Weitere Konzepte und Techniken

Klassenvariablen und –Methoden, Singletons, Dateien, Kopieren von Objekten

Konzepte und Techniken

- Singleton Pattern
- Klassenvariablen
- Klassenmethoden
- Utility Klassen
- Klassenobjekt ***self***
- Ein/Ausgabe auf Konsole und Datei
- Objekte auf Dateien schreiben und von Dateien lesen
- Objekterzeugung durch Kopieren
- Symbols

Aufgabenstellung

- Wir wollen für unsere Ruby Projekte einen Klasse schreiben, die Fehler auf der Konsole oder in einer Datei ausgibt.
- Zu der Klasse ***ProjectLogger*** soll es zur Laufzeit **nur genau eine Instanz geben**, so dass alle Fehlerausgaben über dieses eine Objekt synchronisiert werden können.
- Das ***ProjectLogger*** Objekt wird entweder nur für die Ausgabe auf der Konsole, oder die Ausgabe auf Konsole und in eine Datei erzeugt.
- ***ProjectLogger*** soll folgende Methoden haben:
  - ***log(typ,klasse,methode,text)*** gibt zu einem Fehlertyp den Namen der Klasse, der Methode und den Fehlertext aus.
  - Meldungen sollen nur ausgegeben werden, wenn der Logger eingeschaltet ist (***logging?*** liefert ***true***).
  - Zu Beginn ist der Logger ausgeschaltet.
  - Mit ***logging = true*** kann er eingeschaltet werden.

Eine Klasse mit nur einer Instanz: Singletonklasse

- Die erste Aufgabe ist es, eine Klasse zu schreiben, von der wir nur eine einzige Instanz erzeugen können.

- Klassen mit nur einer Instanz heißen auch **Singleton**-Klassen.

- Dazu müssen wir:
  - das normale Erzeugen von Objekten blockieren, indem wir ***new*** zu einer privaten Klassenmethode machen.
  - eine eigene Methoden für das Erzeugen von Objekten einführen, die die einzige Instanz der Klasse ***ProjectLogger*** erzeugt und zurückgibt.
  - eine Variable einführen, in der wir uns für die Klasse ***ProjectLogger*** diese einzige Instanz merken. Diese Variable kann keine Instanzvariable sein, da wir ihren Inhalt unabhängig von der Erzeugung eines Objektes kennen müssen. Diese Variablen heißen **Klassenvariablen**.
  - Wir erzeugen genau eine einzige Instanz, nur dann wenn die Klassenvariable ***nil*** ist. In allen anderen Fällen geben wir den Inhalt von ***@@logger*** zurück.

Klasse *ProjectLogger*

```
class ProjectLogger
  private_class_method :new
  @@logger = nil
  def ProjectLogger.erzeuge_instanz(
      datei=nil)
    if (@@logger.nil?())
      @@logger = new(datei)
    end
    @@logger
  end

  def initialize(datei)
    @logging = false
    @datei = datei
  end
end
```

- Um das normale Erzeugen von Objekten zu unterbinden blockieren wir ***new***, und machen diese Methode zu einer privaten Klassenmethode (***private_class_method***).
- Wir merken uns die einzige Instanz in der Klassenvariable ***@@logger***. Diese ist zu Anfang ***nil***.
- Wir definieren die Klassenmethode ***erzeuge_instanz***, der wir optional einen Dateinnamen übergeben können. (Ist der Dateiname ***nil***, dann soll die Ausgabe auf der Konsole erfolgen.) Die Methode erzeugt nur genau einmal eine Instanz von der Klasse (***new(datei)***), nämlich dann wenn ***@@logger.nil?()*** gilt. Sonst gibt sie die einzige Instanz zurück.
- Im ***initialize*** merkt sich die ***ProjectLogger*** Instanz die Log-Datei ***(@datei***) und setzt den Zustand für das Loggen auf ***false***.

Nachweis des Singleton-Verhaltens

- Mehrfacher Aufruf von ***erzeuge_instanz()*** liefert immer dasselbe Objekt.
- Versuch mit ***new*** unkontrolliert mehrere Objekte zu erzeugen, gibt einen Fehler: ***new*** ist ***private***.
- **Anmerkung**: Wir werden gleich lernen, wie wir dennoch unbemerkt von der Klasse ***ProjectLogger*** mehr als eine Instanz erzeugen können. Jetzt aber zurück zu den Klassenvariablen – methoden.

```
require "ProjectLogger"

puts ProjectLogger.erzeuge_instanz()
puts ProjectLogger.erzeuge_instanz()
ProjectLogger.new("never_created",nil)
```

```
#<ProjectLogger:0x2bd1440>
#<ProjectLogger:0x2bd1440>

D:/haw/vorlesungen/rubypr1/sose10/workspace/v10-Singleton/project_logger_singleton_test.rb:5: private method `new' called for ProjectLogger:Class (NoMethodError)
```

Klassenvariablen und Klassenmethoden

```
class ProjectLogger
  private_class_method :new
  @@logger = nil
  def ProjectLogger.erzeuge_instanz(
      datei=nil)
    if (@@logger.nil?())
      @@logger = new(datei)
    end
    @@logger
  end
end
```

- Klassenvariablen wie ***@@logger*** beginnen immer mit ***@@.***
- Klassenvariablen sind wie Instanzvariablen ***private***.
- Klassenmethoden wie ***erzeuge_instanz***, sind Methoden, denen der Name der Klasse vorangestellt ist.

Klassenvariablen

```ruby
class Student
  # Pruefungen aller Studenten
  @@pruefungen=0
  def initialize(matnr)
    @matnr = matnr
    # Pruefungen dieses Studenten
    @pruefungen=0
  end
  def pruefung_ablegen()
    @pruefungen +=1
    @@pruefungen +=1
  end
  def to_s()
    "pruefungen:#{@matnr}=#{@pruefungen} gesamt=#{@@pruefungen}"
  end
end
```

- Klassenvariable ***@@pruefungen*** ist für alle Objekte der Klasse ***Student*** sichtbar und änderbar. Die Änderungen sind immer für alle Instanzen sofort sichtbar.
- ***@@pruefungen*** wird immer erhöht, wenn ein beliebiger Student eine Prüfung ablegt. ➔ In ***@@pruefungen*** steht immer die Anzahl aller insgesamt von allen Studenten abgelegten Prüfungen.
- In ***@pruefungen*** hingegen nur die von einem Studenten abgelegten Prüfungen.

Klassenvariablen

```
s1 = Student.new(121344)
s2 = Student.new(444444)
s3 = Student.new(999999)

s1.pruefung_ablegen()
s1.pruefung_ablegen()
s1.pruefung_ablegen()
s2.pruefung_ablegen()
s2.pruefung_ablegen()
s3.pruefung_ablegen()
puts s1, s2 ,s3
```

```
Pruefungen:121344=3 gesamt=6
Pruefungen:444444=2 gesamt=6
Pruefungen:999999=1 gesamt=6
```

- Das Ergebnis nach der ersten Prüfungswoche.

Klassenmethoden

```
puts ProjectLogger.erzeuge_instanz()
puts ProjectLogger.erzeuge_instanz()
```

- Klassenmethoden wie ***erzeuge_instanz***, sollten immer auf dem Klassenobjekt (hier ***ProjectLogger***) aufgerufen werden.
- Klassenmethoden können auch dann aufgerufen werden, wenn noch keine Objekt der Klasse existiert.

Klassenmethoden

```
class ProjectLogger
  private_class_method :new
  @@logger = nil
  def
   self.erzeuge_instanz(datei=nil)

    if (@@logger.nil?())
      @@logger = new(datei)
    end
    @@logger
  end
end
```

- Klassenmethoden beziehen sich auf das Klassenobjekt (hier ***ProjectLogger***). In der Definition von Klassenmethoden können wir anstelle des Klassennamens auch die Referenz auf das Klassenobjekt verwenden (***self***).

- Im direkten Klassenscope und in Klassenmethoden ist ***self*** die Referenz auf ein Klassenobjekt.
- Klassenname: Stellt den Gesichtspunkt „Klassenmethode“ in den Vordergrund.
- ***self*** stellt den Gesichtspunkt „Methode des Klassenobjekts“ in den Fordergrund.

Klassenmethoden

```
class ProjectLogger
  private_class_method :new
  @@logger = nil
  def
   self.erzeuge_instanz(datei=nil)

    if (@@logger.nil?())
      @@logger = new(datei)
    end
    return @@logger

  end
end
```

- In Klassenmethoden können nur Klassenmethoden aufgerufen werden. Klassenmethoden können **weder** Instanzvariablen **noch** –methoden verwenden.
- ***new*** ist eine Klassenmethode.
- Das Klassenobjekt ***self*** hat Zugriff auf ***private*** Klassenmethoden. Daher kann ***new*** in ***erzeuge_instanz*** problemlos aufgerufen werden.

self in Klassenmethoden und *self* in Objektmethoden

```
class ProjectLogger
...
  def self.erzeuge_instanz(datei=nil)
    if (@@logger.nil?())
      @@logger = new(datei)
    end
   @@logger
  end
 def log(typ,klasse,methode,text)
    log_info = log_info(typ,klasse,methode,text)
    if self.logging?()
      $stdout << log_info
      if @datei
        File.open(@datei,"a") do |datei_var|
          datei_var <<  log_info
        end
      end
    end
  end
end
```

self in Klassenmethoden und *self* in Instanzmethoden

```
class ProjectLogger
...
  def
   self.erzeuge_instanz(datei=nil)
    ...
  end
  def log(typ,klasse,methode,text)
    ...
    if self.logging?()
      ...
    end
  end
end
```

- Im direkten Klassenscope und in Klassenmethoden ist ***self*** die Referenz auf ein Klassenobjekt.
- In Instanzmethoden ist ***self*** die Referenz auf eine Instanz der Klasse.
- In ***self.erzeuge_instanz*** bezieht sich ***self*** auf das **Klassenobjekt** ***ProjectLogger***.
- In der Instanzmethode ***log*** bezieht sich im Ausdruck ***self.logging*** self auf eine **Instanz** der Klasse ***ProjectLogger***

Utilities

```
Math::PI
Math::E
Math.cos(Math::PI/3)
Math.tan(Math::PI/4)
Math.log(Math::E**2)
(1 + Math.sqrt(5))/2
```

- Klassen, die nur Klassenmethoden und Konstanten enthalten und von denen keine Instanzen erzeugt werden können (wie z.B. ***Modul Math***), heißen Utilities.
- Sie stellen Funktionen, aber keine Objekte bereits.

Loggen von arithmetischen Summenfunktionen

- In Aufgabe A2 haben Sie die Methode ***naehere_einsx_invers(x,n)*** geschrieben und die dafür vorgegeben Tests ausgewertet.
- Das Ein-/Ausgabeverhalten dieser Methode soll mit dem ***ProjectLogger*** auf der **Konsole** und in ***formeln_log.txt*** gelogged werden.
- Wenn die Eingabewerte ***x,n*** die vorgeschriebenen Wertebereiche nicht erfüllen, soll ein Fehler gelogged werden.
- Wenn n <= 2, soll eine Warnung anzeigen, dass der Ergebniswert eine schlechte Näherung darstellt. Der Ergebniswert wird mit ausgegeben.
- Wenn n > 2, soll eine Information mit Ergebniswert gelogged werden.
- Zu Beginn soll eine Information anzeigen, mit welchen Eingabewerten die Methode aufgerufen wird.
- Wir haben **drei Ereignistypen** für das Loggen: Informationen, Warnungen und Fehler, die alle unmittelbar mit der Klasse ***ProjectLogger*** zusammenhängen.

➔ Kandidaten für Klassenkonstanten

Loggen von arithmetischen Summenfunktionen

```
class ProjectLogger
  ...
  @@logger = nil

  INFO = :info
  WARNING = :warnung
  ERROR = :fehler

  def log(typ,klasse,methode,text)
    ...
     if self.logging?()
       $stdout << log_info
       if @datei
         File.open(@datei,"a") ...
    end
     end
  end
end
```

- Wir erweitern ***ProjectLogger*** zunächst um die entsprechenden Klassenkonstanten.
- Konstanten sind im Gegensatz zu Klassenvariablen für die Außenwelt sichtbar, so dass wir diese Konstanten für das Loggen von Programmen nutzen können. Sollten nur einmal zugewiesen werden.
- Dazu übergeben wir den Ereignistyp als ersten Parameter beim Aufruf der Methode ***log(typ,klasse,methode,text)***
- Wir erzeugen den ***ProjectLogger*** mit dem Dateinamen ***formeln_log.txt***, dann erfolgt die Ausgabe auf der Konsole ***($stdout << loginfo)*** und in eine Datei, die im Wurzelverzeichnis des Projektes liegt.

Loggen von arithmetischen Summenfunktionen

```
def naehere_einsx_invers_v1(x,n)
   logger = ProjectLogger.erzeuge_instanz("formeln_log.txt")
   logger.logging=true
   logger.log(ProjectLogger::INFO,self.class,'naehere_einsx_invers_v1',"(x=#{x} n=#{n})")
   if (!(x.abs < 1))
     logger.log(ProjectLogger::ERROR,self.class,'naehere_einsx_invers_v1',
                                                    "(|x| > 1) fuer x =#{x}")
     return
   end
   if ((n <0) || !n.is_a?(Integer))
     logger.log(ProjectLogger::ERROR,self.class,'naehere_einsx_invers_v1',
                                                    "#{n} < 0 oder kein Integer")
     return
   end
   sum = 0
   0.upto(n) {|i| sum = sum + Float((-1)**i)*(x**i)}
   if (n <= 2)
     logger.log(ProjectLogger::WARNING,self.class,'naehere_einsx_invers_v1',
                                        "Ergebnis #{sum} keine gute Naeherung fuer n<=2")
   else
     logger.log(ProjectLogger::INFO,self.class,'naehere_einsx_invers_v1',"Ergebnis #{sum}")
   end
  sum
end
```

Loggen von arithmetischen Summenfunktionen

```
naehere_einsx_invers_v1(0.5, 10)
puts("--------------------")
naehere_einsx_invers_v1(2, 10)
$stdout << "--------------------\n"
naehere_einsx_invers_v1(0.5, 2)
```

```
info :: Object.naehere_einsx_invers_v1, (x=0.5 n=10)
info :: Object.naehere_einsx_invers_v1, Ergebnis 0.6669921875
--------------------
info :: Object.naehere_einsx_invers_v1, (x=2 n=10)
fehler :: Object.naehere_einsx_invers_v1, (|x| > 1) fuer x =2
--------------------
info :: Object.naehere_einsx_invers_v1, (x=0.5 n=2)
warnung :: Object.naehere_einsx_invers_v1, Ergebnis 0.75 keine gute
   Naeherung fuer n<=2
```

Zusammenfassung Klassenvariablen und Methoden

Klassenvariablen

- beginnen immer mit ***@@.***
- werden im direkten Scope der Klasse definiert
- sind in Klassenmethoden und in Instanzmethoden der Klasse sichtbar und modifizierbar
- sind private Variablen der Klasse und für die Außenwelt im Gegensatz zu **Klassenkonstanten** nicht sichtbar
- existieren unabhängig von den Instanzen einer Klasse

Klassenmethoden

- werden durch Voranstellen des Klassennamens oder ***self*** definiert
- sollten immer durch Voranstellen des Klassennamens aufgerufen werden
- haben Zugriff auf Klassenvariablen und Klassenmethoden (auch private Klassenmethoden)
- haben **keinen Zugriff** auf Instanzvariablen und –methoden
- ***self*** in Klassenmethoden bezieht sich auf das Klassenobjekt

Dateien

```
def log(typ,klasse,methode,text)
    log_info =
   log_info(typ,klasse,methode,text)
    if self.logging?()
      $stdout << log_info
      unless @datei.nil?
        File.open(@datei,"a") do
         |datei_obj|
            datei_obj <<  log_info
         end
      end
    end
  end
```

- Die Methode ***log*** von ***ProjectLogger*** gibt Informationen auf der Konsole und ggf. in eine Datei aus.
- In beiden Fällen wird ***log_info*** auf ein Objekt geschoben (***<<***).
- Für die Konsole auf das Objekt ***$stdout***, für die Dateiausgabe auf ein Dateiobjekt.
- Offenbar ist ***$stdout*** das Objekt, das die Ausgabe auf der **Konsole** übernimmt.
- Offenbar haben **Dateien** und ***$stdout*** gemeinsame Eigenschaften für die Ausgabe von Zeichenketten.

Dateien und Ein/Ausgabe

- ***$stdout*** und eine Datei, die mit ***File.open(datei_name)*** erzeugt wird, sind beide ***IO*** (Stream) Objekte. ***is_a?(IO)*** liefert ***true.***
- Wir können uns IO-Objekte wie zwei Flüsse vorstellen, durch die Zeichen in zwei Richtungen strömen können.
- An einem Ende des Flusses befindet sich das Objekt, das die Zeichen enthält, am anderen Ende ein Programm, das Zeichen liest (Mündung) oder Zeichen schreibt (Quelle).
- Diese Flüsse heißen professioneller auch Ströme (engl. stream).

- ***IO*** – Objekte können mit ***open(name, richtung)*** zum Lesen und/oder Schreiben geöffnet werden.
- ***IO*** ist die Klasse, die die Basisfunktionalität für Ein-Ausgabe zur Verfügung stellt:
  - ***readline, readlines, read_char, gets, getc***
  - ***print, printf, puts, putc, write, <<***
- ***File*** ist die einzige Standard Subklasse von ***IO*** und enthält weitere nützliche Methoden für den Umgang mit Dateien und Verzeichnissen. ***File*** hat alle Methoden von ***IO***, auch die Klassen-methoden. Andere Bibliotheken bieten Unterklassen wie TCPSocket uvam.

Dateien und Ein/Ausgabe

- ***IO*** – Objekte werden mit den Klassenmethoden ***open(name, richtung)*** oder ***new(name, richtung)*** zum Lesen und/oder Schreiben geöffnet.
- Das zweite Argument ***richtung*** gibt die Art des Öffnens eines IO Objektes an.
- Wenn ***open*** ein Block übergeben wird, dann wird das ***IO***-Objekt an die Block-variable gebunden und das ***IO***-Objekt geschlossen, wenn der Block durchlaufen wurde.
- ***new*** ignoriert Blöcke. Ein mit ***new*** oder ***open*** ohne Block erzeugtes ***IO*** Objekt, muss im Programm mit ***close*** geschlossen werden.
- Da ***File*** alle Methoden von IO erbt, und wir uns im 1'ten Semester nur mit Datei Ein/ Ausgabe beschäftigen, beziehen sich alle nachfolgenden Beispiele auf Dateien.

richtung	**Erläuterung**
"r"	nur Lesen, wenn ***IO*** Objekt ***name*** nicht existiert, dann Fehler. Der Default, wenn für den optionalen Parameter ***richtung*** kein Wert übergeben wird.
"r+"	Lesen und Schreiben, wenn ***IO*** Objekt ***name*** nicht existiert, dann Fehler.
"w"	Schreiben, wenn ***IO*** Objekt ***name*** nicht existiert, dann wird ein Objekt erzeugt, wenn es existiert, dann wird dessen Inhalt gelöscht.
"w+"	Schreiben und Lesen: vgl. ***"w"***
"a"	Schreiben, wenn ***IO*** Objekt ***name*** nicht existiert, dann wird ein Objekt erzeugt, wenn es existiert, dann wird am Ende weitergeschrieben (a steht für append = anhängen).
"a+"	Schreiben und Lesen: vgl. ***"a".***

Mit Iteratoren Dateiinhalte auslesen

- ***IO*** (und damit auch ***File***) inkludiert das Modul ***Enumerable*** und implementiert ***each***. ***Each*** liefert nacheinander die Zeilen in einer Datei.

- Wir öffnen die in der Methode ***naehere_einsx_invers*** erzeugte Logdatei und geben den Inhalt zeilenweise aus.

- In dem Block der Methode ***open,*** wird das Dateiobjekt zum Dateinamen ***"formeln _log.txt"*** an die Blockvariable ***datei*** gebunden.

- Über dieses Dateiobjekt iterieren wir ***each*** und geben den Inhalt zeilenweise aus.

```
File.open("formeln_log.txt") do |datei|
    datei.each { |line| puts line }
end
```

```
info :: Object.naehere_einsx_invers_v1,
    (x=0.5 n=10)
info :: Object.naehere_einsx_invers_v1,
    Ergebnis 0.6669921875
info :: Object.naehere_einsx_invers_v1,
    (x=2 n=10)
fehler :: Object.naehere_einsx_invers_v1,
    (|x| > 1) fuer x =2
info :: Object.naehere_einsx_invers_v1,
    (x=0.5 n=2)
warnung ::
    Object.naehere_einsx_invers_v1,
    Ergebnis 0.75 keine gute Naeherung
    fuer n<=2
```

Mit readline / readlines eine Datei auslesen

- Alternativ könnten wir die Datei auch mit ***readline*** auslesen. Dabei müssen wir aber selber aufpassen, dass wir nicht über das Dateiende (***!datei.eof?()***) hinauslesen.

- ***readlines*** liest alle Zeilen einer Datei in ein Array.

```
File.open("formeln_log.txt") do |datei|
  while  !datei.eof?()
    puts datei.readline()
  end
end
```

```
File.open("formeln_log.txt") do |datei|
    puts datei.readlines()
end
```

Zeilenweises Schreiben auf eine Datei

```
def log(typ,klasse,methode,text)
    log_info =
    log_info(typ,klasse,methode,text)
    if self.logging?()
      $stdout << log_info
      if @datei.nil
        File.open(@datei,"a") do |datei|
          datei <<  log_info
        end
      end
    end
  end

 def log_info(typ,klasse,methode,text)
   "#{typ} :: #{klasse}.#{methode},
    #{text}\n"
 end
```

- ***IO*** Objekten zum Schreiben öffnen mit eine der Option ***"a" / "a+"*** oder ***"w" / "w+"***.
- Methoden ***<<*** und ***write*** schreiben Strings auf ***IO*** Objekte ohne Zeilenumbruch.
- ***log_info*** bereitet daher die Strings mit Zeilenumbruch vor (***"\n"*** am Ende des Strings).
- ***$stdout*** ist das ***IO*** Objekt für die Ausgabe auf der Konsole. Es ist exklusiv zum Schreiben geöffnet. ***Kernel.puts*** ruft ***puts*** auf ***$stdout*** auf.
- ***$stdin*** ist das Objekt für das Einlesen von der Konsole. ***Kernel.gets*** ruft ***gets*** auf ***$stdin*** auf.

- **Ü10-1**: Wir wollen Objekte der Klasse Kreis in eine Datei schreiben und Objekte aus dieser Datei wieder auslesen.
- Der Inhalt der Datei sieht wie folgt aus:

  1::1::4

  3::2::1

- Das erste Element ist die x, das zweite die y Koordinate eines Kreises, das dritte der Radius. Alle drei Werte sollen Floats sein.
- Schreiben Sie bitte eine Objektmethode ***fuer_datei_aufbereiten,*** die für einen Kreis eine Zeichenkette wie oben gezeigt erzeugt sowie eine Klassenmethode ***von_string_konstruieren*** die eine Zeichenkette in obigem Format als Parameter hat und daraus ein Objekt konstruiert. Verwenden Sie diese Methoden in einem Script und schreiben und lesen Sie 2 Kreise in /von eine/einer Datei.

```ruby
class Kreis
  attr_reader :mittelpunkt, :radius
  def initialize(mittelpunkt,radius)
    @mittelpunkt = mittelpunkt
    @radius = radius
  end
  def to_s
    "K(#@mittelpunkt,#@radius)"
  end
end

class Point
  attr_reader :x,:y
  def initialize(x,y)
    @x = x
    @y = y
  end
  def to_s
    "P(#@x,#@y)"
  end
end
```

Lösung zu Ü1-10

```
class Kreis
  attr_reader :mittelpunkt, :radius
  def initialize(mittelpunkt,radius)
    @mittelpunkt = mittelpunkt
    @radius = radius
  end
  def to_s
    "K(#@mittelpunkt,#@radius)"
  end
  def fuer_datei_aufbereiten()
    "#{@mittelpunkt.fuer_datei_aufbereiten}::#@radius"
  end
  def Kreis.von_string_konstruieren(string)
    radius = Float(string.split("::")[2])
    new(Point.von_string_konstruieren(string),radius)
  end
end
```

Lösung zu Ü1-10

```
class Point
  attr_reader :x,:y
  def initialize(x,y)
    @x = x
    @y = y
  end
  def to_s
    "P(#@x,#@y)"
  end
  def fuer_datei_aufbereiten()
    "#{@x}::#@y"
  end
  def self.von_string_konstruieren(string)
    x = Float(string.split("::")[0])
    y = Float(string.split("::")[1])
    return new(x,y)
  end
end
```

Lösung zu Ü1-10

```
k1 = Kreis.new(Point.new(1,1),4)
k2 = Kreis.new(Point.new(3,2),1)

p k1
p k2

File.open("kreise","w") {|datei|
  datei.puts(k1.fuer_datei_aufbereiten())
  datei << k2.fuer_datei_aufbereiten()
}

kreise = []
File.open("kreise") {|datei|
  datei.each {|line| kreise << Kreis.von_string_konstruieren(line)}
}

p kreise
```

Serialisieren und Deserialisieren von Objekten

```
require "Kreis"

k0 = Kreis.new(Point.new(2,4),5)
k1 = Kreis.new(Point.new(7,2),1)
puts "k0 = #{k0.inspect()}"
puts "k1 = #{k1.inspect()}"

File.open("kreis_objekte","w") {|file|
  Marshal.dump(k0,file)
  Marshal.dump(k1,file)
}

kreise = []
File.open("kreis_objekte","r") {|file|
  while !file.eof?()
    kreise << Marshal.load(file)
  end
}

kreise.each_index {|i| puts "k#{i} =
    #{kreise[i].inspect()}"}
```

- ***Marshal.dump(obj, datei)*** schreibt den Inhalt eines Objektes in eine Datei. Jedes Objekt wird in eine Zeile geschrieben. Man nennt dies auch **Serialisieren** eines Objektes.
- ***Marshal.load(datei)*** liest Objekte aus einer Datei. Dieser Vorgang heißt auch **Deserialisieren** von Objekten.
- Im nebenstehenden Quelltext werden zwei Kreisobjekte in eine Datei geschrieben (**serialisiert**) und anschließend wieder eingelesen (**deserialisiert**) und ausgegeben.

Objekte in Datei schreiben und von Datei lesen

```
k0 = #<Kreis:0x2bd0a2c @radius=5,
    @mittelpunkt=#<Point:0x2bd0aa4
    @y=4, @x=2>>
k1 = #<Kreis:0x2bd0040 @radius=1,
    @mittelpunkt=#<Point:0x2bd00b8
    @y=2, @x=7>>

k0 = #<Kreis:0x2bcfdc0 @radius=5,
    @mittelpunkt=#<Point:0x2bcfd5c
    @y=4, @x=2>>
k1 = #<Kreis:0x2bcfce4 @radius=1,
    @mittelpunkt=#<Point:0x2bcfc80
    @y=2, @x=7>>
```

- Das Ergebnis ist eine Liste von inhalts gleichen aber nicht identischen Kreisen.

Objekte in Datei schreiben und von Datei lesen

```
pl1 = ProjectLogger.erzeuge_instanz()
puts "pl1 = #{pl1.inspect}"
pl2 = nil
File.open("pl_objekt","w")do|file|
  Marshal.dump(pl1,file)
end
File.open("pl_objekt")do|file|
  pl2= Marshal.load(file)
end
puts "pl2 = #{pl2.inspect}"
```

```
pl1 = #<ProjectLogger:0x2bce0ec
    @logging=false, @datei=nil>
pl2 = #<ProjectLogger:0x2bcdcc8
    @logging=false, @datei=nil>
```

- Jetzt schreiben wir ein Objekt der Klasse ***ProjectLogger*** in eine Datei und lesen es anschließend wieder ein.
- Offenbar haben wir über den Umweg der Dateiausgabe jetzt doch ein zweites ***ProjectLogger*** Objekt erzeugt.

Duplizieren über (De)Serialisieren unterbinden

```
class ProjectLogger

  def self._load(string)
    erzeuge_instanz()
  end
  def _dump(depth)
     ""
  end
end

pl1 = ProjectLogger.erzeuge_instanz()
puts "pl1 = #{pl1.inspect}"
...
puts "pl2 = #{pl2.inspect}"
```

```
pl1 = #<ProjectLogger:0x2bcddf4
    @logging=false, @datei=nil>
pl2 = #<ProjectLogger:0x2bcddf4
    @logging=false, @datei=nil>
```

- Wenn eigene Klassen die Instanzmethoden ***_dump*** und die Klassenmethode ***_load*** implementieren, dann verwendet ***Marshal*** diese Methoden für das Schreiben und Lesen von Objekten.
- ***_dump(depth)*** liefert eine String-Repräsentation des Objektes. Da wir nichts schreiben wollen, liefert ***_dump*** den leeren String.
- ***self._load(string)*** erzeugt aus ***string*** ein Objekt der Klasse, da wir kein neues Objekt erzeugen wollen, liefern wir die einzige Instanz zurück.
- Dann erzeugt dasselbe Programm von der Folie zuvor keine neue Instanz.

Duplizieren durch Kopieren von Objekten

```
pl1 = ProjectLogger.erzeuge_instanz()
puts "pl1 = #{pl1.inspect}"
pl2 = pl1.clone()
puts "pl2 = #{pl2.inspect}"
pl3 = pl1.dup()
puts "pl3 = #{pl3.inspect}"
```

```
pl1 = #<ProjectLogger:0x2bceb00
    @datei=nil, @logging=false>
pl2 = #<ProjectLogger:0x2bceab0
    @datei=nil, @logging=false>
pl3 = #<ProjectLogger:0x2bcea4c
    @datei=nil, @logging=false>
```

- Wir sind noch nicht am Ende:
- Die Methoden ***dup*** und ***clone*** erzeugen Kopien von Objekten.
- Wenn wir auf die einzige Instanz von ***ProjectLogger*** ***dup*** oder ***clone*** anwenden, erhalten wir wieder zwei neue nicht zu der einzigen Instanz identische Objekte.
- Um das Duplizieren über den Umweg des Kopierens zu verhindern, schreiben wir die Methoden ***dup*** und ***clone*** in ***ProjectLogger*** so um, dass sie einen ***TypeError*** erzeugen.

Duplizieren durch Kopieren von Objekten

```
class ProjectLogger
  def clone()
    raise TypeError, "clone nicht
    unterstützt"
  end

  def dup()
    raise TypeError, "dup nicht
    unterstützt"
  end
end
```

- Wir sind noch nicht am Ende:
- Um das Duplizieren über den Umweg des Kopierens zu verhindern, schreiben wir die Methoden ***dup*** und ***clone*** in ***ProjectLogger*** so um, dass sie einen ***TypeError*** erzeugen.
- So unterbinden wir jeden Versuch Objekte der Singleton Klasse zu kopieren.

Eine letzte Frage: Wie tief kopieren *clone* und *dup*?

```
k1 = Kreis.new(Point.new(1,1),1)
puts "k1 = #{k1.inspect}"
k2 = k1.clone()
puts "k2 = #{k2.inspect}"
```

```
k1 = #<Kreis:0x2bcf3e8 @radius=1,
    @mittelpunkt=#<Point:0x2bd10a8 @x=1,
    @y=1>>
k2 = #<Kreis:0x2bce984 @radius=1,
    @mittelpunkt=#<Point:0x2bd10a8 @x=1,
    @y=1>>
```

- Betrachten wir Kreise. Kreise enthalten Referenzen auf ein ***Point*** Objekt.
- Frage: Werden bei ***clone*** und ***dup*** nur Kreis oder auch alle referenzierten Objekte kopiert.
- Das Ergebnis des Programms rechts zeigt, dass nur die höchste Ebene, nämlich Kreis kopiert wird, alle referenzierten Objekte bleiben unverändert.
- Man sagt auch ***clone*** und ***dup*** erzeugen **flache Kopien**: Sie kopieren das Objekt und alle Instanzvariablen, aber **nicht den Inhalt der Instanzvariablen.**

Wir hätten es uns auch leichter machen können

```ruby
require "Singleton"

class ProjectLogger
  include Singleton
  # Klassenmethode _load muss implementiert
  # werden

  def ProjectLogger._load(_string)
    self.instance()
  end

  INFO = :info
  WARNING = :warnung
  ERROR = :fehler
  def log_to(datei) @datei = datei end

  def log(typ,klasse,methode,text) ... end

  def log_info(typ,klasse,methode,text) ...
    end

  def logging?() ... end
  def logging=(logging) ...  end
end
```

- Modul ***Singleton*** implementiert das Singleton Verhalten für alle Klassen.
- Mit der Methode ***instance()*** wird eine Instanz erzeugt. Sie entspricht der Methode ***erzeuge_instanz*** in unserer Lösung.
- Da wir ***instance()*** keinen Parameter übergeben können, müssen wir die Datei mit der Methode ***log_to()*** bekanntgeben.
- Die Instanzvariable ***@@logger*** entfällt, die Methoden, ***_dump, clone*** und ***dup*** sind bereits in ***Singleton*** implementiert.
- Alle anderen Methoden bleiben unverändert.

Wir hätten es uns auch leichter machen können

- Die Testscripts und die 2‘te Version der Klasse ***ProjectLogger*** finden Sie im Verzeichnis ***modul_singleton*** im mit dieser Vorlesung ausgelieferten Ruby-Projekt.

Aufgaben

- **Ü10-2**: Schreiben Sie bitte eine Klasse ***Season***, die nur vier Instanzen für Frühjahr, Sommer, Herbst und Winter hat.

- **Ü10-3**: Ihre Klasse ***Season*** soll mehr-sprachig werden, d.h., für den jeweiligen Sprach- und Kulturraum sollen die Ausgaben, in der jeweiligen Sprache erfolgen!

:symbols

- Alles, was mit einem Doppelpunkt beginnt (:) ist ein symbol (***:show, :red***).
- Symbols sind eindeutige Objekte in einer Ruby runtime, die Namen representieren.
- Sie sind ein effizientes Mittel um Namen darzustellen.
- Wählen Sie Symbols, wenn Sie einen String wieder und wieder in einem Programm verwenden.
- Verwenden Sie stattdessen Strings, so haben Sie jedes Mal ein neues String Objekt.
- Da symbol nur einmal existieren, können Sie Identität test: (***:rot.equal?(:rot)***). Wenn Sie Strings verwenden, mpssen Sie auf Gleichheit prüfen (***"rot" == "rot"***), also die Inhalte vergleichen (teuer!)
- Ruby verwendet symbols intern um Methodennamen darzustellen. Alle Ruby symbols erhalten Sie mittels ***Symbol.all_symbols***
- Brauchen Sie den zugehörigen String, verwenden Sie: ***:show.to_s***
- Andersherum können Sie auch leicht einen String in ein symbol verwandeln:: ***"show".to_sym***

Wann sollten Sie symbols verwenden?

- Let‘s have a look at Dr. Jones, a psychologist performing association tests with his clients.
- A typical test szenario:

Dr Jones:	red
Client:	Ruby
Dr Jones:	Transport
Client:	Rails
Dr Jones:	clumsy
Client:	cat

- test results are recorded in hash tables

```
{
 :client1 => {
    "red" => "Ruby",
    "transport" => "Rails",
    "clumsy" => "cat" },
 :client2 => {
    "red" => "color",
    "transport" => "car",
    "clumsy" => "little boy" }
}
```

When to use symbols?

- Dr. Jones is an extremely busy psychologist, who performs 10-thousand tests.
- All of a sudden his programs runs slower and slower.
- **The reason**: Dr. Jones's program has produced 60 thousand string objects.
- A good friend of him gives him a hint: Take symbols for your keys and you can cut the required space in half.
- Dr Jones did and could go ahead for another 10-thousand clients.

```
{
  :client1 => {
    :red => "Ruby",
    :transport => "Rails",
    :clumsy => "cat" },
  :client2 => {
    :red => "color",
    :transport => "car",
    :clumsy => "little boy" }
}
```

Zusammenfassung

- Eine Singletonklasse ist eine Klasse, von der es nur eine Instanz gibt.
- Serialisieren und anschließendes Deserialisieren von Singletons zerstört die Eigenschaften eines Singletons.
- Kopieren von Objekten zerstört die Eigenschaften eines Singletons.
- In Ruby werden Klassen zu Singletons durch Inkludieren des Moduls ***Singleton.*** Das Modul ***Singleton*** stellt sicher, dass die uns bekannten Tricks für das Zerstören von Singletons nicht mehr möglich sind.
- Kopieren von Objekten mit ***clone*** und ***dup*** erzeugt nur flache Kopien von Objekten.
- **Klassenvariablen** sind private Variablen einer Klasse, die für die Klasse und alle Objekte der Klasse sichtbar sind. Jeweils eine Referenz pro Klasse.
- **Klassenmethoden** sind Methoden einer Klasse (Methoden des Klassenobjektes ***self***).
- Im direkten Scope der Klasse referenziert ***self*** das Klassenobjekt.
- Klassenmethoden haben **keinen Zugriff** auf Instanzmethoden und Instanz-variablen.
- **Dateien** sind Objekte der Klasse ***File***, die Methoden von ***IO*** zur Verfügung hat, um Dateien zu öffnen, zu schreiben und zu lesen.