PM1 / PT Ruby

Kontrollstrukturen

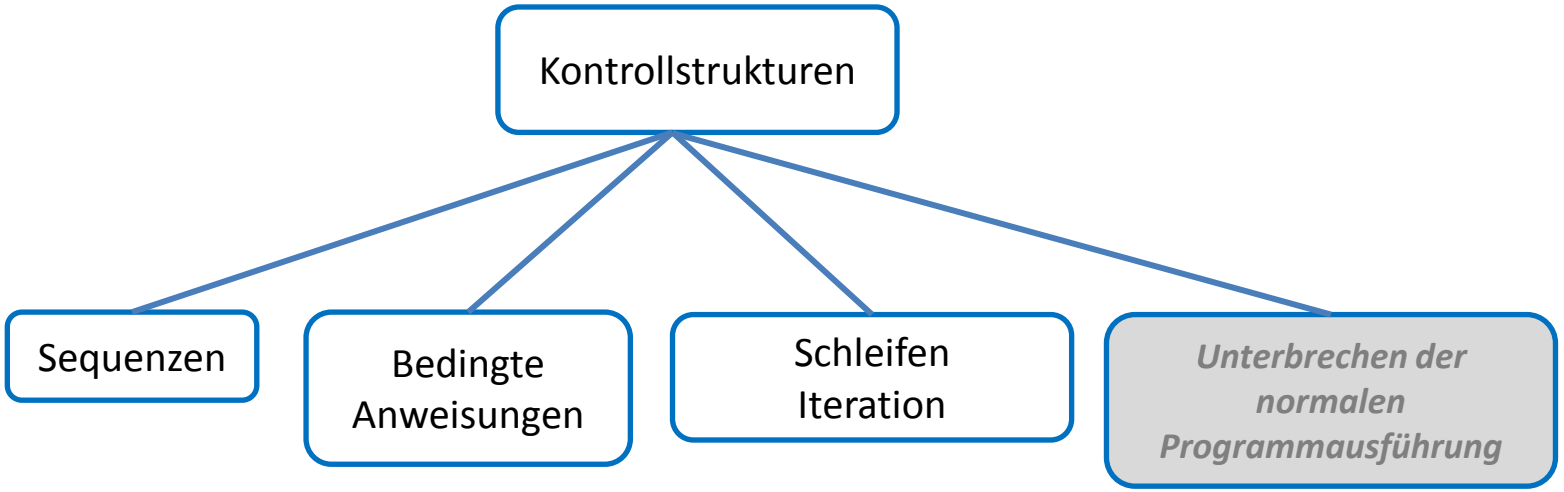
Logische Ausdrücke

Konzepte

- Kontrollstrukturen
- Anweisungen und Ausdrücke
- Logische Ausdrücke

Einleitung

- Objekte erledigen ihre Aufgabe, indem sie Dienstleistungen anderer Objekte in Anspruch nehmen.
- Dazu rufen Objekten Methoden auf anderen Objekten auf und verarbeiten die Ergebnisse
- Häufig müssen Objekte, um ihre Aufgabe zu erfüllen, Dienstleistungen mehrerer Objekte kombinieren. Dazu benötigen sie **Kontrollstrukturen,** die den Ablauf eines Programmes steuern.



Sequenzen

- Sequenzen sind Abfolgen von Anweisungen, die nacheinander abgearbeitet werden.
  - werden entweder als Block dargestellt, dann sind diese in **{}** oder ***begin … end*** eingeschlossen.
  - oder sind in einem Methodenblock eingeschlossen ***def … end*** .
  - oder in anderen Kontrollstrukturen wie ***if, while …*** und ***end*** eingeschlossen .
  - oder sind implizit gegeben durch das umgebende Rubyscript .

```ruby
def ticket_drucken()
    puts "------------------"
    puts "- HAW Express-Line"
    puts "- Ticket"
    puts "- " + bisher_gezahlt.to_s +
       " Cent"
    puts "------------------"
    # die Gesamtsumme, mit dem der
    #Automat nach der letzten Leerung
    #gefüttert wurde
    @gesamt_summe =
       @gesamt_summe + @bisher_bezahlt
    @bisher_bezahlt = 0
end
```

Bedingte Anweisungen

- der Programmfluss hängt von einer Bedingung ab, die als ***boolescher Ausdruck*** formuliert ist

```
x = 13
puts("if")
if x < 10
  puts "#{x} ist kleiner als 10"
end

Puts "if else"
if x < 10
  puts "#{x} ist kleiner als 10"
else
  puts "#{x} ist größer gleich 10"
end
```

Bedingungen Anweisungen Ausdrücke

- **Ausdrücke** sind Konstrukte in einer Sprache, die einen Ergebniswert zurückliefern.
- Bedingungen sind **logische Ausdrücke**. Die liefern wahr (true) oder falsch (false) zurück.
- **Anweisungen** sind Zuweisungen, Methodenaufrufe, Objekterzeugungen oder Kontrollstrukturen.
- Alle Anweisungen in Ruby haben einen Wert. Daher sind Anweisungen auch immer **Ausdrücke**.
- In Ruby sind beliebige Ausdrücke in der Bedingung zugelassen, da alles, was ungleich ***false*** und ***nil*** ist, ***true*** ergibt

Bedingte Anweisungen

- Die Syntax für **einfache bedingte Anweisungen**:

  if [(]<cond>[)]

  <statements>

  end

- Die Syntax für **bedingte Anweisungen mit Alternative**

  if [(]<cond>[)]

  <statements>

  else

  <statements>

  end

```
x = 13
puts "if"
if x < 10
  puts "#{x} ist kleiner als 10"
end

puts "if else"
if x < 10
  puts "#{x} ist kleiner als 10"
else
  puts "#{x} ist größer gleich 10"
end
```

Bedingte Anweisungen mit negativer Bedingungsprüfung

- ***unless:*** wenn die Bedingung nicht erfüllt ist, werden die Anweisungen ausgeführt.

```
unless [([]<cond>[])]
    <statements>
end
```

```
x = 13
puts "unless"
unless (x >= 10)
  puts "#{x} ist kleiner als 10"
end
```

- Die Syntax für **bedingte Anweisungen mit Alternative mit negativer Bedingungs-prüfung**

```
unless [([]<cond>[])]
        <statements>
else
        <statements>
end
```

```
puts "unless else"
unless (x >= 10)
  puts "#{x} ist kleiner als 10"
else
  puts "#{x} ist größer gleich 10"
end
```

Geschachtelte *if-else* Anweisung

- Manchmal sind mehrere Bedingungen zu prüfen.
- **Beispiel**: Die Übersetzung des vom Benutzer eingegebenen Farbnamens in einen Farbnamen für ein Graphikobjekt (*rot* -> *:red* etc..) aus dem Projekt Geometrie in den Methoden *farbe_aendern*
- Mit *if-else* Anweisungen würden wir diese Mehrfachprüfungen wie rechtsstehend aufschreiben. Wenn die eingegebene Farbe nicht bekannt ist, dann soll die Farbe auf *:rot* gesetzt werden.

```
if  neue_farbe == "gruen"
  @farbe = :green
    else
      if  neue_farbe == "rot"
        @farbe = :red
      else
        if neue_farbe == "blau"
          @farbe = :blue
        else
          if neue_farbe == "gelb"
            @farbe = :yellow
          else
            if neue_farbe == "weiss"
              @farbe = :white
            else
              if neue_farbe == "schwarz"
                @farbe = :black
              else
                @farbe = :red
              end
            end
          end
        end
    end
end
```

if-Kaskaden als Alternative zu geschachteltem *if-else*

- Um zu tief verschachtelte ***if-else*** Anweisungen zu vermeiden, gibt es die ***if-*** Kaskaden.

- Es werden nacheinander die Bedingungen im ***if*** und den ***elsif*** Zweigen ausgewertet. Wenn eine der Bedingungen im ***if*** oder ***elsif*** wahr wird, dann werden die nachfolgenden Anweisungen bis zum nächsten ***elsif*** oder ***else*** ausgeführt.

- Danach wird nach dem ***end*** fortgefahren.

- Wird keine der Bedingungen wahr, wird die Anweisung im **else** Zweig ausgeführt.

```
if neue_farbe == "gruen"
  @farbe = :green
elsif neue_farbe == "rot"
  @farbe = :red
elsif neue_farbe == "blau"
  @farbe = :blue
elsif neue_farbe == "gelb"
  @farbe = :yellow
elsif neue_farbe == "weiss"
  @farbe = :white
elsif neue_farbe == "schwarz"
  @farbe = :black
else
  @farbe = :red
end
```

Exkurs: Syntaktische Varianten von *if* und *unless*

- Jeder **If**-Ausdruck hat mehrere syntaktische Formen:
  - Mehrzeilen-Schreibweise mit **then**
  - Mehrzeilen-Schreibweise ohne **then**
  - Einzeilen-Schreibweise mit **then**
- **If**-Ausdrücke definieren **keinen** eigenen Block. Lokale Variablen der Umgebung (im folgenden Beispiel **handle**) werden modifiziert.

Exkurs: Syntaktische Varianten von *if* und *unless*

```ruby
artist = "Gillespie"

# Mehrzeilenschreibe ohne then

if artist == "Gillespie"
  handle = "Dizzy"
elsif artist == "Parker"
  handle = "Bird"
else
  handle = "unknown"
end

puts handle      #=> Dizzy
```

```ruby
artist = "Gillespie"

# Mehrzeilenschreibe mit then

if artist == "Gillespie" then
  handle = "Dizzy"
elsif artist == "Parker" then
  handle = "Bird"
else
  handle = "unknown"
end

puts handle      #=> Dizzy
```

Exkurs: Syntaktische Varianten von *if* und *unless*

```
artist = "Gillespie"
# Einzeilenschreibweise mit then
if artist == "Gillespie" then handle = "Dizzy"
elsif artist == "Parker" then handle = "Bird"
else handle = "unknown"
end
puts handle      #=> Dizzy
```

Der **ternäre Ausdruck** als Kurzform für *if-else* / *unless-else*

- **Syntax**: ***<cond>*** **?** ***<expr1>*** **:** ***<expr2>***
- **Auswertung**: wenn ***<cond>*** wahr wird, dann ist ***<expr1>*** das Ergebnis des Ausdrucks sonst **<expr2>**

```
if x < 10
  "#{x} ist kleiner als 10."
else
  "#{x} ist größer gleich 10."
end
# äquivalent zu if-else
x<10 ? "#{x} ist kleiner als 10." :  "#{x} ist größer gleich 10."

unless x >= 10
  "#{x} ist kleiner als 10."
else
  "#{x} ist größer gleich 10."
end
# äquivalent zu unless-else
!(x >=10) ?  "#{x} ist kleiner als 10." : "#{x} ist größer gleich 10."
```

Ternäre Ausdrücke und alle *if*-Anweisungen haben einen Wert

- Der Wert einer ***if, if-else, if – elsif –else*** Anweisung ist der Wert der Anweisung dessen Bedingung wahr wird. (Gleiches gilt für ***unless***)
- Der Wert eines ternären Ausdrucks ist ***expr1***, wenn die Bedingung wahr wird, sonst ***expr2***

```
x = 8
# s speichert den Wert der if-else Anweisung
s = if x < 10
  "#{x} ist eine Ziffer"
else
  "#{x} ist keine Ziffer"
end
Puts s                          # => "8 ist eine Ziffer„

# s speichert den Wert des ternären Ausdrucks
s = (x<10) ? "#{x} ist eine Ziffer" : "#{x} ist keine Ziffer"
puts s                          # => "8 ist eine Ziffer"
```

Case Ausdrücke

- Zwei syntaktische Formen.
- ***case*** direkt gefolgt
  - von einer Liste von ***when*** Anweisungen, in denen logische Ausdrücke stehen, gefolgt von Anweisung nach dem ***:***
  - und einer **else** Klausel, die immer ausgewertet wird, wenn alle Bedingungen in den ***when*** Zweigen nicht erfüllt sind.
- **case *<target>*** gefolgt
  - von einer Liste von **when** Klauseln, die Objekte listen, gefolgt von Anweisungen
  - und einer **else** Klausel, die immer ausgewertet wird, wenn alle Bedingungen in den ***when*** Zweigen nicht erfüllt sind.

```
require 'Date'
jahr = Date.today().year
schaltjahr = case
  when jahr % 400 == 0 then true
  when jahr % 100 == 0 then false
  else jahr % 4 == 0
  end
puts schaltjahr

neue_farbe = "ROT"
case neue_farbe
when "rot", "Rot", "ROT"
  farbe = :red
when "blau"
  farbe = :blue
when "gelb"
  farbe = :yellow
# ....
else
  farbe = :red
end
puts(farbe)
```

Auswertung von *case <target>*

- In ***case <target>*** ***when <compare>*** kann ***<compare>*** ein einzelnes Objekt oder eine Aufzählung von Objekten sein.
- Ist ***<compare>*** ein einzelnes Objekt, dann wird ***<compare>*** mittels ***===*** mit ***<target>*** verglichen.
- **Methode**: ***<compare>.===(<target>)***
- Im Beispiel rechts:

```
"blau".===(neue_farbe)
ist das, was im 2. „when“
ausgeführt wird.
```

```ruby
neue_farbe = "ROT"
case neue_farbe
when "rot", "Rot", "ROT"
  farbe = :red
when "blau"
  farbe = :blue
when "gelb"
  farbe = :yellow
# ....
else
  farbe = :red
end
puts farbe
```

Auswertung von *case <target>*

- Ist ***<compare>*** eine Aufzählung von Objekten ***<comp1>, <comp2> …,*** dann werden ***<comp1>*** etc.. einzeln mit ***<target>*** mittels ***===*** verglichen und die Einzelvergleiche mit oder verknüpft.

- ***<comp1>===<target>) ||***
  (<comp2> ===<target>

- Im Beispiel des ersten „when“:

```
"rot" === neue_farbe ||
"Rot" === neue_farbe ||
"ROT" === neue_farbe
```

- Objekte, die in **case *target*** Ausdrücken als ***comparison*** auftreten müssen die Methode **===** implementieren.

- **In Object** ist **===** als **==** definiert

```
neue_farbe = "ROT"
case neue_farbe
when "rot", "Rot", "ROT"
  @farbe = :red
when "blau"
  @farbe = :blue
when "gelb"
  @farbe = :yellow
# ....
else
  @farbe = :red
end
puts @farbe
```

Ein Vorgriff: das Pärchen *case <target>* und *===*

- Objekte, die in **case *<target>*** Ausdrücken als ***<comparison>*** auftreten wollen, müssen die Methode **===** implementieren.
- In ***Object*** ist ***===*** als ***==*** definiert.
- In ***Range*** ist **===** als ***member?*** bzw. ***include?*** definiert.
- Für ***Class*** Objekte wird mit ***===(an_obj)*** geprüft, ob ***an_obj*** von Typ der Klasse ist:
- Eigene Klassen können ***===*** überschreiben, wenn die Definition in ***Object*** nicht passend ist.

```
monat = Date.today().month()
quartal = case monat
  when 1..3 then 'erstes Quartal'
  when 4..6 then 'zweites Quartal'
  when 7..9 then 'drittes Quartal'
  else 'viertes Quartal'
  end
```

```
figur = Kreis.new()
s= case figur
when Kreis
   "ein Kreis #{figur}"
when Rechteck
    "ein Rechtek #{figur}"
when Dreieck
    "ein Dreieck #{figur}"
end
```

Ü3-b 1:

- Schreiben Sie bitte ein Programm, das die englischen Begriffe für Jahreszeiten ***spring, summer, autumn, winter*** ins Deutsche übersetzt. (Fruehjahr, Sommer, Herbst, Winter)

1. als geschachteltes ***if-else***
2. als ***if*** Kaskade
3. mit ***case-target***
4. mit ***case*** ohne ***target***

Ü3-b 2: Übersetzen Sie bitte die *if-else* Schachtelung in die ternäre Schreibweise!

```
# Vorlage fuer den ternären Op
farbe = :green
puts (if farbe==:red
  "rot"
  else
      if farbe==:green
        "gruen"
      else
        if farbe==:blue
          "blau"
        else
          "kein rgb"
        end
      end
end)
```

Boolesche Ausdrücke

- Boolesche Ausdrücke kombinieren boolesche Werte (Wahrheitswerte) mit logischen (booleschen) Operatoren.
- Wahrheitswerte sind in Programmier-sprachen **true** (für wahr) und **false** (für nicht wahr).
- Jeder boolesche Ausdruck hat einen Wahrheitswert.
- Die Wahrheitswerte boolescher Ausdrücke werden in Wahrheitstabellen der booleschen Operatoren festgelegt

Definition boolescher Ausdrücke

- Jede Variable mit einem booleschen Wert ist ein boolescher Ausdruck.
- **true** und **false** sind boolesche Ausdrücke.
- Wenn **A** und **B** boolesche Ausdrücke sind, dann sind
  - **(A==B), (A.eql?(B)), (A.equal?(B)), (A===B),(A<B), (A<=B), (A>B), (A>=B)** boolesche Ausdrücke
- Wenn **B** ein boolescher Ausdruck ist, dann ist auch **!B** oder **not B** ein boolescher Ausdruck
- Wenn **A** und **B** boolesche Ausdrücke sind, dann sind auch **(A&&B), (A and B), (A || B), (A or B)** boolesche Ausdrücke.
- **In Ruby gilt zusätzlich**:
  - Jeder Ausdruck und jede Anweisung liefert einen Wert.
  - Jeder Wert ungleich ***nil*** oder ***false*** ist ***true***, ***nil*** ist ***false***
  - ➔ Jeder Ausdruck und jede Anweisung liefert einen booleschen Wert
  - Verwenden Sie bitte &&, ||, ! und **nicht** „and“, „or“, „not“

Vergleichsmethoden

Methode	Beschreibung
==	Gleichheit
eql?	Gleicher Wert und gleicher Typ
equal?	Identität
===	Vergleichsmethode für *case*
<=>	allgemeiner Vergleichsmethode auch ***Spaceship***
<, <=, >=, >	Vergleichsoperatoren werden auf <=> zurückgeführt

- ***==*** prüft auf gleiche Werte, auf Inhaltsgleichheit. ***!=*** Negation
- ***eql?***: wird wahr, wenn Empfänger und Argument den gleichen Typ und den gleichen Wert/Inhalt haben:
  → ***1 == 1.0*** ist wahr ***1.eql?(1.0)*** liefert falsch
- ***equal?*** Wird wahr, wenn Empfänger und Argument identisch sind.
- ***<=>*** allgemeiner Vergleichsoperator. Liefert -1,0, +1, wenn der Empfänger kleiner, == oder größer dem Argument ist.
- ***===*** wird in ***case*** Anweisungen auf die Objekte in den ***when*** Klauseln angewendet → ***case*** Anweisung

Bedeutung und Auswertung boolescher Operatoren

Operatoren und Bedeutung

- ***&& - and*** wird wahr:
  wenn erster und zweiter Operand wahr werden.
- ***|| - or*** wird wahr:
  sobald einer der Operanden wahr wird
- ***! -not*** liefert den gegenteiligen Wahrheitswert des Operanden

Auswertung

- boolesche Ausdrücke werden von links nach rechts unter Berücksichtigung der Präzedenzregeln (! vor && vor ||) ausgewertet.
- **Partielle Auswertung**: Sobald der Wahrheitswert eines booleschen Ausdrucks eindeutig entschieden werden kann, wird die weitere Auswertung des Ausdrucks abgebrochen.

Wahrheitstabellen boolescher Operatoren

- Wahrheitstabellen legen für boolesche Operatoren das Ergebnis der Verknüpfung von booleschen Werten fest.

&& bzw. and	**true**	**false**
true	**true**	**false**
false	**false**	**false**

|| bzw. or	**true**	**false**
true	**true**	**true**
false	**true**	**false**

! bzw. not	
true	**false**
false	**true**

Ü3-b 3: Welche Ausgaben erzeugt das Script?

```
puts ("Logische  Ausdruecke")
x0=1
x1=3
x2=5
x3=10
x4=-1
puts "x0=#{x0} x1=#{x1} x2=#{x2} x3=#{x3} x4=#{x4}"

puts x1 <= x0
puts x1 <= x0 || x4 <= x0
puts x1 > x0
puts x1 <= x2 &&  x1 >= x0
puts x1 >= x2
puts x1 >= x2 && x3 >= x2
puts (x1 >= x2 || x3 >= x2) && ( x1<= x0 || x4 <= x0)
```

Ü3-b 4: Welche Ausgaben erzeugt das Script?

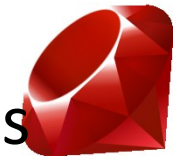
```
puts true && true
puts true and false
puts  false || true
puts false or false

puts

puts true && "eins"
puts false || "zwei"
puts true || "drei"

puts
vier = "vier"
sechs = "sechs"
puts !true || ! vier
puts !true || "fünf"
puts !false && (!sechs)
puts !false || "sieben"
puts !true || "acht"
```

Ü 3-b 4.2: Welche Ausgaben erzeugt das Script? Was drücken die booleschen Ausdrücke aus?



```
x0=1
x1=3
x2=5
x3=10
x4=-1
puts "x0=#{x0} x1=#{x1} x2=#{x2} x3=#{x3} x4=#{x4}„

puts x1 <= x0 && "#{x1} <= #{x0}"
puts x1 <= x0 || "#{x1} > #{x0}"
puts x1 == x0 and "#{x1} != #{x0}"
puts x1 == x0 or "#{x1} != #{x0}"

puts("Intervalpruefung fuer Intervall [#{x0},#{x2}]")
puts (x1 >= x0 && x1 <= x2) and "#{x0} <= #{x1} && #{x1} <= #{x2}"
puts (x1 >= x0 && x1 <= x2) or "#{x0} <= #{x1} && #{x1} <= #{x2}"
puts (x4 <= x0 || x4 >= x2) and "#{x4} <= #{x0} || #{x4} >= #{x1}"
puts (x4 <= x0 || x4 >= x2) or "#{x4} <= #{x0} || #{x4} >= #{x1}"
puts (x3 <= x0 || x3 >= x2) and  "#{x3} <= #{x0} || #{x3} >= #{x1}"
puts (x3 <= x0 || x3 >= x2) or "#{x3} <= #{x0} || #{x3} >= #{x1}"
```

Alle Ausdrücke ungleich *nil* oder *false* sind *true.*
Alle Ausdrücke gleich *nil* sind *false*
Ü 3-b 5: Welche Ausgaben erzeugt das folgende Script?

```
puts "Alle Ausdrücke ungleich nil oder false sind true"
ein_kreis = Kreis.new()
puts ein_kreis && ein_kreis.farbe()
i = 34
puts i && "Zahl #{i} hat den Wahrheitswert true"
puts (not puts) && "puts liefert nil, not nil liefert true"

puts("Alle Ausdrücke gleich nil oder false sind false")
ein_dreieck = nil
puts ein_dreieck || "Variable 'ein_dreieck' nicht definiert"
puts nil || "wird immer ausgegeben"
puts false || "wird immer ausgegeben"
puts (puts ein_kreis) || "Wert von puts ist immer nil"
```

Schleifen und Iteratoren

- **Kontrollstrukuren für Schleifen**: eine Folge von Anweisungen wird mehrmals wiederholt, bis eine Abbruchbedingung erreicht ist.

  while <cond>
  <statements>
  end
  until <cond>
  <statements>
  end

- **Iteratoren:** eine Folge von Anweisungen wird mehrfach wiederholt. Die Wiederholung wird durch Eigenschaften von Objekten gesteuert. (***später***)

```ruby
# while - until

a = 1
while a < 100
 a = a*2
end
puts a.to_s

until a < 100
  a = a - 10
end
puts a.to_s

while line = file.gets
  puts(line)
end
```

Schleifen mit *while* und *until*

- **while** führt die *<statements>* solange aus, wie <*cond*> noch erfüllt ist.
- **until** führt die *<statements>* solange aus, bis *<cond>* das erste Mal erfüllt ist.
- Wie arbeitet das Programm rechts?

```
a = 1
while a < 100
 a = a*2
end
puts a.to_s

until a < 100
  a = a - 10
end
puts a.to_s
```

Nachprüfende Schleifen

- In den Beispielen zu ***while*** und ***until*** werden die Bedingungen im Kopf der Schleife geprüft. Diese Schleifen heißen daher auch **vorprüfende** Schleifen.
- Zu diesen gibt es die **nachprüfenden** Schleifen, bei denen der Bedingungsteil einem ***begin … end*** Block nachgestellt wird.
- Der Unterschied zu den vorprüfenden Schleifen ist, dass die Anweisungen im Block immer mindestens einmal durchlaufen werden.

```
a =60
begin
  a = a*2
end while a < 100
puts a.to_s

a = 90
begin
    a = a -10
end until a < 100
puts a.to_s
```

Zählschleifen sind Iteratoren

- Zählschleifen sind Schleifen, die das Erhöhen oder Erniedrigen einer Zählvariable selbständig übernehmen.
- In Ruby sind alle Zählschleifen mittels Iteratoren realisiert.
- Iteratoren sind Objektmethoden, die sukzessive die Einzelkomponenten eines Objektes zurückliefern.
- Iteratoren für ganze Zahlen liefern Zahlen, in der für sie definierten Ordnung zurück.
- Drei Beispiele vorab, die alle die Zahlen 0 bis 7 ausgeben.

```ruby
for i in 0..7
  p(i)
end
```

```ruby
7.times do |n|
  p(n)
end
```

```ruby
0.upto(7) {|k|
  p(k)
}
```

Ü3-b 6:

- Schreiben Sie bitte eine Methode, die die ersten ***n*** positiven natürlichen Zahlen solange summiert wie die Gesamtsumme kleiner oder gleich 100 ist! Geben Sie bitte das größte ***n*** zurück, für das die Summe noch kleiner oder gleich 100 ist.

- Schreiben Sie bitte eine Methode, die die Summe der ersten ***n***-positiven natürlichen Zahlen berechnet. Die Methode hat einen Parameter für dieses ***n.*** Die Methode macht eine Ausgabe auf die Konsole und gibt ***-1*** zurück, wenn ***n*** nicht positiv ist.

Exkurs: „Statement Modifier“

- Für ***if, unless, while*** und ***until*** gibt es in Ruby Kurzschreibweisen.
- Der Bedingungsteil dieser Kontrollstrukturen darf nach dem Anweisungsteil stehen, wenn der gesamte Ausdruck in eine Zeile passt.
- Der Anweisungsteil wird hier nur ausgewertet, wenn die Bedingung erfüllt ist.
- Das ist „***syntactic sugar***“ und „***never ever***“ bei mir prüfungsrelevant

```
x = 8
puts("#{x} ist eine Ziffer") if x < 10
puts("#{x} ist eine Ziffer") unless x >= 10
a = 1
a = a*2 while a < 100
puts a
a = a - 10 until a < 100
```

Zusammenfassung

- **Kontrollstrukturen** ermöglichen es, den sequentiellen Programmfluss zu modifizieren. Wir unterscheiden Sequenzen, bedingte Anweisungen, Schleifen und Strukturen, die den normalen Programmablauf in Schleifen beeinflussen.
- In bedingten Anweisungen und Schleifen steuern **boolesche Ausdrücke** den Programmfluss.
- **Zählschleifen** sind Iteratoren.
- Komplexe boolesche Ausdrücke entstehen durch beliebige Verknüpfungen von booleschen Ausdrücken durch **boolesche Operatoren**.
- **Anweisungen** sind in Ruby **Ausdrücke**, da jede Anweisung einen Wert liefert.
- Auch **Kontrollstrukten** sind Ausdrücke, die einen Wert liefern.
- In Ruby hat ***nil*** den Wahrheitswert ***false*** und alle Werte ungleich ***nil*** und ***false*** haben den Wahrheitswert ***true.***
- **Statement Modifier** sind Kurzformen für bedingte Anweisungen und Schleifen.