PM1/PT Ruby: Klassendefinitionen

Prof. Dr. Birgit Wendholt

Konzepte

- Instanzvariablen (Attribute)
- Initialisierung
- Parameter
- sondierende und verändernde Methoden
- Zuweisung und bedingte Anweisung

Beispiel: Ein naiver Ticketautomat

- Unser Ticketautomat ist eine sehr einfache Variante der Automaten, wie wir sie auf Bahnhöfen antreffen.
- Dieses einfache Modell lässt sich jederzeit erweitern, um sich der Realität mehr anzunähern.
- Er arbeitet wie folgt:
  - Ein Kunde wirft Geld ein und fordert ein Ticket an.
  - Der Automat merkt sich, wie viel Geld während des Betriebs in ihn geworfen wurde.
  - Er kann Auskunft geben, über den für ein Ticket eingeworfenen Betrag.
- Wir öffnen in Eclipse das Projekt ***v02-NaiverTicketautomat*** und starten die ***Toolbox***
- Wie schon in den Beispielen vorher sehen wir uns altbekannte Klassen, die wir nur nutzen, aber nicht verstehen müssen
  - ***Toolbox***
  - ***ObjectInspector***
  - ***Einstellungen, EinstellungenTicketAutomat***
- Sowie die wichtige Klasse, die den Quelltext für den Automaten enthält:
  - ***TicketAutomat***

Übungen

Ü 2.1: Wir erzeugen eine Instanz von ***TicketAutomat*** und rufen die Methode ***ticket_preis*** auf. Ergebnis?

- Wir simulieren das Bezahlen durch die Methode ***geld_einwerfen*** und prüfen mit der Methode ***bisher_bezahlt***, ob der Automat unseren eingeworfenen Betrag registriert hat.
- Wir bezahlen das Ticket in mehreren Teilbeträgen.

Ü2.2: Welcher Wert wird geliefert, wenn wir den bereits bezahlten Betrag ausgeben lassen, nachdem wir das Ticket gedruckt haben?

Ü2.3 Wir experimentieren mit unterschiedlichen Beträgen, bevor wir das Ticket drucken.

- Was passiert, wenn wir zu viel Geld einwerfen? Gibt es Wechselgeld zurück? Was passiert, wenn wir zu wenig Geld einwerfen. Wird das Ticket gedruckt?

Ü2.4: Wer liefert uns die Information über die während des Betriebs des Automaten eingeworfenen Geldbeträge?

Ü2.5: Wir erzeugen einen zweite Instanz der Klasse TicketAutomaten mit einem anderen Ticketpreis. Wie sieht dann das gedruckte Ticket aus?

Instanzvariablen, Initialisierung und Methoden

- Der Quelltext der Klasse findet sich im Ruby Script ***TicketAutomat***.

- Der Quelltext der meisten Klassen lässt sich in zwei Bereiche aufteilen. Den schmalen äußeren Rahmen und den viel umfangreicheren Innenteil der Klasse.

Ü 2.6: Schreiben Sie bitte auf, wie die äußeren Rahmen der Klasse Student und Kreis aussehen!

```
class TicketAutomat

    // Innenteil der Klasse

end
```

Der Innenteil einer Klasse

- Der Innenteil enthält in den meisten Fällen
  - einen Initialisierungsteil, der die Instanzvariablen der Objekte der Klasse mit sinnvollen Werten belegt. Für den Initialisierungsteil ist der Methodennamen ***initialize*** verbindlich festgelegt.
  - einen Methodenteil, der eine oder mehrere Objekt-Methoden enthält. Diese Methodennamen sollten sprechend gewählt werden.
  - Methodendefinitionen beginnen immer mit ***def*** und werden immer mit ***end*** abgeschlossen

Ü 2.7: Erstellen Sie für die Klasse ***Kreis*** eine ähnliche Schablone wie die rechts für den Ticketautomaten.

```
class TicketAutomat

  // Initialisierungsteil
  def initialize(ticket_preis) ...
  end

  // Methodenteil
  def ticket_preis() ...
  end
  def bisher_bezahlt() ...
  end
  def geld_einwerfen(betrag)...
  end
  def ticket_drucken() ...
  end
  def to_s() ...
  end

end
```

Der Initialisierungsteil

- Im Initialisierungsteil, werden die Instanzvariablen der Objekte der Klasse mit sinnvollen Werten belegt.
- Instanzvariablen beginnen in Ruby immer mit genau einem **@** Zeichen, das fester Bestandteil des Namens ist.
- Instanzvariablen müssen immer in ***initialize*** definiert werden.
- Sie dürfen **niemals außerhalb** des ***initialize*** definiert werden, da sie dann keine Instanzvariablen sind.

```
class TicketAutomat

  def initialize(ticket_preis)
    @ticket_preis = ticket_preis
  end
```

Böse Falle bei Instanzvariablen

- Die Variable **@counter**, die im Block der Klasse definiert ist, ist **keine** Instanzvariable der Objekte der Klasse.
- Daher ist ***@counter*** auf der rechten Seite in ***initialize*** nicht definiert, also ***nil.***
- Die Addition von ***nil*** und 1 liefert einen uns schon bekannten Fehler:

  undefined method `+' for nil:NilClass (NoMethodError)

  MERKE: Instanzvariablen müssen immer innerhalb des ***initialize*** eingeführt und definiert werden.

```
class InstanzvariablenBoeseFalle

  @counter = 1
  def initialize()
    @counter = @counter +1
  end

  def counter
    @counter
  end
end

puts(InstanzvariablenBoeseFalle.new().
                                    counter)
```

Die Rolle von Instanzvariablen

- Betrachten wir den Quelltext der Klasse ***TicketAutomat*** genauer, so sehen wir, dass der ***@ticket_preis*** in mehreren Methoden genutzt wird.

- Instanzvariablen haben die Aufgabe, Informationen über Objekte für die **gesamte Lebensdauer** des Objektes zu speichern.

- Wir sagen: Instanzvariablen halten den **Zustand** des Objektes.

Ü2.9: Schreiben Sie für jede Instanzvariable der Klasse auf, wie häufig diese in Methoden-Definitionen benutzt wird.

Lebensdauer von Objekten

Ü2.10: Wie lange lebt das Objekt ***ticketAutomat1*** in der ***Toolbox***?

Ü2.11: Von wann bis wann lebt das Objekt ***ticket_automat*** in dem rechts stehenden Rubyscript?

```
require 'TicketAutomat'

ticket_automat =
   TicketAutomat.new(500)
```

Das Pärchen *new* und *initialize*

- Zu jedem ***initialize*** gehört eine Klassenmethode ***new*** mit exakt soviel Parametern wie in ***initialize***. ***new*** muss mit exakt dieser Parameterzahl aufgerufen werden.

- Für die Klasse ***TicketAutomat*** haben wir ein Beispiel auf der vorhergehenden Folie.

- ***Ü2.12:*** Wie sieht der Aufruf von ***new*** für ***Kreis***, ***Student*** und ***Laborkurs*** aus?

Das Pärchen *new* und *initialize*

- Wird auf einer Klasse die Methode ***new*** aufgerufen,
  - wird zuerst eine „leere“ Instanz erzeugt.
  - danach ruft die Klasse die Methode ***initialize*** auf dieser Instanz auf.

- In „leeren“ Instanzen sind die Instanzvariablen noch nicht mit sinnvollen Werten belegt. Sie haben den Wert des „NichtObjekts“ ***nil***.

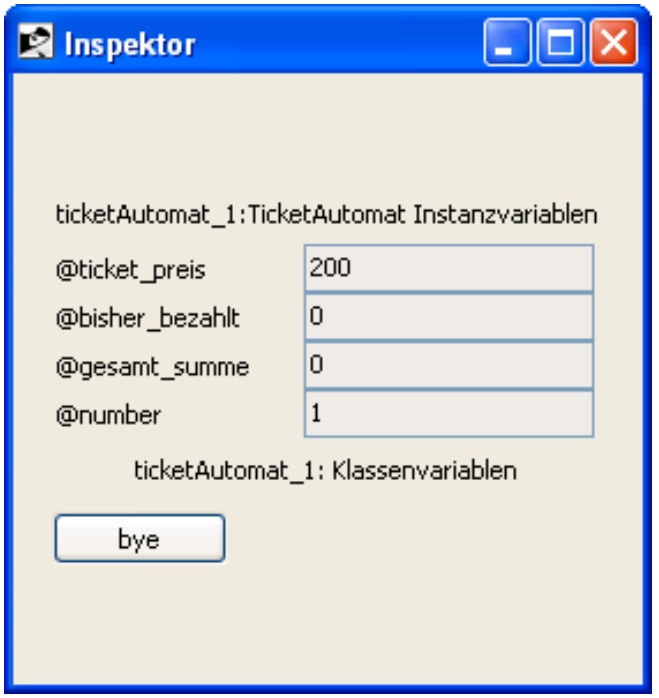
- Jedes Objekt außer ***nil*** antwortet auf die Methode ***nil?*** mit ***false***.

Ü2.13: Wir führen das Rubyscript ***objekt_vor_nach_initialisierung.rb*** aus. Was gibt das Script aus?

```ruby
class TicketAutomat
  def initialize(ticket_preis)
    puts 'Vor der Zuweisung'
    puts @ticket_preis.nil?
    puts @bisher_bezahlt.nil?
    puts @gesamt_summe.nil?
    puts @number.nil?
    @ticket_preis = ticket_preis
    @bisher_bezahlt = 0
    @gesamt_summe = 0
    @number = instanzen_zaehler
    puts 'Nach der Zuweisung'
    puts @ticket_preis
    puts @bisher_bezahlt
    puts @gesamt_summe
    puts @number
  end
end
TicketAutomat.new(200)
```

Ein Objekt nach der Initialisierung

- Nach der Erzeugung und Initialisierung eines Objektes der Klasse ***TicketAutomat*** mit einem Ticketpreis von 200 Cent, sieht das Objekt wie nebenstehend aus.



Kommentare sind kein ausführbarer Quelltext

- Kommentare (nach dem **#**) werden im Quelltext angegeben, um dem menschlichen Leser Erläuterungen zu geben.
- Sie haben **keinen** Einfluss auf die Funktionalität einer Klasse.

```ruby
class TicketAutomat

  # initialize: initialisiert die
  # Instanzvariablen der Objekte der
  # Klasse TicketAutomat
  # Die Methode wird auf einer mit new
  # erzeugten Instanz der Klasse
  # aufgerufen
  # Hier wird im initialize der
  # ticket_preis übergeben, der beim
  # Erzeugen den Objektes mit new
  # ebenfalls übergeben werden muss.

  def initialize(ticket_preis)
    ...
  end
```

Datenübergabe mit Parametern

- Methoden erhalten beim **Aufruf** Werte über Parameter.
- Parameter werden im Kopf einer Methode definiert.

```
class TicketAutomat

  def initialize(ticket_preis)
    @ticket_preis = ticket_preis
    @bisher_bezahlt = 0
    @gesamt_summe = 0
    @number = instanzen_zaehler()
  end

    ...
end
```

Datenübergabe mit Parametern

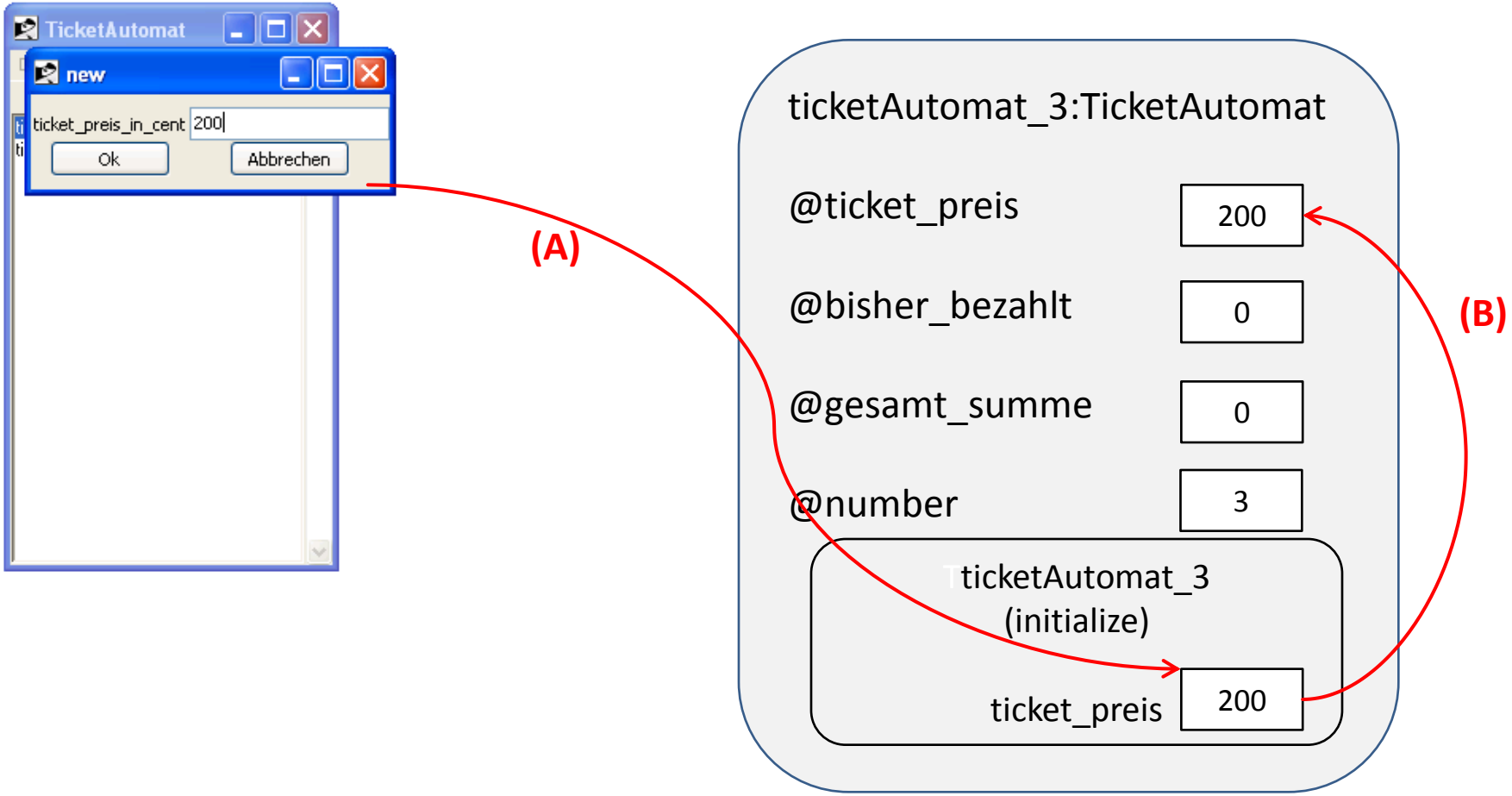
- Wird beim Aufruf von ***initialize*** der Wert 200 übergeben (diesen haben wir über einen Dialog der Toolbox eingegeben), dann wird der Wert in den Parameter ***ticket_preis*** der Methode kopiert (nächste Folie Abb. Pfeil (A)).

- Der graue Kasten mit der Überschrift ***initialize*** stellt zusätzlichen Platz dar, der nur während der Ausführung der Methode zur Verfügung steht. Wir nennen diesen den *Methodenspeicher*.

- Der *Methodenspeicher* bietet Platz für die Werte der Methodenparameter und andere Variablen, die wir später noch kennenlernen.

- Beim Ausführen der Methode ***initialize*** wird der Werte aus dem Methodenparameter ***ticket_preis*** in die Instanzvariable ***@ticket_preis*** übertragen. (Pfeil (B))

Datenübergabe mit Parametern



Formale und aktuelle Parameter

- Wir unterscheiden zwischen den ***formalen Parametern*** innerhalb einer Methode und den ***aktuellen Parametern*** außerhalb der Methode.
- Im vorausgehenden Beispiel ist ***ticket_preis*** der *formale Parameter* und der **Wert 200** der *aktuelle Parameter.*

Ü2.14: Wir bestimmen alle formalen und aktuellen Parameter im untenstehenden Quelltext.

```
class Kreis
 def horizontal_bewegen(anzahl)
    bewegen_um_punkt(Point.new(anzahl,0))
  end

 def bewegen_um_punkt(punkt)
    @mittelpunkt = @mittelpunkt + punkt
    if (sichtbar?)
      Leinwand.gib_einzige_instanz().bewege(self,punkt.x,punkt.y)
    end
  end
end
```

Sichtbarkeit und Lebensdauer

- Ein formaler Parameter steht einem Objekt nur im Rumpf der Methode zur Verfügung.
- Die ***Sichtbarkeit* eines formalen Parameters** ist auf den Methodenrumpf beschränkt.
- Hingegen ist die **Sichtbarkeit von Instanzvariablen** die gesamte Klassendefinition – sie können in allen Objektmethoden der Klassendefinition verwendet werden.
- Die ***Lebensdauer eines aktuellen Parameters*** ist auf die Ausführung der Methode beschränkt.
- Nach dem Ausführen der Methode verschwindet der Methodenspeicher und mit ihm alle Parameterwerte, die in ihm gehalten wurden.
- Die ***Lebensdauer einer Instanzvariable*** entspricht dagegen der Lebensdauer eines Objektes.
- Daher müssen wir, um uns den ***ticket_preis*** merken wollen, diesen in einen beständigeren Speicher, die Instanzvariable ***@ticket_speicher*** übertragen.

Zuweisungen

- Zuweisungen speichern den Wert auf der rechten Seite der Zuweisung in der Variablen die auf der linken Seite steht.
- Die rechte Seite der Zuweisung ist ein Ausdruck, d.h. etwas, was einen Wert liefert.

- Eine einfache Zuweisung

```
@ticket_preis = ticket_preis
```

Zuweisungen

Ü2.15: Wir nehmen an, dass wir eine Klasse ***Song*** mit einer Instanzvariable für den **Titel** haben. Schreiben Sie bitte die Methode mit der wir die Instanzvariable für den Titel initialisieren können!

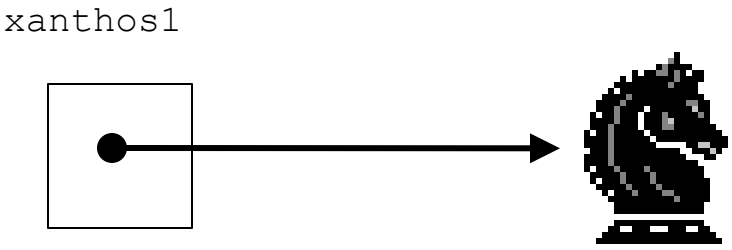
Ü2.16: Was ist falsch in der nebenstehenden Version der Initialsierung eines ***TicketAutomaten***?

```
class TicketAutomat

    def initialize(preis)
       ticket_preis = preis
       ...
    end
end
```

Was genau passiert bei Zuweisungen zu Variablen?

- Betrachten wir zunächst, was bei Zuweisungen von Objekten auf Variablen passiert.
- Wir erzeugen ein Pferd und referenzieren dieses Objekt über ***xanthos1.***
- Wir stellen uns Variablen als Kästchen vor, die auf Objekte zeigen.
- Dann entspricht der Zuweisung das Bild rechts unten.
- Der Pfeil steht für die Referenz auf das Pferde Objekt.

```
xanthos1 = Pferd.new(1.9,65,0)
```

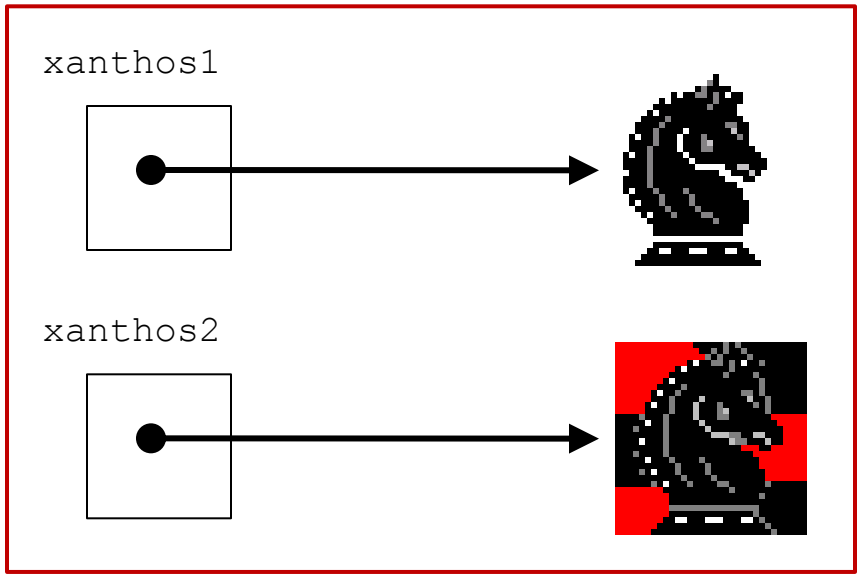


Was genau passiert bei Zuweisungen zu Variablen?

- Wir erzeugen ein zweites Pferd und referenzieren dieses Objekt über ***xanthos2.***

```
xanthos1 = Pferd.new(1.9,65,0)
xanthos2 = Pferd.new(1.9,65,0)
```

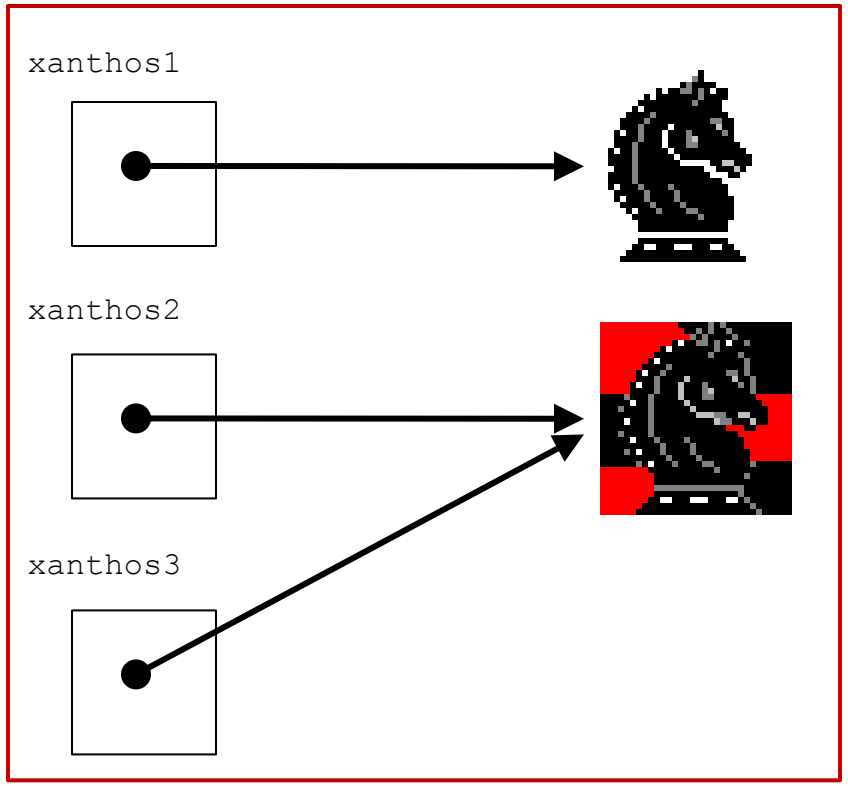
- Das Bild rechts unten zeigt das Ergebnis dieser Zuweisungen
- Wir haben zwei Variablen, die auf unterschiedliche Objekte zeigen.



Was genau passiert bei Zuweisungen zu Variablen?

- Nun weisen wir einer Variable ***xanthos3*** den Wert von ***xanthos2*** zu.
- ***xanthos3*** ist so etwas wie ein Zweitname oder Spitzname (Alias) für das Pferdeobjekt.
- ***xanthos3*** und ***xanthos2*** zeigen auf dasselbe Objekt.

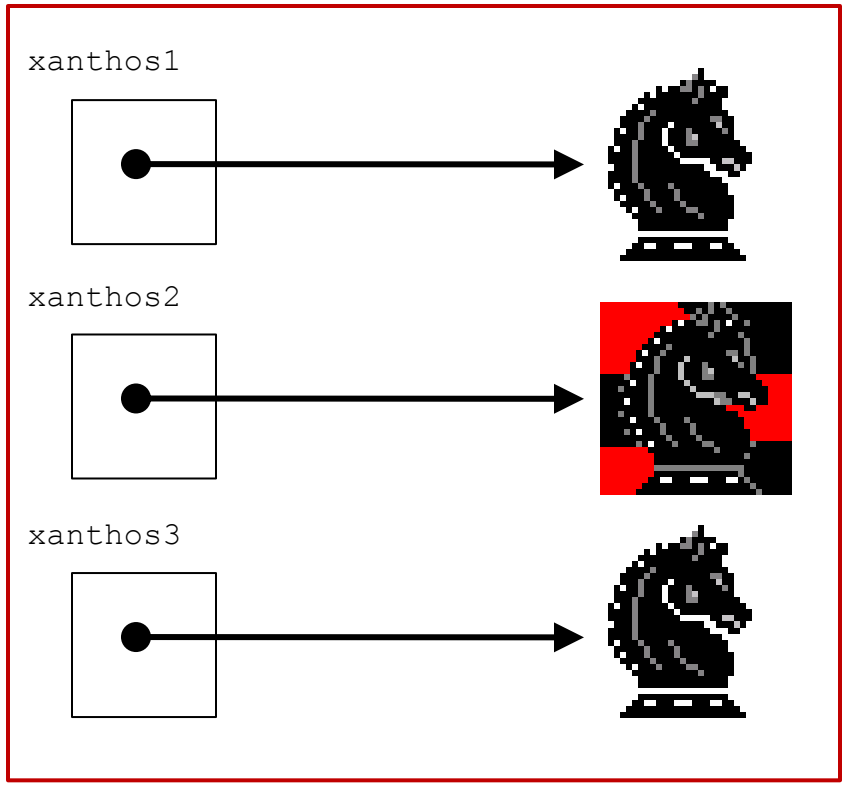
```
xanthos1 = Pferd.new(1.9,65,0)
xanthos2 = Pferd.new(1.9,65,0)
xanthos3 = xanthos2
```



Was genau passiert bei Zuweisungen zu Variablen?

- Im finalen Schritt entscheiden wir uns um und wollen nun doch ***xanthos3*** ein eigenes Objekt zuweisen.
- Dann zeigt ***xanthos3*** nach der Zuweisung auf das neu erzeugte Pferdeobjekt.
- ***xanthos2*** zeigt nach wie vor auf das rot hinterlegte Pferdeobjekt.

```
xanthos1 = Pferd.new(1.9,65,0)
xanthos2 = Pferd.new(1.9,65,0)
xanthos3 = xanthos2
xanthos3 = Pferd.new(1.9,65,0)
```

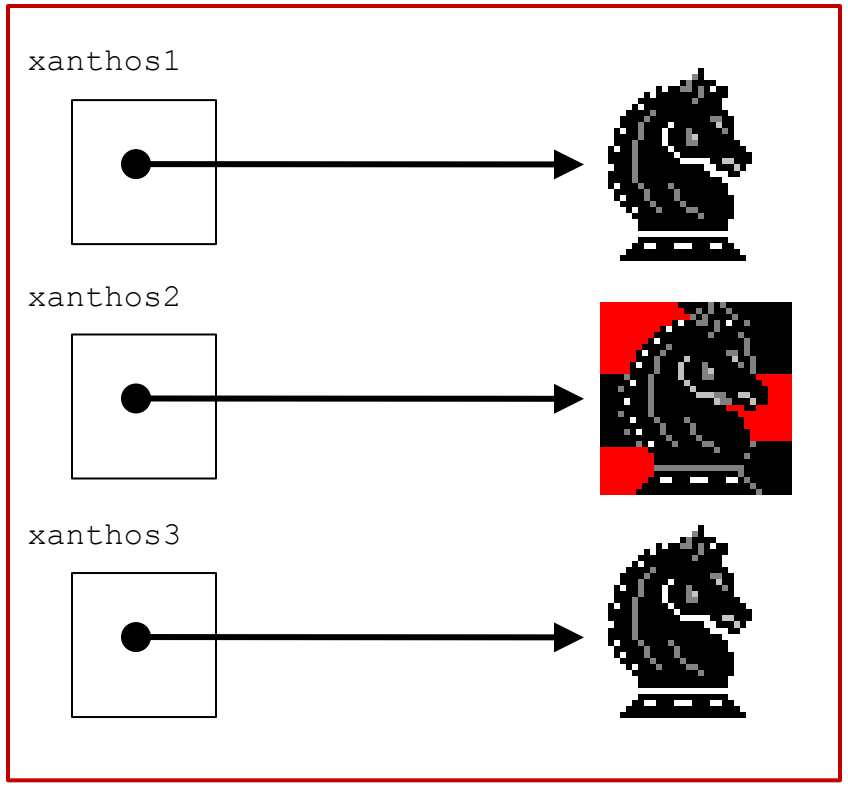


Was genau passiert bei Zuweisungen zu Variablen?

Das heißt:

- wenn zwei Variable ***xanthos2***, ***xanthos3*** auf dasselbe Objekt zeigen
- und einer dieser Variablen (***xanthos3***) ein neues Objekt zugewiesen wird,
- ändert sich der Inhalt der anderen Variablen (***xanthos2***) nicht.

```
xanthos1 = Pferd.new(1.9,65,0)
xanthos2 = Pferd.new(1.9,65,0)
xanthos3 = xanthos2
xanthos3 = Pferd.new(1.9,65,0)
```



Was genau passiert beim Aufruf einer Methode mit Parametern?

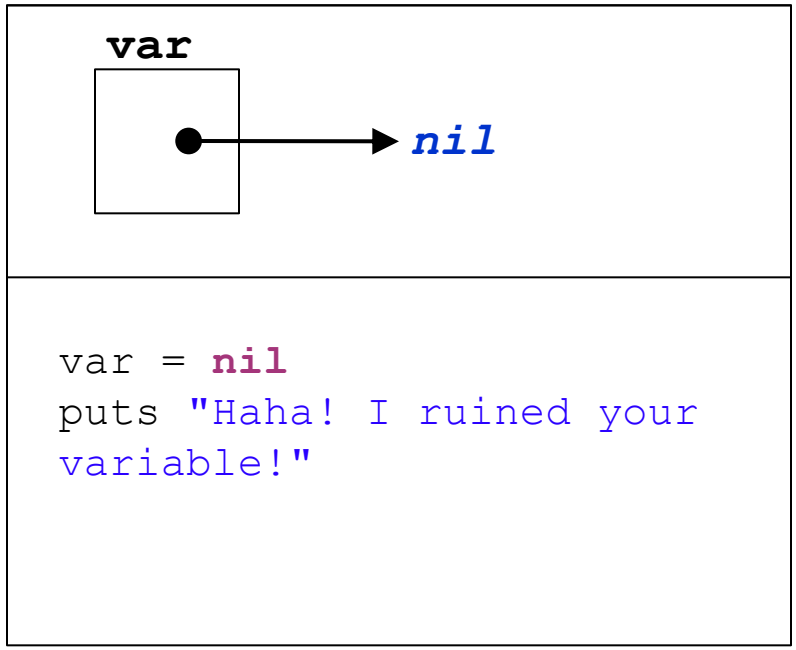
- Wir betrachten die Methode ***little_pest***.
- Die Methode hat einen formalen Parameter ***var*** (❶) .
- In einer Graphik veranschaulichen wir die Methode und trennen den Methodenrumpf von den Methoden-parametern.

```
def little_pest(var) ❶
  var = nil
  puts "Haha! I ruined your var!"
end
```

Was genau passiert beim Aufruf einer Methode mit Parametern?

- In oberen Rechteck stehen die formalen Parameter = lokalen Variablen der Methode (hier nur ***var***).
- In der Definition der Methode ist der Parameter ***var*** nicht initialisiert und zeigt auf ***nil***.
- Im unteren Rechteck steht der Methodenrumpf. Dieser verwendet den formalen Parameter und weist diesem einen neuen Wert zu.

little_pest



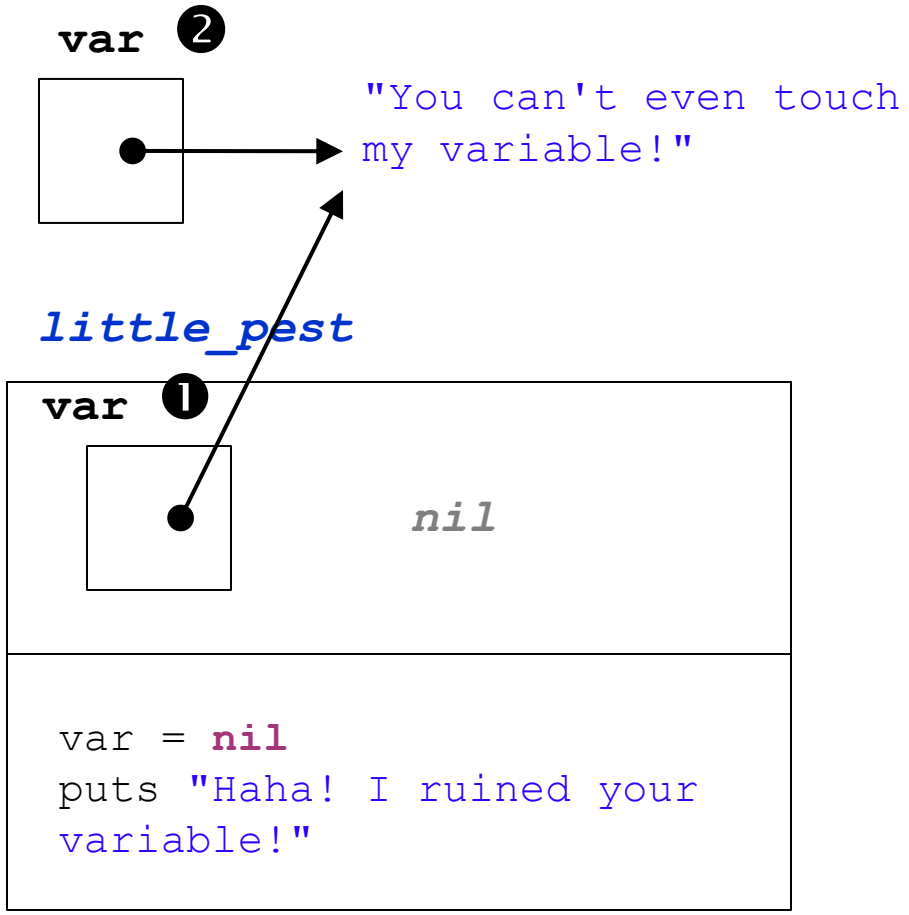
Was genau passiert beim Aufruf einer Methode mit Parametern?

- Im äußeren Programm weisen wir der Variablen ***var*** (❷) eine Zeichenkette zu.
- Und rufen die Methode ***little_pest*** mit dem aktuellen Parameter ***var*** (❷) auf.
- Die folgenden Seite veranschaulichen das Prinzip der Parameterübergabe anhand von Graphiken.

```
def little_pest(var)❶
  var = nil
  puts "Haha! I ruined your variable!"
end

❷var = "You can't even touch my
variable!"

little_pest(var)
```

Was genau passiert beim Aufruf einer Methode mit Parametern?

```
❷var = "You can't even
touch my variable!"

little_pest(var)
```

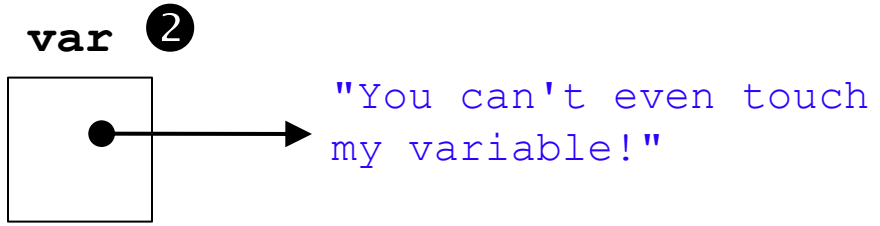


❷ Der aktuelle Parameter (***var*** ❷), der Wert ***"You can't even touch my variable!“***, wird beim Methoden-aufruf dem formalen Parameter (***var*** ❶) zugewiesen.

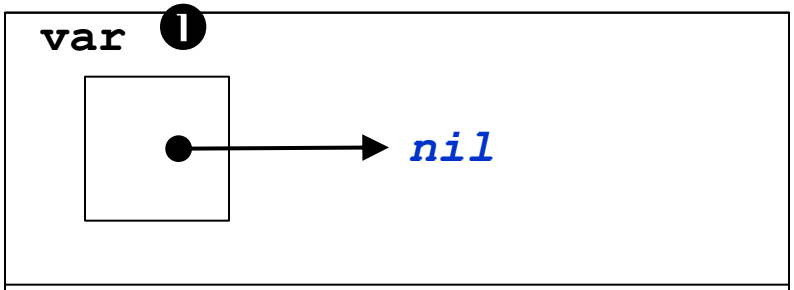
Zwei unterschiedliche Variablen zeigen jetzt auf dieselbe Zeichenkette.

Was genau passiert beim Aufruf einer Methode mit Parametern?

- Jetzt wird der Methodenrumpf von ***little_pest*** ausgeführt.
- Hier wird dem formalen Parameter, der lokalen Variablen ***var*** ❶ das ***nil*** Objekt zugewiesen.
- ***var*** ❷ bleibt unverändert und zeigt auf das ursprüngliche String-Objekt.



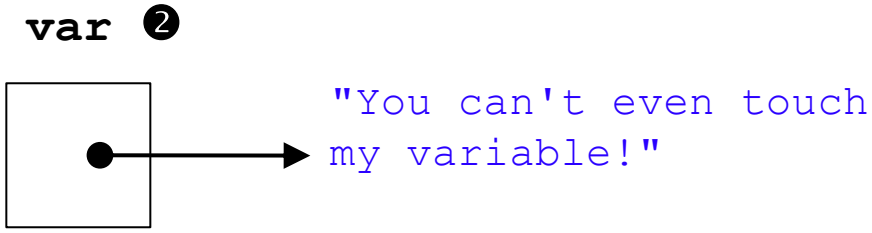
little_pest




```
var = nil
puts "Haha! I ruined your variable!"
```

Was genau passiert beim Methodenaufruf mit Parametern

- Deshalb liefert die Ausgabe von ***var*** ❷ auch das ursprüngliche String-Objekt.




```
❷ var = "You can't even touch
my variable!"
little_pest(var)
puts var ❷
```

"You can't even touch my variable!"

Sondierende Methoden

- Sondierende Methoden liefern Informationen über ein Objekt.
- Sie **verändern** den **Zustand** des Objektes **nicht!**
- Bsp.: ***ticket_preis()***. Diese Methode gibt Auskunft über den Preis der Tickets einer Ticketautomaten-Instanz.

Ü2.17: Bitte bestimmen Sie alle sondierenden Methoden der Klasse ***TicketAutomat***!

```ruby
class TicketAutomat

  # Liefert den @ticket_preis
  # des Automaten
  def ticket_preis()
     return @ticket_preis
  end
end
```

Methodendefinitionen

- Methodendefinitionen bestehen aus
  - einem Rahmen ***def … end***, der die Definition der Methode begrenzt und den Block der Methode definiert.
  - einem Kopf ***ticket_preis()***, der den Methodennamen und die Liste der formalen Parameter nennt
  - einem Rumpf, der einen Deklarationsteil (für lokale Variablen) und einen Anweisungsteil enthält.
- Typisch für sondierenden Methoden ist die ***return*** Anweisung, die das Ergebnis der Methode zurückliefert.
- Nach dem Ausführen der ***return*** Anweisung wird die Methode beendet.
- Der Rumpf der Methode wird **ausgeführt**, wenn die Methode auf einem Objekt **aufgerufen** wird.
- Beim Aufruf werden der Methode aktuelle Parameter von der aufrufenden Umgebung übergeben.
- Der Rumpf der Methode wird **nicht** zum Zeitpunkt der Definition der Methode ausgeführt.

Ü2.18: Schreiben Sie bitte eine sondierende Methoden der Klasse ***TicketAutomat,*** die die ***@gesamt_summe*** zurückliefert***!***

Verändernde Methoden

- Die Methoden ***geld_einwerfen*** und ***ticket_drucken*** ändern den Wert eines oder mehrere Instanzvariablen.
- Sie ändern den Zustand des Objektes, wenn Sie aufgerufen werden.
- Charakteristisch für verändernde Methoden ist, dass sich ein Objekt nach deren Aufruf anders verhält als vor dem Aufruf.
- **Ü2.19:** Erzeugen Sie bitte einen TicketAutomat und rufen als erstes die Methode ***bisher_bezahlt*** auf. Dann rufen Sie die Methode ***geld_einwerfen*** und erneut ***bisher_bezahlt*** auf.

```
def geld_einwerfen(betrag)
  @bisher_bezahlt =
 @bisher_bezahlt + betrag
end

def ticket_drucken()
  puts "------------------"
  puts "- HAW Express-Line"
  puts "- Ticket"
  puts "- " + ticket_preis + "
 Cent"
  puts "------------------"
  @gesamt_summe = @gesamt_summe +
 @bisher_bezahlt
  @bisher_bezahlt = 0
end
```

Verändernde Methoden

- Im Kopf einer verändernden Methode werden formale Parameter definiert, die im Rumpf verwendet werden, um eine oder mehrere Instanzvariablen zu ändern.
- Der beim Aufruf der Methode übergebene aktuelle Parameter wird dann für diese Änderungen verwendet.
- Im Rumpf der Methoden sehen wir zwei Formen von Anweisungen:
  - ein einfache Variablenzuweisung
  - eine Zuweisung eines Wertes, der Ergebnis einer arithmetischen Berechnung ist.
- Wir sagen auch, dass einer Variable der Wert eines arithmetischen Ausdrucks zugewiesen wird.

```
def geld_einwerfen(betrag)
  @bisher_bezahlt =
 @bisher_bezahlt + betrag
end

def ticket_drucken()
  puts "------------------"
  puts "- HAW Express-Line"
  puts "- Ticket"
  puts "- " + ticket_preis + "
 Cent"
  puts "------------------"
  @gesamt_summe = @gesamt_summe +
 @bisher_bezahlt
  @bisher_bezahlt = 0
end
```

Objektverhalten nach Aufruf verändernder Methoden

Ü2.20: Schreiben Sie bitte eine Methode ***setze_ticket_preis,*** die einen neuen Ticketpreis setzt!

Ü2.21: **S**chreiben Sie bitte eine Methode ***reduzieren,*** die den Ticketpreis um einen übergebenen Betrag reduziert!

Spezialformen sondierender und verändernder Methoden

- Spezialformen von sondierenden Methoden sind Methoden, die genau eine Instanzvariable lesen.
- Spezialformen von verändernden Methoden sind Methoden, die genau eine Instanzvariable ändern.
- In Ruby heißen diese Spezialformen für sondierende und verändernde Methoden:
  - **Attribut-Reader**
  - **Attribut-Writer**

- Reader heißen wie die Instanzvariable ohne das **@** Zeichen
- Writer heißen wie Instanzvariablen ohne das **@** Zeichen gefolgt von einem **=** Zeichen, gefolgt von einem formalen Parameter, der beim Aufruf den neuen Wert für die Instanzvariable aufnimmt.

Ü2.21: Erfüllen die sondierenden Methoden der Klasse ***TicketAutomat*** diese Konvention?

Ü2.22: Schreiben Sie bitte die Methode ***setze_ticket_preis*** nach der Konvention für Writer in Ruby.

Abkürzung für Reader und Writer

- Wenn Reader und Writer Instanzvariablen nur lesen oder schreiben, gibt es Abkürzung für diese beiden Spezial-formen.
- ***attr_reader :ticket_preis*** wird übersetzt in eine Reader-Methode für die Instanzvariable ***@ticket_preis***
- ***attr_writer :ticket_preis*** wird übersetzt in eine Writer-Methode für die Instanzvariable ***@ticket_preis.***
- Die beiden Abkürzungsformen werden im Block der Klasse auf gleicher Ebene wie die Methodendefinition verwendet.

- **Ü2.23:** Löschen Sie bitte die Reader und Writer Definitionen für ***@ticket_preis*** und ersetzen diese durch die Kurzformen! Testen Sie bitte, dass nach dieser Ersetzung, die Reader und Writer vorhanden sind!

Bewertung des Entwurfs des naiven Ticketautomaten

- Keine Prüfung, ob der Kunde genug Geld für ein Ticket eingeworfen hat.
- Es wird kein Geld zurückgegeben, wenn der Kunde zu viel für das Ticket bezahlt hat
- Es wird nicht überprüft, ob der Kunde sinnvolle Beträge einwirft. (Es werden z.B. negative Beträge akzeptiert)
- Es wird nicht geprüft, ob der Ticketpreis bei der Initialisierung sinnvoll ist.