässt sich mit ***block_given?()*** prüfen.
- Alternativ können Blöcke auch an andere Methoden **weitergereicht** werden. Dazu müssen Blöcke in der Definition einer Methode als besonderer Parameter (mit vorangestellten ***&***) gekennzeichnet werden.
- Nahezu alle Methoden von ***Enumerable*** nutzen den Basisiterator ***each***. Die Implementierungen nutzen die **Übergabe von Blöcken**.
- Wie ***Enumerable*** definiert ***Range*** einen Vertrag für andere Klassen. **Objekte** dieser Klassen müssen vergleichbar sein (<=>) und die Methode ***succ*** implementieren, um als **Intervallgrenzen** eines Ranges verwendet werden zu können.