MAGISTERARBEIT

Java versus Ruby: Anwendbarkeit des objektorientierten Paradigmas

Ausgeführt am

Institut für Computersprachen,
Arbeitsgruppe für Programmiersprachen und Übersetzerbau
der Technischen Universität Wien

unter der Anleitung von

Ao.Univ.Prof.Dipl.-Ing.Dr.techn. Franz Puntigam

durch

Peter Arpad Wesel Bakk.techn.
Matr.Nr. 9227527
Vivenotgasse 3/6/2/6
1120 Wien

Wien, Dezember 2003

Kurzfassung

Von den diversen Paradigmen der Programmierung ist das objektorientierte eines der populärsten. Es ist daher natürlich, dass es verschiedenste Ansätze für die Umsetzung des objektorientierten Paradigmas gibt. Diese Arbeit untersucht die Anwendbarkeit von objektorientierten Konzepten in den Programmiersprachen Java und Ruby. Die betrachteten Konzepte Einfachvererbung, Ersetzbarkeitsprinzip, Mehrfachvererbung, Delegation, Informationsverheimlichung, Abstrakte Klassen, Objektspezialisierung, Finalisierung, Überladen, Überschreiben und Design by Contract sind für die Erstellung qualitativ hochwertiger Programme von großer Bedeutung. Ausgestattet mit den allgemeinen Sprachkonzepten werden die genannten objektorientierten Konzepte bestmöglichst in Java und Ruby eingesetzt. Dort wo die Sprache keine direkte Unterstützung für ein Konzept anbietet, wird mit alternativen Möglichkeiten eine adäquate Lösung realisiert. Es zeigt sich, dass objektorientierte Sprachkonzepte nicht unbedingt in der Sprache enthalten sein müssen, da in einigen Fällen mit alternativen Methoden eine ebenso gute Anwendbarkeit erzielt werden kann.

Abstract

From the various programming paradigms the object-oriented paradigm is one of the most popular. It is natural therefore that there are diverse approaches for the support of object-oriented concepts. This thesis examines the applicability of object-oriented concepts in the programming languages Java and Ruby. The considered concepts such as single inheritance, subtyping, multiple inheritance, delegation, information hiding, abstract classes, object specialization, final class or object, overloading, overriding, and Design by Contract are of great importance for the production of high-quality programs. Equipped with the general language concepts we use the object-oriented concepts in the best possible way in Java and Ruby. Where the language does not offer direct support for a concept, we realize an adequate solution with alternative possibilities. We will show that an object-oriented language concept does not have to be contained necessarily in the language, since in some cases just as good applicability can be obtained with alternative methods.

Inhaltsverzeichnis

Kapitel 1

Einleitung

In dieser Arbeit wird die Anwendbarkeit der beiden objektorientierten Programmiersprachen Java und Ruby beleuchtet.

Mit der Programmiersprache Java bzw. mit ihren Technologien (J2SE, J2EE, J2ME) verbindet man oft Plattformunabhängigkeit und vielseitige Einsatzmöglichkeiten im Internet (Applets, JavaServer Pages, Servlets, etc.). Diese Unabhängigkeit der Applikationen von der Hardware und dem Betriebssystem erspart bei Portierungen auf leistungsfähigere Plattformen hohe Aufwände an Kosten und Zeit.

Die Skriptsprache Ruby ist in ihrem Ursprungsland Japan auf Grund ihrer Natürlichkeit[1] und Lesbarkeit in relativ kurzer Zeit auf sehr viel Interesse gestossen. Ruby kann für einfache Verarbeitungen von Texten, als Werkzeug für Serveradministratoren, als Bindeglied zwischen heterogenen Applikationen aber auch für größere Applikationen eingesetzt werden.

Die Programmiersprache Java wurde für den Vergleich gewählt, weil sie eine vielseits bekannte objektorientierte Programmiersprache ist, die auch in komplexen Anwendungen erfolgreich eingesetzt wird und ein statisches Typsystem hat.
Ruby mit seinem dynamischen Typsystem ist dagegen eine sehr junge und weniger bekannte objektorientierte Skriptsprache, mit der Prototypen recht schnell entwickelt werden können. Die Antwort auf die Frage, ob Java oder Ruby das objektorientierte Paradigma besser unterstützt, wird in dieser Arbeit diskutiert.

Wie jedes Produkt unterliegen auch Programmiersprachen gewissen Qualitätsanforderungen. Unmittelbar nach der Einleitung werden in Kapitel 2 die notwendigen Qualitätsanforderungen an Programmiersprachen definiert.

In Kapitel 3 und 4 werden essentielle Spracheigenschaften für die Anwendbarkeit objektorientierter Konzepte in Java und Ruby beschrieben. Sowohl die Syntax als auch die Programmbeispiele dieser Arbeit beziehen sich auf die derzeit gültige Java Version 1.4.2 bzw. auf die stabile Ruby Version 1.8.0.

[1] Natürlichkeit im Sinne des Prinzips der geringsten Überraschungen – die Dinge funktionieren so, wie man es von ihnen erwartet.

Um qualitativ gute objektorientierte Programme zu erstellen, sind die folgenden Konzepte von grosser Bedeutung:

- Einfache Vererbung
- Ersetzbarkeitsprinzip
- Mehrfache Vererbung
- Delegation
- Informationsverheimlichung
- Abstrakte Klassen
- Objektspezialisierung
- Finalisierung
- Überladen
- Überschreiben
- Design by Contract

Das Paradigma der Generizität – auch „parametrischer Polymorphismus" genannt – wurde in diese Arbeit nicht aufgenommen. Generizität ist unabhängig vom objektorientierten Paradigma und kann auch in rein prozeduralen oder funktionalen Sprachen[2] vorhanden sein. Dazu kommt noch der Aspekt, dass die demnächst verfügbare Java Version 1.5 [Bracha, 2001] Typgenerizität unterstützen wird, die der Java Erweiterung GJ [Generic Java] sehr ähnlich ist, aber eben nicht vollständig entspricht. Ein Vergleich der Generizität zwischen Java und Ruby wäre daher inadäquat.

Aber noch wichtiger ist die Antwort auf die Frage, in welcher Qualität diese Konzepte dem Entwickler zur Verfügung gestellt werden. Da nicht alle Sprachkonzepte in Java oder Ruby direkt unterstützt sind, muss mit den zur Verfügung stehenden Spracheigenschaften versucht werden, anwendbare Lösungen zu finden.

In Kapitel 5, das den Kern dieser Arbeit darstellt, werden die genannten Konzepte vorgestellt und versucht, sie in Java und Ruby optimal umzusetzen. Die dabei entstandenen Probleme und gewonnenen Erkenntnisse werden detailliert beschrieben und am Ende jedes Abschnitts in einem Fazit zusammengefasst.

Im letzten Unterabschnitt werden die untersuchten Konzepte nochmals reflektiert und in Form einer Tabelle dargestellt.

Eine abschließende Zusammenfassung rundet diese Arbeit ab.

[2]Haskell [Thompson, 1999] ist ein Beispiel für eine funktionale Sprache, in der die Generizität (generic functions) umgesetzt wurde.

Kapitel 2

Qualitätsanforderungen

Sehr stark gekoppelt sind die Begriffe Anwendbarkeit, Verständlichkeit und Qualität. Die Anwendbarkeit von Programmiertechniken leidet, sobald die Qualität der Programmiersprache nicht gewährleistet ist. Die Qualitätsanforderungen an eine Programmiersprache werden in [Ghezzi et al., 1998] mit den Anforderungen an die Qualität einer Software gleichgesetzt. Dies begründet sich mit der Tatsache, dass eine Programmiersprache ein Werkzeug für die Entwicklung von Software darstellt. Folgende Qualitätsanforderungen sind daher sicherzustellen:

- *Software muss verlässlich sein.* Der Anwender einer Software sollte sich auf diese verlassen können. Zum Beispiel sollte die Wahrscheinlichkeit eines Ausfalls durch einen Fehler gering sein. So weit wie möglich sollte das System ein „fault-tolerant" Verhalten aufzeigen, damit Benutzer bei unerwünschten Ereignissen wie Hardware- oder Softwarefehlern von der Anwendung zufriedenstellend unterstützt werden. Dies ist umso wichtiger, wenn es sich um komplexe oder kritische Software handelt.

- *Software muss wartbar sein.* Es ist ökonomisch nicht vertretbar, bestehende Software bei Änderungswünschen zu verwerfen und durch eine gänzliche Neuentwicklung zu ersetzen. Ferner sollte darauf geachtet werden, dass bestehende Software leicht auf neue Anforderungen angepasst werden kann. Natürlich ist es nicht einfach, zu Beginn eines komplexen Softwareprojektes alle tatsächlichen Anforderungen zu kennen. Hier kann prinzipiell nur gehofft werden, dass sich das Projekt in die gewünschte Richtung entfaltet.

- *Software muss effizient durchführbar sein.* Effizienz ist das Ziel eines jeden Softwaresystems. Auch wenn Programme auf modernen Architekturen laufen, ist es wichtig, die Kosten der Hardware (schnelle Prozessoren, Hauptspeicher) durch sparsames Ausnutzen von Ressourcen niedrig zu halten, da in einem konkurrierenden System Programme mit geringerem Ressourcenbedarf eine höhere Ausführungswahrscheinlichkeit erhalten und somit schneller abgearbeitet sind als ineffiziente Programme.

2.1 Verlässlichkeitsanforderungen

Eine unvollständige Auflistung von Verlässlichkeitsanforderungen betreffend Qualität für Programmiersprachen, die sich einerseits überlappen und andererseits teilweise ausschließen wird nun genauer betrachtet:

- *Schreibbarkeit:* Darunter ist primär zu verstehen, dass die Ausdrucksweise in einem Programm in der für das Problem natürlichen Art und Weise beschritten wird. Der Programmierer sollte also durch die Programmiersprache nicht mit trickreichen und ablenkenden Details von der wichtigen Problemlösung abgelenkt werden. Subjektiv betrachtet sind die Programmiersprachen höherer Generationen[1] schreibbarer als die der 1. bzw. 2. Generation. Ein Assemblerprogrammierer ist neben der Lösung des Problems sehr intensiv mit der Adressierung von Daten, Registerinhalten etc. konfrontiert. Je einfacher es ist, sich mit der Problemlösung auseinanderzusetzen, desto weniger fehleranfällig ist das Schreiben von Programmen.

- *Lesbarkeit:* Ermöglicht der Logik eines Programms zu folgen und Fehler beim Suchen zu entdecken. Auch diese Eigenschaft ist ein subjektives Kriterium, das vom Geschmack und Stil des Einzelnen abhängt. Die Lesbarkeit hängt wiederum von der verwendeten Programmiersprache (Generation) ab.

- *Einfachheit:* Eine einfache Sprache ermöglicht Algorithmen einfacher zu implementieren, wodurch wiederum das Vertrauen auf Korrektheit erhöht wird. Einfachheit kann auch darauf abzielen, dass die Fähigkeiten einer Programmiersprache minimiert werden, wodurch die Leistung dieser Sprache wiederum reduziert wird. Zum Beispiel ist Pascal einfacher aber nicht so leistungsstark wie C++.

- *Sicherheit:* Die Sprache sollte keine Möglichkeiten anbieten, um schlechte Programme entwickeln zu können. Wenn eine Sprache keine „goto“ Anweisungen oder Zeigervariablen anbietet, werden damit Quellen gefährlicher Fehler vermieden. Diese beiden Sprachmöglichkeiten können spitzfindige Fehler verursachen, die bei der Programmentwicklung nur sehr schwer erkennbar sind und auch nach der Programmauslieferung noch böse Überraschungen liefern. Auch hier gilt wieder die Aussage, dass Einschränkungen dieser Möglichkeiten die Leistungen und Effizienz einer Programmiersprache verringern können.

- *Robustheit:* Eine Sprache unterstützt Robustheit, wenn sie die Fähigkeit besitzt, mit unerwarteten Ereignissen, wie zum Beispiel arithmetischem Überlauf, ungültigen Eingabedaten etc., fertig zu werden. Wenn Ereignisse dieser Art abgefangen und darauf dementsprechend reagiert wird, ist das Verhalten solcher Systeme in unerwarteten Situationen voraussagbarer.

2.2 Wartbarkeit

Es ist offensichtlich wichtig, dass Programmiersprachen die einfache Veränderbarkeit, Lesbarkeit und Einfachheit in Programmen ermöglichen sollen. Zwei wichtige Eigenschaften, die die Wartbarkeit unterstützen, sind Faktorisierung und Lokalität:

- *Faktorisierung:* Programmiersprachen sollen dem Programmierer helfen, zusammengehörige Charakteristiken zu einer Einheit zusammenzufassen. Ein triviales Beispiel: Wenn eine Operation in mehreren Punkten auftritt, so sollte es möglich sein, diese zu einer Routine zu faktorisieren und dann diese Operation durch Funktionsaufrufe zu ersetzen. Dadurch wird das Programm lesbarer (sofern sprechende Namen für Routinen verwendet werden) und leichter wartbar – Änderungen sind nur an einer

[1] 1. Generation: Maschinensprache, 2. Generation: Assembler, 3. Generation: u.a. Java, Ruby

Stelle durchzuführen. Ein anderes Beispiel für Faktorisierung wäre, dass mehrere Programmiersprachen es erlauben, Konstanten mit symbolischen Namen zu versehen. Auch hier wird die Lesbarkeit durch adäquate Namen erhöht, z.B.: PI für 3.14. Noch mehr Bedeutung gewinnt diese Eigenschaft, wenn später dieser Wert zum Beispiel auf 3.14159 geändert wird. Statt diesen Wert an mehreren Stellen zu ändern, muss durch diese Eigenschaft der Wert nur an einer Stelle modifiziert werden.

- *Lokalität:* Ein Effekt einer Spracheigenschaft sollte sich nur auf einen kleinen, lokal begrenzten Teil im Programm beziehen. Andernfalls, wenn sich ein Effekt auf weite Teile im Programm auswirkt, werden Veränderungen außerordentlich komplex. Zum Beispiel ist bei Verwendung abstrakter Datentypen garantiert, dass Modifikationen der Datenstruktur innerhalb der Klasse keine Auswirkungen auf den restlichen Teil des Programms haben, sofern die Operationen, welche die Datenstruktur verändern, nach wie vor auf dieselbe Art und Weise aufgerufen werden.

2.3 Effizienz

Die Notwendigkeit der Effizienz in einer Programmiersprache gibt es seit den frühen Tagen der elektronischen Datenverarbeitung. Viele Sprachen haben die Effizienz entweder explizit oder implizit als primäres Ziel in ihrem Entwurf. Die Bedeutung der Effizienz hat sich im Laufe der Zeit maßgeblich geändert. Unter Effizienz wird jetzt nicht mehr nur die Geschwindigkeit und der Speicherplatzbedarf zur Laufzeit verstanden, sondern auch der Aufwand, um ein Programm oder System zu erstellen. Auch der Aufwand für die Wartung kann als Faktor für die Messung von Effizienz betrachtet werden. Anders ausgedrückt kann gesagt werden, dass auf die Produktivität während des Entwicklungsprozesses mehr Gewicht gelegt wird, als auf das resultierende Produkt. Oft wird auf die *Wiederverwendbarkeit* von Komponenten in zukünftigen Systemen sehr viel Wert gelegt, in anderen Bereichen wird wiederum verstärkt Aufmerksamkeit auf die *Portabilität* für andere Plattformen und den damit größeren Anwenderkreis gelegt. Dies mit dem bewussten Nachteil, dass eine ad hoc optimierte Version für die jeweilige Plattform eine bessere Performance aufweisen würde.

Effizienz ist häufig ein Qualitätsmerkmal in der Kombination von Programmiersprache und Implementation. Die Programmiersprache beeinträchtigt Effizienz, wenn sie bestimmte Optimierungen dem Compiler verwehrt. Die Implementierung wiederum beeinträchtigt Effizienz, wenn sie nicht alle Möglichkeiten ausschöpft, um Speicherplatz und Durchführungsgeschwindigkeit zu verbessern. Folgendes Beispiel:

```
x = fun(y) + z + fun(y);
```

Diese Anweisung kann in der Programmiersprache C nicht zu

```
x = 2 * fun(y) + z;
```

optimiert werden, obwohl diese Variante auf Grund des einmaligen Aufrufs der Funktion *fun* vermutlich effizienter wäre. Die Spracheigenschaft von C, welche die Veränderung von globalen Variablen zulässt (wie in diesem Beispiel z), verbietet diese Optimierung.

Ein anderes Beispiel wäre eine Sprache, die eine Effizienz durch Gleichzeitigkeit konkurrierender Berechnungen erlaubt. Eine Implementierung, die den freigegebenen Speicher in einem Programm nicht wiederverwendet, würde diese Effizienz zunichte machen. Weiters würde eine Programmiersprache, welche die Sicht bzw. den Zugriff auf die darunterliegende Speichergröße ermöglicht (zum Beispiel die Art und Weise wie Gleitkommazahlen gespeichert werden), den Vorteil der Portabilität der Software auf andere Maschinen sofort einschränken.

Kapitel 3

Die Programmiersprache Java

3.1 Was ist Java?

Im Jahr 1995 führte Sun Microsystems die Java Technologie, welche von Beginn an für das Internet und Firmen-Intranet ausgelegt war, ein. In [Gosling, 1995] wird die Programmiersprache Java mit folgenden Eigenschaften beschrieben:

> „A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high perfomance, multithreaded, dynamic language.“

Die Syntax in Java entspricht weitgehend der von C [Kernighan et al., 1988] und C++ [Stroustrup, 1997]. Um die Sprache jedoch einfach und übersichtlich zu lassen, wurden Konzepte wie Zeigermodell, Header-Dateien, Mehrfachvererbung und Templates nicht übernommen. Die Erweiterungen für das dynamische Auflösen von Methoden stammen teilweise von Objective-C[1]. Für die „nebenläufigen Prozesse“ (Multithreading) wurde auf ein weit verbreitetes, jedoch bereits veraltetes, Monitor- und Zustandsvariablenparadigma von C.A.R.Hoare [Hoare, 1974] zurückgegriffen.

In der Diskussion um Java sind die Begriffe *Virtual Machine* und *Plattform* zu unterscheiden [Flanagan, 2002]:

- Die Java Virtual Machine (JVM), auch Java Interpreter genannt, ist das ausschlaggebende Stück einer Java Installation. Vom Design her ist jedes Java-Programm portierbar, jedoch nur auf solche Plattformen, auf die auch der Java Interpreter portiert wurde. Sun stellt JVM Implementationen für ihr eigenes Solaris Betriebssystem, Microsoft Windows und Intel-Linux zur Verfügung. Viele weitere Hersteller wie Apple und andere Unix-Hersteller stellen für ihre Plattformen einen Java Interpreter bereit. Weiters gibt es auch für Hand-Held Geräte (PalmOS, PocketPC) und eingebettete Systeme (embedded systems) entsprechende Versionen.
  Die JVM Technologie ermöglicht auch eine just-in-time (JIT) Compilierung, in der der Java Byte Code fließend in einen plattformabhängigen Maschinencode übersetzt wird. Codeteile, die wiederholt ausgeführt werden, gewinnen dadurch an Geschwindigkeit. Eine Implementierung dieser JIT Compilierung stellt Sun mit der HotSpot Technologie bereit.

[1] Objective-C ist eine objektorientierte Erweiterung der Programmiersprache C. Sie wurde in den frühen 80er Jahren von Brad Cox und Tom Love entwickelt.

- Für einen Programmierer ist die Plattform durch APIs (Application Programming Interface) definiert, auf die er sich beim Schreiben von Programmen bezieht. Diese APIs werden prinzipiell vom Betriebssystem des Zielrechners vorgegeben. Das bedingt, dass ein Entwickler, der ein Programm für Microsoft Windows erstellt, ein anderes API verwenden muss als ein Entwickler, der das gleiche Programm für einen Macintosh oder ein Unix-basiertes Betriebssystem schreibt. Unter diesem Aspekt sind Windows, Macintosh und Unix drei verschiedene Plattformen. Eine Applikation welche mit der Java Plattform entwickelt wurde, sollte auf jedem Betriebssystem, das die Java Plattform unterstützt, funktionieren. Die von Sun propagierte Devise „Write once, run anywhere“ stimmt in der Praxis aber nur bedingt, da es immer wieder zu Inkompatibilitäten zwischen den einzelnen Plattformen, im Speziellen zwischen ältern und neueren, kommt.

3.2 Versionen

- *Java 1.0* war die erste offizielle Version von Java. Die ursprünglich 212 Klassen waren in 8 packages organisiert. Diese Version war sehr einfach.

- *Java 1.1* hat die Plattform auf 504 Klassen in 23 packages verdoppelt. Es wurden signifikante Geschwindigkeitsverbesserungen in der JVM erwirkt und die *inner classes* eingeführt. Prinzipiell ist diese Version ebenfalls veraltet, jedoch gibt es noch sehr viele Installationen alter Web-Browser, die die Java Version 1.1 interpretieren und somit für einfache Applets noch eine Bedeutung haben.

- *Java 1.2* war eine wichtige Freigabe von Java. Die Größe stieg auf das dreifache – 1520 Klassen in 59 packages – an. Neu hinzugekommen ist das Swing GUI API sowie Collection APIs für den Umgang mit sets, lists und maps Objekten. Aufgrund der vielen Erweiterungen in der Version 1.2 wird die Plattform nun als „Java 2 Plattform“ bezeichnet. Die vielen Änderungen bewirkten auch, dass vieles inkompatibel zu den vorherigen Versionen wurde.

- *Java 1.3* war eigentlich nur eine Wartungsversion, die sich primär auf Fehlerbehebung, Stabilität und Geschwindigkeitsverbesserung (inklusive der HotSpot virtual machine) fokussierte. Neue hinzugekommen zur Plattform sind JNDI (Java Naming and Directory Interface) und Java Sound API. Insgesamt beinhaltet diese Version 1.842 Klassen in 76 packages.

- *Java 1.4* ist eine weitere grosse Version, die nun 2991 Klassen in 135 packages aufweist. Neue Funktionalitäten sind ein schnelleres low-level I/O API, Unterstützung für Pattern-Matching mit Regular Expressions, ein Logging-API, ein user preference API, neue Collection Klassen, ein XML-basierender persistenter Mechanismus für Java Beans, Unterstützung für XML-Parsing (DOM-, SAX-API), Benutzerauthentifizierung mit JAAS-API (Java Authentication and Authorization Service), Unterstützung für sichere Netzwerkverbindungen mit Hilfe von SSL (Secure Socket Layer) Protokoll, Unterstützung für Kryptographie, ein API für Lesen und Schreiben von Bilddateien, ein API für Drucken im Netz, eine Handvoll neuer GUI Komponenten für das Swing API. Zusätzlich zu diesen Plattformerweiterungen wurde die assert Anweisung zur Java Sprache hinzugefügt.

Bis zur Version 1.1 benötigte man das Java Development Kit (JDK) für die entsprechende Version. Seit Java 1.2 wurde das JDK auf Software Development Kit (SDK) umbenannt, das nun „Java 2 SDK, Standard Edition“ lautet. Das Java Runtime Environment (JRE) beinhaltet alles, was für das Ausführen von Java-Programmen nötig ist. Es beinhaltet aber keine Werkzeuge für die Entwicklung von Java-Programmen wie zum Beispiel den Compiler. Weiters gibt es noch das Java Plug-in, das eine Version von JRE für die Web-Browser (Netscape, Microsoft Internet Explorer) darstellt.
Zusätzlich zur Standard Edition von Java, welche von den meisten Java Programmierern verwendet wird, hat Sun noch die Java 2 Plattform Enterprise Edition (J2EE) für Entwickler von komplexeren Applikationen in Unternehmen und die Java 2 Plattform Micro Edition (J2ME) für elektronische Systeme in Endkundengeräten wie PDAs (Personal Digital Assistant) und Handys freigegeben.

C-Sharp

Quasi als Antwort auf die Java-Technologie von Sun wurde im Jahr 2000 die .NET Plattform und die Programmiersprache C# [Ecma-334, 2002] von Microsoft angekündigt. C# ist dem Konzept von Java sehr ähnlich. Statt dem Java Byte-Code gibt es den IL-Code (Intermediate Language) mit Metadaten, der in der CLR (Common Language Runtime), ähnlich der JVM, ausgeführt wird. Jedoch wird die CLR nicht interpretiert sondern aus Geschwindigkeitsgründen ausschließlich als JIT-Compiler ausgeführt. Ein komparativer Überblick von C# findet sich in [Albahari, 2000].

3.3 Sprachkonzepte

In diesem Abschnitt werden die wichtigsten Sprachkonzepte von Java für die Untersuchung der objektorientierten Konzepte im Kapitel 5, Seite 65 beschrieben [Arnold et al., 2000].

3.3.1 Kommentare

Kommentare innerhalb vom Quellcode dienen der Verständlichkeit für den Programmierer und werden vom Compiler ignoriert. Ab der Zeichenfolge `//` werden alle Zeichen bis zum Zeilenende als Kommentar betrachtet. Ein mehrzeiliger Kommentar kann mit der Zeichenfolge `/*` eingeleitet und mit der Zeichenfolge `*/` beendet werden.

3.3.2 Typen

Die Programmiersprache Java ist eine mehr oder weniger streng typisierte Sprache. Das bedeutet, dass der Compiler Typkonsistenz aller Ausdrücke garantiert und somit alle Typfehler zum Zeitpunkt der Compilierung erkannt werden [Cardelli et al., 1985]. Die Typen in der Programmiersprache Java werden in die zwei Kategorien *Primitive Datentypen* und *Referenz-Datentypen* unterteilt:

Primitive Datentypen

In Java werden acht einfache Datentypen, genannt *primitive Typen*, unterstützt. Die primitiven Datentypen umfassen: einen Boolean-Typ, einen Character-Typ, vier Integer-Typen und zwei Floating-Point-Typen. Die vier Integer-Typen und die zwei Floating-Point-Typen

Typ	**Inhalt**	**Default**	**Größe**	**Bereich**
boolean	true, false	false	1 bit	N/A
char	Unicode Character	\u0000	16 bits	\u0000 bis \uFFFF
byte	Signed Integer	0	8 bits	-128 bis 127
short	Signed Integer	0	16 bits	-32768 bis 32767
int	Signed Integer	0	32 bits	-2147483648 bis 21747483647
long	Signed Integer	0	64 bits	-9223372036854775808 bis 9223372036854775807
float	IEEE 754 Float	0.0	32 bits	±1.4E-45 bis ±3.4028235E+38
double	IEEE 754 Float	0.0	64 bits	±4.9E-324 bis ±1.7976931348623157E+308

Tabelle 3.1: Primitive Datentypen in Java

unterscheiden sich in der Anzahl von Bits, die sie repräsentieren, und somit im Wertebereich. Tabelle 3.1 fasst diese primitiven Datentypen zusammen.

Referenztypen

Zu den Referenztypen gehören die Klassen, Interfaces und Arrays. Ein Java Programm besteht aus Klassendefinitionen, wobei jede Klasse einen neuen Datentyp, der vom Programm verändert werden kann, darstellt. In einem Programm wird zum Beispiel eine Klasse mit dem Namen Point definiert, um Punkte eines kartesischen Koordinatensystems abzulegen bzw. zu verändern. Durch diese Definition wird Point ein neuer Datentyp in diesem Programm. Die Interfaces sind sehr ähnlich den Klassen und somit auch ein Konstrukt für einen neuen Datentyp. Ein Array Typ repräsentiert eine Liste von Werten eines anderen Typs. Zum Beispiel ist char ein Datentyp und ein char[] (Array von char) ein anderer Datentyp.

Null-Typ

Da der null-Typ keinen Namen besitzt, ist es auch nicht möglich, eine Variable dieses Typs zu deklarieren oder ein cast auf *null* durchzuführen. Die null-Referenz ist der einzige mögliche Wert für einen Ausdruck eines null-Typs. Die null-Referenz kann zu jedem beliebigen Referenztyp mittels cast umgewandelt werden.

String-Literal

Der Typ String ist eine Klasse und kein Vertreter der primitiven Datentypen. Da aber Zeichenketten weit verbreitet sind, gibt es in der Java-Syntax die Möglichkeit, Zeichenketten direkt im Programm zu verwenden. Ein String-Literal beinhaltet einen beliebigen Text innerhalb von doppelten Hochkommas, z.B.: `"hallo"`.

Zeichenketten können durch den `+` und `+=` Operator mit anderen Zeichenketten verknüpft werden. Der Java Interpreter hat eine eingebaute Zeichenkettenkonvertierung für alle primitiven Datentypen. Ein Objekt wird zu einem String konvertiert, indem die Methode toString() aufgerufen wird. Arrays werden ebenfalls durch diese Methode zu einem String konvertiert[2].

Objekt

Ein Objekt ist eine dynamisch erstellte Instanz eines Klassentyps oder eines Arraytyps. Alle Objekte, auch Arrays, unterstützen die Methoden des Objektes *Object*. Die Referenzwerte sind Zeiger auf diese Objekte, wobei die spezielle null-Referenz auf kein Objekt referenziert.

3.3.3 Klassen

In Java ist es nicht möglich, einen Code zu erstellen, in dem keine Klasse definiert ist. Alle Java Anweisungen befinden sich innerhalb von Methoden und alle Methoden sind wiederum innerhalb von Klassen definiert. Durch die Deklaration einer Klasse wird in Java ein neuer Referenztyp definiert und beschrieben, wie dessen Implementierung aussieht. Jede Klasse in Java mit Ausnahme von *Object* ist eine Erweiterung (Subklasse) einer existierenden Klasse und implementiert möglicherweise auch ein Interface, siehe Abschnitt 3.3.4, Seite 18.

Klasse Object

Die Klasse *Object* ist die Wurzelklasse in Java. Alle Klassen sind Subklassen von Object und können somit die public und protected Methoden dieser Klasse aufrufen.

```
public class Object {
  public Object();

  public boolean equals(Object obj);
  public final Class getClass();
  public int hashCode();
  public final void notify();
  public final void notifyAll();
  public String toString();
  public final void wait() throws InterruptedException;
  public final void wait(long timeout) throws InterruptedException;
  public final void wait(long timeout, int nanos)
                       throws InterruptedException;

  protected Object clone() throws CloneNotSupportedException;
  protected void finalize() throws Throwable;
}
```

Klassen, die das Interface Cloneable unterstützen, können mit der Methode clone() Byte für Byte Kopien ihrer Objekte erstellen.

[2]Bei einem Array jedoch repräsentiert die resultierende Zeichenkette in der Regel keine brauchbare Zeichenkette. Dieses Problem kann durch eine selbstdefinierte toString() Methode umgangen werden.

Die Methode getClass() gibt das Objekt Class, welches mit jedem Objekt assoziiert ist, zurück. Die Methoden notify(), notifyAll() und wait() sind für die Thread Synchronisation von Objekten einsetzbar.
Einige dieser Object-Methoden sollten von Object-Subklassen überschrieben werden. Zum Beispiel sollte eine Subklasse ihre eigene Definition der Methode toString() zur Verfügung stellen, damit Verknüpfungsoperationen mit Zeichenketten und die Methode PrintWriter.println() sinnvolle Ergebnisse liefern. Durch die Definition der Methode toString() für alle Objekte wird das Suchen von Fehlern oft erleichtert.
Die standardmäßige Implementierung der Methode equals() verwendet den Operator == zum Testen, ob sich diese Objektreferenz auf dasselbe Objekt bezieht wie eine zweite Objektreferenz. Wenn Subklassen individuelle Felder von zwei unterschiedlichen Objekten vergleichen wollen, sollte die Methode equals() von der Subklasse dementsprechend überschrieben werden. Einige Klassen, unter anderem solche, welche die Methode equals() überschreiben, überschreiben auch die Methode hashCode(), um einen adäquaten Hashcode beim Abspeichern in einer Hashtable-Datenstruktur zu generieren.
Eine Klasse welche Systemressourcen zum Beispiel auf Dateien allokiert, sollte die Methode finalize() überschreiben, um diese nicht mehr benötigten Ressourcen wieder freizugeben.

Mitglieder

Der Block einer Klasse deklariert Mitglieder (Felder, Methoden, nested classes, Interfaces), Instanzen, statische Initialisierungen und Konstruktoren. Es gibt folgende vier Arten:

- Klassenfelder
- Klassenmethoden
- Instanzfelder
- Instanzmethoden

Die folgende einfache Klassendefinition der Klasse Circle zeigt alle vier genannten Mitgliedertypen:

```
public class Circle {
  // Ein Klassenfeld
  public static final double PI=3.14159;

  // Eine Klassenmethode
  public static double radiansToDegrees(double rads) {
    return rads * 180 / PI;
  }

  // Ein Instanzfeld
  public double r;

  // Eine Instanzmethode, operiert mit Instanzfeldern eines Objektes
  public double area() {
    return PI * r * r;
  }
}
```

Das Schlüsselwort *static* bei Variablen sagt aus, dass das Feld ein Klassenfeld ist und es genau eine Kopie für alle Instanzen dieser Klasse gibt. Das Schlüsselwort *final* sagt aus, dass sich der Wert des Feldes nicht ändern kann. Somit repräsentiert PI eine Konstante in Java, diese werden üblicherweise mit Großbuchstaben geschrieben. Oft werden Klassenfelder als Konstanten (static und final) definiert und im Wesentlichen wie eine globale Variable eingesetzt. Innerhalb der Klasse Circle kann das Feld einfach als PI referenziert werden. Ausserhalb der Klasse müssen sowohl Klassenname und Feldname für die eindeutige Referenzierung angegeben werden. Methoden, die nicht Teil der Klasse Circle sind, greifen via Circle.PI auf das Feld zu.
So wie die Klassenfelder sind die Klassenmethoden mit der Klasse assoziiert und nicht mit einem Objekt. Auch Klassenmethoden müssen ausserhalb der Klasse mit dem Klassen- und dem Methodennamen spezifiziert werden, z.B.: Circle.radiansToDegrees(2.0).

Eine Klassenmethode kann Klassenfelder verwenden, jedoch können keine Arten von Instanzfeldern oder Instanzmethoden verwendet werden. Dies ist damit begründet, dass Klassenmethoden nicht mit einer Instanz dieser Klasse verbunden sind. So wie das Klassenfeld einer globalen Variable entspricht, entspricht die Klassenmethode einer globalen Methode.
Wie in nicht objektorientierten Programmiersprachen können auch in Java Programme erstellt werden, in denen alle Methoden oder Funktionen global sind. Dies ist jedoch nicht im Sinne des objektorientierten Paradigmas, bei dem die Instanzfelder und Instanzmethoden eine wichtige Rolle spielen. Jedes Feld, das ohne das Schlüsselwort *static* definiert ist, ist ein Instanzfeld. Instanzfelder sind nicht mit der Klasse assoziiert, sondern mit den Instanzen der Klasse. Jedes Objekt der Klasse Circle im obigen Beispiel hat seine eigene Kopie des Feldes r und ist somit unabhängig von allen anderen Circle-Objekten. Instanzfelder definieren ein Objekt, und die Werte dieser Felder bewirken, dass Objekte sich voneinander unterscheiden. Außerhalb des Klassencodes muss der Name eines Instanzfeldes mit dem entsprechenden Objekt referenziert werden.

```
Circle c = new Circle();  // Neues Circle Objekt
c.r = 2.0;                // Wertzuweisung auf das Instanzfeld r
Circle d = new Circle();  // Erstellt ein anderes Circle Objekt
d.r = c.r * 2;            // Wert ist zweimal so gross

double a = c.area();      // Ruft Instanzmethode des Objektes auf
```

In diesem Beispiel wird ein neues Circle-Objekt mit dem Namen c erstellt und kann jetzt das Instanzfeld r als c.r referenzieren. Jede nicht mit dem Schlüsselwort *static* deklarierte Methode ist eine Instanzmethode. Eine Instanzmethode operiert auf der Instanz einer Klasse – somit auf einem Objekt anstatt auf der Klasse selbst. Die Instanzmethode area() berechnet die Fläche des Kreises anhand des aktuellen Circle Objektes. Um eine Instanzmethode ausserhalb der Klasse, in der sie definiert wurde, zu verwenden, muss wie bei den Instanzfeldern davor die Instanz des gewünschten Objektes gestellt werden (c.area()).
Der Unterschied zwischen Variablen und Feldern in Java ist der, dass Variablen innerhalb von Methoden definiert werden, während Felder Mitglieder (members) von Klassen darstellen.

Abstrakte Klasse

Eine abstrakte Klasse ist eine Klasse, die unvollständig ist oder als solche betrachtet wird. Nur abstrakte Klassen dürfen abstrakte Methoden haben. Abstrakte Methoden sind Methoden, die zwar deklariert, aber (noch) nicht implementiert sind. Wenn eine Klasse eine abstrakte Methode beinhaltet und die Klasse nicht als abstrakt deklariert wurde, so generiert der Compiler eine entsprechende Fehlermeldung. Eine Klasse A hat abstrakte Methoden, wenn eine der folgenden Aussagen zutrifft:

- A hat explizit eine abstrakte Methode deklariert.
- Eine der Superklassen von A hat eine abstrakte Methode deklariert, die weder in A noch in einer unterhalb dieser Superklasse liegenden Superklasse implementiert ist.
- Ein direktes Superinterface von A deklariert oder erbt eine Methode, welche notwendigerweise abstrakt sein muss, und weder A deklariert noch erbt eine Methode, die diese implementiert.

```
abstract class Point {                         // abstrakte Klasse
  int x = 1; int y = 1;
  void move(int dx, int dy) {
    x += dx;
    y += dy;
    alert();
  }

  abstract void alert()                        // abstrakte Methode
}

abstract class ColoredPoint extends Point {    // abstrakte Klasse
  int color;
}

class SimplePoint extends Point {              // konkrete Klasse
  void alert() { }
}
```

In dem Beispiel wird eine Klasse Point zwingendermaßen als abstrakt deklariert, weil die abstrakte Methode alert() in ihr enthalten ist. Weil die Subklasse von Point ColoredPoint die abstrakte Methode erbt, muss auch diese Klasse abstrakt deklariert werden. Andererseits braucht die Klasse SimplePoint nicht abstrakt deklariert zu sein, da sie die geerbte Methode alert() implementiert. In diesem Beispiel kann von den abstrakten Klassen Point und ColoredPoint keine Instanz erstellt werden. Der Compiler würde ein solches Unterfangen durch eine entsprechende Fehlermeldung mitteilen. Die folgende Anweisung ist jedoch korrekt:

```
Point p = new SimplePoint();
```

Hierbei wird eine nicht abstrakte Subklasse einer abstrakten Klasse instanziert und der DefaultKonstruktor sowie die Initialisierungen der Felder x und y von der Klasse Point durchgeführt. Eine Klasse sollte nur dann als abstrakt deklariert werden, wenn es die

Intention gibt, dass eine Subklasse die entsprechenden Implementationen komplettiert. Wenn die Intention ist, dass eine Klasse nicht instanzierbar sein soll, ist es sinnvoller, nur einen Konstruktor ohne Argumente als private zu deklarieren. Eine Klasse dieser Form beinhaltet in der Regel Methoden und Variablen.

```
public final class Car {   // ist durch final nicht erweiterbar
  private Car() {          // kann nicht instanziert werden
    ...                    // Deklaration von Methoden und Variablen
  }
}
```

Innere Klassen

Seit Java Version 1.1 gibt es neben den Top-Level Klassen noch vier andere Typen von Klassen, die als innere Klassen bezeichnet werden.

- *Statische Mitgliederklassen (static member class)* sind Klassen oder Interfaces, definiert als statische Teile einer anderen Klasse. Sie entsprechen der Analogie der Felder und Methoden einer Klasse, die gleichfalls als statisch deklariert wurden. Eine statische Mitgliederklasse hat Zugriff auf alle statischen Mitglieder ihrer enthaltenen Klasse, auch auf die privaten Mitglieder.

  ```
  public class LinkedStack {
    ...
    // Statisches Mitgliedsinterface
    public static interface Linkable { ... }
    }
    ...
  }
  // Implementierung des statischen Mitgliedsinterfaces
  class LinkableInteger implements LinkedStack.Linkable { ... }
  ```

  Das Beispiel zeigt auch, wie das Interface innerhalb der Klasse und von externen Klassen verwendet werden kann. Zu beachten ist auch die Bezeichnung des hierarchischen Namens in der externen Klasse.

- *Mitgliederklassen* sind Klassen, die als nicht statische Mitglieder von Klassen oder Interfaces definiert sind. So wie die Analogie zwischen statischen Mitgliederklassen zu den Feldern und Methoden einer Klasse, sind die Mitgliederklassen analog zu den Instanzfeldern und Instanzmethoden.

  ```
  public class LinkedStack {
    ...
    // Enumeration Schnittstelle als Mitgliedsklasse
    protected class Enumerator implements java.util.Enumeration { ... }
    }
    ...
  }
  ```

- *Lokale Klassen* werden lokal innerhalb eines Java Codeblockes deklariert, im Gegensatz zu Mitgliedern einer Klasse. Typischerweise wird eine lokale Klasse innerhalb einer Methode definiert. Da alle Blöcke eines Java-Codes in Klassendefinitionen auftreten, sind alle lokalen Klassen ebenfalls in Klassen verschachtelt. Aus diesem Grund teilen lokale Klassen viele Gemeinsamkeiten mit Mitgliederklassen. Die Definitionscharakteristik einer lokalen Klasse ist die eines lokalen Blocks in einem Code. Wie die lokalen Variablen sind die lokalen Klassen nur innerhalb des Sichtbarkeitsbereichs ihres eingeschlossenen definierten Blocks gültig.

```
public class LinkedStack {
  ...
  // Erstellt und retourniert ein Enumeration Objekt
  public java.util.Enumeration enumerate() {

    // Die Definition des Enumerators als lokale Klasse
    class Enumerator implements java.util.Enumeration { ... }
    // Retourniert die Instanz der neuen Enumerator Klasse
    return new Enumerator();
  }
  ...
}
```

  Lokale Klassen können nicht als public, protected, private oder static deklariert werden.

- *Anonyme Klassen* sind lokale Klassen ohne Namen. Eine anonyme Klasse wird sehr knapp definiert und mit dem *new* Operator instanziert. Während eine lokale Klassendefinition eine Anweisung in einem Block im Java-Code darstellt, ist die anonyme Klassendefinition ein Ausdruck, was bedeutet, dass dieser als Teil eines größeren Ausdrucks wie dem eines Methodenaufrufs inkludiert werden kann. Wenn eine lokale Klasse nur einmal benötigt wird, ist zu betrachten, ob die Verwendung der anonymen Klassensyntax nicht stattdessen zu benutzen ist.

```
public class LinkedStack {
  ...
  public java.util.Enumeration enumerate() {
    // Anonyme Klasse ist definiert als Teil der return Anweisungen.
    return new java.util.Enumeration { ...
    };  // Notwendiges Semikolon fuer die return Anweisung
  }
  ...
}
```

Finalisieren von Klassen

Eine Klasse kann als *final* markiert werden, wenn ihre Definition vollständig ist und keine Subklassen gewünscht oder benötigt werden. Eine Meldung zum Zeitpunkt des Compilierens würde auftreten, wenn der Name einer finalisierten Klasse in einer *extends* Klausel einer anderen Klassendeklaration aufscheint. Das impliziert, dass eine finalisierte Klasse keine Subklassen haben kann. Die Kombination *final* und *abstract* in einer Klasse würde ebenfalls einen Compilezeit-Fehler erwirken, weil die Implementierung einer solchen Klasse niemals vervollständigt werden könnte.

Schlüsselwort extends

Das optionale Schlüsselwort *extends* in einer Klassendeklaration spezifiziert die direkte Superklasse der aktuellen Klasse. Wenn eine Klasse keine *extends*-Klausel hat, dann hat diese Klasse implizit die Klasse *Object* als Superklasse. Die Klasse *Object* ist die Wurzelklasse in Java und hat selbst keine direkte Superklasse.

```
class Point { int x; int y; }
final class ColoredPoint extends Point { int color; }
```

Folgende Verhältnisse weisen obere Klassen auf:

- Die Klasse Point ist eine direkte Subklasse von Object.
- Die Klasse Object ist die direkte Superklasse von der Klasse Point.
- Die Klasse ColoredPoint ist eine direkte Subklasse von der Klasse Point.
- Die Klasse Point ist die direkte Superklasse von der Klasse ColoredPoint.

Wenn Klassen von sich selbst abhängig sind, wird ein Fehler vom Compiler gemeldet. Die folgenden zyklisch deklarierten Klassen verursachen einen solchen Fehler:

```
class Point extends ColoredPoint { int x; int y; }
class ColoredPoint extends Point { int color; }
```

Schlüsselwort implements

Das optionale Schlüsselwort *implements* in einer Klassendeklaration listet die Interfaces (siehe Abschnitt 3.3.4, Seite 18) auf, die die direkten Superinterfaces der aktuellen Klasse sind.

```
public interface Colorable { ... }
public interface Paintable extends Colorable { ... }

class Point { ... }
class ColoredPoint extends Point implements Colorable { ... }
class PaintedPoint extends ColoredPoint implements Paintable { ... }
```

Die folgenden Verhältnisse ergeben sich dadurch:

- Das Interface Paintable ist ein Superinterface der Klasse PaintedPoint.
- Das Interface Colorable ist ein Superinterface der Klassen ColoredPoint und PaintedPoint.
- Das Interface Paintable ist ein Subinterface des Interfaces Colorable und Colorable ist ein Superinterface von Paintable.

Eine Klasse kann ein Superinterface auf mehreren Wegen besitzen. Im oberen Beispiel hat die Klasse PaintedPoint das Superinterface Colorable zweimal; einmal als Superinterface durch die Erweiterung von ColoredPoint und einmal durch das Superinterface Paintable. Wenn die Klasse nicht als abstract deklariert ist, müssen alle deklarierten Methoden des Superinterfaces entweder in dieser oder in einer der Superklassen implementiert sein. Es ist möglich, mit nur einer Methode in einer Klasse Methoden von mehreren Superinterfaces zu implementieren.

```
interface Person { int getName(); }
interface Student { int getName(); }

class Inskribient implements Person, Student {
  int getName() { return "Peter"; }
}
```

Die Methode getName() in Klasse Inskribient hat einen Namen, Signatur und einen Rückgabewert, der auf die Interfaces Person und Student zutrifft und somit beide implementiert.

3.3.4 Interfaces

Durch die Deklaration eines Interfaces wird ein neuer Referenztyp eingeführt, dessen Mitglieder Klassen, Interfaces, Konstanten und abstrakte Methoden sind, jedoch hat dieser Typ keine Implementierung. Die abstrakten Methoden können von beliebigen Klassen implementiert werden. Die meisten Aussagen über Klassen gelten auch für Interfaces. Die Definition eines Interfaces ist sehr ähnlich der Definition einer abstrakten Klasse mit dem Unterschied, dass die Schlüsselwörter *abstract* und *class* durch *interface* ersetzt werden. Es gibt ein paar Einschränkungen, welche die Mitglieder von Interfaces betreffen:

- Ein Interface beinhaltet keine Implementierung. Alle Methoden eines Interfaces sind implizit abstrakt, auch ohne den Modifikator *abstract*. Statt dem Methodenteil wird ein Semikolon (;) geschrieben. Weil Interfaces nur abstrakte Methoden haben können und Klassenmethoden nicht abstrakt sein können, müssen alle Methoden von Interfaces Instanzmethoden sein.

- Ein Interface definiert ein API (Application Programming Interface). Alle Methoden eines Interfaces sind implizit public, auch wenn der public-Identifikator fehlt.

- Obwohl Klassen Daten und Methoden, die auf diesen Daten operieren, definieren, kann ein Interface keine Instanzfelder definieren.
  Dies auf Grund der Tatsache, dass Felder Details der Implementierung sind und Interfaces reine Spezifikationen ohne Implementierung darstellen. Die einzige Ausnahme von Feldern, die in einem Interface erlaubt sind, stellen die Konstanten dar, welche als *static* und *final* deklariert sein müssen.

- Im Gegensatz zu den Klassen können Interfaces nicht lokal innerhalb eines Codeblockes definiert werden.

- Ein Interface kann nicht instanziert werden, somit gibt es auch keinen Konstruktor.

Das folgende Beispiel zeigt ein Interface mit dem Namen Centered:

```
public interface Centered {
  public void setCenter(double x, double y);
  public double getCenterX();
  public double getCenterY();
}
```

In Java kann eine Klasse nur eine Superklasse mit *extends* erweitern, jedoch können mehrere Interfaces mit *implements* implementiert werden. Wenn eine Klasse ein Interface in ihrer *implements*-Klausel deklariert, bedeutet das, dass in dieser Klasse für jede Methode des Interfaces eine Implementierung vorhanden ist. Wenn dies jedoch nicht der Fall ist und die Klasse ein oder mehrere Implementierungen nicht anbietet, so erbt sie diese nicht implementierten abstrakten Methoden vom Interface und muss mit dem Schlüsselwort *abstract* deklariert werden. Falls mehrere Interfaces implementiert werden, müssen alle Methoden aller Interfaces implementiert sein oder die Klasse wird als abstrakt deklariert. Beispiel einer Implementierung des Interfaces Centered von oben:

```
public class CenteredRectangle extends Rectangle implements Centered {
  // Instanzfelder
  private double xy, cy;

  // Konstruktor
  public CenteredRectangle(double cx, double cy, double w, double h) {
    ...
  }

  // Es werden alle Methoden von Rectangle geerbt, jedoch muessen
  // alle Centered Methoden implementiert werden.
  public void setCenter(double x, double y) { ... }
  public double getCenterX() { ... }
  public double getCenterY() { ... }
}
```

Interfaces können auch Subinterfaces besitzen, so wie Klassen Subklassen besitzen können. Ein Subinterface erbt alle abstrakten Methoden und Konstanten des Superinterfaces und kann neue abstrakte Methoden und Konstanten definieren. Anders als bei den Klassen unterscheiden sich Interfaces in einem wichtigen Punkt, sie können mehr als ein Superinterface nach dem *extends* Schlüsselwort auflisten. Auch ein solches Interfaces erbt alle abstrakten Methoden und Konstanten von jedem Interface und kann selber noch weitere abstrakte Methoden und Konstanten definieren. Eine Klasse, die ein solches Interface implementiert, muss sowohl die abstrakten Methoden vom direkten Interface, als auch die abstrakten Methoden, die von den Superinterfaces geerbt werden, implementieren.

Manchmal ist es sinnvoll, ein Interface zu definieren, das vollständig leer ist. Eine Klasse kann dieses Interface einfach durch Angabe dessen Namens nach dem Schlüsselwort *implements* realisieren, ohne dass eine Methode implementiert werden muss. In diesem Fall ist jede Instanz dieser Klasse auch eine gültige Instanz des Interfaces. In Java kann mit dem *instanceof* Operator überprüft werden, ob ein Objekt eine Instanz dieses Interfaces ist oder nicht. Diese Technik ermöglicht es, eine weitere Information über ein Objekt zu erhalten. Das Interface Cloneable in java.lang ist ein Beispiel dieser *Marker Interface* Technik. Es kennzeichnet die Klasse als eine, in der der interne Zustand einer Instanz mit der Methode clone() in der Object-Klasse geklont werden darf.

3.3.5 Arrays

Der Typ Array ist ein Vertreter von Referenztypen in Java. In Java sind Arrays Objekte, sie werden dynamisch erstellt und können an Variablen des Typs Object zugewiesen werden. Alle Methoden der Klasse Object können auch im Array aufgerufen werden. Ein Array stellt eine nummerierte Liste von Werten dar. Die Werte können primitive Werte, Objekte oder auch Arrays sein, mit der Einschränkung, dass alle Werte eines Arrays vom selben Typ sein müssen. Der Typ eines Arrays ist der Typ von Werten die in diesem Array enthalten sind, gefolgt von den Zeichen [ ], zum Beispiel:

```
byte[] arrayOfBytes;                  // byte[] Arrays von byte
byte[][] arrayOfArrayOfBytes;         // byte[][] Arrays von byte[]
Point[] points;                       // Point[] Array von Point Objekten

String [] responses = new String[2]; // Array von zwei Zeichenketten
responses[0] = "Ja";                  // Setzt das erste Element im Array
responses[1] = "Nein";                // Setzt das zweite Element im Array
```

3.3.6 Überschreiben, Verdecken und Überladen

Eine Klasse erbt von ihrer direkten Superklasse und direkten Superinterfaces alle nicht privaten Methoden, gleichgültig ob sie abstrakt sind oder nicht, sofern sie in der Klasse selbst nicht durch eine Deklaration verdeckt (hidden) oder außer Kraft gesetzt (overridden) sind.
Wenn eine Methodendeklaration die Deklaration einer anderen Methode verdeckt oder außer Kraft setzt und die Rückgabewerte unterschiedlichen Typ aufweisen, tritt ein Compilezeit-Fehler auf. Weiters dürfen auch die *throws*-Klauseln (siehe Abschnitt 3.3.11, Seite 31) keinen Konflikt mit den außer Kraft gesetzten oder verdeckten Methoden aufweisen, ansonsten tritt ebenfalls ein vom Compiler gemeldeter Fehler auf.

Überschreiben (Overriding) von Instanzmethoden

Eine deklarierte Instanzmethode m2 in einer Klasse B überschreibt eine andere deklarierte Methode m1 mit derselben Signatur in der Klasse A, wenn und nur wenn die beiden Bedingungen erfüllt sind:

1. B ist eine Subklasse von A
2. Entweder
   - ist m1 nicht privat und zugreifbar von B aus, oder
   - m2 überschreibt eine Methode m3 und m3 ist verschieden von m2 und m3 ist verschieden von m1 und m3 überschreibt m1.

   Wenn außerdem noch gilt, dass m2 nicht abstrakt ist, hat m2 die Eigenschaft, dass alle beliebigen von m2 außer Kraft gesetzten als abstrakt deklarierten Methoden durch m2 implementiert sind.

Ein Compilezeit-Fehler tritt auf, wenn eine Instanzmethode versucht, eine statische Methode außer Kraft zu setzen. Durch diesen Gesichtspunkt unterscheidet sich das Überschreiben von Methoden vom Verdecken von Feldern, da es für Instanzvariablen erlaubt ist, eine statische Variable zu verdecken. Auf eine überschriebene Methode kann durch den Ausdruck des Methodenaufrufs mit dem Schlüsselwort *super* zugegriffen werden. Das folgende Beispiel zeigt den Aufruf einer außer Kraft gesetzten Methode:

```
class A {
  int i = 1;                    // Instanzfeld
  int f() { return i; }         // Instanzmethode
}
class B extends A {
  int f() {                     // setzt Methode f() in A ausser Kraft
    return super.f() + 1;       // Aufruf von A.f()
  }
}
```

Verdecken (hiding) von Klassenmethoden

Wenn eine Klasse eine statische Methode deklariert, dann sagt man, dass die Deklaration dieser Methode alle Methoden mit derselben Signatur in den Superklassen und Superinterfaces, die durch diese Klasse zugreifbar sind, verdeckt. Bei den Interfaces gilt diese Aussage ebenfalls, jedoch mit der Einschränkung, dass nur Methoden von Superinterfaces und nicht von Superklassen verdeckt werden können. Ein Compilezeit-Fehler tritt auf, wenn versucht wird, eine Instanzmethode durch eine statische Methode zu verdecken. Durch diesen Gesichtspunkt unterscheidet sich das Verdecken von Methoden vom Verdecken von Feldern, da es für statische Variablen möglich ist, Instanzvariablen zu verdecken. Man beachte, dass es zwischen hiding, shadowing und obscuring Unterschiede gibt. Eine verdeckte Methode kann durch den qualifizierten Namen oder durch den Ausdruck eines Methodenaufrufs mit dem Schlüsselwort *super* aufgerufen werden.

```
class A {
  static char g() { return 'A'; }     // Eine Klassenmethode
}
class B extends A {
  static char g() { return 'B'; }     // verdeckt Klassenmethode g() in A
}
public class HiddenTest {
  public static void main(String args[]) {
  B b = new B();                      // Neues Objekt B
  System.out.println(b.g());          // verweist auf B.g(); gibt B aus
  System.out.println(B.g());          // bessere Variante; gibt B aus

  A a = (A) b;                   // Castet b zu einer Instanz der Klasse A
  System.out.println(a.g());     // verweist auf A.g(); gibt A aus
  System.out.println(A.g());     // bessere Variante; gibt A aus
}
```

Überladen (Overloading)

Wenn zwei Methoden einer Klasse oder eines Interfaces den gleichen Namen, aber unterschiedliche Signaturen aufweisen und diese Methoden entweder

- in derselben Klasse oder demselben Interface deklariert sind, oder
- beide in dieser Klasse oder diesem Interface geerbt werden, oder
- eine deklariert und eine wird geerbt wird,

bezeichnet man diese Methoden als überladen. Aus der Sicht der Sprachimplementierung bereitet dieses Prinzip keine Schwierigkeit, womit es auch niemals zu einem Compilezeit-Fehler kommen kann. Es besteht auch keine Beziehung zwischen den Rückgabewerten oder *throws*-Klauseln von zwei Methoden mit demselben Namen aber unterschiedlichen Signaturen. Wenn zum Beispiel eine Klasse zwei public Methoden mit dem gleichen Namen deklariert und eine Subklasse eine dieser Methoden außer Kraft setzt, dann erbt die Subklasse weiterhin noch die andere Methode.

```
class Point {
  int x = 0, y = 0;
  void move(int dx, int dy) { ... }
  void move(float dx, float dy) { ...  }
  int color;
}

class RealPoint extends Point {
  float x = 0.0f, y = 0.0f;              // hiding
  void move(int dx, int dy) { ... }      // overriding
}
```

Die Klasse RealPoint verdeckt die Deklaration der Instanzvariablen x und y der Klasse Point mit seinen eigenen float Instanzvariablen x und y. Die erste Instanzmethode in RealPoint setzt die Instanzmethode move der Klasse Point außer Kraft. Die zweite Instanzmethode überlädt (overload) den Namen move mit einer weiteren Methode gleichen Namens unterschiedlicher Signatur. Welche der beiden überladenen move-Methoden der Klasse Realpoint ausgewählt wird, wird zur Compilezeit entschieden.

Obskure Deklarationen (obscures)

Es kann vorkommen, dass ein einfacher Name als Variable, Typ oder Paket gedeutet wird. Basierend auf den Regeln, dass eine Variable vor einem Typ und ein Typ vor einem Paket ausgewählt wird, kann es zu der Situation kommen, dass auf eine sichtbare Deklaration eines Typs oder Paketes nicht Bezug genommen werden kann. Eine solche Deklaration wird als obskur bezeichnet. Durch das Einhalten einer Namenskonvention kann das Phänomen der obskuren Deklarationen reduziert werden.

Abgeschattete Deklarationen (shadowing)

Einige Deklarationen können auf Teile ihres Bereiches durch eine andere Deklaration mit demselben Namen dadurch abgeschattet werden, dass auf die deklarierte Entität nicht mit dem einfachen Namen Bezug genommen werden kann.

```
class Test {
  static int x = 1;                          // statische Klassenvariable
  public static void main(String[] args) {
    int x = 0;                               // lokale Variable
    System.out.print("x=" + x);
    System.out.println(", Test.x=" + Test.x);
  }
}
```

Im Beispiel bezieht sich der Ausdruck x im Aufruf von print auf den Wert der lokalen Variablen x. Der Aufruf von println verwendet einen qualifizierten Namen Test.x, der mit dem Namen des Klassentyps Test auf die Klassenvariable x zugreift. Dies ist notwendig, weil die Deklaration von Test.x zu diesem Zeitpunkt abgeschattet ist und nicht mit dem einfachen Namen bezeichnet werden kann.

Weitere Möglichkeiten, um auf abgeschattete Felder zuzugreifen besteht mit dem Schlüsselwörtern *this* und *super* und mit der cast-Technik. Gegeben sei folgende Klassenhierarchie:

```
public class A {
 int x;
}
public class B extends A {
 int x;
}
public class C extends B {
 int x;
}
```

In den Methoden der Klasse C kann auf die verschiedenen Felder wie folgt zugegriffen werden:

```
  x             // Feld x in Klasse C
  this.x        // Feld x in Klasse C
  super.x       // Feld x in Klasse B
  ((B)this).x   // Feld x in Klasse B
  ((A)this).x   // Feld x in Klasse A
```

Der Aufruf von super.super.x in einer Methode der Klasse C ist syntaktisch falsch und daher nicht möglich.

3.3.7 Zugriffskontrolle

Wie bereits erwähnt besteht eine Klasse aus einer Kollektion von Daten und Methoden. Die Technik, Daten innerhalb der Klassen unsichtbar zu machen und vor unberechtigten Methodenaufrufen zu schützen, nennt man „information hiding“˙

All die Felder und Methoden einer Klasse können innerhalb dieser Klasse immer verwendet werden. Java bietet Regeln für die Zugriffsmöglichkeiten von Klassenmitgliedern ausserhalb der verwendeten Klasse an. Die Schlüsselwörter public, protected und private werden als „access control modifiers“ bezeichnet und spezifizieren die Zugriffsregeln für Felder und Methoden.

Zugriff auf Pakete

Ein Paket ist immer zugreifbar von Code innerhalb dieses Paketes. Ob ein Paket auch zugreifbar von anderen Paketen ist, ist vom Host des installierten Paketes abhängig. Wenn zum Beispiel die Klassendateien von Paketen in einem Verzeichnis abgelegt sind, muss der Benutzer dieser Pakete Leserechte auf dieses Verzeichnis besitzen. Paketzugriffe sind also nicht Teil der Java Programmiersprache.

Zugriff auf Klassen

Standardmäßig sind Klassen auf oberster Ebene (keine nested Klassen) innerhalb des Paketes, in dem sie definiert sind, zugreifbar. Wenn eine Klasse der obersten Ebene als public deklariert ist, ist sie von überall zugreifbar, genauer gesagt von überall wo auch das Paket selbst zugreifbar ist.

Zugriff auf Mitglieder

Standardmäßig sind die Mitglieder einer Klasse nicht nur innerhalb der Klasse zugreifbar, sondern auch durch Klassen, die im selben Paket definiert sind. Das impliziert, dass Klassen, die sich im selben Paket befinden, einander bezüglich der internen Implementierungsdetails vertrauen sollten. Diese standardmäßige Stufe des Zugriffs wird oft „package access“ genannt. Das ist eine der vier möglichen Stufen, weitere Stufen werden durch die Modifikatoren public, protected und private definiert.

- Wenn ein Mitglied in einer Klasse mit dem public Modifikator deklariert ist, bedeutet dies, dass das Mitglied von überall dort ansprechbar ist, von wo auch die Klasse selbst ansprechbar ist. Das ist die schwächste Art der Zugriffskontrolle.

- Wenn ein Mitglied in einer Klasse mit dem private Modifikator deklariert ist, bedeutet dies, dass das Mitglied niemals ansprechbar ist, ausgenommen von der Klasse in der es deklariert ist. Das ist die restriktivste Art der Zugriffskontrolle.

- Wenn ein Mitglied in einer Klasse mit dem protected Modifikator deklariert ist, bedeutet dies, dass das Mitglied von allen Klassen innerhalb des Pakets (entspricht dem standardmäßigen package access) ansprechbar ist. Ausserdem ist diese Klasse innerhalb von Subklassen dieser Klasse, unabhängig davon, in welchem Paket diese Subklasse definiert ist, ansprechbar. Dies ist restriktiver als der public Zugriff, aber weniger restriktiv als der package Zugriff.

- Wenn ein Mitglied in einer Klasse ohne Modifikator deklariert ist, hat des den standardmäßigen package access. Dieses Mitglied ist von allen Klassen ansprechbar, die im selben Paket definiert sind. Alle ausserhalb des Pakets liegenden Codeteile können nicht zugreifen.

Wenn man sich nicht sicher ist, ob der protected, package access oder private Zugriff der richtige Mechanismus ist, ist es besser mit dem restriktivsten Zugriffsmechanismus zu beginnen. Das Lockern der Zugriffsbeschränkung zu einem späteren Zeitpunkt ist jederzeit bei Bedarf möglich. Ein umgekehrtes Verfahren stellt keine gute Idee dar, denn dieser Schritt ist keine abwärtskompatible Veränderung. Tabelle 3.2 fasst die Zugriffsregeln zusammen.

	Mitglied Sichtbarkeit			
Ansprechbar von	*public*	*protected*	*package*	*private*
Definierter Klasse	ja	ja	ja	ja
Klasse im selben Paket	ja	ja	ja	nein
Subklasse in einem anderen Paket	ja	ja	nein	nein
Nicht-Subklasse von einem anderen Paket	ja	nein	nein	nein

Tabelle 3.2: Zugriffsmöglichkeiten auf Klassenmitglieder

3.3.8 Methoden

Eine Methode ist eine mit Namen versehene Sequenz von Java Anweisungen, die von anderem Java-Code aufgerufen werden kann. Wenn eine Methode aufgerufen wird, werden Werte als Parameter übergeben. Der interessanteste Teil einer Methode ist die Signatur, die wie folgt spezifiziert ist [Flanagan, 2002]:

- Name der Methode
- Anzahl[3], Reihenfolge, Typ und Namen der verwendeten Parameter
- Typ des Rückgabewertes dieser Methode
- Überprüfte Ausnahmebedingungen, die in der Methode geworfen werden. Die Signatur kann auch ungeprüfte Ausnahmebedingungen auflisten, aber diese sind irrelevant.
- Verschiedene Methodenmodifikatoren, die zusätzliche Informationen über die Methode liefern.

Kurz gesagt definiert eine Methodensignatur alle notwendigen Informationen, die für einen Methodenaufruf benötigt werden. Dies stellt die *Spezifikation* der Methode dar und definiert das API (Application Programming Interface) für die Methode. Eine Methodensignatur sieht in etwa so aus:

modifiers type name (*paramlist*) [throws *exceptions*]

- *Die Modifikatoren (modifiers)* können keine oder mehrere spezielle Schlüsselwörter, die mit Leerzeichen getrennt sind, sein. Zum Beispiel kann eine Methode als public und static deklariert werden. Andere gültige Modifikatoren sind abstract, final, native, private, protected und synchronized.
- *Der Typ (type)* spezifiziert den Rückgabewert der Methode. Wenn die Methode einen Wert retourniert, so bezeichnet *type* einen primitiven Datentyp, einen Arraytyp oder eine Klasse, andernfalls muss *type* als void spezifiziert werden. Ein Konstruktor stellt eine spezielle Methode dar, dessen Signatur keinen *type* spezifiziert.
- *Name (name)* einer Methode erfolgt im Anschluss an die Modifikatoren und dem Typ. Methodennamen sind genauso wie Variablennamen Javaidentifikatoren. Alle Javaidentifikatoren können Buchstaben jeder Landessprache basierend auf dem Unicode Zeichensatz beinhalten. Es ist möglich eine Methode mehrmals mit demselben Namen zu definieren, solange jede Methode eine andere Parameterliste hat.

[3] In Java ist es derzeit nicht möglich, Methoden mit einer variablen Anzahl an Argumenten zu definieren. In der kommenden Java-Version 1.5 wird sich das möglicherweise ändern. Derzeit kann in einfachen Fällen durch Überladen von Methoden oder durch die Verwendung des Argumentes Object[] diese Eigenschaft simuliert werden.

- *Parameterliste (paramlist)* einer Methode muss in eingeschlossenen Klammern definiert werden. Die Parameterliste kann beliebig viele Argumente beinhalten, die an die Methode übergeben werden. Wenn eine Parameterliste spezifiziert wird, so hat jedes Argument einen Typ und einen Namen und ist durch ein Komma vom nächsten Argument getrennt. Die Werte, die an diese Methode übergeben werden, müssen nicht den exakt gleichen Typ aufweisen wie in der Signatur angegeben wurde. Jedoch müssen diese zu diesen Typen konvertierbar (ohne casting) sein.

- *Die auftretende Ausnahmebedingung (throws exception)* ist der Abschluss einer Methodensignatur. Wenn eine Methode eine *throw* Anweisung zum Werfen einer geprüften Ausnahmebedingung einsetzt, oder Methoden aufruft, die eine geprüfte Ausnahmebedingung werfen, und diese unbehandelt lässt, so muss diese Methode explizit deklarieren, dass diese Ausnahmebedingungen auftreten können. Sofern bei einer Methode eine oder mehrere geprüfte Ausnahmebedingungen auftreten können, sind diese nach dem Schlüsselwort *throws*, welches im Anschluss an die Parameterliste folgt, gefolgt von dem Namen der Klasse, welche mögliche Ausnahmebedingungen darstellt, anzugeben. Wenn eine Methode mehrere Typen von Ausnahmebedingungen wirft, so sind die Namen der Ausnahmeklassen durch ein Komma voneinander zu trennen.

Finalisieren von Methoden

Eine Methode kann als *final* deklariert werden, damit verhindert wird, dass Subklassen diese Methode überschreiben oder verdecken. Falls dies versucht wird, meldet der Compiler einen Fehler. Private Methoden und alle Methoden einer finalisierten Klasse sind implizit finalisiert, weil es unmöglich ist, diese zu überschreiben. Es ist erlaubt, aber nicht notwendig, bei der Deklaration solcher Methoden trotzdem das Schlüsselwort *final* zu verwenden. Ein Compilezeit-Fehler tritt auch dann auf, wenn versucht wird, eine finalisierte Methode als *abstract* zu deklarieren.

Terminologie: Pass by Value

Es wurde bereits erwähnt, dass in Java Objekte und Arrays als Referenzobjekte behandelt werden. Das hat jedoch nichts mit der Phrase „pass by reference“ zu tun. „Pass by reference“ ist ein Term, der die Konvention von Methodenaufrufen in Programmiersprachen beschreibt. In einer pass-by-reference Sprache werden Werte nicht direkt an die Methoden übergeben, sondern es werden Referenzen auf die Werte geliefert. Wenn die Methode die Werte der Parameter verändert, sind diese Modifikationen auch nach der Rückkehr sichtbar.
In Java passiert das nicht, denn Java ist eine „pass by value“ Sprache.

```
public void badswap(Point arg1, Point arg2)
{
  arg1.x = 100;
  arg1.y = 100;

  Point tmp = arg1;
  arg1 = arg2;
  arg2 = tmp;
}
```

Sobald die Methode badswap beendet ist, beinhalten die als Argumente übergebenen Variablen nach wie vor ihren ursprünglichen Wert. Die Methode funktioniert auch nicht, wenn statt dem primitiven Datentyp int eine Instanz von Object zum Einsatz gebracht wird, da Object-Referenzen ebenfalls als „pass by value“ übergeben werden. Zur Veranschaulichung dieses Problems ein verzwicktes Beispiel:

```
public static void main(String [] args)
{
  Point pnt1 = new Point(0,0);
  Point pnt2 = new Point(0,0);
  System.out.println("X: " + pnt1.x + " Y: " + pnt1.y);  // X: 0 Y: 0
  System.out.println("X: " + pnt2.x + " Y: " + pnt2.y);  // X: 0 Y: 0
  System.out.println(" ");
  badswap(pnt1,pnt2);
  System.out.println("X: " + pnt1.x + " Y:" + pnt1.y);   // X: 100 Y: 100
  System.out.println("X: " + pnt2.x + " Y: " + pnt2.y);  // X: 0 Y: 0
}
```

Die Methode main() verändert zwar erfolgreich die Werte von pnt1, obwohl diese ebenso „pass by value“ übergeben wurden, doch ein Austausch von pnt2 schlug fehl. Dies ist die hauptsächliche Quelle für die Verwirrung. In der main() Methode sind pnt1 und pnt2 auch nur einfache Objekt-Referenzierungen. Wenn nun pnt1 und pnt2 an die Methode badswap() übergeben wird, gibt Java die Referenzen der Werte („pass by value“) wie bei jedem herkömmlichen Parameter weiter. Das bedeutet, dass die mitgegebenen Referenzen an die Methode aktuelle Kopien der Originalreferenzen sind.

In der Abbildung 3.1 zeigen zwei Referenzen auf das gleiche Objekt nachdem Java ein Objekt an die Methode übergeben hat. Java kopiert und übergibt die Referenzen als Wer-

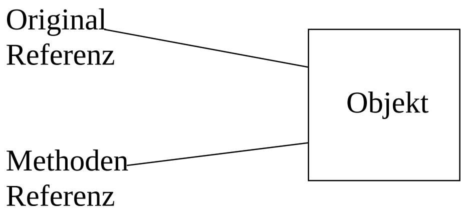


Abbildung 3.1: Ein Objekt mit zwei Referenzen

te und nicht das Objekt. Daher wird durch die Manipulation in der Methode das Objekt, welches auf das Originalobjekt referenziert, verändert. Da aber die Referenzen nur Kopien sind, schlägt badswap fehl.

In Abbildung 3.2 wird illustriert wie die Methodenreferenzen vertauscht werden, aber die Originalreferenzen unverändert bleiben. Nach dem Methodenaufruf hat man nach wie vor die unvertauschten Originalreferenzen.

3.3.9 Operatoren

Um komplexere Ausdrücke zu bilden, werden einfache Ausdrücke mit Hilfe von Operatoren verknüpft. Im Ausdruck `e = 2.56;` wird mit Hilfe des Zuweisungsoperators `=` der

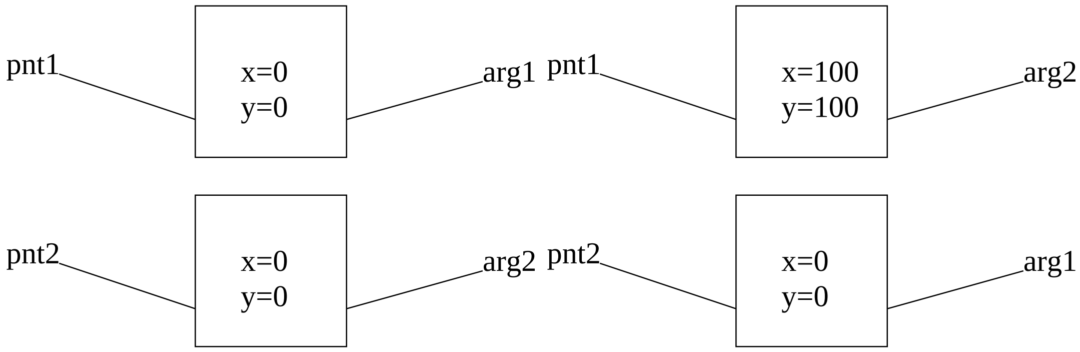


Abbildung 3.2: Links vor dem Methodenaufruf, rechts danach

Variablen e der einfache Ausdruck des Gleitkommaliterals zugewiesen. Operatoren können neben den einfachen Ausdrücken auch Ausdrücke mit jeder Art von Komplexität verwenden. Die Prioritätsstufe der Operatoren wurde aus Kompatibilitätsgründen von C übernommen. Die Entwickler von C wählten die Prioritäten so, dass die meisten Ausdrücke in einer natürlichen Form ohne Klammerung möglich sind. Einen Fall, in dem eine Klammerung notwendig ist, zeigt folgendes Beispiel:

```
// Zuweisung kombiniert mit einem Vergleich
while((line = in.readLine()) != null) { ... }
```

Wenn Operatoren die gleiche Prioritätsstufe haben, wird anhand der Assoziativitätseigenschaft die Reihenfolge der Operationen bestimmt. Die meisten Operatoren sind links-rechts assoziativ. Die unären Operatoren, der Operator *new* und der cast Operator sind rechts-links assoziativ. Durch die Prioritäts- und Assoziativitätsvorgabe ist die Standardreihenfolge der Ausführung von Ausdrücken bestimmt. Diese Standardreihenfolge kann durch Setzen von Klammern verändert werden, was in Java jedoch durch die natürliche Form der Syntax bei den Ausdrücken selten notwendig ist. In Java stehen den Programmierern

- Arithmetische Operatoren,
- Operatoren für Zeichenverkettungen,
- Inkrement und Dekrement Operatoren,
- Vergleichsoperatoren,
- Boolesche Operatoren,
- Bitweise und Schiebe Operatoren und
- Zustandsoperatoren

zur Verfügung. Daneben gibt es noch den *instanceof*-Operator und ein paar weitere spezielle Operatoren in Java.

Der instanceof Operator

Der *instanceof* Operator benötigt ein Objekt als linken Operanden (keinen primitiven Datentyp) und den Namen eines Referenztyps als rechten Operanden. Dieser evaluiert zu true, wenn das Objekt oder Array eine Instanz von dem spezifizierten Typ ist, andernfalls retourniert er false. Wenn der *instanceof* Ausdruck zu true evaluiert, bedeutet dies, dass man ein sicheres cast und somit eine Zuweisung des linken Operanden auf eine Variable mit dem Typ des rechten Operanden durchführen kann.

```
"string" instanceof String // true: alle strings sind Instanzen von String
"" instanceof Object       // true: strings sind auch Instanzen von Object
null instanceof String     // false: null is keine Instanz von irgendetwas

// Die Verwendung von instanceof um sicher zu gehen,
// dass ein Objekt mittels cast zugewiesen werden kann.
if (object instanceof Point) {
  Point p = (Point) object;
}
```

Spezielle Operatoren

Es gibt fünf Sprachkonstrukte in Java, die manchmal als Operatoren und manchmal als einfacher Bestandteil der Sprachsyntax angesehen werden.

- *Zugriff auf ein Objektmitglied (.)*: Ein Objekt ist eine Kollektion von Daten und Methoden, die auf Daten operieren. Die Datenfelder und Methoden eines Objektes werden Mitglieder (member) genannt. Mit dem Punkt wird auf die Mitglieder zugegriffen. Wenn o ein Ausdruck ist, der auf eine Objektreferenz evaluiert und f der Name eines Feldes von diesem Objekt ist, so evaluiert o.f zum Wert dieses Feldes. Wenn m der Name einer Methode ist, kann diese mit o.m() aufgerufen werden.

- *Methodenaufruf ()*: Eine Methode ist eine mit einem Namen bezeichnete Ansammlung von Java-Code, die mit dem Namen und mit beliebig vielen durch Komma getrennten Ausdrücken und mit Klammern umgeben ist. Die Werte der Ausdrücke stellen die Argumente der Methode dar. Die Methode bearbeitet die Argumente und gibt optional einen Wert zurück. Wenn die Methode zum Beispiel drei Argumente erwartet, kann diese mit einem Ausdruck wie o.m(x,y,z) aufgerufen werden. Bevor der Java Interpreter eine Methode aufruft, werden alle an diese Methode übergebenen Argumente evaluiert. Dabei ist garantiert, dass die Ausdrücke in der Reihenfolge von links nach rechts, was im Falle von Seiteneffekten von Bedeutung ist, evaluiert werden.

- *Objekterstellung (new)*: In Java werden Objekte und Arrays mit dem *new* Operator, gefolgt vom Typ des zu erstellenden Objektes und einer Liste von Argumenten in Klammern an den Konstruktor des Objektes übergeben. Ein Konstruktor ist eine spezielle Methode, die ein neu erstelltes Objekt initialisiert. Die Syntax beim Erstellen eines Objektes ist der eines Methodenaufrufs ähnlich: new o().

- *Zugriff auf ein Array Element* `[ ]`: Ein Array ist eine nummerierte Liste von Werten. Jedes Element eines Arrays kann mit seiner Nummer (Index) referenziert werden. Der `[ ]` Operator ermöglicht, sich auf das individuelle Element eines Array zu beziehen. Wenn a ein Array und i ein Ausdruck ist, der zu einem int evaluiert, so referenziert a[i] ein Element von a. Anders als andere Operatoren, die mit ganzzahligen Werten arbeiten, ist dieser Operator für Werte von Indizes auf Arrays auf den Typ int oder enger beschränkt.

- *Typ Konvertierungen und casting ()*: Dieser Operator ermöglicht engere Typkonvertierungen oder casts. Der erste Operand von diesem Operator ist der Typ auf den konvertiert werden soll. Er wird mit Klammern umgeben. Der zweite Operand, der den Klammern folgt, ist der Wert, welcher konvertiert werden soll.

```
(byte) 28        // Integer Literal wird zum byte-Typ gecastet.
(int) (x+3.14f)  // Gleitkommazahl wird zum Integer-Wert gecastet.
(String)h.get(k) // Objekt wird zum spezielleren Typ String gecastet.
```

3.3.10 Zusicherungen (asserts)

Die assert-Anweisung wird verwendet, um Entwurfsannahmen zu dokumentieren bzw. im Java-Code zu verifizieren. Es ist auch möglich, die assert-Anweisung als Werkzeug für das Debugging und Testen einzusetzen. Da es die assert-Anweisung erst seit der Java Version 1.4 gibt und *assert* davor kein reserviertes Schlüsselwort war, muss die Aktivierung dieser neuen Anweisung beim Compilieren eines Java Programms durch die Mitgabe des Argumentes -source 1.4 in der Befehlszeile durchgeführt werden.

Solange dieses Argument nicht spezifiziert ist, erlaubt der Java Compiler *assert* als Identifikator zu benutzen. Laut [Flanagan, 2002] wird diese Option in der Befehlszeile bei zukünftigen Java Versionen nicht mehr notwendig sein, da die assert-Anweisung standardmäßig erkannt wird. Das setzt voraus, dass Code, welcher *assert* bisher als Identifikator eingesetzt hat, nicht mehr benötigt bzw. korrigiert wird.
Eine Behauptung besteht aus dem Schlüsselwort *assert* gefolgt von einem booleschen Ausdruck, von dem der Programmierer annimmt, dass er immer zu true evaluiert. Wenn der Ausdruck einer Behauptung den booleschen Wert true liefert, macht die assert-Anweisung nichts. Wenn der Ausdruck den Wahrheitswert false evaluiert, so ging die Behauptung schief und die assert-Anweisung wirft einen java.lang.AssertionError. Die assert-Anweisung kann auch noch einen zweiten optionalen Ausdruck, vom ersten durch einen Doppelpunkt getrennt, besitzen. Wenn die Behauptungen aktiviert sind und der erste Ausdruck zu false evaluiert, wird der Wert des zweiten Ausdrucks als Fehlercode oder Fehlernachricht an den Konstruktor von AssertionError() geliefert. Die volle Syntax diese Anweisung lautet:

```
assert assertion;
assert assertion : errorcode;
```

Wie bereits erwähnt, sind Behauptungen standardmäßig deaktiviert. Wenn der Compiler mit dem -source 1.4 nun die assert-Anweisung als solche erkennt, kann *assert* nun mit dem Argument -ea für alle Klassen ausser den Systemklassen aktiviert werden. Wenn nach dem Argument -ea ein Doppelpunkt, gefolgt von einer Klasse angegeben wird, so werden Behauptungen nur für diese spezifische Klasse aktiviert. Um Behauptungen für Systemklassen zu aktivieren, ist das Argument -esa zu verwenden. Um Behauptungen für alle Klassen in einem Paket und dessen Subpakete zu aktivieren, ist das Argument -ea gefolgt von einem Doppelpunkt und dem Paketnamen und drei Punkten anzugeben. Es können auch Behauptungen durch das Argument -da deaktiviert werden. Schliesslich gibt es auch noch die Möglichkeit der Langform dieser Argumente: -enableassertions für -ea, -disableassertions für -da und -enablesystemassertions für -esa.

```
java -ea:com.example.sorters.MergeSort com.example.sorters.Test
```

```
java -ea:com.example.sorters... com.example.sorters.Test

java -ea:com.example.sorters... -da:com.example.sorters.QuickSort
                                        com.example.sorters.Test
```

Zusicherungen können auch im Programm aktiviert oder deaktiviert werden. Dazu gibt es folgende vier Methoden in der Klasse java.lang.ClassLoader:

- *void clearAssertionStatus()*
  Setzt den standardmäßigen Zusicherungsstatus für diesen Klassenlader auf false und verwirft alle Paket- oder Klasseneinstellungen betreffend Zusicherungen, die mit diesem Klassenlader verknüpft sind.

- *void setClassAssertionStatus(String className, boolean enabled)*
  Setzt den gewünschten Status für die angegebene oberste Klasse und die darin geschachtelten Klassen. Diese Zusicherungseinstellung hat Vorrang gegenüber der Standardeinstellung des Klassenladers und einer eventuellen Paketeinstellung. Zu beachten ist, dass diese Methode keine Auswirkungen hat, wenn die angegebene Klasse bereits initialisiert wurde. Der Zusicherungsstatus einer bereits initialisierten Klasse kann nicht mehr geändert werden.

- *void setDefaultAssertionStatus(boolean enabled)*
  Setzt die Standardeinstellung für Zusicherungen von Klassen, die mit diesem Klassenlader zukünftig initialisiert werden. Diese Einstellung kann auf Paketbasis durch die Methode setPackageAssertionStatus() und auf Klassenbasis durch die Methode setClassAssertionStatus() überschrieben werden.

- *void setPackageAssertionStatus(String packageName, boolean enabled)*
  Setzt den gewünschten Status für noch nicht initialisierte Klassen, die sich im angegebenen Paket und dessen Unterpakete befinden. Falls mehrere Paketeinstellungen auf eine Klasse zutreffen, so zieht die mit der genauesten Paketangabe. Zusicherungen auf Paketbasis haben Vorrang gegenüber Standardzusicherung auf Klassenladerbasis, können aber auf Klassenbasis durch die Methode setClassAssertionStatus() überschrieben werden.

3.3.11 Ausnahmebehandlungen (exceptions)

Wenn ein Programm die semantischen Beschränkungen der Programmiersprache Java verletzt, signalisiert die virtuelle Maschine von Java (JVM) diesen Fehler an das Programm als Ausnahme. Ein Beispiel einer solchen Verletzung ist der Versuch, mit einem Index außerhalb der Grenzen eines Arrays zuzugreifen. Einige Programmiersprachen reagieren darauf mit der vorzeitigen Programmbeendigung, andere weisen dann ein willkürliches oder undefiniertes Verhalten auf. Keine dieser Varianten ist mit den Designzielen der Java Plattform kompatibel: Portabilität und Robustheit zu gewähren. Stattdessen spezifiziert die Programmiersprache Java, dass eine Ausnahme geworfen werden soll, wenn eine semantische Beschränkung verletzt wird. Dadurch wird ein nicht ortsabhängiger Transfer der Kontrolle von einem Punkt, wo die Ausnahme auftritt, zu einem anderen Punkt, welcher vom Programmierer spezifiziert werden kann, bewirkt. Eine Ausnahme wird am Punkt des Auftretens als geworfen (thrown) und am Punkt, zu dem die Kontrolle transferiert wurde, als aufgefangen (caught) bezeichnet. Programme können durch Verwendung der throw-Anweisung Ausnahmebedingungen explizit werfen.

throw

Eine Ausnahme zu werfen bedeutet einen Ausnahmezustand zu signalisieren. In Java wird mit der throw-Anweisung eine Ausnahme geworfen:

```
throw expression;
```

Der Ausdruck (expression) muss zu einem Ausnahmeobjekt, das die aufgetretene Ausnahme oder den aufgetretenen Fehler beschreibt, evaluiert werden können.

```
public static double factorial(int x) {
  if (x < 0)
    throw new IllegalArgumentException("x muss >= 0");
  ...
}
```

Wenn der Java-Interpreter eine throw-Anweisung ausführt, wird der weitere Ablauf des Programms gestoppt und nach einem Ausnahmebehandler, der diese Ausnahme abfängt (catch) und behandelt, gesucht. Der Java Interpreter sucht dabei zuerst im eingeschlossenen Codeblock nach einem assoziierten Ausnahmebehandler. Wenn ein solcher existiert, wird der aktuelle Codeblock verlassen und dieser Ausnahmebearbeiter ausgeführt. Nach dessen Ausführung setzt der Java-Interpreter bei der Anweisung, das dem Ausnahmecode folgt, fort.
Wenn der einschließende Codeblock keinen adäquaten Ausnahmebehandler hat, überprüft der Java-Interpreter den nächsthöheren einschließenden Codeblock der aktuellen Methode. Dies wird solange fortgesetzt bis ein Ausnahmebehandler gefunden wurde. Hat diese Methode keinen Ausnahmebehandler, der die von der throw Anweisung geworfene Ausnahme abfängt, so stoppt der Java-Interpreter diese Methode und kehrt zum Aufrufer der Methode zurück.
Nun sucht der Java Interpreter in diesem Codeblock der aufrufenden Methode nach einem passenden Ausnahmebehandler.

Auf diese Art und Weise wird die Ausnahme entlang der dynamischen Struktur der Java Methoden den Stack der Aufrufer hinauf propagiert. Jede Ausnahme, die bis nach oben zur main() Methode propagiert und dabei nicht aufgefangen wurde, wird nach Ausgabe einer Fehlermeldung und eines stack trace vom Java-Interpreter vorzeitig beendet. Der stack trace ist ein Auszug der Threads und Methoden in der JVM.

Typen von Ausnahmen

Eine Ausnahme ist in Java ein Objekt. Der Typ des Objektes ist java.lang.Throwable oder eine Subklasse davon. Zwei Standard Subklassen von Throwable sind

- *java.lang.Error*: Im Allgemeinen sind Ausnahmen, die Subklassen von Error sind Fehler, die nicht behebbar sind, wie zum Beispiel „kein Speicher mehr in der JVM“, „Klassendatei ist fehlerhaft und kann nicht gelesen werden“. Ausnahmen dieser Art können zwar aufgefangen und behandelt werden, dies ist aber selten der Fall.

- *java.lang.Execption*: Ausnahmen, die Subklassen von Exception sind, können besser gehandhabt werden, als die von Error. Diese Ausnahmen sollten vernünftigerweise aufgefangen und behandelt werden. Beispiele für diese Art von Ausnahmen sind

java.io.EOFException (signalisiert das Ende einer Datei) und
java.lang.ArrayIndexOutOfBoundsException (signalisiert, dass ein Programm versucht hat, über das Ende des Arrays zuzugreifen).

Da eine Ausnahme ein Objekt ist, kann es Daten beinhalten und ihre Klasse kann Methoden definieren um auf diese Daten zu operieren. Die *Throwable* Klasse und alle ihre Subklassen haben ein *String* Feld, das die lesbare Fehlermeldung des Ausnahmezustandes beschreibt. Dessen Inhalt wird zum Zeitpunkt der Erstellung des Ausnahmeobjektes gesetzt und kann dann in der entsprechenden Ausnahme mit der Methode getMessage() ausgelesen werden.

Deklarieren von Ausnahmebedingungen

Neben der Unterscheidung zwischen Error- und Exception-Klassen macht das Java-Ausnahmebehandlungsschema auch eine Unterscheidung zwischen geprüften und ungeprüften Ausnahmen.

Jedes Ausnahmeobjekt von Error ist ungeprüft und jedes Ausnahmeobjekt von Exception ist geprüft, wobei der Sonderfall java.lang.RuntimeException und dessen Subklassen ungeprüft sind. Der Grund dieser Unterscheidung ist, dass prinzipiell jede Methode zu jeder Zeit eine ungeprüfte Ausnahme werfen kann. Es kann zum Beispiel nicht vorhergesagt werden, wann ein OutOfMemoryError auftritt. Weiters kann jede Methode, die Objekte oder Arrays benutzt, eine NullPointerException werfen, wenn ihr ein ungültiges null Argument geliefert wurde. Geprüfte Ausnahmen treten andererseits nur unter spezifischen, wohldefinierten Umständen auf. Wenn zum Beispiel Daten von einer Datei gelesen werden, muss zumindest die Möglichkeit betrachtet werden, dass eine FileNotFoundException geworfen werden kann, wenn die entsprechende Datei nicht gefunden wird.

In Java gibt es verschieden Regeln für geprüfte und ungeprüfte Ausnahmen. Falls eine Methode geschrieben wurde, die geprüfte Ausnahmen wirft, muss diese mit der throws-Klausel in der Methodensignatur deklariert werden. Der Grund, warum diese Ausnahmen in Java als geprüfte Ausnahmen bezeichnet werden, ist der, dass der Java Compiler überprüft, ob diese Ausnahme in der Methodensignatur deklariert wurde. Falls nicht wird ein Fehler vom Compiler gemeldet. Auch wenn nie eine Ausnahme mittels throw geworfen wird, kann es manchmal notwendig sein, eine throws-Klausel zum Deklarieren einer Ausnahme zu verwenden. Wenn eine Methode eine weitere Methode aufruft, die geprüfte Ausnahmen wirft, so muss entweder ein Code zur Ausnahmebehandlung dieser Ausnahme erstellt werden oder eine throws Deklaration spezifiziert werden, wodurch angezeigt wird, dass diese Methode diese Ausnahme ebenfalls werfen kann.

Anweisung try/catch/finally

Die try/catch/finally-Anweisung ist Javas Mechanismus für die Ausnahmebehandlung. Die try-Klausel etabliert einen Codeblock für die Ausnahmebehandlung. Dem try-Block folgen beliebig viele catch-Klauseln. Jede dieser catch-Klauseln stellt einen Block für die Behandlung eines bestimmten Ausnahmetyps dar. Der catch-Klausel kann optional ein finally-Block folgen, der abschließende Codezeilen beinhaltet, die unabhängig davon, was im try-Block passierte, auf jeden Fall durchgeführt werden. Zwar sind sowohl der catch- als auch der finally-Block optional, jedoch muss jedem try-Block zumindest ein catch- oder ein finally-Block folgen. Die try/catch/finally-Blöcke beginnen und enden mit geschwungenen Klammern.

```
try { ... }
catch {EineAusnahme e1) { ... }
catch (AndereAusnahme e2) { ... }
finally { ... }
```

- Die try-Klausel eröffnet einen Codeblock, dessen Ausnahmebedingungen abgefangen werden sollen oder der im Falle einer (vorzeitigen) Beendigung noch spezielle Aufräumungsarbeiten durchführt.

- Die catch-Klausel behandelt verschiedene Typen von Ausnahmebedingungen. Jede catch-Klausel ist mit einem Argument deklariert, das aus dem Typ der zu behandelnden Ausnahme, gefolgt von einem Namen, mit dem die Klausel sich auf das Ausnahmeobjekt beziehen kann, besteht. Der Typ und Name einer Ausnahmebehandlung in einer catch-Klausel entspricht dem Typ und Namen, der als Argument an eine Methode übergeben wird. Die einzige Bedingung ist, dass der Argumenttyp der catch-Klausel Throwable oder eine Subklasse von Throwable sein muss. Wenn eine Ausnahme geworfen wird, sucht der Java-Interpreter nach der ersten catch-Klausel, dessen Argument der gleiche Typ wie der des Ausnahmeobjektes – oder eine Superklasse davon – ist. Der Code in einer catch-Klausel sollte den aufgetretenen Ausnahmezustand bewältigen können, welche Aktionen dafür auch immer notwendig sind. Wenn zum Beispiel die Ausnahme java.io.FileNotFoundException auftritt, kann diese Ausnahme durch eine Aufforderung an den Benutzer, die Eingabe zu überprüfen und es nochmals zu versuchen, behandelt werden. Es ist nicht notwendig, für jede nur mögliche Ausnahme eine eigene catch-Klausel zu haben. In einigen Fällen ist es richtiger, dass die Ausnahme nach oben propagiert wird und von der aufgerufenen Methode abgefangen wird. In anderen Fällen ist es wiederum sinnvoller, zum Beispiel bei einem Programmierfehler bei dem eine NullPointerException auftritt, diese Ausnahme nicht aufzufangen und den Java Interpreter die Ausführung des Programms mit einem stack trace und einer Fehlermeldung vorzeitig beenden zu lassen.

- Die finally-Klausel wird allgemein für das saubere Beenden von Ressourcen, zum Beispiel Schliessen von Dateien, Beenden von Netzwerkverbindungen, die im Code der try-Klausel initiiert wurden, verwendet. Das besondere der finally-Klausel ist, dass sie auf jeden Fall durchgeführt wird, egal ob der Code in der try-Klausel erfolgreich beendet wurde oder nicht. Die einzige Möglichkeit, dass eine try-Klausel beendet wird ohne dass die anschließende finally-Klausel auch durchgeführt wird, ist durch Aufruf der System.exit() Methode, die den Java-Interpreter veranlasst, das Programm unmittelbar zu stoppen. Wenn eine Ausnahme im try-Block auftritt und es einen passenden catch-Block für die Behandlung dieser Ausnahme gibt, wird die Kontrolle zuerst an den catch-Block und dann an den finally-Block übergeben. Wenn es lokal keinen entsprechenden catch-Block gibt, wird die Kontrolle zuerst an den finally-Block übergeben und dann an den nächsten oberen catch-Block, der diese Ausnahme behandeln kann, propagiert. Wenn ein finally-Block selber die Kontrolle durch eine return-, continue-, break- oder throw-Anweisung oder durch den Aufruf einer Methode, die eine Ausnahme wirft, abgibt, wird die derzeitige Kontrolle herrenlos und die neue Stelle übernimmt die Kontrolle. Wenn zum Beispiel eine finally-Klausel eine Ausnahme wirft, ersetzt diese Ausnahme die derzeit behandelte Ausnahme. Falls eine finally-Klausel eine return-Anweisung verwendet, endet die Methode ganz normal, als ob eine Ausnahme geworfen und nicht abgefangen worden wäre.

3.3.12 Variablen

Eine Variable ist die Bezeichnung für einen Speicherplatz und hat einen Typ. Dieser wird manchmal compile-time Typ genannt. Dieser compile-time Typ ist entweder ein primitiver Datentyp oder ein Referenzdatentyp. Eine Variable beinhaltet einen Wert, der zuweisungskompatibel mit seinem Typ ist und durch einen Zuweisungs-, prefix oder postfix Inkrement `++` oder Dekrement `--` Operator ihren Wert verändert.

Eine Variable von einem primitiven Datentyp hat immer einen Wert vom exakt gleichen primitiven Datentyp. Die Variable eines Referenzdatentyps kann entweder eine null-Referenz oder eine Referenz auf ein beliebiges Objekt, welches zuweisungskompatibel mit dem Typ der Variable ist, als Wert besitzen.

Java hat einen sehr sauberen Namensraum. Pakete beinhalten Klassen, Klassen haben Felder und Methoden, und Methoden haben lokale Variablen. Es gibt aber keine globalen Variablen in Java und somit ist es nicht möglich, dass es im Namensraum zu Kollisionen dieser Variablen kommt.

Es gibt sieben Arten von Variablen:

1. *Die Klassenvariable* ist ein Feld, welches als static innerhalb einer Klasse deklariert wurde. Eine Klassenvariable kann mit dem Schlüsselwort *static* – aber auch ohne dieses – in einem Interface deklariert werden. Die Klassenvariable wird mit ihrem Standardwert erstellt, sobald ihre Klasse oder Interface angelegt ist. Sie ist nicht mehr verwendbar, nachdem die Klasse oder das Interface entladen wurde.

2. *Die Instanzvariable* ist ein Feld, welches innerhalb einer Klasse ohne das Schlüsselwort static deklariert wurde. Wenn eine Klasse T ein Feld a als Instanzvariable besitzt, so wird mit jedem neuen Objekt dieser Klasse T auch eine neue Instanzvariable a angelegt und mit einem Standardwert initialisiert. Die Instanzvariable hört auf zu existieren, sobald das dazugehörende Objekt nicht mehr länger von einem Feld referenziert wird und alle notwendigen Beendigungen von dem Objekt abgeschlossen sind.

3. *Die Array-Komponenten* sind unbenannte Variablen, die, wann immer ein Objekt als Array erstellt wird, angelegt und mit Standardwerten initialisiert werden. Die Array-Komponenten hören auf zu existieren, sobald das Array nicht mehr referenziert wird.

4. *Die Methodenparameter* sind Werte von Argumenten, die an eine Methode mitgegeben werden. Für jeden deklarierten Parameter in einer deklarierten Methode wird bei jedem Methodenaufruf eine neue Parametervariable erstellt. Die neue Variable wird mit dem korrespondierenden Argumentwert vom Methodenaufruf initialisiert. Die Methodenparameter hören auf zu existieren, sobald der Block der Methode vollständig ausgeführt wurde.

5. *Die Konstruktorparameter* sind Werte von Argumenten, welche an einen Konstruktor übergeben werden. Für jeden deklarierten Parameter in einer Konstruktor-Deklaration wird eine neue Variable erstellt, wenn eine Klasse instanziert wird und somit den Konstruktor aufruft, oder der Konstruktor explizit aufgerufen wird. Die neue Variable wird mit den korrespondierenden Argumentwerten des Ausdrucks bei der Erstellung oder mit den Argumentwerten des Konstruktoraufrufs initialisiert. Die Konstruktorparameter hören auf zu existieren, sobald der Block vom Konstruktor beendet ist.

6. *Die Ausnahmebehandlungsparameter* werden dann erstellt, wenn eine Ausnahmebedingung durch eine catch-Klausel von einer try-Anweisung aufgefangen wird. Die neue Variable wird initialisiert mit dem aktuellen Objekt, welches aus den Ausnahmebedingungen hervorgegangen ist. Der Ausnahmebehandlungsparameter hört auf zu existieren, sobald die Ausführung des assoziierten Blocks der catch-Klausel beendet ist.

7. *Die Lokalen Variablen* sind in einer „lokalen Variablendeklarationsanweisung" deklariert. Jede lokale Variablendeklarationsanweisung ist in einem Block enthalten und kann beliebig mit anderen Arten von Anweisungen vermischt werden. Eine lokale Variablendeklaration kann auch im Kopf einer for-Anweisung aufscheinen. Wann immer die Flusskontrolle nun auf einen Block oder auf eine for-Anweisung stößt, wird für jede deklarierte lokale Variable eine neue Variable in diesem Block oder zu dieser for-Anweisung erstellt. Eine lokale Variablendeklarationsanweisung kann auch einen Ausdruck enthalten, mit dem die Variable initialisiert wird. Die lokale Variable mit einem Initialisierungsausdruck ist solange nicht initialisiert, bis die Variablendeklarationsanweisung, welches diese lokale Variable initialisiert, ausgeführt wurde. Die lokale Variable hört auf zu existieren, sobald die Ausführung des Blockes oder der for-Anweisung beendet ist. Eine Ausnahme stellt die switch-Anweisung dar, bei der es möglich ist, dass ein Block zwar die Kontrolle erhält, aber die Ausführung der lokalen Variablendeklarationsanweisung umgangen wird.

final-Variable

Eine Variable kann als endgültig (final) deklariert werden. Einer final-Variablen wird nur einmal ein Wert zugewiesen, diesen behält sie dann für immer bei. Falls ihr ein weiterer Wert zugewiesen wird, meldet der Compiler einen entsprechenden Fehler. Wenn eine final-Variable eine Referenz auf ein Objekt hat, so kann der Zustand des Objektes nach wie vor verändert werden, aber die Variable zeigt immer auf dasselbe Objekt. Durch die Deklaration einer Variable als endgültig wird in einem Programm implizit dokumentiert, dass der Wert nicht veränderbar ist, wodurch Programmierfehler vermieden werden.

Kapitel 4

Die Programmiersprache Ruby

4.1 Was ist Ruby?

Ruby ist eine frei verfügbare objektorientierte Programmiersprache, die mit der Zielsetzung entwickelt wurde, das Programmieren sowohl angenehm als auch schnell zu ermöglichen. Mit seinem einfach verwendbaren Interpreter, seiner einfach zu verstehenden Syntax, einer komplett objektorientierten Funktionalität und einer mächtigen Klassenbibliothek wurde Ruby zu einer Sprache, die ein breites Einsatzspektrum abdeckt. Ruby kann von einfachen Textbearbeitungen, CGI-Skripten bis hin zu großangelegten Programmen eingesetzt werden.

Der Erfinder dieser Sprache ist der Japaner Yukihiro Matsumoto, kurz Matz genannt. Beeinflusst von der Programmiersprache Perl wollte Matz seiner neuen Sprache eine Bezeichnung eines Juwels geben, und entschied sich für den Geburtsstein eines Kollegen. Später stellte sich heraus, dass er mit Ruby in mehreren Belangen einen Nachfolger von Perl gefunden hat. Bei den Geburtssteinen ist Pearl im Juni, Ruby im Juli. Bei der Messung von Schriftgrößen ist Pearl 5pt und Ruby 5.5pt. Somit fühlte sich Matz bestätigt, dass er einen guten Namen für eine Programmiersprache gefunden hat, die neuer als Perl ist.

Ruby wurde von Beginn an dafür entworfen, objektorientierte Programmierung zu unterstützen. In Ruby kann auch „prozedural" programmiert werden. Insgeheim wandelt Ruby diese Prozeduren aber in Methoden um und stellt sie in Form eines global zugänglichen Objekts zur Verfügung. In Ruby ist alles ein Objekt, es gibt keine Ausnahmen [Matz, 2002].

Matz legte großen Wert darauf, Programme schneller und einfacher zu machen. Um dies zu realisieren, entwickelt er Ruby nach dem Prinzip der wenigsten Überraschungen. Alle Eigenschaften in Ruby, auch die objektorientierten, sind so entworfen, dass die Sprache sich so verhält, wie es gewöhnliche Programmierer erwarten würden. Einige dieser Eigenschaften sind:

- *Interpreter-basierende Programmierung:* Programme müssen nicht compiliert werden. Sie können nach dem Bearbeiten sofort dem Interpreter zugewiesen werden. Dieser raschere Entwicklungszyklus erhöht die Freude am Programmierprozess.

- *Dynamische Programmierung:* Beinahe alles in Ruby wird zur Laufzeit erledigt. Typen von Variablen und Ausdrücke werden zur Laufzeit ermittelt, genauso wie Klassen und Methodendefinitionen. Es ist sogar möglich, Programme mit Programmen zu erstellen und auszuführen.

- *Vertraute Syntax:* Ein Programmierer von Java, Perl, Python, C/C++ oder auch Smalltalk findet sich mit der Syntax von Ruby rasch zurecht.

- *Iteratoren:* Die Funktionalität eines Iterators für Schleifenabstraktion ist in der Sprache integriert. Das bedeutet, dass ein Block mit Code an einen Methodenaufruf angehängt werden kann. Zum Beispiel hat ein Array die Methode each, mit der die Elemente iteriert werden können. Der Programmierer braucht sich dadurch keine Gedanken über den Schleifenzähler oder über dessen Bereichsgrenzen zu machen.

- *Ausnahmebehandlungen:* Wie von einer modernen objektorientierten Sprache erwartet wird, bietet Ruby auf Sprachebene Ausnahmebehandlungen (exceptions) an.

- *Klassenbibliotheken:* Ruby wird mit mehreren stabilen Klassenbibliotheken ausgeliefert, die eine Menge an Aufgaben abdecken. Von einem einfachen Datentyp (Zeichenketten, Arrays, Hashes) bis zu Netzwerk- und Multitasking-Programmierung. Da Ruby noch sehr jung ist, steigt die Zahl der verfügbaren Bibliotheken täglich an.

- *Portierbar:* Ruby läuft auf vielen Plattformen ohne Modifikation des Programms, unter anderem Unix und Windows.

- *Garbage collection:* Die objektorientierte Programmierung tendiert dazu, sehr viele Objekte während der Laufzeit anzulegen. Der garbage collector von Ruby bereinigt zyklisch unbenötigte Objekte automatisch.

- *Eingebaute Sicherheitsüberprüfung:* Ruby bietet ein so genanntes taint-Modell an, mit dem nicht vertrauenswürdige Daten und Programme gefahrlos verarbeitet werden können.

4.2 Geschichte

Matz beabsichtigte 1993 eine mächtigere Sprache als Perl und eine objektorientiertere Sprache als Python zu erschaffen. Als im Jahre 1995 die Sprache so weit fortgeschritten war, dass sie Perl ersetzen konnte, gab Matz diese Sprache mit dem Namen Ruby in der Version 0.95 in einer japanisch dominierten NewsGroup für die Öffentlichkeit frei. Im japanischen Raum wurde daraufhin die Sprache immer beliebter und bald populärer als Python. Nachdem die japanische Literatur ins englische übersetzt wurde, gewann Ruby auch in der restlichen Welt Bekanntheit.

Die wichtigsten Versionen und ihr Erscheinungsdatum:

Geburtstag 24. Februar 1993

0.95 Veröffentlichung von Ruby

1.0 Dezember 1996

1.1 August 1997

1.2 Dezember 1998

1.4 August 1999

1.6 September 2000

1.8 August 2003

Seit Dezember 1998 wird zwischen stabilen (gerade Minor-Version) und Entwickler Versionen (ungerade Minor-Version) von Ruby unterschieden. Nachwievor ist die Major-Version 1 gültig und laut Matz gibt es derzeit auch noch keine Ideen und Bestrebungen für die Major-Version 2.

4.3 Sprachkonzepte

Die in diesem Kapitel beschriebenen Sprachkonzepte wurden aus [Thomas et al., 2001], [Matz, 2002] und [Fulton, 2001] entnommen und dienen als Basis für die Untersuchung des objektorientierten Paradigmas im Kapitel 5, Seite 65.

4.3.1 Kommentare

Innerhalb eines Quellcodes können Kommentare der besseren Verständlichkeit dienen. Wenn in einer Zeile das Zeichen `#` auftritt, wird der Rest der Zeile vom Interpreter bei der Ausführung ignoriert. Mehrzeiliger Kommentar kann durch eine Zeile, die mit der Zeichenfolge `=begin` an erster Position startet, bis zur einer Zeile, die mit der Zeichenfolge `=end` an erster Position endet, verfasst werden.

4.3.2 Typen

Die Programmiersprache Ruby ist eine dynamisch typisierte Sprache, die erst zur Laufzeit den entsprechenden Typ ermittelt. Somit kann eine Variable eine Instanz jeden beliebigen Typs referenzieren. In Ruby stehen folgende einfache Typen zur Verfügung: numbers, strings, arrays, hashes, ranges, symbols und regular expressions.

Integer und Gleitkommazahlen

Ein Integer ist entweder ein Objekt der Klasse *Fixnum* oder der Klasse *Bignum*. Objekte von *Fixnum* können Werte in der Größe eines Maschinenwortes abzüglich eines Bits aufnehmen, darüber werden sie automatisch zu einem *Bignum* Objekt umgewandelt. Dieses ist dann nur mehr auf den verfügbaren Speicher limitiert. Wenn wiederum das Ergebnis einer Operation in ein *Fixnum* passt, wird es automatisch in dieses zurückkonvertiert.
Beispiele für Instanzen von *Fixnum* und *Bignum*:

```
1605                    # Fixnum
16_05                   # Fixnum, Unterlinie wird ignoriert
-1605                   # negatives Fixnum
0x1f                    # Fixnum, Hexadezimal
0377                    # Fixnum, Oktal
0b1010                  # Fixnum, Binaer
?A                      # Fixnum, Zeichencode fuer A
1234567890_1234567890   # Bignum, Unterlinie wird ignoriert
```

Eine Besonderheit bezüglich Objekten vom Typ *Fixnum* gilt es in Ruby zu beachten: Es gibt effektiv nur eine Instanz von *Fixnum* für alle Integerzahlen. Dadurch ist es nicht möglich, eine Singletonmethode an *Fixnum* hinzuzufügen, näheres zu Singletonmethoden und Klassen im Abschnitt 4.3.5, Seite 52 und Abschnitt 4.3.7, Seite 55.

Gleitkommazahlen sind Instanzen der Klasse *Float*:

```
16.05                   # Float
1.0e2                   # Float, wissenschaftliche Notation
314159e-5               # Float, Vorzeichen beim Exponenten
```

Strings

Strings stellen in Ruby eine einfache Sequenz von 8-bit Zeichen dar und sind Instanzen der Klasse *String*.

Der Begrenzer ! in einem Ausdruck wie `%q!...!` kann mit einem beliebigen Zeichen dargestellt werden. Wenn der Begrenzer eines der Zeichen `([{<` ist, so ist der zweite Begrenzer das entsprechende schliessende Zeichen. Damit ist es möglich, die Begrenzer auch verschachtelt einzusetzen.

`%!foo!` und `%Q!foo!` sind äquivalent zur doppelten Hochkomma-Zeichenkette `"foo"`,
`%q!foo!` ist äquivalent zur einfachen Hochkomma-Zeichenkette `'foo'` und
`%x!foo!` ist äquivalent mit der Befehlsausgabe \`foo\`.

Zeichenkettenliterale mit einfachen Hochkommas `'foo'`, sowie die Schreibweise `%q/foo/` erlauben die Backslash-Notation, jedoch keine Substitution.

```
'hallo'                 # hallo
'ein Backslash \'\\\''  # ein Backslash '\'
%q/hallo/               # hallo
%q(verschachtelte())    # verschachtelte ()
%q ohne_Leerzeichen     # ohne_Leerzeichen
```

Zeichenkettenliterale mit doppelten Hochkommas `"foo"`, sowie die Schreibweisen `%Q/foo/` und `%/foo/` erlauben neben der Backslash-Notation auch die Substitionen von undruckbaren Zeichen wie `\n` für den Zeilenumbruch, `\nnn` für die Oktale Notation, `\xnn` für die Hexadezimale Notation, `#{Ausdruck}` für den Wert eines Ausdrucks, um nur einige aufzuzählen.

```
"\150allo"              # hallo
"ein Backslash '\\'"    # ein Backslash '\'
%Q$hallo\thi$           # hallo   hi
%Q!"2B or not 2B = FF"! # 2B or not 2B = FF
%Q{8 % 2 = #{8%2}}      # 8 % 2 = 0
%<8 % 2 = #{8%2}>       # 8 % 2 = 0
"8 % 2 = #{8%2}"        # 8 % 2 = 0
```

Zeichenketten können sich auch über mehrere Zeilen erstrecken, sogenannte „here documents“. Wenn Ruby eine Sequenz beginnend mit <<*Begrenzer* findet, werden bis zum

Auffinden des zweiten Begrenzers am Beginn einer Zeile alle Zeichen in ein Zeichenkettenliteral umgewandelt. Der Begrenzer kann ein Identifikationsmerkmal oder eine unter Hochkomma (einfach oder doppelt) gesetzte Zeichenkette sein. Wenn unmittelbar nach den << Zeichen ein Minus (-) steht, sucht der Parser auch nach einem eingerückten Begrenzer.

```
puts <<FOO
     Ein Text
     ueber mehrere
     Zeilen!
FOO

puts <<-"FOO"
     hallo
     FOO
```

Arrays

Ein Array ist eine Container Klasse, in der sich eine Sammlung von Objekten befindet, die mit einem Integer indiziert werden können. Ein Array in Ruby ist heterogen, womit mehrere unterschiedliche Typen im Array abgelegt werden können. Durch Hinzufügen eines Elements wächst das Array automatisch ohne Intervention seitens des Programmierers. Erstellt werden Arrays entweder durch *Array.new* oder mittels Literal:

```
Array.new(3, "a")                  # ["a", "a", "a"]
arr = [x, 16, 3.14159, "foo"]      # Array mit unterschiedlichen Objekten
[]                                 # leeres Array
[1, [2, 3]]                        # verschachteltes Array
```

Ein Array von Zeichenketten kann durch die Kurzform %w gefolgt von mit Leerzeichen voneinander getrennten Zeichenketten einfach erstellt werden. Falls ein Leerzeichen notwendig ist, kann dieses mit einem vorangestellten Backslash versehen werden.

```
%w(foo bar baz)                    # ["foo", "bar", "baz"]
%w(foo bar\ baz)                   # ["foo", "bar baz"]
```

Dadurch, dass ein Array seine Länge selbst aktualisiert, ist es nicht notwendig, eine separate Variable über dessen Länge synchron mitzuführen. Mit Hilfe der Iteratoren wird die Länge eines Arrays für den Programmierer noch unbedeutender. Fehler, die durch inkonsistente Längenaufzeichnungen entstanden sind, werden dadurch ausgeschlossen.

```
a = %w(foo bar baz)
a.each do |s|
  puts s.upcase
end                                # FOO BAR BAZ
```

Hashes

Ein Hash ist eine Sammlung von Schlüssel/Wert-Paarungen in geschwungenen Klammern, die mit der Sequenz => zwischen Schlüssel und Wert getrennt sind. Es gibt keine Notwendigkeit, dass die Schlüssel und/oder Werte in einem Hash den gleichen Typ besitzen.

```
farbe = {"rot"    => 0xf00,
         "gruen"  => 0x0f0,
         "blau"   => 0x00f}
```

Hash Schlüssel unterliegen folgender Restriktion: Sie müssen auf Methodenaufrufe von hash mit einem entsprechenden Wert antworten, wobei der Wert für einen bestimmten Schlüssel konstant bleiben muss. Andernfalls muss mit `Hash#rehash` der Index von hash neu erstellt werden.

Ranges

Ein Range repräsentiert ein Intervall. Der Range kann durch s..e und s...e oder durch Range.new erstellt werden. Das Intervall s..e startet bei s und endet bei e jeweils inklusive s und e ([s,e]). Bei s...e wird der Endwert vom Intervall exkludiert ([s,e[).

```
(1..3).each {|x|
  puts x*x
}                                  # 1 4 9
(1...3).each {|x|
  puts x*x
}                                  # 1 4
```

Symbols

Ein Symbol in Ruby ist die interne Repräsentation eines Namens. Das Symbol für einen Namen wird durch einen vorangestellten Doppelpunkt (:) erstellt. Ein bestimmter Name erstellt immer das gleiche Symbol, gleichgültig welchen Zweck der Name im Programm darstellt. In manchen Programmiersprachen wird dieser Prozess als „interning“ und Symbole als „Atome“ bezeichnet.

```
:foo                              # Symbol fuer foo
:$bar                             # Symbol fuer $bar
```

Regular Expressions

Reguläre Ausdrücke (regular expressions) sind eine Minisprache um Muster von Zeichenketten zu beschreiben. Durch sie werden mächtige Möglichkeiten für den Umgang mit Texten und Daten zur Verfügung gestellt [Friedl, 2002].
Die Literale von regulären Ausdrücken sind in Ruby Objekte vom Typ Regexp. Sie können entweder explizit durch Aufruf des Regexp.new Konstruktors oder durch Verwendung eines Literals in der Form `/pattern/` oder `%r{pattern}` erstellt werden. Literale von regulären Ausdrücken können eine oder mehrere der folgenden Optionen verwenden:

i Die Groß- / Kleinschreibung wird ignoriert.
m Ermöglicht die Suche über mehrere Zeilen. Zeilenvorschübe werden als normales Zeichen behandelt.
o Die Substitution wird nur einmal bei der ersten Evaluierung durchgeführt.
x Aktiviert den erweiterten Modus, wodurch Whitespaces (Leerzeichen, etc.) und Kommentare zur besseren Verständlichkeit hinzugefügt werden dürfen.

```
/pattern/
/pattern/im                  # mit Option im
%r{pattern}                  # allgemein abgegrenzte Eingabe
%r{pattern}im                # allgemein abgegrenzte Eingabe mit Option im
Regexp.new('pattern', im)    # Konstruktor mit Option im
```

4.3.3 Variablen

In Ruby halten Variablen und Konstanten Referenzen auf Objekte[1]. Variablen selbst haben keinen realen Typ. Sie können zu unterschiedlichen Zeiten unterschiedliche Referenzen auf verschiedene Typen von Objekten haben. Der Typ einer Variablen ist ausschließlich von der antwortenden Nachricht des Objektes, auf welches die Variable zeigt, definiert.

Globale Variablen

Globale Variablen beginnen mit einem $ Zeichen ($foo). Wenn globale Variablen nicht initialisiert sind, ist ihr Wert *nil*. Es gibt auch vordefinierte globale Variablen, von deren Verwendung in Bibliotheken jedoch abgeraten wird, da die meisten Werte durch andere Möglichkeiten eruiert werden können.

Instanzvariablen

Instanzvariablen beginnen mit einem @ Zeichen. Uninitialisierte Instanzvariablen weisen den Wert *nil* auf. Jede Instanz einer Klasse hat eine Menge an individuellen Instanzvariablen. In Klassenmethoden kann nicht auf Instanzvariablen zugegriffen werden.

Klassenvariablen

Klassenvariablen beginnen mit den Zeichen @@ und müssen, bevor sie in einer Methodendefinition verwendet werden können, initialisiert werden. Eine Referenz auf eine nicht initialisierte Klassenvariable produziert einen Fehler. Das Überschreiben von Klassenvariablen ist möglich. Klassenvariablen beziehen sich immer auf die innerste Klasse oder auf das innerste Modul. Wenn eine Klassenvariable an der obersten Stufe verwendet wird, so wird sie in *Object* definiert. Damit ist ihr Verhalten ähnlich dem einer globalen Variable.

Konstanten

Konstanten sind in Ruby gleichfalls Referenzen auf Objekte. Sie werden erst erstellt, wenn ihnen ein Wert zugewiesen wird. Dies geschieht im Allgemeinen in einer Klassen- oder Moduldefinition. Anders als in vielen anderen Programmiersprachen ist es in Ruby möglich, Konstanten zu verändern. Eine neuerliche Zuweisung an eine Konstante führt während der Laufzeit zu einer entsprechenden Warnmeldung. Um die verschiedenen Arten von Variablen und Konstanten zu unterscheiden, verwendet Ruby spezielle Zeichen.
Konstanten beginnen mit einem Großbuchstaben. Innerhalb von Klassen oder Modulen definierte Konstanten können auch nur innerhalb der Klasse oder des Moduls verwendet werden. Konstanten, die ausserhalb einer Klasse oder eines Moduls definiert sind, können von überall angesprochen werden. Innerhalb von Methoden dürfen keine Konstanten definiert werden. Die Verwendung uninitialisierter Konstanten verursacht einen Fehler, eine

[1] Ausgenommen Fixnum, true, false und nil (siehe Abschnitt 4.3.3, Seite 45).

Zuweisung auf eine bereits existierende Konstante liefert jedoch nur eine Warnmeldung. Die Begründung liegt darin, dass Konstanten zu den fünf Typen der Variablen gehören, somit ist eine Konstante eben nur eine spezielle Variable.

Lokale Variablen

Lokale Variablen beginnen mit einem Kleinbuchstaben oder mit einem Unterstrich (_). Der Sichtbereich einer lokalen Variable reicht von den Schlüsselwörtern *class*, *module*, *def* oder *do* bis zum korrespondierenden *end* oder vom Beginn eines Blockes { bis zum Blockende } (Blöcke siehe Abschnitt 4.3.6, Seite 53). Parameter von Methoden werden wie lokale Variablen der Methode betrachtet. Wenn ein Block aufgerufen wird, so wird der Wert an die entsprechende Variable zugewiesen. Wenn eine lokale Variable erstmals im Block einen Wert zugewiesen bekommt, so ist diese Variable lokal im Block. Wenn jedoch eine Variable mit demselben Namen bereits zuvor existiert hat, so erbt der Block diese Variable bei jedem Aufruf.

```
[1,2,3].each { |i| puts i }      # i ist lokal im Block
[1,2,3].each { |@i| puts @i }    # Instanzvariable @i
[1,2,3].each { |$i| puts $i }    # Globale Variable $i
[1,2,3].each { |I| puts I }      # Konstante I (liefert Warnungen)

sum = 0
var = nil
[1,2,3].each { |var| sum+=var }
p sum, var                       #  6, 3
```

Die in Beispielen häufig verwendete Methode *p* zeigt Objekte mit Hilfe der *inspect* Methode an.

Pseudo-Variablen

Pseudo-Variablen werden bezüglich Sichtbarkeit wie lokale Variablen behandelt. Jedoch sind Zuweisungen an Pseudo-Variablen nicht erlaubt. Die folgenden Pseudo-Variablen existieren in Ruby:

`self`	Das Empfängerobjekt der aktuellen Methode.
`true`	Der Wert, der true repräsentiert.
`false`	Der Wert, der false repräsentiert.
`nil`	Der Wert, der „undefiniert“ repräsentiert. Bei logischen Operationen entspricht dies dem Wert false.
`__FILE__`	Der Name des aktuellen Quellprogramms
`__LINE__`	Die aktuelle Zeilennummer im Quellprogramm

Vordefinierte Variablen und Konstanten

Die meisten vordefinierten Variablen sind globale Variablen, der kleinere Rest sind lokale Variablen. Die vordefinierten Variablen beeinflussen das gesamte Programm, wodurch von ihrer Verwendung in Bibliotheken abgeraten wird. Die Werte der vordefinierten Variablen können meist auch anders ermittelt werden.

Vordefinierte Konstanten stellen dagegen kein Risiko dar. Ein in dieser Arbeit eingesetzter Repräsentant dieser vordefinierten Konstanten ist *ARGV*. *ARGV* ist ein Array, das die Argumente der Kommandozeile vom Programmaufruf beinhaltet. Weitere vordefinierte Variablen und Konstanten können in [Matz, 2002] nachgelesen werden.

Fixnum, true, false, nil

Objekte von Fixnum, true, false und nil speichern ihre Werte nicht als Referenz, sondern als unmittelbaren Wert ab. Das bedeutet, dass zwei gleiche Fixnum-Werte dieselbe Instanz haben und somit ihre Instanzvariablen auf ein und dasselbe Objekt zeigen.

```
o1 = "A"     # String
o2 = "A"     # String
o3 = 16      # Fixnum
o4 = 16      # Fixnum

p o1.object_id == o2.object_id   # false
p o3.object_id == o4.object_id   # true
```

4.3.4 Operatoren

Wie man sich von einer modernen Sprache erwartet, besitzt Ruby eine Vielzahl an Operatoren. Die objektorientierte Natürlichkeit der Sprache lässt erwarten, dass die meisten Operatoren als Methoden zur Verfügung stehen. Somit können semantische Veränderungen an Operatoren durchgeführt werden, wann immer es notwendig ist. Zum Beispiel wird `a + b` als `a.+(b)` interpretiert. Dabei wird die Methode `+`, welche durch die Variable a referenziert ist, mit dem Parameter b aufgerufen. Folgende Arten an Operatoren werden in Ruby unterschieden:

- Methoden-Operatoren
  - Elementoperatoren (`[] []=`)
  - Arithmetische Operatoren[2] (`+ - **` etc.)
  - Relationen und Vergleichsoperatoren (`=== < <= <=> ==` etc.)
  - Bit und Schiebe Operatoren (`& ^ | << >>`)
  - Unäre Operatoren (`+ - ! ~`)
  - defined? Operator
- Nicht-Methoden-Operatoren
  - Zuweisungsoperatoren (`= -= ||=` etc.)
  - Logische Operatoren (`! not &&` etc.)
  - Bereichs Operatoren (`.. ...`)
  - Ternärer Operator (`? :`)

[2] In Ruby gibt es keinen inkrement (++) bzw. dekrement (−−) Operator.

Zuweisungsoperator

Für jeden der Operatoren `+ - * / % ** & | ^ << >> && ||` existiert auch eine kombinierte Form mit einem Zuweisungsoperator (`+= -=`). Der Zuweisungsoperator weist ein oder mehrere *rvalues* (Werte an der rechten Seite) an einen oder mehrere *lvalues* (Werten an der linken Seite) zu. Das bedeutet, dass die Zuweisung von jedem einzelnen *lvalue* abhängt. Wenn *lvalue* der Name einer Variablen oder Konstante ist, bekommt die Variable bzw. Konstante eine Referenz auf das korrespondierende *rvalue*.

```
a, b, c = 1, "Peter", [2, 3]
```

Wenn der *lvalue* ein Objektattribut ist, so wird für das korrespondierende Attribut die setter-Methode des Empfängers mit dem *rvalue* als Parameter aufgerufen.

```
obj = A.new
obj.wert = "foo"      # ist aequivalent zu obj.wert("foo")
```

Wenn der *lvalue* eine Referenz auf ein Arrayelement darstellt, so ruft Ruby den Zuweisungsoperator `[]=` des Empfängers auf, wobei jeder Index zwischen den eckigen Klammern gefolgt vom *rvalue* als Parameter übergeben wird.

```
obj[] = "foo"         # obj.[]=("foo")
obj[1] = "foo"        # obj.[]=(1, "foo")
obj["a", 3] = "foo"   # obj.[]=("a", 3, "foo")
```

Parallele Zuweisungen

Wie bereits erwähnt, können Zuweisungen mehrere *rvalues* bzw. *lvalues* aufweisen. Dabei setzt Ruby folgende Regeln ein:

1. Wenn der letzte *rvalue* mit einem `*` versehen ist und ein Objekt der Klasse Array ist, so wird der *rvalue* mit den Elementen des Arrays, in dem jedes Element ein eigenes *rvalue* darstellt, ersetzt.

2. Wenn es mehrere *lvalues* gibt, jedoch nur ein *rvalue*, so wird der *rvalue* in ein Array konvertiert und wie in 1. behandelt.

3. Mehrere *rvalues* werden sukzessiv an die *lvalues* zugewiesen. Die Zuweisung erfolgt quasi parallel, z.B.: a, b = b, a (swap Funktion).

4. Wenn es mehr *lvalues* als *rvalues* gibt, so wird dem Rest der *lvalues* *nil* zugewiesen.

5. Wenn es mehr *rvalues* als *lvalues* gibt, so werden die überschüssigen *rvalues* ignoriert.

6. Eine leichte Abänderung der Regeln tritt auf, wenn der letzte *lvalue* mit einem führenden `*` versehen ist. Diesem *lvalue* werden alle restlichen *rvalues* als Array zugewiesen.

7. Wenn ein *lvalue* in einer geklammerten Liste steht, wird es als verschachtelte Zuweisung behandelt. Die Liste wird vom korrespondierenden *rvalue* – wie in diesem Regelwerk beschrieben – zugewiesen.

```
b, c = [1, 2, 3]                 # b=1, c=2
b, *c = [1, 2, 3]                # b=1, c=[2,3]
b, *c = 0, [1, 2, 3]             # b=0, c=[[1,2,3]]
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
```

Spezielle Operatoren

Neben dem Vergleichsoperator `==` gibt es in Ruby auch den Relationenoperator `===`. Der Operator wird in der when-Klausel der case-Anweisung als Gültigkeitsoperator verwendet. Das Verhalten des Relationenoperators muss nicht unbedingt ident mit dem des Vergleichsoperators sein. In einigen Fällen kann dies der Fall sein, jedoch ist der Relationenoperator in unterschiedlichen Klassen unterschiedlich definiert und kann bei verschiedenen Typen von Operanden auch noch unterschiedliche Ergebnisse liefern. Aufgrund dieser Tatsache muss das Ergebnis von `x === y` nicht notwendigerweise gleich dem Ergebnis von `y === x` entsprechen. Der Relationenoperator ist also im Gegensatz zum Vergleichsoperator nicht kommutativ.

Ein anderer spezieller (kommutativer) Operator ist `<=>`, auch allgemeiner Vergleichsoperator genannt. Wenn der Empfänger kleiner als sein Argument ist, liefert diese Methode -1 zurück. Bei Gleichheit 0 und bei größer +1. Die Methodenoperatoren `==` und `!=` verwenden diesen allgemeinen Vergleichsoperator in ihrer Implementierung. Beim Vergleichen von Arrays muss beachtet werden, dass das Modul Compareable, welches zusätzlich noch die Operatoren `<`, `>`, `<=` und `>=` anhand des Operators `<=>` zur Verfügung stellt, nicht enthalten ist. Dies schränkt den Vergleich bei Arrays auf Gleichheit bzw. Ungleichheit ein. Bei Bedarf können jedoch die weiteren Vergleichsoperatoren recht einfach dazudefiniert werden.

Einige der Operatoren in Ruby haben mehr als nur einen Verwendungszweck. Zum Beispiel wird der Operator `<<` einmal für eine bitweise Linksverschiebung eingesetzt, ein anderes mal als Anfügeoperator für Arrays, Zeichenketten etc. und dann auch noch als Markierung für den here-Dokument Operator (siehe Abschnitt 4.3.2, Seite 40).

Der *defined?*-Operator ist ein Methodenparameter, der überprüft, ob der angegebene Ausdruck definiert ist oder nicht. Wenn der Ausdruck nicht definiert ist, retourniert *defined? nil*, ansonsten eine Beschreibung in Form einer Zeichenkette vom definierten Ausdruck.

```
defined? foo     # "local-variable"
defined? bar     # nil
defined? $0      # "globale variable" (vordefiniert, Programmname)
```

Redefinieren von Operatoren

Jeder Operator, der durch eine Methode implementiert ist, kann in Ruby redefiniert werden.

```
class Fixnum
  def +(other)
    self * other
  end
```

```
end

4 + 4   # 16
```

Zu beachten ist, dass die unären Operatoren `+` und `-` als `+@` und `-@` zu definieren sind.

4.3.5 Methoden

In Ruby wird unter dem Begriff Methode sowohl die benannte Operation (z.B.: puts), als auch der Code, den eine bestimmte Klasse zur Durchführung einer Operation benötigt, verstanden. Streng gesagt gibt es in Ruby keine Funktionen[3], sondern nur Methoden. Jeder Code in Ruby stellt eine Methode zu einem Objekt dar. Obwohl einige Methoden in Ruby eine ähnliche Flexibilität aufweisen wie Funktionen in anderen Programmiersprachen, sind diese hinter der Kulisse immer noch Methoden und nicht Funktionen.

Methodendefinition

> def *dname [([arg [=val], ...] [,* `*`*vararg][,* `&`*blockarg])]*
>   *body*
> end

Ein *dname* ist entweder ein redefinierter Operator oder ein Name. Wenn *dname* einen Namen darstellt, so muss dieser mit einem Unterstrich (_) oder einem Kleinbuchstaben beginnen. Danach können optional Groß-, Kleinbuchstaben, Zahlen und Unterstriche beliebig folgen.
In Ruby ist es üblich, bei speziellen Methoden eines der folgenden Suffixes beim Methodennamen anzugeben:

? für Methoden, die Abfragen darstellen, z.B.: instance_of?, defined?

! bei Methoden, die gefährlich sind oder den Empfänger verändern, z.B.: gsub!, chop!

= signalisiert, dass die Methode ein Attribut setzt, z.B.:

```
class Foo
  def name=(other)
    @name = other
  end
end
```

Innerhalb von Methoden dürfen keine Klassen-, Modul- oder Instanzmethoden definiert werden. Jedoch dürfen Singletonmethoden in Methodendefinitionen enthalten sein. Der Hauptteil (*body*) einer Methode verhält sich wie ein `begin/end` Block, in dem Anweisungen für Ausnahmebehandlungen enthalten sein dürfen (siehe Abschnitt 4.3.12, Seite 62).

[3] Funktionen im Sinne von Code, der keinem Objekt zugewiesen ist.

Methodenargumente

Eine Methode in Ruby kann beliebig viele reguläre Argumente, ein optionales Array-Argument und ein optionales Block-Argument haben. Eine Liste von Argumenten wird zwischen einfachen Klammern nach dem Methodennamen geschrieben. Die Argumente in der Liste werden durch Kommas getrennt.

Unter einem regulären Argument wird eine lokale Variable verstanden. Dieses kann optional von einem Gleichheitszeichen (=) und einem Ausdruck, der einen Defaultwert darstellt, ergänzt werden. Es ist sogar erlaubt, im Ausdruck einen Parameter zu referenzieren, der in der Argumentliste – jedoch vor diesem Ausdruck – aufscheint.

```
def foo(a=16, b=a+1, c)
  [a,b,c]
end
```

Ein optionales Array-Argument kann erst nach einem regulären Argument folgen und darf keinen Defaultwert aufweisen. Beim Aufruf einer Methode wird das Array-Argument auf eine Referenz eines neuen Objektes der Klasse *Array* gesetzt. Wenn beim Aufruf der Methode mehr Argumente angegeben wurden als reguläre Argumente definiert sind, so werden alle überzähligen Parameter in dem neu erstellten Array zusammengefasst. Eventuelle Defaultwerte bei den regulären Parametern werden vorrangig überschrieben, danach werden erst die restlichen Parameter an das Array übergeben.

```
def foo(a, b=0, *c)
  [a,b,c]
end
foo 1              # [1, 0, []]
foo 1, 2, 3        # [1, 2, [3]]
foo 1, 2, 3, 4     # [1, 2, [3, 4]]
```

Das optionale Blockargument muss als letzter Parameter angegeben werden. Sobald die Methode aufgerufen wird, überprüft Ruby die Existenz eines Blocks. Ist diese gegeben, so wird dieser zu einem Objekt der Klasse *Proc* konvertiert und dem Blockargument zugewiesen. Wenn kein Block beim Aufruf angegeben wurde, so wird das Blockargument auf *nil* gesetzt (siehe Abschnitt 4.3.6, Seite 53).

```
def foo(a, &b)
  b.call(a)
end
foo(2) { |x| x*x } # 4
```

Grundsätzlich werden in Ruby beim Aufruf einer Methode die Parameter als Referenz auf ein Objekt übergeben (Call by Reference). Die Parameter können somit potentiell geändert werden. Der einfachere Typ Fixnum wird als unmittelbarer Wert übergeben (Call by Value), womit Änderungen nur innerhalb der Methode ihren Wirkungsbereich haben (siehe destruktive Methoden Abschnitt 4.3.5, Seite 52). Das gleiche gilt auch für true, false und nil.

```
def foo(v)
  v.succ!                     # destruktive Methode (ermittelt den Nachfolger)
```

```
end

x = y = "A"                 # ein Objekt
foo(x)                      # call by reference
x == "B" && y == "B"        # true
```

Methodenaufruf

[receiver.] name [parameters] [block]
[receiver::] name [parameters] [block]

parameters ← (*[param, ...] [, hash] [* ∗*array] [* &*aProc]*)

block ← { *block* }
do *block* end

Zuerst werden die Initialparameter den Argumenten der Methode zugewiesen. Danach kann eine Liste von `key => value` Paaren folgen. Diese Paare werden in ein neues Objekt der Klasse *Hash* zusammengefasst und als ein Parameter übergeben. Anschließend kann ein Parameter mit einem `*` als Prefix (Array) und eventuell ein weiterer Parameter mit einem `&` als Prefix (Block) übergeben werden (siehe Methodenargumente). Auf Grund dieser Eigenschaften ist es in Ruby möglich, Methoden mit einer variabel langen Argumentliste zu definieren.

```
def foo(a, b, *c, &d)
  puts "a = #{a}"
  puts "b = #{b}"
  puts "c = #{c}"
  puts "d = #{yield(d)}"
end

foo("Peter", 1 => "Run", 2 => "Bike", *[0,1], &proc { true })

# Ergebnis: a=Peter   b=1,Run,2,Bike   c=0,1   d=true
```

Eine Methode wird aufgerufen, indem ihr Name an den Empfänger geschickt wird. Wenn kein Empfänger gegeben ist, so wird *self* angenommen. Der Empfänger sucht zuerst nach dem Methodennamen in seiner eigenen Klasse und danach in den Superklassen. Die Instanzmethoden der inkludierten Module verhalten sich dabei so, als ob sie anonyme Superklassen von der Klasse, von der sie inkludiert wurden, wären. Wird die Methode nicht gefunden, führt Ruby die Methode *method_missing* im Empfänger aus. Das Standardverhalten dieser Methode, welches in *Kernel.method_missing* definiert ist, meldet einen entsprechenden Fehler und beendet das Programm.

Die explizite Angabe eines Empfängers während des Methodenaufrufs kann durch den Namen des Empfängers, gefolgt von einem Punkt (.) oder zwei Doppelpunkten (::) erfolgen. Der einzige Unterschied zwischen den beiden Varianten kommt dann zur Geltung, wenn ein Methodenname mit einem Grossbuchstaben beginnt. In diesem Falle nimmt Ruby an, dass es sich um einen Aufruf einer Konstanten im angegebenen Empfänger handelt, sofern nicht durch Klammerung der Parameter die Eindeutigkeit eines Methodenaufrufs gegeben ist.

```
Foo.A()      # Methodenaufruf
Foo.A        # Methodenaufruf
Foo::A()     # Methodenaufruf
Foo::A       # Zugriff auf Konstante
```

Private Methoden dürfen nicht mit einem expliziten Empfänger aufgerufen werden, da sie nur im aktuellen Objekt verfügbar sind (*self*).
Methodenaufrufe mit Parametern können auch ohne Klammern erfolgen, sofern ihr Aufruf eindeutig ist. Dies hat zur Konsequenz, dass unsauber geschriebene Ausdrücke zu einer Fehlermeldung oder im schlimmeren Fall sogar zu einem Fehlverhalten führen können.

```
a = b + c
a = b+c
a = b+ c
a = b +c
```

Die ersten drei Varianten sind äquivalent zueinander. Beim letzten Beispiel nimmt der Parser an, dass es sich bei b um einen Methodenaufruf handelt und `+c` der dazugehörende Parameter ist. Falls dies nicht der Fall ist, wird eine entsprechende Fehlermeldung ausgegeben. Es wird daher empfohlen, Leerzeichen an gegebenen Stellen zu platzieren.

Rückgabewert

return *[expr, ...]*

Der Rückgabewert einer Methode ist der Wert des zuletzt ausgeführten Ausdrucks. Eine *return* Anweisung beendet unmittelbar die Methode. Der Wert von *return* ist nil wenn kein Parameter angegeben wird, oder der Wert des Parameters wenn ein Parameter beim Aufruf angegeben wurde, oder ein Array mit allen Parametern, die bei einem Aufruf von mehr als einem Parameter angegeben wurden.

Klassenmethoden

Klassenmethoden unterscheiden sich in der Definition gegenüber Instanzmethoden. In Ruby werden Klassenmethoden durch Voranstellen des Klassennamens, gefolgt von einem Punkt und dem eigentlichen Methodennamen definiert.

```
class Foo
  def instMeth             # Instanzmethode
  end
  def Foo.classMeth        # Klassenmethode
  end
end
```

Super

super *[([param, ...] [*array])] [block]*

Außerhalb von Klassen und Modulen stellt eine „alleinstehende“ Methodendefinition eine private Methode der Klasse *Object* dar. Eine solche Methode kann in jedem Kontext ohne expliziter Angabe eines Empfängers aufgerufen werden.

Die Anweisung super führt die Methode mit dem gleichen Namen in der Superklasse aus. Wenn weder Argumente noch Klammern angegeben wurden, werden die Methodenargumente direkt an die Superklasse weitergegeben. Das bedeutet, dass ein Aufruf von *super()*, der keine Argumente an die Superklasse weitergibt, sich vom einfachen Aufruf *super* unterscheidet.

Destruktive Methoden

Eine destruktive Methode[4] ist eine Methode, die das Attribut eines Objektes verändert. In String, Array und Hash existieren solche destruktiven Methoden.
In diesen Klassen befinden sich oft zwei Methoden mit dem gleichen Namen. Die eine Methode retourniert eine Kopie vom Objekt und die andere – mit einem ! als Suffix – stellt eine destruktive Methode dar.

```
def up(str)
  str.upcase!
end

def down(str)
  str.downcase
end

s = "foo"
up(s)           # destruktive Methode
p down(s)       # foo
p s             # FOO
```

In diesem Fall wird das Argument str in der Methode up durch die Verwendung der Methode str.upcase! „bösartig“ verändert. Die Methode up hätte demnach ebenfalls mit ! gekennzeichnet werden sollen. Destruktive Methoden müssen nicht unbedingt mit ! gekennzeichnet sein, jedoch wird dies empfohlen.

Singletonmethoden

In Ruby können Methoden so definiert werden, dass sie nur mit einem spezifischen Objekt assoziiert sind. Solche Methoden werden Singletonmethoden genannt. Die Definition einer Singletonmethode lautet:

> def *receiver.dname [([arg [=val], ...][,* `*`*vararg][,* `&`*blockarg])]*
> *body*
> end

Statt dem Punkt nach *receiver* können auch zwei Doppelpunkte (::) verwendet werden. Die Funktionalität ist die gleiche, jedoch werden Doppelpunkte meistens für Klassenmethoden verwendet.

[4]Destruktive Methoden können eigentlich nur durch das C-API in Ruby erstellt werden.

Singletonmethoden dürfen auf Grund einer Spracheinschränkung nicht auf Instanzen von *Fixnum*- oder *Symbol*- Klassen definiert werden.

4.3.6 Blöcke

methodcall { [| *[variable, ...]* |]
    body
}

methodcall do [| *[variable, ...]* |]
    body
end

Ein Block befindet sich entweder zwischen geschwungenen Klammern oder zwischen einem `do/end` Paar. Der Block startet optional mit einer Liste von Argumenten zwischen vertikalen Strichen, gefolgt von einem *body*-Teil, in dem Ruby Anweisungen und Ausdrücke stehen.

Der Code im Block wird ausgeführt nachdem ein Wert von der Methode an den Block und an die Argumentenliste zugewiesen wurde. Ein Block hat seinen eigenen Sichtbereich für lokale Variablen. Die lokalen Variablen in der Argumentenliste werden im Block zuerst definiert. Der Sichtbereich während der Ausführung des Blocks kann sich auch auf lokale Variablen außerhalb des Blocks erstrecken. Die Sicht auf äußere lokale Variablen zu erweitern ist ein Spezifikum der Blöcke. Klassen, Module und Methodendefinitionen haben diese Eigenschaft nicht.

Der Unterschied zwischen den geschwungenen Klammern und der `do/end` Form bezieht sich auf die Priorität der Bindung der Parameter bei der Ausführung der Methode. Dabei haben die geschwungenen Klammern `{...}` eine höhere Priorität als die `do/end` Form. Traditionell ist es in Ruby üblich, dass einzeilige Blöcke mit geschwungenen Klammern und mehrzeilige Blöcke mit dem `do/end` Paar codiert werden.

```
a=4
[1, 2, 3].each {|x| p x * a}     # 4 8 12
```

4.3.7 Klassen

class *classname [< superclass]*
    body
end

Eine Klassendefinition in Ruby erstellt oder erweitert ein Objekt der Klasse *Class* durch die Ausführung des Codes im *body*. Das erstellte Objekt *Class* wird einer globalen Konstante mit dem Namen *classname* zugewiesen. Somit ist ein Klassenname eine Konstante und wie diese muss der Name mit einem Großbuchstaben beginnen.
Wenn eine *superclass* angegeben wurde, so muss der Ausdruck von *superclass* zu einem Objekt der Klasse *Class* evaluieren. Wenn keine *superclass* definiert wurde, so ist implizit *Object* die Superklasse. Durch die Anweisung class wird ein neuer Sichtbereich für die lokalen Variablen im *body*-Teil eingeführt. Im *body*-Teil der Klassendefinition können Klassenkonstanten, Klassenvariablen, Klassenmethoden, Instanzvariablen und Instanzmethoden

definiert sein. Die Klassendaten sind für alle Objekte verfügbar, wogegen Instanzdaten nur für das eine Objekt zur Verfügung stehen.

```
class Foo           # implizit abgeleitet von Object
end

class Bar < Fixnum  # erweitert Superklasse Fixnum
end
```

Klasseninstanzierung

obj = receiver.new [([args, ...])]

Die Klasse *Class* hat die Instanzmethode *Class#new* definiert. Diese

- erstellt ein Objekt von der Klasse des Empfängers,
- setzt den Objekttyp entsprechend dem des Empfängers
- und ruft die Instanzmethode *initialize* im neu erstellen Objekt auf und gibt dabei die an *new* übergebenen Parameter unverändert an diese weiter.

Wenn eine Klassendefinition die Methode *new* überschreibt ohne die Methode *super* aufzurufen, können keine Objekte von dieser Klasse mehr erstellt werden.
Nicht immer werden neue Instanzen einer Klasse durch die Methode *new* erstellt. Für die Klasse *Fixnum* gibt es zum Beispiel keine Methode *new*.

```
a = Array::new
o = Object::new
i = 3
s = "Foo"
```

Alle Klassen in Ruby sind weiters Objekte der Metaklasse *Class*, siehe Abschnitt 4.3.11, Seite 59. Weiters sind Klassen in Ruby offen, wodurch jederzeit Methoden zu bestehenden Klassen hinzugefügt, aber auch entfernt werden können. Die Bereinigung von nicht mehr benötigten Objekten erfolgt in Ruby transparent durch die Speicherverwaltung (garbage collection).

Verschachtelte Klassen

Verschachtelte Klassen werden als Konstanten innerhalb der Klasse abgelegt. Sie können außerhalb der definierten Klasse mit zwei Doppelpunkten (::) und ihrem Namen angesprochen werden. Mit verschachtelte Klassen wird eine Art Kapselung bewirkt, die ähnlich der Instanzvariablen und Instanzmethoden ist.

```
class Foo
  class Bar
    def hallo
      "hallo"
    end
  end
end

Foo::Bar.new.hallo    # "hallo"
```

Singletonklassen

> class << *[object]*
>   *body*
> end

Attributdefinitionen (Methoden und Konstanten) für ein spezifisches Objekt können mit Hilfe der class-Definition konstruiert werden. Durch die Anwendung dieser class Definition wird eine virtuelle Klasse für das Objekt *object* erstellt.

```
class Foo
end
a , b = Foo.new, Foo.new

class << b                  # Singletonklasse
  undef freeze
end
a.methods == b.methods      # false
```

Object und Kernel

Object ist die Superklasse aller Klassen. Eine auf der obersten Stufe im Programm definierte Methode wird eine private Methode dieser Klasse. Diese Eigenschaft bewirkt, dass die Methode sich wie eine Funktion in anderen Programmiersprachen verhält. Die einzige Methode, die *Object* zur Verfügung stellt, ist die private Instanzmethode initialize. Diese wird in den meisten Fällen jedoch in einer Subklasse überschrieben.

Object inkludiert aber auch das Modul *Kernel*, wodurch eine Menge an Instanzmethoden zur Verfügung steht. Unter diesen zahlreichen Methoden befinden sich unter anderem die Vergleichsoperatoren sowie die häufig verwendeten Methoden class, freeze, froze?, id, inspect, instance_of?, methods und to_s.

Eingebaute Klassenbibliothek

Die eingebaute Klassenbibliothek von Ruby bietet eine reichhaltige Menge an Klassen zum Erstellen von Programmen an. Darunter sind Klassen für Textmanipulationen (String), Betriebssystem-Services (IO, File, Process, etc.), Zahlen (Integer, Fixnum, etc.) und viele mehr. Mit diesen einfachen Klassen können bereits mächtige Ruby Programme generiert werden.

Standard-Klassenbibliothek

Die Ruby Standard-Klassenbibliothek erweitert das Fundament der eingebauten Klassenbibliothek. Darunter befinden sich Klassen für Netzwerk-Programmierung, weitere Betriebssystem–Services, nebenläufige Prozesse et cetera. Die Existenz dieser Bibliothek erspart dem Entwickler sehr viel an Programmier-, Debugging- und Testaufwand. Der Quellcode dieser Bibliothek ist Bestandteil der Ruby-Distribution und wird bei der Installation im Verzeichnis *lib* abgelegt.

4.3.8 Zugriffskontrolle

In Ruby können nur für Methoden in Klassen und Modulen Sichtbarkeitsstufen definiert werden. Auf Variablen kann von aussen mit Hilfe von setter/getter Methoden kontrolliert zugegriffen werden. Klassenkonstanten sind öffentlich zugänglich und können nicht abgeschottet werden. Eine Methode muss bereits definiert sein, damit eine der drei folgenden Stufen der Sichtbarkeit auf ihr angewendet werden kann:

> public([*symbol...*])
> Macht die spezifizierte Methode *symbol* öffentlich zugänglich, womit sie von überall her aufrufbar ist. Wenn kein Argument angegeben wird, so werden alle folgend definierten Methoden in der Klasse oder im Modul öffentlich zugänglich.
>
> protected([*symbol...*])
> Macht die spezifizierte Methode *symbol* protected. Die Methode kann nur von Instanzen derselben Klasse aufgerufen werden. Wenn kein Argument angegeben wird, so werden alle folgend definierten Methoden in der Klasse oder im Modul protected gemacht.
>
> private([*symbol...*])
> Macht die spezifizierte Methode *symbol* private. Die Methode kann nur ohne Angabe eines Empfängers (ist implizit *self*) aufgerufen werden. Somit ist sie nur in der definierten Klasse und von direkt abgeleiteten Klassen desselben Objektes aufrufbar. Wenn kein Argument angegeben wird, so werden alle folgend definierten Methoden in der Klasse oder im Modul private gemacht.

Standardmäßig ist eine Methode in einer Klasse oder im Modul öffentlich zugänglich. Methoden außerhalb einer Klasse oder eines Moduls (top level) sind jedoch private. Die Begründung für Letzteres liegt darin, dass diese Methoden in der Klasse *Object* definiert sind. Sie sind somit öffentlich aufrufbar, jedoch ohne Angabe eines Empfängers.

```
class Foo
  def a       # oeffentlich
  end

  protected
  def b       # protected
  end

  def c       # protected
  end

  def d       # private
  end
  private :d
end
```

4.3.9 Alias

Mit *alias* wird ein neuer Name für eine existierende Methode, einen Operator oder eine globale Variable vergeben. Lokale Variablen, Instanzvariablen, Klassenvariablen und Konstante können nicht mit einem Alias versehen werden. Die Parameter von *alias* können Namen oder Symbole sein.

```
def foo
  puts "foo"
end

alias _foo foo

def foo
  puts "_foo"
end

foo     # _foo
_foo    # foo
```

4.3.10 Module

> module *modulename*
>   *body*
> end

Ein Modul ist einer Klasse sehr ähnlich, hat aber im Gegensatz zu dieser keine Superklasse und kann auch nicht instanziert werden. Die Klasse *Module* ist die Superklasse der Klasse *Class*.

Eine Klasse kann zwar von einer anderen Klasse abgeleitet werden, jedoch nicht von einem Modul. So wie der Klassenname ist auch der Modulname eine Konstante und daher mit einem großen Anfangsbuchstaben zu versehen. Falls ein *modulename* bereits existiert, werden die Eigenschaften des neuen Moduls zu denen des alten Moduls hinzugefügt.

Mit Modulen ist es möglich, Methoden, Klassen und Konstanten zu vereinigen. Die zwei primären Vorzüge von Modulen in Ruby sind:

1. Module stellen einen eigenen Namensbereich zur Verfügung.
2. Module ermöglichen mixins.

Namensraum

Dadurch dass Module einen eigenen Namensraum besitzen, können weniger Namenskollisionen auftreten. Eine Modulmethode wird mit dem Modulnamen aufgerufen, also ohne „echten“ Empfänger. Dies entspricht der Vorgehensweise beim Aufruf von Klassenmethoden. Analog gilt dies auch für die Konstanten in den Modulen.

```
module Foo
  PI = 3.14159        # Modulkonstante
```

```
  def Foo.flaeche(r)  # Modulmethode
    r*r*PI
  end
end

Foo::PI               # 3.14159
Foo.flaeche(2)        # 12.56636
```

Mixins

Mit mixins eröffnet sich eine wunderbare und kontrollierte Möglichkeit, den Klassen in Ruby weitere Funktionalität zukommen zu lassen. Ein mixin ist eine abstrakte Subklasse, die an unterschiedlichen Superklassen angewendet werden kann [Bracha et al., 1990]. Die enorme Wirkung ergibt sich, wenn mixins beginnen, mit dem Code einer Klasse zu interagieren. Neben den Modulmethoden kann ein Modul auch Instanzmethoden aufweisen. Dies wirkt im ersten Moment widersprüchlich, da Module nicht instanzierbar sind und eine Instanzmethode nicht ohne einen Empfänger aufrufbar ist. Die Methode *include* ermöglicht aber, dass Instanzmethoden des Moduls als Instanzmethoden in die Klasse gemixt werden können. Eventuell vorhandene Modulmethoden werden dabei von der *include* Methode ignoriert.

```
module Foo
  PI = 3.14159        # Modulkonstante
  def flaeche(r)      # Instanzmethode
    r*r*PI
  end
end

include Foo

p PI                  # 3.14159
p flaeche(2)          # 12.56636
```

Wenn ein Modul sich nicht in der selben Datei befindet, muss es zuvor mit *require* geladen werden. Ein *include* kopiert nicht wirklich die Instanzmethoden des Moduls in die Klasse, sondern erstellt eine Referenz auf das Modul. Das hat den Vorteil, dass sich eine Änderung in einem Modul auf alle Klassen, die dieses Modul inkludiert haben, auswirkt. Durch diese Eigenschaft verhält sich ein Modul ähnlich einer Superklasse.

4.3.11 Zusammenspiel von Klassen und Objekten

Zum besseren Verständnis des Zusammenspiels zwischen Klassen, Objekten, Klassenobjekten, Instanzmethoden, Klassenmethoden und Singletonklassen in Ruby werden diese Konzepte nun genauer betrachtet. Die graphische Notation wurde aus [Thomas et al., 2001] übernommen.
Im Prinzip läuft die Interaktion von Klassen und Objekten nach einem konstanten Schema ab. Objekte referenzieren Klassen und Klassen referenzieren entweder keine, eine oder mehrere Superklassen.

Einfache Klassen

Im folgenden Beispiel wird eine Klasse *Zaehler* definiert, in der wiederum die Instanzmethode addiere definiert ist. Danach wird die Klasse instanziert und der Variablen *z* als Referenz zugewiesen.

```
class Zaehler
  def addiere
    ...
  end
end

z = Zaehler.new
```

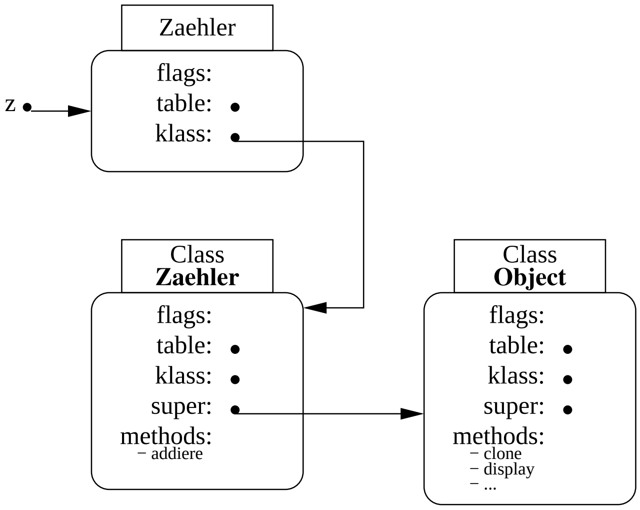


Abbildung 4.1: Ein einfaches Objekt mit seiner Klasse und Superklasse

Jede Ruby-Klasse ist ein Objekt der Klasse *Class*. In der Abbildung 4.1 zeigt die Referenz der Variable z auf ein Objekt, dessen Objektklasse *Zaehler* und dessen Superklasse *Object* ist. Wie man sieht, zeigt die Klassenreferenz *klass* des neuen Objektes auf das Klassenobjekt, und dessen super Referenz zeigt auf die Klassenreferenz der Superklasse.
Wenn nun der Aufruf z.addiere ausgelöst wird, so beginnt Ruby beim Empfänger *z* und folgt dessen *klass* Referenz zum Klassenobjekt *Zaehler*. In diesem Klassenobjekt wird die Methode gefunden und ausgeführt.
Wenn stattdessen z.display aufgerufen wird, so startet Ruby auf die gleiche Art und Weise. Jedoch findet er die Methode *display* diesmal nicht im Klassenobjekt *Zaehler* und verfolgt daher den *super*-Zeiger zur Superklasse *Object* und wird schlußendlich dort fündig.

Metaklasse

Wenn nun zum oberen Beispiel noch eine Klassenmethode hinzukommt, funktioniert dieser Weg nicht, da in diesem Fall der Empfänger die Objektklasse selbst ist.

```
class Zaehler
  def Zaehler.stand
```

```
  end
  ...
end
```

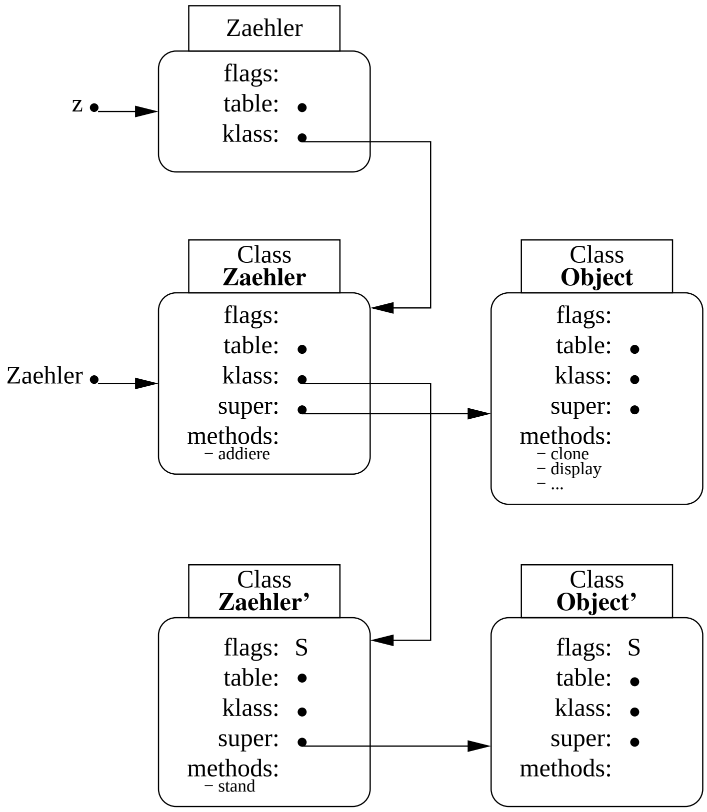


Abbildung 4.2: Metaklasse zum Zaehler hinzufügen

Damit wie im ersten Fall das Prinzip konsistent bleibt, ist es notwendig, eine Metaklasse *Zaehler'* einzuführen. Dies alleine ist jedoch nicht ausreichend. So wie die Klasse *Zaehler* ist auch die Metaklasse *Zaehler'* von *Object's* Metaklasse abgeleitet, weshalb noch die Metaklasse Object' in der Abbildung 4.2 eingetragen ist.
Wenn nun der Aufruf Zaehler.stand erfolgt, läuft der Suchprozess wie zuvor ab. Die Empfängerklasse *Zaehler* wird kontaktiert, danach wird über *klass* die Metaklasse *Zaehler'* referenziert und die Methode lokalisiert. Man beachte, dass das *flag* auf „S" für Singletonklasse gesetzt wurde. Klassen, die Ruby automatisch generiert, werden damit gekennzeichnet[5].

Objektspezifische Klassen

In Ruby ist es möglich, eine Klasse zu einem bestimmten Objekt zu erstellen. Das folgende Beispiel erstellt zwei Instanzen der Klasse *Zaehler*. Danach wird eine anonyme Klasse an eines dieser Objekte hinzugefügt und eine neue Methode *loeschen* erstellt (Singletons siehe Abschnitt 4.3.7, Seite 55).

[5]Der von außen offensichtlichste Unterschied einer Singletonklasse zu einer normalen Klasse ist der, dass diese Singletonklasse niemals in einer Liste von Objekten, die von Methoden wie Module#ancestors oder ObjectSpace.each_object zurückgegeben werden, aufscheint.

```
a = Z.new
b = a.dup

class << a
  def loeschen
  end
end
```

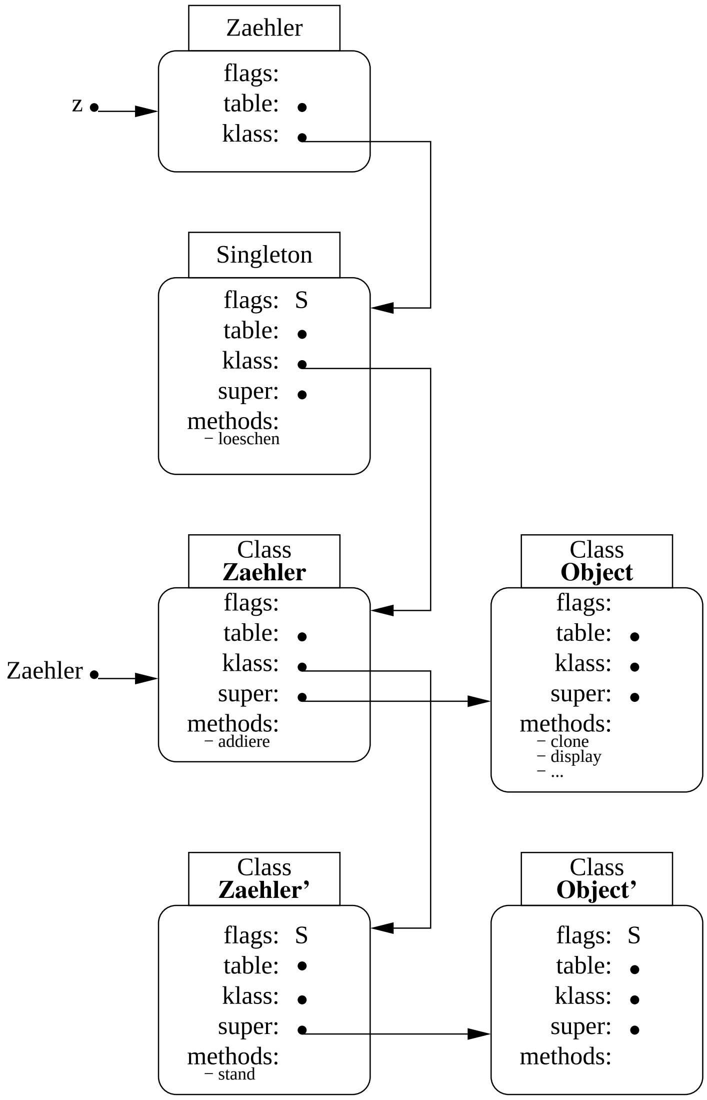


Abbildung 4.3: Hinzufügen einer Singletonklasse zu einem Objekt

Wie in der Abbildung 4.3 zu sehen ist, wurde diese Singletonklasse als direkter Nachfolger der Klasse *z* eingefügt und der ursprüngliche Nachfolger, die Objektklasse *Zaehler*, wird zur Superklasse vom Singleton. Sobald einmal eine Singletonklasse erstellt wurde, verwendet Ruby aus Effizienzgründen immer diese und legt keine weiteren Singletonklassen für dieses Objekt an.

Mixins

```
module Persistenz
  def schreiben
  end
end
class Zaehler
  include Persistenz
end
```

Falls eine Klasse auch noch ein Modul inkludiert, werden die Instanzmethoden des Moduls als Instanzmethoden der Klasse verfügbar. Dies verhält sich nahezu so, als ob das Modul zu einer Superklasse der auslösenden Klasse wird.
Wenn ein Modul inkludiert wird, so erstellt Ruby dafür eine anonyme Proxy-Klasse, die dieses Modul referenziert. Platziert ist es als direkte Superklasse der Klasse, die es inkludiert hat. Die Proxy-Klasse hat Referenzen zu den Instanzvariablen und Methoden des Moduls. Das ist eine wichtige Eigenschaft, denn dieses Modul kann von mehreren Klassen inkludiert worden sein und könnte somit in verschiedenen Vererbungsketten auftreten. Dank der Proxy-Klasse ist in Wirklichkeit nur *ein* darunterliegendes Modul vorhanden. Dies hat folgende Eigenschaft. Wenn eine Veränderung in der Definition des Moduls durchgeführt wird, so betrifft dies alle bisherigen und zukünftigen Klassen, die dieses Modul inkludiert haben bzw. von diesem inkludiert werden, siehe Abbildung 4.4.

Falls mehrere Module von der Klasse inkludiert werden, so werden sie in der Kette einfach angehängt. Wenn ein Modul selber noch andere Module inkludiert hat, so wird eine Kette an Proxy-Klassen auf die Klasse, die das Modul inkludiert, angefügt. Somit ist sichergestellt, dass es pro Modul, ob direkt oder indirekt inkludiert, je eine Proxy-Klasse gibt.

Folgende Aussage findet sich in [Fulton, 2001] wieder:

> „*Class* is an object and that *Object* is a class.“

4.3.12 Ausnahmebehandlungen (exceptions)

Wie viele andere moderne Programmiersprachen unterstützt Ruby das Konzept der Ausnahmebehandlung. Ausnahmen sind in Ruby Objekte der Klasse *Exception*. Von *Exception* direkt abgeleitet sind die Klassen *Fatal, Interrupt, NoMemoryError, SignalException, ScriptError, StandardError* und *SystemExit*.

Werfen einer Ausnahme

> raise
> raise *message*
> raise *exceptionobject*
> raise *exceptionclass [, message [stacktrace]]*

Mit Hilfe der Methode *Kernel.raise* wird eine Ausnahme ausgelöst. Die erste Variante *raise* ohne Argumente in einer *rescue* Klausel wirft die letzte Ausnahme nochmals. Falls dies außerhalb einer *rescue* Klausel geschieht, wird eine nachrichtenlose *RuntimeError* Ausnahme geworfen. Falls ein *exceptionobject* verwendet wird, darf keine *message*[6] angefügt werden.

[6] Wird laut Hinweis beim Aufruf in zukünftigen Versionen möglich sein.

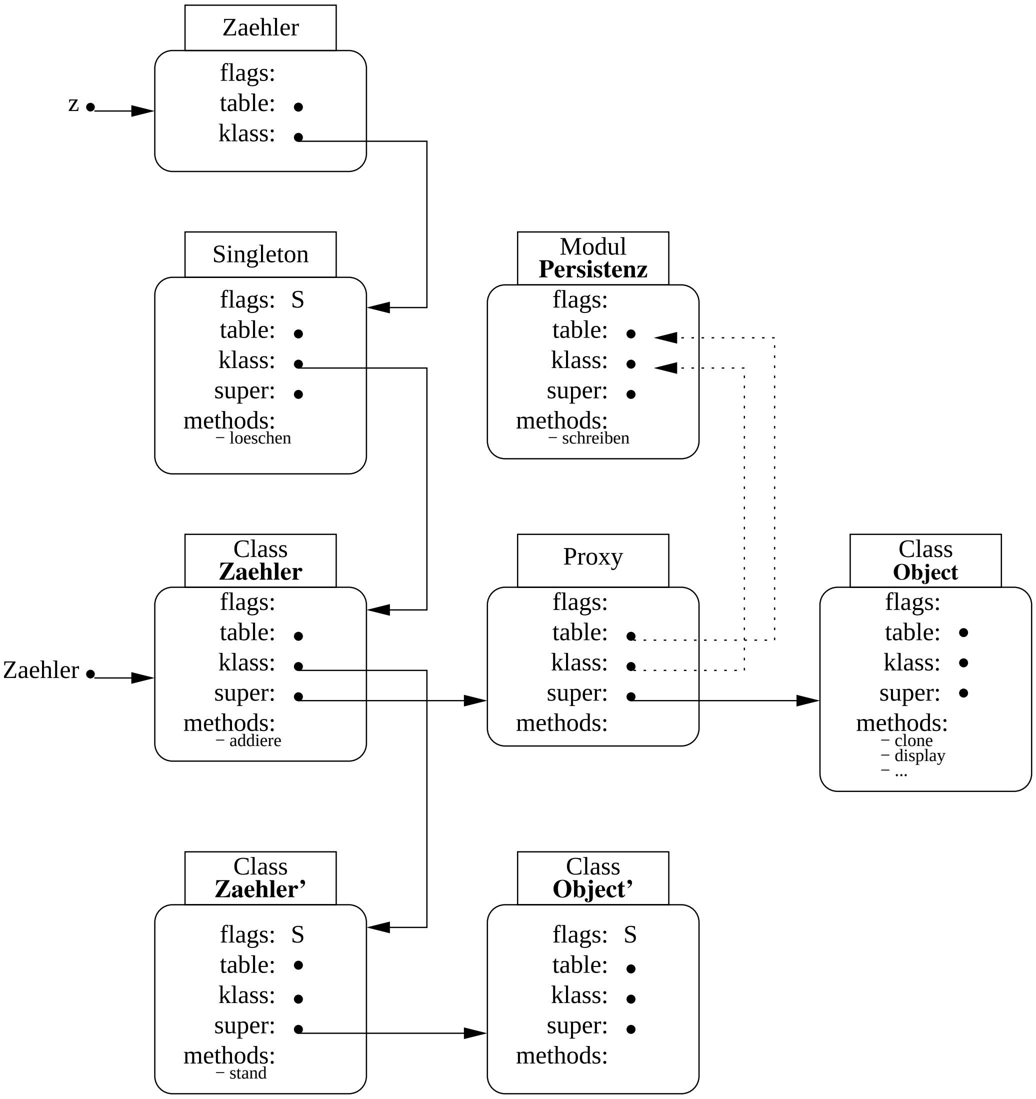


Abbildung 4.4: Ein inkludiertes Modul und seine Proxy-Klasse

Der *stacktrace* gibt an, auf welcher Ausführungsebene die Ausnahme zu werfen ist.

Ausnahmebehandlung

begin
    code ...
[rescue *[exceptionclass [, exceptionclass ...]] [=> variable] [* then *]*
    error handling code ...]
[else
    no exception code ...]
[ensure
    always execute code ...]
end

Die `begin` Anweisung umschliesst den *code* und behandelt Ausnahmen, vorausgesetzt die Klauseln *rescue* und *ensure* wurden spezifiziert.
Wenn eine *rescue* Klausel angegeben ist, werden Ausnahmen, die in den *exceptionclasses* spezifiziert sind, abgefangen und der *error handling code* wird ausgeführt. Der Rückgabewert der gesamten `begin` Umhüllung entspricht dem Wert der letzten ausgeführten Zeile, unbeachtet des *ensure*-Blocks. Je nach Situation ist dies die letzte Zeile vom *error handling code* oder *no exception code*. Wenn keine *exceptionclass* angegeben ist, so wird implizit *StandardError* angenommen. Wenn eine *variable* spezifiziert wurde, wird in ihr das Ausnahmeobjekt gespeichert. Die *rescue*-Klausel ist entweder durch das reservierte Wort *then*, eine Neuzeile oder durch einen Strichpunkt vom Rest des Codes getrennt. Falls keine Ausnahmen auftreten, wird, wenn vorhanden, die *else*-Klausel ausgeführt. Wenn eine *ensure*-Klausel spezifiziert ist, wird immer der *always execute code* ausgeführt, bevor der `begin/end` Block endet. Dies ist auch dann der Fall, wenn der Block auf Grund einer Ausnahme, die nicht in der *rescue*-Klausel angegeben ist, vorzeitig terminiert.

code rescue *expression*

Diese Kurzform evaluiert *expression*, wenn eine Ausnahme im *code* auftritt. Die abzufangende Ausnahme muss eine Subklasse von *StandardError* sein.

```
a = false
f = begin
      raise StandardError, "Provozierter Fehler!"
      rescue => error
        "Fehlermeldung: " + error
      else
        "Unerreichbar"
      ensure
        a = true
    end
p a, f       # true, "Fehlermeldung: Provozierter Fehler!"

8 / 0 rescue p "Division durch Null!"
```

retry

Wenn die *retry*-Anweisung innerhalb einer *rescue*-Klausel aufgerufen wird, so springt diese zum Anfang des `begin/end` Blocks. Andernfalls wiederholt die *retry*-Anweisung den Methodenaufruf des assoziierten Blocks.

Kapitel 5

Anwendung des objektorientierten Paradigmas

In [Wegner, 1990] findet sich eine ausführliche Diskussion über die vielen Aspekte der objektorientierten Programmierung. Dieses Kapitel untersucht die wichtigsten objektorientierten Konzepte in Java und Ruby auf Anwendbarkeit. Zu diesen zählen die einfache Vererbung, das Ersetzbarkeitsprinzip, mehrfache Vererbung, Delegation, Informationsverheimlichung, abstrakte Klassen, Objektspezialisierung, Finalisierung, Überladen, Überschreiben und Design by Contract.

Zu Beginn jedes Abschnitts wird das Konzept erläutert und anhand von schrittweisen Ansätzen und in Beispielen versucht, bestmöglich in Java und Ruby umzusetzen.
Ein Fazit am Ende jeden Abschnitts fasst die dabei gewonnenen Erkenntnisse zusammen.

5.1 Einfache Vererbung (single inheritance)

Die Vererbung stellt ein essentielles Konzept in der objektorientierten Programmierung dar. Mit Hilfe der Vererbung ist es möglich, eine Klasse (Subklasse) zu erstellen, die eine Verfeinerung oder Spezialisierung einer bestehenden Klasse (Superklasse) darstellt. Dieses Prinzip ermöglicht die direkte Wiederverwendung von Codeteilen in der Subklasse. Diese Eigenschaft begünstigt auch die Wartbarkeit des Programms, da der Code nur an einer Stelle im Programm geändert werden muss. Beim Entwurf des Klassenmodells sollte auch das Ersetzbarkeitsprinzip (siehe Abschnitt 5.2, Seite 74) gewährleistet sein.
In [Halbert et al., 1987] werden Typen als die wichtigsten Bestandteile der objektorientierten Programmierung gesehen und die Vererbung als Beziehung zwischen ihnen.

Im folgenden Java- und Ruby-Beispiel wird eine Superklasse Person mit den Attributen name, alter und geschlecht und deren Subklasse Student mit den Attributen matrnr (Matrikelnummer) und kennz (Studienkennzahl) definiert. Sowohl in Java als auch in Ruby ist die Klasse Person implizit von der Wurzelklasse Object abgeleitet.

Beispiel 5.1.1 *[Einfache Vererbung in Java]*

```
class Person {
  private String name;
```

```
  private int alter;
  private char geschlecht;

  Person(String name, int alter, char geschlecht) {
    this.name = name;
    this.alter = alter;
    this.geschlecht = geschlecht;
  }
}

class Student extends Person {
  private String matrnr;
  private String kennz;

  Student(String name, int alter, char geschlecht,
          String matrnr, String kennz) {
    super(name, alter, geschlecht);
    this.matrnr = matrnr;
    this.kennz = kennz;
  }
}

// Erstelle zwei Objekte
Person a = new Person("Max Mustermann", 38, 'm');
Student b = new Student("Kurt Niemand", 21, 'm', "1234567", "E881");
```

Beispiel 5.1.2 *[Einfache Vererbung in Ruby]*

```
class Person
  def initialize(name, alter, geschlecht)
    @name, @alter, @geschlecht = name, alter, geschlecht
  end
end

class Student < Person
  def initialize(name, alter, geschlecht, matrnr, kennz)
    super(name, alter, geschlecht)
    @matrnr = matrnr
    @kennz = kennz
  end
end

# Erstelle zwei Objekte
a = Person.new("Max Mustermann", 38, "m")
b = Student.new("Kurt Niemand", 21, "m", "1234567", "E881")
```

Beim direkten Vergleich dieser Beispiele ist sofort der Unterschied zwischen dem statischen Typkonzept in Java und dem dynamischen Typkonzept in Ruby erkennbar. In Ruby wurde zum Beispiel für die Variablen a und b vorab kein Typ deklariert.
Im Java-Beispiel lautet der Name des Konstruktors gleich dem Namen der Klasse und in Ruby wird der Konstruktor mit dem Namen initialize definiert. Bei beiden Sprachen wird durch die Instanzierung der Objekte a und b der Konstruktor ausgeführt. Im Ruby-Beispiel hat die Methode initialize in der Klasse Person drei Parameter. Diese Parameter agieren wie lokale Variablen innerhalb einer Methode, wodurch sie der Konvention der lokalen Variablennamen, beginnend mit einem Kleinbuchstaben, entsprechen. Die Konvention der Instanzvariablen in Ruby verlangt, dass vor dem Variablennamen das Zeichen „@“ steht.
Zum Beispiel in Java sei noch bemerkt, dass der Compiler auch ohne dem voranstehenden Schlüsselwort *this* und dem nachfolgenden Punkt die Instanzvariablen (z.B. name = name;) korrekt initialisieren kann, dies hier aber zur besseren Lesbarkeit unterlassen wurde.

Damit Student eine Subklasse von Person wird, muss bei der Klassendefinition im Falle von Java das Schlüsselwort extends und im Falle von Ruby das einfache Symbol „<“ jeweils gefolgt vom Namen der Superklasse (Person), verwendet werden. Der Konstruktor der Subklasse Student ruft in beiden Sprachen mit Hilfe des Schlüsselwortes *super* und den drei Parametern name, alter und geschlecht den Konstruktor der Superklasse Person auf.

5.1.1 Vererbungshierarchie

Dem Programmierer wird durch die direkte Codewiederverwendung bei der Vererbung nicht nur das wiederholte Schreiben desselben Codes erspart, sondern auch die Wartbarkeit wird dadurch begünstigt. Obwohl der Code an mehreren Stellen wiederverwendet wird, muss eine Änderung nur an einer Stelle vorgenommen werden. Damit beim Überschreiben von Methoden auf die Oberklasse zugegriffen werden kann, steht in Java und Ruby das Schlüsselwort *super* zur Verfügung.

In beiden Sprachen ist es nicht möglich, mit *super* über mehrere Vererbungsebenen hinweg auf eine Methode der Oberklasse zuzugreifen, was eine gravierende Einschränkung bei komplexen Vererbungshierarchien darstellt.
Eine Möglichkeit, diese Einschränkung in Java und Ruby zu umgehen, wird nun untersucht:

Beispiel 5.1.3 *[Schlüsselwort super in Java]*

```
class A {
  public void foo() {
    System.out.println("A");
  }
}

class B extends A {
  public void foo() {
    System.out.println("B");
```

```
  }

  protected void orig_foo() {
    super.foo();
  }
}

class C extends B {
  public void foo() {
    orig_foo();
  }
}

class MySuper {
  public static void main(String args[]) {
    A a = new A();
    B b = new B();
    C c = new C();
    a.foo();          // "A"
    b.foo();          // "B"
    c.foo();          // "A"
    b.orig_foo();     // "A" nur im selben Paket moeglich
    ((A) b).foo();    // "B"
  }
}
```

Im oberen Beispiel definiert die Klasse A eine allgemein verfügbare Methode foo(). Diese Methode wird in der direkten Subklasse B überschrieben. Die protected Methode orig_foo() in der Klasse B ermöglicht allen Klassen im selben Paket die ursprüngliche Methode foo() der Klasse A weiterhin zu verwenden. Dies ist jedoch nur mit dem Nachteil möglich, dass der Name der ursprünglichen Methode geändert werden muss, in diesem Beispiel orig_foo(). Durch die Verwendung des Schlüsselwortes *super*, das eine Referenz auf die Instanz der Superklasse darstellt, gefolgt von einem Punkt und dem Methodennamen foo() wird bewirkt, dass der Code der Methode foo() in der Klasse A ausgeführt wird. Die Subklasse C, die ein Nachfolger der Klasse B darstellt, bewirkt durch ein neuerliches Überschreiben eine Rückbenennung auf foo(), womit sich diese Methode foo() genauso verhält wie die ursprüngliche Methode foo() in der Klasse A. Das Verhalten der Methode foo() konnte in diesem Beispiel also in der Zwischenklasse B unterbrochen werden.

Beispiel 5.1.4 *[Ausführungsanweisung super in Ruby]*

```
class A
  def foo
    puts "A"
  end
end
```

```
class B < A
  alias orig_foo foo

  def foo
    puts "B"
  end

  protected :orig_foo
end

class C < B
  alias foo orig_foo
  public :foo
end

a = A.new
b = B.new
c = C.new
a.foo              # "A"
b.foo              # "B"
#b.orig_foo        # nicht moeglich
c.foo              # "A"
#c.orig_foo        # nicht moeglich
```

In Ruby ist *super* keine Referenz auf eine Elternklasse, sondern eine Ausführungsanweisung (execute-Statement), welche die momentane Methode nochmals aufruft, wobei die Suche nach einer entsprechenden Methode in der Superklasse begonnen wird, siehe Abschnitt 4.3.5, Seite 51. Um ein ähnliches Verhalten wie im oberen Java-Programm zu ermöglichen, wird stattdessen das Schlüsselwort *alias* verwendet. Mit Hilfe von *alias* bleibt der Zugriff auf die ursprüngliche Methode foo() unter dem neuen Namen orig_foo() erhalten. Die Sichtbarkeit dieser Methode wird durch das Schlüsselwort *protected* auf Subklassen wie C eingeschränkt. In der Subklasse kann wiederum mit *alias* eine Rückbenennung durchgeführt werden. Ein Zugriff von außen auf die Methode von foo() in der Klasse C ist nur dann möglich, wenn auch der Sichtbarkeitsbereich explizit auf *public* gesetzt wird. Dies ist aufgrund der getätigten Zugriffskontrolle (siehe Abschnitt 4.3.8, Seite 56) in der Elternklasse B notwendig. Die unter Kommentar gesetzten Methodenaufrufe sind daher in diesem Beispiel auf Grund der Sichteinschränkung nicht durchführbar.

Eine weitere Möglichkeit, diese Problematik zu lösen, besteht mit dem instanceof-Operator in Java, oder wie im folgenden Beispiel gezeigt wird, mit der gleichwertigen instance_of?-Methode in Ruby:

Beispiel 5.1.5 *[instance_of? in Ruby]*

```
class A
  def foo
    puts "A"
```

```
  end
end

class B < A
  def foo
    if instance_of?(B)
      puts "B"
    else
      super
    end
  end
end

class C < B
end

a = A.new
b = B.new
c = C.new
a.foo                # "A"
b.foo                # "B"
#b.orig_foo          # nicht moeglich
c.foo                # "A"
#c.orig_foo          # nicht moeglich
```

Der Vorteil an dieser Variante liegt darin, dass nur die Klasse B eine Entscheidung auf den zu verwendenden Codeteil treffen muss. Die Subklasse C muss im Gegensatz zur ersten Variante keine Umbenennungen oder Sichtbarkeiten rückgängig machen. Als Nachteil muss erwähnt werden, dass standardmäßig das Verhalten der Methode foo() in der Subklasse C nicht dem Verhalten der Methode foo() in der Elternklasse B entspricht und die Subklasse C somit nicht wirklich dem objektorientierten is-a Prinzip folgt.

In Java kann im Gegensatz zu Ruby mit *super* Bezug auf eine Konstante, Instanzvariable oder Klassenvariable vorgenommen werden. Mit Hilfe der expliziten Typumwandlung können in Java auch Namen angesprochen werden, die bereits weiter oben in der Klassenhierarchie verdeckt wurden. Im folgenden Beispiel gibt die Methode foo() in der Klasse C mit super.a den Wert x aus der Klasse B und mit ((A)this).a den Wert a aus der in der Klassenhierarchie weiter oben definierten Klasse A aus. Es sei noch angemerkt, dass die Typumwandlung nicht auf überschriebene Instanzmethoden angewendet werden kann, da beim dynamischen Binden der deklarierte Typ keinen Einfluss auf die auszuführende Methode hat. Sowohl in Java als auch in Ruby können statische Variablen und Methoden durch Voranstellen des Klassennamens – unbeachtet der Vererbungshierarchie – angesprochen werden.

Beispiel 5.1.6 *[In Java mit super auf Variablen zugreifen]*

```
class A {
```

```
  char a='a';
  final char f='f';
  static char s='s';
  final String ss="SS";
}

class B extends A {
  char a='x';
  final char f='x';
  static char s='x';
  final String ss="xx";

  public void foo() {
    System.out.println(super.a);
    System.out.println(super.f);
    System.out.println(super.s);   // Zugriff auch mit A.s moeglich
    System.out.println(super.ss);
  }
}

class C extends B {
  public void foo() {
    System.out.println(super.a);
    System.out.println(((A)this).a);
  }
}

class MySuper3 {
  public static void main(String args[]) {
    // Erstelle drei Objekte
    B b = new B();
    b.foo();                  // a,f,s,SS
    C c = new C();
    c.foo();                  // x,a
  }
}
```

Die alternierende Verwendung von Variablen aus der Superklasse und der aktuellen Klasse ist in Ruby nicht möglich. In Ruby sind Klassen standardmäßig nicht geschlossen[1] und können daher dynamisch modifiziert werden. Durch Aufruf von remove_method(name) in einer Subklasse kann die Methode mit der Bezeichnung name aus der Superklasse sichtbar gemacht werden. Wird die Methode undef_method(name) in einer Klasse oder Modul verwendet, so werden alle Methoden mit der Bezeichnung name, auch die in den Superklassen, für diese Klasse bzw. Modul unsichtbar. Obwohl es in Ruby die Methode remove_instance_variable(name) (eine private Instanzmethode von Kernel) zum Entfernen von

[1] Mit der Methode freeze kann in Ruby verhindert werden, dass Objekte verändert werden.

Instanzvariablen mit der Bezeichnung name gibt, hat sie nicht die gleiche Eigenschaft wie die Methode remove_method, die eine private Instanzmethode von Modul ist [Matz, 2002]. Mit remove_instance_variable wird die Instanzvariable vollständig entfernt und somit keine Sicht auf Superklassen gewährt.

Beispiel 5.1.7 *[Methoden undef und remove in Ruby]*

```
class A
  def initialize
    @x="A"
  end

  def foo
    puts "A " + ((defined? @x) == nil ? "nil" : @x)
  end
end

class B < A
  def initialize
    @x="B"
  end

  def foo
    puts "B " + ((defined? @x) == nil ? "nil" : @x)
  end
end

b = B.new
b.foo                  # "B B"

class B
  remove_method :foo    # nur von Klasse B entfernt

  def remove_x
    remove_instance_variable("@x")
  end
end

b.foo                  # "A B"
b.remove_x             # entfernt Instanzvariable x
b.foo                  # "A nil"

class B
  undef_method :foo     # entfernt alle Methoden foo
end

b.foo                  # Fehler: undefined method foo
```

In diesem Beispiel wird eine Instanz der Subklasse B der Variablen b zugeordnet und die Methode foo aufgerufen. Diese gibt zu diesem Zeitpunkt die Werte von der Klasse B aus. Danach wird die Subklasse B so modifiziert, dass die Methode foo entfernt und eine neue Methode mit der Bezeichnung remove_x, die die Instanzvariable x entfernt, definiert wird. Ein neuerlicher Aufruf der Methode foo zeigt, dass jetzt die Methode aus der Superklasse ausgeführt wurde. Nachdem auch die Instanzvariable x entfernt wurde, zeigt die nächste Ausgabe von foo, dass diese nicht mehr vorhanden ist. Zuletzt wird noch die Subklasse B dahingehend verändert, dass auch die Methode foo vollständig entfernt wird, wie ein neuerlicher Aufruf zeigt. Auf ähnliche Art und Weise können auch Klassendefinitionen, die in Ruby Konstanten darstellen, mit der Methode remove_const unsichtbar gemacht werden. Ein Anwendungsfall wäre zum Beispiel, die weitere Verwendung von gefährlichen Klassen zu unterbinden.

Beispiel 5.1.8 *[Klassen mit undef in Ruby entfernen]*

```
class A
  def foo
    puts "A"
  end
end

a = A.new

class Object
  remove_const :A
end

# A kann nicht mehr instanziert werden
#x = A.new

# die Instanz a ist weiterhin verfuegbar
a.foo
```

Wie man an diesem Beispiel sieht, kann diese Klasse nach ihrer Entfernung zwar nicht mehr instanziert werden, bestehende Instanzen stehen aber weiterhin zur Verfügung. Es ist offensichtlich, dass die Verwendung von remove bzw. undef Methoden beim Programmieren zu unangenehmen Seiteneffekten führen können, deren Auffinden nur schwer möglich ist. Die Verwendung dieser Methoden kann mit einem durchdachten Klassendesign, in dem entsprechende Namenskonventionen und Zugriffseinschränkungen vorab spezifiziert werden, vermieden werden.

5.1.2 Fazit

Sowohl in Java als auch in Ruby kann einfache Vererbung auf Klassen angewendet werden. Beide Sprachen unterstützen beim Überschreiben von Methoden auch den Zugriff auf die direkte Superklasse, jedoch nicht auf höhere Superklassen. Java bietet mit dem Schlüsselwort *super* auch den Zugriff auf Konstanten bzw. Variablen an.

Klassen sind in Ruby standardmäßig nicht geschlossen, wodurch diese dynamisch verändert werden können. Solche Veränderungen können aber leicht zu Seiteneffekten führen, die im Fehlerfalle nur schwer nachvollziehbar sind. Prinzipiell ist der Unterschied bezüglich einfacher Vererbung zwischen den beiden Programmiersprachen als gering anzusehen.

5.2 Ersetzbarkeitsprinzip (subtyping)

Die Wiederverwendbarkeit des Programmcodes ist sowohl für den Implementierer als auch für den Benutzer von großer Wichtigkeit ([LaLonde et al., 1991], [Porter, 1992]), jedoch in unterschiedlicher Art und Weise. Der Implementierer strebt nach der Möglichkeit, weitere Verhaltensweisen auf Basis des bestehenden Codes einfach hinzuzufügen (subclassing). Anders der Benutzer, dieser versucht existierende Codeteile so vielseitig wie möglich einzusetzen (subtyping).

- *Subclassing* – auch Vererbung genannt – ist ein Mechanismus für die Implementierung von gemeinsam benutztem Code zwischen Klassen.
- *Subtyping* – auch enthaltener Polymorphismus genannt – ist ein Ersetzbarkeitsprinzip. Die Implementierung hat bei subtyping keine Bedeutung. Hier spielt die Kompatibilität des Verhaltens vom Subtyp zum Supertyp eine wichtige Rolle.

Die Definition des Ersetzbarkeitsprinzips[2] aus [Liskov, 1988] lautet:

> If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T.

Ist eine Instanz von S (S ist Untertyp eines Typs T) überall verwendbar, wo eine Instanz von T erwartet wird, so ist das Ersetzbarkeitsprinzip gewährleistet. In den Untertypen[3] können die deklarierten Typen der einzelnen Elemente variieren. Dabei werden die folgenden drei Varianten unterschieden [Abardi et al., 1996]:

Kovarianz: Der deklarierte Typ eines Elements im Untertyp ist ein Untertyp des deklarierten Typs des entsprechenden Elements im Obertyp.

Kontravarianz: Der deklarierte Typ eines Elements im Untertyp ist ein Obertyp des deklarierten Typs des Elements im Obertyp.

Invarianz: Der deklarierte Typ eines Elements im Untertyp ist gleich dem deklarierten Typ des entsprechenden Elements im Obertyp.

Generell müssen die folgenden Bedingungen für eine Untertypbeziehung gegeben sein, damit das Ersetzbarkeitsprinzip erfüllt ist:

- Die deklarierte Typen von Konstanten und Ausgangsparametern müssen kovariant sein.
- Die deklarierte Typen von Variablen und Durchgangsparametern müssen invariant sein.

[2]LSP: Liskov Substitution Principle

[3]Ein Typ ist implizit ein Untertyp von sich selbst.

- Die deklarierte Typen von formalen Eingangsparametern müssen kontravariant sein.

Eine fundamentale Entwurfsregel besagt: Wenn ein Anwender eine Variable (Parameter, lokale Variable, Returnwert, etc.) deklariert, so sollte dieser immer anhand des Typs die Variable deklarieren und nicht anhand der Klasse. Die einzige Stelle, an der ein Anwender einen Klassennamen verwenden muss, ist bei der Instanzierung [D'Souza, 1997].

Oft ist zum Compilierungszeitpunkt der dynamische Typ, der sich vom statischen Typ unterscheiden kann, nicht bekannt. Die richtige Auswahl der Methode kann somit nur während der Programmausführung ermittelt werden (dynamisches Binden). Sowohl Java als auch Ruby sind im Zusammenhang mit dem Ersetzbarkeitsprinzip auf das dynamische Binden angewiesen.

5.2.1 Ersetzbarkeit in Java

In Java stellen die Klassen bzw. Interfaces Typen dar, bei denen Untertypbeziehungen mit Hilfe der einfachen Vererbung realisiert werden können. Da mit den primitiven Datentypen (siehe Tabelle 3.1, Seite 10) keine Vererbungsbeziehungen möglich sind, können von ihnen auch keine Untertypen gebildet werden.

Beispiel 5.2.1 *[Überladen statt ersetzen in Java]*

```
class A {
  public A foo(B p) { ... }
}

class B extends A {
  public B foo(A p) { ... }
}
```

Dieser Code entspricht zwar den allgemeinen Bedingungen für eine Untertypbeziehung, jedoch überschreibt die Methode foo() in der Subklasse B die gleichnamige Methode aus der Superklasse A nicht. Grund dieser Ursache ist das Konzept der Überladung in Java. Die Überladung verhindert, dass die Methode foo() in der Subklasse B überschrieben wird, stattdessen stehen in der Subklasse B beide Methoden nebeneinander zur Verfügung (siehe Abschnitt 5.8.1, Seite 110).

In Java sind Untertypbeziehungen wesentlich stärker eingeschränkt, als es durch die obigen allgemeinen Bedingungen notwendig wäre. Es gibt keine kovarianten und kontravarianten Typen – alle Typen sind invariant.

Beispiel 5.2.2 *[Ersetzbarkeitsprinzip in Java]*

```
class A {
  public Integer foo(A p) { ... }
}
```

```
class B extends A {
  public Integer foo(A p) { ... }
}
```

In diesem Beispiel sind sowohl der Parametertyp, als auch der Ausgangsparameter invariant, womit die Methode foo() in der Subklasse B überschrieben wird. Eine Erklärung, warum Konstanten invariant statt kovariant sein müssen, ist vermutlich die Spracheigenschaft, dass Konstanten spezielle Variablen darstellen und diese ohnehin invariant sein müssen [Puntigam, 2002]. Auch für die formalen Parameter, die durch das Konzept des Überladens nur schwer von Methoden mit kontravariant veränderten Typen auseinanderzuhalten sind, ist eine plausible Erklärung gegeben. Die einzige nicht erklärbare Tatsache in Java ist, dass Ausgangsparameter nicht kovariant[4] sein dürfen.

5.2.2 Uneingeschränkte Ersetzbarkeit in Java

Vom Java-Compiler wird überprüft, ob die Bedingungen für einen Untertyp eines anderen Typs gegeben sind. Daneben können aber noch weitere Bedingungen notwendig sein, um die uneingeschränkte Ersetzbarkeit einer Instanz eines Obertyps durch eine Instanz eines Untertyps zu gewährleisten. Dazu folgendes Beispiel:

Beispiel 5.2.3 *[Uneingeschränkte Ersetzbarkeit in Java, Teil 1]*

```
class List {
  // fuegt ein Element an das Ende der Liste hinzu
  public void addElement(Object element) { ... }

  // retourniert das i-te Element aus der Liste
  public Object getElement(int i) { ... }

  // retourniert true, wenn Element in der Liste enthalten ist
  public boolean hasElement(Object element) { ... }

  // retourniert die Anzahl der Elemente in der Liste
  public int size() { ... }

  // entfernt alle Elemente aus der Liste
  public void clear() { ... }

  // kopiert den Inhalt dieser Liste in die als
  // Parameter uebergebene Liste, wobei diese zuvor
  // geloescht wird;
  public void copy(List other) {
    other.clear();
    for (int i = 0; i < size(); ++i) {
      other.addElement(getElement(i));
```

[4] In der nächsten Java-Version 1.5 ist diese Einschränkung vermutlich nicht mehr gegeben.

```
    }
  }
}
```

Die Klasse List stellt mehrere Methoden für die Handhabung einer Liste mit Elementen vom Typ *Object* zur Verfügung. Unter anderem auch die Methode *copy()*, von der jedes Listenelement in die als Argument übergebene Liste, deren Elemente zuvor noch gelöscht werden, kopiert wird.
Zu einem späteren Zeitpunkt erfolgt nun der Bedarf nach einer Liste, die nur eindeutige Elemente aufnehmen kann:

Beispiel 5.2.4 *[Uneingeschränkte Ersetzbarkeit in Java, Teil 2]*

```
class UniqueList extends List {
  . . .
  public void addElement(Object element) {
    if (!hasElement(element)) {
      super.addElement(element);
    }
  }
  . . .
}
```

Hierbei wird die Methode *addElement()*, nach obigen Bedingungen überschrieben. Sowohl der Typ des formalen Parameters, als auch der Typ des Ausgangsparameters sind – wie in Java gefordert – invariant.

Trotzdem ist die Verwendung dieser Typen nicht uneingeschränkt möglich. Die Methode *copy()* arbeitet korrekt, wenn eine List in eine andere List, eine UniqueList in eine andere UniqueList oder eine UniqueList in eine List kopiert wird. Jedoch können Probleme auftreten, wenn eine List in eine UniqueList kopiert wird, deren Elemente zum Beispiel $\{0, 1, 3, 1\}$ sind. Durch die Methode *copy()* werden nur die Elemente $\{0, 1, 3\}$ in die UniqueList kopiert. In diesem Fall ist UniqueList keine Kopie von List und nicht uneingeschränkt ersetzbar. Dadurch, dass die Methode *addElement()* in der Subklasse UniqueList überschrieben wurde, hat sich auch das Verhalten der Methode *copy()* in der Klasse List verändert.

Die uneingeschränkte Ersetzbarkeit einer Instanz eines Obertyps durch eine Instanz eines Untertyps bedarf in Java neben dem Ersetzbarkeitsprinzip weiterer Maßnahmen. Neben einer wohlüberlegten Klassenhierarchie, die das Verhalten der Klasse in den Vordergrund stellt und nicht die Daten, bietet auch *Design by Contract* (siehe Abschnitt 5.9, Seite 117) hierfür Unterstützung an.

5.2.3 Keine Typen in Ruby

Auf Grund des dynamischen Typkonzeptes ist es in Ruby eigentlich nicht möglich, das Erfülltsein des Prinzips der Ersetzbarkeit statisch zu garantieren. Eine Variable in Ruby kann jeden Typ aufnehmen und ist somit keiner Einschränkung unterworfen.

Beispiel 5.2.5 *[Keine Typen in Ruby]*

```
class A
  Value = 2

  def more(parm)
    parm + Value
  end
end

class B < A
  def more(parm)
    parm * Value
  end
end

a = A.new
b = B.new
p a.more(8)     # 10
p b.more(8)     # 16
p b.more("A")   # "AA"  nicht erkannter Typfehler
p a.more("A")   # TypeError: cannot convert Fixnum into String
```

Die Subklasse B überschreibt die Methode *more* von der Superklasse A. Der Name des Parameters *parm* sagt nichts über den erwarteten Typ aus. In der Subklasse B wurde die Methode *more* so überschrieben, dass statt der Methode `+` nun die Methode `*` zur Anwendung kommt. Dies hat in Ruby den Effekt, dass neben den numerischen Klassen (Fixnum, Float, etc.) nun auch die Klasse String zum Einsatz kommen kann. Die Methode `+` führt standardmäßig entweder die Addition zweier numerischer Werte oder die Verknüpfung zweier Zeichenketten durch, wodurch in der Superklasse A nur ersteres möglich ist, da als zweiter Parameter bereits eine Zahl feststeht. Anders die Methode `*`, sie akzeptiert zusätzlich noch als ersten Operanden einen String und als zweiten Operanden einen numerischen Wert. Je nach Ausprägung des numerischen Wertes wird der String auf Grund dieser Methode mehrmals mit sich selbst verknüpft. Strings stehen aber in keiner Beziehung zu Fixnum oder dergleichen, wodurch die Variable *parm* mit deren aufnehmbaren Typen in der Subklasse B nicht kontravariant ist. Dementsprechend liefert die Ausgabe von `a.more("A")`, im Gegensatz zu `b.more("A")`, einen Typfehler.

5.2.4 Typdefinitionen mit Kommentaren in Ruby

In diesem Falle obliegt es dem Ersteller einer Klasse, entsprechende Kommentare zu verfassen bzw. einer Variable einen dem Typ entsprechenden Namen zu geben und dem Anwender, diese Hinweise betreffend Ersetzbarkeitsprinzip zu befolgen. Vom Compiler wird diesbezüglich keine Unterstützung geboten.

Beispiel 5.2.6 *[Kommentare in Ruby]*

```
class A
  FixnumStep = 2

  def more(fixnumParm)
    # Eingangsparameter: fixnumParm - Typ Fixnum
    # Ausgangsparameter: Typ String
    "Ergebnis: " + (fixnumParm + FixnumStep).to_s
  end
end
. . .
```

Diese Lösung erhöht zwar die Lesbarkeit, erfordert aber mehr Schreibaufwand für den Entwickler. Damit es nicht zu Divergenzen zwischen der Codierung und den Kommentaren kommt, wären bei dieser Lösung auch noch Codeinspektionen mit Schwerpunkt auf konvergenten Code/Kommentar-Verlauf durchzuführen.

5.2.5 Kontravarianz in Ruby

Eine weitere Möglichkeit stellt die vom Ersteller zu codierende Typüberprüfung mit der instance_of? Methode dar, um den Typ des Eingangsparameters – analog zu Java – auf Invarianz zu überprüfen. Da in Ruby Methoden nur überschrieben werden können (siehe Abschnitt 5.8.2, Seite 112), bietet sich jedoch die Möglichkeit an, die Typüberprüfung des Eingangsparameters gleich auf Kontravarianz und die des Ausgangsparameters auf Kovarianz durchzuführen.

Beispiel 5.2.7 *[Kovarianz und Kontravarianz in Ruby]*

```
class Object
  def covariance?(type)
    return if self.kind_of?(type)
    raise ArgumentError, self.class.to_s +
          " ist kein kovarianter Typ von " + type.to_s
  end

  def contravariance?(type)
    t = type
    while t
      return if self.class == t
      t = t.superclass
    end
    raise ArgumentError, self.class.to_s +
          " ist kein kontravarianter Typ von " + type.to_s
  end
end
```

```
class Ergebnis
#class Ergebnis < String
   def initialize(s)
     @s = "Ergebnis: " + s.to_s
   end
  def inspect
    @s
  end
end

class A
  FixnumStep = 2

  def more(fixnumParm)
    # Eingangsparameter: fixnumParm - Typ Fixnum
    # Ausgangsparameter: Typ String
    fixnumParm.contravariance?(Fixnum)
    retvalue = "Ergebnis: " + (fixnumParm + FixnumStep).to_s
    retvalue.covariance?(String)
    retvalue
  end
end

class B < A
  def more(fixnumParm)
    # Eingangsparameter: fixnumParm - Typ Fixnum
    # Ausgangsparameter: Typ Ergebnis
    fixnumParm.contravariance?(Fixnum)
    retvalue = Ergebnis.new(fixnumParm * FixnumStep)
    retvalue.covariance?(String)
    retvalue     # Ergebnis kein kovarianter Typ von String
  end
end

a = A.new
b = B.new
p a.more(8)      # "Ergebnis: 10"
p b.more(8)      # "Ergebnis: 16"
p b.more("A")    # String ist kein kontravarianter Typ von Fixnum
p a.more("A")    # String ist kein kontravarianter Typ von Fixnum
```

In diesem Beispiel wurden die beiden Methoden *contravariance?* und *covariance?* in der Klasse *Object* zur Überprüfung des Ersetzbarkeitsprinzips in Ruby definiert. Ruby bietet mit der Kernel-Methode *kind_of?(type)*[5] die Möglichkeit zu prüfen, ob das Objekt eine

[5] Alternativ könnte auch die Methode is_a? verwendet werden.

Instanz von type oder eines seiner Nachfolgers ist – entsprechend der Definition von Kovarianz. Für die Kontravarianz muss in einer Schleife und mit den beiden Methoden *class* und *superclass* die Hierarchie der Klasse solange nach oben durchlaufen werden, bis der gesuchte Typ oder *nil* gefunden wird.
Wenn in diesem Beispiel die Klasse Ergebnis von der Klasse String abgeleitet wird (Austausch der Zeile `class Ergebnis` durch die nachfolgende unter Kommentar befindliche Zeile), entspricht die Methode *more* in der Subklasse B dem Ersetzbarkeitsprinzip, da die Eingangsparameter kontravariant und die Ausgangsparameter kovariant wären. Auch der Typfehler durch die Anweisung `b.more("A")` würde in diesem Falle erkannt werden.
Neben der aufwändigen Überprüfung zur Laufzeit ist auch der Codieraufwand in den Klassen A und B nicht unerheblich, wodurch neue Programmfehler auftreten können.

5.2.6 Typüberprüfung mit types.rb

Für Ruby gibt es das Modul *types.rb*, mit dem der Typ der Eingangsparameter von Methoden überprüft werden kann. In *types.rb* werden Typüberprüfungen nur auf Kovarianz und Invarianz durchgeführt. Die für das Ersetzbarkeitsprinzip notwendige Kontravarianz ist im Modul (derzeit) nicht enthalten. Folgendes Beispiel zeigt die Verwendung von *types.rb*[6].

Beispiel 5.2.8 *[types.rb in Ruby]*

```
require "types.rb"
include TypeChecker

class Ergebnis < String
end

class A
  typesig String
  def foo(stringParm)
    stringParm
  end

  typesig :exact, String
  def bar(stringParm)
    stringParm
  end

  typesig :rest, Type::Multi(String, Integer)
  def baz(*stringParms)
    stringParms
  end
end

p A.new.foo(Ergebnis.new("Peter"))
p A.new.bar("Peter")
```

[6]Die Version 1.1 von http://people.freebsd.org/~eivind/ruby/types/ verwendet die abgelehnte (deprecated) Kernelmethode *Object#type* statt *Object#class*.

```
#p A.new.bar(Ergebnis.new("Peter"))   # Arg 0 is of invalid type ...
p A.new.baz("Quartett", 4, "Sextett", 6)
#p A.new.baz("Quartett", "Sextett")   # Arg 1 has invalid type ...
```

Nachdem das Modul *types.rb* mittels require und include zur Verfügung steht, können mit der Anweisung *typesig* vor der gewünschten Methode die Typen der Eingangsparameter definiert werden. Im Beispiel akzeptiert die Methode *foo* in der Klasse A als Eingangsparameter den Typ String und alle Subtypen von String. Vor der Methode *bar* wird der erlaubte Typ durch die Angabe von *:exact* auf den Typ String (ohne Subtypen) eingegrenzt. Mit der Angabe von *:rest* werden alle restlichen Eingangsparameter auf den danach folgenden Typ überprüft. Vor der Methode *baz* wurde bei der *typesig*-Anweisung ein Tupel von String- und Integer-Typen durch `Type::Multi(String, Integer)` definiert, die sich beliebig oft wiederholen dürfen. Die am Ende des Beispiels unter Kommentar gestellten Aufrufe stellen unerlaubte Typübergaben für die Methodenaufrufen dar.
Neben der nicht vorhandenen Überprüfung auf Kontravarianz für die Typen der Eingangsparameter ist es auch nicht möglich, Typen für Ausgangsparameter mit *types.rb* zu definieren. Dies schränkt die Verwendbarkeit dieses Moduls für das Ersetzbarkeitsprinzip in Ruby ein.

5.2.7 Fazit

Java unterstützt das Ersetzbarkeitsprinzip mit strikteren Bedingungen als es notwendig wäre. Für die Einschränkung, dass kovariante Ausgangsparameter in Java nicht möglich sind, findet sich jedoch keine plausible Erklärung. Es gibt bereits Ansätze, diese Einschränkung in einer der nächsten Java-Versionen aufzuheben, womit eine mächtigere Umsetzung des Ersetzbarkeitsprinzips in Java gegeben wäre.

Ruby bietet mit seinem dynamischen Typsystem für die Einhaltung des Ersetzbarkeitsprinzips keine in der Praxis umsetzbaren Lösungen an. Ursache hierfür ist, dass Typen bei der Programmierung in Ruby so gut wie keine Bedeutung haben. Schon die Verwendung der Anweisung *undef* kann im Zusammenhang mit dem Ersetzbarkeitsprinzip zu unerwünschten Phänomenen führen. Falls Ruby für größere bzw. komplexere Programme eingesetzt wird, müssen dafür aufwändige Verfahren wie zusätzliche Kommentare, Inspektionen, Namensgebung und Laufzeiteinbußen in Kauf genommen werden. Dem ist noch anzufügen, dass der Aufwand, um externe Programme bzw. Bibliotheken nachträglich mit dem Ersetzbarkeitsprinzip zu versehen, Dimensionen annehmen kann, die für ein Projekt nicht tragbar wären.

5.3 Mehrfachvererbung (multiple inheritance)

Obwohl einfache Vererbung ein mächtiges Instrumentarium für die Programmerstellung ist, ist es hin und wieder wünschenswert, von mehr als nur einer Klasse direkt zu erben. Wenn zum Beispiel eine Klasse für Läufer, eine für Schwimmer und eine für Radfahrer existiert, könnte man durch Mehrfachvererbung eine Klasse Triathlet generieren, die von allen drei erbt. Bei einfacher Vererbung würde man von einer dieser Klassen erben und die beiden anderen als Variablen, die Referenzen auf die Instanzen dieser Klassen sind, definieren.

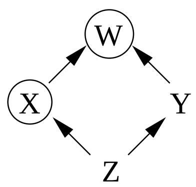
Die einfache Vererbung resultiert in eine strikte Baumhierarchie, in der jede Subklasse zu einer einzigen Superklasse in Form einer is-a Beziehung steht. Bei der Mehrfachvererbung kann eine Subklasse von mehreren Superklassen erben, womit die Klassen als ein gerichteter azyklischer Graph (**D**irected **A**cyclic **G**raph, kurz DAG) dargestellt sind. Es gibt vier wichtige Situationen, in denen Mehrfachvererbung wünschenswert ist [Singh, 1995]:

- Mehrere unabhängige Protokolle: Dies umfasst Situationen, bei der eine Klasse auf Grund mehrerer komplett unabhängiger Klassen erstellt werden soll.
- Mix-and-match: Hier werden mehrere spezifische Klassen erstellt, die später kombiniert werden. Diese speziellen Klassen sind auch als mixins bekannt.
- Submodularität: Dies umfasst Situationen bei der Systemerstellung, in denen erkannt wurde, dass die Modularität bzw. Unterteilung zweckmäßig für die gute Darstellung des Systemdesigns ist.
- Trennung von Schnittstelle und Implementierung

Diese Situationen bekräftigen die Notwendigkeit der Mehrfachvererbung. Leider wird mit der Mehrfachvererbung auch die Komplexität erhöht. Probleme wie Namenskollision, wiederholte Vererbung, Methodenzusammenführung und missbräuchliche Verwendung können dadurch auftreten. Die Implementierung der Mehrfachvererbung in einer Programmiersprache stellt auch für Sprachentwickler eine große Herausforderung dar. All dies sind Gründe, warum Mehrfachvererbung in vielen Programmiersprachen nicht oder nur eingeschränkt angeboten wird. Java und Ruby sind Programmiersprachen, die nur einfache Vererbung auf Klassen unterstützen. Diese Einschränkung wird in Java durch die Interfaces und in Ruby durch die mixins teilweise aufgehoben.

5.3.1 Interfaces in Java

Interfaces (siehe Abschnitt 3.3.4, Seite 18) definieren Typen in abstrakter Form. Sie haben keine Implementierung, womit auch keine Instanzen von ihnen erstellt werden können. Ein Interface ist ausschließlich ein Ausdruck des Designs, Klassen hingegen stellen einen Mix aus Design und Implementierung dar. Interfaces können nur von Interfaces, jedoch auch von mehreren, erben. Klassen können ihren Typ nun durch die Implementierung eines oder mehrerer Interfaces erweitern. Das sogenannte „diamond inheritance problem“ kann in Java nicht auftreten [Arnold et al., 2000]:




```
interface W { ... }
interface X extends W { ... }
class Y implements W { ... }
class Z extends Y implements X { ... }
```

Abbildung 5.1: Umgehung des „diamond inheritance problem“ in Java

Die in Z geerbten Attribute und Methoden können nur aus Y stammen und nicht indirekt auch aus W kommen. Ein weiteres Beispiel veranschaulicht die Mehrfachvererbung anhand von Interfaces in Java:

Beispiel 5.3.1 *[Interfaces in Java]*

```
interface X {
  String S = new String("S");
  void fooX();
}

interface Y {
  void fooY();
}

interface Z extends X, Y {
  void fooZ();
}

class A implements X, Y {
  public void foo() { System.out.println("A" + S); }
  public void fooX() { System.out.println("AX"); }
  public void fooY() { System.out.println("AY"); }
}

class B extends A implements Z {
  public void fooY() { System.out.println("BY"); }
  public void fooZ() { System.out.println("BZ"); }
}

class MyMI {
  public static void main(String args[]) {
    A a = new A();
    B b = new B();

    a.foo();          // "AS"
    a.fooX();         // "AX"
    a.fooY();         // "AY"

    b.foo();          // "AS"
    b.fooX();         // "AX"
    b.fooY();         // "BY"
    b.fooZ();         // "BZ"
  }
}
```

In den Interfaces sind alle Methoden implizit public und die Variablen public static und final, wodurch die Angabe entfallen kann. Das Interface Z erbt durch das Schlüsselwort *extends* von den beiden Interfaces die Methoden fooX() und fooY(), sowie den Wert der Konstante S vom Interface X. Die konkrete Klasse A implementiert die Methoden fooX()

und fooY() aus den Interfaces X und Y. Die Klasse B, die eine Subklasse von A ist, überschreibt die Methode fooY() aus der Superklasse A und implementiert das Interface Z. Als Nachteil ist sofort erkennbar, dass die Implementierung der Interfaces sehr viel redundanten Code bewirken kann. Mehrfachvererbung anhand von Interfaces ist diesbezüglich gegenüber Mehrfachvererbung auf Implementierungsebene benachteiligt.

5.3.2 Mixins

Ruby liefert ein etwas anderes Konzept bei der Mehrfachvererbung. Module bieten neben den Namensräumen auch die Möglichkeit an, mixins zu implementieren. Wie in Abschnitt 4.3.11, Seite 62 beschrieben, können Module zwar nicht instanziert werden, jedoch kann ein Modul mit *include* in eine Klassendefinition inkludiert werden. Dadurch werden alle Instanzmethoden des Moduls sofort als Methoden der Klasse verfügbar – mixed-in. Ein mixin kann als eingeschränkte Mehrfachvererbung, ohne mit den Nachteilen und Schwierigkeiten der Mehrfachvererbung konfrontiert zu sein, betrachtet werden. Matz bezeichnet dies als „single inheritance with implementation sharing“.

Beispiel 5.3.2 *[Mixins in Ruby]*

```
module X
  attr :c

  def fooX
    puts "foox: " + self.class.name + c
  end
end

module Y
  def fooY
    puts "fooy: " + self.class.name + c
  end
end

module Z
  include X, Y

  def fooZ
    puts "fooz: " + self.class.name + c
  end
end

class A
  include X, Y

  def initialize
    @c = "S"
  end
```

```
  def foo
    puts "foo: " + self.class.name + c
  end
end

class B < A
  include Z
end

class C
  include Z

  def initialize
    @c = "T"
  end
end

a = A.new
b = B.new
c = C.new
a.foo         # foo:  AS
a.fooX        # foox: AS
a.fooY        # fooy: AS
b.foo         # foo:  BX
b.fooX        # fooX: BS
b.fooY        # fooy: BS
b.fooZ        # fooz: BS
c.fooX        # foox: CT
c.fooY        # fooy: CT
c.fooZ        # fooz: CT
```

In diesem Beispiel werden drei Module X, Y und Z mit drei unterschiedlichen Methoden definiert. Im Gegensatz zu den Interfaces in Java sind die Implementierungen bereits in den Modulen enthalten. Ausserdem können in Ruby auch Instanzvariablen, wie die Variable c im Modul X, definiert werden.
Wie im Modul Z ersichtlich, können Module wiederum andere Module inkludieren. Die Klasse A inkludiert die beiden Module X und Y, initialisiert die Instanzvariable s vom Modul X und hat noch eine weitere Instanzmethode foo definiert. Die Subklasse B ist von der Klasse A abgeleitet und inkludiert zusätzlich das Modul Z. Die Klasse B hat somit die Methoden foo, fooX, fooY, fooZ und die Methode initialize sowie die Instanzvariable c definiert. Die Klasse C, die von keiner der Klassen abgeleitet ist, inkludiert das Modul Z. Somit stehen hier die Methoden fooX, fooY, fooZ und die in der Klasse direkt definierte Methode initialize zur Verfügung. Dadurch, dass die Module auch Implementierungen enthalten, ist im Gegensatz zu den Interfaces in Java keine Coderedundanz gegeben.

Obwohl Module einen eigenen Namensraum darstellen, kann es, wie im folgenden Beispiel demonstriert, zu Namenskollisionen kommen:

Beispiel 5.3.3 *[Namenskollision in Modulen]*

```
module X
  def fooX
     @myvar = 1 if @myvar.nil?
     @myvar
  end
end

module Y
  def fooY
     @myvar = 2 if @myvar.nil?
     @myvar
  end
end

class A
  include X, Y

  def initialize
    puts fooX      #  sollte 1 sein
    puts fooY      #  sollte 2 sein
  end
end

A.new              # liefert immer 1
```

Die beiden Module X und Y, welche in die Klasse A inkludiert werden, verwenden dieselbe Instanzvariable @myvar. Wenn die Instanzvariablen miteinander nichts gemein haben, entspricht das Ergebnis nicht den Erwartungen des Programmierers. Diese Situation sollte in der Regel nicht oft auftreten, da Module via getter/setter-Methoden auf Instanzvariablen des auftraggebenden Objektes zugreifen sollten. Wenn jedoch ein mixin mit einem eigenen Zustand erstellt werden soll, so sollte sichergestellt werden, dass die Instanzvariablen eindeutige Namen innerhalb des Systems aufweisen. Dies kann zum Beispiel durch Verwendung des Modulnamens als Teil des Variablennamens gewährleistet werden.

An dieser Stelle sei noch bemerkt, dass ein auf Mehrfachvererbung basierendes Modell mittels einer Konvertierungsstrategie in ein Einfachvererbungsmodell übergeleitet werden kann [Crespo et al., 2002]. Daneben gibt es noch das Prinzip der Delegation, mit dem Mehrfachvererbung simuliert werden kann.

5.3.3 Fazit

Mehrfachvererbung ist in Java und Ruby mit Klassen nicht möglich. Java bietet mit den Interfaces eine Mehrfachvererbung auf Basis der reinen abstrakten Klassen an. Wird beim Design der Klassen diese Einschränkung berücksichtigt, stellen die Interfaces eine brauchbare Lösung dar. Module in Ruby definieren einen Namensraum, in dem Methoden und

Konstanten abgeschottet sind. Module können wie die Interfaces in Java nicht instanziert werden. Stattdessen können in einer Klasse die Methoden des Moduls mit include als Instanzmethoden verfügbar gemacht werden. In Ruby wird dies als mixin bezeichnet. Mixins bieten somit eine brauchbare Möglichkeit, um zusätzliche Funktionalität in eine Klasse zu bekommen, ohne dabei mit den Problemen der Mehrfachvererbung konfrontiert zu werden. Die Mächtigkeit der Module als mixins ist viel stärker als die der Interfaces in Java.

5.4 Delegation

Die Delegation ist neben der Vererbung eine weitere Möglichkeit, neue Objekte für die Wiederverwendung zu erstellen. In [Liebermann, 1986] wird zwischen abstrakten Sets (Klassen), die Konzepte repräsentieren und den Prototypen, die ein Standardverhalten für ein Konzept aufweisen, unterschieden. Dieses Standardverhalten kann in neuen Objekten entsprechend adaptiert werden. Die Implementierung dieses Mechanismus in objektorientierten Sprachen nennt man Delegation. Eine Differenzierung zwischen Klassen und Instanzen ist bei der Verwendung von Prototypen nicht mehr notwendig.

5.4.1 Delegation in Java

Das folgende Beispiel in Java implementiert das Interface Marathon in der Klasse Runner und das Interface Crawl in der Klasse Swimmer und Duathlete.

Beispiel 5.4.1 *[Delegation in Java]*

```
interface Marathon {
  String getTimeMarathon();
}
class Runner implements Marathon {
  private String TimeMarathon;

  Runner(String time) { TimeMarathon = time; }
  public String getTimeMarathon() { return TimeMarathon; }
}

interface Crawl {
  String getTimeCrawl100m();
}
class Swimmer implements Crawl {
  private String TimeCrawl100m;

  Swimmer(String time) { TimeCrawl100m = time; }
  public String getTimeCrawl100m() { return TimeCrawl100m; }
}

class Duathlete extends Runner implements Crawl {
  private Swimmer swimmer;

  Duathlete(String rtime, String stime) {
```

```
    super(rtime);
    swimmer = new Swimmer(stime);
  }

  // implementiert Methode von Crawl durch Weiterleitung
  public String getTimeCrawl100m() { return swimmer.getTimeCrawl100m(); }
}

class MyDelegation {
  public static void main(String args[]) {
    Duathlete duathlete = new Duathlete("3:12:04", "0:01:59");

    System.out.println("Marathonzeit: " + duathlete.getTimeMarathon());
    System.out.println("100m Kraulen: " + duathlete.getTimeCrawl100m());
  }
}
```

Die Implementierung in der Klasse Duathlete ermöglicht die Mehrfachvererbung durch Delegation. Die Methode getTimeCrawl100m() in der Klasse Duathlete dient als Hülle. Sie delegiert die Anfrage an die Klasse Swimmer, die den Prototyp darstellt, unverändert weiter. Dieser Delegationscode muss für jede benötigte Methode erstellt werden, womit der Schreibaufwand beim Programmieren rasch ansteigen kann.
Wenn statt dieser Delegationsmethode die Instanzvariable Swimmer von private auf public geändert wird, kann ein Benutzer direkt auf die Methoden der Klasse Swimmer zugreifen und der Schreibaufwand würde entfallen. Die Klasse Duathlete implementiert somit das Interface Swimmer nicht mehr. Dies hat den gravierenden Nachteil, dass die Klasse Duathlete nicht mehr selektiv das Verhalten der Klasse Swimmer beeinflussen kann.
Um den Schreibaufwand für die Delegationsmethoden in Java zu reduzieren, kann der in [Viega et al., 1998] vorgestellte Präprozessor Jamie[7], der den Prozess zur Erstellung des Delegationscodes automatisch generiert, eingesetzt werden.

5.4.2 Typinkonsistenz in Java

Ein Nachteil des Delegationsprinzips gegenüber der Mehrfachvererbung ist, dass es keine explizite Typbeziehung zwischen den Klassen gibt [Tempero et al., 1998]. Im folgenden Beispiel wird diese Problematik dargestellt:

Beispiel 5.4.2 *[Delegationsproblem in Java]*

```
interface Marathon { ... }
class Runner implements Marathon { ... }

interface Crawl { ... }
class Swimmer implements Crawl { ... }
```

[7] Jamie ist frei verfügbar unter http://www.list.org/jamie.

```
class Duathlete extends Runner implements Crawl { ... }

class Multiathlete {
  private Runner runner;
  private Swimmer swimmer;

  public void addRunner(Runner r) { runner=r; }
  public void addSwimmer(Swimmer s) { swimmer=s; }
}

class MyDelegation2 {
  public static void main(String args[]) {
    Duathlete duathlete = new Duathlete("3:12:04", "0:01:59");
    Multiathlete multiathlete = new Multiathlete();
    multiathlete.addRunner(duathlete);    // gueltig
    multiathlete.addSwimmer(duathlete);   // Typfehler
  }
}
```

Es wird eine weitere Klasse Multiathlete mit den Methoden addRunner(Runner r) und addSwimmer(Swimmer s) definiert. Im Hauptprogramm wird eine Instanz von Duathlete diesen beiden Methoden als Parameter übergeben. Zum Compilierungszeitpunkt meldet der Java-Compiler beim Methodenaufruf von multiathlete.addSwimmer(duathlete) eine Typinkonsistenz, da es zwischen dem erwarteten Typ Swimmer und dem Typ Duathlete keine entsprechende Beziehung gibt.

5.4.3 Delegation in Ruby

In Ruby können Delegationen recht einfach mit den Methoden method_missing und respond_to? realisiert werden.

Beispiel 5.4.3 *[Delegation in Ruby]*

```
class Runner
  attr :timeMarathon;

  def initialize(time)
    @timeMarathon = time
  end
end

class Swimmer
  attr :timeCrawl100m

  def initialize(time)
    @timeCrawl100m = time
  end
```

```
end

class Duathlete
  def initialize(rtime, stime)
    @runner = Runner.new(rtime)
    @swimmer = Swimmer.new(stime)
  end

  def method_missing(meth, *args)
    method = meth.to_s
    case true
      when @runner.respond_to?(method)
        @runner.send(method, *args)
      when @swimmer.respond_to?(method)
        @swimmer.send(method, *args)
      else super
    end
  end
end

duathlete = Duathlete.new("3:12:04", "0:01:59")
p duathlete.timeMarathon, duathlete.timeCrawl100m
p duathlete.foo         # NoMethodError
```

Die Klasse Duathlete instanziert im Konstruktor die beiden Klassen Runner und Swimmer mit den entsprechenden Parametern. Die Methode method_missing wird von Ruby aufgerufen, wenn eine Methode der entsprechenden Klasse nicht gefunden wird. Hier wird die fehlende Methode mit respond_to? in den jeweiligen Objekten gesucht und wenn vorhanden auch aufgerufen. Falls die Methode auch in keiner der delegierten Klassen anzufinden ist, wird die Standardmethode von method_missing durch die super-Anweisung ausgeführt, die eine entsprechende Fehlermeldung ausgibt.
Der Nachteil bei dieser Lösung ist, dass es bei vielen Delegationsklassen und deren Methoden zu aufwändigen Suchvorgängen kommen kann, womit die Laufzeit negativ beeinflusst wird. Häufig verwendete Delegationsklassen sollten also so früh wie möglich in der Suchkette bearbeitet werden.

5.4.4 Modul Forwardable in Ruby

Eine kleine Anzahl häufig verwendeter Design Patterns [Gamma et al., 1995] befinden sich bereits in der Standardbibliothek von Ruby. Diese umfassen unter anderem auch Klassen bzw. Module für die Delegation. Wenn in einer Masterklasse nur ausgewählte Methoden weitergeleitet werden sollen, so bietet sich dafür das Forwardable Modul an:

Beispiel 5.4.4 *[Modul Forwardable in Ruby]*

```
require 'forwardable'
```

```
class Runner
  attr :timeMarathon;

  def initialize(time)
    @timeMarathon = time
  end
end

class Swimmer
  attr :timeCrawl100m

  def initialize(time)
    @timeCrawl100m = time
  end
end

class Duathlete
  extend Forwardable

  def initialize(rtime, stime)
    @runner = Runner.new(rtime)
    @swimmer = Swimmer.new(stime)
  end

  def_delegators(:@runner, :timeMarathon)
  def_delegators(:@swimmer, :timeCrawl100m)
end

duathlete = Duathlete.new("3:12:04", "0:01:59")
p duathlete.timeMarathon, duathlete.timeCrawl100m
```

Mit *require* wird die Bibliothek forwardable in das Programm geladen. Das Objekt Duathlete wird durch die Methode *extend* mit den Eigenschaften vom Modul Forwardable erweitert. Im Konstruktor werden wie zuvor die Instanzvariablen der Delegationsklassen Runner und Swimmer mit den entsprechenden Parametern initialisiert. Statt wie vorhin jede gültige Methode an das entsprechende Objekt weiterzuleiten, wird mit der Instanzmethode def_delegators die delegierte Klasse und dessen Methode definiert.

Das in Abschnitt 5.4.2, Seite 89 beschriebene Problem in Java mit der Klasse Multiathlete und den typspezifischen Parametern in den Methoden addRunner(Runner r) bzw. addSwimmer(Swimmer s) tritt auf Grund des dynamischen Typkonzeptes in Ruby nicht auf.

5.4.5 Fazit

Der Mechanismus der Delegation ist in Java auf Grund der Problematik mit der Typinkonsistenz nur eingeschränkt anwendbar. Die Programmiersprache Ruby bietet gleich mehrere

Möglichkeiten an, um das Delegationsprinzip zu implementieren. Einerseits durch die Methoden method_missing und respond_to?, die bei Klassen mit zahlreichen Methoden zu langen Suchvorgängen führen können, und andererseits mit den in der Standardbibliothek enthaltenen Design Patterns. Ein Vertreter dieser Patterns ist das Modul Forwardable, mit dem selektiv die weiterzuleitenden Methoden spezifiziert werden können. Ruby bietet in dieser Hinsicht mehr Möglichkeiten mit weniger Problemen an.

5.5 Informationsverheimlichung (information hiding)

Eine wichtige Eigenschaft der Informationsverheimlichung ist, dass bei einer Modifikation eines Moduls nur die versteckten (geheimen) Elemente geändert werden und die öffentlichen unberührt bleiben. Je kleiner der öffentliche Teil ist, desto höher ist die Wahrscheinlichkeit, dass Änderungen sich nur auf den versteckten Teil auswirken. In [Meyer, 1997] wird weiters ein Modul, das *information hiding* unterstützt, als ein Eisberg skizziert. Die für den Anwender sichtbare Spitze stellt dabei die Schnittstelle dar.

5.5.1 Zugriffsmethoden

In objektorientierten Sprachen ist es üblich, dass der Zugriff auf die Variablen einer Klasse nur durch Methoden erlaubt wird. Dadurch, dass die eigentlichen Variablen unsichtbar sind, wird der Programmcode leichter wiederverwendbar und wartbar. Das Java Beispiel zeigt eine Klasse Person, auf deren private Variable name nur innerhalb der Klasse Person direkt zugegriffen werden darf. Sowohl abgeleitete Klassen als auch aussenstehende Benutzer können nur mit Hilfe der öffentlichen Methoden setName() und getName() auf die Variable Einfluss nehmen.

Beispiel 5.5.1 *[getter/setter in Java]*

```
class Person {
  private String name;
  public void setName(String s) {
    name=s;
  }

  public String getName() {
    return name;
  }

  public static void main(String args[]) {
    Person p = new Person();
    p.setName("Nobody");              // setter
    System.out.println(p.getName());  // getter
  }
}
```

Eine gleichwertige Lösung in Ruby sieht wie folgt aus:

Beispiel 5.5.2 *[getter/setter in Ruby]*

```
class Person
  def name
    @name
  end

  def name=(newName)
    @name = newName
  end
end

pers = Person.new()
pers.name = "Peter"  # Setter
p pers.name          # Getter
```

Ruby bietet jedoch auch eine stark verkürzte Schreibweise an:

Beispiel 5.5.3 *[attr in Ruby]*

```
class Person
  attr :name, true  # setter und getter
end
```

Neben der *attr* Attributdeklaration gibt es in Ruby noch *attr_accessor*, *attr_reader* und *attr_writer*. Die Deklaration *attr_accessor* entspricht der *attr* mit dem Flag true, wodurch neben der getter-Methode auch eine setter-Methode erstellt wird. In [Meyer, 1997] wird das „Uniform Access Principle“ als ein Prinzip beschrieben, in dem alle Services eines Moduls durch eine einheitliche Notation abrufbar sind, bei dem nicht verraten wird, ob die Implementierung durch eine Speichervariable oder durch eine Berechnung erfolgt. Das folgende Temperaturbeispiel zeigt eine Attributmethode fahrenheit, die eine virtuelle Instanzvariable darstellt.

Beispiel 5.5.4 *[virtuelle Instanzvariable in Ruby]*

```
class Temperatur
  attr :celsius, true

  def fahrenheit
    (@celsius * 9/5) + 32
  end

  def fahrenheit=(wert)
    @celsius = (wert - 32) * 5/9
```

```
  end
end

t = Temperatur.new()
t.celsius=25
p t.fahrenheit    # 77
t.fahrenheit=40
p t.celsius       # 4
```

Diese Attributdeklarationen sind technisch gesehen nicht Teil der Programmiersprache Ruby, da sie durch einfache Methoden, die in der Klasse Module definiert sind, automatisch erstellt werden.
Der Java-Code wäre, wie im Beispiel davor schon erkennbar, viel umfangreicher. Sowohl die setter/getter-Methoden, als auch die Instanzvariablen müssen in Java explizit definiert werden.

5.5.2 Zugriffskontrolle

Sowohl in Java als auch in Ruby gibt es die Möglichkeit, Methoden und Variablen in unterschiedlichen Graden zugänglich zu machen. In Abschnitt 3.3.7, Seite 23 wurden die vier Zugriffskontrollen public, protected, package und private von Java und in Abschnitt 4.3.8, Seite 56 die drei Zugriffskontrollen public, protected und private von Ruby erläutert. An dieser Stelle wird nun der feine Unterschied bei den Zugriffskontrollen private und protected zwischen den beiden Programmiersprachen diskutiert.

Die privaten Methoden von Instanzen können in Java nicht in einer Subklasse aufgerufen werden. Der Versuch wird bereits vom Compiler unterbunden.

Beispiel 5.5.5 *[Private Methoden in Java]*

```
class A {
  private void foo() {
    System.out.println("A");
  }
}

class B extends A {
  void foo2() {
     foo();          // Fehler beim Kompilieren
  }
}
```

In Ruby ist der Aufruf einer privaten Methode auch dann noch möglich, wenn sich diese Methode in einer Oberklasse des aktuellen Objekts *self* befindet. Der Aufruf ist nur ohne Verwendung eines expliziten Empfängers möglich, da nur *self* als Empfänger erlaubt ist. Das folgende Beispiel illustriert diese Eigenschaft:

Beispiel 5.5.6 *[Private Methoden in Ruby]*

```
class A
  private
  def foo
    puts "A"
  end
end

class B < A
  def foo2
    foo
  end
end

B.new.foo2    # "A"
```

In Java ist wiederum der Aufruf von privaten Methoden über Objektgrenzen hinweg möglich:

Beispiel 5.5.7 *[Private Methoden über Objektgrenzen in Java aufrufen]*

```
class A {
  private void foo() {
    System.out.println("A");
  }

  public void foo2(A a) {
   a.foo();
  }
}

class MyPrivate2 {
  public static void main(String args[]) {
    A a=new A();
    a.foo2(new A());     // "A"
  }
}
```

Wie diese Beispiel zeigt, kann mit Hilfe einer binären Methode foo2() die private Methode foo() von einem anderen Objekt des gleichen Typs aufgerufen werden. Das Pendant funktioniert in Ruby nicht:

Beispiel 5.5.8 *[Private Methoden über Objektgrenzen in Ruby aufrufen]*

```
class A
  private
  def foo
    puts "A"
  end

  public
  def foo2(p)
    p.foo     # NoMethodError
  end
end

A.new.foo2(A.new)
```

Methoden, die mit protected gekennzeichnet sind, können in Ruby nur von Instanzen der definierten Klasse, einschließlich der Subklassen, aufgerufen werden. In Java besteht daneben auch noch die Möglichkeit der Verwendung innerhalb des Paketes, in dem die Methode definiert worden ist. Im folgenden Beispiel verwendet sowohl die Subklasse B als auch die Klasse C – die keine Subklasse von A ist, sich aber im selben Paket befindet – die protected Methode foo() der Klasse A.

Beispiel 5.5.9 *[Protected Methoden in Java]*

```
class A {
  protected void foo() {
    System.out.println("A");
  }
}

class B extends A {
  public void foo2() {
    foo();
  }
}

class C {
  public void foo3(A a) {
    a.foo();
  }
}

class MyPrivate3 {
  public static void main(String args[]) {
    B b=new B();
    C c=new C();
    b.foo();        // "A"
    c.foo3(b);      // "A"
```

```
  }
}
```

Die direkte Entsprechung in Ruby funktioniert nur bezüglich der Subklasse B, jedoch nicht in der Klasse C:

Beispiel 5.5.10 *[Protected Methoden in Ruby]*

```
class A
  protected           # public funktioniert
  def foo
    puts "A"
  end
end

class B < A
  def foo2
    foo
  end
end

class C
  def foo3(p)
    p.foo
  end
end

B.new.foo2         # "A"
C.new.foo3(B.new)  # NoMethodError
```

In Java können die Zugriffskontrollen überschriebener Methoden über Vererbungshierarchien hinweg nicht abgeschwächt werden.

Beispiel 5.5.11 *[Abschwächung der Zugriffskontrollen in Java]*

```
class A {
  protected void foo() {
    System.out.println("A");
  }
}

class B extends A {
  public void foo() {  // schwaecheres Zugriffsprivileg: okay
    super.foo();
  }
```

```
}

class C extends A {
  private void foo() { // Fehler: staerkeres Zugriffsprivileg
    super.foo();
  }
}
```

Die Subklasse B überschreibt die Methode foo() der Superklasse A und ändert dabei auch die Zugriffsmöglichkeit von protected auf public. Die Verwendung eines schwächeren Zugriffsprivilegs ist in Java möglich, die Verstärkung wie in diesem Beispiel in der Subklasse C auf private wird hingegen nicht unterstützt. Es erfolgt eine entsprechende Fehlermeldung beim Compilieren.

In Ruby gibt es, wie das folgende Beispiel zeigt, keine Einschränkung hinsichtlich des Veränderns von Privilegien bei überschriebenen Methoden.

Beispiel 5.5.12 *[Abschwächung der Zugriffskontrollen in Ruby]*

```
class A
  protected
  def foo
    puts "A"
  end
end

class B < A
  public
  def foo
    super
  end
end

class C < B
  private
  def foo
    super
  end
end

B.new.foo     # "A"
#C.new.foo    # nicht moeglich, private Methode
```

Ergänzend ist zu erwähnen, dass in Ruby die Zugriffsrechte dynamisch ermittelt werden. Das bedeutet, dass sich der Code zur Laufzeit ändern kann und eine Zugriffsverletzung erst beim Ausführen der entsprechenden Codestelle entdeckt wird. In Java sind die Zugriffsrechte statisch, die Ermittlung erfolgt also zum Compilierungszeitpunkt.

5.5.3 Fazit

Java bietet keine automatische Sprachunterstützung für den Zugriff auf Variablen durch sogenannte setter/getter-Methoden an. Ruby ist in dieser Hinsicht mit seiner Attributdeklaration attr komfortabler. Dieser Komfort ist in der Praxis aber nur bedingt vorhanden, denn eine intensive Nutzung von setter/getter-Methoden deutet im Allgemeinen auf ein schlechtes Design[8] hin.
Die Zugriffskontrollen private und protected agieren in Java anders als in Ruby. In Ruby können private Methoden unter bestimmten Voraussetzungen auch von einer Subklasse aufgerufen werden. In Java ist wiederum der Aufruf privater Methoden mit Hilfe binärer Methoden über Objektgrenzen hinweg möglich. Auf protected Methoden können in Java auch nicht abgeleitete Klassen zugreifen, wenn sich diese im selben Paket befinden.
Die Veränderung der Zugriffsrechte in Subklassen unterliegen in Ruby keiner Einschränkung. Im Gegensatz dazu erlaubt Java keine Abschwächung der Zugriffsrechte.

5.6 Abstrakte Klassen

Eine sinnvolle Eigenschaft des objektorientierten Programmierens ist das Konzept der abstrakten Klassen. Durch die Verwendung von abstrakten Klassen können Klassen deklariert werden, die nur Teile von Implementierungen enthalten. Von dieser abstrakten Klasse abgeleitete Klassen können nun spezifische Implementierungen einiger oder aller Methoden durchführen. Das Gegenstück von abstrakten Klassen sind die konkreten Klassen, in denen alle Methoden implementiert sind. Abstrakte Klassen sind hilfreich, wenn einige Verhaltensweisen nur für manche Objekte eines bestimmten Typs zutreffen.

Die Abstraktheit von Klassen ist in [Hürsch, 1994] definiert als die Unmöglichkeit sie zu instanzieren. Damit ein Programmierer nicht alle Methoden einer Klasse und die der zugehörenden Superklassen auf die Existenz einer abstrakten Methode inspizieren muss, sollte die abstrakte Klasse durch ein entsprechendes Schlüsselwort (z.B.: abstract, deferred) syntaktisch gekennzeichnet sein.

5.6.1 Abstrakte Klassen in Java

Sind alle Methoden vorerst nur zu definieren und später zu implementieren, so sind in Java Interfaces zu verwenden, welche in Abschnitt 3.3.4, Seite 18 erläutert sind.
Jede nicht implementierte Methode in Klassen wird in Java mit *abstract* markiert.

Beispiel 5.6.1 *[Abstrakte Klasse in Java]*

```
abstract class Benchmark {
  abstract void benchmark();

  public final long run() {
    long start = System.currentTimeMillis();
    benchmark();
```

[8] Von Klassen erwartet man sich, dass sie Dienste zur Verfügung stellen. Die Daten sollten sich im Zuge der Ausführung der angebotenen Dienste im Hintergrund verändern und nicht direkt über die Schnittstelle erreichbar sein.

```
    return (System.currentTimeMillis() - start);
  }
}

class NoneBenchmark extends Benchmark {
  void benchmark() { }    // mache nichts
}

class LoopBenchmark extends Benchmark {
  void benchmark() { for (long l = 0; l < 1000000; ++l); }
}

class MyAbstract {
  public static void main(String args[]) {
    System.out.println(new NoneBenchmark().run());  // 0
    System.out.println(new LoopBenchmark().run());  // 10
    //Benchmark b=new Benchmark();                  // nicht instanzierbar
  }
}
```

Das Schlüsselwort *abstract* darf bei der Klassendefinition von Benchmark in Java nicht fehlen. Wie eingangs erwähnt, ist es sinnvoll, explizit auf die Abstraktheit einer Klasse hinzuweisen. Die konkreten Subklassen NoneBenchmark und LoopBenchmark implementieren jeweils die abstrakte Methode benchmark() aus der Superklasse. Beim Aufruf der Methode run(), die in der abstrakten Klasse Benchmark implementiert ist, wird die implementierte Methode benchmark() der entsprechenden Subklasse ausgeführt. Eine Instanzierung einer abstrakten Klasse, wie der Klasse Benchmark in diesem Beispiel, ist in Java nicht möglich.

5.6.2 Simulation abstrakter Klassen in Ruby

Die dynamische Programmiersprache Ruby bietet ein solches Konzept der abstrakten Klassen[9] nicht an. Eine sehr einfache Variante, abstrakte Methoden zu signalisieren, kann durch Auslösen (raise) einer Ausnahme bewirkt werden:

Beispiel 5.6.2 *[Simulation abstrakter Methoden in Ruby]*

```
class Benchmark # abstrakt
  def benchmark
    raise "Abstrakte Methode!"
  end

  def run
    start = Time.now
    benchmark
    Time.now - start
```

[9] Jedoch findet sich in [Flanagan, 2002] der Begriff der abstrakten Klassen für Numeric und Integer wieder. Bei der Klasse Integer ist zum Beispiel die Methode new nicht definiert.

```
  end
end

class NoneBenchmark < Benchmark
  def benchmark      # mache nichts
  end
end

class LoopBenchmark < Benchmark
  def benchmark
     x = 0
     1000000.times do
        x += 1
     end
  end
end

p NoneBenchmark.new.run  # 0.0
p LoopBenchmark.new.run  # 0.972
Benchmark.new.run        # Ausnahme: Abstrakte Methode!
```

Der Benutzer wird im Gegensatz zu einer undefinierten Methode, die in Ruby durch die Fehlermeldung „undefined Method“ signalisiert wird, mit der selbstdefinierten Meldung „Abstrakte Methode!“ auf das Fehlen einer nicht implementierten Methode zur Laufzeit hingewiesen. Die Kennzeichnung der Klasse Benchmark als abstrakt im Kommentar verhindert nicht die Instanzierung dieser Klasse, womit diese Lösung nicht der Definition von Abstraktheit von Klassen entspricht.

Um die Instanzierung einer Klasse in Ruby zu verhindern, kann die Methode new der gewünschten Klasse überschrieben oder als undefiniert (*undef*) gekennzeichnet werden. Zuvor müsste jedoch die Klasse mit den gewünschten Eigenschaften definiert und einmal instanziert werden. Dieser Lösungsansatz entspricht nicht wirklich der einer abstrakten Klasse, sondern eher dem Prinzip des Singleton Patterns [Gamma et al., 1995].

Eine weitere und auch anwendbare Variante, die das Konzept der Module und die dynamischen Eigenschaften der Sprache miteinschließt, zeigt das folgende Beispiel:

Beispiel 5.6.3 *[Module als abstrakte Klassen]*

```
class Module
  def abstract(*ids)
    ids.each do | id |
      name = id.id2name
      class_eval %Q{
         def #{name} (*a)
           raise "Abstrakte Methode!"
         end
```

```
      }
    end
  end

  alias implements include
end

module Benchmark
  abstract :benchmark

  def run
    start = Time.now
    benchmark
    Time.now - start
  end
end

class NoneBenchmark; implements Benchmark
  def benchmark
  end
end

class LoopBenchmark; implements Benchmark
  def benchmark
     x = 0
     1000000.times do
        x += 1
     end
  end
end

class WrongBenchmark; implements Benchmark
end

p NoneBenchmark.new.run  # 0.0
p LoopBenchmark.new.run  # 0.972
p WrongBenchmark.new.run # Ausnahme: Abstrakte Methode!
Benchmark.new.run        # Modul ist nicht instanzierbar
```

Wie in Abschnitt 4.3.10, Seite 57 erläutert, können Module in Ruby nicht instanziert werden. Diese Eigenschaft wäre also analog den abstrakten Klassen in Java und entspricht auch der Definition von abstrakten Klassen. Die Kennzeichnung der Abstraktheit ist somit durch die Anweisung module implizit gegeben.
Die Klasse Module wird im oberen Beispiel durch die Methode abstract erweitert. Diese Methode übernimmt als Parameter ein beliebig langes Array von Methodennamen. Die Elemente dieses Arrays werden in der Folge einzeln behandelt. Mit der Instanzmethode id.id2name() wird der lokalen Variable name der zur id gehörende Methodenname

zurückgeliefert. Durch die Evaluierung von class_eval wird dynamisch eine Methode mit dem Namen der abstrakten Methode im Kontext des eingesetzten Moduls definiert. Diese neue Methode signalisiert, wie im Beispiel davor, die fehlende Implementierung durch das Auslösen einer Ausnahmebedingung. Die Zeichen %Q evaluieren den nachfolgenden String, begrenzt durch die Delimiter { und }, zu einer Zeichenkette zwischen doppelten Hochkommas.
Das Modul Stack, welches selbst nicht instanzierbar ist, definiert mit abstract die abstrakten Methoden push und pop. Eine interessante Sprachspielerei wird durch die Verwendung der alias-Anweisung in der Klasse Module erreicht. Die konkrete Klasse ConcreteStacks wird, ähnlich wie in Java, mit einem implements (anstatt mit include) von Stack erweitert[10].
Durch die Definition der abstrakten Methoden in Module ist es auch möglich, mehrere abstrakte Module miteinander zu kombinieren (Mehrfachvererbung).

5.6.3 Fazit

In Java werden abstrakte Klassen, die im Gegensatz zu den Interfaces auch implementierte Methoden beinhalten können, mit dem Schlüsselwort *abstract* definiert. Wie bei den Interfaces dürfen auch abstrakte Klassen nicht instanziert werden. Die Mehrfachvererbung von abstrakten Klassen ist in Java nicht möglich.
Ruby bietet keine Spracheigenschaften für abstrakte Klassen an. Begründet kann dieser Aspekt dadurch werden, dass die Sinnhaftigkeit abstrakter Klassen in einer Programmiersprache, in der Methoden zur Laufzeit dynamisch hinzugefügt und entfernt werden können, in Frage gestellt wird. Wenn diese dynamischen Möglichkeiten nicht genutzt werden, ist die Simulation durch ein einfaches Auslösen einer Ausnahme keine Ersatzlösung für abstrakte Klassen in Ruby, da unter anderem solche Klassen immer noch instanzierbar sind. Die Lösung mit Hilfe der Module ist hingegen eine anwendbare Alternative in Ruby, bei der auch die Mehrfachvererbung unterstützt wird.

5.7 Objektspezialisierung und Finalisierung

In den meisten objektorientierten Programmiersprachen haben alle Objekte einer bestimmten Klasse dasselbe Verhalten. Die Klasse agiert ähnlich einer Schablone. Jedesmal wenn der Konstruktor aufgerufen wird, wird ein Objekt mit den gleichen Schnittstellen kreiert.

In Ruby kann das Verhalten eines Objektes auch während der Laufzeit geändert werden. Dabei wird dem Objekt eine private virtuelle Subklasse hinzugefügt. Dieses eine spezielle Objekt erhält dadurch ein zusätzliches Verhalten. Andere Objekte derselben Klasse besitzen dieses Verhalten daher nicht.

Methoden, Klassen und Objekte, die nur einmal auftreten, werden in Programmiersprachen Singleton genannt. Der Begriff Singleton ist auch im Bereich der Design-Patterns [Gamma et al., 1995] wohl bekannt. Anders als bei der Objektspezialisierung bezieht sich das Singleton-Pattern[11] auf eine Klasse, die nur einmal instanziert werden kann.

[10] Der Strichpunkt ist notwendig, da in Ruby mehrere Anweisungen in einer Zeile durch diesen getrennt werden müssen.
[11] In Ruby kann das Singleton Pattern mit dem Modul Singleton implementiert werden.

5.7.1 Singletonmethoden und -klassen in Ruby

Das folgende Beispiel zeigt zwei Objekte a und b, die Zeichenketten darstellen. Für das zweite Objekt b wird eine Methode upcase hinzugefügt, die das Verhalten der ursprünglichen Methode überschreibt.

Beispiel 5.7.1 *[Singletonmethode in Ruby]*

```
a = "hello world"
b = "hello world"

def b.upcase       # Singleton Methode
  gsub(/[aeiou]/) { |x| x.upcase }
end

p a.upcase      # HELLO WORLD
p b.upcase      # hEllO wOrld
```

Wenn noch keine Singletonklasse existiert, so wird beim Hinzufügen einer Singletonmethode zu einem Objekt eine solche implizit erstellt. Die Superklasse der Singletonklasse ist die Originalklasse des Objekts, womit die Singletonklasse eine virtuelle Subklasse der Originalklasse darstellt. Um mehrere Singletonmethoden zu erstellen, ist es oft sinnvoll die Singletonklasse direkt zu erstellen:

Beispiel 5.7.2 *[Singletonklasse in Ruby]*

```
b = "hello world"

class << b           # Singleton Klasse
  def upcase         # Singleton Methode
    gsub(/[aeiou]/) { |x| x.upcase }
  end

  def upcase!        # Singleton Methode
    gsub!(/[aeiou]/) { |x| x.upcase }
  end
end

p b.upcase        # hEllO wOrld
p b               # hello world
b.upcase!
p b               # hEllO wOrld
```

In diesem Beispiel wurde mit „class << b“ eine neue virtuelle Klasse für das Objekt b explizit kreiert. In Ruby ist es üblich, ein ! nach einem Methodennamen zu setzen, wenn die Methode mit mehr Sorgfalt zu verwenden ist. Wie zu sehen ist, verändert die Methode upcase! den Wert vom Objekt b, während upcase ein neues Objekt retourniert.

Eine Restriktion in der Implementierung von Ruby untersagt die Definition von Singletonmethoden auf Instanzen von Fixnum und Symbol Klassen [Matz, 2002]. Die Details über die Zusammenhänge von Klassen, Metaklassen, Singletonklassen und Proxy-Klassen sind in Abschnitt 4.3.11, Seite 59 nachzulesen.

5.7.2 Anwendbarkeit einer Singletonklasse

Anhand eines Anwendungsbeispiels wird die Anwendbarkeit von Singletonklassen gegenüber der herkömmlichen Vorgangsweise diskutiert.

Im folgenden Java-Beispiel werden für die drei privaten Instanzvariablen a, b und c individuelle Methoden erstellt, die den Wert der entsprechenden Variable in Form einer Zeichenkette retournieren.

Beispiel 5.7.3 *[accessorString in Java]*

```
class Foo {
  private int[] a;
  private Float b;
  private java.util.Date c;

  public Foo() {
    a = new int[] {1,2,3};
    b = new Float(-12.4);
    c = new java.util.Date();
  }

  public String a_accessorString() {
    StringBuffer s = new StringBuffer("");
    for (int i=0; i<a.length; ++i) s.append(a[i]);
    return s.toString();
  }

  public String b_accessorString() {
    return b.toString();
  }

  public String c_accessorString() {
    return c.toString();
  }
}

class MyAccessor {
  public static void main(String args[]) {
    Foo o = new Foo();
    System.out.println(o.a_accessorString()); // 123
```

```
    System.out.println(o.b_accessorString()); // -12.4

    // Sat Sep 13 20:16:12 CEST 2003
    System.out.println(o.c_accessorString());
  }
}
```

Statt wie in Java für jede Variable eine individuelle Methode zu erstellen, kann in Ruby mit Hilfe einer Singletonklasse auf Ebene der Klasse eine Funktion accessorString definiert werden, die eine solche Methode generiert.

Beispiel 5.7.4 *[accessString in Ruby]*

```
class Foo
  class << self
    def accessorString(*names)
      names.each do |name|
        class_eval <<-EOD
          def #{name}
            @#{name}.to_s
          end
        EOD
      end
    end
  end

  def initialize
    @a = [1,2,3]
    @b = -12.4
    @c = Time.now
  end

  accessorString :a, :b, :c
end

o = Foo.new
p o.a      # 123
p o.b      # -12.4
p o.c      # Sat Sep 13 19:41:23 Westeuropaeische Sommerzeit 2003
```

Innerhalb der Definition eines Klassenteils bezieht sich *self* auf die soeben definierte Klasse. Ein darauf basierendes Singleton modifiziert somit die Klasse class. Für jedes übergebene Argument wird in der Klasse Foo eine Methode mit dessen Name hinzugefügt. Die Methode to_s, ähnlich der Methode toString() in Java, retourniert in der neu definierten Methode eine Stringrepräsentation vom Argument.

Die Definition von „class << self“ innerhalb einer Klasse bewirkt, dass das damit erstellte Singleton die Klasse class verändert. Einfach formuliert bedeutet dies, dass Instanzmethoden dieser Singletonklasse als Klassenmethoden aufrufbar sind.

5.7.3 Finalisieren versus Einfrieren

In Java können neben Feldern auch Methoden und Klassen als *final* markiert werden. Solche Methoden und Klassen können die Sicherheit erhöhen. Wenn eine Klasse als *final* markiert ist, kann keine Subklasse erstellt werden und die Kontrakte können nicht verändert werden. Wenn eine Methode als *final* spezifiziert wurde, kann auf ihre Implementierung vertraut werden, vorausgesetzt ihre Methodenaufrufe beziehen sich auf Methoden, die ebenfalls mit *final* gekennzeichnet sind.

Ein Beispiel für eine *final* Methode könnte eine Kennwortüberprüfungsmethode validatePassword() sein, in der sichergestellt wird, dass die Implementierung das Kennwort ordnungsgemäß überprüft. Wenn eine solche Methode nicht als *final* markiert wird, ist es leicht möglich, diese Methode so zu überschreiben, dass für jedes Kennwort fälschlicherweise gültig retourniert wird.

Beispiel 5.7.5 *[final Methode in Java]*

```
class Password {
  final boolean validatePassword() {
    // ...
  }
}
```

Es kann auch die Klasse, in der die Methode validatePassword() enthalten ist, als *final* markiert werden. Die damit finalisierte Klasse kann dadurch keine Subklassen enthalten, die die Implementierung von validatePassword() beeinflussen könnten.

Beispiel 5.7.6 *[final Klasse in Java]*

```
final class Password {
  boolean validatePassword() {
   // ...
  }
}
```

Wenn eine *final* Markierung durchgeführt wird, sollte sichergestellt sein, dass diese Restriktion auch wirklich gewünscht ist, denn die Flexibilität wird für andere Programmierer, die diese Klasse oder Methode als Basis für ihren Programmcode verwenden wollen, stark eingeschränkt. In vielen Fällen kann die Sicherheit bereits dadurch gewährleistet werden, dass anstatt der Klasse alle Methoden der Klasse als *final* markiert werden. Dadurch kann dem Verhalten der einzelnen Methoden, die somit nicht überschreibbar sind, vertraut werden und die Funktionalität der Klasse kann erweitert werden. Damit auch kein Missbrauch mit den Variablen der Klasse verursacht werden kann, müssen diese als Konstante bzw. private Variablen geschützt werden.

Ruby bietet eine solche Möglichkeit auf Klassenebene nicht an. Stattdessen können Objekte mit der Methode freeze eingefroren werden. Nachdem ein Objekt eingefroren ist, wird jede Änderung des Objekts mit einem „TypeError“ signalisiert. Es gibt in Ruby keine Möglichkeit, ein eingefrorenes Objekt wieder aufzutauen.

Beispiel 5.7.7 *[freeze in Ruby]*

```
s = "Peter"
s.freeze

begin
  s << "Arpad"          # wirft Ausnahme wegen unerlaubter Modifikation
rescue => err
  p "#{err.class} #{err}"
end

p s.frozen?             # true
s += "Arpad"            # erlaubt, da neues Objekt
```

Mit der Methode frozen? kann festgestellt werden, ob ein Objekt eingefroren ist oder nicht. Im Beispiel liefert die Methode erwartungsgemäß true. Interessant ist jedoch die letzte Zeile. Diese signalisiert keinen TypeError, wie man vielleicht intuitiv erwarten würde, da es sich um keine Veränderung des ursprünglichen Objekts handelt. Hier wird der Variablen s eine Referenz auf ein *neues* Objekt zugewiesen. Verständlicher wird diese Tatsache, wenn man statt der Zeile s += “...“ die äquivalente Variante s = s + “...“ betrachtet. Die Anweisung s + “...“ evaluiert zuerst zu einem neuen Objekt, das dann der Variablen s zugewiesen wird. Alle reflexiven Zuweisungsoperatoren verhalten sich so, wodurch die Methode freeze in Ruby leicht zu Verwirrung führen kann.

5.7.4 Fazit

Auf Grund der dynamischen Typen ist es in Ruby leicht möglich, Objekte zur Laufzeit zu verändern. Singletonklassen und deren Singletonmethoden sind hierfür ein mächtiges Instrumentarium, um die Schreibarbeit bei analogen Methoden zu reduzieren. Das Konzept der Singletonklassen ist für Ruby-Novizen jedoch schwer verständlich, wodurch deren Einsatz wenig verbreitet ist. Java bietet dieses Konzept nicht an, womit der Code zwar verständlicher, jedoch umfangreicher ist.
Das Schlüsselwort *final* in Java ermöglicht es, neben Variablen auch Methoden und Klassen vor Veränderungen zu schützen. Obwohl es bequemer ist, eine Klasse als *final* zu markieren, ist in einigen Anwendungsfällen auf Grund der Erweiterbarkeit der Klasse das Markieren jeder einzelnen Methode und Variable vorzuziehen.
Ruby bietet nur das Einfrieren (freeze) von Objekten an. Wie gezeigt wurde, kann es bei reflexiven Zuweisungsoperatoren leicht zu Problemen mit der Verständlichkeit des

Programms kommen. Ein weiterer Vorteil in Java von finalisierten Methoden gegenüber Standardmethoden ist die einfachere Ermittlung der korrekten Implementierung des entsprechenden Typs zur Laufzeit, wodurch die Performance erhöht wird. Dies führt zur Schlussfolgerung, dass das Finalisieren von Klassen bzw. Methoden in Java dem Einfrieren von Objekten bei Ruby bezüglich Anwendbarkeit und Verständlichkeit überlegen ist.

5.8 Überladen (overloading), Überschreiben (overriding)

Überladen von geerbten Methoden bedeutet, dass eine neue Methode mit dem gleichen Namen hinzugefügt wird, sich jedoch die Signatur (Parameteranzahl, Parametertypen) von der bestehenden Methode unterscheidet. Bei Vererbung existieren in der Subklasse beide Methoden, die Auswahl erfolgt anhand der deklarierten Parametertypen. Methoden mit dieser Eigenschaft nennt man *ad-hoc-polymorph* [Cardelli et al., 1985].
Im Gegensatz dazu bedeutet Überschreiben, dass der Name und die Signatur ident sind und die Methode nur einmal existiert. Beim Überschreiben erfolgt die Methodenauswahl zur Laufzeit anhand des dynamischen Typs.

5.8.1 Überladen versus Überschreiben in Java

Java bietet sowohl das Überladen als auch das Überschreiben von Methoden an:

Beispiel 5.8.1 *[Überladen, Überschreiben in Java]*

```
class A {
  public void foo(int i) {
    System.out.println("A foo: " + i);
  }

  public void foo2(char a) {
    System.out.println("A foo2: " + a);
  }
}

class B extends A {
  public void foo(int i) {            // overriding
    System.out.println("B foo: " + i);
  }

  public void foo2(int i) {           // overloading
    System.out.println("B foo2: " + i);
  }
}

class MyOverloading {
  public static void main(String args[]) {
    B b = new B();
    b.foo(1);        // "B foo: 1"
    ((A) b).foo(1);  // "B foo: 1"
```

```
    b.foo2('a');      // "A foo2: a"
    b.foo2(1);        // "B foo2: 1"
  }
}
```

In der Subklasse B wurde die Methode foo() überschrieben und die Methode foo2() überladen. Wie zu sehen ist, sind sowohl der Name foo, die Parameteranzahl und der Parametertyp (int) gleich. Auch die explizite Typumwandlung (casting) durch ((A)b).foo() liefert das Ergebnis der Methode foo() aus der Klasse B. Anders bei den Methoden foo2(), diese haben zwar den gleichen Namen und die gleiche Anzahl an Parametern, aber der Parametertyp in der Klasse A ist char und in der Klasse B ist int. Die Subklasse B erbt einerseits die Methode foo2() mit dem Parametertyp char von der Klasse A und bietet daneben noch die eigene Methode foo2() mit dem Parametertyp int an.

Ist die Anzahl der Parameter beim Überladen verschieden, ist es für den Compiler relativ einfach, die gewünschte Methode zu ermitteln. Wenn jedoch, wie in diesem Beispiel, die Anzahl der Parameter gleich ist, müssen vom Compiler komplexere Algorithmen zum Auffinden der richtigen Methode eingesetzt werden.

Das Überladen von Methoden birgt auch Gefahren in sich, die im ersten Moment nicht offensichtlich sind, zum Beispiel um Collections (sets, lists) durch Überladen der Methode classify() zu unterteilen [Bloch, 2001]:

Beispiel 5.8.2 *[Vorsicht bei Überladungen]*

```
import java.util.*;

class CollectionClassifier {
  public static String classify(Set s) {
    return "Set";
  }

  public static String classify(List l) {
    return "List";
  }

  public static String classify(Collection c) {
    return "Unknown Collection";
  }

  public static void main(String args[]) {
    Collection[] tests = new Collection[] {
      new HashSet(),                        // ein Set
      new ArrayList(),                      // ein List
      new HashMap().values()                // weder Set noch List
    };
```

```
    for (int i = 0; i < tests.length; ++i) {
      System.out.println(classify(tests[i])); // "Unknown Collection"
    }
  }
}
```

Man würde erwarten, dass die Ausgabe „Set“ gefolgt von „List“ und „Unknown Collection“ erscheint, stattdessen wird dreimal „Unknown Collection“ ausgegeben. Der deklarierte Typ ist bei allen drei Iterationen der Schleife gleich, nämlich Collection. Der dynamische Typ zur Laufzeit ist bei jeder Iteration unterschiedlich, nämlich Set, List bzw. Map. Dieser beeinflusst jedoch nicht die Wahl beim Überladen, womit die einzig anwendbare Methode, die dritte Variante classify(Collection c) bleibt und es zum überraschenden Ergebnis kommt. Beim Überladen ist der deklarierte Typ entscheidend. Die Lösung für das Überladeproblem in Java ist, alle drei Methoden zu einer Methode zusammenzufassen und mit *instanceof* den Typ zur Laufzeit zu ermitteln (siehe Abschnitt 3.3.9, Seite 28).

Für den Benutzer einer Klasse kann es oft sehr schwer ersichtlich sein, welche Methode beim Überladen in Frage kommt. Es ist daher ratsam, keine Methoden zu exportieren, die die gleiche Anzahl an Parametern aufweisen. [Meyer, 2001] empfiehlt die Verwendung von Überladungen wegen der Unvereinbarkeit mit der Objekttechnologie generell zu vermeiden. Sobald bei der Suche nach einem Methodennamen der erste entsprechende gefunden wurde, sollte diese beendet werden. Die Suche nach weiteren konkurrierenden Methoden mit dem gleichen Namen ist weder komfortabel noch sinnvoll. Die einfache Regel des „No-Overloading Principle“ lautet:

> Innerhalb eines gegebenen Kontextes sollten unterschiedliche Dinge auch unterschiedliche Namen haben.

5.8.2 Überschreiben und Dispatching in Ruby

Wenn eine abgeleitete Klasse eine Methode in Ruby erbt, so wird diese durch ein neuerliches Definieren überschrieben. Anders als Java kann Ruby auf Grund seiner dynamischen Typen keine Unterscheidung mit Hilfe der Parameter vornehmen, wie im folgenden Beispiel dargestellt:

Beispiel 5.8.3 *[Überschreiben in Ruby]*

```
class A
  def foo(a)
    puts "A"
  end
end

class B < A
  def foo
    puts "B"
```

```
  end
end

b = B.new()
b.foo()                # "B"
b.foo(1)               # ArgumentError, ungltige Anzahl
```

Obwohl die Methode foo() in der Klasse A mit einem Parameter und in der Subklasse B ohne Parameter definiert ist, wird die Methode foo() in der Klasse B überschrieben.

Auf Grund des dynamischen Typkonzeptes ist es laut Matz nicht notwendig, Methoden zu überladen, dafür gäbe es andere Alternativen. In einigen Fällen kann die folgende Lösung, in der die Ausführung anhand der Klasse eines Objektes getroffen wird (dispatching), verwendet werden:

Beispiel 5.8.4 *[Dispatching in Ruby]*

```
class A
  def foo(*args)
    case args.collect {|a| a.class}
      when [Float, Fixnum, String]
        f, i, s = args
        puts "Float, Fixnum, String"
      when [String]
        f = args
        puts "String"
    end
  end
end

a = A.new
a.foo(3.1, 3, "Peter")    # "Float, Fixnum, String"
a.foo("Peter")            # "String"
```

Die Argumente der Methode foo werden im Array args abgelegt. Die Methode collect hat als Parameter einen Block (siehe Abschnitt 4.3.6, Seite 53). Dieser Block wird für jedes Element von args, mit dem Element als Parameter, einmal aufgerufen. Die im Block enthaltene Methode class liefert als Ergebnis die Klasse des Objektes zurück. Von collect werden diese Ergebnisse in einem neuen Array gesammelt und an die case Anweisung retourniert.

Eine weitere Alternative für das Überladen von Methoden ist „double dispatching“[12]:

[12] Die Technik von double dispatching wird beim Visitor Pattern eingesetzt. Mit diesem Pattern können Operationen zu Klassen hinzugefügt werden, ohne dass die Klassen der Elemente, auf denen sie operieren, geändert werden müssen [Gamma et al., 1995].

Beispiel 5.8.5 *[Double Dispatching in Ruby]*

```
class Foo
  attr_accessor :name

  def initialize
    self.name = "Foo"
  end

  def action(arg)
    arg.action_Foo(self)
  end
end

class Bar
  attr_accessor :name

  def initialize
    self.name = "Bar"
  end

  def action_Foo(arg)
    puts "#{self.name} und #{arg.name}\n"
  end
end

f=Foo.new
f.action(Bar.new)       # "Bar und Foo"
```

Zuerst wird durch die Zeile f.action(Bar.new) die Methode action des Originalempfängers f mit dem Parameter Bar.new aufgerufen (1. dispatch). In dieser Methode wird dann das Argument Bar.new zum neuen Empfänger und ruft die Methode action_foo auf (2. dispatch).
Bei der Anwendung von „double dispatching“ ist zu beachten, dass die Berechnungen zur Laufzeit nicht unerheblich sind.

5.8.3 Redefinieren von Operatoren in Ruby

Ruby bietet zahlreiche Operatoren an (siehe Abschnitt 4.3.4, Seite 45). Der objektorientierten Natürlichkeit von Ruby entsprechend sind die meisten Operatoren Methoden und können daher wie jede andere Methode überschrieben werden.
Das folgende Beispiel überschreibt den Operator `[ ]` um Filmsequenzen auszuschneiden.

Beispiel 5.8.6 *[Redefinieren von Operatoren in Ruby]*

```
class Film
```

```
  def[](fromTime, toTime)
    puts "Ausschneiden von #{fromTime} bis #{toTime}."
  end
end

f=Film.new
f["00:00:11","01:10:05"]   # "Ausschneiden von 00:00:11 bis 01:10:05."
```

Java bietet kein Überschreiben oder Überladen von Operatoren an. Oft genügt es, den Operator durch eine entsprechende Methode zu ersetzen:

Beispiel 5.8.7 *[Methoden statt Operatoren in Java]*

```
class Film {
  // ...
  public void cut(String fromTime, String toTime) {
    System.out.println("Ausschneiden von " + fromTime +
                       " bis " + toTime + ".");
  }
  // ...
}

class MyOperator {
  public static void main(String args[]) {
    Film f = new Film();

    // "Ausschneiden von 00:00:11 bis 01:10:05."
    f.cut("00:00:11", "01:10:05");
  }
}
```

In einigen Fällen bietet das Redefinieren von Operatoren nicht nur eine vereinfachte Syntax. In Ruby spielt der Operator `<=>`, der zwei Objekte vergleicht und je nachdem, ob der Empfänger kleiner, gleich oder größer ist, die Werte -1, 0 oder +1 retourniert, eine bedeutende Rolle.
Wenn in Ruby Objekte sortiert werden sollen, ist es naheliegend, das Modul Comparable zu verwenden. Für dieses muss nur der Operator `<=>` entsprechend definiert werden und man erhält zusätzlich noch die Operatoren `<`, `<=`, `==`, `>=`, und `>`, sowie die Methode between? implementiert.

Beispiel 5.8.8 *[Modul Comparable in Ruby]*

```
class Foo
  include Comparable
  attr :str
```

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

  def <=>(other)
    str.size <=> other.str.size
  end

  def inspect
    @str
  end
end

class Bar
  attr_reader :str

  def initialize(str)
    @str = str + str
  end
end

a = Foo.new("FF")
b = Foo.new("FFF")
c = Foo.new("FFFF")
d = Foo.new("F")
e = Bar.new("BB")      # "BBBB"

p a < d                # false
p a.between?(b,c)      # false
p b.between?(a,c)      # true
p b >= e               # false
p [c, b, d, a].sort    # [F, FF, FFF, FFFF]
p [a, e, d].sort       # Fehler: undefinierte Methode <=>
```

Hier wurde der Operator `<=>` mit einer Funktion belegt, der die Länge zweier Objekte vergleicht. Da das Modul Comparable in die Klasse inkludiert wurde, stehen auch die oben genannten Vergleichsoperatoren zur Verfügung.
Bei der Ausführung eines binären Vergleichsoperators wird die Methode des Empfängers (der links stehende Operand) aufgerufen, womit das Argument (rechter Operand) der Implementierung des Empfängers entsprechen muss. Den Operator `<=>` muss das Argument nicht definiert haben.
Im Beispiel ist das Objekt e eine Instanz von Bar, in der die notwendige getter-Methode str definiert ist, der Operator `<=>` ist nicht definiert. Die fehlende Operatordefinition hat die Auswirkung, dass dieses Objekt nicht als Element in einem zu sortierenden Array enthalten sein darf. Die letzte Zeile im Code reklamiert den undefinierten Operator `<=>`.

Die Methode inspect in der Klasse Foo wurde definiert, damit die Ausgabe durch die Methode p in menschlich lesbarer Form erfolgt.

5.8.4 Fazit

In Java können Methoden überschrieben und überladen werden. Dagegen gibt es in Ruby nur die Möglichkeit, Methoden zu überschreiben. Der Anwender einer überladenen Methode muss alle konkurrierenden Methoden in einer Vererbungshierarchie ausfindig machen, um sicher zu gehen, dass er die richtige auswählt. Ausserdem ist beim Programmieren zu berücksichtigen, dass der deklarierte Typ für die Auswahl der Methode zuständig ist. Diese Tatsache kann leicht zu unerwünschten Ergebnissen führen, wodurch das Überladen von Methoden gewisse Erfahrung seitens des Programmierers erfordert.
Alternative Techniken für das Überladen sind dynamic dispatching und double dispatching. Beide Varianten können in Ruby und Java eingesetzt werden.

Das Überschreiben von Methoden ist dagegen intuitiv anwendbar. Der entsprechendste dynamische Typ entscheidet in Java welche Methode aufgerufen wird. Ruby mit seinen dynamischen Typen überschreibt geerbte Methoden mit demgleichen Namen unabhängig der formalen Parameter, wodurch eine bessere Verständlichkeit gegenüber der Situation in Java gegeben ist.

Ruby erlaubt auch das Redefinieren von Operatoren, die als Methoden implementiert sind. Durch die Definition des Operators `<=>` und der Inkludierung des Moduls Comparable stehen zahlreiche weitere Vergleichsoperatoren sofort zur Verfügung. Dies reduziert nicht nur den Schreibaufwand sondern erhöht auch die Lesbarkeit des Codes.

5.9 Design by Contract (DbC)[13]

Um qualitativ hochwertige Programme zu erstellen, sind die in Kapitel 2 beschriebenen Kriterien essentiell. Ein Programm wird als korrekt bezeichnet, wenn es der Spezifikation entspricht. Eine Möglichkeit, die Korrektheit gegen die formale Spezifikation zu überprüfen, ist die Programmverifikation mit dem Hoare-Kalkül [Gellner, 2001].
Basierend auf dieser formalen Methode hat Bertrand Meyer in [Meyer, 1992] eine Methode für die Softwareentwicklung erstellt, die als Design by Contract bezeichnet wird.

Die Grundidee von DbC ist, dass ein Server und sein Client miteinander einen Vertrag bilden. Der Client muss, bevor er eine Methode des Servers aufruft, gewisse Vorbedingungen (preconditions) erfüllen. Der Server garantiert wiederum, dass nach Ausführung der Methode sogenannte Nachbedingungen (postconditions) erfüllt sind. Der Server ist daneben auch noch für Invarianten (invariants) zuständig, die sowohl vor als auch nach der Methodenausführung gewährleistet sein müssen. Mit Hilfe von Zusicherungen (assertions) wird das Programm zur Laufzeit auf Korrektheit überwacht.

Zusicherungen erfüllen unter anderem die folgenden vier Zwecke:

- *Unterstützung bei der Konstruktion korrekter Programme:* Ein Client ist bezüglich Spezifikation nur dann korrekt, wenn seine Implementierung konsistent bezüglich

[13] „Design by Contract“ is a trademark of Interactive Software Engineering.

Vorbedingungen, Nachbedingungen und Invarianten ist. Mit Hilfe von DbC ist die korrekte Anwendung eines unbekannten Programmcodes, dessen Verhalten im Vertrag genau festgelegt wurde, möglich. Bei der Implementierung von Methoden müssen keine zusätzlichen Sicherheitsabfragen, wie zum Beispiel die Überprüfung der Parameter auf sinnvolle Werte, programmiert werden. Dies steigert nicht nur die Effizienz des Codes, sondern erlaubt dem Entwickler auch, sich auf die eigentliche Aufgabenstellung zu konzentrieren.

- *Dokumentationsunterstützung:* Der Vertrag dient gleichzeit auch als formale Spezifikation und Dokumentation. Dem Entwickler wird hiermit die Benutzung von Bibliotheken erleichtert. Die Möglichkeit, Vertragsvereinbarungen aus dem Quelltext maschinell zu extrahieren und für Dokumentationszwecke zu verwenden ist ebenfalls gegeben.

- *Hilfreich bei der Fehlersuche:* In [Jézéquel et al., 1997] wird die Ursache für den 500 Millionen US-Dollar teuren Absturz der Ariane 5 am 4. Juni 1996 analysiert. In diesem Artikel kamen die Autoren zu der Erkenntnis, dass mit konsequenten Einsatz eines Vertragsmodells wie DbC, dieser Absturz wahrscheinlich hätte vermieden werden können.

- *Grundlage für Ausnahmebehandlungen (exceptions):* In einer Methode kann der Ausnahmezustand durch einen Fehler, der beim Aufruf einer weiteren Methode aufgetreten ist, ausgelöst werden. Ursache für eine Ausnahmesituation können interne oder externe Ursachen sein. Interne Ursachen sind Ereignisse, die eine Ausnahmebehandlung während der Programmausführung erfordern. Externe Ursachen werden durch die Hardware ausgelöst.
  Eine Methode kann auf diese Ausnahme entweder durch neuerliches Ausführen (retry) des Codeteils oder durch eine organisierte Panikreaktion reagieren. Letzteres verursacht in der Methode einen Fehler und sendet die Ausnahme an den Aufrufer der Methode. Die formale Aufgabe des Ausnahmebehandlers in diesem Fall ist die Wiederherstellung der Invariante.
  Wenn die Ausnahme ein *retry* durchführt, so muss neben der Invariante auch die Vorbedingung wiederhergestellt werden, damit beim neuerlichen Ausführen der Methode das Vertragsziel erfüllt werden kann.
  Zusicherungen sollten so eingesetzt werden, dass der Klassenzusammenhalt maximiert und die Objektkopplung minimiert wird. Damit Client und Server voneinander relativ unabhängig sind, sollten nur wirklich notwendige Zusicherungen getroffen werden. Dies vermeidet Verwirrungen und verbessert die Wartbarkeit. Andererseits sollten keine notwendigen Zusicherungen fehlen, denn dadurch könnten Programmänderungen zu unerwarteten Fehlern führen, die auf Grund der nicht ausgewiesenen Abhängigkeit nur schwer auffindbar sind.

Bis zur Java Version 1.3 wurden meist Kommentare oder Ausnahmebehandlungen (siehe Abschnitt 3.3.11, Seite 31) als Ersatz für Zusicherungen eingesetzt. Zusicherungen haben den Zweck, dass Programmierer Zustände spezifizieren können, von denen sie annehmen, dass sie wahr sind. Ausnahmebehandlungen stellen in Java einen Mechanismus dar, mit dem auf unerwartete Ereignisse reagiert werden kann.

5.9.1 Kommentare als Zusicherungen in Java

Das erste Beispiel zeigt die Verwendung von Kommentaren als Zusicherungen:

Beispiel 5.9.1 *[Kommentare als Zusicherungen]*

```
class Quadrat {
  // berechnet die Flaeche eines Quadrates; seite >= 0
  public double flaeche(double seite) {
    double wert;
    wert = seite * seite;
    return wert;
  }
  // Retourniert wird ein wert >= 0
}


class MyZusicherung {
  public static void main(String args[]) {
    Quadrat q = new Quadrat();
    System.out.println(q.flaeche(9d));     // in Ordnung
    System.out.println(q.flaeche(0d));     // Vertragsverletzung
  }
}
```

Diese Variante hat den Nachteil, dass nur der Entwickler die Zusicherungen verifizieren kann. Wie bereits erwähnt ist der Client für die Einhaltung der Vorbedingungen verantwortlich, definiert werden Zusicherungen aber beim Server.
Problematisch kann die Situation werden, wenn der Server im oberen Beispiel seine Zusicherung verstärkt (von seite >= 0 auf seite > 0) und Klienten auf die alte Vorbedingung vertraut haben. Eine manuelle Überprüfung aller Klienten auf die neue Vertragssituation verursacht nicht nur sehr viel Aufwand, es können auch Vertragsverletzungen leicht übersehen werden. Hier können automatische Testfälle, die natürlich ebenfalls an die neue Situation angepasst sein müssen, unterstützend wirken.
Eine andere Möglichkeit wäre, die Überprüfung der seite > 0 direkt als if-Anweisung in der Methode flaeche() zu implementieren.

5.9.2 Anweisung assert in Java

Seit J2SE 1.4 werden in Java Zusicherungen durch die assert-Anweisung unterstützt.

Beispiel 5.9.2 *[Anweisung assert in Java]*

```
class Quadrat {
  // berechnet die Flaeche eines Quadrates;
  public double flaeche(double seite) {
    assert (seite > 0) : "Seite muss > 0 sein";
```

```
    double wert;
    wert = seite * seite;
    assert (wert / seite == seite) : fehler();
    return wert;
  }

  // retourniert Standardfehlermeldung
  private String fehler() { return "Interner Fehler"; }
}

class MyAssert {
  public static void main(String args[]) {
    Quadrat q = new Quadrat();
    System.out.println(q.flaeche(9d));  // in Ordnung
    System.out.println(q.flaeche(0d));  // Vertragsverletzung
  }
}

// compilieren mit Option -source 1.4; ausfuehren mit Option -ea
```

An dieser Stelle sei nochmals bemerkt, dass die Verwendung der assert-Anweisung zumindest in der J2SE 1.4 eine besondere Berücksichtigung beim Ausführen von Java-Programmen verlangt, siehe Abschnitt 3.3.10, Seite 30. Zu Beginn und am Ende der Methode flaeche() wurden die Kommentare durch assert-Anweisungen ersetzt. Wenn zur Laufzeit die Behauptungen aktiviert sind und eine Zusicherung nicht erfüllt wird, so wird wie bei einer Ausnahmebehandlung eine Ausnahme geworfen.

Im Allgemeinen sind solche Meldungen in einer Produktionsumgebung unerwünscht, wodurch beim Ausführen der Java-Programme die assert-Option deaktiviert wird. Damit zumindest die Vorbedingungen in öffentlichen Methoden gewährleistet sind, empfiehlt es sich, diese mit if-Anweisungen zu implementieren. Nachbedingungen sowie Invarianten sollten mit assert-Anweisungen codiert werden.

Der Programmierer muss darauf achten, dass er keine Seiteneffekte mit den assert-Anweisungen produziert. Diese würden je nach Aktivierung oder Deaktivierung der Zusicherungen unterschiedliche Verhaltensweisen bei der Programmausführung bewirken. Diese Eigenschaft kann aber auch im positiven Sinne genutzt werden, um beispielsweise festzustellen, ob die Zusicherungen aktiv sind oder nicht:

Beispiel 5.9.3 *[Verifizieren, ob in Java Zusicherungen aktiv sind]*

```
class MyAssert2 {
  public static void main(String args[]) {
    boolean bAssert = false;

    assert(bAssert = true);
    if (!bAssert)
```

```
      System.out.println("Zusicherungen nicht aktiviert!");
  }
}

// compilieren mit Option -source 1.4; ausfuehren mit Option -ea
```

Ein weiterer Nachteil neben der Ausgabe in Form einer Ausnahmebehandlung ist, dass Vor- und Nachbedingungen nicht an abgeleitete Klassen vererbt werden, falls diese die Methode überschreiben. Zusicherungen mit der assert-Anweisung sind in Interfaces auf Grund der Abstraktheit in Java auch nicht möglich. Stattdessen könnten Zusicherungen in Interfaces in Form von Kommentaren formuliert werden.
Schlussendlich gilt es noch zu beachten, dass Java-Bytecode mit assert-Anweisungen (Spracherweiterung), der mit dem Compiler des SDK 1.4 oder höher erzeugt wurde, auf älteren JVM nicht einsatzfähig ist.

5.9.3 Alternativen in Java

Es gibt die Möglichkeit, Zusicherungen in Form von JavaDoc-Kommentaren[14] im Quellprogramm zu formulieren, diese werden dann von einem Precompiler in den Java-Quellcode übersetzt. Dieser Vorgang wird als Instrumentierung bezeichnet. Vorteile sind

- die Nachvollziehbarkeit des durch den Precompiler generierten Java-Codes,
- das Quellprogramm kann ohne Zusicherungen (ohne Precompiler) direkt compiliert werden und
- die automatische Übernahme der Zusicherungen in die Dokumentation.

Wie bei der assert-Anweisung können aber auch hier Seiteneffekte bösartige Überraschungen liefern. Der vom Precompiler instrumentierte Java-Code könnte bei der Durchführung ein anderes Verhalten aufweisen als bei direkter Übersetzung. Als zusätzlicher Nachteil muss erwähnt werden, dass Zusicherungen bei nicht verfügbaren Quellprogrammen, zum Beispiel bei einer externen Programmbibliothek, nicht durchgeführt werden können.

Eines dieser Werkzeuge, die Zusicherungen mit Hilfe eines Precompilers zur Verfügung stellen ist *iContract* [Kramer, 2003]. Dieses freie Werkzeug bietet Vor-, Nachbedingungen und Invarianten nicht nur in Klassen, sondern auch in Interfaces an. In iContract können Zusicherungen auch vererbt werden. Ein Nachteil dieses Precompilers ist, dass er derzeit nicht gemeinsam mit GJ [Generic Java] genutzt werden kann.

In der Arbeit von [Lackner, 2001] wird gezeigt, wie Java um die beiden Mechanismen Zusicherungen und Generizität mit FJC[15] erweitert werden kann. Das folgende Beispiel aus dieser Arbeit zeigt den Einsatz von Vorbedingungen (require), Nachbedingungen (ensure) und Invarianten (invariant) in Interfaces. In der Nachbedingung wird mit dem Operator @@() auf das Ergebnis der Methode zugegriffen.

[14] Ist ein Werkzeug von Sun Microsystems, das eine standardisierte Dokumentation in verschiedenen Ausgabeformen (HTML, PS, XML, etc.) erzeugt.
[15] FJC ist eine Erweiterung des Kopi Java Compilers (http://www.dms.at/kopi/).

Beispiel 5.9.4 *[Zusicherungen in Java-Interfaces]*

```
public interface Account {
  int MAX_OVERDRAWN = -30000;

  invariant {
    assert current() > MAX_OVERDRAWN;
  }

  int current();

  int deposit(int deposit);
  require {
    assert deposit > 0;
  }

  int withdraw(int withdrawall};
  ensure {
    assert @@() > MaX_OVERDRAWN;
  }

  ...
}
```

Dadurch, dass FJC ein Compiler ist und den Quellcode nicht in einen weiteren Quellcode wie iContract übersetzt, können Vorbedingungen in einem Konstruktor noch vor dem Aufruf des Superkonstruktors überprüft werden.

5.9.4 Selbstdefinierte assert-Methode in Ruby

Die Programmiersprache Ruby mit ihrem dynamischen Typkonzept unterstützt das Konzept Design by Contract nicht. Es gibt auch keine assert-Anweisung wie in Java. Somit können standardmäßig nur Kommentare für Zusicherungen eingesetzt werden. Eine selbstdefinierte assert-Methode mit Hilfe der Ausnahmebehandlung (siehe Abschnitt 4.3.12, Seite 62) könnte in Ruby wie folgt aussehen:

Beispiel 5.9.5 *[assert Methode in Ruby]*

```
def assert(bedingung)
  return if not ASSERTjn
  begin
    raise Exception, "Zusicherungsfehler" if not bedingung
  rescue Exception => e
    filename = e.backtrace()[1].split(/:/)[0]
    lineno = e.backtrace()[1].split(/:/)[1].to_i()
    file = File::new(filename)
```

```
    puts e.message + ": " + file.readlines()[lineno-1]
    puts "File: " + filename + ", Zeile: " + lineno.to_s
    exit! 1
  end
end

ASSERTjn = ARGV.include?("-ea")
assert(0<1)
assert(0<0)
p "Code wird nicht ausgefuehrt wenn -ea gesetzt ist!"

# ruby -w MyAssert.rb -ea
```

In der Methode assert(bedingung) wird zuerst die globale Variable ASSERTjn auf den Wert false überprüft. Ist dies der Fall, endet die Methode unmittelbar. Ist der Wert von ASSERTjn jedoch true, wird die als Parameter übergebene Bedingung überprüft. Liefert die Bedingung false, wird eine Ausnahme mit der Meldung „Zusicherungsfehler“ geworfen. Diese analysiert den aktuellen backtrace, um neben dem Programmnamen und der Zeilennummer auch die auslösende Programmzeile ausgeben zu können. Danach terminiert das Programm mit dem Returnwert 1.

Je nachdem, ob das Programm mit dem Argumentwert „-ea“ aufgerufen wurde oder nicht, wird die globale Variable ASSERTjn mit true oder false initialisiert. Die Argumente der Befehlszeile können in Ruby über das globale Array ARGV[16] angesprochen werden.

5.9.5 Zusicherungen mit dbc.rb

Eine Möglichkeit, DbC in Ruby einzusetzen, bietet die Bibliothek dbc.rb [Hunt, 2003]. Mit dieser Bibliothek können Codeblöcke für Vor-, Nachbedingungen und Invarianten definiert werden.

Beispiel 5.9.6 *[Zusicherung mit dbc.rb]*

```
require 'dbc'

class Quadrat
  def flaeche(seite)
    pre  { seite > 0 }
    post { DbC::Runtime.result / seite == seite }
    seite * seite
  end
end

q = Quadrat.new
p q.flaeche(9)   # alle Zusicherungen erfuellt
```

[16]Hinweis: Das erste Argument an das Programm wird mit ARGV[0] angesprochen, anders als in einigen anderen Programmiersprachen befindet sich der Programmname in der globalen Variable $0.

```
p q.flaeche(0)   # Vorbedingung nicht erfuellt
```

Mit require wird die Bibliothek dbc geladen. Die Blöcke für die Vorbedingung (pre) und Nachbedingung (post) müssen vor dem Methodencode definiert werden. Das Ergebnis der Methode kann mit dem Aufruf DbC::Runtime.result im post-Block verifiziert werden.

Bei Verwendung einer Methode invariant sollte diese am Ende einer Klasse definiert werden. Die Invariante wird nach jedem Ausführen einer öffentlichen Instanzmethode in dieser Klasse automatisch aufgerufen und muss immer true liefern, gleichgültig ob eine Methode regulär oder vorzeitig endet.

Für einfache Prüfungen, ähnlich der assertion() Methode in Java, ist in dbc.rb die Methode check[17] vorgesehen:

Beispiel 5.9.7 *[Methode check in dbc.rb]*

```
require 'dbc'

a, b = 4, 0
p a / b unless check("Division durch 0") { b != 0 }
```

Wenn die Bedingung einer der Methoden check, invariant, pre oder post false liefert, wird eine Ausnahme geworfen, die den Namen der Verletzung (pre, post, ...), den Programmnamen und die Zeilennummer ausgibt. Nach der Ausnahme darf das Objekt laut [Hunt, 2003] nicht mehr verändert werden. Begründet ist dies damit, dass sich das Objekt in einem inkonsistenten Zustand befindet und darauf angewandte Änderungen die Erkennung weiterer Vertragsverletzungen unmöglich machten.

Ruby ist eine sehr dynamische Sprache, in der Objekte starken Änderungen unterliegen. Zum Beispiel können mit der globalen Methode eval Zeichenketten mit Programmcode compiliert und ausgeführt werden. Die Bibliothek dbc.rb analysiert jedoch nur den Quellcode und unterstützt daher Zusicherungen in Zeichenketten, die mit der Methode eval ausführbar sind, nicht.

Standardmäßig sind alle Vor-, Nachbedingungen, Invarianten und die Methode check aktiviert. Individuell eingeschaltet werden können diese mit den folgenden, miteinander durch *or* verknüpfbaren, Optionen:

- *DbC::Runtime::PRE* aktiviert die Überprüfung von Vorbedingungen
- *DbC::Rumtime::POST* aktiviert die Überprüfung von Nachbedingungen
- *DbC::Runtime::INV* aktiviert die Überprüfung von Invarianten

[17] In der Version 1.7 von dbc.rb unter Windows ist ein Fehler in der Methode check enthalten. Der Aufruf von raise mit dem Argument caller[1] gehört auf caller[0] korrigiert. Durch caller[0] wird die direkt aufrufende Methode auf den backtrace Stapel gelegt.

- *DbC::Runtime::CHECK* aktiviert die Zusicherungen durch die check Methode
- *DbC::Runtime::NONE* deaktiviert alle Zusicherungen

Durch den Aufruf von DbC::Runtime.enable werden alle genannten Zusicherungen wieder aktiviert.

Im folgenden Beispiel wird gezeigt, welche Probleme mit der Bibliothek dbc.rb bei der Verwendung von Vererbung auftreten können:

Beispiel 5.9.8 *[Vererbung mit dbc.rb]*

```
require 'dbc'

class A
  @@MINWERT = 0

  def initialize(wert = @@MINWERT)
    @wert = wert
    invariant
  end

  def inkrement
    post { @wert == old(@wert) + 1 }
    @wert += 1
  end

  def dekrement
    pre { @wert > @@MINWERT }
    post { @wert == old(@wert) - 1 }
    @wert -= 1
  end

  protected

  def istGueltig?
    @wert >= @@MINWERT
  end

  def invariant
    istGueltig?
  end
end

class B < A
  @@MINWERT = 1
  @@MAXWERT = 8

  def inkrement
```

```
    pre { @wert < @@MAXWERT }
    post { @wert == old(@wert) + 1 }   # nochmals zu definieren
    @wert += 1                         # super liefert SystemStackError
  end

  def istGueltig?
    super && @wert <= @@MAXWERT
  end
  protected :istGueltig?
end

DbC::Runtime.enable # aktiviere alle Zusicherungen

a=A.new           # wert = 0
a.inkrement       # wert = 1
a.dekrement       # wert = 0

b=B.new           # wert = 1
7.times do
  b.inkrement
end               # wert = 8
begin
  b.inkrement
rescue
  puts "Ausnahme durch Vorbedingung in B!"
end

# neues Object, da Objekt inkonsistent ist
b=B.new           # Wert = 1
begin
  b.dekrement
rescue
  puts "Ausnahme durch Vorbedingung in A!"
end

begin
  b=B.new(9)
rescue
  puts "Ausnahme durch Invariante in A!"
end
```

In der Klasse A wird die Klassenvariable MINWERT mit dem Wert 0 initialisiert. Die protected Methode invariant am Ende der Klasse ruft wiederum die protected Methode istGueltig? auf. In dieser wird der aktuelle Wert der Instanzvariablen wert auf den Mindeswert von MINWERT überprüft. Wird der Konstruktor initialize ohne Parameter aufgerufen, so erhält die Instanzvariable wert den Standardwert MINWERT, ansonsten den Wert des mitgelieferten Parameters. Zu beachten ist, dass die protected Methode invariant nach dem Konstruktor von dbc.rb nicht automatisch aufgerufen wird, weshalb dies hier explizit getan werden muss.

Kein expliziter Aufruf von invariant ist bei den öffentlichen Methoden inkrement und dekrement, die den Wert der Instanzvariablen wert jeweils um eins erhöhen bzw. verringern, notwendig. In der Methode inkrement überprüft die Nachbedingung (post), ob sich der Inhalt von wert gegenüber dem ursprünglichen Inhalt (old(wert)) um eins erhöht hat. Die Methode old wird von der Bibliothek dbc.rb zur Verfügung gestellt und darf nur in einer Nachbedingung verwendet werden. Damit die Variable wert nicht unter dem Wert von MINWERT gelangt, wird in der Methode dekrement die entsprechende Vorbedingung (pre) definiert.

Die Klasse B ist eine Subklasse von A, die den Wert der Klassenvariablen MINWERT mit 1 und die neue Klassenvariable MAXWERT mit 8 definiert hat. Die Methode istGueltig? wird in der Subklasse B durch die zusätzliche Abprüfung einer Obergrenze (MAXWERT), überschrieben. Damit die Methode nicht öffentlich sichtbar ist, muss die Sichtbarkeit explizit auf protected gesetzt werden. Weiters wird die öffentliche Methode inkrement in der Subklasse B überschrieben. Eigentlich müsste in dieser Methode nur die neue Vorbedingung definiert werden, jedoch anders als in [Hunt, 2003] beschrieben, wird die Nachbedingung von der Superklasse A nicht exekutiert, womit sie hier redundant definiert werden muss. Auch der Hauptteil der Methode dekrement musste hier nochmals codiert werden, da der Aufruf von super zu einem SystemStackError in dbc.rb führt. Eine Möglichkeit letzteres Problem zu umgehen wäre eine zusätzliche protected Methode ohne Vor- und Nachbedingungen in der Superklasse A zu definieren. Diese zusätzliche, von dbc.rb nicht observierte Methode, kann ohne SystemStackError und Coderedundanz von beiden Klassen aufgerufen werden.

Bei Verwendung von dbc.rb werden Vorbedingungen in Superklassen von Vorbedingungen in Subklassen überschrieben. Dadurch können Vorbedingungen in Subklassen abgeschwächt werden. Im Gegensatz dazu werden Invarianten und Nachbedingungen in Superklassen laut [Hunt, 2003] auch in allen Subklassen überprüft, womit Nachbedingungen nur verschärft werden können. Die im oberen Beispiel dargestellte Situation zeigt, dass dies bei der derzeit verfügbaren Version 1.7 von dbc.rb nicht zutrifft. Sobald eine Vor- *oder* eine Nachbedingung in einer überschriebenen Methode verwendet wird, werden *beide* Bedingungen in der Superklasse ignoriert und müssen daher in der Subklasse nochmals definiert werden. Diese Unstimmigkeiten und Nachteile führen dazu, dass der Einsatz dieser Bibliothek derzeit nicht zu empfehlen ist. Der verfügbare Quellcode bietet jedoch die Möglichkeit, eigene Korrekturen und Ergänzungen durchzuführen.

5.9.6 Fazit

Die Anwendung des Design by Contract Konzeptes ist in der Programmiersprache Java bis zur Version 1.3 nur mit Hilfe von Kommentaren möglich. Seit Version 1.4 gibt es die assert-Anweisung, die eigentlich nur wie eine besser schreibbare Ausnahmebehandlung agiert. Der markanteste Unterschied zwischen Ausnahmebehandlungen und Zusicherungen ist die Option, die beim Compilieren bzw. Ausführen von assert-Anweisungen anzugeben ist. Diese bringt aber mehr Nachteile wie Seiteneffekte und Versionsabhängigkeit mit sich als Vorteile.

Vorbehaltlich der Verfügbarkeit des Quellcodes stellt die Instrumentierung des Java-Codes in Form von JavaDoc-Kommentaren eine bessere Möglichkeit dar. Jedoch kann auch hier mit Seiteneffekten ein unangenehmes Verhalten erwirkt werden.
Bei den alternativen Möglichkeiten für Java hat sich gezeigt, dass die Spracherweiterung FJC den gewünschten Anforderungen am besten entspricht.

In Ruby gibt es für Zusicherungen keine direkte Unterstützung. Es gibt weder eine assert-Anweisung noch ist die verfügbare Bibliothek dbc.rb sinnvoll einsetzbar. Hier muss der Entwickler zur Selbsthilfe greifen und sich eine assert-Methode programmieren oder den frei verfügbaren Code der dbc.rb Bibliothek überarbeiten.
Die Formulierung von Zusicherungen in Form von Kommentaren müsste sehr konsequent durchgeführt werden. Dies ist jedoch nur dann möglich, wenn ein hohes Maß an Selbstdisziplin bei den Programmierern gegeben ist. Dies führt zu dem Schluss, dass die Umsetzung des DbC Konzeptes in der Programmiersprache Ruby in den gezeigten Variationen zu mehr Nachteilen als Vorteilen führt.

5.10 Untersuchungsergebnis

5.10.1 Sprachspezifika

Bei der Erstellung dieser Arbeit wurden zahlreiche Sprachunterschiede zwischen Java und Ruby erkannt. Diese stellten die Grundlage für die Bewertung der objektorientierten Konzepte dar.

Java

- Compiler (und Interpreter)
- Statische Typen
- Primitive Datentypen
- Typumwandlung (casting) möglich
- Keine globalen Variablen
- Arrays können nur Objekte eines Typs aufnehmen
- Arrays haben eine fixe Größe
- Pakete, Import Anweisung
- Call by Value
- Beim Konstruktor muss super als erste Anweisung stehen
- Sichtbarkeit des Konstruktors kann auch private sein
- Unicodeunterstützung
- Finalisieren von Variablen
- Routine main()

Ruby

- Interpreter
- Dynamische Typen
- Fixnum, true, false und nil sind keine Referenzen
- Nur nil und false liefern als einzige false Werte, 0 liefert true
- Zahlenwert kann beliebig gross werden (Bignum)
- Globale Variablen

- Anweisungen ohne abschliessenden Strichpunkt
- Leerzeichen haben syntaktische Bedeutung
- Klassen sind offen
- Iteratoren statt eingebauter Schleifen
- Blöcke
- Zeichensatz 7-bit ASCII
- Call by Reference
- Metaklassen, Metaobjekte
- Methodenaufruf mit variabler Anzahl an Parametern möglich
- Reguläre Argumente mit Defaultwerten
- Mit super ist ein Zugriff auf Variablen nicht möglich
- Viele Operatoren sind Methoden
- Attributdeklaration (setter/getter-Methoden)
- Klassennamen sind Konstanten
- Klassenkonstanten sind immer public
- Sicherheitskonzept (tain Objekte)

5.10.2 Vergleichstabelle

Objektorientierte Konzepte	**Java**	**Ruby**
Einfachvererbung	√	√
Ersetzbarkeitsprinzip	√	–
Mehrfachvererbung	– [Interfaces]	+ [Mixins]
Delegation	–	+
Informationsverheimlichung	√	–
Abstrakte Klassen	+	√ [Module]
Objektspezialisierung	N/A	√
Finalisierung, Einfrieren	+	√
Überladen	√	√ [Dispatching]
Überschreiben	√	+
Design by Contract	+ [FJC]	– [dbc.rb]

Tabelle 5.1: Anwendbarkeit objektorientierter Konzepte

N/A	nicht anwendbar
–	schlecht anwendbar
√	anwendbar
+	gut anwendbar
[ ]	Alternativkonzept

Die Vergleichstabelle 5.1 stellt die Programmiersprachen Java und Ruby gegenüber und bewertet die in dieser Arbeit untersuchten objektorientierten Konzepte bezüglich Anwendbarkeit.
Die Bewertung in der Vergleichstabelle hat sich auf Grund folgender Erkenntnisse ergeben:

- *Einfachvererbung* wird von beiden Programmiersprachen unterstützt. Sowohl Java als auch Ruby bieten die Möglichkeit, beim Überschreiben von Methoden auf die ursprüngliche Methode der direkten Superklasse zuzugreifen. Es fehlt jedoch die Möglichkeit, auch auf Superklassen zuzugreifen, die in der Vererbungshierarchie weiter oben liegen.

- *Ersetzbarkeitsprinzip* ist in Java derzeit stärker eingeschränkt als es notwendig wäre. Ruby bietet auf Grund seines dynamischen Typsystems in der Praxis nur bedingt anwendbare Möglichkeiten an.

- *Mehrfachvererbung* von Klassen ist in Java und Ruby nicht möglich. Die Verwendung von Interfaces für die Mehrfachvererbung hat den Nachteil, dass es zu unnötigen identen Codeimplementierungen kommen kann. Ruby bietet mit den mixins auch die Möglichkeit, definierte Methoden in Klassen zu inkludieren.

- *Delegation* konnte in Java nur mit den Interfaces realisiert werden. Neben dem Problem der Coderedundanz kann es wegen fehlender Beziehungen zwischen den Typen zu Typinkonsistenzen kommen. In Ruby konnten sogar mehrere gute anwendbare Lösungen gefunden werden.

- *Informationsverheimlichung* mit den Zugriffskontrollen private und protected besitzen beide Sprachen. Die Möglichkeit, in Ruby aus einer Subklasse eine private Methode der Superklasse aufzurufen sowie die beliebige Veränderung der Zugriffsrechte in Subklassen, wurde negativ bewertet. Die Bewertung für Java fiel neutral aus, da es noch keine einfache automatische Schreibweise für setter/getter-Methoden gibt.

- *Abstrakte Klassen* sowie reine abstrakte Klassen (Interfaces) werden in Java direkt unterstützt. In Ruby konnte gezeigt werden, dass mit Modulen ein ähnliches Verhalten simuliert werden kann.

- *Objektspezialisierung* bedeutet, ein spezielles Objekt zur Laufzeit zu verändern. Dies ist in Java nicht möglich. Die Lösung in Ruby, dies mit Singletonklassen zu realisieren, wurde als schwer verständlich empfunden.

- *Finalisieren* von Klassen, Methoden und Variablen ist in Java sehr einfach möglich. In Ruby können dagegen nur Objekte eingefroren werden. Das Einfrieren kann im Zusammenhang mit den reflexiven Operatoren jedoch zu Verständlichkeitsproblemen führen.

- *Überladen* von Methoden wird nur in Java unterstützt. Dieses objektorientierte Konzept ist jedoch auf Grund der Mehrdeutigkeit der Methodensignaturen nicht gut anwendbar. In Ruby kann als Alternative nur Dispatching angewendet werden.

- *Überschreiben* ist sowohl in Java als auch in Ruby möglich. In Java können aber nur Methoden überschrieben werden, deren Signatur mit der zu überschreibenden Methode übereinstimmt. In Ruby wird bereits bei Namensgleichheit die Methode überschrieben. Zusätzlich können auch Operatoren redefiniert werden.

- *Design by Contract* ist in Java mit der Spracherweiterung JFC gut umsetzbar. In Ruby wurden in der Bibliothek dbc.rb einige Fehler und Probleme entdeckt, wodurch deren Einsatz nicht zu empfehlen ist.

Kapitel 6

Zusammenfassung

In dieser Arbeit wurde untersucht, wie gut die am häufigsten eingesetzten objektorientierten Konzepte in Java und Ruby anwendbar sind. Zum Abschluss werden nun die wichtigsten Fakten zusammengefasst.

Bei der Erstellung von Software muss sichergestellt werden, dass diese verlässlich, wartbar und effizient durchführbar ist. Je nach Berücksichtigung dieser Qualitätsanforderungen kann nun qualitativ bessere oder schlechtere Software entwickelt werden.

Eine wichtige Rolle für die Erstellung qualitativ hochwertiger Software spielt das objektorientierte Paradigma. Die Umsetzung dieses Paradigmas in einer Programmiersprache stellt für die Entwickler einer Sprache eine große Herausforderung dar. Dem Programmierer obliegt es dann, dieses so gut wie möglich in seinen Programmen anzuwenden. Daher ist es wichtig, die allgemeinen Konzepte einer Sprache zu kennen und zu wissen, inwieweit mit den vorhandenen Möglichkeiten die Anwendung unterstützt wird. Dem wurde in den Kapiteln drei für Java und vier für Ruby Rechnung getragen. In diesen Kapiteln wurden unter anderem die Konzepte der Typen, Variablen, Operatoren, Methoden und Klassen beschrieben. Bei diesem Prozess wurden einige Sprachunterschiede erkannt und in Abschnitt 5.10.1, Seite 128 festgehalten.

Danach wurde mit diesen Werkzeugen nach einer Lösung für die Umsetzung des Sprachkonzeptes gesucht. Die daraus resultierenden Ergebnisse wurden in den Fazits der jeweiligen Kapitel zusammengefasst. Nachdem die Ergebnisse zur Verfügung standen, wurden diese einer Bewertung unterzogen. Die resultierende Vergleichstabelle sowie eine kurze Erläuterung der Bewertung findet sich im Abschnitt 5.10.2, Seite 129 wieder.

Das objektorientierte Paradigma bietet zahlreiche Konzepte an, die je nach Aufgabenstellung unterschiedlich zum Tragen kommen. Um qualitativ hochwertige Programme zu erstellen, sind die Konzepte *Einfachvererbung*, *Ersetzbarkeit*, *Informationsverheimlichung*, *Abstrakte Klassen*, *Überschreiben* und *Design by Contract* von besonderer Bedeutung.
In beiden Sprachen ist die *Einfachvererbung* ähnlich gut umsetzbar. Eines der wichtigsten Konzepte im objektorientierten Paradigma ist das *Ersetzbarkeitsprinzip*. Ruby ist diesbezüglich auf Grund seines dynamischen Typsystems stark benachteiligt, womit die Wiederverwendbarkeit von Programmcode geringer ist als in Java. Das Konzept der *Informationsverheimlichung* bietet in Ruby zuviel Spielraum, wodurch der gewünschte Schutz nicht immer den Erwartungen entspricht. Ruby bietet kein Konzept für *Abstrakte Klassen*

an. Es müssen trickreiche Simulationen, die aber nur bedingt anwendbar sind, herangezogen werden. Das *Überschreiben* von Methoden ist in Ruby einfacher anwendbar als in Java, da es zu keinen Missverständnissen auf Grund der Methodensignaturen kommen kann. Durch den Einsatz von *Design by Contract* erhöht sich die Zuverlässigkeit und Sicherheit des entwickelten Codes. Eine gute Lösung für Java konnte mit der Spracherweiterung FJC gefunden werden. In Ruby konnte keine anwendbare Lösung gefunden werden. Die Bibliothek dbc.rb wurde dafür als zu fehlerhaft erachtet.

Die beiden Programmiersprachen Java und Ruby sind relative junge Sprachen, die sich in den nächsten Versionen stark weiterentwickeln werden. Beispielsweise wird in der kommenden Version von Java die Generizität eine wichtige Rolle einnehmen, und in den Ruby-Foren wird viel über die Möglichkeiten von strong-typing diskutiert. Ob diese Spracherweiterungen eine Sprache unnötig komplex machen oder steigende Programmqualität bewirken, wird die Zukunft zeigen.

Literaturverzeichnis

[Abardi et al., 1996] Martin Abadi, Luca Cardelli, *A Theory of Objects*, Springer Verlag, ISBN 0-387-94775-2, 1996.

[Albahari, 2000] Ben Albahari, *A Comparative Overview of C#*, http://genamics.com /developer/csharp_comparative.htm, Download 27. November 2003.

[Arnold et al., 2000] Ken Arnold, James Gosling, David Holmes, *The Java Programming Language* 3rd edition, Addison-Wesley, ISBN 0-201-70433-1, 2000.

[Bloch, 2001] Joshua Bloch, *Effective Java*, Addison Wesley, ISBN 0-201-31005-8, 2001.

[Bracha et al., 1990] Gilad Bracha, William Cook, *Mixin-based Inheritance*, OOPSLA/ECOOP '90 Conference Proceedings, ACM SIGPLAN Notices, pages 303-311, October 1990.

[Bracha, 2001] Gilad Bracha, *Add Generic Types To The Java Programming Language*, http://jcp.org/en/jsr/detail?id=14, Download 29. Oktober 2003.

[Cardelli et al., 1985] Luca Cardelli, Peter Wegner, *On understanding types, Data Abstraction, and Polymorphism*, Computing Surveys, Vol. 17, No. 4, December 1985.

[Craig, 2002] Iain Craig, *The Interpretation of Object-Oriented Programming Languages, 2nd edition*, Springer Verlag London Limited 2002, ISBN 1-85233-547-5, 2002.

[Crespo et al., 2002] Yania Crespo, José Manual Marqués, Juan José Rodriguez, *On the Translation of Multiple Inheritance Hierarchies into Single Inheritance Hierarchies*, http://www.cs.jyu.fi/~sakkinen/inhws/papers/Crespo.pdf, Download 9. Oktober 2003.

[D'Souza, 1997] Desmond D'Souza, *Types and classes: A language-independent view*, Journal of Object-Oriented Programming (JOOP), pages 10-13, vol. 10, no. 1, March/April, 1997.

[Ecma-334, 2002] Standard Ecma-334 *C# Language Specification* 2nd edition, (Dec. 2002), http://www.ecma-international.org/publications/standards/Ecma-334.htm, Download 26. November 2003.

[Flanagan, 2002] David Flanagan, *Java in a nutshell* 4th edition, O'Reilly & Associates, ISBN 0-596-00283-1, 2002.

[Friedl, 2002] Jeffrey E.F. Friedl, *Mastering Regular Expression* 2nd edition, O'Reilly & Associates, ISBN 0-596-00289-0, 2002.

[Fulton, 2001] Hal Fulton, *The Ruby Way*, Sams Publishing, ISBN 0-672-32083-5, 2001.

[Gamma et al., 1995] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns*, Addison Wesley Longman, Inc., ISBN 0-201-63361-2, 1995.

[Gellner, 2001] Michale Gellner, *Der Umgang mit dem Hoare-Kalkül zur Programmverifikation*, http://wwwswt.informatik.uni-rostock.de/deutsch/Mitarbeiter/michael /lehre/pt_2001/Hoare_akt_008.pdf, Download 9. Oktober 2003.

[Generic Java] *A Generic Java Language Extension*, http://www.cis.unisa.edu.au/~pizza /gj/, Download 20. Oktober 2003.

[Ghezzi et al., 1998] Carlo Ghezzi, Mehdi Jazayeri, *Programming language concepts* 3rd edition, John Wiley & Sons, ISBN 0-471-10426-4, 1998.

[Gosling, 1995] *Java: an Overview*, http://java.sun.com/people/jag /OriginalJavaWhitepaper.pdf, Download 24. April 2003.

[Halbert et al., 1987] Daniel C. Halbert, Patrick D. O'Brien, *Using Types and Inheritance in Object-Oriented Languages*, ECOOP 1987, LNCS 276, pages 20-31, 1987.

[Hoare, 1974] Hoare, *C.A.R. Monitors: An Operating System Structuring Concept*, Communications of the ACM (CACM) 10:549-557, October 1974.

[Hunt, 2003] Andrew Hunt, *Design By Contract in Ruby*, http://people.cs.uchicago.edu /~songyanf/research/contract_paper/other/dbc_ruby.pdf, Download 9. Oktober 2003.

[Hürsch, 1994] Walter L. Hürsch, *Should Superclasses be Abstract?*, ECOOP '94, September 1994.

[Jézéquel et al., 1997] Jean-Marc Jézéquel, Bertrand Meyer, *Design by Contract: The Lessons of Ariane*, http://archive.eiffel.com/doc/manuals/technology/contract/ariane /page.html, Download 9. Oktober 2003.

[Kernighan et al., 1988] Brian W. Kernighan, Dennis Ritchie, *The C Programming Language* 2nd edition, Prentice Hall, ISBN 0-131-10362-8, 1988.

[Kramer, 2003] Reto Kramer, *iContract – the Java Design by Contract Tool*, Proc. Technology of Object-Oriented Languages and Systems (TOOLS-USA), pages 295-307, 1998.

[Lackner, 2001] Martin Lackner, *Extending Java*, Diplomarbeit am Institut für Computersprachen, Abteilung für Programmiersprachen und Übersetzerbau der Technischen Universität Wien, Mai 2001.

[LaLonde et al., 1991] Wilf LaLonde, John Pugh, *Subclassing ≠ subtyping ≠ Is-a*, Journal of Object-Oriented Programming (JOOP), pages 57-62, vol. 3, no. 5, January 1991.

[Liebermann, 1986] Henry Lieberman, *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems*, Proceedings of First ACM Conference on OOPSLA, Portland, September 1986.

[Liskov, 1988] Barbara Liskov, *Data Abstraction and Hierarchy*, ACM SIGPLAN Notices, pages 17-34, vol. 23, no. 5, October 1988.

[Matz, 2002] Yukihiro Matsumoto, *Ruby in a Nutshell*, O'Reilly & Associates, ISBN 0-596-00214-9, 2002.

[Meyer, 1992] Bertrand Meyer, *Applying "Design by Contract"*, IEEE Computer, pages 40-51, vol. 25, no. 10, October 1992.

[Meyer, 1997] Bertrand Meyer, *Object-Oriented Software Construction* 2nd edition, Prentice-Hall, ISBN 0-136-29155-4, 1997.

[Meyer, 2001] Bertrand Meyer, *Overloading vs. Object Technology*, Journal of Object-Oriented Programming (JOOP), pages 3-7, vol. 14, no. 3, Octorber/November 2001.

[Mingardi, 2002] Paul Mingardi, *Preparing for Generics*, http://developer.java.sun.com /developer/technicalArticles/releases/generics/ Download 29. Oktober 2003.

[Porter, 1992] Harry H. Porter, III, *Separating the subtype hierarchy from the inheritance of implementation*, Journal of Object-Oriented Programming (JOOP), pages 20-29, vol. 4, February, 1992.

[Puntigam, 2002] Franz Puntigam, *Skriptum zu Objektorientierte Programmierung*, Abteilung für Programmiersprachen und Übersetzerbau der Technischen Universität Wien, 2002.

[Singh, 1995] Ghan Bir Singh, *Single Versus Multiple Inheritance in Object Oriented Programming*, OOPS Messenger Vol.6, No. 1, January 1995.

[Stroustrup, 1997] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, ISBN 0-201-88954-4, 1997.

[Tempero et al., 1998] Ewan Tempero, Robert Biddle, *Simulating Multiple Inheritance in Java*, Victoria University of Wellington, School of Mathematical and Computing Sciences, Technical Report CS-TR-98/1, May 1998.

[Thomas et al., 2001] David Thomas, Andrew Hunt, *Programming Ruby: the pragmatic programmer's guide*, Addison-Wesley, ISBN 0-201-71089-7, 2001.

[Thompson, 1999] Simon Thompson, *Haskell: the Craft of Functional Programming*, 2nd edition, Addison-Wesley, ISBN 0-201-34275-8, 1999.

[Viega et al., 1998] John Viega, Bill Tutt, Reimer Behrends, *Automated Delegation is a Viable Alternative to Multiple Inheritance in Class Based Languages*, University of Virginia, Technical Report CS-98-03, 1998.

[Wegner, 1990] Peter Wegner, *Concepts and Paradigms of Object-Oriented Programming*, OOPS Messenger, pages 7-87, vol. 1, no. 1, August 1990.