Universität Osnabrück

Fachbereich Mathematik/Informatik

Seminararbeit

zum Seminar "Programmiersprachen" im Sommersemester 2011

Thema: Die Programmiersprache Ruby

eingereicht von: Manuel Schwarz <manschwa@uos.de>
Matrikelnr.: 935488

eingereicht am: 15. Juni 2011

Betreuerin: Frau Dr. rer. nat. Jutta Göers

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Die vorliegende Seminararbeit entstand im Rahmen des Seminars "Programmiersprachen" an der Universität Osnabrück im Sommersemester 2011 unter der Leitung von Dr. rer. nat. Jutta Göers als Ausarbeitung eines 45-minütigen Vortrages.
Das Seminar hatte zum Ziel Einführungen in eine Reihe von Programmiersprachen zu den unterschiedlichsten Paradigmen zu geben. Sowohl die vorliegende Ausarbeitung als auch der zugehörige Vortrag beschäftigen sich mit der Programmiersprache Ruby und erheben keinesfalls einen Anspruch auf Vollständigkeit. Sie stellen lediglich eine Einführung in die Thematik und die Sprache dar. Der interessierte Leser möge sich zwecks Vertiefung an die angegebene Literatur halten. Die beiden Quellen, auf die sich diese Arbeit hauptsächlich bezieht, sind zum einen das Buch "*The Ruby Programming Language*"[1], an dem der Entwickler von Ruby (Yukihiro Matsumoto) selbst mitgearbeitet hat, und zum anderen das Buch "*Programming Ruby 1.9 - The Pragmatic Programmers' Guide*"[2].

Als kleiner Vorgeschmack auf die folgenden Kapitel sollen hier vorweg ein paar Besonderheiten genannt werden, die die Programmiersprache Ruby auszeichnen. An erster Stelle ist Ruby für reine Objektorientierung bekannt (alles ist ein Objekt) und gilt als sehr dynamisch. Die Speicherverwaltung ist, wie in Java, automatisiert. Variablen werden dynamisch typisiert, das bedeutet, dass bei der Deklaration keine Typen angegeben werden müssen. Wie in C++ kann man auch in Ruby Operatoren überladen bzw. überschreiben. Ruby bietet eine sehr mächtige Block-Syntax, die in Kombination mit Iteratoren Schleifen oft vorgezogen wird. Ein weiteres sehr interessantes Konzept ist das sogenannte Duck-Typing, welches vorsieht Objekte anhand ihrer Funktionalität zu behandeln. Ganz kurz angesprochen werden soll zum Schluss noch das Metaprogramming.

Zu Beginn dieser Ausarbeitung wird ein kurzer Überblick über die Geschichte und Einflüsse Rubys sowie die weitere Entwicklung bis heute gegeben. Darauffolgend werden die Grundlagen der Syntax dargelegt, wobei die Besonderheiten im Mittelpunkt stehen. Im Anschluss daran widmet sich der Text einigen Konzepten die Ruby unterstützt, wie zum Beispiel den Modulen, der Vererbung oder dem angesprochenen Duck-Typing. Abschließend folgt ein kurzes, persönliches Fazit.

[1] Flanagan, David/Matsumoto, Yukihiro: The Ruby Programming Language. 1. Auflage. O'Reilly Media Inc., 2008.
[2] Thomas, Dave/Flower, Chad/Hunt, Andy: Programming Ruby 1.9 - The Pragmatic Programmers' Guide. 3. Auflage. Pragmatic Programmers, 2009.

2 Entstehung und Entwicklung

2.1 Geschichte und Einflüsse

Im folgenden Kapitel wird etwas näher auf die Entstehungsgeschichte der Programmiersprache Ruby eingegangen. Dabei wird zunächst die noch recht kurze Geschichte der Sprache und anschließend die Entwicklung bis heute im Mittelpunkt stehen.
Alles begann im Jahr 1993 als Yukihiro Matsumoto, in der Ruby-Community besser bekannt als "Matz"[3], anfing eine Sprache nach seinen Vorstellungen zu entwickeln. Sein Ziel dabei war es eine Skriptsprache zu erschaffen, die mächtiger sein sollte als Perl und objektorientierter als Python.[4] Nach zwei Jahren Entwicklungszeit wurde Ruby erstmals 1995 veröffentlicht.[5] Heraus kam eine interpretierte und rein objektorientierte Programmiersprache,[6] deren Syntax einfach und deren Grammatik für Java- und C-Programmierer leicht zu erlernen sei.[7] Außerhalb Japans erlangte Ruby allerdings erst um das Jahr 2000 größere Bekanntheit, da 1999 die erste englischsprachige Mailing-Liste und 2000 das erste englischsprachige Buch zu Ruby erschien.
Ruby kann durchaus als eine Art Multi-Paradigmen-Sprache bezeichnet werden, da neben der Objektorientierung ebenfalls rein prozudurales und funktionales Programmieren unterstützt wird.[8] Matsumoto hat sich bei der Entwicklung größtenteils von den Programmiersprachen Lisp, Smalltalk und Perl beeinflussen lassen und versucht funktionales und imperatives Programmieren in Einklang zu bringen.[9]

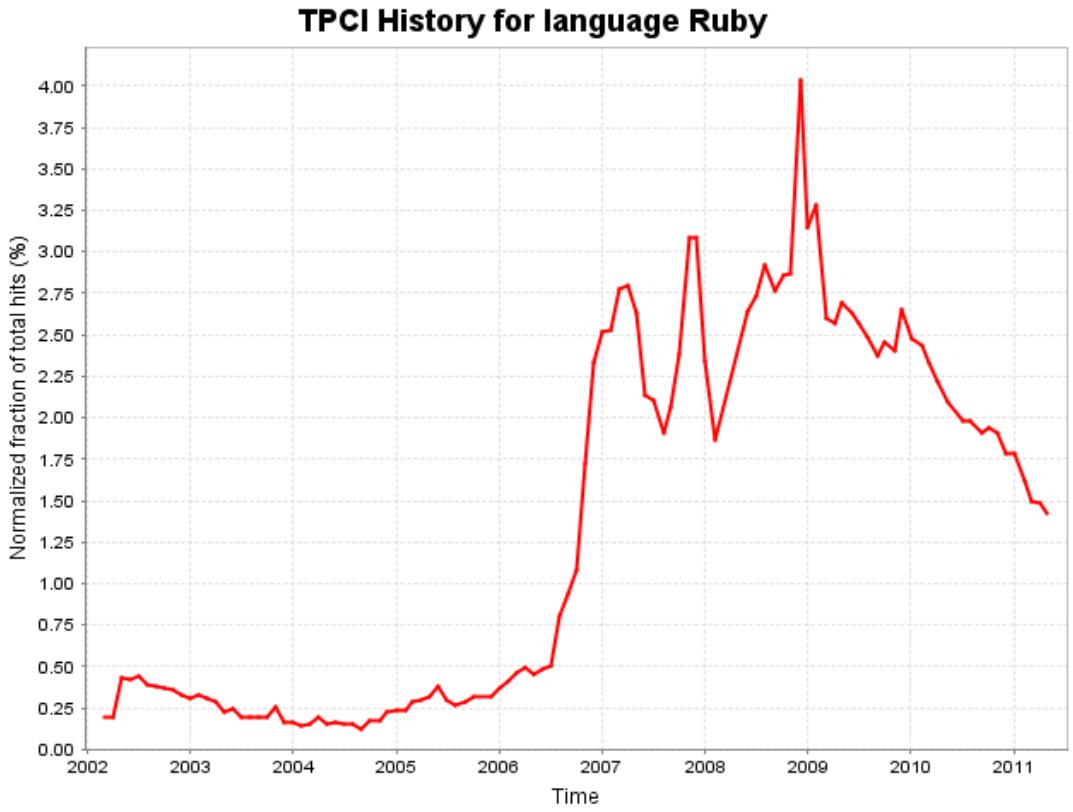
2.2 Entwicklung

Die Entwicklung von Ruby soll hier am Beispiel des TIOBE-Index verdeutlicht werden. Der TIOBE Programming Community Index (TPCI) ist ein Indikator für die Popularität einer Programmiersprache. Er wird monatlich ermittelt und basiert auf der Anzahl der Suchergebnisse des Strings +"<language> programming" bei folgenden Suchmaschinen (in Klammern steht der jeweilige Anteil am Gesamtergebnis): Google (32%), Youtube (10%), Yahoo! (3%), Bing (3%), Wikipedia (16%), Blogger (32%) und Baidu (3%).[10]
Im Mai 2011 steht Ruby mit einem Anteil von ca. 1.4% auf Position 10. Verglichen mit Mai 2010, als Ruby auf Position 11 lag, eine leichte Verbesserung. Mit den Plätzen 1 (Java mit ca. 18%)

[3] vgl. Flanagan/Matsumoto (2008), S.2.
[4] vgl. Groh, Jens: Ruby. In Taschenbuch Programmiersprachen. 2. Auflage. Carl Hanser Verlag, 2007, S.486.
[5] vgl. Ruby-Homepage. ⟨URL: `http://www.ruby-lang.org/en/about/`⟩ – Zugriff am 15.07.2011.
[6] vgl. Groh (2007), S.486.
[7] vlg. Flanagan/Matsumoto (2008), S.2.
[8] vgl. a. a. O.
[9] vgl. Ruby-Homepage.
[10] vgl. TIOBE-Index-Definition. ⟨URL: `http://www.tiobe.com/index.php/content/paperinfo/tpci/tpci_definition.htm`⟩ – Zugriff am 15.07.2011.

und 2 (C mit ca. 16%) kann Ruby zwar nicht mithalten, erreicht aber dennoch eine beachtliche Platzierung, bedenkt man das noch recht geringe Alter der Sprache.
Im Jahr 2006 wurde Ruby mit dem Award "Programming Language of the Year" ausgezeichnet, eine jährliche Auszeichnung, die die Programmiersprache erhält, die innerhalb eines Jahres den größten Anstieg der Suchmaschinentreffer verzeichnen kann.[11]
In der nachstehenden Grafik ist dieser Anstieg sehr schön zu sehen. Des Weiteren lässt sich beobachten, dass der niedrigste Wert im August 2004 (Position 27 mit 0.124%) und der höchste Wert im Dezember 2008 (Position 8 mit 4.034%) ermittelt wurde. Zudem kann man einen stetigen Rückgang der Treffer seit dem Höchstwert 2008 erkennen.

Abbildung 2.1: TIOBE-Index



Quelle: www.tiobe.com/index.php/paperinfo/tpci/Ruby.html, 2011.

[11] vgl. TIOBE-Index. ⟨URL: `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`⟩ – Zugriff am 15.07.2011.

3 Syntax

Das folgende Kapitel beschäftigt sich mit der Ruby-Syntax. Aus Java bekannte Elemente werden zwar angesprochen, jedoch wird nur kurz auf sie eingegangen. Dennoch ist dieses Kapitel wichtig um die späteren Beispiele und Konzepte besser zu verstehen.

3.1 Kommentare

Begonnen werden soll in diesem Kapitel mit einem kurzen Abschnitt über Kommentare. Kommentare beginnen in Ruby immer mit einem `#`-Zeichen und setzen sich bis zum Zeilenende fort. Der Interpreter ignoriert das `#`-Zeichen und jeglichen Folgetext, jedoch nicht den 'newline'-Character.[12] Dieser dient in Ruby zudem als Anweisungsendmarkierung (zu vergleichen mit dem Semikolon "`;`" in Java, das in Ruby auch genutzt werden kann, aber meist weggelassen wird).[13] Falls das `#`-Zeichen hingegen innerhalb eines Strings oder eines regulären Ausdrucks steht, so leitet es keinen Kommentar ein, sondern ist einfach nur ein String, bzw. ein regulärer Ausdruck.

```
# Die komplette Zeile ist auskommentiert
x = "#Dies ist ein String"
y = /#Dies ist ein regulärer Ausdruck/

#
# Multiline-Kommentare werden üblicherweise
# mit voranstehedem '#' in jeder Zeile
# realisiert
#
```

Kommentare über mehrere Zeilen werden einfach mit einem `#` vor jeder Zeile realisiert. Es ist noch wichtig zu erwähnen, dass Ruby keine Inline-Kommentare (wie zum Beispiel `/* ... */` in Java oder C) unterstützt.[14]

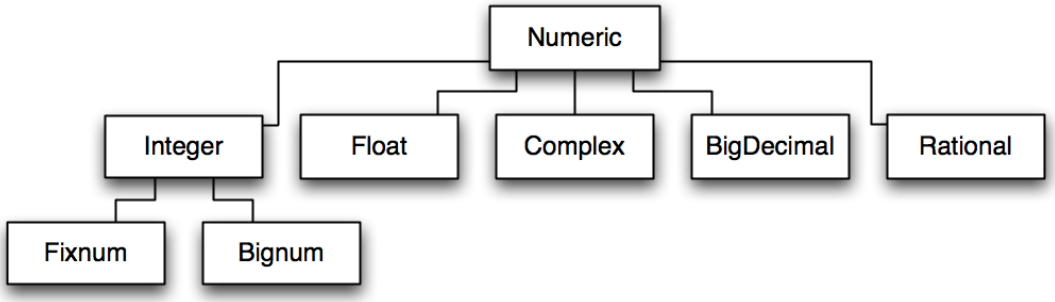
3.2 Basisdatentypen

Als nächstes werden die Basisdatentypen in Ruby behandelt. Dazu gehören `Integer`, wobei hier zwischen `Fixnum` und `Bignum` unterschieden wird, die beide Unterklassen von `Integer` sind, und `Float`. `Fixnum`-Objekte können ganze Zahlen bis zu einer Größe von 31 Bit darstellen. Sollte der Wertebereich darüber hinausgehen, so wird das `Fixnum` automatisch in ein `Bignum`-Objekt

[12] vgl. Flanagan/Matsumoto (2008), S.26.
[13] vgl. a. a. O., S.32.
[14] vgl. a. a. O., S.26.

konvertiert. Die Begrenzung eines `Bignum`-Objektes liegt hierbei effektiv bei der Größe des zur Verfügung stehenden Speichers.[15]
Eine Zahl mit einem Dezimalpunkt und/oder einem Exponenten wird als `Float`-Objekt interpretiert. Dabei ist es wichtig zu beachten, dass mindestens eine Ziffer vor dem Dezimalpunkt steht und eine danach. Ist dies nicht der Fall (z.B. `1.e4`), so versucht Ruby die Methode `e4` an dem `Fixnum 1` aufzurufen.[16]
Zusätzlich existieren noch drei weitere Klassen zur Repräsentation von Zahlen: `Complex`, `BigDecimal` und `Rational`. Die Klassennamen sprechen in diesen Fällen für sich und geben gleichzeitig ihre jeweilige Funktion an. Übrigens sind in Ruby alle numerischen Objekte *immutable*.[17]

Abbildung 3.1: `Numeric` Klassenhierarchie



Um Zeichenketten darzustellen gibt es auch in Ruby `String`s. Anders als in Java sind `String`-Objekte in Ruby *mutable* und die `String`-Klasse bietet jede Menge Methoden um auf ihren Instanzen zu operieren. Strings können in einfache oder doppelte Anführungszeichen gesetzt werden, wobei Strings in doppelten Anführungszeichen wesentlich flexibler sind und einige Backslash-Escape-Sequenzen (wie z.B. `\n` für Newline oder `\t` für Tab) unterstützen. Strings in einfachen Anführungszeichen substituieren lediglich die Sequenz `\\` in einen einzigen Backslash und `\'` in ein Apostroph. Alle weiteren Escape-Sequenzen erscheinen unausgewertet in dem jeweiligen String.[18][19]

```
42         # ist ein Integer (Fixnum)
42.even?   # liefert TRUE
3.14       # ist vom Typ Float
"Hallo"    # ein String mit Inhalt Hallo
:symbol    # ein Symbol ist ein "konstanter" String
:"symbol"  # ebenfalls ein Symbol
```

An diesem kleinen Beispiel lässt sich gut erkennen, das wirklich alles ein Objekt ist, selbst die Zahl `42`, an der die Methode `even?` aufgerufen werden kann. Dabei geschieht ein Methodenaufruf, genau wie in Java, über die bekannte Punktsyntax.
Als letzter Datentyp sollen die `Symbol`e vorgestellt werden. Sie beginnen stets mit einem Doppelpunkt gefolgt von einem Identifier oder einem String. Dabei können Anführungszeichen hilfreich

[15] vgl. Thomas/Flower/Hunt (2009), S.315.
[16] vgl. a. a. O., S.316.
[17] vgl. Flanagan/Matsumoto (2008), S.42-43.
[18] vgl. a. a. O., S.46-47.
[19] vgl. Thomas/Flower/Hunt (2009), S.316-317.

sein, falls ein String durch Leerzeichen getrennt ist. Symbole können als "konstante" Strings verstanden werden (sie sind *immutable*). Zwei normale Strings können zum Beispiel den gleichen Inhalt haben und dennoch zwei völlig verschiedene Objekte sein. Dies ist bei Symbolen nie der Fall. Zwei String-Symbole mit dem gleichen Inhalt werden beide zum selben `Symbol`-Objekt konvertiert. Folglich besitzen zwei verschiedene `Symbol`-Objekte immer einen unterschiedlichen Inhalt. In Ruby werden Symbole häufig bei `Hash`-Objekten verwendet, die im nachfolgenden Abschnitt erläutert werden. Zudem hält der Ruby-Interpreter eine `Symbol`-Tabelle vor, in der alle Klassennamen, Methoden und Variablen abgelegt werden, um die relativ teuren `String`-Vergleiche durch günstigere `Integer`-Vergleiche zu ersetzen.[20]

```
>> :symbol.object_id    # Symbol ohne Anführungszeichen
=> 327228
>> :"symbol".object_id  # Symbol mit Anführungszeichen, das gleiche wie oben
=> 327228
>> "symbol".object_id   # ein einfacher String
=> 2156589680
>> :symbol.object_id    # immernoch die selbe object_id
=> 327228
>> "symbol".object_id   # eine neuer einfacher String mit anderer object_id
=> 2156574540
```

3.3 Arrays und Hashes

Objekte der `Array`-Klasse werden erzeugt, indem man Objektreferenzen durch Kommata getrennt zwischen eckige Klammern schreibt. Ein nachstehendes Komma wird vom Interpreter ignoriert. Auf die jeweiligen Werte kann direkt über ihre Position, bzw. ihren Index (mit dem `[]`-Operator) zugegriffen werden. Die Methoden `size` und `length` liefern die Anzahl der Elemente des Arrays zurück. Des Weiteren sind Arrays dynamisch vergrößerbar. Weist man einen Wert einem Index jenseits der aktuellen Arraylänge zu, so wird das Array automatisch erweitert und mit `nil`-Elementen aufgefüllt.[21]

```
[]             # erzeugt ein neues Array
a = [3, 5, 7]  # erzeugt ein gefülltes Array
b = [3, 5, 7,] # s.o. (a == b => TRUE)
a[0]           # liefert den 1. Eintrag von a
```

Ein Objekt der Ruby `Hash`-Klasse wird durch das Setzen von Schlüssel/Wert-Paaren zwischen zwei geschwungene Klammern erzeugt und ist mit der Java `HashMap` zu vergleichen. Einem Schlüssel wird dabei durch einen Pfeil (`=>`) ein Wert zugewiesen.[22]

```
{}                                        # erzeugt einen leeren Hash
h = {:key => "value", :key2 => "value2"} # erzeugt einen gefüllten Hash
h[:key]                                   # liefert den Wert für :key zurück
```

[20] vgl. Flanagan/Matsumoto (2008), S.70-71.
[21] vgl. Thomas/Flower/Hunt (2009), S.318.
[22] vgl. a. a. O., S.318-319.

3.4 Operatoren

Abbildung 3.2: Operatoren in Ruby

Table 22.4. Ruby Operators (High to Low Precedence)

Method	Operator	Description
✓	[ ] [ ]=	Element reference, element set
✓	**	Exponentiation
✓	! ~ + −	Not, complement, unary plus and minus (method names for the last two are +@ and -@)
✓	* / %	Multiply, divide, and modulo
✓	+ −	Plus and minus
✓	>> <<	Right and left shift (<< is also used as the append operator)
✓	&	"And" (bitwise for integers)
✓	^ |	Exclusive "or" and regular "or" (bitwise for integers)
✓	<= < > >=	Comparison operators
✓	<=> == === != =~ !~	Equality and pattern match operators
	&&	Logical "and"
	||	Logical "or"
	.. ...	Range (inclusive and exclusive)
	? :	Ternary if-then-else
	= %= /= −= += |= &= >>= <<= *= &&= ||= **=	Assignment
	not	Logical negation
	or and	Logical composition
	if unless while until	Expression modifiers
	begin/end	Block expression

Quelle: Thomas/Flower/Hunt (2009), S.333.

Die oben stehende Tabelle ist eine Auflistung aller Operatoren in Ruby. In der Spalte "Method" sind die Operatoren gekennzeichnet, die als Methoden implementiert wurden und somit überschrieben werden können (ähnlich zu C++). Aufgrund der großen Überschneidungen zu Java soll im Folgenden nur auf einige wenige Operatoren näher eingegangen werden.
Im Gegensatz zu Java existiert in Ruby explizit der `**`-Operator zum Potenzieren. `2 ** 10` bedeutet also 2^{10}. Ebenfalls interessant ist der `===`-Operator, auch Case-Equality-Operator genannt, da er implizit bei `case`-Anweisungen genutzt wird. Er prüft auf Zugehörigkeit (Membership) oder auf Muster (Pattern-Matching).[23]
Einen direkten Inkrementierungs- bzw. Dekrementierungs-Operator (wie z.B. `++` oder `--` in Java) gibt es in Ruby nicht.

[23] vgl. Flanagan/Matsumoto (2008), S.106.

3.4.1 Zuweisungen

Wie aus der Tabelle ersichtlich ist, lassen sich die Zuweisungsopreatoren nicht überladen bzw. überschreiben. Eine Besonderheit, die hier allerdings herausgestellt werden soll ist die parallele Zuweisung mehrerer Werte.
Besitzt eine Zuweisung mehr als einen Wert auf der linken Seite (lWerte), so gibt der Zuweisungsausdruck ein Array der Werte auf der rechten Seite (rWerte) zurück. Überwiegt die Anzahl der lWerte gegenüber den rWerten, dann werden die überschüssigen lWerte auf `nil` gesetzt. Sind im Gegensatz mehr rWerte vorhanden als lWerte, so werden die überschüssigen rWerte ignoriert. Werden mehrere rWerte einem einzigen lWert zugewiesen, so werden die rWerte seit Ruby 1.6.2 automatisch in ein Array umgewandelt.
Eine weitere Funktionalität bietet der *-Operator. Mit diesem lassen sich Arrays auf- und wieder zuklappen. Steht zum Beispiel vor dem letzten lWert ein Sternchen, so werden alle restlichen rWerte gesammelt und diesem lWert als ein Array zugewiesen. Ist der letzte rWert ein Array, dann kann diesem ein Sternchen vorangestellt werden, wordurch es in seine Elemente zerlegt wird und es möglich ist, diese einzeln zuzuweisen. Dies ist nicht nötig, wenn auf der rechten Seite nur ein Array steht. In diesem Fall erfolgt das Aufklappen automatisch.[24]

```
x, y = y, x       # Werte vertauschen
a = [1, 2, 3, 4] # <=> a = 1, 2, 3, 4
b, c  = a         # => b == 1,  c == 2
b, *c = a         # => b == 1,  c == [2, 3, 4]
b, c  = 42, a     # => b == 42, c == [1, 2, 3, 4]
b, *c = 42, a     # => b == 42, c == [[1, 2, 3, 4]]
b, c  = 42, *a    # => b == 42, c == 1
b, *c = 42, *a    # => b == 42, c == [1, 2, 3, 4]
```

Eine weitere Eigenschaft der parallelen Zuweisung ist die Verschachtelung. Dabei kann die linke Seite einer Zuweisung eine in Klammern gesetzte Liste von Termen enthalten. Diese werden von Ruby so behandelt, als seien es geschachtelte Zuweisungsanweisungen. Der entsprechende rWert wird extrahiert und den in Klammern stehenden Termen zugewiesen, bevor mit der Zuweisung auf höherer Ebene fortgefahren wird.[25]

```
b, (c, d), e = 1,2,3,4      # =>  b == 1,  c == 2,  d == nil,    e == 3
b, (c, d), e = [1,2,3,4]    # =>  b == 1,  c == 2,  d == nil,    e == 3
b, (c, d), e = 1,[2,3],4    # =>  b == 1,  c == 2,  d == 3,      e == 4
b, (c, d), e = 1,[2,3,4],5  # =>  b == 1,  c == 2,  d == 3,      e == 5
b, (c,*d), e = 1,[2,3,4],5  # =>  b == 1,  c == 2,  d == [3, 4], e == 5
```

3.4.2 == vs. equal?

Für den Java-Programmierer zunächst etwas gewöhnungsbedürftig dürfte die Verwendung des ==-Operators sein. Dieser prüft in Ruby nämlich auf inhaltliche und nicht auf Referenzengleich-

[24] vgl. Flanagan/Matsumoto (2008), S.97-100.
[25] vgl. Thomas/Flower/Hunt (2009), S.134-135.

heit. Dafür gibt es in Ruby die `equal?`-Methode, die die jeweilige `object_id` zweier Objekte miteinander vergleicht.[26]

```
1 + 2 == 3            # => true, denn == testet inhaltliche Gleichheit
"a" == "a"            # => true
"a".equal? "a"        # => false, denn equal? testet Identität (object_id)
```

3.4.3 Range

Ein `Range`-Objekt repräsentiert die Werte zwischen einem Start- und einem Endwert. Erzeugt werden sie durch das Schreiben von zwei bzw. drei Punkten zwischen dem Start- und dem Endwert (also `..` bzw. `...`). Zwei Punkte bedeuten, dass der Endwert noch in der Range enthalten ist (inklusive), bei drei Punkten ist der Endwert nicht mehr Teil der Range (exklusive).[27] Deutlich wird dies im unteren Beispiel, in dem nun auch der `===`-Operator zum Einsatz kommt.

```
1..10           # inklusive 10
1...10          # exklusive 10
'a'..'z'        # inklusive z
'a'...'z'       # exklusive z
(1..10) === 5   # => true
(1..10) === 10  # => true
(1...10) === 10 # => false
```

3.5 Reguläre Ausdrücke

Wie auch in anderen Sprachen gibt es in Ruby Reguläre Ausdrücke (oder auch `Regexp`). Sie bieten eine komfortable Möglichkeit Pattern-Matching auf Strings durchzuführen. In der Praxis werden reguläre Ausdrücke meist dazu verwedet, um zu testen, ob ein String einem bestimmten Muster entspricht, Teile aus einem String zu extrahieren, die einem Muster entsprechen oder Teile eines Strings zu ersetzen.[28] Reguläre Ausdrücke werden durch zwei Slashes begrenzt. Zum Prüfen, ob ein String einem bestimmten Muster entspricht, wird der `=~`-Operator benutzt.[29]
Da die Möglichkeiten mit Regulären Ausdrücken zu arbeiten recht umfangreich sind, sollen hier nur einige wenige Beispiele dargestellt werden, um zu zeigen dass Ruby diese Funktionalität bietet. Wer sich intensiver mit den `Regexp` beschäftigen möchte, wird in der angegebenen Literatur fündig.
Zur Veranschaulichung seien hier die eckigen Klammern genannt, die ihre beinhaltenden Character mit einem logischen "oder" verknüpfen. Es gibt zusätzlich vorgefertigte Character-Klassen wie z.B. `/\d/`, die testet, ob ein String eine Ziffer enthält. Möchte man wissen, ob ein String mit `y` oder `Y` beginnt, so testet man mit `/^[yY]/`.[30]

[26] vgl. Flanagan/Matsumoto (2008), S.76.
[27] vgl. a. a. O., S.68-70.
[28] vgl. Thomas/Flower/Hunt (2009), S.99.
[29] vgl. a. a. O., S.100.
[30] vgl. Flanagan/Matsumoto (2008), S.310-321.

```ruby
/[Rr]uby/ =~ "Ruby"       # => true
/[Rr]uby/ =~ "ruby"       # => true
/\d{5}/ =~ asf34152809kl # => false, da keine 5 aufeinanderfolgenden Ziffern
/\d{5}/ =~ asf12345kl     # => true
/^[yY]/                   # true, falls der String mit y oder Y beginnt
```

3.6 Methoden und Variablen

Nachdem die Grundlagen der Ruby-Syntax behandelt wurden, geht es nun weiter zu den nächst komplexeren Gebilden, den Methoden.
Methoden werden in Ruby durch das Schlüsselwort `def` eingeleitet, gefolgt von dem Methodennamen und den formalen Parametern in Klammern (Klammern können weggelassen werden, aber nach Konvention nur, wenn keine Parameter an die Methode übergeben werden). Durch das Schlüsselwort `end` wird das Ende der Methode angezeigt.[31] Laut Konvention beginnen Methodennamen immer mit einem Kleinbuchstaben, da sie sonst mit Konstanten verwechselt werden können, und folgen dem `snake_case` anstatt dem `camelCase`-Schema.[32] Methodennamen können, müssen aber nicht, mit einem Fragezeichen, einem Ausrufungszeichen oder einem Gleichheitszeichen enden. In jedem dieser Fälle ist für den Aufrufer der Methode eine bestimmte Funktionalität zu erkennen. Methoden, die mit einem `?` enden (z.B. `even?` oder `equal?`) liefern `true` oder `false` zurück. Methoden, die auf ein `!` enden, wobei zu diesen Methoden meist ein Pendant mit dem selben Namen ohne Ausrufungszeichen existiert, sind zum Beispiel `sort` und `sort!`. Die `sort`-Methode liefert ein modifiziertes Objekt zurück und arbeitet auf einer Kopie, die `sort!`-Methode hingegen operiert auf dem Originalobjekt. Methoden, die auf ein `=`-Zeichen enden fungieren als Setter-Methoden und können mit der Zuweisungssyntax aufgerufen werden.[33]
Ein Methodenaufruf an einem Objekt geschieht mit Hilfe der (auch aus Java) bekannten Punktsyntax. Die Parameterübergabe folgt dem Prinzip “Call by Value”.[34]
Der Rückgabewert einer Methode muss nicht explizit im Methodenkopf angegeben werden. Terminiert die Methode normal, so wird der letzte ausgewertete Ausdruck innerhalb des Rumpfes implizit zurückgeliefert (im unteren Beispiel würde also der `String ‘‘überall sichtbar’’` zurückgegeben). Es besteht weiterhin die Möglichkeit das Schlüsselwort `return` zu verwenden, welches allerdings meistens nur genutzt wird, um einen Wert vorzeitig an den Aufrufenden zurück zu liefern.[35]

```ruby
def method
  lokale_variable1 = 42  #dynamisch typisiert
  lokale_variable2 = "Hallo Welt!"
  KONSTANTE = 3.14
```

[31] vgl. Flanagan/Matsumoto (2008), S.177.
[32] vgl. a. a. O., S.180.
[33] vgl. Thomas/Flower/Hunt (2009), S.119-120.
[34] vgl. a. a. O., S.122.
[35] vgl. Flanagan/Matsumoto (2008), S.177-178.

```
  @instanz_variable = "in der Instanz sichtbar"
  @@klassen_variable = "vgl. static in Java"
  $globale_variable = "überall sichtbar"
end
```

Neben der Methodensyntax soll ebenfalls auf die vier unterschiedlichen Arten von Variablen eingegangen werden. Zunächst fällt auf, dass in Ruby auf Typdeklarationen vor Variablen verzichtet werden kann, da diese dynamisch typisiert werden.[36]
Lokale Variablen beginnen mit einem Unterstrich oder einem Kleinbuchstaben und sind nur innerhalb des aktuellen Blocks oder der aktuellen Methode sichtbar.[37] Variablen, die mit `@` oder `@@` beginnen sind Instanz-, bzw. Klassenvariablen.[38] Instanzvariablen werden so gekapselt, dass man nur über die Instanzmethoden des jeweiligen Objekts auf sie zugreifen kann.[39] Klassenvariablen sind in der gesamten Klasse sichtbar und werden von den Klassen-, sowie Instanzmethoden geteilt. Sie sind zu vergleichen mit `static`-Variablen aus Java. Klassenvariablen werden ebenfalls gekapselt und sind für den Benutzer der Klasse nicht sichtbar.[40] Globale Variablen beginnen mit einem `$`-Zeichen und sind überall sichtbar.[41]
Weiterhin existieren in Ruby zusätzlich Konstanten. Diese beginnen mit einem Großbuchstaben und werden meist durchgängig groß geschrieben. Wird einer Konstante das erste Mal ein Wert zugewiesen, so sollte dieser konstant bleiben. In Ruby lassen sich die Werte von Konstanten im Programmverlauf allerdings ändern. Dieses Vorgehen ruft keinen Fehler hervor, sondern generiert lediglich eine Warnmeldung durch den Interpreter. Aus diesem Grund werden die Konstanten unter der Rubrik Variablen mit aufgeführt.[42]
Zuletzt seien noch die Methodensichtbarkeiten angesprochen, die mit sogenannten Modifiern gesteuert werden können. Die drei unterschiedlichen Sichtbarkeitstypen werden mit `public`, `private` und `protected` betitelt. Methoden sind von Natur aus `public`, außer sie werden explizit als `private` oder `protected` deklariert. Eine Ausnahme bildet hier die `initialize`-Methode, die stets implizit `private` ist (s. Abschnitt "Klassen"). Ist eine Methode `protected`, so kann diese von jeder Instanz der definierenden oder einer erbenden Klasse aufgerufen werden. Falls eine Methode als `private` deklariert ist, so kann sie nur innerhalb des aufrufenden Objekts benutzt werden. Es ist nicht möglich direkt auf private Instanzmethoden anderer Objekte zuzugreifen, selbst, wenn es ein Objekt der selben Klasse ist. Dennoch sind private Methoden für vererbte Klassen sichtbar. Folglich ist es in Ruby nicht möglich Methoden komplett zu verstecken. Für den Aufruf einer privaten Methode `method` bedeutet dies konkret, dass man sie wie eine Funktion ohne konkreten Empfänger aufrufen muss, also nur `method`. Man kann weder `object.method`, noch `self.method` schreiben.[43]

[36] vgl. Thomas/Flower/Hunt (2009), S.324.
[37] vgl. Flanagan/Matsumoto (2008), S.87.
[38] vgl. a. a. O.
[39] vgl. a. a. O., S.216.
[40] vgl. a. a. O., S.230.
[41] vgl. a. a. O., S.87.
[42] vgl. Thomas/Flower/Hunt (2009), S.324.
[43] vgl. Flanagan/Matsumoto (2008), S.232-234.

3.7 Klammern

Eine weitere Besonderheit von Ruby sind die optionalen Klammern bei Methodenaufrufen und konditionalen Ausdrücken (im Gegensatz zu Java). Die Regel lautet hier, dass die Klammern in einfachen Fällen und bei Eindeutigkeit weggelassen werden können.[44] Das Ergebnis ist sauberer und klarer Code. In komplexeren Fällen und bei Verwirrungsgefahr sollte man Klammern um jeden Preis nutzen, da es sonst schnell zu syntaktischer Unklarheit kommen kann.[45]
In dem unteren Beispiel wird jeweils die Variante mit und ohne Klammern aufgeführt. Bei Methoden, denen ein Hash übergeben wird, können zusätzlich zu den runden, sogar die geschwungenen Klammern weggelassen werden.

```
[3, 7].include? 3  # => ture
[3, 7].include?(3) # das gleiche wie oben
methode :key => "value"      # Methode mit einem Hash als Übergabeparameter
methode({:key => "value"})  # das gleiche wie oben
```

3.8 Kontrollstrukturen

Wie in jeder Sprache gibt es auch in Ruby gewisse Kontrollstrukturen, um den Programmfluss zu verzweigen, Codeteile nur unter bestimmten Voraussetzungen auszuführen oder zu wiederholen. Vorweg ist es noch wichtig zu erwähnen, dass in Ruby jeder Wert wie `true` behandelt wird, außer `false` und `nil`. Dies sollte man bei der Verwendung von Kontrollstrukturen im Hinterkopf behalten.[46]

3.8.1 Bedingte Anweisungen und Verzweigung

Die wohl geläufigste bedingte Anweisung ist die `if`-Anweisung. Diese funktioniert in Ruby ähnlich wie in Java.[47]

```
if boolean_expression [then]
  body
[elsif boolean_expression [then]
  body , ...]
[else
  body ]
end
```

Die offensichtlichen Unterschiede sind zum einen die fehlenden Klammern um die Bedingung sowie der Abschluss des Statements mit `end` anstatt geschweiften Klammern. Möchte man eine Mehrfachverzweigung realisieren, so nutzt man `elsif` um weitere Bedingungen zu prüfen und `else` für alle anderen Fälle. Mit einem `then` ist es zudem möglich die Anweisung in die selbe

[44] vgl. Thomas/Flower/Hunt (2009), S.124.
[45] vgl. Flanagan/Matsumoto (2008), S.183.
[46] vgl. a. a. O., S.11.
[47] vgl. Thomas/Flower/Hunt (2009), S.337.

Zeile zu schreiben in der auch die Bedingung steht.[48]
Die `unless`-Anweisung bildet das Gegenteil zu `if`, da der `body` nur ausgeführt wird, wenn der `boolean_expression` als `false` oder `nil` ausgewertet wird.[49] Die Funktionsweise wird im Folgenden gezeigt.[50]

```
unless boolean_expression [then]
  body
[else
  body ]
end
```

Anstatt nun einzeilige Bedingungen mit `then` und `end` schreiben zu müssen, können `if` und `unless` auch als Modifier genutzt werden.
Dabei wird der `expression` nur ausgewertet, falls `boolean_expression true` (für `if`) oder `false` (für `unless`) ist.[51]

```
expression if     boolean_expression
expression unless boolean_expression
```

Des Weiteren verfügt Ruby über eine `case`-Anweisung, die üblicherweise wie folgt eingesetzt wird:

```
case target
when comparison [, comparison]... [then]
  body
when comparison [, comparison]... [then]
  body
...
[else
  body ]
end
```

Hier kommt nun der zuvor schon angesprochene Case-Equality-Operator `===` implizit beim Vergleichen zum Einsatz. Es wird eine Übereinstimmung des `target` mit einem der `when`-Fälle geprüft (`comparison === target`) und entsprechend abgearbeitet.[52]

3.8.2 Schleifen

Neben den Bedingten Anweisungen existieren in Ruby die Schleifen. Dazu gehören die `while`- und die `until`-Schleife. Diese führen den Code in ihrem Rumpf aus, solange (`while`) eine bestimmte Bedingung `true` ist oder bis (`until`) diese Bedingung `true` wird. Die Syntax der beiden Schleifen ergibt sich wie folgt:[53]

[48] vgl. Flanagan/Matsumoto (2008), S.118-122.
[49] vgl. a. a. O., S.122-123.
[50] vgl. Thomas/Flower/Hunt (2009), S.337.
[51] vgl. a. a. O.
[52] vgl. a. a. O., S.338.
[53] vgl. a. a. O.

```
while boolean_expression [do]
  body
end

until boolean_expression [do]
  body
end
```

Ebenso wie `if` und `unless` können auch `while` und `until` als Modifier genutzt werden:

```
expression while boolean_expression
expression until boolean_expression
```

3.8.3 Blöcke und Iteratoren

Blöcke und Iteratoren nehmen eine Sonderstellung ein und beschreiben eine von Rubys besonderen Stärken. Es ist möglich Code-Blöcke an Methoden zu übergeben, fast so als wären es Parameter. Dabei bestehen Code-Blöcke lediglich aus Code, der zwischen zwei geschweiften Klammern oder einem `do/end`-Konstrukt steht. Hierbei muss die öffnende geschwungene Klammer bzw. das `do` immer in der selben Zeile stehen wie der Methodenaufruf. Die Konvention schreibt vor, die geschwungenen Klammern bei einzeiligen und das `do/end` bei mehrzeiligen Blöcken zu benutzen.[54]
Grundsätzlich können Blöcke nicht allein stehen und treten nur in Verbindung mit einem Methodenaufruf auf. Prinzipiell kann ein Block nach jeder beliebigen Methode stehen. Ist diese allerdings weder ein Iterator noch ruft sie den Block mit `yield` auf, so wird der Block nicht weiter beachtet.[55] Blöcke können Parameter enthalten, die, anders als bei Methoden, in vertikalen Strichen (z.B. `|x|`) stehen müssen. So wird bei der `each`-Methode im unteren Beispiel für jedes Element aus der `Range (1..10)` die Quadratzahl ausgegeben. Dabei wird intern eine Schleife generiert und (in unserem Fall) das `i` durch die jeweilige Zahl ersetzt. Der Code-Block wird also für jedes Element des aufrufenden Objektes einmal ausgeführt.[56]

```
10.times {print "Hallo! "}     # gibt 10-mal "Hallo! " aus

1.upto(10) {|j| puts j}        # gibt die Zahlen von 1-10 aus

(1..10).each {|i|  puts i*i}   # gibt die Quadratzahlen von 1-10 aus

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10].each do |x|
  if x.even?
    print x                    # gibt alle geraden Zahlen des Arrays aus
  end
end
```

[54] vgl. Thomas/Flower/Hunt (2009), S.24.
[55] vgl. Flanagan/Matsumoto (2008), S.141.
[56] vgl. Thomas/Flower/Hunt (2009), S.56.

Die Methoden `times`, `upto` und `each` (und einige andere) sind Iteratoren und anhand ihres Names und der obigen Beispiele recht selbsterklärend. Diese Methoden können mit einem Code-Block als Parameter aufgerufen werden. Die Variante der Blöcke wird üblicherweise der normalen `while`-Schleife vorgezogen.[57]

3.9 Klassen

Der letzte Teil des Syntax-Kapitels befasst sich mit den Klassen, die in der objektorientierten Programmierung eine entscheidene Rolle spielen. In Ruby werden Klassen mit dem Schlüsselwort `class` eingeleitet und enden mit dem Schlüsselwort `end`. Der Name einer Klasse muss immer mit einem Großbuchstaben beginnen.[58] Klassen stellen eine Sammlung von Methoden dar, die auf dem Zustand eines Objektes operieren können. Dieser Objektzustand wird durch die Instanzvariablen repräsentiert.
Anhand der folgenden Beispielklasse `Person` sollen die wesentlichen Bestandteile und Funktionalitäten von Klassen und bestimmten Methoden verdeutlicht werden.

```
class Person
  attr_accessor :name, :age, :footsize     #fügt getter und setter ein

  # Konstruktor
  def initialize(name = "Hans", age = 42, footsize = 45)
    @name, @age, @footsize = name, age, footsize
  end

  # to_String-Methode geerbt von Object
  def to_s
    "Name: #{@name}\nAlter: #{@age}\nSchuhgroesse: #{@footsize}"
  end

end
```

Um Instanzen einer Klasse erstellen zu können, wird ein Konstruktor benötigt. Dieser heißt in Ruby immer `initialize` und wird mit `Person.new` aufgerufen.[59] Es besteht die Möglichkeit den Variablen, die dem Konstruktor übergeben werden, direkt Default-Werte zuzuweisen. Auffällig ist, dass die Instanzvariablen unmittelbar initialisiert werden können ohne vorher deklariert zu werden (man beachte die Parallenzuweisung). In der ersten Zeile der Klasse Person wird die Methode `attr_accessor` aufgerufen, die sehr praktisch ist, da sie automatisch Getter- und Setter-Methoden für alle nachstehenden Instanzvariablen erstellt. Die Getter-Methoden haben dann die gleichen Namen wie die zugehörigen Instanzvariablen (in diesem Fall werden also die drei Methoden `name`, `age` und `footsize` erstellt). Die Setter-Methoden heißen ebenfalls so wie die entsprechenden Instanzvariablen mit dem Zusatz "`=`" am Ende des Namens (hier folglich `name=`,

[57] vgl. Flanagan/Matsumoto (2008), S.3-4 und S.130.
[58] vgl. a. a. O., S.215.
[59] vgl. a. a. O., S.215-216.

`age=` und `footsize=`). Der Aufruf der Setter-Methoden kann zum einen mit `name=` und zum anderen mit `name =` (mit Leerzeichen) geschehen. Diese Hilfsmethode erspart einem somit viel Schreibarbeit und Zeit. Möchte man lediglich lesenden Zugriff auf die Instanzvariablen gewähren, so benutzt man `attr_reader`, um nur Getter erstellen zu lassen.[60]
Wie auch in Java verfügt jedes Objekt über eine geerbte `toString()`-Methode, die in Ruby einfach nur `to_s` heißt. Diese wird in dem Beispiel überschrieben. Interessant daran ist das Einfügen von Variablen in einen String. Dies geschieht mit Hilfe von `#{expression}`. Dabei wird die Variable, oder der Ausdruck in den geschweiften Klammern ausgewertet und zu einem String substituiert.[61]

[60] vgl. Flanagan/Matsumoto (2008), S.218-219.
[61] vgl. Thomas/Flower/Hunt (2009), S.90-91.

4 Konzepte

Nachdem im vorherigen Kapitel das Thema Grundlagen abgeschlossen wurde, widmet sich dieses Kapitel nun einigen Konzepten, die Ruby unterstützt. Dazu gehören die Module, Vererbung, Duck-Typing und das Metaprogramming, wobei letzteres nur sehr kurz thematisiert wird.

4.1 Module

Module können ebenso wie Klassen Methoden, Konstanten und Klassenvariablen beinhalten. Definiert werden sie ebenfalls wie Klassen, allerdings mit den Schlüsselwort `module` anstatt `class`. Zudem können Module nicht instanziiert werden und man kann nicht von ihnen erben. Das bedeutet, dass Module für sich allein stehen und es keine Modulhierarchie oder ähnliches gibt.
So wie ein `class`-Objekt eine Instanz der Klasse `Class` ist, so ist ein `module`-Objekt eine Instanz der Klasse `Module`. Hierbei ist `Class` eine Unterklasse von `Module`. Das bedeutet, dass alle Klassen Module sind, aber nicht alle Module Klassen. Module mit Modulmethoden bilden Namespaces, ebenso wie Klassen mit Klassenmethoden. Module mit Instanzmethoden werden als Mixins benutzt, wobei eine Klasse niemals als ein solches fungieren kann.[62]
Im weiteren Verlauf soll etwas näher auf Module als Namespaces bzw. Mixins eingegangen werden.

4.1.1 Module als Namespace

Module bieten eine gute Möglichkeit verwandte Methoden zu gruppieren, wenn objektorientiertes Programmieren nicht notwendig ist. Als ein Ruby-internes Beispiel sei das `Math`-Module genannt, welches als Namespace fungiert und viele mathematische Funktionen bereitstellt (z.B. `Math.sqrt(9)` liefert `3`).[63]

```
module Mein_Modul
  def self.modulmethode
    puts "Eine Modulmethode!"
  end

  def Mein_Modul.zweite_modulmethode
    puts "Noch eine!"
  end
end
```

[62] vgl. Flanagan/Matsumoto (2008), S.247-248.
[63] vgl. a. a. O., S.248.

```
Mein_Modul.modulmethode #=> Eine Modulmethode!
```

Das obere Beispiel zeigt einen selbst definierten Namespace. Wichtig ist hierbei, dass nur Modulmethoden bzw. Klassenmethoden implementiert werden dürfen. Erwirkt wird dies durch den Modul- bzw. Klassennamen `Mein_Modul` oder das Schlüsselwort `self` gefolgt von einem Punkt und dem Methodennamen.[64] Der Aufruf einer Modul- oder Klassenmethode erfolgt einfach durch Nennung des Modul- bzw. Klassennamens, einem Punkt und der entsprechenden Methode.

4.1.2 Module als Mixins

Die alternative Verwendung von Modulen sind die sogenannte Mixins, die mächtiger sind als die Namespaces. Sie sind zu vergleichen mit den Interfaces in Java. Definiert ein Modul Instanzmethoden anstatt Klassenmethoden, so können diese Instanzmethoden in andere Klassen hineingemixt werden (daher der Name). Sehr bekannt sind zum Beispiel das `Enumerable` oder das `Comparable` Mixin. Anders als bei den Java-`Interfaces` können die Instanzmethoden der Mixins Code enthalten und diesen anderen Klassen zur Verfügung stellen. Dies wird durch die `include`-Methode realisiert, wobei eine beliebige Anzahl von Mixins angegeben werden kann.[65]

```
module Mein_Modul
  def meine_methode
    puts "Hello!"
  end
end

class Meine_Klasse
  include Mein_Modul            #<== Einbinden des Moduls
end

Meine_Klasse.new.meine_methode #=> Hello!
```

Auch hier soll ein kleines Beispiel zur besseren Veranschaulichung dienen. `Mein_Modul` besitzt eine Instanzmethode `meine_methode`. `Meine_Klasse` bindet `Mein_Modul` als Mixin ein und der Aufruf der Methode `meine_methode` von einem neuen `Meine_Klasse`-Objekt liefert den Output `Hello!`.

4.2 Vererbung

Das Thema Vererbung wurde bereits angesprochen und wird in diesem Abschnitt etwas weiter vertieft. Das Prinzip der Vererbung ist aus den meisten objektorientierten Sprachen bereits bekannt, weshalb nur auf die wesentlichen Dinge eingegangen werden soll.
Alle Klassen in Ruby erben implizit von der Basisklasse `Object` (seit Ruby 1.9 von `BasicObject`).

[64] vgl. Flanagan/Matsumoto (2008), S.248-249.
[65] vgl. a. a. O., S.250-251.

Somit werden jeder Klasse bereits einige Funktionalitäten zur Verfügung gestellt (z.B. die `to_s`-Methode).
Um eine Klasse von einer anderen erben zu lassen schreibt man ein < in der Klassendefinition zwischen den Namen der erbenden und der vererbenden Klasse.
In Ruby gibt es keine direkte Mehrfachvererbung. Dennoch kann diese mit Hilfe von Mixins simuliert werden (warum das so ist wird im Kapitel Duck-Typing deutlich).[66] Geerbte Methoden können einfach überschrieben werden, z.B. um ihre Funktionalität anzupassen oder zu erweitern. Manchmal kann es sein, dass man eine Methode nicht gänzlich überschreiben möchte, sondern lediglich etwas Code hinzufügen will. Mit dem Schlüsselwort `super` lässt sich die überschriebene Methode von der überschreibenden Methode aufrufen. Dieses Prinzip nennt sich Chaining. `super` kann genutzt werden um Methoden oder Konstruktoren der Oberklasse aufzurufen. Grundsätzlich wird in Ruby alles an die Subklasse vererbt, also Instanzmethoden, private Methoden, Klassenmethoden, Instanzvariablen, Klassenvariablen und Konstanten.[67]

```
class Student < Person        # erbt von Person
  attr_accessor :matrikel

  def initialize(name, age, footsize, matrikel)
    super(name, age, footsize)
    @matrikel = matrikel
  end

  def to_s                    #überschreibt to_s
    "#{super.to_s}\nMatrikel: #{@matrikel}"
  end
end
```

Die Beispielklasse `Person` wird nun durch die Klasse `Student` spezialisiert, indem `Student` von `Person` erbt. Es kommt die Instanzvariable `matrikel` neu hinzu. Um nun nicht den gesamten Konstruktor neu schreiben zu müssen, wird mit `super` der Konstrukter von `Person` aufgerufen und lediglich `@matrikel` in dieser Klasse gesetzt. Die `to_s`-Methode wird ebenfalls überschrieben, allerdings so, das sie lediglich erweitert wird, weshalb erneut mit `super` die `to_s`-Methode von `Person` aufgerufen wird.

4.3 Duck-Typing

Der Begriff Duck-Typing leitet sich von einem Gedicht von James Whitcomb Riley ab: *"When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."*
Duck-Typing bietet ein Gegenkonzept zu typisierten Sprachen. Es ist weniger eine Menge von Regeln, sondern vielmehr ein Programmierstil. Duck-Typing sagt aus, dass ein Objekt in Abhängigkeit

[66] vgl. Thomas/Flower/Hunt (2009), S.71-77.
[67] vgl. Flanagan/Matsumoto (2008), S.234-241.

seiner Funktionalität (implementierte Methoden) behandelt wird und nicht in Abhängigkeit seines Typs. Das bedeutet, dass die Klassenzugehörigkeit eines Objektes egal ist, solange es die Methoden zur Verfügung stellt, die benötigt werden. Folglich ist nicht der Typ, sondern die bereitgestellte Funktionalität entscheidend. Nun wird nochmals deutlicher, weshalb mit Mixins eine Mehrfachvererbung sehr gut simuliert werden kann, denn letztlich ist es egal von welchem Typ ein Objekt ist, solange es die Methoden implementiert (evtl. auch durch Mixins), die gebraucht werden.
Das Prinzip des Duck-Typings spaltet die Programmierer in zwei Lager: Befürworter und Gegner. Manche mögen diese Art der Flexibilität und fühlen sich wohl dynamisch typisierten Code zu schreiben und andere (besonders C++ oder Java-Programmierer) können sich nicht vorstellen mit Ruby ernsthafte Applikationen zu schreiben.[68] Verdeutlicht werden soll das Konzept Duck-Typing an einem einfachen Beispiel.

```
class Schaf
  def sag_was
    "Maeh Maeh..."
  end
  def wer_bist_du
    "Ein Schaf"
  end
end

class WolfImSchafspelz
  def sag_was
    "Maeh Maeh grrr..."
  end
  def wer_bist_du
    "Ein Schaf"
  end
end

def schafstest( schaf )                  # Parameter kann ein beliebiges
  puts "--"                              # Objekt sein, hauptsache
  puts schaf.sag_was                     # die Methode sag_was und
  puts schaf.wer_bist_du                 # wer_bist_du existieren
  if schaf.wer_bist_du =~ /Ein Schaf/
    puts "Das ist wirklich ein Schaf"
  end
end

schaf1 = Schaf.new
schaf2 = WolfImSchafspelz.new
```

[68] vgl. Thomas/Flower/Hunt (2009), S.359-372.

```
schafstest( schaf1 )                    # läuft durch
schafstest( schaf2 )                    # ebenfalls kein Fehler
```

Zunächst erstellen wir zwei Klassen: `Schaf` und `WolfImSchafspelz`. Beide besitzen die gleichen Methoden: `sag_was` und `wer_bist_du`. Nun gibt es eine `schafstest`-Methode, die prüfen soll, ob das übergebene Objekt wirklich ein Schaf ist (allerdings nur mit einem String-Vergleich und nicht mit `instance_of?`). Aufgrund der dynamischen Typisierung von Variablen ist es der Methode zunächst egal, was für einen Objekttyp sie übergeben bekommt, solange dieses die Methoden `sag_was` und `wer_bist_du` zur Verfügung stellt. Da nun auch der `WolfImSchafspelz` in seiner Methode `wer_bist_du` den String `Ein Schaf` zurückliefert, ist die Ausgabe bei beiden Tests `Das ist wirklich ein Schaf`.
Für Java-Programmierer ist diese Möglichkeit zunächst sehr ungewöhnlich und vermittelt den Eindruck von Uneindeutigkeit, Verwirrung und evtl. Unsicherheit, doch in der Praxis scheint sich dieses Prinzip bewährt zu haben.

4.4 Metaprogramming

Was ist Metaprogramming? Metaprogramming bedeutet, wie der Name schon suggerieren lässt, auf einer höheren Abstraktionsebene zu arbeiten. Man ist nicht mehr durch die Abstraktionen der Programmiersprache eingeschränkt, sondern erschafft neue Abstraktionen, die in die Ursprungssprache integriert werden. Das heißt, man schreibt Code, der dabei hilft neuen Code zu schreiben oder zu generieren. Des Weiteren erlaubt Metaprogramming das Ändern von Code bzw. Sprachkonstrukten (wie z.B. Klassen, Modulen, Instanzvariablen) zur Laufzeit. Es ist sogar möglich, Code zur Laufzeit zu schreiben und auszuführen, ohne das Programm neu starten zu müssen.[69]
So interessant dieses Konzept auch ist, so umfangreich ist es. Da dies allerdings nur eine Einführung in die Programmiersprache Ruby ist und es den Rahmen dieser Arbeit sprengen würde, soll das Thema Metaprogramming an dieser Stelle nicht weiter vertieft werden. Interessenten können sich allerdings in der weiterführenden Literatur einlesen.

[69] vgl. Thomas/Flower/Hunt (2009), S.373-408.

5 Fazit

Man kann sich nun die Frage stellen, wofür Ruby überhaupt benötigt wird. In der Tat ist es so, dass eher selten Applikationen allein mit Ruby erstellt werden. Viel häufiger ist es der Fall, dass Ruby in Verbindung mit dem Framework Rails zum Einsatz kommt und damit ein sehr mächtiges Werkzeug für die Entwicklung von Web- oder Datenbankapplikationen zur Verfügung stellt, den Hauptanwendungsgebieten von Ruby.
Trotzdem ist die Sprache nicht zu unterschätzen. Neben dem in dieser Arbeit Gezeigten können in Ruby auch C- oder Java-Bibliotheken eingebunden oder GUI-Programme erstellt werden. Ruby bietet wirklich eine große Bandbreite an Möglichkeiten. Die Sprache ist, besonders im Empfinden von Java- oder C-Programmierern, sehr dynamisch, bietet aber gleichzeitig einen leichten Einstieg aufgrund der schnell zu erlernenden Syntax und Grammatik. Da das POLS (Principle of least surprise), also das Prinzip der geringsten Überraschung unterstützt wird[70], ist Ruby in den meisten Fällen sehr intuitiv in der Anwendung.
Die zur Verfügung stehende, aktuelle Literatur ist leicht verständlich und gut geschrieben. Es existiert eine große und aktive Community, was zum Vorteil hat, dass man bei der Suche im Internet schnell auf viele Tutorials und Beispiele trifft. Weitere Pluspunkte sind die Plattformunabhängigkeit, die große API sowie die gute und umfangreiche Dokumentation.[71] Abschließend lässt sich sagen, dass es Spaß gemacht hat, sich in diese Programmiersprache einzuarbeiten, die unterhaltsame Literatur zu lesen und Ruby selbst zu schreiben und auszuprobieren.

[70] vgl. Groh (2007), S.486.
[71] Ruby-Dokumentation. ⟨URL: `http://ruby-doc.org/`⟩ – Zugriff am 15.07.2011.

Literaturverzeichnis

Ruby-Dokumentation. ⟨URL: `http://ruby-doc.org/`⟩ – Zugriff am 15.07.2011

Ruby-Homepage. ⟨URL: `http://www.ruby-lang.org/en/about/`⟩ – Zugriff am 15.07.2011

TIOBE-Index. ⟨URL: `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`⟩ – Zugriff am 15.07.2011

TIOBE-Index-Definition. ⟨URL: `http://www.tiobe.com/index.php/content/paperinfo/tpci/tpci_definition.htm`⟩ – Zugriff am 15.07.2011

Flanagan, David/Matsumoto, Yukihiro: The Ruby Programming Language. 1. Auflage. O'Reilly Media Inc., 2008

Groh, Jens: Ruby. In Taschenbuch Programmiersprachen. 2. Auflage. Carl Hanser Verlag, 2007, S.486–507

Thomas, Dave/Flower, Chad/Hunt, Andy: Programming Ruby 1.9 - The Pragmatic Programmers' Guide. 3. Auflage. Pragmatic Programmers, 2009