03-05-H
-709.53

Informatik für Nichtinformatiker (3)

Prof. Dr. Udo Frese
Tobias Hammer

Klassen, Objekte und Variablen

Beispiel: Die Klasse Bruch

Was bisher geschah

- **Objektorientierte Sichtweise ist, die Realität als Ansammlung von Objekten und deren Interaktion zu sehen.**
- **Objektorientierte Softwareentwicklung bedeutet, von der Realität objektweise zu abstrahieren, d.h. jedes relevante realen Objekt hat eine abstrahierte Entsprechung als symbolische Information.**
- **Eine Klasse ist ein Typ von Objekten, die im Programm gleich behandelt werden**
  - Zustand: Welche Daten beschreiben ein Objekt?
  - Funktionalität: Was kann das Objekt tun?
- **Vererbung: Objekte einer Unterklasse sind auch Objekte der Oberklasse, Zustand und Funktionalität wird übernommen**
- **Polymorphie: Es wird immer die Funktionalität genommen, die in der wirklichen Klasse eines Objektes definiert ist, auch wenn es die Funktionalität in einer Oberklasse schon gibt.**
- **Datenverkapselung: Das Objekt ist für das Wie verantwortlich.**

Klassen, Objekte und Variablen

- **(D. Thomas et al., Programming Ruby, Kapitel 3)**

Klassen, Objekte und Variablen

- **Einordnung**
- **In Ruby ist alles ein Objekt.**
- **Letzte Woche: CRC-Karten - Architektur der Klassen im Großen**
  - Welche Klassen gibt es?
  - Welche Verantwortlichkeiten haben sie?
  - Wie hängen sie zusammen?
- **Heute: Einzelne Klasse im Kleinen**
  - Wie zerlegt sich die Verantwortlichkeit in einzelne Teile (Methoden)?
  - Wie schreibt man sie in Ruby auf?
  - Übung im Implementieren kleiner Methoden

Klassen, Objekte und Variablen

Definition einer Klasse in Ruby

- **`class` Klassenname**
  Definitionen
  end
- **Konvention: Klassennamen mit Großbuchstaben**
- **Alle Definitionen gelten für die Klasse und ihre Objekte**
- **Objekte erschaffen : Klasse.new (Parameter)**
  - erschafft frisches Objekt der Klasse
  - ruft Methode `initialize` auf, um Zustand (Daten) auf einen Anfangswert zu setzen
  - das Objekt ist Ergebnis

Klasse: Song	
Daten eines Liedes speichern	

```
class Song
  def initialize(name,
    artist, duration)
    @name = name
    @artist = artist
    @duration = duration
  end
end

Song.new („Wind of Change“,
  „Scorpions“, 310)
```

Klassen, Objekte und Variablen

Definition einer Klasse in Ruby

- **Zustand eines Objektes liegt in Instanzvariablen des Objektes**
- **Instanzvariablen**
  - beginnen mit @
  - jedes Objekt hat sein Exemplar der Instanzvariablen
  - Bsp: jeder Song hat seinen @name
  - leben, so lange das Objekt lebt
- **Zuweisung @name = name**
  - definiert, dass es Instanzvariable @name gibt
  - Setzt ihren Wert auf Wert von name
- **Meist: alle Objekte haben Instanz-variablen gleichen Namens**

Klasse: Song	
Daten eines Liedes speichern	

```
class Song
   def initialize(name,
     artist, duration)
     @name = name
     @artist = artist
     @duration = duration
   end
end

Song.new („Wind of Change“,
   „Scorpions“, 310)
```

Klassen, Objekte und Variablen

Definition einer Klasse in Ruby

- **Kommentieren was die Klasse tut!**
- **Kommentieren was die Instanzvariablen bedeuten!**
- **Gute Kommentare**
  - sind aussagekräftig
  - wiederholen nicht nur den Namen
  - beantwortet Fragen, die ein Leser wohl hat

```
# represents a song to be played by the jukebox
class Song
  def initialize(name, artist, duration)
    @name = name      # title of the song
    @artist = artist # name of the artist/group performing
    @duration = duration # in seconds
  end
end
```

Klassen, Objekte und Variablen

Die Methode .to_s

- **bedeutet „to String“**
- **definiert für alle Objekte**
- **definiert in Klasse `Object`**
- **konvertiert das Objekt in eine Zeichenkette**
- **damit man sich den Inhalt ansehen kann**
- **Vorgang**
  - wir rufen `puts song` auf
  - `puts` ruft `to_s` auf
  - da `song` ein `Song` ist, wird `Song.to_s` ausgeführt,
  - obwohl `puts` von `Song` nichts weiß
  - ⇒ Polymorphie

```
class Song
  def to_s
    „Song #@name – by: #@artist
  (#@duration)“
  end
end
```

Klassen, Objekte und Variablen

Vererbung

- **`class` Klassenname < Oberklasse**
  Definitionen
  end
- **Klasse ist abgeleitet von Oberklasse**
  - erbt alle Definitionen (besonders Methoden)
  - erweitert sie um eigene
- **KaraokeSongs sind auch Songs**

Klasse: KaraokeSong	
Untertitel speichern	abgeleitet von Song TimedLyrics

```
class KaraokeSong < Song
  def initialize(name, artist,
  duration, lyrics)
    @name = name
    @artist = artist
    @duration = duration
    @lyrics = lyrics
  end
end
```

Klassen, Objekte und Variablen

Frage an das Auditorium:

Was ist hieran unschön?

```
class KaraokeSong < Song
  def initialize(name, artist,
  duration, lyrics)
    @name = name
    @artist = artist
    @duration = duration
    @lyrics = lyrics
  end
end
```

Klassen, Objekte und Variablen

Frage an das Auditorium:

Was ist hieran unschön?

- **Die Klasse Song ist für `@name`, `@artist` und `@duration` verantwortlich.**
- **Alle anderen sollen über Methoden von Song damit arbeiten.**
- **⇒ Datenkapselung**
- **Aufrufen von `Song.initialize`**
- **`super` ruft namensgleiche Methode der Oberklasse auf**

```
class KaraokeSong < Song
  def initialize(name, artist,
  duration, lyrics)
    super(name, artist, duration)
    @lyrics = lyrics
  end
end
```

Klassen, Objekte und Variablen

Frage an das Auditorium:

Warum fehlen die Lyrics in der Ausgabe?

- **Song.to_s wird geerbt und von puts aufgerufen**
- **Methode KaraokeSong.to_s erneut definieren**
- **dabei Song.to_s aufrufen**

```
class KaraokeSong < Song
  def to_s
    super + „ [#@lyrics]“
  end
end
```

Klassen, Objekte und Variablen

Attribute

- **Klasse kontrolliert den Zugriff auf ihre Daten**
  - ⇒ Datenverkapselung (Geheimnisprinzip)
  - Instanzvariablen (z.B. `@name`) sind nur innerhalb der Klasse zugreifbar
  - kein `song1.@name`
- **von außen zugreifbare Eigenschaften**
  - heißen Attribute
  - werden explizit durch Zugriffmethode verfügbar gemacht
  - Lesemethode oft mit selbem Namen ohne @

```
class Song
   def name
         @name
   end
   def artist
         @artist
   end
   def duration
         @duration
   end
end

song = Song.new („Winds of
   Change“, „Scorpions“, 310)
puts song.name
puts song.artist
puts song.duration
```

Klassen, Objekte und Variablen

Attribute

- **einfacher: namensgleiche Lesemethode durch `attr_reader` erzeugen**
- **`:name` ist das Symbol name**
- **name ist der Wert von name**
- **wenn a=b=7,**
  - dann a==b,
  - aber nicht `:a==:b`

```
class Song
  attr_reader :name, :artist,
  :duration
end
```

Klassen, Objekte und Variablen

Schreibbare Attribute

- **besonders schreiben muss von der Klasse kontroliert werden**
- **Datenkapselung / Geheimnisprinzip**
- **im Prinzip: Methode set_name**
- **besser: Methode name=**
  - erlaubt Zuweisung zu schreiben
  - Notation, wie bei Variablen
  - aber Ausführung über Methode

```
class Song
  def duration=(duration)
    @duration=duration
  end
end

song.duration = 400
```

Klassen, Objekte und Variablen

Schreibbare Attribute

- **kürzer mit `attr_writer`**

```
class Song
  attr_writer :duration
end

song.duration = 400
```

Klassen, Objekte und Variablen

Warum Attribute, nicht Variablen?

- überprüfen von Gültigkeiten
- berechnen von Werten aus internem Zustand
- aktualisieren abhängiger Daten
- invalidieren abhängiger Daten
- all dies vor dem Nutzer der Klasse versteckt
- ⇒ Geheimnisprinzip

```
class Song
  def duration_in_min ()
    @duration/60.0
  end
  def duration_in_min= (duration)
    @duration = duration*60
  end
end

song.duration_in_min = 3
puts song.duration
```

Klassen, Objekte und Variablen

Warum Attribute, nicht Methoden?

- **einfachere Notation für den Nutzer**
- **Botschaft: verhält sich von außen wie eine Variable**
  - lesen ändert nichts
  - nach `attribute=value` ergibt `attribute` als Ergebnis `value`
  - erst `attribute=value1` und dann `attribute=value2` wirkt, wie nur `attribute=value2`
- **wenn etwas sich so verhält, ist es ein Attribut, sonst eine allgemeine Methode**

```
class Song
   def duration_in_min ()
     @duration/60.0
   end
   def duration_in_min= (duration)
     @duration = duration*60
   end
end

song.duration_in_min = 3
puts song.duration
```

Klassen, Objekte und Variablen

Klassenmethoden

- manchmal gehören Methoden konzeptuell zu einer Klasse, aber arbeiten nicht auf einem konkreten Objekt
- Bsp.: new
- Bsp.: Hilfsberechnungen
- Bsp.: Spezielle Arten Objekte zu erstellen
- starten im Namen mit Klasse und Punkt
- können nicht auf Instanzvariablen zugreifen
- können nur Klassenmethoden aufrufen

```
class Song
  def Song.min_to_sec(min)
    min*60.0
  end
  def Song.recorded(duration)
    Song.new („recorded“, nil,
       duration)
  end
end

song.duration_in_min = 3
puts song.duration
```

Klassen, Objekte und Variablen

Zugriffskontrolle

- **nur eigenes Objekt hat Zugriff auf Instanzvariablen**
- **Zugriff auf Methoden kontrolliert**
  - public (Vorgabewerte): alle
  - protected: nur eigene Klasse und Unterklassen
  - private: eigenes Objekt (`self`)
- **Ziel:**
  - nicht Sicherheit (gegen Böswilligkeit)
  - dem Nutzer zeigen, welche Methoden für ihn gedacht sind
  - Interne Details verstecken

```
class Song
   public :name, :artist, :duration
   protected :min_to_sec
end

song.duration_in_min = 3
puts song.duration
```

Klassen, Objekte und Variablen

Frage an das Auditorium:

- **Die Klasse `Secret` hat eine Methode `openLock`. Diese ist `private`, kann also nicht von außen aufgerufen werden.**
  Wie kommt man trotzdem dran?

```
class Secret
   def openLock
      …
   end
   private :openLock
End

Class SecretUnlocked < Secret
  def openLockForEveryone
    openLock
  end
  public :openLockForEveryone
end
```

Klassen, Objekte und Variablen

Variable

- **eine Variable ist ein „Platzhalter“ für eine Objektreferenz**
- **Frage an das Auditorium: Was wird ausgegeben?**

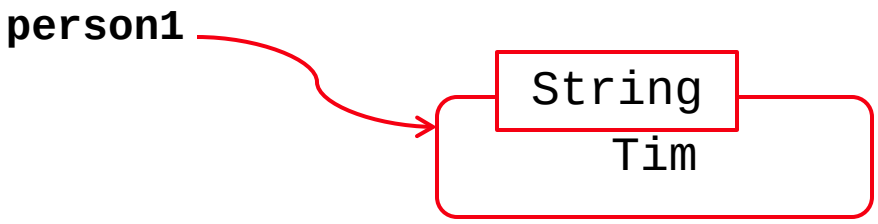
```
person1 = „Tim“
person2 = person1
person1[0] = „J“
puts person1
puts person2
```

Klassen, Objekte und Variablen

Variable

- **eine Variable ist ein „Platzhalter“ für eine Objektreferenz**
- **Frage an das Auditorium: Was wird ausgegeben?**
  - Objekt der Klasse String mit Daten „Tim“ wird erschaffen
    `person1` hält Referenz auf dieses Objekt

```
person1 = „Tim“
person2 = person1
person1[0] = „J“
puts „#{person1} grüßt #{person2}“
```

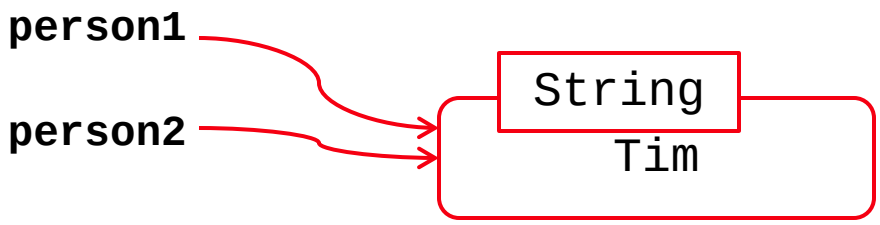


Klassen, Objekte und Variablen

Variable

- **eine Variable ist ein „Platzhalter“ für eine Objektreferenz**
- **Frage an das Auditorium: Was wird ausgegeben?**
  - Objekt der Klasse String mit Daten „Tim“ wird erschaffen
    `person1` hält Referenz auf dieses Objekt
  - `person2` erhält *dieselbe* Referenz

```
person1 = „Tim“
person2 = person1
person1[0] = „J“
puts „#{person1} grüßt #{person2}“
```

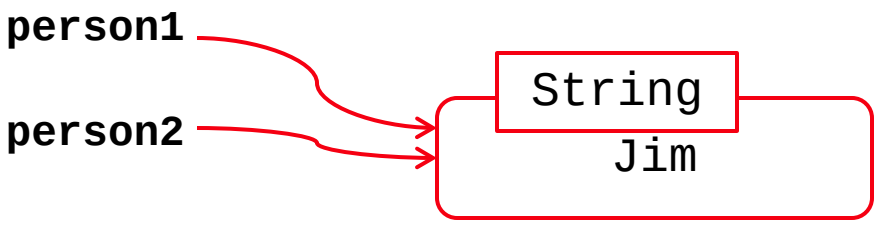


Klassen, Objekte und Variablen

Variable

- **eine Variable ist ein „Platzhalter“ für eine Objektreferenz**
- **Frage an das Auditorium: Was wird ausgegeben?**
  - Objekt der Klasse String mit Daten „Tim“ wird erschaffen
    `person1` hält Referenz auf dieses Objekt
  - `person2` erhält *dieselbe* Referenz
  - in dem einen String-Objekt wird der erste Buchstabe durch „J“ ersetzt

```
person1 = „Tim“
person2 = person1
person1[0] = „J“
puts „#{person1} grüßt #{person2}“
```

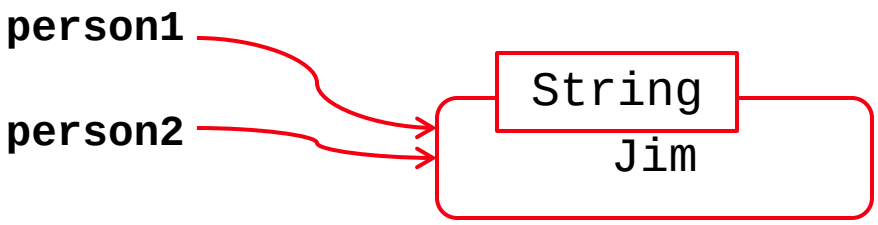


Klassen, Objekte und Variablen

Variable

- **eine Variable ist ein „Platzhalter“ für eine Objektreferenz**
- **Frage an das Auditorium: Was wird ausgegeben?**
  - Objekt der Klasse String mit Daten „Tim“ wird erschaffen
    `person1` hält Referenz auf dieses Objekt
  - `person2` erhält *dieselbe* Referenz
  - in dem einen String-Objekt wird der erste Buchstabe durch „J“ ersetzt
  - `person1` und `person2`, also zweimal das String-Objekt „Jim“ werden ausgegeben
  - „Jim grüßt Jim.“

```
person1 = „Tim“
person2 = person1
person1[0] = „J“
puts „#{person1} grüßt #{person2}“
```



Klassen, Objekte und Variablen

Variablen

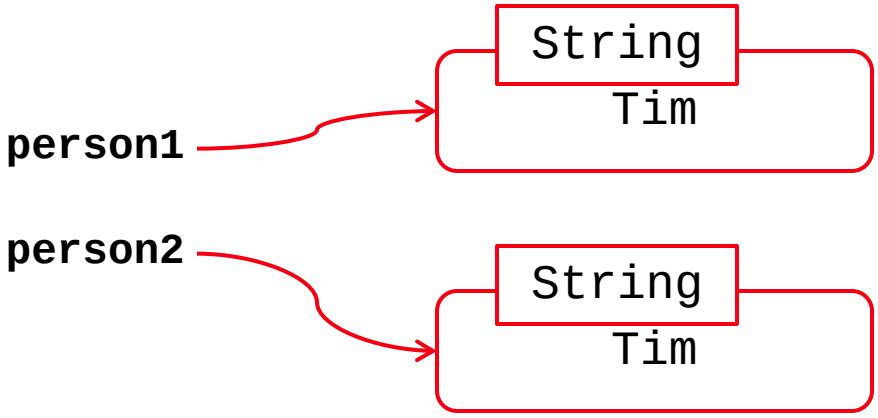
- **Gefahr: Veränderung von Strings erzeugt irgendwo anders im Programm ebenfalls veränderten String**
- **vor dem Verändern String kopieren mit `String.dup` (duplicate)**

Klassen, Objekte und Variablen

Variable

- **vor dem Verändern String kopieren mit `String.dup` (duplicate)**
  - Objekt der Klasse String mit Daten „Tim“ wird erschaffen
    `person1` hält Referenz auf dieses Objekt

```
person1 = „Tim“
person2 = person1.dup
person1[0] = „J“
puts „#{person1} grüßt #{person2}“
```

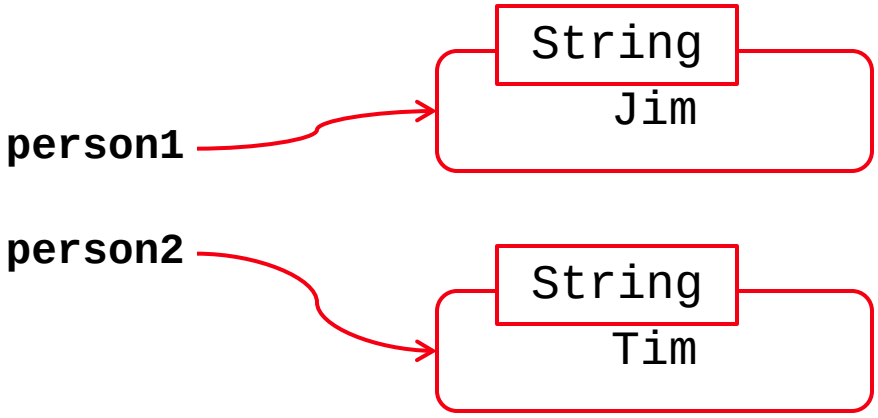


Klassen, Objekte und Variablen

Variable

- **vor dem Verändern String kopieren mit `String.dup` (duplicate)**
  - Objekt der Klasse String mit Daten „Tim“ wird erschaffen
    `person1` hält Referenz auf dieses Objekt
  - `person2` erhält *neues Objekt* mit kopiertem Inhalt
  - in dem ersten String-Objekt wird der erste Buchstabe durch „J“ ersetzt

```
person1 = „Tim“
person2 = person1.dup
person1[0] = „J“
puts „#{person1} grüßt #{person2}“
```

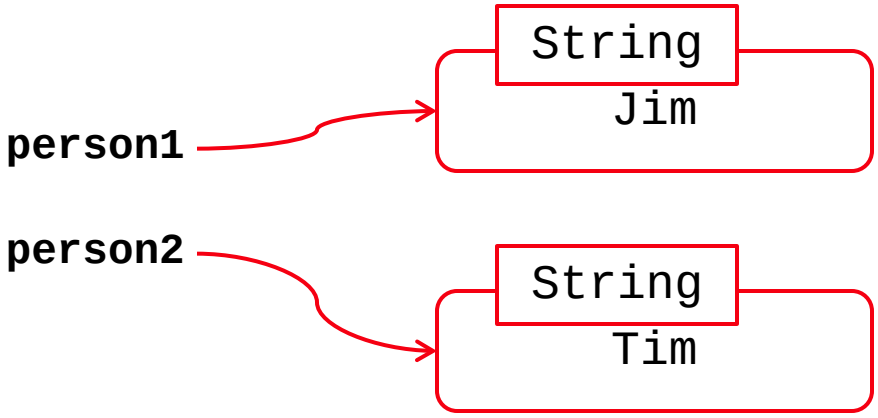


Klassen, Objekte und Variablen

Variable

- **vor dem Verändern String kopieren mit `String.dup` (duplicate)**
  - Objekt der Klasse String mit Daten „Tim“ wird erschaffen
    `person1` hält Referenz auf dieses Objekt
  - `person2` erhält *neues Objekt* mit kopiertem Inhalt
  - in dem ersten String-Objekt wird der erste Buchstabe durch „J“ ersetzt
  - `person1` und `person2`, also das erste und das zweite String-Objekt werden ausgegeben
  - „Jim grüßt Tim.“

```
person1 = „Tim“
person2 = person1.dup
person1[0] = „J“
puts „#{person1} grüßt #{person2}“
```



Beispiel: Die Klasse Bruch

Beispiel: Die Klasse Bruch

- **Wir wollen eine Klasse „Bruch“ entwickeln**
- **Frage an das Auditorium: Von welcher Klasse könnten wir ableiten?**

Klasse: Bruch	
- Darstellung eines Bruches - Rechnen mit Brüchen	abgeleitet von

Beispiel: Die Klasse Bruch

- **Wir wollen eine Klasse „Bruch“ entwickeln**
- **Frage an das Auditorium: Welche Instanzvariablen beschreiben den Zustand?**

Klasse: Bruch	
- Darstellung eines Bruches - rechnen mit Brüchen	abgeleitet von Zahl

Beispiel: Die Klasse Bruch

- **Wir wollen eine Klasse „Bruch“ entwickeln**
- **Frage an das Auditorium: Welche Instanzvariablen beschreiben den Zustand?**

Klasse: Bruch	
- Darstellung eines Bruches - rechnen mit Brüchen - nom, denom	abgeleitet von Zahl

Beispiel: Die Klasse Bruch

- **Wir wollen eine Klasse „Bruch“ entwickeln**
- **Frage an das Auditorium: Welche Methoden brauchen wir?**

Klasse: Bruch	
- Darstellung eines Bruches - rechnen mit Brüchen - nom, denom	abgeleitet von Zahl

Beispiel: Die Klasse Bruch

- **Wir wollen eine Klasse „Bruch“ entwickeln**
- **Frage an das Auditorium: Welche Methoden brauchen wir?**

Klasse: Bruch	
- Darstellung eines Bruches - rechnen mit Brüchen - nom, denom - initialize - to_s - +, -, *, / - to_f	abgeleitet von Zahl

Beispiel: Die Klasse Bruch

Operatoren

- **Methoden können den Namen von Operatoren, d.h. speziellen Symbolen haben, u.a.**
  - Arithmetik: +, -, *, /
  - Vergleich: <, >, =
  - Zugriff: []
- **Termen werden in Aufrufe dieser Methoden umgesetzt**
- **5+7 ⇒ 5.+(7)**
- **Programmes in Datei inifrese0903_Fraction.rb**

```
class Fraction < Numeric
 # implement adding self and other
   def +(other)
   end
end
```

Zusammenfassung

- **`class` Klassenname < Oberklasse Definitionen end**
- **`initialize` legt Instanzvariable (`@name`) an und weist einen Anfangswert zu (meist als Parameter übergeben)**
- **wird beim Erstellen eines Objektes mit `new` aufgerufen**
- **Methoden von Oberklasse werden geerbt**
  - können neu definiert werden
  - werden mit super aufgerufen
- **Instanzvariablen (z.B. `@name`) nicht von außen zugreifbar**
  - Methode `name` zum lesen (kurz `attr_reader`)
  - Methode `name=` zum schreiben (kurz `attr_writer`)
  - für Wertekontrollen, Aktualisierung, Invalidierung
- **Klassenmethoden Class.method**
- **Zugriffskontrolle für Methoden: `public`, `protected`, `private`**
- **Variablen sind Platzhalter für *Referenzen* auf Objekte**