Prof. Dr. Friedrich Steimann

Modul 63612

Objektorientierte Programmierung

LESEPROBE

Fakultät für
Mathematik und Informatik

FernUniversität in Hagen

Inhaltsverzeichnis

Vorwort zur dritten Auflage

Seit der ersten Auflage dieser Lehrveranstaltung im Wintersemester 2006/2007 hat sich viel getan: JAVA als vormals dominierende objektorientierte Programmiersprache hat weiter Grund u. a. an C# verloren und beiden wird zunehmend von JAVASCRIPT der Rang abgelaufen. Alle drei sind nicht nur objektorientiert, sondern haben (mittlerweile) auch starke funktionale Sprachanteile (die sog. Lambda-Ausdrücke), die im Vergleich zur imperativen Programmierung wesentlich kompaktere Programme ermöglichen. JAVA hat zudem seit der Übernahme durch ORACLE viel von seinem Community-Charakter eingebüßt; die zögerliche Weiterentwicklung der Sprache hat JAVA-basierten Konkurrenten wie SCALA oder KOTLIN Raum für Entwicklung gegeben. Ironischerweise sind Vorbild für manche der wichtigsten Weiterentwicklungen alte objektorientierte Programmiersprachen wie SMALLTALK oder EIFFEL; beide spielen in der aktuellen Programmierpraxis zwar nur noch eine Nebenrolle, ihr Erbe prägt aber die heutige Programmierung maßgeblich.

Zum Inhalt

Voraussetzungen

Dieser Lehrtext stellt die objektorientierte Programmierung so dar, dass ein Gegensatz zur klassischen imperativen Programmierung entsteht. Um diesen Gegensatz wahrzunehmen, müssen Sie natürlich die imperative Programmierung kennen, also z. B. die Lehrveranstaltung 01613 „Einführung in die imperative Programmierung“ absolviert haben. Gleichwohl ist dieses Wissen keine Voraussetzung — Sie können mit dem Ihnen vorliegenden Lehrtext die objektorientierte Programmierung auch als erstes lernen.

So sind die formalen Voraussetzungen zur erfolgreichen Bearbeitung dieses Lehrtextes zunächst gering. Insbesondere bringt es Ihnen wenig, wenn Sie schon „Java können“ — im ungünstigsten Fall müssen Sie sogar erst einiges vergessen, um die ersten Lektionen unvoreingenommen verinnerlichen zu können. Wichtig für den Erfolg ist jedoch die grundsätzliche Bereitschaft, sich mit der Programmierung und deren Werkzeugen auseinanderzusetzen — insbesondere wird Ihnen die Lehrveranstaltung nur wenig bringen, wenn Sie nicht bereit sind, die damit verbundenen Programmierübungen wirklich durchzuführen. Wenn alles gutgeht, sollten Ihnen diese Übungen allerdings auch Freude bereiten.

Ziele

Wenn Sie diese Lehrveranstaltung erfolgreich absolviert haben, dann sollten Sie das folgende können:

- objektorientiert denken,
- objektorientiert programmieren,
- die Bestandteile objektorientierter Programmiersprachen erkennen und dabei insbesondere die Bedeutung eines Typsystems richtig einschätzen,
- die Schwächen der objektorientierten Programmierung benennen,

- Kriterien für die Auswahl einer für ein bestimmtes Projekt geeigneten Programmiersprache aufstellen sowie
- Aussagen zum eigenen und zum Programmierstil anderer machen.

Dem ersten Ziel kommt übrigens eine größere Bedeutung bei, als man bei oberflächlicher Betrachtung annehmen würde: Nur wer objektorientiert denkt, schreibt Programme, die das Attribut „objektorientiert“ verdienen. Zwar tun die pureren unter den Vertretern der objektorientierten Programmiersprachen einiges, um ihren Programmierern einen objektorientierten Stil aufzuzwängen, aber wer nicht will, kann diese Schubse auch ignorieren. Wer hingegen objektorientiert in Sprachen programmieren soll, die selbst gar nicht unbedingt objektorientiert sind, der wird enorme Schwierigkeiten bekommen, wenn er nicht wenigstens objektorientiert denken kann. Und nicht zuletzt ist das objektorientierte Denken Voraussetzung für so manch andere Aktivität im Softwareerstellungsprozess, die u. U. noch weit von der Programmierung entfernt ist: So findet die Modellierung von Softwaresystemen heute meistens mit Modellierungssprachen wie der *Unified Modeling Language* (*UML*) statt, die als objektorientiert gelten. Wenn Sie also nicht objektorientiert denken, dann können Sie auch mit solchen Modellen wenig anfangen und werden zu einem entsprechenden Prozess nicht wirklich beitragen können.

kein Programmierkurs!

Eine universitäre Lehrveranstaltung zur objektorientierten Programmierung kann kein Programmierkurs sein. Sie muss sich vielmehr auf die Konzepte der objektorientierten Programmierung konzentrieren und sie im Kontext darstellen, ganz so wie Lehrveranstaltungen zur imperativen, logischen oder funktionalen Programmierung dies auch tun müssen. Programmieren lernt man ohnehin am besten in der Praxis; eine Lehrveranstaltung wie diese soll allerdings helfen, dass dieser Lernprozess informiert, ohne Um- und Irrwege, verläuft.

die Lektionen im Einzelnen

Der Vorkurs bringt einen kurzen praktischen Einstieg in die objektorientierte Programmierung, der Ihnen beim Verständnis des Haupttextes helfen soll. Wichtig für Sie ist hier vor allem, dass Sie sich, wie in den Übungen gefordert, mindestens eine SMALLTALK-Umgebung besorgen und mit dieser vertraut machen. Das Verständnis von weiten Teilen der folgenden Lektionen hängt entscheidend davon ab, ob Sie in der Lage sind, den Stoff praktisch nachzuvollziehen und Ihr Erlerntes an eigenen kleinen Experimenten zu überprüfen. Programmieren ist eine praktische Tätigkeit; es muss geübt werden wie das Feilen in einer Eisenwerkstatt und wer nur glaubt, er hätte alles verstanden, kann noch lange nicht programmieren. Objektorientiertes Programmieren ist davon keine Ausnahme.

Lektion 1 beginnt mit einer ausführlichen Einführung der wenigen Grundkonzepte der objektorientierten Programmierung. Sie behandelt im Wesentlichen Objekte, Variablen, Beziehungen zwischen Objekten, Nachrichten, die zwischen Objekten ausgetauscht werden können, sowie Methoden, die die Reaktion auf den Empfang einer Nachricht festlegen. Auf die Definition von Objekten als Konglomerat von Variablen und Methoden sowie auf die Erzeugung von Objekten, die nicht „schon da“ sind, wird hier nicht eingegangen. Diese strikte Trennung erlaubt es, den Klassenbegriff, der für die meisten objektorientierten Programmiersprachen eine zentrale Rolle spielt, als einen Freiheitsgrad objektorientierter Programmierung darzustellen.

In Lektion 2 wird dann die Klasse als struktur- und ordnungsstiftendes Konstrukt der objektorientierten Programmierung vorgestellt. Eng mit dem Klassenbegriff verbunden sind die Instanziierung von Klassen zu Objekten, der Übergang von Klassen zu ihren Metaklassen sowie die Vererbung zwischen Klassen. Insbesondere die Vererbung hat in der objektorientierten Programmierung eine wechselhafte Geschichte hinter sich; auf die mit ihr verbundenen Probleme wird jedoch erst in den folgenden Lektionen eingegangen. Stattdessen werden in Lektion 2 noch das dynamische Binden und seine Bedeutung als universelle Kontrollstruktur dargestellt.

Mit Lektion 3 wird in das Thema Typsysteme für die objektorientierte Programmierung eingeführt. Dabei wird ausgenutzt, dass SMALLTALK kein Typsystem hat; Typsysteme können hier also ohne Vorbelastung durch eine spezielle Sprache auf die objektorientierte Programmierung aufgepfropft werden. Dabei wird insbesondere auf den für die objektorientierte Programmierung so zentralen Begriff des Subtyping hingearbeitet; aber auch die generischen Typen, die spätestens seit JAVA 5 und C# 2.0 auch zum täglichen Brot kommerzieller Programmierer gehören dürften, werden so eingeführt. Der dabei zum Einsatz kommende SMALLTALK-Nachfolger STRONGTALK wird dazu als Vorlage genommen, aber nur insoweit, als sich die Eigenschaften dessen Typsystems auch in anderen Programmiersprachen wiederfinden.

Ab Lektion 4 werden dann andere Programmiersprachen vorgestellt. Lektion 4 zollt zunächst JAVA als der heute wohl immer noch wichtigsten objektorientierten Programmiersprache Tribut: Die Präsentation orientiert sich dabei an der Einführung in die objektorientierte Programmierung aus den vorangegangenen Lektionen. Dies erlaubt eine prägnante Darstellung der wesentlichen Konzepte und Unterschiede (und führt zu einer eher ungewöhnlichen Sicht auf JAVA als Programmiersprache). In Lektion 5 werden dann auf analoge Weise der Reihe nach C#, C++ und EIFFEL präsentiert. Dies soll nicht zuletzt Ihrem Verständnis von der Vielfalt der Sprachenlandschaft dienen und Ihnen die Einsicht vermitteln, dass die guten Seiten der objektorientierten Programmierung nicht in einer Sprache allein zu finden sind.

Lektion 6 thematisiert die wichtigsten Probleme der objektorientierten Programmierung. Es sind dies im Wesentlichen die des Subtyping und der damit verbundenen Substituierbarkeit von Objekten sowie der Kapselung von Objekten zu größeren Einheiten, die ihr Inneres verbergen. Beide scheinen auf den ersten Blick gelöste Probleme zu sein; tatsächlich sind sie es aber keineswegs. Auch auf Probleme der Skalierbarkeit der objektorientierten Programmierung wird hier eingegangen.

In Lektion 7 lassen wir schließlich die Konstrukte objektorientierter Programmiersprachen hinter uns und gehen ausführlich auf das ein, was man gemeinhin objektorientierten Stil nennt. Der Programmierstil hat sich nämlich seit den Anfängen der prozeduralen Programmierung mit Sprachen wie PASCAL, wie er von Größen wie EDSGER DIJKSTRA, TONY HOARE oder NIKLAUS WIRTH gelehrt wurde, mindestens so sehr geändert wie die Sprachen selbst. Objektorientierte Sprachen zeichnen sich dadurch aus, wie gut sie objektorientierten Stil umzusetzen erlauben, und Sie als Programmierer, wie Sie ihn umzusetzen in der Lage sind.

Zur Form

Dieser Lehrtext verwendet das aktuelle Layout der Fernuniversität, mit einigen Erweiterungen (und einer anderen Schriftart).

Piktogramme im Rand

Hinter den Piktogrammen im Rand verbergen sich Links auf Online-Quellen. Sie ersetzen konventionelle Quellenangaben und Hinweise auf weiterführende Literatur gleichermaßen. Auf verlinkte Artikel digitaler Bibliotheken sollten Sie aus dem Netz der Fernuni zugreifen können; evtl. müssen Sie dafür das VPN der Fernuni nutzen. Ein Verzeichnis aller verlinkten Quellen finden Sie am Ende des Lehrtextes.

Links im Text

Neben der üblichen Verlinkung von Querverweisen sind im Text auch Begriffsanführungen mit ihren -definitionen verknüpft, soweit letztere explizit vorhanden sind. Sie erkennen Anführungen am kursiven und Definitionen am fetten Schriftsatz. Alle übrigen Bergriffsanführungen und die Definitionen selbst verweisen in den Index; dessen Seitenzahlen verweisen wiederum auf die Vorkommen der Begriffe im Text. Prosit!

Barcode im Seitenfuß

Der Barcode in den Fußzeilen des Hauptteils dieses Lehrtextes codiert die Lehrveranstaltungsnummer, eine zweistellige Variantennummer sowie eine dreistellige, absolute Seitenzahl. Er dient meiner **Papier/Digital-Brücke**, einer experimentellen Android-App, die sie sich unter nebenstehendem Barcode (www.feu.de/pdb/) herunterladen können. Mit Hilfe der Papier/Digital-Brücke können Sie alle Links des PDFs auch in der Papierversion direkt nutzen, sich den Lehrtext in erweiterter Form seitenweise vorlesen lassen oder im Studienbrief notierte Anmerkung mit anderen teilen. Die App befindet sich noch im Experimentierstadium; ob sie weiterentwickelt wird, hängt auch von Ihrer Rückmeldung ab.

Genderisierung

Gemäß Vorgabe der Fernuni ist dieser Lehrtext genderisiert. Da die konsequente Verwendung geschlechtsneutraler Formulierungen die persönliche Ansprache verwässert, habe ich mich entschlossen, zwei Varianten zu erstellen: eine, die das generische Femininum bemüht und eine, die das Maskulinum verwendet (die vorliegende). Da Lehrveranstaltungsbelegung und Materialversand an der Fernuni derzeit keine Varianten unterstützen, bekommen Sie die erste Variante gedruckt und geliefert — sofern Sie die zweite bevorzugen, können Sie sich das entsprechende PDF aus der VU herunterladen. Falls Sie wissen wollen, wie groß der Unterschied zwischen den beiden Varianten ist: Es sind aktuell 284 Phrasen geschlechtsspezifisch (wobei ich sicher noch einige übersehen habe).

Hagen, im Januar 2024

Vorkurs

In diesem Vorkurs werden die praktischen Voraussetzungen für die folgenden Lektionen geschaffen. Sie werden dazu an eine objektorientierte Programmiersprache, SMALLTALK, und die damit verbundene Programmierumgebung herangeführt. Es ist für den weiteren Verlauf wichtig, dass Sie mit dem Umgang mit mindestens einem SMALLTALK-System vertraut sind, auch wenn Sie noch nicht verstehen, wie es funktioniert und warum es so anders zu sein scheint als alles, was Sie vielleicht schon kennen.

Parallel zum Lesen dieses Vorkurses, der ausgesprochen leichte Kost ist, sollten Sie ein oder mehrere SMALLTALK-Systeme installieren. Wie das geht, finden Sie in den Übungsaufgaben zu diesem Vorkurs in der Moodle-Umgebung der Lehrveranstaltung. Die Aufgaben sollten Sie unbedingt durchführen und ggf. so oft wiederholen, bis Sie die Anleitung dazu nicht mehr brauchen. Es wäre ärgerlich, wenn Sie die Einsendeaufgaben der folgenden Lektionen nur deswegen nicht lösen können, weil Sie mit Ihrem SMALLTALK-System nicht zurechtkommen.

Objektorientierte Programmierung

Das Adjektiv „objektorientiert“ wurde nach eigenem Bekunden von ALAN KAY geprägt:

> *... a new design paradigm — which I called object-oriented — for attacking large problems of the professional programmer ...*

KAY erhielt 2003 den renommierten Turing-Award (auch als Nobelpreis der Informatik bezeichnet), und zwar u. a. für seine Rolle bei der Entwicklung von SMALLTALK:

> *For pioneering many of the ideas at the root of contemporary object-oriented programming languages, leading the team that developed SMALLTALK, and for fundamental contributions to personal computing.*

Hütern der englischen Sprache, die nichts mit Programmieren zu tun haben, stößt der Begriff „object-oriented“ übrigens sauer auf, denn sprachlich korrekt müsste es „object-orientated“ heißen. Zum Glück wurde dieser Frevel, der oft Skandinaviern zugeschrieben wurden, nach eigenem Bekunden von einem Muttersprachler begangen.

Weltbild der objektorientierten Programmierung

Das der objektorientierten Programmierung zugrundeliegende Weltbild ist das von *Objekten*, die jeweils eine *Identität* haben, die einander *Nachrichten* schicken und die in Reaktion auf die Nachrichten durch Anweisungen, die in *Methoden* festgehalten sind, ihren *Zustand* verändern. Welche Nachrichten ein Objekt versteht, zählt zu seinen *Eigenschaften*. Objekte können zudem entstehen und wieder vergehen — das objektorientierte Weltbild ist also in vielerlei Hinsicht dynamisch.

Damit sich Objekte gezielt Nachrichten schicken können, müssen sie sich kennen. Welche anderen Objekte ein Objekt kennen kann, zählt zu seinen Eigenschaften; welche es kennt, macht den Zustand eines Objektes aus und unterliegt mit diesem der Veränderung. Um neue Bekanntschaften zu schließen, können einem Objekt ein oder mehrere andere Objekte als Parameter einer Nachricht geschickt werden. Der Empfang einer Nachricht durch ein Objekt führt in der Regel zum Versand weiterer Nachrichten durch das empfangende Objekt sowohl an andere Objekte als auch an sich selbst. Auch das Entstehen und Vergehen von Objekten erfolgt in der Regel als Reaktion auf den Empfang einer Nachricht.

Da Objekte so allgemeine Dinge wie Personen oder Dokumente, aber auch so spezielle Dinge wie Zahlen oder Wahrheitswerte sein können und Nachrichten so allgemeine wie „schreib dich ein“ oder „drucke dich aus“, aber auch so spezielle wie „+“ oder „−“, hat man damit schon fast alles, was man zum Programmieren braucht. Die einzigen zusätzlich benötigten Konzepte sind das der *Variable* und der *Wertzuweisung*, die Sie vermutlich bereits aus anderen Programmiersprachen kennen.

Vererbung bzw. Subtyping

Der Begriff der Objektorientierung verlangt aber noch mindestens eine weitere Eigenschaft, die der *Vererbung*. Sie besagt, dass Objekte Eigenschaften von anderen erben können, dass also insbesondere nicht jedes Objekt für sich seine Eigenschaften definieren muss. Allerdings hat sich der Begriff der Vererbung als ein zwar leicht zu fassender, aber ungemein schwierig umzusetzender erwiesen, zumindest, wenn man sich durch Vererbung nicht in dem Umfang Schwierigkeiten einhandeln will, in dem man sich Arbeit erspart. Und so sehen viele heute eher das sog. *Subtyping* anstelle der Vererbung als für die objektorientierte Programmierung wesensbildende Eigenschaft.

Eigenschaften der objektorientierten Programmierung

Als die objektorientierte Programmierung noch angepriesen werden musste, wurden immer wieder dieselben Vorteile genannt:

- einfache *Wiederverwendung*,
- natürliche *Modularisierung* sowie
- *Durchgängigkeit der Konzepte* von der Analyse über das Design (beide heute gern unter dem Begriff „Modellierung“ zusammengefasst) bis hin zur Implementierung.

Auch wenn kein kausaler Zusammenhang bestehen sollte: Seit sich die objektorientierte Programmierung durgesetzt hat, redet kaum einer noch von der *Softwarekrise*.

Objektorientierte Programmiersprachen

Inzwischen gibt es eine große Anzahl von Programmiersprachen, die objektorientiert sind. Viele ältere Programmiersprachen wie z. B. PASCAL oder ADA sind in Fortschreibungen wie OBERON oder ADA-95 objektorientiert geworden, aber auch jüngere Sprachen wie z. B. PHP haben sich nach und nach „objekt-orientiert“. Nicht mehr wegzudenken aus der modernen Softwareentwicklung ist die Objektorientierung jedoch erst seit Einführung der von C und SMALLTALK abgeleiteten objektorientierten Sprachen C++, JAVA und C#: Diese haben die kommerzielle Softwareentwicklung quasi im Sturm erobert.

bekannte objektorientierte Programmiersprachen

Zu den bekannteren objektorientierten Programmiersprachen (oder zumindest Programmiersprachen mit objektorientierten Anteilen) zählen heute (in alphabetischer Reihenfolge): ADA 95, BETA, C++, C#, COMMON LISP, D, OBJECT PASCAL (BORLAND DELPHI), DYLAN, EIFFEL, SATHER, FORTRAN 2003, JAVA, JAVASCRIPT, MODULA-3, OBERON, OBJECTIVE-C, PERL 5, PHP 5, SELF, PYTHON, RUBY, SCALA, SIMULA, SMALLTALK, SELF und VISUAL BASIC. Diese haben teilweise ihre Wurzeln in der prozeduralen Programmierung (mit Programmiersprachen wie ALGOL, PASCAL oder C), teilweise in der funktionalen (mit Programmiersprachen wie LISP); aber auch ganz eigenständige wie beispielsweise BETA sind darunter. Auch wenn nicht alle Sprachen gleich praxisrelevant sind, kann es sich doch lohnen, sie zu kennen, denn jede hat ihre ganz eigenen Stärken und Schwächen und auch wenn man in einer anderen Sprache programmieren muss, kann man doch vielleicht die eine oder andere Idee aus einer anderen Sprache emulieren.

WIKIPEDIA

SMALLTALK

SMALLTALK wurde seit dem Beginn der 1970er Jahre bei der Firma Xerox in Palo Alto, USA, (genauer: im Palo Alto Research Center PARC) entwickelt, wurde dort 1983 unter der Bezeichnung SMALLTALK-80 der Öffentlichkeit präsentiert und gilt bis heute als die objektorientierteste objektorientierte Programmiersprache. Auch wenn ALAN KAY in seinem Rückblick ein anderes Bild zeichnet, wird als einzige direkte Vorgängerin SMALLTALKs eigentlich immer die Programmiersprache SIMULA genannt, die bereits ca. zehn Jahre früher, also in den 1960er Jahren, von den beiden Norwegern OLE-JOHAN DAHL und KRISTEN NYGAARD (die dafür schon vor KAY im Jahr 2001 den Turing-Award erhielten und nach denen inzwischen selbst ein Preis benannt ist) an der Universität Oslo konzipiert wurde und die unter dem Namen SIMULA-67 in Europa (aber wohl auch nur dort) einige Verbreitung gefunden hatte. Ein anderer wichtiger Einfluss auf SMALLTALK war aber sicher LISP, denn gewisse *funktionale* Eigenschaften (also die Tatsache, dass in der Sprache Funktionsausdrücke ausgewertet werden) kann und will SMALLTALK nicht verbergen.

WIKIPEDIA

WIKIPEDIA

Grundidee SMALLTALKs

Die Grundidee SMALLTALKs lässt sich in einem einzigen, einfachen Satz zusammenfassen:

> Alles ist ein Objekt.

Wie Sie noch sehen werden, ist mit „alles“ wirklich fast alles gemeint: Klassen, Instanzen, Variablen, Nachrichten und Methoden sind neben anderen Dingen wie Browsern, Editoren, dem Compiler, dem Dateisystem und jeder Änderung, die Sie machen, alles Objekte. Auch kennt SMALLTALK keine primitiven Werte: Selbst so grundlegende Dinge wie Zahlen und die Booleschen Wahrheitswerte `true` und `false` sind Objekte, die grundsätzlich nicht anders behandelt werden als etwa ein Dokument oder eine Grafik. Dadurch bedingt kommt SMALLTALK mit einer minimalen Menge von Konzepten aus, was der Sprache eine gewisse Eleganz verleiht. Das Sparsamkeitsprinzip wird dadurch auf die Spitze getrieben, dass viele der von anderen Programmiersprachen bekannten Konstrukte in SMALLTALK nicht primitiv (Bestandteil der Sprachdefinition) sind, sondern darin ausgedrückt (emuliert) werden. Die Kargheit der Sprache wird denn auch durch eine umfangreiche Klassenbibliothek, die dadurch gewissermaßen Bestandteil der Sprachdefinition wird, kompensiert, so dass einem trotz der Andersartigkeit vieles bei der Programmierung in SMALLTALK vertraut vorkommt. Es ist sogar so, dass in SMALLTALK viele der bekannten Programmierkonstrukte Seite an Seite mit solchen stehen, die man schon immer gern gehabt hätte, die andere Programmiersprachen aber (noch) nicht anbieten, und dass der bereits vorhandene Vorrat mit relativ wenig Aufwand von einem selbst um weitere Konstrukte ergänzt werden kann.

SMALLTALK ist mehr

SMALLTALK ist aber mehr als nur eine Programmiersprache: Es ist zugleich ein Betriebssystem und eine vollständig integrierte Entwicklungsumgebung. Jede mit SMALLTALK entwickelte Anwendung ist dabei lediglich eine Änderung des SMALLTALK-Systems, mit dem sie entwickelt wurde. Der Programmierer macht also nichts weiter, als seine SMALLTALK-Installation solange zu verändern, bis sie die Funktionalität der gewünschten Anwendung hat. Dazu kann er nicht nur Funktionen hinzufügen, sondern auch existierende verändern oder entfernen. Die Programmierung in SMALLTALK ist dabei vollkommen unabhängig von so primitiven Konzepten wie Datei oder Verzeichnis, die in anderen Sprachen Teil der Definition sind (oder zumindest diese maßgeblich beeinflussen).[1] Auch die üblichen, langwierigen Editier-Speicher-Kompilier-Test-Zyklen gibt es nicht — Sie arbeiten immer an einem laufenden System (die sog. *Live-Programmierung*). SMALLTALK ist ein experimentelles System im besten Sinne und Programmieren in SMALLTALK ist mit seinen extrem kurzen Editier-Kompilier-Ausprobier-Zyklen eine höchst dynamische Angelegenheit.

kommerzielle Verwendung

Wer kommerzielle Softwareentwicklung aus eigener Anschauung kennt, ahnt vielleicht, dass es mit dem praktischen Einsatz von SMALLTALK Probleme geben könnte: Die Philosophie, keine Auslieferung von binären, d. h. im Wesentlichen nicht durch den Benutzer veränderbaren, Programmen vorzusehen, mutet aus kommerzieller Sicht etwas eigentümlich an. Zudem war die gegenüber nativ kompilierten Programmen geringere Performance (Ausführungsgeschwindigkeit) sicher ein Problem. Dennoch setzte Mitte der neunziger Jahre ein SMALLTALK-Boom ein: Vor allem Banken und andere Unternehmen mit

[1] Diese Verbannung des Begriffs der Datei aus der Definition der Programmiersprache findet man auch in JAVA, wo er durch den Begriff der *Compilation unit* ersetzt wird. IBMs Entwicklungsumgebung VISUALAGE FOR JAVA (die der Vorläufer von ECLIPSE war) speicherte Programme auch zunächst in einer Datenbank; mit dem Übergang zu ECLIPSE ging man aber wieder auf Dateien über, schon weil viele andere Programmierwerkzeuge (darunter Versionskontrollsysteme) auf Dateibasis arbeiten.

großen betrieblichen Informationssystemen begannen, erste hausinterne Applikationen mit SMALLTALK zu entwickeln.[2] Geschätzt wurde vor allem die (im Vergleich zu den branchenüblichen Sprachen wie COBOL, aber auch zu sog. höheren Programmiersprachen wie MODULA, C oder C++) sehr viel höhere Produktivität. Ironischerweise wurde dieses zarte Pflänzchen wenig später ausgerechnet von JAVA, einer Programmiersprache, die sich manches von SMALLTALK abgeguckt hat, in Sachen Laufzeit um keinen Deut besser ist, in Sachen Produktivität jedoch trotz aller Neuerungen heute immer noch nicht an SMALLTALK heranreicht, plattgewalzt. Während der kurzen Blütezeit von SMALLTALK entwickelte IBM übrigens seine Entwicklungsumgebung VISUALAGE, die wenig später (und noch in SMALLTALK programmiert) für JAVA adaptiert wurde und aus der das heutige ECLIPSE-Projekt hervorgegangen ist. Doch während ECLIPSE als Entwicklungsumgebung für JAVA das SMALLTALK-System, aus dem es hervorgegangen ist, seit langem weit übertrifft, hat die Programmiersprache JAVA bis heute gebraucht, um an die Produktivität von SMALLTALK heranzureichen (manche würden sagen: Sie hat es bis heute nicht geschafft).

Quell zahlreicher Innovationen

Vielleicht das größte Problem SMALLTALKs war aber, dass es zu früh kam. Seine Architektur, die am unteren Ende auf einer virtuellen Maschine (VM) mit automatischer Speicherbereinigung (*Garbage collection*) beruhte und am oberen auf einer Bedienoberfläche mit pixelgraphischen, überlappenden Fenstern und Zeigegeräten wie Maus oder Tablet, stellte recht deftige Anforderungen an die Hardware, die damals noch klobig und teuer war. Außerdem wurden fast alle innovativen Konzepte SMALLTALKs quasi auf der grünen Wiese entwickelt und mussten daher erst einmal in der Praxis erprobt und über einen langen Zeitraum verbessert werden, bis sie wirklich zu gebrauchen waren. Am Ende dieser Entwicklung haben dann nur wenige das Potential erkannt, das in SMALLTALK steckte, darunter aber immerhin Steve Jobs, dessen Firma Macintosh-System ganz wesentliche Elemente SMALLTALKs (Maus, überlappende Fenster) übernommen und zur Verbreitung geführt hat.

Warum gerade SMALLTALK?

Man kann lange darüber diskutieren, welche der zahlreichen objektorientierten Programmiersprachen am besten für das Erlernen der objektorientierten Programmierung im Rahmen der universitären Lehre geeignet ist. In dem Ihnen vorliegenden Lehrtext habe ich mich vor allem aus didaktischen Gründen für SMALLTALK entschieden. Dabei soll der Lehrtext kein SMALLTALK-Kurs sein; Sie sollen vielmehr die objektorientierten Konzepte und damit auch das objektorientierte Denken verinnerlichen, und SMALLTALK scheint mir dafür am besten geeignet. Gleichwohl setzt das Verinnerlichen einiges an Übung „am Objekt“ voraus, und deswegen haben die ersten Lektionen durchaus den Charakter einer SMALLTALK-Einführung. Spätestens ab Lektion 4 wird

[2] Das oft zitierte C3-Projekt, mit dem die Disziplin des Extreme Programming begründet wurde, war ein SMALLTALK-Projekt. Leider ist mir keine Arbeit bekannt, die untersucht hätte, auf welche Faktoren genau sich der Erfolg dieses Projekts zurückführen lässt — man kann aber wohl davon ausgehen, dass mit Kent Beck, Ward Cunningham, Ron Jeffries und Martin Fowler einige der bekanntesten (und wohl auch besten) Programmierer der Zeit mit an Bord waren, die nun einmal alle SMALLTALK sprachen.

jedoch der Vorhang gehoben und der Blick auf die Weiten der heutigen objektorientierten Programmiersprachenlandschaft freigegeben.

leicht zu lernen

SMALLTALK ist von Anfang an als eine leicht zu erlernende Sprache konzipiert worden. Insbesondere erweist sich als didaktischer Vorteil, dass man zum Erlernen der Sprache keine Programme schreiben, also genau genommen gar nicht programmieren muss — stattdessen führt man mit dem System einfach „Smalltalk“. Dazu kommt, dass die Grammatik der Sprache ausgesprochen klein ausfällt: Es gibt keine Schlüsselwörter, nur wenige (fünf!) reservierte Namen (JAVA hingegen hat 50 Schlüsselwörter und PASCAL immerhin auch schon 35) und lediglich die aus dem normalen Schriftgebrauch bekannten Interpunktionszeichen haben eine durch die Sprachdefinition festgelegte Bedeutung. So reicht in SMALLTALK für „Hello World!“ bereits der einfache Ausdruck

```
1 Transcript show: 'Hello World!'
```

um den Text auf dem Ausgabeterminal des SMALLTALK-Systems, Transcript genannt, auszugeben. Dabei handelt es sich jedoch nicht um ein Programm im landläufigen Sinne, sondern lediglich um einen Ausdruck, der in der Entwicklungsumgebung ausgewertet („ausgeführt“) werden kann und als Ergebnis die Ausgabe der Zeichenkette „Hello World!“ bewirkt.[3] Tatsächlich gibt es in SMALLTALK wie bereits erwähnt so etwas wie ein Programm, das Sie schreiben, eigentlich gar nicht; stattdessen handelt es sich beim SMALLTALK-System selbst um ein (laufendes) Programm, dass Sie als Programmierer — während es läuft — so verändern, dass es neben allen anderen Dingen, die es kann, auch das tut, was Sie wollen (das bereits erwähnte *Live programming*). Je nach zu lösendem Problem kann Ihr eigener Beitrag dabei aus einem einzigen Ausdruck bestehen oder die Definition einer Vielzahl neuer Klassen umfassen. Die Werkzeuge, die Sie dabei benutzen, sind selbst auf diese Weise entstanden und können Teil Ihrer Anwendung sein.

Lernen durch Ausprobieren

Diese Besonderheit des SMALLTALK-Systems ist aber nicht nur ein faszinierender Denkansatz, sondern ermöglicht auch eine sehr praktische Herangehensweise bei der Vermittlung der Sprache. Es ist nämlich möglich, die Sprache durch Ausprobieren zu erlernen, ohne sich vorher Gedanken über Dinge wie Editoren, Dateien oder gar Verzeichnisse, den Aufruf des Compilers oder den Start des Programms machen zu müssen. All das ist aber in anderen Sprachen wie z. B. JAVA trotz aufwendigster Entwicklungsumgebungen immer noch der Fall. Mit SMALLTALK können Sie hingegen sofort loslegen.

keine Ablenkung durch Typsystem

Ein weiterer wichtiger didaktischer Grund für die Wahl SMALLTALKs ist, dass es kein Typsystem hat. Bei den meisten anderen Sprachen müsste man mit Syntax und Semantik auch das Typsystem lernen und es fällt schwer, das Typsystem vom Rest

[3] Man vergleiche dies mit dem Erklärungsnotstand, den eine Sprache wie JAVA bereits bei so einem simplen Beispiel mit sich bringt. Wer ECLIPSE und ähnlich mächtige IDEs zur JAVA-Programmierung benutzt, wird einwenden, dass dies „in JAVA auch geht“; tatsächlich geht es aber nicht „in JAVA“, sondern lediglich „in ECLIPSE“, also der IDE. Wer dieses Feature nicht kennt und es ausprobieren möchte: `File > New > Other > Java > Java Run/Debug > Scrapbook Page`. Es ist dies eines der vielen Erbstücke aus der Zeit, als die Entwickler von ECLIPSE noch an VISUALAGE saßen.

der Sprache zu trennen. Objektorientierte Typsysteme unterscheiden sich aber zum Teil erheblich, ja sind sogar Gegenstand beinahe fanatisch geführter Auseinandersetzungen, obwohl sie mit den Grundgedanken der objektorientierten Programmierung zunächst nur wenig zu tun haben. Die Verwendung von SMALLTALK hingegen erlaubt, objektorientierte Programmierung unabhängig vom Typbegriff zu lehren und Typsysteme als das darzustellen, was sie sind: aufgepfropfte, semantische Regeln, die dazu dienen, logische Fehler in einem Programm zu finden, die aber zur eigentlichen Ausführung des Programms gar nicht notwendig sind.

Latein der Objektorientierung

Wenn also SMALLTALK Ihre erste objektorientierte Programmiersprache ist und Sie stattdessen lieber eine andere erlernt hätten, dann sollten Sie bedenken, dass es eine Vielzahl (die vermutlich auch zukünftig eher wachsen denn schrumpfen wird) verschiedener Programmiersprachen gibt, und dass der größte gemeinsame Teiler dieser Sprachen selbst keine existierende Programmiersprache ist. SMALLTALK ist jedoch, aufgrund seiner extremen Reduziertheit, recht dicht daran und beinahe alle objektorientierten Programmiersprachen wurden von SMALLTALK beeinflusst. Sie lernen also nicht für die *Beherrschung einer* objektorientierten Programmiersprache, sondern für ein *Verständnis aller*.

Programmierung mit SMALLTALK

Transcript

Wenn Sie Ihr SMALLTALK-System nach der Installation das erste Mal starten, sehen Sie ein oder mehrere Fenster bzw. können Sie über den Menüpunkt „Browse“ öffnen. Eines darunter ist das sogenannte Transcript, die „Konsole“ des SMALLTALK-Systems. Auf ihr werden bestimmte Meldungen vom System ausgegeben; sie entspricht im Wesentlichen dem allgemeinen Ausgabestrom anderer Programmiersprachen, die dort z. B. über Print-Statements angesprochen werden. In JAVA entspräche das Transcript etwa `System.out`.

Ausgehend vom Transcript können Sie weitere Fenster öffnen, und zwar entweder, indem Sie entsprechende Ausdrücke eingeben und auswerten, oder, indem Sie entsprechende Menübefehle aktivieren. (Wie das genau geht, erfahren Sie bei der Bearbeitung der Übungsaufgaben.) Zwei wichtige Arten von Fenstern sind sogenannte Workspaces und Klassenbrowser.

Workspaces und Klassenbrowser

In einem Workspace (im SMALLTALK-System PHARO „Playground“ genannt) können Sie Ausdrücke eingeben und auswerten lassen. Workspaces dienen in der Regel zum Herumexperimentieren. In einem Klassenbrowser können Sie sich die Klassen, aus denen das SMALLTALK-System, an dem Sie gerade arbeiten, besteht, anschauen und diese manipulieren. Einen solchen werden Sie vor allem gebrauchen, wenn Sie programmieren.

Image und Change log

Jeder Ausdruck, den Sie auswerten, und jede Klasse, die Sie ändern, bewirkt eine Veränderung des SMALLTALK-Systems. Sie wird, ohne dass Sie das merken, in einem Log file, gemeinhin „**Change log**“ genannt, mitgeschrieben. Wenn Sie SMALLTALK beenden, werden Sie gefragt, ob Sie das aktuelle **Image** speichern wollen. Das Image ist der Inhalt des (virtuellen) Speichers Ihres SMALLTALK-Systems, also all seine Objekte inklusive aller Dinge, die Sie sehen, also auch der Fenster und deren Inhalt. Wenn Sie das Image speichern, wird SMALLTALK sich beim nächsten Start genauso präsentieren, wie Sie es verlassen

haben, inklusive aller Änderungen, inklusive aller Fenster und deren Inhalt. Es gibt also keinen Neustart — dazu müssten Sie das System schon neu installieren. SMALLTALK ist also wie ein reelles, physisches System, an dem Sie bauen (und das sich ja auch nicht jedes Mal in seine Ausgangsmaterialien zerlegt, wenn Sie ins Bett gehen).

Wenn Sie das Image nicht speichern, z. B. weil Sie eine fatale Änderung (solche gibt es und sind leicht möglich!) rückgängig machen wollen oder weil Ihnen der Strom ausgefallen ist, ist das kein Problem, denn Sie haben ja noch das Change log. Das Change log enthält wie gesagt alle Änderungen, die Sie gemacht haben, und zwar nicht in binärer Form, sondern als SMALLTALK-Ausdrücke, die Sie auswerten können. Wenn Sie das tun, wiederholen Sie damit Ihre gemachten, aber nicht im letzten Image gespeicherten Änderungen. Sie können den zuvor nicht gespeicherten Zustand Ihres Systems also ganz oder auch nur teilweise — ganz so, wie Sie es wünschen — wiederherstellen. Mit Hilfe des Change logs verlieren Sie in SMALLTALK nie auch nur eine einzige Zeile Code. Und das schon seit 1980.

Dieses wenige ist eigentlich alles, was Sie wissen müssen, um loszulegen. Welche Ausdrücke Sie sinnvollerweise eingeben und wie Sie programmieren, erfahren Sie in den Übungen zu diesem Vorkurs (siehe Moodle-Umgebung der Lehrveranstaltung) sowie in den folgenden ersten beiden Lektionen.

Verfügbare SMALLTALK-Systeme

Prinzipiell können Sie den Lehrtext mit jedem SMALLTALK-System durcharbeiten und es ist Ihnen freigestellt, welches Sie einsetzen. Sie sollen nur wissen, dass sich die Systeme in Details unterscheiden, zwar nicht in ihrer Syntax, aber in ihren Klassenbibliotheken und damit auch in ihrer Benutzung. Entsprechend fallen auch die Lösungen zu den Übungsaufgaben teilweise leicht unterschiedlich aus. Getestet wurden die Selbsttest- und Einsendeaufgaben mit dem SMALLTALK-Systeme PHARO in der Version 11.0. Auch die Übungsaufgaben zu diesem Vorkurs beziehen sich auf PHARO, sollten sich aber leicht auf die anderen Systeme übertragen lassen. Einige der ernstzunehmenden SMALLTALK-Systeme, die Sie für diese Lehrveranstaltung nutzen können, möchte ich Ihnen kurz vorstellen:

- VISUALWORKS ist der direkte Nachfolger von SMALLTALK-80; es stellt die Weiterentwicklung des zunächst von den ursprünglichen Autoren selbst (unter dem Namen Parc Place Systems firmierend) vertriebenen originalen Systems dar. Gleichzeitig ist es unter den hier vorgestellten SMALLTALK-Systemen wohl das einzige, das auch für die kommerzielle Softwareentwicklung verwendet werden dürfte. Gleichwohl ist es für den akademischen Gebrauch frei erhältlich, und das auch noch für zahlreiche Plattformen. Wenn Sie sich jedoch von einem riesigen Funktionsumfang eher abgeschreckt als angezogen fühlen, ist VISUALWORKS eher nicht das Richtige für Sie.

- SQUEAK ist ein Ableger des originalen SMALLTALK-80-Systems, der von seinen beiden Vätern DAN INGALLS und ALAN KAY in eine etwas andere, strikt nicht-kommerzielle Richtung getrieben wurde, in die unter anderem die Unterrichtung von Kindern fällt. So enthält SQUEAK umfangreiche Multimediabibliotheken und damit Konstrukte, die man normalerweise nicht mit einer Programmiersprache assoziieren würde. Leider beobachte ich unter Windows fortgesetzt Probleme mit den Ruhezuständen (Standby und Hibernate).

- PHARO ist wiederum ein Ableger von SQUEAK, mit einem vergleichsweise aktiven Entwicklerteam. Es ist vermutlich das am Besten mit freien Lernmaterialen ausgestattete System und schon von daher eine Empfehlung wert.

Daneben gibt es noch einige andere, von denen ich zwei zumindest erwähnen möchte:

- Das heute immer noch im Netz herumgeisternde SMALLTALK EXPRESS ist eine in vielen Belangen vereinfachte und an die damalige Windows-PC-Hardware angepasste Version des ursprünglichen SMALLTALK-80. Da es mit einer vergleichsweise kleinen Klassenbibliothek und einer simplen Benutzungsschnittstelle auskommt, ist es für Anfänger gut geeignet, wenn man es denn mit seinem 16-bit-Executable auf einem aktuellen Betriebssystem noch zum Laufen bekommt.

- AMBER ist der Versuch, SMALLTALK in einem Browser laufen zu lassen. Entsprechend der Philosophie SMALLTALKs (das ja ein selbstmodifizierendes System ist) können so Webanwendungen entwickelt werden. Erfahrungen damit habe ich keine.

LITERATUR ZU SMALLTALK

Für alle mit einem Interesse an der Entwicklungsgeschichte von Programmiersprachen ist der Rückblick ALAN KAYs auf die Entstehung der Objektorientierung im Allgemeinen und SMALLTALKs im speziellen unbedingt lesenswert. Sie werden nach der Lektüre einiges, das Sie heute als gegeben hinnehmen, mit ganz anderen Augen sehen.

Ein sehr gutes (wenn nicht das beste) Werk zur Einführung in SMALLTALK und die objektorientierte Programmierung ist das Buch „SMALLTALK-80“ von ADELE GOLDBERG und DAVID ROBSON. Es steht didaktisch mit Klassikern wie denen von KERNIGHAN & RITCHIE für C und WIRTH für PASCAL in einer Reihe. Da es sich bei dem Ihnen vorliegenden Text aber um einen Lehrtext zur objektorientierten Programmierung und nicht zu SMALLTALK handelt und der Text deswegen auch Aspekte zu berücksichtigen hat, die in einer Einführung in SMALLTALK außen vor bleiben können, folgt dieser Lehrtext nicht dem Aufbau des Buchs. Zudem habe ich mir erlaubt, allzu euphorische Kommentare der Autoren, insbesondere zu Modularität und Skalierbarkeit objektorientierter Programmierung, zu unterschlagen, da sie heute, wenn nicht überholt, so doch zumindest deutlich relativierungsbedürftig sind. Dennoch seien Leser, die eine etwas langsamere Einführung in die Sprache bevorzugen, hiermit auf das Original verwiesen.

Eine wichtige Quelle für Bücher über SMALLTALK sind übrigens die Webseiten von STÉPHANE DUCASSE („Stef's Free Online Smalltalk Books"). Dort gibt es auch den Artikel von ALAN KAY zum Herunterladen; allerdings ist die Qualität der Reproduktion ziemlich schlecht.

Lektion 1: Grundkonzepte der objektorientierten Programmierung

Ein laufendes objektorientiertes Programm muss man sich als eine Menge interagierender Objekte vorstellen. Damit die Objekte interagieren können, müssen sie verbunden sein; sie bilden dazu ein Geflecht, das neben Objekten aus Beziehungen zwischen diesen besteht. Das Geflecht verändert sich dynamisch infolge der Interaktion zwischen Objekten; es unterliegt aber gewissen, durch das Programm vorgegebenen statischen Strukturen.

Statik vs. Dynamik

Die *Unterscheidung zwischen Statik und Dynamik* ist eine klassische der Programmierung. Während Programme traditionell statische Gebilde sind, die auf Papier oder in einem Nur-lese-Speicher festgehalten werden können, ist ihre Ausführung immer etwas Dynamisches. Wenn aber Programme selbst als Daten aufgefasst werden, dann verwischt die Grenze zwischen Statik und Dynamik: Programme werden veränderlich. Insbesondere, wenn Programme sich selbst verändern können, ist die Unterscheidung zwischen Statik und Dynamik nur noch bedingt nützlich.

Struktur vs. Verhalten

Alternativ zu Statik und Dynamik kann man auch zwischen *Struktur* und *Verhalten* unterscheiden, wobei mit Struktur das oben erwähnte Objektgeflecht und mit Verhalten die (Spezifikation der) Folge seiner Veränderungen gemeint ist. Diese Unterscheidung liegt der Gliederung des Rests dieser Lektion zugrunde: von Objekten (Kapitel 1) und Beziehungen zwischen diesen (Kapitel 2) geht es über den Zustand als Bindeglied (Kapitel 3) zu den Elementen der Verhaltensbeschreibung (Kapitel 4).

1 Objekte

In rein objektorientierten Programmiersprachen sind sämtliche Daten, die ein Programm verarbeiten kann, in Form von Objekten im Speicher abgelegt. Der Reiz dieses Merkmals der objektorientierten Programmierung ist, dass unser Weltbild, zumindest in weiten Teilen, auf einem ähnlichen Modell basiert: Die Welt besteht aus Objekten, die miteinander in Beziehung stehen. Dabei ist der Objektbegriff nicht auf das rein Materielle beschränkt: Nach allgemeinem Verständnis sind Personen ebenso Objekte wie Dokumente, Zahlen oder Zeichen.

verschiedene Arten von Objekten

Bei der Übertragung von realen (d. h., aus einer Anwendungsdomäne stammenden) Sachverhalten in ein objektorientiertes Programm ergibt sich das Problem, dass die Übertragung, aufgrund der Homogenität der Objektorientierung (alles ist ein Objekt), gewisse fundamentale Unterschiedlichkeiten der Kategorien unserer Begriffswelt ignoriert: Zahlen beispielsweise sind im Gegensatz zu Dingen Objekte ohne Identität, Zustand oder Lebensdauer (sie werden daher auch häufig als **Werte** bezeichnet); Mengen nicht weiter abgrenzbarer Elemente wie z. B. 1 Liter Wasser sind gar keine Objekte im eigentlichen Sinn (auch sie haben keine Identität) usw. Gleichwohl kommen sie alle in objektorientierten Programmen vor und werden dort — zumindest der reinen Lehre nach — durch Objekte repräsentiert. Der Ansatz, alles trotz evidenter ontologischer Unterschiede programmiersprachlich über einen Kamm zu scheren, führt hier und da zu gewissen Inkonsistenzen im ansonsten klaren, ja puristischen objektorientierten Weltbild, mit denen wir leben müssen, wenn wir objektorientiert programmieren wollen (vgl. dazu auch Kapitel 60 in Lektion 6). Es ist dies der Preis des auch „Ockhams Rasiermesser“ genannten Sparsamkeitsprinzips, das auch für die objektorientierte Programmierung Leitlinie ist.

WIKIPEDIA

1.1 Was ist ein Objekt?

Wie bereits erwähnt sind Objekte im Speicher abgelegte Daten. Dabei ist jedes Objekt an genau einer Stelle im Speicher abgelegt: Es wird damit durch seine Speicherstelle eindeutig identifiziert. Aufgrund dieser eindeutigen Identifizierbarkeit spricht man auch von der **Identität eines Objekts**; sie kann aus technischer Sicht mit der Speicherstelle, an der das Objekt abgelegt ist, gleichgesetzt werden. Da keine zwei Objekte an derselben Stelle abgelegt werden können, haben auch keine zwei Objekte dieselbe Identität.

Objekte vs. Werte

Objekte sind grundsätzlich von *Werten* zu unterscheiden. Werte werden auch im Speicher abgelegt, haben aber keine Identität. Es folgt, dass derselbe Wert an verschiedenen Stellen im Speicher vorkommen kann. Viele objektorientierte Programmiersprachen (wie etwa JAVA oder C#) unterscheiden ganz offen zwischen Werten und Objekten; SMALLTALK tut dies nur hinter den Kulissen und folgt ansonsten seinem Motto „alles ist ein Objekt“.

Speicherbedarf von Objekten

Die Menge des Speichers, den ein Objekt belegt, ist aus technischen Gründen konstant. Objekte können somit weder wachsen noch schrumpfen. Sollte dies trotzdem notwendig sein, bleibt nur, ein neues Objekt zu erzeugen, das an die Stelle (nicht die Speicherstelle!) des anderen tritt. Das neue Objekt hat jedoch eine andere Identität, so dass alle Stellen im Programm, die sich auf das alte Objekt beziehen, entsprechend angepasst werden müssen. Wie das geht, wird in Lektion 2, Abschnitt 14.2 erläutert.

1.2 Literale

Ein **Literal** (von lat. littera, der Buchstabe) ist eine in der Syntax der Programmiersprache ausgedrückte Repräsentation eines Objektes. Literale sind somit textuelle Spezifikationen von Objekten: Wenn der Compiler ein Literal übersetzt, erzeugt er daraus — bei der Übersetzung! —

das entsprechende Objekt im Speicher. Dies steht im Gegensatz zu objekterzeugenden *Anweisungen* eines Programms, denn diese werden erst zur Laufzeit des Programms ausgeführt. Da wir uns mit der programmgesteuerten Erzeugung von Objekten aber erst in der nächsten Lektion systematisch befassen, müssen wir hier zunächst mit Objekten mit literaler Repräsentation vorliebnehmen. Wohlgemerkt: Literale repräsentieren Objekte, es sind nicht selbst welche.

Zeichenliterale

Die einfachsten Literale repräsentieren Zeichen (genauer: Zeichenobjekte). Um die literale Repräsentation eines Zeichens von anderen Vorkommen von Zeichen in einem Programm zu unterscheiden, wird ihnen in SMALLTALK ein $-Zeichen vorangestellt. So bezeichnet das Literal

```
2 $a
```

das Zeichenobjekt „a“ [4]. Dieses Objekt ist **atomar**, d. h., es ist nicht aus anderen Objekten zusammengesetzt. Zeichen sind in anderen Programmiersprachen — auch objektorientierten — übrigens typischerweise Werte.

Zahlliterale

Eine andere Art von Literalen, die atomare Objekte repräsentieren, sind die für Zahlen:

```
3 1
```

ist z. B. ein Literal, das das Objekt „1“ bezeichnet;

```
4 -12.7
```

ist ebenfalls ein Zahlliteral. Zahlliterale bezeichnen ebenfalls atomare (nicht zusammengesetzte) Objekte; sie sind in anderen Programmiersprachen typischerweise ebenfalls Werte (nicht jedoch sehr große Zahlen mit beliebiger Genauigkeit — die werden auch in anderen objektorientierten Sprachen durch Objekte repräsentiert).

Literale vs. (Pseudo-) Variablen

Die in anderen Programmiersprachen vorzufindenden Literale (oder, je nach Sprache, *Schlüsselwörter*) `true`, `false` und `nil` (oder `null`), die genau wie Zeichen- und Zahlliterale atomare Objekte repräsentieren, sind in SMALLTALK nicht Literale, sondern *Pseudovariablen* (s. Abschnitt 1.7). Der Grund dafür scheint eher pragmatischer Natur zu sein: SMALLTALK kennt keine Schlüsselwörter und indem man `true`, `false` und `nil` als (Pseudo-) Variablen auffasst, müssen sie vom Compiler syntaktisch nicht von *Variablen* (s. Abschnitt 1.5) unterschieden werden. So oder stehen sie für jeweils ein entsprechendes Objekt (die in anderen Sprachen wiederum Werte sind).

[4] Ich verwende hier doppelte Anführungsstriche, um Objekte metasprachlich (also im Kurstext, nicht in einem Programm) zu benennen. Gemeint ist damit immer die Repräsentation des Objekts im Speicher. Um ein Literal, also die Repräsentation eines Objekts in einem Programm, zu benennen, setze ich es in diesem Kurstext in der Schriftart für (Programm-)Code.

String-Literale Wenn es atomare Objekte gibt, dann muss es auch **zusammengesetzte** geben. So können beispielsweise Zeichen zu **Zeichenketten**, den sogenannten **Strings**, zusammengesetzt werden, die ebenfalls Objekte sind. Ein String kann aber selbst wieder durch ein Literal bezeichnet werden; so steht in SMALLTALK

```
5 'Smalltalk'
```

für ein String-Objekt „Smalltalk“. Dieses Objekt ist selbst aus Objekten, nämlich den von den Zeichenliteralen `$S`, `$m`, `$a`, `$l`, `$l`, `$t`, `$a`, `$l` und `$k` repräsentierten Zeichenobjekten, zusammengesetzt. Was Zusammensetzung von Objekten heißt und wie sie funktioniert, darauf gehe ich in den Abschnitten 2.1 und 2.3 noch genauer ein.

Es repräsentieren also String-Literale zusammengesetzte Objekte. Daraus ergibt sich die Frage, ob zwei gleiche String-Literale dasselbe Objekt im Speicher repräsentieren. Dies ist nicht grundsätzlich so, wie wir noch sehen werden.

Symbolliterale Um durch syntaktisch gleiche Zeichenketten stets dasselbe Objekt zu bezeichnen, bietet SMALLTALK sog. **Symbole** als weitere Art von Objekten mit literaler Repräsentation. So ist

```
6 #Smalltalk
```

die literale Repräsentation eines Objekts. Es bezeichnet bei jedem Vorkommen im Programm dasselbe Symbolobjekt „Smalltalk“ (nicht zu verwechseln mit dem obigen String-Objekt). Symbole dürfen, anders als Strings, nicht alle Zeichen enthalten (so z. B. keine Leerzeichen).

Verwaltung von Symbolliteralen Da gleiche Symbolliterale immer dasselbe Objekt repräsentieren, ist die Erzeugung eines solchen Objekts durch den Compiler technisch aufwendiger als beispielsweise die anhand eines String-Literals. Der Compiler muss nämlich vor dem Erzeugen erst prüfen, ob das Literal schon einmal irgendwo vorkam. Ist das der Fall, erzeugt er kein neues Objekt, sondern verwendet stattdessen das bereits vorhandene. Das setzt natürlich eine entsprechende Verwaltung aller Symbolliterale und dazugehörigen Objekte durch den Compiler voraus.[5] Wie man sich leicht vorstellen kann, wäre diese Vorgehensweise für die universell und in großer Anzahl verwendeten Strings sehr zeitaufwendig. Gleichwohler versuchen manche SMALLTALK-Compiler, gleiche Literale, die zusammen kompiliert werden, auf dasselbe Objekt abzubilden. Das führt manchmal, durch das sog. *Aliasing* (s. Abschnitt 1.8), zu unerwarteten Ergebnissen bei der Verwendung dieser Literale.

Array-Literale Die letzte wichtige Kategorie von Literalen in SMALLTALK sind **Array-Literale**. Die von ihnen repräsentierten Objekte sind genau wie Strings zusammengesetzt, bestehen aber im Gegensatz dazu nicht nur aus Zeichen, sondern aus einer Folge beliebiger, wieder durch Literale repräsentierter Objekte. Ein Array-Literal wird in SMALLTALK vom #-Zeichen und

[5] Die dafür verwendete Symboltabelle ist übrigens selbst ein SMALLTALK-Objekt mit Namen `SymbolTable`.

einer öffnenden Klammer angeführt, der durch Leerzeichen getrennte Literale folgen; es wird durch eine schließende Klammer abgeschlossen.

```
7 #(1 2 3)
```

ist ein solches Array-Literal,

```
8 #('Smalltalk' 'ist' 1.0 'Klasse')
```

ein anderes. Array-Literale können ineinander geschachtelt sein; das #-Zeichen entfällt jedoch bei allen inneren Arrays. In

```
9 #(($S $m $a $l $l $t $a $l $k) 'ist' 1.0 'Klasse')
```

beispielsweise ist das String-Literal `'Smalltalk'` in Zeile 8 durch ein gleichbedeutendes Array-Literal, das aus Zeichenliteralen besteht, ersetzt.

Für Array-Literale gilt ansonsten das Gleiche wie für String-Literale: Dass zwei syntaktisch gleich sind heißt nicht, dass sie dasselbe Objekt erzeugen (oder, richtiger, dass aus ihnen nur ein Objekt erzeugt wird).

1.3 Änderbarkeit von Objekten

Während atomare Objekte grundsätzlich nicht änderbar sind (welchen Sinn hätte es beispielsweise, aus einer „1“ eine „2“ zu machen oder aus einem „a“ ein „b“?), so gilt das für zusammengesetzte zunächst nicht: Es ist leicht vorstellbar (und auch grundsätzlich sinnvoll), in einem Array-Objekt eine Komponente durch eine andere zu ersetzen. Die Frage ist allerdings, ob dies auch für Array-Objekte gilt, die aus Literalen erzeugt wurden: Soll es erlaubt sein, dass das zusammengesetzte Objekt, das aus dem Array-Literal `#(1 2 3)` hervorgegangen ist, durch ein Programm abgeändert wird, so dass es nicht mehr seiner (ursprünglichen) literalen Repräsentation im Programm entspricht? Dies ist Ansichtssache und wird zumindest für String-und Array-Literale von unterschiedlichen SMALLTALK-Dialekten unterschiedlich gehandhabt. Objekte, die aus Symbolliteralen hervorgegangen sind, sollten dagegen nie änderbar sein.

Grundsätzlich sind zusammengesetzte Objekte in SMALLTALK jedoch änderbar. Es ist dies Voraussetzung dafür, dass Objekte einen *Zustand* haben können (s. Kapitel 3), was wiederum die objektorientierte Programmierung zu einer Form der *imperativen Programmierung* macht. Durch die Zunahme *funktionaler* Einflüsse auf die objektorientierte Programmierung findet man jedoch auch zunehmend Sprachen, die unveränderliche Objekte anbieten (so z. B. SCALA).

1.4 Gleichheit und Identität von Objekten

Wie oben schon zur Unterscheidung von String- und Symbolliteralen angedeutet, wird durch das Vorkommen des gleichen Literals an mehreren Stellen eines Programms nicht notwendigerweise dasselbe, also identische, Objekt repräsentiert — es kann auch sein, dass die erzeugten

Objekte nur gleich sind. Das wirft natürlich sofort die Frage auf, was der Unterschied zwischen Gleichheit und Identität bei Objekten ist, und wie überhaupt Objekte unterschieden werden können.

Gleichheit von Objekten

Die **Gleichheit von Objekten** ist Definitionssache und orientiert sich in der Regel an ihrem Erscheinungsbild oder ihrer Bedeutung. Gleichheit wird in SMALLTALK durch den Gleichheitsoperator = getestet. So liefern

```
10 $a = $a
11 1 = 1
12 1.0 = 1.0
13 'Smalltalk' = 'Smalltalk'
14 #Smalltalk = #Smalltalk
15 #(1 2 3) = #(1 2 3)
```

alle `true`. Aber auch

```
16 1 = 1.0
```

liefert `true`: Obwohl die beiden Zahlliterale verschieden sind (und für verschiedene, also zwei, Objekte stehen), bezeichnen sie doch (aus mathematischer Sicht) die gleiche Zahl, so dass man sie in SMALLTALK als gleich definiert hat.

Identität von Objekten

Die **Identität zweier Objekte** (alternativ: die gleiche Identität zweier Objekte) wird in SMALLTALK durch == getestet. So liefern

```
17 #Smalltalk == #Smalltalk
18 1 == 1
```

erwartungsgemäß `true`,

```
19 'Smalltalk' == 'Smalltalk'
```

kann hingegen zu `false` auswerten — zwei syntaktisch gleiche String-Literale können also zwei Objekte mit verschiedener Identität repräsentieren. Dies ist zumindest dann sinnvoll, wenn die durch die String-Literale erzeugten Objekte unabhängig voneinander änderbar sein sollen und deswegen tatsächlich zwei Objekte sein müssen. Übrigens: Phrasen wie „zwei identische Objekte“ sind strenggenommen Unsinn, denn es handelt sich bei vorliegender Identität definitionsgemäß nicht um zwei, sondern nur um ein Objekt. Die Frage nach der Identität von Objekten ist nur dann sinnvoll, wenn die Objekte durch *Namen* (oder *Variablen*) repräsentiert werden. Mehr dazu in Abschnitt 1.5.

Selbsttestaufgabe 1.1

Prüfen Sie in einem oder mehreren Ihnen zur Verfügung stehenden SMALLTALK-Systemen für verschiedene gleiche Literale, ob die repräsentierten Objekte identisch sind. Beschränken Sie sich dabei nicht nur auf die Beispiele der Zeilen 10–16. Was fällt Ihnen auf?

Ist es immer sinnvoll, von Identität zu sprechen?

Während man sich unter der Identität einer Person oder eines Dokuments leicht etwas vorstellen kann, scheint der Begriff der Identität für manch andere Objekte merkwürdig. Was hat man sich beispielsweise unter der Identität der Zahl „1“ vorzustellen? Und wenn „1“ tatsächlich ein Objekt mit Identität ist, was macht dieses Objekt zur Eins? Oder ist die 1 vielleicht die Identität des Objekts „1“, sind also das Objekt und seine Identität dasselbe?

Gleichsetzung von Identität und Erscheinung

Im Falle atomarer (also nicht zusammengesetzter) Objekte könnte man versucht sein, die Identität zweier Objekte mit der Gleichheit ihrer Erscheinungen gleichzusetzen: Es erscheint wenig sinnvoll, zwei immer gleiche Objekte mit unterschiedlicher Identität zu haben. So kann man sich beispielsweise fragen, warum man mehrere „1“ mit unterschiedlicher Identität in einem System haben sollte. Tatsächlich würde es wohl kaum auffallen, wenn zwei solche gleichen, aber sich dennoch aufgrund ihrer Identität unterscheidenden Objekte zu einem verschmelzen würden. Ganz anders ist das bei veränderlichen Objekten: Aufgrund ihrer Veränderlichkeit können sie sich auch nur vorübergehend gleichen, müssen aber selbst während dieser (vorübergehenden) Gleichheit voneinander zu unterscheiden sein, da sie sich hinterher wieder auseinanderentwickeln können und man dann nicht mehr wüsste, welches welches war. Da dies aber für unveränderliche Objekte nicht der Fall sein kann, ist es durchaus berechtigt, zu fragen, warum sie sich nur aufgrund ihrer Identität unterscheiden sollten.

warum auch unveränderliche Objekte verschiedene Identitäten haben

Die Antwort ist vor allem technischer Natur. Wenn sich ein unveränderliches Objekt wie beispielsweise eine Zahl nicht aus einem Literal, sondern aus einer Operation (einer Rechenoperation) ergibt, dann müsste, für eine Zusammenlegung gleicher Objekte zu einem, immer erst überprüft werden, ob ein gleiches Objekt bereits angelegt wurde. Da dies Programme stark verlangsamen würde, nimmt man lieber in Kauf, mehrere gleiche, aber nicht identische Objekte zu haben. Aber warum sind dann gleiche Zahlen manchmal identisch, manchmal nicht? Die Antwort ist noch technischer: Sie hat etwas mit der Repräsentation von Objekten im Speicher zu tun und wird im nächsten Abschnitt gegeben. Und so werden in SMALLTALK bestimmte Objekte eben anders behandelt als der Rest: Ganze Zahlen (Integer) bis zu einer bestimmten Größe und Zeichen sind aus technischen Gründen immer auch identisch, wenn sie gleich sind — für den Rest (mit Ausnahme der Symbole!) gilt das nicht. Konzeptuelle Gründe für die Sonderbehandlung gibt es nicht.

Quintessenz

Zusammenfassend merken Sie sich am besten folgendes:

> „Das gleiche“ und „dasselbe“ sind auch in der objektorientierten Programmierung nicht das gleiche (und schon gar nicht dasselbe)!

Die Missachtung dieses Merksatzes ist eine der häufigsten Fehlerquellen der objektorientierten Programmierung. Besonders beim Vergleichen von Strings ist die Verwendung des Tests auf

Identität anstelle des Tests auf Gleichheit ein häufiger logischer Programmierfehler. Deswegen noch einmal ganz deutlich:

> Zwei Objekte können zwar gleich, aber nie dasselbe sein, oder es sind nicht zwei Objekte, sondern eins!

Verwenden Sie also grundsätzlich den Test auf Gleichheit (=), nicht auf Identität (==), es sei denn, Sie wollen prüfen, ob Sie es mit einem oder mit zwei Objekten zu tun haben.

1.5 Variablen

Weil Literale immer die gleichen Objekte repräsentieren, reichen sie zum Programmieren nicht aus. Was man vielmehr auch noch benötigt, sind **Namen**, die zu verschiedenen Zeitpunkten verschiedene Objekte bezeichnen können[6], die sog. **Variablen**.

Variable vs. Literal

Genau wie ein Literal steht eine Variable in einem Programm für ein Objekt. Anders als bei Literalen wird aus einer Variable jedoch kein Objekt erzeugt: Sie ist lediglich ein Name für ein bereits existierendes Objekt. Dazu kommt, dass eine Variable zu unterschiedlichen Zeitpunkten für unterschiedliche Objekte stehen kann (deswegen der Name „Variable"!). Es können zudem Variablen mit unterschiedlichen Namen für dasselbe Objekt stehen, das damit gewissermaßen verschiedene Namen hat (die sog. *Aliase*; s. Abschnitt 1.8). Wir werden daher im folgenden davon sprechen, dass Variablen Objekte *benennen* oder *bezeichnen*.

1.5.1 Inhalt

Wert- und Verweissemantik von Variablen

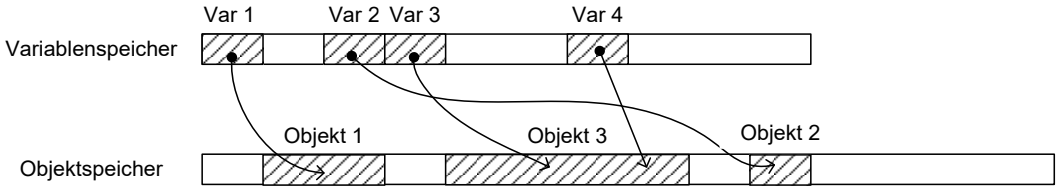
Das bezeichnete Objekt wird manchmal auch „Wert" oder „Inhalt" der Variable genannt (und die Variable selbst *Platzhalter* des Objekts). Besonders die Verwendung von „Inhalt" ist aber gefährlich, da sie nahelegt, ein Objekt könne zu einem Zeitpunkt nur von genau einer Variable bezeichnet werden, so wie ein Gegenstand zu einer Zeit immer nur Inhalt eines Behälters sein kann. Tatsächlich können aber mehrere Variablen gleichzeitig ein und dasselbe Objekt bezeichnen — die Variablen haben nämlich nur **Verweise** (auch **Referenzen** oder **Pointer** genannt) auf Objekte (genauer: auf die Speicherstellen, an denen die Objekte abgelegt sind; s. o.) zum Inhalt. Man spricht deswegen auch von einer **Verweis**- oder **Referenzsemantik** von Variablen, im Gegensatz zur **Wertsemantik**, bei der das bezeichnete Objekt tatsächlich auch Inhalt der Variable ist.

Variableninhalt auf Speicherebene

Aus technischer Sicht entspricht einer Variable eine Stelle im Speicher. Allerdings steht an dieser Stelle bei Variablen mit Verweissemantik nicht das Objekt, das sie bezeichnen, sondern lediglich ein Verweis auf die Speicherstelle, an der das

[6] Achtung: In der *funktionalen Programmierung*, in der der Begriff des Namens gebräuchlicher ist als in der objektorientierten, steht ein Name immer für dasselbe Objekt.

Objekt gespeichert ist. Es handelt sich also bei Variablen mit Verweissemantik aus technischer Sicht um **Pointervariablen**, wie man sie auch aus nicht objektorientierten Programmiersprachen wie PASCAL oder C kennt.



Bedeutung der Unterscheidung von Verweis- und Wertsemantik

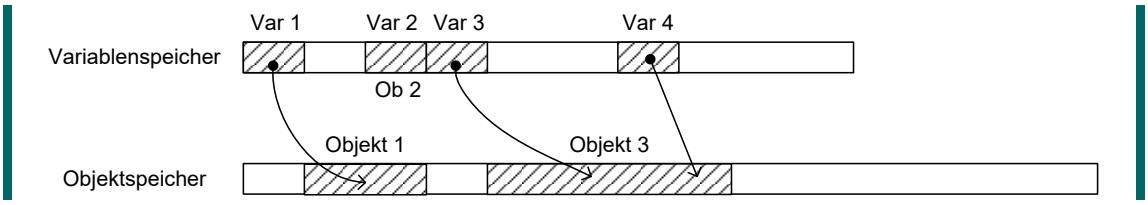
Verweis- und Wertsemantik von Variablen unterscheiden sich fundamental: Unter Wertsemantik können, solange jedes Objekt seine eigene Identität hat, zwei Variablen niemals dasselbe Objekt bezeichnen. Dies wird aber nur den wenigsten Programmierproblemen gerecht. Da zudem die Verweissemantik einen wesentlich speicher- und recheneffizienteren Umgang mit Objekten erlaubt und da unterschiedliche Objekte wie oben beschrieben unterschiedlich viel Speicherplatz belegen, so dass man im Vorfeld nicht immer weiß, wie viel davon man für eine Variable vorsehen muss, ist sie in der objektorientierten Programmierung vorherrschend. In machen Sprachen, die neben Objekten auch Werte kennen, haben Variablen, die Objekte aufnehmen, stets Verweissemantik und Variablen, die Werte aufnehmen, stets Wertsemantik (z. B. JAVA); andere objektorientierte Sprachen erlauben dem Programmierer, für jede Variable getrennt festzulegen, ob sie Wert- oder Verweissemantik haben soll (so z. B. C++ und EIFFEL).

wechselnde Semantik von Variablen in SMALLTALK

Nun ist besonders für unveränderliche Objekte, deren interne Repräsentation klein ist (die also wenig Speicherplatz belegt), die Forderung nach der Speicherung eines Objektes an genau einem Ort und Speicherung von Verweisen in Variablen (also die Speicherung in Variablen mit Verweissemantik) ineffizient. Welchen Sinn hätte es beispielsweise, allen Zeichen eine Identität zu geben, an der mit der jeweiligen Identität verbundenen Stelle im Speicher die internen Repräsentationen zu hinterlegen und dann in Variablen die Speicherstelle (Identität) zu speichern, wenn der Verweis mehr Speicher belegt als das Zeichenobjekt, auf das verwiesen wird? Das Gleiche gilt auch für Zahlen bis zu einer gewissen Größe.

Wertsemantik bei kleinen Objekten

In den meisten SMALLTALK -Implementationen hat man dieses Problem so gelöst, dass Variablen, die Zeichen, kleine Zahlen und die Booleschen Werte `true` und `false` bezeichnen, Wertsemantik haben. Die Objekte können damit aber tatsächlich an mehreren Stellen im Speicher gespeichert werden, was einen Widerspruch zur reinen Lehre darstellt. Zwar geht damit der Begriff der Identität für diese Objekte verloren, aber für den Programmierer ist die damit verbundene mehrfache Existenz identischer Objekte im Speicher insofern ohne größere Bedeutung, als hier Gleichheit problemlos an die Stelle der Identität treten kann. Der Preis für diese Flexibilität ist allerdings, dass man den Variablen nicht mehr fix Wert- oder Verweissemantik zuordnen kann — diese hängt vielmehr jeweils von der Art der Objekte ab, die sie gerade bezeichnen. In diesem Fall würde man Wert- bzw. Verweissemantik eher als eine Eigenschaft des Objekts denn der der Variable ansehen; das ist jedoch ziemlich SMALLTALK-spezifisch.



1.5.2 Sichtbarkeit

Eine Variable bezeichnet also ein Objekt. Wer auf eine Variable zugreifen kann, kann damit automatisch auch auf das Objekt zugreifen, das die Variable bezeichnet. Tatsächlich sind alle Objekte, für die es keine eindeutige literale Repräsentation (wie sie Zeichen, manche Zahlen und Symbole haben) gibt, nach ihrer Erzeugung nur noch über Variablen zugreifbar. Die einzige Ausnahme bilden hier die sog. *konstanten Methoden*, die jedoch erst in Abschnitt 4.3.6 behandelt werden.

Nun ist es nicht sinnvoll, dass in einem Programm alle Variablen (und damit auch alle Objekte) von überall her zugreifbar sind. Um den Zugriff auf Variablen einzuschränken, gibt es den Begriff der *Sichtbarkeit* und *Regeln für die Sichtbarkeit von Variablen*. Kurzgefasst ist die Sichtbarkeit einer Variable gleichbedeutend damit, dass man ihren Namen verwenden kann (und damit auch Zugriff auf das von diesem Namen bezeichnete Objekt hat). Dabei bezieht sich die Sichtbarkeit immer auf einen Abschnitt von Programmcode: Wenn eine Variable in einem Abschnitt sichtbar ist, dann entspricht jedes Vorkommen des Variablennamens in diesem Abschnitt einer ihrer Verwendungen.

lokale und globale Variablen

Die einzelnen Programmiersprachen unterscheiden sich zum Teil deutlich in der Definition ihrer Sichtbarkeitsregeln. Häufig wird jedoch zwischen sog. **globalen** und **lokalen Variablen** unterschieden. Dabei sind beide Begriffe relativ zu verstehen: lokale Variablen sind in ihrer Sichtbarkeit auf den Programmabschnitt beschränkt, um den es gerade geht (sowie ggf. auf dessen Unterabschnitte), globale Variablen sind auch außerhalb davon (insbesondere in Überabschnitten) sichtbar. Variablen, die überall sichtbar sind, sind also immer (relativ zu jedem Programmabschnitt) global. Wenn eine Variable außerhalb eines bestimmten Programmabschnitts, aber nicht überall sichtbar ist, sagt man auch, sie sei global zu dem Programmabschnitt; sie ist dann lokal zu einem übergeordneten (umschließenden) Programmabschnitt. Lokale Variablen überdecken übrigens immer globale Variablen gleichen Namens; man spricht dann auch von *Hiding*.

Benennungskonvention in SMALLTALK

In SMALLTALK müssen globale Variablen mit einem Großbuchstaben beginnen. `Smalltalk` und `Transcript` sind zwei prominente Beispiele für globale Variablen. Lokale Variablen beginnen hingegen mit einem Kleinbuchstaben und sind auf den Sichtbarkeitsbereich eines Objekts (oder auch nur Teilen davon) beschränkt. Für die genaue Angabe der *Sichtbarkeitsregeln* SMALLTALKs fehlt uns noch einiges; wir werden daher erst in den folgenden Abschnitten darauf eingehen; wir können aber schon hier schlussfolgern, dass in SMALLTALK der Unterschied zwischen lokal und global nicht relativ ist (es also nur zwei verschiedene Programmabschnitte gibt).

Selbsttestaufgabe 1.2

Versuchen Sie, durch Experimentieren herauszufinden, was in der Variable `Smalltalk` zu finden ist.

1.6 Zuweisung

Damit eine Variable ein Objekt bezeichnet, muss ihr dieses durch eine sog. **Zuweisung**, in anderen Kontexten auch **Wertzuweisung** genannt, zugeordnet werden. Ursprünglich wurde als *Zuweisungsoperator* das Symbol „←“ gewählt; wegen der mangelnden Verfügbarkeit auf Tastaturen wurde es jedoch in den meisten SMALLTALK-Implementierungen durch das aus ALGOL und PASCAL bekannte := (englisch als „becomes“ gelesen) ersetzt.[7] Die Variable `lieblingszahl` bezeichnet also in Folge der Zuweisung

```
20 lieblingszahl := 2
```

ein Objekt „2“ (in der Zuweisung repräsentiert durch das Literal 2). Nach einer Zuweisung

```
21 x := y
```

keine Einschränkungen bei der Zuweisung

bezeichnen x und y das gleiche Objekt (genau welches ist hier nicht ersichtlich); ob sie auch dasselbe bezeichnen, hängt von der Semantik der Variablen ab. Man beachte, dass in SMALLTALK (anders als in typisierten Sprachen) aus Sicht des Compilers nichts dagegenspricht, der Variable x erst eine Zahl und dann einen String zuzuweisen. Auch Array-Literale können jeder beliebigen Variable zugewiesen werden.

zwei Seiten einer Zuweisung

Man beachte weiterhin, dass die Zuweisung (anders als der Test auf Gleichheit = oder Identität ==) nicht kommutativ ist: x := y hat nur dann dieselbe Bedeutung wie y := x, wenn x und y schon vor der jeweiligen Zuweisung denselben Wert hatten. Zur besseren sprachlichen Unterscheidung der Seite, der zugewiesen wird, und der, die zugewiesen wird, spricht man häufig von der *linken und der rechten Seite einer Zuweisung*.

Beispiel

Nach den drei Zuweisungen

```
22 x := 5
23 y := 3
24 x := y
```

bezeichnen x und y beide die „3“. Wäre die letzte Zuweisung hingegen y := x gewesen, bezeichneten x und y beide „5“.

[7] Dass in C und allen davon abgeleiteten Sprachen (sowie in BASIC) das einfache Gleichheitszeichen = für die Zuweisung steht, darf als eine der Tragödien in der Geschichte der Programmiersprachen angesehen werden. Ich möchte nicht wissen, wie viele fatale Fehler auf die dadurch provozierte Verwechselung von Test auf Gleichheit und Zuweisung zurückzuführen sind.

Die Zuweisung ist ein elementares Konstrukt der objektorientierten Programmierung sowie der Programmierung überhaupt. Nur die wenigsten Sprachen kommen ohne sie aus. Neben der expliziten Zuweisung durch den Zuweisungsoperator kommt auch eine *implizite* (bei *Methodenaufrufen*) vor; diese wird jedoch erst in Abschnitt 4.3.2 behandelt.

Wert- und Verweissemantik bei der Zuweisung

Der oben geschilderte Unterschied zwischen Wert- und Verweissemantik von Variablen hat für die Zuweisung erhebliche Konsequenzen: Bei einer Zuweisung unter Wertsemantik muss, da die Variable das Objekt *zum Inhalt* hat (also in der Variable gespeichert ist) und *ein* Objekt nicht *in zwei* Variablen gespeichert sein kann, eine Kopie angefertigt werden. Das hat zur Folge, dass die beiden Variablen x und y nach der Zuweisung aus Zeile 21 nicht dasselbe (also identische) Objekt bezeichnen (was ja unter Wertsemantik, wie oben bereits gesagt, auch gar nicht geht), so dass z. B. Änderungen am in x gespeicherten Objekt das in y gespeicherte Objekt nicht betreffen. Bei einer Zuweisung unter Verweissemantik wird jedoch nur der Verweis der rechten Seite kopiert und in der Variablen auf der linken gespeichert. Wenn die Variablen auf der linken und der rechten Seite unterschiedliche Semantiken haben, dann liegt entweder eine *unzulässige Zuweisung* (s. Kapitel 18) vor oder es muss, je nach Art der Variable auf der linken Seite, eine Kopie eines Objektes oder ein Verweis auf ein Objekt angefertigt werden (s. dazu auch Abschnitt 52.5.2 in Lektion 5).

Selbsttestaufgabe 1.3

Finden Sie (durch Experimentieren) heraus, welche Objekte Ihres SMALLTALK-Systems per Wertsemantik gespeichert werden. Nutzen Sie dabei aus, das SMALLTALK vor der Erzeugung eines Objekts mit Ausnahme von Symbolen nicht prüft, ob das Objekt schon vorhanden ist.

Hinweis: Verwenden Sie den Identitätstest (==).

1.7 Pseudovariablen

Während es für Variablen charakteristisch ist, dass sich ihr Wert ändern kann, so sieht SMALLTALK dennoch einige vor, für die das nicht der Fall ist. Hier sind vor allem die Variablen mit Namen `true`, `false` und `nil` zu nennen, die auf Objekte entsprechender Bedeutung verweisen.[8] Für diese Variablen ist die Zuweisung nicht zulässig.

Eine ganze Reihe weiterer Variablen kann zwar ihren Wert ändern (also zu unterschiedlichen Zeiten auf verschiedene Objekte verweisen), jedoch erhalten sie ihren Wert vom System; auch diesen kann durch den *Zuweisungsoperator* `:=` kein Wert zugewiesen werden. Dies sind z. B. die Variablen mit Namen `self` und `super` sowie alle *formalen Parameter* von Methoden (s. Abschnitt 4.3). Nicht zuletzt sind auch alle Klassennamen (s. Lektion 2) Variablen, denen man als

[8] Die Pseudovariablen `true`, `false` und `nil` benennen spezielle, unveränderliche Objekte, auf deren Bedeutung wir noch ausführlich eingehen werden. Bis dahin kann der Leser davon ausgehen, dass `true` und `false` für die Booleschen Werte „wahr“ und „falsch“ stehen und `nil` für ein spezielles Objekt, das meistens „kein Objekt“ repräsentieren soll.

Programmierer nichts explizit zuweisen kann. All diese Variablen werden in SMALLTALK einheitlich **Pseudovariablen** genannt.

1.8 Aliasing

Wenn Variablen keine Objekte enthalten, sondern lediglich auf sie verweisen, wenn sie also Verweissemantik haben, ist es möglich, dass mehrere Variablen gleichzeitig dasselbe Objekt benennen. Das nennt man **Aliasing**. Das Aliasing ist eines der wichtigsten Phänomene der objektorientierten Programmierung; zugleich ist es leider nur wenig als solches bekannt. Versuchen Sie trotzdem, es sich stets bewusst zu machen — es wird Sie vor manch böser Überraschung bewahren.

Entstehung von Aliasing

Aliase, also weitere Namen für ein bereits benanntes Objekt, entstehen immer bei der Zuweisung. Dazu ist es notwendig, dass die Variable auf der linken Seite Verweissemantik hat. Da in SMALLTALK die Semantik von Variablen nicht mit der Variablendeklaration (s. Kapitel 19) festgelegt wird, sondern von der Art eines Objekts abhängt, ist nicht immer klar, bei welcher Zuweisung ein Alias entsteht. Dabei kann beides, die fälschliche Annahme von Verweissemantik bei tatsächlicher Wertsemantik und die fälschliche Annahme von Wertsemantik bei tatsächlicher Verweissemantik, zu erheblichen (und schwer zu findenden) Programmierfehlern führen.

Beispiel

Nach den beiden Zuweisungen

```
25 x := #Smalltalk
26 y := #Smalltalk
```

hat das eine Objekt, das der Compiler für `#Smalltalk` erzeugt, zwei Namen, nämlich `x` und `y`.

Effizienz durch Aliasing

Das Aliasing ist zunächst erwünscht: Da jedes Objekt nur einmal im Speicher hinterlegt werden muss, ermöglicht es die extrem effiziente Informationsverarbeitung (es ist weder ein Kopieren notwendig, wenn ein Objekt weitergereicht werden soll, noch müssen die Änderungen an den verschiedenen Kopien zusammengeführt werden, die notwendig sind, wenn die Kopien immer noch dasselbe logische Objekt bezeichnen sollen). Doch diese Effizienz hat ihren Preis.

mögliche Probleme durch Aliasing

Dass die Veränderung des durch eine Variable bezeichneten Objekts zugleich die Veränderung der durch all seine Aliase bezeichneten Objekte (die ja alle dieselben sind) bewirkt, kann nämlich unerwünscht, ja ein Programmierfehler sein. So könnte man beispielsweise bei den beiden Zuweisungen

```
27 petersNachname := 'Müller'
28 paulasNachname := petersNachname
```

lediglich bezwecken wollen, dass Peter und Paula *zunächst* gleich heißen, z. B. weil sie Geschwister sind. Bei einer späteren Promotion Paulas fügt sie die Zeichen `$D`, `$r` und `$.` in den

ihren Nachnamen repräsentierenden String ein, ändert also das entsprechende Objekt. Man hat nun sicher nicht beabsichtigt, dass das auch `petersNachname` betrifft, aber wenn die Änderung an einer weit entfernten Stelle im Programm erfolgt, ist die Identität der von `petersNachname` und `paulasNachname` benannten Objekte nicht mehr offensichtlich. Tatsächlich hat man es dann mit einem recht subtilen und schwer zu findenden Programmierfehler zu tun. Deswegen (und aufgrund etwas überzeugenderer Beispiele, die zu verwenden aber noch mehr Vorbereitung bedarf) sind in einigen SMALLTALK-Systemen alle auf Basis literaler Repräsentationen erzeugten Objekte als unveränderlich markiert (wenn Sie es nicht schon, wie beispielsweise Zahlen, von Haus aus sind), so dass Programmierfehler dieser Art vermieden werden. Sollte wie im obigen Beispiel eine Zuweisung mit Wertsemantik benötigt werden, so schreibt man statt Zeile 28 in SMALLTALK einfach

```
29 paulasNachname := petersNachname copy
```

Dabei sorgt das hintangestellte `copy` dafür, dass von dem Objekt, das durch `petersNachname` bezeichnet wird, eine Kopie angefertigt wird, also ein neues Objekt, das dem alten gleicht (mehr zur Syntax und dazu, wofür `copy` steht, folgt unten). Nicht nötig wird das Kopieren, wenn ich die Änderung durch die Zuweisung eines neuen Objekts bewerkstellige, wie das beispielsweise in

```
30 paulasNachname := 'Dr. Müller'
```

oder gar

```
31 paulasNachname := 'Dr. ' , paulasNachname
```

der Fall ist (wobei das Komma hier für die *String-Konkatenation* steht).

Fehler dieser Art sind häufig die Folge dessen, dass sich ein Programmierer der aliasbildenden Wirkung der Zuweisung nicht bewusst war. Das ist insbesondere bei den Programmierern der Fall, die nicht mit der objektorientierten Programmierung großgeworden sind, die insbesondere bei einer Zuweisung `y := x` das Kopieren des Inhalts der Variablen auf der rechten Seite (`x`) vermuten. Tatsächlich muss man in anderen Sprachen (wie beispielsweise PASCAL oder C) eine Variable explizit als *Pointervariable* deklarieren, um einen Alias bilden zu können. In SMALLTALK, genau wie in JAVA und C#, ist Aliasing jedoch der Regelfall und Kopie die Ausnahme. Wer das nicht verinnerlicht hat, schreibt höchstens zufällig korrekte Programme.

1.9 Lebenslauf von Objekten

In SMALLTALK beginnt der Lebenslauf eines Objekts mit seiner Erzeugung und endet mit seiner Entsorgung durch eine Speicherbereinigung, die sog. **Garbage collection**. Garbage collection ist ein Mechanismus, der Objekte aus dem Speicher entfernt, wenn diese nicht mehr zugreifbar sind. Da in SMALLTALK auf Objekte nach ihrer Erzeugung ausschließlich über Variablen (Namen) zugegriffen werden kann, kann ein solches Objekt genau dann entfernt werden, wenn keine Variable mehr auf es verweist. Es *kann* entfernt werden, muss aber nicht; aus Sicht des

Programmierers ist es ausreichend, dass das Objekt nicht mehr bekannt/benannt ist — es kann somit nicht mehr aufgefunden und durch eine Zuweisung einer Variable zugewiesen werden. Bei der Implementierung von Garbage-collection-Algorithmen besteht denn auch erhebliche Freiheit.

Beispiel

Wenn Peter und Michaela heiraten, dann schlägt sich dies u. a. in der Zuweisung

```
32 petersNachname := michaelasNachname
```

nieder. Wenn `'Müller'` keine Aliase (wie beispielsweise `paulasNachname`) hatte, kann es nach der Zuweisung aus dem Speicher entfernt werden — es wäre selbst bei Bedarf nicht mehr auffindbar.

Sonderbehandlung bestimmter Objekte

Von der automatischen Speicherbereinigung ausgenommen sind bestimmte Objekte mit eindeutiger literaler Repräsentation (wie z. B. kleine Zahlen, Zeichen und Symbole). Im Falle von Zahlen und Zeichen liegt das jedoch weniger an der Natur dieser Objekte selbst als vielmehr an der Tatsache, dass diese in der Regel nicht als Objekte im Speicher angelegt werden (so dass Variablen darauf verweisen könnten), sondern dass sie selbst, als Werte (und anstelle von Zeigern), in Variablen gespeichert werden (Abschnitt 1.5.1). Sie werden „entfernt“, indem einer Variable ein neuer Wert zugewiesen wird. Symbole werden schon deswegen nicht aus dem Speicher entfernt, weil sie in einer Symboltabelle abgelegt (und somit immer mindestens einmal referenziert; s. Fußnote 5) werden.

warum automatische Speicherbereinigung richtig und wichtig ist

Der Mut zur Verabschiedung von der expliziten Speicherfreigabe war eine der wichtigsten Entscheidungen beim Entwurf SMALLTALKs. Man hat einfach anerkannt, dass die genaue Buchführung darüber, auf welche Objekte noch zugegriffen wird, zu schwierig ist, um die Verantwortung dafür Anwendungsprogrammierern zu überlassen. Wem das Problem nicht unmittelbar einsichtig ist, der halte sich vor Augen, dass

- der Ort der Erzeugung eines Objektes und seine erste Zuweisung zu einer Variable im Programm möglicherweise weit entfernt sind von der Stelle, an der dieser Variable ein anderes Objekt zugewiesen wird, dass
- es möglicherweise viele solcher Stellen gibt, von denen mal die eine, mal die andere zuerst erreicht wird, und dass
- in der Zwischenzeit beliebig viele Aliase auf das Objekt angelegt worden sein können, die alle mitberücksichtigt werden müssen, um zu entscheiden, ob das Objekt noch in Verwendung ist.

Eine vorzeitige Entfernung aus dem Speicher hingegen führt dazu, dass Variablen ins Nichts zeigen (eine häufige Quelle von Programmabstürzen) oder dass, bei einer Wiederverwendung des Speichers, die Variable plötzlich auf ein anderes Objekt verweist, das ihr aber nie explizit

zugewiesen wurde — ein quasi zufälliges Programmverhalten, das mit hoher Wahrscheinlichkeit zu schweren Programmfehlern führen würde. Ein anderes Beispiel entsteht, wenn in einer Verzweigung eines Programms entweder ein neues oder ein bereits vorhandenes Objekt einer Variable zugewiesen wird. Woher weiß man bei der weiteren Benutzung dieser Variable, ob das Objekt schon vorher existierte und vielleicht schon andere Variablen auf es verweisen, oder ob es gerade erst neu erzeugt wurde und damit noch unbenutzt ist? Wer ist für die Entsorgung des Objekts verantwortlich? All diese Betrachtungen kann man sich in Gegenwart der Garbage collection ersparen.

„Lebenszyklus“ ist irreführend

Im objektorientierten Jargon spricht man übrigens häufig auch vom **Lebenszyklus** („life cycle“) eines Objekts. Genaugenommen ist dies aber irreführend, denn das Wort „Zyklus“ verspricht, dass das Leben nach seinem Ende wieder neu beginnt. Gerade dies ist aber, wie eben erläutert, nicht der Fall: Objekte werden nicht recycelt, sondern höchstens der Speicherplatz, den sie belegen.

1.10 Zusammenfassung

Objekte können also über textuelle Repräsentationen, die sog. *Literale*, in ein Programm eingeführt und *Variablen* zugewiesen werden. Dabei speichern die Variablen die Objekte in aller Regel nicht, sondern benennen sie lediglich. Technisch gesehen enthalten die Variablen dann *Verweise* (Referenzen, Zeiger, Pointer) auf die Objekte, die selbst durch Stellen im Speicher repräsentiert werden (sog. *Referenzsemantik*). Wenn mehrere Variablen dasselbe Objekt bezeichnen, spricht man von einem *Aliasing*. Aliasing erlaubt über die gemeinsame Nutzung von Objekten eine außerordentlich speicher- und recheneffiziente Informationsverarbeitung. Gleichzeitig stellt es aber eine der größten Fehlerquellen der objektorientierten Programmierung dar, da im Allgemeinen nicht bekannt ist, ob und wie viele Aliase auf ein Objekt existieren. Ein besonderes Problem liegt vor, wenn sich der Programmierer nicht bewusst ist, dass er (nur) mit Aliasen hantiert, und sich dann wundert, wenn sich an entfernten Orten plötzlich die (vermeintlichen, denn eigentlich sind es ja gar keine) Inhalte von Variablen ändern.

2 Beziehungen

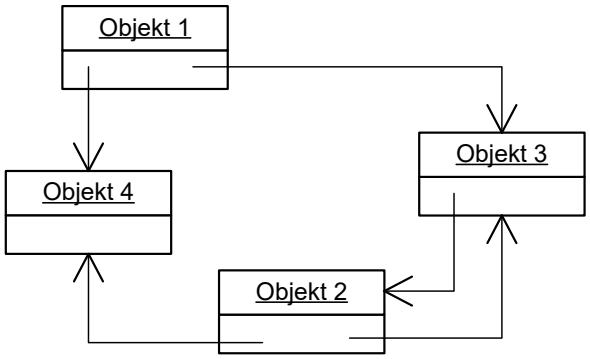
Kein Objekt ist eine Insel. Ganz im Gegenteil: Damit Objekte eine Bedeutung haben, müssen sie mit anderen in Beziehung stehen. Das Objekt „1“ beispielsweise ist ohne Bedeutung, solange es nicht eine bestimmte Eigenschaft eines anderen Objekts beschreibt, wie z. B. die Hausnummer eines Hauses oder die Anzahl der Elemente eines Arrays. Tatsächlich werden die meisten Objekte eines Systems erst durch ihre Beziehungen zu anderen zu etwas Nützlichem. Ein Objekt, das beispielsweise eine Person repräsentiert, macht den String „Hans Mustermann“ zum Namen der Person, sobald er mit der Person in entsprechender Beziehung steht; umgekehrt wird für den Benutzer des Systems die Person erst über den Namen identifizierbar.

Information als Geflecht von Objekten

Tatsächlich wird, wie bereits eingangs dieser Lektion erwähnt, in der objektorientierten Programmierung sämtliche Information als ein *Geflecht von Objekten* dargestellt. Dieses Geflecht kann

1. navigiert werden, um von einem Datum (Stück Information) zu einem anderen zu kommen, oder auch
2. manipuliert werden, um die repräsentierte Information zu verändern.

Das Datenmodell der objektorientierten Programmierung ähnelt damit stark dem *Netzwerkmodell*, das ebenfalls ein navigierendes ist, das vor einigen Jahrzehnten einmal die Grundlage großer Datenbankmanagementsysteme war, das aber schnell vom *relationalen Datenmodell* verdrängt wurde und das erst heute, im Zuge der Einführung von objektorientierten Datenbanken, wieder an (theoretischer) Bedeutung gewinnt.



Richtung von Beziehungen

In der objektorientierten Programmierung werden Beziehungen zwischen Objekten über Verweise hergestellt, durch deren Verfolgung man von einem Objekt zum nächsten gelangen, eben „navigieren“ kann. Das Besondere dieser Verknüpfung ist, dass sie stets gerichtet ist: Dass man von einem Objekt zum nächsten navigieren kann, heißt nicht, dass man auch wieder zurückkommt. Dazu ist dann ein Zeiger in Gegenrichtung nötig.

Nun enthalten ja Variablen ebenfalls Verweise. Wer also Zugriff auf die Variable hat, hat damit auch Zugriff auf das referenzierte Objekt — und ist somit mit dem Objekt verknüpft. Was noch fehlt, ist, dass Variablen Objekten so zugeordnet werden, dass nur noch die Objekte Zugriff darauf haben, und schon kann man auf einfache Weise Beziehungen ausdrücken.

2.1 Instanzvariablen

Jedem Objekt kann eine Menge von *lokalen Variablen* zugeordnet werden. Aus Gründen, auf die wir noch zu sprechen kommen, heißen diese Variablen **Instanzvariablen**; sie werden aber manchmal auch *Felder* oder *Attribute* (zu Attributen s. Abschnitt 2.4) genannt. Die Instanzvariablen eines Objekts sind in gewisser Weise in seinem Besitz: Sie sind für andere Objekte nicht sichtbar und damit auch nicht zugreifbar. Die Sichtbarkeit ist also auf das jeweils besitzende

Objekt eingeschränkt.[9] Außerdem ist die Existenz dieser Variablen an die Existenz (oder Lebensdauer; s. Abschnitt 1.9) des besitzenden Objekts gebunden.

Instanzvariablen und die Zusammensetzung von Objekten

Instanzvariablen bestimmen den Aufbau, oder die **Struktur**, zusammengesetzter Objekte (die manchmal deswegen auch *strukturierte Objekte* genannt werden) — atomare Objekte haben keine Instanzvariablen. Jede Instanzvariable eines Objekts belegt dabei einen Teil des Speichers aus seiner Repräsentation (s. Abschnitte 1.1 und 1.5). Da Instanzvariablen in der Regel Verweise enthalten und Verweise immer den gleichen Platz belegen, ist die Größe eines Objekts (der zu seiner Speicherung benötigte Platz) mit der Anzahl seiner Instanzvariablen festgelegt.

Unterscheidung von benannten und indizierten Instanzvariablen

In SMALLTALK werden zwei Arten von Instanzvariablen unterschieden: **benannte** und **indizierte**. Jede benannte Instanzvariable benennt (oder verweist auf) jeweils ein Objekt; der Name der Variable wird somit für die Dauer, die die Variable auf das Objekt verweist, auch zum Namen des Objekts. Da es sich bei Instanzvariablen um lokale Variablen handelt, muss der Name einer benannten Instanzvariablen in SMALLTALK mit einem Kleinbuchstaben beginnen.

indizierte Instanzvariablen

Indizierte Instanzvariablen haben keine Namen, sondern werden über einen Index relativ zu dem Objekt, zu dem sie gehören, angesprochen. Damit ist der Index gewissermaßen der Name der Instanzvariable. Der Index muss eine natürliche Zahl größer 0 sein. Um den Inhalt der indizierten Instanzvariable an der Indexposition `i` (genauer: an der Indexposition, die durch das Zahlobjekt bestimmt wird, auf das `i` verweist) zu erhalten, schreibt man

```
33 at: i
```

Wer bei indizierten Instanzvariablen an Arrays denkt, liegt richtig: Tatsächlich speichern Array-Objekte ihre Elemente in indizierten Instanzvariablen. So liefert beispielsweise

```
34 #($a $b $c) at: 2
```

das Zeichenobjekt „b“. Um den Wert einer indizierten Instanzvariable an derselben Indexposition zu setzen, schreibt man z. B.

```
35 #($a $b $c) at: 2 put: 'toll!'
```

[9] Während das in SMALLTALK Gesetz ist, weichen andere Sprachen (wie z. B. JAVA) diese Regel auf, indem sie öffentlich zugängliche Instanzvariablen erlauben und selbst für private nicht verhindern, dass ein anderes Objekt der gleichen Klasse darauf zugreift. Etwas anderes ist es jedoch auch in SMALLTALK mit den Objekten, auf die die Variablen verweisen: Aufgrund möglicher Aliase kann der Zugriff nicht so leicht einem einzigen Objekt vorbehalten werden.

(aber nur, wenn Ihr SMALLTALK die Änderung der Zusammensetzung für literale Arrays zulässt). Das resultierende Array-Objekt hat die literale Repräsentation `#($a 'toll!' $c)`.[10]

indizierte Instanzvariablen in anderen Sprachen

Indizierte Instanzvariablen sind kein Unikat von SMALLTALK: So bieten beispielsweise C# und VISUAL BASIC sog. *Indexer*, die im Wesentlichen indizierten Instanzvariablen entsprechen (s. Abschnitt 50.3.2 in Lektion 5). Auch verfügen manche Objekte in VISUAL BASIC FOR APPLICATIONS (VBA) über eine Variable `Item`, deren Elemente über Indizierung des Objekts, dem sie zugeordnet ist, angesprochen werden können.

Begrenzung der Anzahl indizierter Instanzvariablen

Die Anzahl der indizierten Instanzvariablen eines Objekts ist fix. Damit ist auch die Größe eines Objekts mit indizierten Instanzvariablen fest; insbesondere können Array-Objekte nicht wachsen (und wenn doch, dann nur über den in Abschnitt 1.1 erwähnten Trick mit dem Wechsel der Identität). Es müssen aber nicht alle indizierten Instanzvariablen belegt sein; die „leeren" enthalten dann `nil` (s. u.).

Zuordnung von Instanzvariablen zu Objekten

Es bleibt noch die Frage, wie Objekte in SMALLTALK in den Besitz von Instanzvariablen gelangen. Um das zu erklären, müsste an dieser Stelle auf das Konzept der Klasse vorgegriffen werden, was aber aus didaktischen Gründen unterbleiben soll. Gelernte PASCAL-Programmierer können sich die Instanzvariablen aber wie die Felder eines Records vorstellen (oder C-Programmierer wie die eines Structs), die in der Record-Definition festgelegt werden und die für jede Variable vom Typ dieses Records zur Verfügung stehen. Für alle anderen mag es reichen, sich vorzustellen, jedes Objekt verfüge automatisch über zwei spezielle Variablen, die die Namen und die zugewiesenen Objekte aller seiner Instanzvariablen verwalten. Wie so etwas gehen kann, wird in der nächsten Lektion klarwerden.

2.2 Unterscheidung von :1- und :*n*-Beziehungen

In der Daten- und Softwaremodellierung werden Beziehungen (oder Relationen) häufig mit sog. *Kardinalitäten* versehen. (Manchmal, besonders im Kontext der Softwaremodellierung mit der Unified Modeling Language *UML* werden diese auch *Multiplizitäten* genannt.) Sie geben an, mit wie vielen anderen Objekten ein Objekt gleichzeitig in derselben Beziehung stehen kann. Beispielsweise kann eine Person zu mehreren anderen Personen in einer Verwandtschaftsbeziehung stehen. Häufig sind die möglichen Kardinalitäten auf ein Intervall beschränkt; sie werden dann durch eben dieses Intervall beschrieben.

Von den theoretisch unendlich vielen möglichen Intervallen, die die Kardinalität beschränken können, sind vor allem drei interessant: [0..1], [1..1] und [0..∞). Dabei ist [1..1], also dass ein

[10] C# und VISUAL BASIC bieten sog. *Indexer*, die im wesentlichen indizierten Instanzvariablen entsprechen (s. Abschnitt 50.3.2 in Lektion 5). Manche Objekte in VISUAL BASIC FOR APPLICATIONS (VBA) verfügen über eine Variable `Item`, deren Elemente über `<objekt>.Item(n)`, aber auch direkt über `<objekt>(n)` angesprochen werden können.

Objekt immer mit genau einem in Beziehung stehen muss, technisch nur schwer umzusetzen, so dass [1..1] hier nicht weiter betrachtet wird[11]; die untere Schranke 0, die den beiden verbleibenden Intervallen gemeinsam ist und die ausdrückt, dass ein Objekt auch mit gar keinem anderen in der Beziehung stehen kann, muss daher nicht erwähnt werden. Im Fall von [0..1] sprechen wir also von **Zu-eins-Beziehungen** (im Folgenden mit **:1-Beziehung** notiert), im Fall von [0..∞) von **Zu-n-Beziehungen** (**:n-Beziehungen**), wobei *n* hier andeuten soll, dass es sich um eine nicht näher spezifizierte Zahl größer als 1 handelt.[12]

:1-Beziehungen

Die Beziehung eines Objekts zu einem anderen wird auf natürliche Weise durch eine benannte Instanzvariable ausgedrückt, wobei die Instanzvariable den Namen der Beziehung oder, besser noch, den Namen der Rolle des von der Variablen referenzierten Objektes in der Beziehung trägt. So zeigt die Instanzvariable `arbeitgeber` beispielsweise auf das Objekt, das in der Beziehung *Anstellung* aus Sicht des Arbeitnehmerobjekts die Rolle des Arbeitgebers spielt. Hat auch das Arbeitgeberobjekt einen Verweis auf das Arbeitnehmerobjekt (die Rückrichtung), so wird die entsprechende Variable sinnvollerweise nach der Gegenrolle `arbeitnehmer` genannt. Steht ein Arbeitnehmerobjekt zur Zeit in keinem Anstellungsverhältnis, ist seine Instanzvariable `arbeitgeber` *leer*, was in SMALLTALK durch den Verweis auf das Objekt `nil` ausgedrückt wird.[13]

:*n*-Beziehungen

Beziehungen sind nicht von Natur aus auf *ein* Gegenüber eingeschränkt: Ein Objekt kann, und wird häufig, in derselben Beziehung zu mehreren andern stehen. Genau dafür sind aber die indizierten Instanzvariablen wie geschaffen: Sie erlauben es, von einem Objekt zu beliebig vielen anderen Objekten zu navigieren, ohne für jedes andere eine eigene (jeweils anders) benannte Instanzvariable vorsehen zu müssen. Die „Namen“ der Gegenüber sind einfach Indizes: 1, 2, 3 usw.

Umsetzung mit Zwischenobjekten

Es ergibt sich nun aber das Problem, dass bei durch indizierte Instanzvariablen eines Objekts realisierten :*n*-Beziehungen nicht zwischen verschiedenen solchen Beziehungen desselben Objekts unterschieden werden kann — die indizierten Instanzvariablen sind ja nicht benannt. Deswegen werden :*n*-Beziehungen in der objektorientierten Programmierpraxis praktisch immer über **Zwischenobjekte** realisiert, deren Aufgabe es ist, mittels ihrer indizierten Instanzvariablen jeweils *eine* Beziehung zu mehreren anderen Objekten herzustellen. Dabei können diese Zwischenobjekte die :*n*-Beziehung ggf. mit weiteren

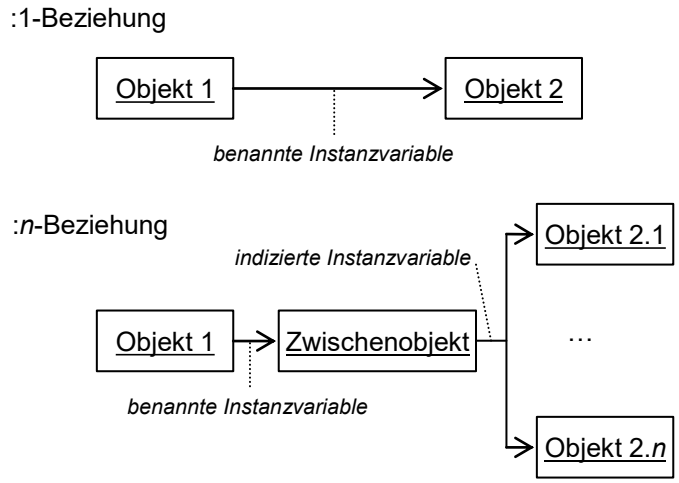
---

[11] Eine Beziehung zu keinem Objekt, wird, im Fall von [0..1], in der Regel durch eine Beziehung zum Objekt `nil` (`null` in anderen Sprachen) ausgedrückt. Nach und nach kommen in verschiedenen objektorientierten Programmiersprachen die sog. Not-null-Annotationen auf, die sicherstellen sollen, dass eine Variable nie den Wert `null` hat.

[12] Aus der relationalen Datenmodellierung sind 1:1-, 1:*n*- und *m*:*n*-Beziehungen bekannt, die Kardinalitäten auch für die Gegenrichtung angeben. Da wir es in der objektorientierten Programmierung aber nicht mit (ungerichteten, d. h. bidirektionalen) Relationen, sondern mit Verweisen (Zeigern) zu tun haben, entfällt die Rückrichtung.

[13] Diese Konvention wurde vom Turing-Preisträger SIR CHARLES ANTONY RICHARD HOARE eingeführt, der sie heute selbst als einen (seinen) „billion dollar mistake“ bezeichnet.

Attributen (z. B. Anzahl *n*, Verweise auf ein bestimmtes Element, Art der Sortierung o. ä.) versehen, die dann in den benannten Instanzvariablen der Zwischenobjekte untergebracht werden. Das Originalobjekt, das die :*n*-Beziehung eigentlich haben sollte, steht dann stattdessen in einer von einer benannten Instanzvariable hergestellten :1-Beziehung zu dem Zwischenobjekt, das die :*n*-Beziehung herstellt.



Vorteile von Zwischenobjekten

Wie wir noch sehen werden, erlaubt der Umstand, dass :*n*-Beziehungen über Zwischenobjekte realisiert werden, die vollwertige Objekte sind, die Beziehungen beliebig auszugestalten. So kann beispielsweise eine (Sortier-)Reihenfolge vorgegeben oder ein ausgezeichnetes Element der Beziehung noch einmal gesondert referenziert werden (z. B. das oberste Element auf einem Stack). Auch besondere Zugriffsverfahren wie z. B. das Auffinden von Elementen (in Beziehung stehenden Objekten) anhand eines Schlüssels können auf diese Weise realisiert werden. Da in SMALLTALK Objekte auch eigene Kontrollstrukturen (wie z. B. spezielle Schleifen) anbieten können, sind der Ausgestaltung von Beziehungen über Zwischenobjekte praktisch keine Grenzen gesetzt.

effiziente Bearbeitung von :*n*-Beziehungen entscheidend für die Produktivität

Da :*n*-Beziehungen häufig vorkommen, ist ihre Handhabung von entscheidender Bedeutung für die Ausdrucksstärke der verwendeten Programmiersprache und die Produktivität der Programmierung insgesamt. Wie sich schon in Abschnitt 4.6.4 zeigen wird, erlaubt die Ausgestaltung von Zwischenobjekten in SMALLTALK Möglichkeiten, die bis heute Vorbildcharakter für andere objektorientierte Programmiersprachen haben.

2.3 Teil-Ganzes-Beziehungen

Eine Sonderrolle unter den Beziehungen nimmt die sog. *Teil-Ganzes-Beziehung*, je nach Kontext und Jargon auch *Komposition* oder *Aggregation* genannt, ein. Teil-Ganzes-Beziehungen bestimmen ganz wesentlich unsere Weltsicht: Alles, was wir anfassen oder betrachten können, ist aus kleineren Teilen zusammengesetzt, die selbst wieder Zusammensetzungen (Aggregate, Komposita) sind bis hinunter zu den Elementar-, d. h., unteilbaren Bausteinen.

keine allgemeingültige Definition der Teil-Ganzes-Beziehung

Nun ist der Begriff der Teil-Ganzes-Beziehung leider nicht so klar definiert, wie es auf den ersten Blick scheint. Tatsächlich bestehen, je nach Art der Zusammensetzung, zum Teil völlig unterschiedliche Wechselwirkungen zwischen dem Ganzen und seinen Teilen. Zudem gibt es neben der physischen Teil-Ganzes-Beziehung auch eine logische: So ist z. B. der Deutsche Bundestag aus einer Anzahl von Abgeordneten zusammengesetzt und jede Familie aus ihren Mitgliedern. Tatsächlich gibt es so viele Varianten der Teil-Ganzes-Beziehung, dass der (philosophische) Diskurs darüber ganze Regale füllt und zu einer eigenen Disziplin geführt hat (der sog. *Mereologie*). Verständlicherweise kann dem eine Programmiersprache nicht folgen und für jede dieser Beziehungen ein eigenes Sprachkonstrukt anbieten.

WIKIPEDIA

mangelnde Sprachunterstützung

Stattdessen bieten die meisten (objektorientierten) Programmiersprachen aber leider überhaupt kein Sprachkonstrukt an, das speziell für die Teil-Ganzes-Beziehung gedacht wäre. Gleichwohl kann man die Unterscheidung zwischen Instanzvariablen mit Referenz- und Wertsemantik, falls vorhanden, dazu nutzen, um zumindest eine spezielle Form der Teil-Ganzes-Beziehung abzubilden: Da bei Wertsemantik mit der Entfernung eines Objekts aus dem Speicher auch alle Objekte, die als Werte seiner Instanzvariablen dienen, aus dem Speicher entfernt werden, kann man hier tatsächlich von der Umsetzung einer bestimmten Form von Teil-Ganzes-Beziehung sprechen, nämlich einer solchen, bei der die Existenz der Teile von der Existenz des Ganzen abhängt (in der UML auch *Komposition* genannt). Für andere Formen, wie z. B. die Bildung einer Gruppe von Objekten als Objekt mit eigener Identität (einem Verein beispielsweise), ist dieses Modell aber nicht geeignet, da sonst mit Auflösung der Gruppe auch die Gruppenmitglieder verschwinden müssten. Für den SMALLTALK-Programmierer sind diese Überlegungen aber sowieso kein Thema, denn er hat die Wahl erst gar nicht.

Mit der Teil-Ganzes-Beziehung auf Programmebene werden wir uns in Lektion 6 (genauer: Kapitel 58 und 59) noch ausführlicher beschäftigen. Hier sei nur schon soviel gesagt, dass die Möglichkeit des (rekursiven) Aufbaus eines Software-Systems aus Teilen, die jeweils ihren inneren Aufbau (ihre Komposition) *kapseln*, also insbesondere die Nichtexistenz von Aliasen auf ihre Teile garantieren, genau das ist, was der objektorientierten Programmierung im Wesentlichen bis heute fehlt.

2.4 Attribute

Logisch kann man Instanzvariablen in zwei Kategorien aufteilen: solche, die Eigenschaften eines Objekts festhalten, und solche, die tatsächliche Beziehungen zwischen Objekten repräsentieren. Typische Eigenschaften sind beispielsweise die Farbe von etwas oder der Name; sie grenzen sich von Beziehungen inhaltlich dadurch ab, dass das bezogene Objekt isoliert betrachtet seine Bedeutung verliert. So ist das Objekt `rot` allein nichts weiter als eine Farbe — erst als Attribut eines Objekts (wie z. B. „Apfel“) bekommt es eine Bedeutung. Das gleiche gilt für „Schmidtchen“ oder „1“. Dazu kommt, dass Attributwerte wie die vorgenannten in der Regel selbst keine Attribute haben oder Beziehungen mit anderen Objekten eingehen. Man beachte

jedoch, dass dieses Unterscheidungskriterium nicht absolut, sondern relativ zur jeweils betrachteten Domäne ist: Wenn es z. B. um Farben geht, ist „rot“ ein Objekt, das für sich genommen schon eine Bedeutung hat und das mit anderen Objekten vollwertige Beziehungen eingeht, so z. B. mit „grün“ als seinem Komplementärkontrast.

Attribute mit Wertsemantik

Wenn wir eben von *Attributwerten* sprachen, so ist das nicht ganz zufällig: Nicht selten haben Variablen, die Attribute repräsentieren, zumindest logisch eine Wertsemantik, d. h., sie halten (oder verweisen auf, je nach Implementierung der Sprache) eigene Kopien eines Objekts. Ein typisches Beispiel hierfür hatten Sie in Abschnitt 1.8 bereits kennengelernt: So ist es sinnvoll, dass die Änderung des Namens (genauer: des Namensobjekts) bei einer Person nicht gleichzeitig andere Personen, die den gleichen (nicht denselben!) Namen haben, betrifft. Eine ähnliche Überlegung spielt im Zusammenhang mit der Betrachtung des Zustandes eines Objekts eine Rolle.

3 Zustand

Wie bereits in Abschnitt 1.3 beschrieben, kann man zwischen *veränderlichen* und *unveränderlichen Objekten* unterscheiden. Veränderungen veränderlicher Objekte erfolgen über die Zeit und die Objekte wechseln dabei ihren **Zustand**; unveränderliche Objekte dagegen haben keinen Zustand.[14] Was aber macht den Zustand eines Objektes aus?

Änderung des Zustands eines Objektes

Der Zustand eines Objektes ist die Summe der Belegungen seiner Instanzvariablen. Insofern Instanzvariablen Beziehungen ausdrücken, wird der Zustand eines Objekts ausschließlich durch seine Verknüpfung mit anderen Objekten definiert. Zudem folgt, dass die einzige Möglichkeit, den Zustand eines Objekts zu ändern, der über die Zuweisung von Instanzvariablen, gleichbedeutend mit der Änderung seiner Beziehungen, ist.

3.1 Eingrenzung

Attribute und Zustand

Eine etwas eingeschränktere Sicht vom Zustand eines Objektes bezieht nur seine *Attributwerte* ein. Dies setzt jedoch voraus, dass überhaupt formal zwischen Attributen und Beziehungen unterschieden werden kann. In Ermangelung spezieller Schlüsselwörter könnte dies, wie bereits oben diskutiert, allenfalls über die Unterscheidung von Variablen mit Wert- und solchen mit Referenzsemantik erfolgen. Dies ist jedoch schon in Sprachen, die dem Programmierer eine entsprechende Entscheidung anbieten, nicht automatisch der Fall (z. B. JAVA, da hier Strings zwar unveränderlich sind und somit eigentlich zu den Werten zählen, aber

[14] Wenn sie einen hätten, dann immer denselben. Da der Begriff des Zustandes jedoch ohne die Möglichkeit der Veränderung kaum einen Sinn hat, sprechen wir auch nicht vom Zustand unveränderlicher Objekte wie z. B. „1“. Manche SMALLTALK-Dialekte sehen jedoch eine Instanzvariable `immutable` vor, die die Veränderlichkeit von Objekten festlegt und die, so ihr Wert denn geändert werden kann, die (Un-)Veränderlichkeit selbst zu einem Teil des Zustandes von Objekten macht.

dennoch Referenzsemantik haben); in SMALLTALK schließlich ist, da alle Instanzvariablen Referenzen enthalten können, diese Einschränkung überhaupt nicht anwendbar.

Reichweite von Zustandsänderungen

Man könnte nun die obige Aussage, dass Zustandsänderungen eines Objekts ausschließlich über Zuweisungen an seine Instanzvariablen erfolgen können, anzweifeln und fragen, ob sich der Zustand eines Objektes auch dann ändert, wenn sich der Zustand eines Objektes, auf das es (per Instanzvariable) verweist, ändert? Zunächst würde man diese Frage mit nein beantworten, denn sonst würde ja, da Objekte in der Regel stark miteinander verflochten sind, die Änderung des Zustandes eines Objektes fast immer zu einer wahren Kettenreaktion führen: Der Zustand aller Objekte, die direkt oder indirekt — also über dritte — darauf verweisen, würde sich mit ändern. Diese Definition von Zustand würde jedoch kaum unserem Weltbild entsprechen: Die Änderung Ihres Familienstandes beispielsweise würde kaum etwas an meinem Zustand ändern, obwohl Sie meine Lehrtexte geschickt bekommen, ich also in einer (direkten oder indirekten) Beziehung zu Ihnen stehe. Wie aber ist es, wenn ich meinen Namen ändere? Ändert sich dann mein Zustand?

Sonderfall Strings

Während man diese Frage sicher mit ja beantworten wird, ist doch mein Name als String ein eigenständiges Objekt, das seinen Zustand wechselt, wenn ich, wie im Beispiel von Abschnitt 1.8 geschehen, einzelne Buchstaben in ihm austausche oder ergänze. Nach obiger Auffassung ändert sich mein Zustand (als Besitzer des Namens) dadurch nicht, denn es bleibt ja dasselbe (per Identität) String-Objekt, das meinen Namen hält — es hat lediglich seinen Zustand gewechselt. Etwas anderes wäre es hingegen, wenn ich das String-Objekt, das meinen Namen repräsentiert, gegen ein anderes austausche: Dann verweist die entsprechende Instanzvariable auf ein anderes Objekt und mein Zustand hat sich sicher geändert. Gleichwohl würde man dasselbe auch von der Namensänderung per String-Manipulation erwarten, nur technisch lässt sich dieser Sonderfall nicht begründen! Wie Sie sehen, ist das objektorientierte Denkmodell nicht ganz ohne Tücken, und Programmierfehler schleichen sich an Stellen ein, die so simpel aussehen, dass man dort nie einen Fehler vermuten würde. Vielleicht auch deswegen ist es sinnvoll, wenn Strings, wie von einigen SMALLTALK-Dialekten (und übrigens auch von JAVA) vorgesehen, immer unveränderlich (immutable) sind.

Zustand und Komponenten

Auch wenn es nicht sinnvoll ist, für ein allgemeines Objektgeflecht den Zustandsbegriff auf mehrere Objekte auszudehnen, so ist dies für Kompositionen, also aus Teilen zusammengesetzten Ganzen (vgl. Abschnitt 2.3), durchaus angemessen: Wenn z. B. ein Dokument aus mehreren Seiten besteht, die wiederum jeweils aus Zeilen und Spalten bestehen, dann ändert die Änderung einer Zeile auch den Zustand des gesamten Dokuments. Dies wird weiter unten noch eine Rolle spielen, vor allem im Zusammenhang mit Aliasing: Wenn man nämlich davon ausgeht, dass Zustandsänderungen eines Objekts stets Sache des Objekts selbst sind, dann dürfen keine externen Aliase auf seine Teile existieren, denn sonst könnte ihm eine Zustandsänderung von außen quasi untergejubelt werden. Man sollte den Zustand also kapseln.

3.2 Kapselung

lokale Variablen und das Geheimnisprinzip

Die Unterscheidung von lokalen und globalen Variablen aus Abschnitt 1.5.2 dient u. a. dem Verbergen von Geheimnissen (das sog. **Geheimnisprinzip** engl. auch **Information hiding** genannt), genauer von **Implementationsgeheimnissen**. So ist es fast immer sinnvoll, die Struktur zusammengesetzter Objekte vor den sie benutzenden Objekten zu verbergen, damit man diese Struktur später ändern kann, ohne dass die benutzenden (abhängigen) Objekte davon betroffen wären. Derartige Änderungsabhängigkeiten werden verhindert, wenn die Variablen von außen gar nicht zugreifbar sind, was für lokale Variablen, die ja von außen unsichtbar sind, automatisch der Fall ist.

lokale Variablen und Kapselung

Vom Geheimnisprinzip abzugrenzen ist der Begriff der **Kapselung**, den man mit der objektorientierten Programmierung verbindet: Ein Objekt soll seinen Zustand kapseln, so dass dieser nur von ihm selbst geändert werden kann. Anders als beim Information hiding geht es bei der Kapselung also nicht um Änderungen des Aufbaus von Objekten, sondern um Änderungen ihres Zustandes. Leider lässt sich die Kapselung nicht mit denselben Mitteln wie das Geheimnisprinzip umsetzen: Aufgrund des Aliasing kann ein Objekt, dessen einer Name (beispielsweise aufgrund des Geheimnisprinzips) unsichtbar ist, über einen anderen zugreifbar sein, ohne dass der erste Name etwas dagegen machen könnte. Über lokale Instanzvariablen kann ein Objekt also verbergen, welche Objekte es kennt; es kann aber nicht verhindern, dass andere Objekte diese Objekte auch kennen und, ohne sein Wissen, manipulieren. Es ist somit wegen der etwaigen Existenz von Aliasen nicht möglich, dass ein Objekt seinen inneren Aufbau vor der Außenwelt *abkapselt*, es sein denn, es hat ganz spezielle Vorkehrungen dafür getroffen. Da diese Vorkehrungen (derzeit noch) in keine gängige objektorientierte Programmiersprache eingebaut sind[15], sondern explizit programmiert werden müssen, werden wir uns hier (in dieser Lektion) nicht weiter damit beschäftigen; eine weitergehende Betrachtung erfolgt in Kapitel 58 von Lektion 6).

4 Verhalten

Wenn Objekte ihren Zustand kapseln, ist es ausschließlich ihr Verhalten, welches ihn ändert, welches also die Zustandsänderungen eines Objekts herbeiführt. Umgekehrt hängt das Verhalten eines Objekts in der Regel von seinem Zustand ab. Wie aber wird das Verhalten eines Objekts beschrieben? Bevor wir uns dieser Frage zuwenden, müssen wir erst wir zunächst den Begriff des *Ausdrucks* klären.

[15] Die Programmiersprache Rust ist hier vielleicht eine Ausnahme.

4.1 Ausdrücke

In einem objektorientierten Programm können Objekte nicht nur durch Literale (Abschnitt 1.2) und Variablen (Abschnitt 1.5) repräsentiert werden, sondern auch durch **Ausdrücke**. Tatsächlich sind Literale und Variablen *primitive Ausdrücke*, also solche, die nicht aus anderen zusammengesetzt sind. Damit mit den Objekten aber etwas geschehen und ein Programm somit etwas tun kann, brauchen wir noch andere Ausdrücke, namentlich *Zuweisungsausdrücke* und *Nachrichtenausdrücke*. Auch sie stehen jeweils für ein Objekt und können deswegen überall da auftreten, wo Objekte verlangt werden. Sie können insbesondere auch geschachtelt werden.

4.1.1 Zuweisungsausdrücke

Zuweisungsausdrücke hatten Sie schon in Abschnitt 1.6 kennengelernt. Sie verlangen auf der linken Seite eine Variable und auf der rechten einen Ausdruck:

```
36 x := 1
```

Zustandswechsel von Objekten durch Zuweisungen

etwa ist ein Zuweisungsausdruck. Wie bereits erwähnt, bewirken Zuweisungen als einzige Ausdrücke den Zustandswechsel von Objekten. So bewirkt etwa

```
37 name := 'Hänschen'
```

wobei `name` eine Instanzvariable sein soll, die Änderung des Zustandes des Objektes, zu dem die Instanzvariable gehört.

Da eine Zuweisung selbst ein Ausdruck ist, kann sie auf der rechten Seite einer anderen Zuweisung erscheinen:

```
38 y := x := 1
```

ist also ein legaler Ausdruck (zu seiner Auswertung kommen wir später, wenn wir — in Abschnitt 4.1.4 — über die Reihenfolge der Auswertung geschachtelter Ausdrücke sprechen).

4.1.2 Nachrichtenausdrücke

Neben der Zuweisung ist der **Nachrichtenversand** die zweite wichtige **Ausdrucksform** der objektorientierten Programmierung. SMALLTALK verwendet hierfür eine Syntax, die stark an die der englischen Sprache angelehnt ist: Sie verlangt ein Subjekt (den Empfänger der Nachricht), ein Prädikat (die **Nachricht**) sowie eine optionale Liste von Objekten als Prädikatsergänzungen (die Parameter der Nachricht). Dabei wird auf die in anderen Sprachen übliche Verwendung des Punkts als Trennzeichen zwischen Empfänger und Nachricht und Klammern zum Umschließen der Parameterliste verzichtet; stattdessen verwendet man bei zwei oder mehr Parametern Partikeln (Präpositionen oder Konjunktionen) ähnelnde Nachrichtenteile, die den Parametern vorangestellt werden. Die allgemeine Syntax lautet also

```
<Empfänger> <Nachricht>
```

für parameterlose Nachrichten,

```
<Empfänger> <Nachricht> <Parameter>
```

für Nachrichten mit einem Parameter,

```
<Empfänger>
  <NachrichtTeil1> <Parameter1>
  <NachrichtTeil2> <Parameter2>
```

für Nachrichten mit zwei Parametern usw., wobei hier die in spitzen Klammern stehenden Namen *metasyntaktische Variablen*, also Platzhalter für Namen in einem konkreten Programm, sind und die Nachrichten(teile), die Parameter nach sich ziehen, immer mit einem Doppelpunkt enden müssen.

```
einObjekt tueEtwas
```

drückt beispielsweise den Versand einer parameterlosen Nachricht `tueEtwas` an das Objekt, auf das die Variable `einObjekt` verweist, aus.[16] Ein etwas konkreteres Beispiel hierfür ist der Ausdruck

```
#abc printString
```

mit dem dem Objekt `#abc` die Nachricht `printString` gesendet wird.

Soll einer Nachricht ein Parameter angehängt werden, so tut das

```
einObjekt tueEtwasMit: einemParameter
```

wobei `einemParameter` hier auch eine Variable ist (es könnte auch ein anderer Ausdruck dort stehen; s. u.). Ein konkretes Beispiel hierfür ist

```
Transcript show: 'Hallo'
```

Ein zweiter Parameter wird durch einen weiteren Nachrichtenbestandteil wie in

```
einObjekt tueEtwasMit: einemParameter und: zweitemParameter
```

hinzugefügt und alle weiteren entsprechend, also etwa

```
einObjekt
  tueEtwasMit: einemParameter
  und: zweitemParameter
```

[16] Es ist übrigens Sitte (aber keineswegs verpflichtend), in SMALLTALK die Zusammensetzung von Namen durch sog. Binnenmajuskeln (eingefügte Großbuchstaben) und nicht durch den Unterstrich kenntlich zu machen.

```
52   und: drittem Parameter
```

Nachrichtenselektor

wobei sich die Nachrichtenteile wie oben ruhig wiederholen dürfen. Es ist also insbesondere nicht so, dass die Reihenfolge der Nachrichtenteile beliebig umgestellt werden dürfte, ohne dass sich dadurch die Bedeutung der Nachricht änderte. Tatsächlich ist die an das Objekt geschickte Nachricht, der sog. **Nachrichtenselektor** (engl. message selector), immer *ein* Symbol, das aus der Konkatenation (Aneinanderfügung) aller seiner Teile, also im Beispiel der Zeile 48 oben `#tueEtwasMit:und:` besteht. Die Namen der Nachrichtenteile sind frei wählbar, beginnen jedoch per Konvention mit einem Kleinbuchstaben.

unäre und binäre Nachrichten

Parameterlose Nachrichten wie beispielsweise `tueEtwas` in Zeile 44 nennt man in SMALLTALK übrigens **unär** (unary messages): Obwohl sie keine expliziten Argumente haben, heißen sie trotzdem unär, weil der Empfänger das erste, implizite Argument ist. So nehmen denn auch sog. **binäre Nachrichten**[17] nur einen Parameter, was aber zwei Argumenten, nämlich dem Empfänger und einem weiteren Argument, entspricht:

```
53 1 + 2
```

beispielsweise drückt aus, dass die Nachricht „+“ mit Argument „2“ an das Objekt „1“ gesendet werden soll.

In SMALLTALK sind binäre Nachrichten eine syntaktische Besonderheit: Sie bestehen nämlich aus einem oder mehreren nicht alphanumerischen, nicht reservierten Zeichen (die Liste der reservierten Zeichen finden Sie in Kapitel 5 am Ende dieser Lektion). Alle anderen Nachrichten, die neben dem Empfänger noch mindestens ein Argument erfordern, werden in SMALLTALK dagegen **Schlüsselwortnachrichten** (keyword messages) genannt, so z. B. `tueEtwasMit:` in Zeile 46 und `tueEtwasMit:und:` in Zeile 48. Dies ist jedoch etwas verwirrend, da sie keine Schlüsselwörter im landläufigen Sinne (von denen SMALLTALK ja gar keine hat) enthalten und da natürlich auch die unären Nachrichten „Schlüsselwörter“ verwenden.

Sender und Empfänger

Ein Nachrichtenausdruck besteht also aus einem Empfängerobjekt, einem Nachrichtenselektor sowie einer Anzahl von Argumentausdrücken, die die Teile des Nachrichtenselektors sperren (dazwischen stehen).[18] Der Ausdruck als ganzes steht für ein Objekt, nämlich das Ergebnis der Auswertung der Nachricht durch den Empfänger. Der Nachrichtensender wird dem Empfänger übrigens nicht mitgeteilt (es sei denn, er wird explizit als Parameter übergeben) — das System weiß automatisch, wohin die Antwort auf die Nachricht zurückgeliefert werden soll (nämlich genau an die Stelle, an der der Nachrichtenversand

[17] nicht zu verwechseln mit den sog. *binären Methoden*, bei denen Empfänger und Parameter den gleichen Typ haben

[18] Nur in Ausnahmefällen ist es sinnvoll, das Empfängerobjekt mit einer Nachricht als (zusätzlichen) Parameter zu übergeben; in den allermeisten Fällen handelt es sich dabei um einen Anfängerfehler, bei dem durchscheint, dass die Programmiererin (die vermutlich noch in Kategorien der prozeduralen Programmierung denkt) nicht verstanden hat, dass jede Nachricht einen Empfänger hat, ein Nachrichtenausdruck diesen also schon beinhaltet.

steht). Mehr dazu gleich, wenn es um die Auswertung von Nachrichtenausdrücken geht (Abschnitt 4.1.3).

geschachtelte Nachrichtenausdrücke

Da ein Nachrichtenausdruck für ein Objekt steht, kann er selbst Teil eines Nachrichtenausdrucks sein, also für den Empfänger der Nachricht oder einen ihrer Parameter stehen. Es ist dann allerdings u. U. notwendig, den so geschachtelten Nachrichtenausdruck zu klammern, da er sonst nicht richtig erkannt werden kann:

```
54 aStack
55   push: ((aStack pop)
56             arg: (aStack pop)
57             arg: (aStack pop))
```

beispielsweise schiebt das Objekt auf den Stack, das Resultat der Nachricht `arg:arg:` gesandt an das oberste Objekt des Stacks mit seinem zweiten und dritten als Parameter ist. (Zu den genauen Regeln zur Reihenfolge der Auswertungen siehe Abschnitt 4.1.4.)

kaskadierte Nachrichtenausdrücke

Nicht selten möchte man eine Serie von Ausdrücken an dasselbe Empfängerobjekt senden. SMALLTALK sieht dafür mit der Kaskadierung eine nette syntaktische Abkürzung vor, die es erlaubt, bei einer Sequenz von Nachrichten an dasselbe Objekt dieses nicht immer wiederholen zu müssen. So sorgt beispielsweise

```
58 Transcript show: '1'; show: #+; show: '2'; cr.
```

anstelle des wesentlich wortreicheren

```
59 Transcript show: '1'.
60 Transcript show: #+.
61 Transcript show: '2'.
62 Transcript cr
```

dafür, dass die Objekte „1“, „+“ und „2“ nacheinander als Parameter der Nachricht `show:` an das von der globalen Variable `Transcript` benannte Objekt, eine Art Systemkonsole, gesendet werden. (Auf die Bedeutung des „.“ kommen wir in Kapitel 4.2 zu sprechen.) Das Beispiel ist übrigens nicht besonders zwingend, da man in SMALLTALK genauso gut auch hätte

```
63 Transcript show: '1', #+ , '2'; cr
```

schreiben können (wobei `,` ein binärer Nachrichtenselektor ist, der für die Konkatenation steht). `cr` ist übrigens eine (unäre) Nachricht, die für einen Wagenrücklauf (carriage return) auf dem Transcript sorgt.

Kaskadierung anstelle temporärer Variablen

Man kann sich aber durch Verwendung der Kaskadierung häufig die Einführung einer *temporären Variablen* (s. Abschnitt 4.3) ersparen, und zwar immer dann, wenn anstelle einer Variable als Empfänger (`Transcript` in obigem Beispiel) ein Ausdruck steht, der das Empfängerobjekt liefert und der nur einmal ausgewertet werden soll. Anstelle von

```
64 eineVariable := <irgendein Ausdruck>.
65 eineVariable <eine Nachricht>.
66 eineVariable <noch eine Nachricht>
```

kann man also

```
67 <irgendein Ausdruck> <eine Nachricht>; <noch eine Nachricht>
```

schreiben, wenn man `eineVariable` hernach nicht mehr benötigt. SMALLTALK unterstützt also von Haus aus sog. *Fluent APIs*.

4.1.3 Auswertung von Ausdrücken

Ausdrücke allein sind nur syntaktische Figuren — um das Objekt zu liefern, für das sie stehen, müssen sie ausgewertet werden. Es ist die Auswertung, bei der ein Programm tatsächlich „etwas tut“.

Auswertung von Zuweisungsausdrücken

Das Elementarste, das ein Programm tun kann, ist die Auswertung einer Zuweisung wie beispielsweise `y := x`. Diese Auswertung führt dazu, dass die Variable y nach der Zuweisung auf dasselbe Objekt verweist wie vorher schon x. Dabei steht der gesamte Ausdruck `y := x` für (eine Referenz auf) das Objekt, auf das x (und nach seiner Auswertung auch y) verweist. Die Zuweisung dieses Objektes an y ist gewissermaßen nur ein *Seiteneffekt* der Auswertung des Ausdrucks.[19]

Man beachte, dass es sich beim *Zuweisungsoperator* := nicht um einen (binären) Nachrichtenselektor handelt: Es wird hier nämlich keine Nachricht an ein Objekt geschickt, sondern der Inhalt einer Variable manipuliert. Die Zuweisung ist tatsächlich eines der wenigen Primitive SMALLTALKs, also eine der Operationen, die nicht in sich selbst programmiert, sondern fest „verdrahteter“ Teil der Sprache sind (vgl. Abschnitt 1.6).

Auswertung von Nachrichtenausdrücken

Nachrichtenausdrücke werden ausgewertet, in dem die Nachricht (das Prädikat des Satzes) an das Empfängerobjekt (das Subjekt) mit den Parametern (den Prädikatsergänzungen) gesendet wird und das Empfängerobjekt als Ergebnis der Nachricht ein Objekt zurückliefert. Wie das Empfängerobjekt das Ergebnisobjekt bestimmt, darauf kommen wir noch; hier ist nur wichtig, dass der Nachrichtenausdruck im Zuge der Auswertung gewissermaßen durch das Objekt, für das er steht, ersetzt wird. Man kann sich das genauso vorstellen wie die Auswertung einer (mathematischen) Funktion *f*, deren Funktionsanwendung *f*(*x*) auf ein Objekt *x* ebenfalls für den Funktionswert steht.

[19] Über das Wort Seiteneffekt wird viel gestritten. Manche meinen, es müsste *Nebeneffekt* heißen, aber dieses Wort ist genauso unnötig, wenn *Nebenwirkung* das ist, was gemeint ist. Als strittig würde ich schon eher ansehen, ob man hier überhaupt von einer Nebenwirkung sprechen sollte, denn der sich einstellende Effekt ist ja das vornehmliche Ziel einer Zuweisung, so dass man eher von Wirkung (oder Effekt) sprechen sollte. Für mich ist *Seiteneffekt* Jargon, bei dem jeder weiß, was gemeint ist.

Beispiele

Eine häufig gebrauchte, von allen Objekten verstandene unäre Nachricht ist `printString`, in Reaktion auf die das Empfängerobjekt eine textuelle Repräsentation von sich zurückgibt:

```
68 12 printString
```

liefert den String `'12'`, die Auswertung von Zeile 45 den String `'abc'`. Der bereits in Zeile 53 angeführte binäre Nachrichtenausdruck

```
69 1 + 2
```

liefert wie erhofft das Objekt „3“. Bei der Auswertung des Schlüsselwortausdrucks

```
70 12 printOn: Transcript
```

bei dem als Seiteneffekt die Zahl „12“ in ihrer textuellen Repräsentation, also als String `'12'` auf dem Ausgabestrom, auf den die Variable `Transcript` verweist, ausgegeben wird, wird das Empfängerobjekt „12“ zurückgeliefert.

Nachrichtenausdrücke als Funktionsanwendungen

Es ist in SMALLTALK, anders als z. B. bei als `void` deklarierten Methoden in JAVA oder C#, nicht möglich, in Reaktion auf einen Nachrichtenversand nichts zurückzugegeben. Dies könnte man in Zeile 70 als sinnvoll erachten, da man hier eigentlich nur an dem *(Seiten-) Effekt*, der Ausgabe von „12“ auf dem Transcript, interessiert ist. Wenn die zu einer Nachricht gehörende Methode keinen sinnvollen Rückgabewert hat, wird *standardmäßig immer das Empfängerobjekt* zurückgegeben; auch das dient, da man Nachrichtenausdrücke so stets verketten kann, einem *Fluent API*. Sollte es für das Rückgabeobjekt im Kontext der Nachricht (des Nachrichtenversands) keine Verwendung geben, es also nicht einer Variable zugewiesen oder sonst wie Teil eines anderen Ausdrucks sein, verfällt es einfach. Da der Rückgabewert aber technisch nur eine Referenz auf das Objekt ist, bleibt die Existenz des Objektes davon zunächst unberührt. Nur wenn es sonst keine anderen Referenzen auf das Objekt gibt, etwa weil es extra für die Rückgabe erzeugt wurde, wird das Objekt bei der nächsten *Speicherbereinigung* entfernt.

Auswertung eines Nachrichtenausdrucks als Methodenaufruf

Da die Auswertung eines Nachrichtenausdrucks zur Abarbeitung der Anweisungen des Rumpfs einer Methode führt, nennt man ihn auch *Methodenaufruf*. Warum diese Bezeichnung legitim ist, wird jedoch erst in Abschnitt 4.3 klar.

Zusammenfassung

Zusammenfassend wird also

- ein Literal zu dem Objekt ausgewertet, das es repräsentiert,
- eine Variable zu dem Objekt, das sie benennt,
- eine Zuweisung zu dem Objekt, zu dem der Ausdruck auf der rechten Seite der Zuweisung ausgewertet wird,

- ein Nachrichtenausdruck zu dem Objekt, das als Antwort auf die Nachricht zurückgegeben wird, sowie
- ein kaskadierter Nachrichtenausdruck zu dem Objekt, zu dem sein letzter Nachrichtenausdruck ausgewertet wird.

4.1.4 Reihenfolge der Auswertung von geschachtelten Ausdrücken

Da Ausdrücke andere Ausdrücke enthalten können, stellt sich die Frage nach der Reihenfolge der Auswertung von geschachtelten Ausdrücken. Diese wird, wie in anderen Sprachen auch, implizit über Präzedenzen und explizit über Klammern geregelt.

Bei der doppelten Zuweisung in Zeile 38 oben ist zunächst vielleicht nicht klar, welche der beiden Zuweisungen zuerst ausgewertet (ausgeführt) werden soll: `y := x` oder `x := 1`. Wenn man jedoch weiß, dass es sich bei `y := x := 1` um einen geschachtelten Ausdruck handelt und jeder (Teil-)Ausdruck für ein Objekt steht, dann muss der zweite Ausdruck zuerst ausgewertet und durch das Ergebnis, „1“, ersetzt werden, denn andernfalls wäre die „1“ dem Ergebnis von `y := x` zuzuweisen, was aber nach Abschnitt 4.1.3 keine Variable, sondern ein Objekt ist. Zeile 38 weist also zuerst x und dann y das Objekt „1“ zu — die Auswertung erfolgt von rechts nach links.

Auswertung von links nach rechts

Das ist allerdings ein Sonderfall. Grundsätzlich werden in SMALLTALK nämlich alle Ausdrücke von links nach rechts ausgewertet. Dabei haben allerdings unäre Ausdrücke Vorrang vor binären und diese wiederum Vorrang vor Schlüsselwortnachrichten, so dass nur bei gleichrangigen Ausdrücken die Auswertung von links nach rechts erfolgt. So wird beispielsweise in

```
71 x + 1 < y
```

zunächst die Nachricht „+“ an das Objekt, auf das x verweist, mit Argument 1 geschickt und an das Ergebnis die Nachricht „<“ mit y als Parameter. Umgekehrt wird bei

```
72 x < y - 1
```

zunächst der Vergleich angestellt und an das Ergebnis, eines der beiden Objekte „true“ und „false“, die Nachricht „–“ mit Argument 1 geschickt. Jedoch wird der Wahrheitswert (das Wahrheitsobjekt) die Nachricht nicht verstehen und entsprechend mit einer Meldung wie „Nachricht nicht verstanden“ reagieren. Um die Präzedenz zu ändern, kann man einfach Klammern setzen:

```
73 x < (y - 1)
```

hat das gewünschte Ergebnis. Individuelle Operatorpräzedenzen (wie z. B. Punktrechnung vor Strichrechnung) kennt SMALLTALK nicht.

mehrstellige Schlüsselwort-nachrichten

Bei mehrstelligen Nachrichten mit Schlüsselwörtern werden alle Schlüsselwörter eines Ausdrucks als zu einer Nachricht gehörig interpretiert, es sei denn, es wurden Klammern gesetzt. Der Ausdruck

```
einEmpfänger
  tueDiesMitDem: erstesArgument
  nachdemDuDasGetanHast: zweitesArgument
```

wird also als *eine* Nachricht interpretiert; falls

```
einEmpfänger
  tueDiesMitDem: (erstesArgument
                        nachdemDuDasGetanHast: zweitesArgument)
```

gemeint gewesen sein sollte, müssen eben die Klammern entsprechend gesetzt werden. (Man beachte, dass im geklammerten Ausdruck `erstesArgument` Empfänger der Nachricht `nachdemDuDasGetanHast:` mit `zweitesArgument` als Parameter ist. Das Ergebnis der Auswertung dieses Ausdrucks ist dann Parameter der Nachricht `tueDiesMitDem:` an `einEmpfänger`.)

4.2 Anweisungen

Anweisungen spezifizieren die Schritte, in denen ein Programm ausgeführt wird. In SMALLTALK sind alle Ausdrücke, die nicht Bestandteil von anderen Ausdrücken sind, Anweisungen. Während ihnen in JAVA und C# dazu noch ein Semikolon hintangestellt werden muss, ist Vergleichbares in SMALLTALK nicht nötig.

der Punkt als Trennzeichen

Folgen von Anweisungen werden in SMALLTALK durch einen Punkt getrennt. Dabei handelt es sich (genau wie in PASCAL oder EIFFEL) um ein Trennzeichen und nicht (wie in den von C abgeleiteten Sprachen wie JAVA oder C#) um einen Teil der Anweisung selbst. Der Punkt am Ende einer Anweisung darf also fehlen, wenn keine weitere Anweisung folgt. Ansonsten entspricht die Wahl des Punktes dem Vorsatz SMALLTALKs, der natürlichen Sprache möglichst ähnlich zu sein. So ist auch die Wahl des Semikolons zur Kaskadierung von Nachrichtenausdrücken zu sehen.

Return-Anweisung

Die einzige andere Form der Anweisung in SMALLTALK ist die **Return-Anweisung**. Auf sie werden wir im Zusammenhang mit Methoden im nächsten Kapitel noch ausführlicher zu sprechen kommen. Sie besteht in SMALLTALK aus dem Sonderzeichen ^ (ursprünglich ↑, jedoch genau wie ← auf den meisten Tastaturen nicht verfügbar), gefolgt von einem Ausdruck. Die Return-Anweisung „retourniert“ das Objekt, zu dem dieser Ausdruck auswertet.

Da alle anderen Anweisungen Ausdrücke sind, die zu einem Objekt auswerten, brauchen Methoden (Abschnitt 4.3) und Blöcke (Abschnitt 4.4) in SMALLTALK keine Return-Anweisungen, um ein Objekt zurückzuliefern; sie liefern dann das Objekt, zu dem die letzte Anweisung auswertet.

4.3 Methoden

Die Auswertung von Nachrichtenausdrücken, also von Nachrichten, die an Objekte verschickt werden, erfolgt mit Hilfe sog. **Methoden**. Was ein Objekt in Reaktion auf den Erhalt einer entsprechenden Nachricht tun soll, wird durch eine **Methodendefinition** beschreiben. Eine Methodendefinition besteht dazu aus einem *Methodenkopf*, der in SMALLTALK auch als *Message pattern*, allgemein (und im Folgenden) aber eher als **Methodensignatur** bezeichnet wird, einer optionalen Liste von *lokalen Variablen* und einem *Methodenrumpf*. Letzterer enthält die *Anweisungen*, die die Methode ausmachen, die also zur Auswertung eines Nachrichtenausdrucks ausgeführt werden. Es ist üblich, jede Methodendefinition mit einem in doppelte Anführungsstriche gesetzten Kommentar zu versehen, der beschreibt, was diese Methode macht.

formale und tatsächliche Parameter

Die Methodensignatur besteht aus dem Namen der Methode und der Liste ihrer **formalen Parameter**. Formale Parameter sind *lokale Variablen*, denen beim Aufruf der Methode (s. Abschnitt 4.3.2) automatisch ein Wert, der sog. **tatsächliche Parameter**[20], zugewiesen wird und deren Sichtbarkeit auf die Methode, in deren Methodensignatur sie vorkommen, beschränkt ist. Wie alle lokalen Variablen müssen sie mit einem Kleinbuchstaben beginnen. Außerdem sind sie *temporäre Variablen*, was soviel heißt wie dass sie nur für die Dauer der Ausführung der Methode existieren. Damit sind auch die Aliase, die durch formale Parameter gebildet werden können, immer temporär. In SMALLTALK sind formale Parameter zudem *Pseudovariablen* (s. Abschnitt 1.7), d. h., es kann ihnen innerhalb der Methode, für die sie sichtbar sind, nichts zugewiesen werden. Syntaktisch unterscheidet sich SMALLTALK von den meisten anderen Programmiersprachen auch bei der Methodendefinition dadurch, dass die formalen Parameter nicht durch Kommata getrennt in einer in Klammern eingeschlossenen Liste hinter dem Methodennamen stehen, sondern jeder für sich von einem Nachrichtenbestandteil eingeleitet wird.

Schema der Methodendefinition

Eine Methodendefinition folgt also dem Schema

```
80 <Methodensignatur>
81       "<Kommentar>"
82    | <Liste lokaler Variablen> |
83    <Folge von Anweisungen>
```

(in spitzen Klammern wieder *metasyntaktische Variablen*, also Platzhalter für entsprechende Programmelemente). Eine einfache Methode wäre etwa

```
84 helloWorld
85       "das berühmt-berüchtigte"
86    | |
```

[20] Der im Englischen gebrauchte Term „actual parameter“ wird auch manchmal mit „*aktueller Parameter*“ ins Deutsche übersetzt, was oft als falsch verhöhnt wird, aber die Sache trotzdem trifft, zumindest, wenn man die zeitliche Dimension, die mit der Programmausführung einhergeht, in Betracht zieht: Es gibt nämlich in der Regel viele tatsächliche Parameter zu einem Methodenaufruf, von denen — zu jedem Zeitpunkt — höchstens einer der aktuelle ist.

```
87    Transcript show: 'Hello World!'; cr
```

oder, wer es generischer mag,

```
88 sag: einenText
89      "nicht so unflexibel"
90    | |
91    Transcript show: einenText; cr
```

Die Methodensignatur als Teil einer Methodendefinition wird im Folgenden immer fett hervorgehoben.

Die Methodensignatur dient der Auswahl der zu einer Nachricht passenden Methode; sie ist das Gegenstück zum *Nachrichtenselektor* aus Abschnitt 4.1.2, anhand dessen die Auswahl der zu einem Nachrichtenausdruck passenden Methode durchgeführt wird. Anders als ein Nachrichtenausdruck nennt eine Methodensignatur aber kein Empfängerobjekt und die offenen Stellen eines Nachrichtenselektors werden ausschließlich durch Variablen, eben die *formalen Parameter*, und nicht durch beliebige Ausdrücke besetzt. Typische Methodensignaturen sind z. B.

```
92 printString
```

für Methoden ohne Parameter,

```
93 + einInteger
```

für binäre Methoden[21] (mit `einInteger` als formalem Parameter) und

```
94 printOn: einStrom
```

für alle anderen Methoden mit einem oder mehreren Parametern (in diesem Fall mit nur einem formalen Parameter, `einStrom`). Die Methodensignaturen „passen“ übrigens jeweils zu den entsprechenden Nachrichtenausdrücken aus den Zeilen 68, 69 und 70. Wenn der Nachrichtenselektor aus mehreren Teilen besteht, werden diese (entsprechend den Beispielen in den Zeilen 48 ff.) einfach angehängt.

Rückkehr von einer Methode und Rückgabe eines Wertes

Wie bereits erwähnt, besteht der Methodenrumpf aus einer Folge von Anweisungen, Ausdrücken, die jeweils durch einen Punkt getrennt sind. Wenn die Anweisungen nichts Anderes vorsehen, wird die Ausführung einer Methode nach Abarbeitung der letzten Anweisung implizit mit der Rückgabe des Empfängerobjekts an den Sender der Nachricht beendet. Für explizite Beendigungen und Rückgabe eines anderen Objekts als des Empfängers ist die *Return-Anweisung* (Abschnitt 4.2) vorgesehen.

[21] Der Begriff der binären Methode bezeichnet hier zweistellige Methoden, die eine Sonderzeichenfolge als Name verwenden. In der Literatur wird er häufig anders gebraucht: Der Empfänger und der Parameter einer binären Methode haben denselben Typ (vgl. Fußnote 17 und Abschnitt 29.5).

```
95 helloWorld
96   Transcript show: 'Hello World!'; cr
97   ^ 'Hello World!'
```

explizite Beendigung einer Methode

Eine Return-Anweisung darf an beliebigen Stellen innerhalb der Methode auftreten. Die Abarbeitung der Methode kann demnach auch vor Erreichen der textuell letzten Anweisung beendet werden. Die Return-Anweisung beeinflusst also den *Kontrollfluss* des Programms. Wichtig ist, dass eine Methode immer ein Objekt zurückgibt; ein Nachrichtenausdruck (oder *Methodenaufruf*; s Abschnitt 4.3.2) steht als Ausdruck also immer für ein Objekt. Prozeduren im Sinne PASCALs oder Void-Methoden im Sinne von C, JAVA usw. gibt es in SMALLTALK nicht (vgl. a. Abschnitt 4.1.3).

temporäre Variablen

Sollte eine Methode zur Durchführung ihrer Berechnungen *temporäre Variablen* benötigen, so müssen diese zu Beginn der Methode (nach der Methodensignatur und vor der ersten Anweisung) deklariert werden. Die Werte dieser Variablen, die standardmäßig mit `nil` initialisiert werden, sind außerhalb der Methode nicht sichtbar; die Variablen werden insbesondere nach Abarbeitung der Methode vom System wieder entfernt. Sie können sich also auch zwischen zwei Ausführungen einer Methode nichts merken.

Temporäre Variablen können auch der besseren Lesbarkeit dienen, indem sie Zwischenergebnissen einen Namen geben:

```
98 helloWorldX2
99   | einmal zweimal |
100  einmal := 'Hello World!'.
101  zweimal := einmal , einmal.
102  Transcript show: zweimal; cr
```

Umgekehrt können temporäre Variablen, die nur einmal verwendet werden, eingespart werden, indem man kaskadierte Nachrichtenausdrücke (Abschnitt 4.1.2) verwendet.

Methoden als Programmbausteine

Methoden sind die Einheiten des Programms, in denen Sie als Programmierer Ihre Anweisungen unterbringen. Sie werden nach der Eingabe (und bei jeder Änderung) mit dem „Speichern“ kompiliert („Speichern“ deswegen in Anführungsstrichen, weil Methoden nicht in Dateien gespeichert werden, sondern in einer Datenstruktur SMALLTALKs, und zwar in Form von Objekten). Tatsächlich besteht der Löwenanteil eines jeden SMALLTALK-Programms, ja des gesamten SMALLTALK-Systems, aus Methodendefinitionen. Die Methodendefinitionen entsprechen im Wesentlichen der Definition von Funktionen (oder, mit obiger Einschränkung, Prozeduren) in anderen Sprachen; jedoch ist es in SMALLTALK nicht möglich (und in der objektorientierten Programmierung allgemein nicht üblich), Methoden zu schachteln, also eine Methode innerhalb einer anderen zu deklarieren. Außerdem gibt es in SMALLTALK keine „Hauptmethode“ wie etwa die Main-Methoden in der C-Sprachfamilie. Sie müssen dem SMALLTALK-System schon sagen, welche Methode Sie ausgeführt haben wollen, z. B. indem Sie einen entsprechenden Ausdruck eingeben und auswerten lassen.

4.3.1 Zuordnung von Methoden zu Objekten

Methoden sind immer Objekten, den sogenannten Empfängerobjekten, zugeordnet. Wir wollen uns in dieser Lektion noch nicht darauf festlegen, wie diese Zuordnung erfolgt; Sie dürfen jedoch davon ausgehen, dass jedes Objekt über einen Katalog von Methoden verfügt und damit auf die entsprechenden Nachrichten reagieren kann. Diesen Katalog nennt man auch das **Protokoll** eines Objekts (s. Abschnitt 4.3.8).

Zugriff auf Instanzvariablen

Die Zuordnung von Methoden zu Objekten erlaubt, dass die Methoden auf die Instanzvariablen des jeweiligen Objekts zugreifen können. Da der Zustand eines Objekts durch die Belegung seiner Instanzvariablen repräsentiert wird (s. Kapitel 3) und weil in den Methoden das Verhalten eines Objektes spezifiziert ist, ergibt sich, dass das Verhalten vom Zustand des Objekts abhängen (durch Berücksichtigung seiner Instanzvariablen) und es ihn gleichzeitig beeinflussen (durch Zuweisungen an die Instanzvariablen) kann. Da Instanzvariablen lokale Variablen sind (lokal zum besitzenden Objekt), sind Methoden sogar die einzige Stelle, an der der Zustand von Objekten geändert werden kann. Mehr dazu in Abschnitt 4.3.4.

die Pseudovariable `self`

Neben den *formalen Parametern* der Methode und den Instanzvariablen des Empfängerobjekts ist im Rumpf jeder Methode eine weitere Variable zugreifbar, die den Namen „**self**“ trägt. Diese Variable, die wie die formalen Parameter eine *Pseudovariable* ist, verweist immer auf das Empfängerobjekt der Nachricht, also auf das Objekt, dessen Instanzvariablen gerade zugreifbar sind. Sie wird immer dann benötigt, wenn eine Nachricht aus einer Methode heraus (also per darin enthaltener Anweisung) an das Objekt geschickt werden soll, dem die Methode zugeordnet ist, also an sich selbst. `self` ist also gewissermaßen der implizite erste Parameter einer Methode.

4.3.2 Methodenaufruf und dynamisches Binden

Wenn das Versenden von Nachrichten bislang als die Übergabe eines entsprechenden Nachrichtenobjekts an den Empfänger dargestellt wurde, so ist das nicht ganz richtig: Vielmehr wird ein Nachrichtenausdruck auch in SMALLTALK aus Effizienzgründen vom Compiler in einen schnöden **Methodenaufruf** übersetzt, der mit dem Funktionsaufruf aus der prozeduralen Programmierung (also z. B. PASCAL oder C) vergleichbar ist. So führt beispielsweise der Ausdruck

```
103 1 + 2
```

zum Aufruf der Methode +, wie sie für das Objekt „1“ (und alle Zahlen) definiert wurde. Dennoch wird, wohl aus didaktischen Gründen, das Mysterium vom Nachrichtenversand in der objektorientierten Literatur weiter gepflegt. Es gibt aber auch einen kleinen, feinen Unterschied zum gewöhnlichen Prozeduraufruf.

Abhängigkeit vom Empfängerobjekt

Die Entscheidung, welche Methode in Reaktion auf einen Nachrichtenversand aufgerufen und abgearbeitet wird, hängt nicht von dem Nachrichtenselektor allein ab, sondern auch von dem Objekt, an das die Nachricht geschickt wird. Es ist

nämlich durchaus üblich, dass verschiedene Objekte mit gleichen Methodensignaturen unterschiedliche Methodenimplementierungen verbinden; so implementieren beispielsweise Zahlen und Symbole die Methode `printString` jeweils anders und selbst

```
104 1.0 + 2
```

führt zum Aufruf einer anderen Methode als `1 + 2`,

```
105 1 + 2.0
```

dagegen nicht.

dynamisches Binden

Aus der Abhängigkeit des Methodenaufrufs vom Empfängerobjekt folgt, dass nicht immer schon zur Übersetzungszeit entschieden werden kann, welche Methodenimplementierung bei einem Methodenaufruf ausgewählt werden muss. Wenn nämlich das Empfängerobjekt durch eine Variable benannt oder von einem Ausdruck geliefert wird, kann die Zuordnung einer Methodendefinition zu einem Nachrichtenausdruck erst zum Zeitpunkt der Auswertung des Nachrichtenausdrucks und damit erst zur Laufzeit erfolgen. Man nennt diesen Vorgang **dynamisches Binden** (im Gegensatz zum **statischen Binden**, bei dem ein Aufruf schon zur Übersetzungszeit an eine Implementierung gebunden wird); es handelt sich dabei um eine von den nur zwei *primitiven Kontrollstrukturen* SMALLTALKs (s. Abschnitt 4.5.2).

zentrale Bedeutung des dynamischen Bindens

Das dynamische Binden ist eine der charakteristischen Eigenschaften der objektorientierten Programmierung. Sie wird auch als **Polymorphismus** oder **Polymorphie**[22] bezeichnet. Auf die Details des dynamischen Bindens können wir erst in der nächsten Lektion (Kapitel 12) zu sprechen kommen und auf Polymorphie erst in Kapitel 26, da uns hier noch zu viel fehlt. Wir vermerken aber schon jetzt, dass es sich dabei um eine *versteckte Fallunterscheidung* handelt: Ein und derselbe Methodenaufruf kann fallweise unterschiedliche Methoden aufrufen (bzw. deren Ausführung veranlassen). Auf diese Eigenschaft kommen wir schon in den Abschnitten 4.5 und 4.6 zurück.

implizite Zuweisungen der tatsächlichen an die formalen Parameter

Steht einmal fest, welche (Implementierung einer) Methode aufgerufen wird, erfolgt als nächstes die Versorgung der formalen Parameter mit Objekten. Zu diesem Zweck findet mit dem Aufruf eine **implizite Zuweisung** (also eine ohne Vorkommen des Zuweisungsoperators im Programmtext) der *tatsächlichen Parameter* des Aufrufs an die formalen Parameter der Methode statt. Tatsächliche Parameter sind dabei die Objekte, die an der Methodenaufrufstelle als Argumente an den Positionen der formalen Parameter der aufgerufenen Methode stehen. Manchmal werden auch die Variablen, die an den entsprechenden Stellen beim Methodenaufruf stehen, als tatsächliche Parameter bezeichnet, aber erstens müssen dort nicht unbedingt Variablen, sondern können beliebige Ausdrücke stehen und zweitens können diese Variablen selbst formale Parameter

[22] Es ist mir bis heute nicht klar, wann man von *Polymorphie* und wann von *Polymorphismus* spricht. Im Gebrauch scheint sich anzudeuten, dass man von Inklusionspolymorphie und von parametrischem Polymorphismus spricht, aber einen Grund dafür kann ich nicht erkennen.

sein, nämlich die der Methode, die den Methodenaufruf enthält. So ist in der Methodendefinition

```
106 m: a
107   self n: a
```

a der formale Parameter von `m:`. Der tatsächliche Parameter von `m:` ist ein Objekt, das beim Aufruf von `m:` genannt wird (hier nicht zu sehen). Dieses Objekt ist dann auch tatsächlicher Parameter des Aufrufs von `n:`, da dieser von a, dem formalen Parameter von `m:`, geliefert wird.

implizite Zuweisung des Rückgabewertes

Je nach Sichtweise erfolgt bei Ausführung der Return-Anweisung eine weitere implizite Zuweisung, nämlich die des Objekts, zu dem der Ausdruck der Return-Anweisung auswertet, an die „Variable“ Methodenname (der ja an der Stelle des Methodenaufrufs ähnlich wie eine Variable für ein Objekt steht). Dies ist vor allem im Zusammenhang mit der Typprüfung, die es jedoch in SMALLTALK nicht gibt (s. Lektion 3), eine wichtige Vorstellung. An der Aufrufstelle selbst steht dann häufig noch eine *explizite Zuweisung*, nämlich wenn das Ergebnis des Aufrufs (das Rückgabeobjekt) einer Variable zugewiesen werden soll.

Selbsttestaufgabe 4.1

Benennen Sie die expliziten und impliziten Zuweisungen für

```
108 a := self f: e
```

mit der Methode

```
109 f: g
110   ^ 2
```

Call by reference und Call by value

In Abschnitt 1.5 waren wir auf den Unterschied zwischen Wertsemantik und Verweissemantik eingegangen. Damit verwandt (und ähnlich benannt) ist die Unterscheidung von **Call by reference** und **Call by value** beim Methodenaufruf: Beim Call by value wird dem formalen Parameter der tatsächliche Parameter als Wert zugewiesen, beim Call by reference hingegen nur eine Referenz. Diese Referenz ist jedoch nicht, wie man vielleicht glauben könnte, eine auf den tatsächlichen Parameter als Objekt, sondern eine auf den tatsächlichen Parameter als Variable (s. o.), der dazu aber eben auch eine Variable sein muss. Dies hat zur Folge, dass Zuweisungen zum formalen Parameter in der Methode unter Call by reference auch die Variable, die den tatsächlichen Parameter darstellt, betreffen — die beiden sind gewissermaßen eins. Mit Call by reference ist es also möglich, dass eine Methode auch über ihre tatsächlichen Parameter, wenn sie denn Variablen sind, Objekte zurückgibt — sie werden damit zu Ein- und Ausgabeparametern der Methode. Bei Call by value bleibt die Zuweisung an die formalen Parameter jedoch ohne Bedeutung für die tatsächlichen — sie sind also reine Eingabeparameter.

Unterschied zu Referenz- und Wertsemantik; SMALLTALK kennt nur Call by value

Nun werden Sie vielleicht einwenden, dass in SMALLTALK formale Parameter Pseudovariablen sind und deswegen gar keine Zuweisung an sie erlaubt ist (außer der *impliziten Zuweisung* beim Aufruf). Das ist richtig. Tatsächlich gibt es in SMALLTALK ein Call by reference auch gar nicht (in JAVA übrigens auch nicht). Gleichzeitig haben aber in SMALLTALK Variablen grundsätzlich Referenzsemantik, so dass bei der Zuweisung der tatsächlichen an die formalen Parameter keine Objekte, sondern lediglich Verweise an diese übergeben werden. Wenn man dann innerhalb der Methode etwas an diesen Objekten ändert (ihren Zustand), dann betrifft das immer die tatsächlichen Parameterobjekte und somit auch den „Inhalt" der tatsächlichen Parametervariablen (wenn es denn Variablen und keine anderen Ausdrücke sind und wenn sie wirklich ein Objekt zum Inhalt hätten; vgl. Abschnitt 1.5): Es sind schließlich dieselben Objekte. Insbesondere erhalten die Methoden also keine Kopien dieser Objekte, sondern lediglich Kopien der Referenzen. Um den Inhalt von tatsächlichen Parametervariablen zu ändern (also sie auf andere Objekte zeigen zu lassen), bräuchte man auch in SMALLTALK (und JAVA) Call by reference, was es dort aber nicht gibt. Es stellt dies eine echte Beschränkung der Programmierung dar; sie wurde denn auch in C# aufgehoben.

Aufruf nicht vorhandener Methoden

Bleibt noch die Frage, was passiert, wenn ein Methodenaufruf ins Leere läuft. Da in SMALLTALK Ausdrücke beliebige Objekte liefern können, kann der Compiler für einen Nachrichtenausdruck nicht garantieren, dass das Empfängerobjekt auch über eine entsprechende Methode verfügt. Da der Nachrichtenausdruck in einen dynamisch gebundenen Methodenaufruf übersetzt wird, dessen Ausführung direkt von der virtuellen Maschine SMALLTALKs vorgenommen wird, ist die Frage, wie das Programm sinnvoll mit einem solchen Laufzeitfehler umgehen soll. Tatsächlich passiert in etwa folgendes: Die virtuelle Maschine macht aus dem Methodenaufruf einen Nachrichtenselektor (und zwar den, aus dem er bei der Übersetzung hervorgegangen ist) und sendet diesen als Parameter an eine vorgegebene Methode `doesNotUnderstand:` des ursprünglichen Empfängers. Diese reagiert typischerweise mit einer der Ausgabe einer Fehlermeldung `<Objekt> does not understand: <Nachrichtenselektor>`; sie kann aber geändert werden, um anders als standardmäßig vorgesehen auf den Fehler zu reagieren.

4.3.3 Methoden als Parameter

... und es gibt ihn doch, den Nachrichtenversand

Nun wurde eingangs dieses Kapitels vom Mysterium des Nachrichtenversands gesprochen. Tatsächlich ist aber zumindest in SMALLTALK seine Realisierung in Form von Methodenaufrufen nur ein Zugeständnis an die Ausführungseffizienz. Und so sind denn auch in SMALLTALK Nachrichten (oder vielmehr Nachrichtenselektoren) auch Objekte — schließlich soll dort ja alles ein Objekt sein. Um tatsächlich als Nachrichtenobjekt an ein Objekt verschickt zu werden, muss man sich aber einer speziellen Methode (genauer: eines speziellen Methodenaufrufs), `perform:`, bedienen, der es erlaubt, einem Empfängerobjekt eine Nachricht als Objekt (wenn auch nur als Parameter von `perform:`) zu senden. Das Empfängerobjekt reagiert darauf mit der Abarbeitung der zur Nachricht passenden Methode ganz so, als hätte es direkt einen entsprechenden Methodenaufruf erhalten. So wertet zum Beispiel

```
111 nil perform: #isNil
```

genau wie

```
112 nil isNil
```

zu `true` aus. Der Nachrichtenselektor ist immer ein Symbol (`#isNil` im gegebenen Beispiel) und darf beim „Versand“ mittels `perform:`, anders als beim direkten Aufruf, auch in einer Variable gespeichert sein. Bei binären und höherstelligen Nachrichten braucht man zusätzlich noch die Argumente (Parameter) zum Nachrichtenselektor; sie können durch Erweiterung von `perform:` zu `perform:with:`, `perform:with:with:` usw. angehängt werden. So ist dann beispielsweise der Ausdruck

```
113 1 perform: #+ with: 2
```

äquivalent zu `1 + 2`.

4.3.4 Verbergen der Repräsentation des Zustands hinter Methoden

In SMALLTALK sind die Instanzvariablen eines Objekts nur für das Objekt selbst sichtbar (und damit auch zugreifbar).[23] Genauer sind seine Methoden die einzigen Stellen im ganzen Programm, an denen auf die Instanzvariablen des Objekts, dem die Methoden zugeordnet sind, direkt zugegriffen werden kann. Die Struktur des Objekts bleibt somit verborgen (das *Implementationsgeheimnis*) und sein *Zustand* wird *gekapselt* (s. Abschnitt 3.2 sowie unten für ein praktisches Beispiel).

Zugriffsmethoden, Getter und Setter

Um die Belegung der Instanzvariablen und damit den Zustand eines Objektes auszulesen oder verändern zu können, sind also Methoden notwendig. Um beispielsweise den Wert einer Instanzvariable mit Namen a auszulesen, muss das Objekt eine Methode vorsehen, die den Wert von a zurückgibt. Diese (parameterlose) Methode wird in SMALLTALK üblicherweise wie folgt notiert:

```
114 a
115     "gib den Wert von a zurück"
116   ^ a
```

Sie entspricht im wesentlichem einem auch in JAVA gebräuchlichen sog. **Getter** einer ansonsten nicht zugreifbaren Variable. Die Namensgleichheit von Methode und Variable soll dabei nicht darüber hinwegtäuschen, dass es sich bei beiden um verschiedene Dinge handelt — Methode und Variable könnten genauso gut verschieden benannt werden.

[23] Dass dies in anderen Sprachen anders ist (z. B. in JAVA), kann man durchaus als Entwurfsfehler ansehen. Auf der anderen Seite schützt die mangelnde Sichtbarkeit von Instanzvariablen ja nicht vor dem Zugriff auf Objekte, auf die sie verweisen (wegen des möglichen *Aliasing*), so dass man sich an dieser Stelle nicht versteigen sollte.

Um den Wert von a zu setzen, definiert man üblicherweise die folgende, auch **Setter** genannte Methode:

```
117 a: einWert
118       "setze den Wert von a auf einWert"
119    a := einWert
```

Getter und Setter werden zusammen auch als **Zugriffsmethoden** (oder **Accessoren**; engl. accessor methods) bezeichnet.

Die obigen Zugriffsmethoden werden aufgerufen, indem man dem Objekt, zu dem die Instanzvariable a gehört, die Nachricht `a` (zum Lesen) bzw. `a:` mit einem Objekt als Parameter (zum Schreiben) schickt. Das Empfängerobjekt antwortet darauf im ersten Fall mit Rückgabe von a und im zweiten Fall mit Rückgabe von sich selbst. Man beachte, dass bei der Rückgabe keine Kopie, sondern der Inhalt der Variable (bzw. ein Verweis darauf) zurückgegeben wird. Nach Auswertung des Ausdrucks

```
120 b := einObjekt a
```

verweisen also `b` und die Instanzvariable `a` von `einObjekt` (genauer: dem von `einObjekt` bezeichneten Objekt) auf dasselbe Objekt, genauso wie nach Auswertung von

```
121 einObjekt a: b
```

Bei beiden ist hinterher `b` ein Alias auf `a`, was de facto die Kapselung des Zustandes von `einObjekt` durchbricht.

Es soll nochmals darauf hingewiesen werden, dass im ersten Ausdruck `a` eine Nachricht darstellt und keinesfalls der Name der Instanzvariable des Objekts ist. Insbesondere handelt es sich bei dem (Teil-)Ausdruck `einObjekt  a` mitnichten um das Äquivalent zu dem aus JAVA bekannten `einObjekt.a`, sondern vielmehr dem von `einObjekt.getA()` (wobei man die Methode `getA()` in JAVA per Konvention natürlich genauso gut nur `a()` nennen könnte). Ein direkter Zugriff auf die Instanzvariable von außen wie in JAVA ist in SMALLTALK nicht möglich. Man kann also den Zugriff auf eine Instanzvariable verhindern, indem man einfach keine Zugriffsmethoden dafür vorsieht; man kann ihn auf *Nur-Lesen* oder *Nur-Schreiben* beschränken, indem man nur die jeweilige Zugriffsmethode zur Verfügung stellt.

Zugriffsmethoden für indizierte Instanzvariablen

Man beachte schließlich, dass anders als benannte Instanzvariablen indizierte Instanzvariablen auch von dem Objekt, zu dem sie gehören, nicht direkt, sondern nur über die beiden vordefinierten Nachrichten `at:` und `at:put:` gelesen und geschrieben werden können:

```
122 einObjekt at: i
```

liefert den Wert der Instanzvariable mit dem Index `i`,

```
123 einObjekt at: i put: einAnderesObjekt
```

setzt ihn. Es ist in SMALLTALK also nicht möglich, indizierte Instanzvariablen eines Objekts (im Gegensatz zu benannten) durch Nicht-Deklaration von Zugriffsmethoden zu verbergen (vor Zugriff zu schützen). Zugleich folgt aus der festen Vorgabe der beiden Zugriffsmethoden, dass jedes Objekt nur genau eine (unbenannte) Menge von indizierten Instanzvariablen haben kann.

Repräsentation des Zustands als Implementationsgeheimnis

Die ausschließliche Abfrage und Änderung des Zustandes von Objekten über Methoden hat den Vorteil, dass man sich nicht festlegt, wie man den Zustand eines Objekts tatsächlich codiert. So kann man beispielsweise einem Punktobjekt Methoden zum Setzen und Abfragen sowohl von kartesischen als auch von polaren Koordinaten zuordnen, muss aber nur Instanzvariablen für eine Art von Koordinaten vorsehen und kann die anderen jeweils berechnen:

```
124 kartesischX: a y: b
125       "setzt kartesische Werte“
126    x := a.
127    y := b

128 x
129       "liefert die x-Koordinate"
130    ^ x

131 y
132       "liefert die y-Koordinate"
133    ^ y

134 polarRadius: r winkel: a
135       "setzt polare Werte"
136    x := a cos * r.
137    y := a sin * r

138 radius
139       "liefert die Radius-Koordinate"
140    ^ (x * x + (y * y)) sqrt

141 winkel
142       "liefert die Winkel-Koordinate"
143    ^ (x / self radius) arcCos
```

Implementationsgeheimnis und Interface

Man könnte die Koordinaten natürlich genauso gut in polarer Form speichern und die kartesischen berechnen — für den Benutzer dieser Objekte spielt das keine Rolle. Man betrachtet die Art und Weise, wie ein Objekt seinen Zustand codiert, als sein *Implementationsgeheimnis* und die Menge der Methodensignaturen, die den Zugriff auf das Objekt (seinen Zustand) erlauben (das *Protokoll*), als sein *Interface*. Mehr dazu in Abschnitt 4.3.8.

4.3.5 Operationen auf zustandslosen (unveränderlichen) Objekten

Auf unveränderlichen Objekten ausgeführte Methoden oder Operationen können deren Zustand naturgemäß nicht ändern. So ändert beispielsweise die Addition von Zahlen nichts an ihren Operanden, sondern liefert als Ergebnis ein anderes Objekt. Derartige Methoden (oder

Operationen) sind also klassische Funktionen: Sie berechnen anhand eines oder mehrerer Argumente (wovon eines das Empfängerobjekt ist) ein Ergebnis, und das ohne *Seiteneffekte* wie die Zustandsänderungen der Argumente.

Bei anderen Objekten wie z. B. den Punktobjekten des Abschnitts 4.3.4 ist es hingegen fraglich, ob Operationen wie die Addition den Zustand des Empfängers verändern oder ein neues Objekt zurückgeben sollen: Bei einer Veränderung des Empfängers würde die Addition dann eher einer Translation gleichkommen (weswegen die entsprechende Methode auch so genannt werden sollte). Um jedoch zu zeigen, wie man neue Objekte erzeugen und zurückgeben kann, fehlt uns hier noch einiges; wir kommen erst in Abschnitt 7.3 darauf zurück.

4.3.6 Konstante Methoden

In SMALLTALK gibt es keine frei bezeichenbaren *Konstanten*[24], sondern nur *Literale* (s. Abschnitt 1.2). Da verschiedene Vorkommen gleicher Literale aber (außer bei Symbolen) verschiedene Objekte erzeugen, sind Literale nicht für alle Zwecke ausreichend. Es gibt dafür einen Trick, mit dem man dasselbe Literal mehrfach verwenden kann, nämlich durch eine sog. **konstante Methode**.

```
144 pi
145     "na ja, so ungefähr …"
146   ^ 3.142
```

ist eine solche konstante Methode. An ihr ist nichts weiter konstant, als dass sie immer dasselbe Objekt zurückgibt. Der Trick besteht nämlich darin, dass das zum Literal `3.142` gehörende Zahlobjekt nur einmal, nämlich zur Übersetzungszeit der Methode, erzeugt wird und die Ausführung der Methode immer auf dieses und damit dasselbe Objekt zurückgreift.

Probleme konstanter Methoden

Nun gibt es leider zwei Probleme. Das erste ist offensichtlich, dass bei einer erneuten Übersetzung der Methode auch ein neues Objekt erzeugt wird, das dann mit früher zurückgegebenen nicht mehr identisch ist (von den bereits bekannten Ausnahmen abgesehen). Das ist immer dann ein Problem, wenn das früher zurückgegebene Objekt in Variablen gespeichert wurde und nun mit dem neuen auf Identität verglichen werden soll.[25] Das sollte man also tunlichst unterlassen.

Das zweite Problem ist noch etwas subtiler: Bei der konstanten Methode

```
147 einsZweiDrei
```

[24] Für gewöhnlich sind Konstanten Namen für (unveränderliche Werte); man kann Sie also als Variablen auffassen, denen nur einmal (bei der Initialisierung) ein Wert zugewiesen wird. Die Definition von Konstanten erfordert dann aber ein Schlüsselwort, um sie von anderen Variablen zu unterscheiden.

[25] Man bedenke, dass in SMALLTALK das (neu) Kompilieren einer Methode nicht bedeutet, dass das Programm danach neu gestartet werden muss — das Programm läuft vielmehr weiter und alle Objekte mit ihren Variablenbelegungen bleiben erhalten.

```
148   ^ #(1 2 3)
```

eines Objekts o und nach Auswertung der Anweisungen

```
149 a := o einsZweiDrei
150 a at: 1 put: 0
151 b := o einsZweiDrei
```

enthält b an erster Stelle ebenfalls eine 0! Die vermeintlich konstante Methode ist also alles andere als konstant!! Interessanterweise ist das Ergebnis dieses Experiments genau konvers zum ersten Problem: Die Identität der von der konstanten Methode zurückgegebenen Objekte bleibt erhalten, ihre Erscheinung ändert sich aber (beim ersten Problem blieb die Erscheinung gleich, aber die Identität änderte sich).

Zumindest das zweite Problem lässt sich eindämmen, indem man auf Sprachebene durch Literale erzeugte Objekte als unveränderlich annimmt und Änderungen der Art von Zeile 150 entsprechend verbietet. Dies machen einige SMALLTALK-Dialekte auch tatsächlich.[26] Die eigentliche Erkenntnis ist aber, dass die Referenzsemantik von Variablen und das damit verbundene Aliasing von Objekten zu höchst subtilen Problemen führen kann, derer man sich immer gewahr sein sollte.

4.3.7 Primitive Methoden

Zwar ist SMALLTALK über weite Teile in sich selbst definiert (was sich darin äußert, dass praktisch die gesamte Sprachdefinition in Form von Methoden hinterlegt und damit für den Programmierer nicht nur sichtbar, sondern auch änderbar ist), aber für einige primitive Operationen greift es doch auf native Implementierungen zurück. Dazu zählt z.B. die Addition (+) für kleine Integer oder auch der Zugriff auf indizierte Variablen mittels `at:` und `at:put:`. Die entsprechenden Methoden sind in SQUEAK wie folgt implementiert:

```
152 + aNumber
153   <primitive: 1>
154   ^ super + aNumber

155 at: index
156   <primitive: 60>
157   index isInteger ifTrue:
158     [self class isVariable
159       ifTrue: [self errorSubscriptBounds: index]
160       ifFalse: [self errorNotIndexable]].
161   index isNumber
162     ifTrue: [^self at: index asInteger]
163     ifFalse: [self errorNonIntegerIndex]

164 at: index put: value
165   <primitive: 61>
```

[26] In JAVA sind Array-Literale übrigens nur in New-Ausdrücken erlaubt; die dadurch beschriebenen Array-Objekte werden erst zur Laufzeit und dann immer wieder neu erzeugt, so dass sich keine Aliase ergeben.

```
166   index isInteger ifTrue:
167     [self class isVariable
168       ifTrue: [(index >= 1 and: [index <= self size])
169            ifTrue: [self errorImproperStore]
170            ifFalse: [self errorSubscriptBounds: index]]
171       ifFalse: [self errorNotIndexable]].
172   index isNumber
173     ifTrue: [^self at: index asInteger put: value]
174     ifFalse: [self errorNonIntegerIndex]
```

Dabei stehen die in spitzen Klammern stehenden Ausdrücke jeweils für Aufrufe von primitiven Methoden, die, da man sie als Programmierer nicht selbst verwenden soll, nur durchnummeriert wurden. Die Anweisungen nach den Aufrufen der primitiven Methoden werden nur ausgeführt, wenn die primitive Methode nicht erfolgreich war. Das bedingt, dass aus einer primitiven Methode mittels Return direkt zurückgesprungen werden kann, und zwar dorthin, wo die Methode `+`, `at:` bzw. `at:put:` aufgerufen wurde. Dieses Verhalten, das einigermaßen ungewöhnlich erscheint, wird uns im Kontext von Blöcken (Abschnitt 4.4) wieder begegnen.

4.3.8 Protokoll

In SMALLTALK ist das **Interface** eines Objekts die Menge der Nachrichten, die es versteht. Dieses Interface wird in Form der sog. **Protokollbeschreibung** o. kurz des **Protokolls** spezifiziert, die aus den *Methodensignaturen* und den *Kommentaren* der zu den Nachrichten passenden Methoden besteht und der die **Implementation**, also die Liste der *Methodenrümpfe*, gegenübersteht. Letztere sind, zusammen mit den Instanzvariablen, das **Implementationsgeheimnis** eines Objekts, das hinter seiner Protokollbeschreibung (dem Interface) verborgen wird. Die Kommentare dürfen übrigens, wie in den meisten anderen Sprachen auch, als (schwacher) Ersatz für eine formale Spezifikation des Verhaltens, das in einer Methode implementiert wird, angesehen werden (vgl. dazu Abschnitt 52.6 in Lektion 5).

Einteilung in Nachrichtenkategorien

Um die Programmierung zu erleichtern, wird in den meisten SMALLTALK-Systemen das Protokoll von Objekten in sog. **Nachrichtenkategorien** eingeteilt, die jeweils einen Namen tragen, der die in der Kategorie enthaltenen Namen zusammenfasst. Da jede Methode in genau eine Nachrichtenkategorie fallen muss, stellen diese eine Partitionierung des Interfaces eines Objekts dar. Unter den Kategorien sind solche, die das Wort „private“ enthalten; deren Methoden sollten dann nicht „von außerhalb“, also nur vom Objekt selbst (über `self`) aufgerufen werden. Dies wird jedoch nicht vom Compiler erzwungen. Nachrichtenkategorien haben auch sonst keinerlei die Programmausführung betreffende Bedeutung, sondern dienen lediglich der besseren Lesbarkeit.

Wie Sie in der nächsten Lektion lernen werden, werden Protokolle in SMALLTALK nicht auf Objektebene, sondern auf Klassenebene spezifiziert. In STRONGTALK, einer Erweiterung von SMALLTALK um ein (optionales) Typsystem, werden Protokolle dann zu Typen erhoben (s. Lektion 3, Kapitel 20). Da Protokolle nur Methoden enthalten, sind sie den Interfaces JAVAs sehr ähnlich. Tatsächlich werden Protokolle in STRONGTALK auch manchmal Interfaces genannt.

4.4 Blöcke

Wir kommen nun zu einer der wichtigsten Ausprägungen von SMALLTALKs Alles-ist-ein-Objekt-Motto: den **Blöcken**. Genau wie eine Methode ist ein Block eine abgegrenzte Sequenz, oder Folge, von Anweisungen. Anders als eine Methode ist ein Block jedoch nicht benannt; er kann aber benannt werden, indem er einer Variable zugewiesen wird.

Definition von Blöcken

Um auszudrücken, dass eine Sequenz von Ausdrücken ein Block ist, wird die Sequenz mit eckigen Klammern markiert. So ist

```
175 [ | temp | temp := x. x := y. y := temp ]
```

Kontext eines Blocks

die Definition eines Blocks, der aus der Deklaration der Variable `temp` und drei Zuweisungen besteht. Die Variablen x und y seien dabei außerhalb des Blocks, im **Kontext des Blocks**, deklariert. Dabei ist der Kontext des Blocks die Methode, in der er definiert wurde.[27]

Ausführung von Blockausdrücken

Bei der Ausführung des obigen Blocks wird ein neues Blockobjekt erzeugt. Mittels

```
176 swap := [ | temp | temp := x. x := y. y := temp ]
```

wird der Block einer Variable swap zugewiesen. Die Anweisungen, die den Block ausmachen, werden dabei *nicht* ausgeführt, selbst dann nicht, wenn der Block (wie in Zeile 175) isoliert steht und ausgeführt wird (das dabei erzeugte Objekt bleibt namenlos und wird von der Speicherbereinigung wieder entfernt).

Auswertung von Blöcken

Um die Anweisungen, die einen Block ausmachen, zur Ausführung zu bringen, muss man ihn auswerten. Dazu schickt man ihm die Nachricht `value`. Der Ausdruck

```
177 swap value
```

bewirkt, dass die Variablen x und y aus dem Kontext des Blocks ihren Wert tauschen. Das Objekt, zu dem `swap value` auswertet, ist das Objekt, zu dem die letzte Anweisung auswertet (s. Abschnitt 4.2; im obigen Beispiel also der Inhalt von `temp`, der derselbe ist wie der von x aus dem Kontext).

Wert eines Blocks

Rückgabewert der Methode `value` ist zunächst immer das Objekt, zu dem der letzte Ausdruck eines Blocks auswertet, im obigen Beispiel das durch `temp` benannte Objekt.

[27] In einem Workspace definierte und mittels `doIt` usw. ausgeführte Blöcke werden zu diesem Zweck in (temporäre) Methoden übersetzt, die keinem Objekt (oder `nil`) zugeordnet sind.

4.4.1 Home context und Closure

Da Blöcke Objekte sind, die Variablen zugwiesen werden können, können sie auch an andere Methoden übergeben werden. Werden sie dort (mittels `value`) ausgewertet, dann findet die Auswertung in einem anderen Kontext statt. In diesem sind die „freien" Variablen des Blocks (also die, die nicht selbst als lokale Variablen deklariert wurden; x und y in Zeile 175) aber gar nicht zugreifbar. Der Block nimmt deswegen seinen Kontext mit (oder, richtiger, der Kontext ist im Block miteingeschlossen). Der Kontext, in dem ein Block definiert wurde (in dem das ihn repräsentierende Objekt erzeugt wurde), nennt man seinen **Home context**. Die Auswertung eines Blocks erfolgt stets in seinem Home context, insbesondere auch dann, wenn ihm `value` in einem anderen Kontext gesendet wurde.

Das folgende Beispiel zweier Methoden soll den Sachverhalt erläutern:

```
178 homeContext
179   | x |
180   x := 2.
181   self otherContext: [Transcript show: x printString]

182 otherContext: x
183   x value
```

gibt auf dem Transcript 2 aus, obwohl im Kontext von `otherContext:` die Variable `x` einen Block und nicht 2 zum Wert hat. Die (Werte der) *Pseudovariablen* `self` und `super` zählen übrigens auch zum Home context; dies ist vor allem im Zusammenhang mit dem dynamischen Binden (Kapitel 12 in Lektion 2) interessant.

Dass ein Block aus seinem Home context herausgelöst und in einem anderen gespeichert werden kann beinhaltet das Problem, dass die lokale Variablen des Home contexts schon verschwunden sein können, wenn der Block ausgewertet wird. Die durch den Block „eingefangenen" lokalen Variablen (inkl. formale Parameter) müssen daher unabhängig von der Ausführung der Methoden, die sie definieren, weiterleben. Die Umsetzung von Blöcken durch den SMALLTALK-Compiler ist alles andere als trivial und verschiedene SMALLTALK-Systeme unterscheiden sich darin zum Teil erheblich voneinander, was sich (leider) auch in unterschiedlichem Verhalten äußert.

Blöcke in anderen Sprachen

Die Blöcke SMALLTALKs heißen in anderen Sprachen übrigens (lexikalische) **Closures**; sie werden für die sog. *Lambda-Ausdrücke*, also (anonyme) Funktionen, die selbst Objekte oder Werte sind und die deswegen aus ihrem Kontext herausgelöst und in andere verschoben werden können, gebraucht. Dabei unterscheiden sich die Sprachen zum Teil erheblich darin, was alles in eine Closure einbezogen werden kann; so können die lokalen Namen (Variablen) beispielsweise auf Konstanten eingeschränkt werden, um zu vermeiden, dass *temporäre Variablen* weiterleben müssen, weil sie in einer Closure enthalten sind.

4.4.2 Continuation

Das Konzept des Home context eines Blocks geht aber noch weiter: Es gehören nicht nur die sichtbaren Variablen aus dem Kontext der Definition des Blocks dazu, sondern auch der sog. *Call stack*, also der Speicher, in dem die Rücksprungadressen von Methodenaufrufen abgelegt werden. Entsprechend bewirkt eine explizite *Return-Anweisung* innerhalb eines Blocks bei dessen Auswertung auch stets die sofortige Rückkehr der Methode, in der der Block definiert wurde und nicht etwa der Methode, in welcher der Block (durch Senden von `value`) ausgewertet wird. Das nachfolgende Beispiel zeigt das:

```
184 homeContext
185   self otherContext: [^ Transcript show: 'home']

186 otherContext: x
187   x value.
188   Transcript show: 'other'
```

Der Aufruf der Methode `homeContext` gibt „home“, aber nicht auch noch „other“ auf der Konsole aus. Man nennt dieses Konzept, das ebenfalls aus der Welt der *funktionalen Programmierung* stammt, auch **Continuation**. Continuations spielen bei der Implementierung von Kontrollstrukturen in SMALLTALK eine entscheidende Rolle (s. Kapitel 4.6).

Das Prinzip der Continuation gilt übrigens auch für geschachtelte Blöcke:

```
189 [[^ true] value. [^ false] value] value
```

liefert `true`, nicht `false`. Return-Anweisung verlässt auch alle umschließenden Blöcke, und zwar sofort. Dies liegt daran, dass explizite Return-Anweisungen aus Blöcken immer dahin zurückkehren, von wo die Definition des Blocks angestoßen wurde.

Laufzeitfehler bei Continuations

Continuations können zu Laufzeitfehlern führen, nämlich wenn durch sie von einer Methode zurückgekehrt werden soll, die bereits beendet wurde. Wenn beispielsweise das Objekt o über die Methode

```
190 merkwuerdig
191   ^ [^ self]
```

verfügt, dann führt

```
192 o merkwuerdig value
```

zu einem Laufzeitfehler, da die Auswertung des von `merkwuerdig` mit ihrer Beendigung zurückgegebenen Blocks `merkwuerdig` noch einmal beenden müsste, was aber nicht geht. Return-Anweisungen in Blöcken sind u. a. deswegen ein umstrittenes Konzept. Für SMALLTALK sind sie aber unverzichtbar, da mit ihnen fast alle *Kontrollstrukturen* implementiert werden (s. Kapitel 4.6).

4.4.3 Parametrisierte Blöcke

Die Blöcke von SMALLTALK können auch als anonyme Funktionen aufgefasst werden, wobei alle bisherigen parameterlos waren: Der Bezug zu ihrer Umwelt wurde ausschließlich über den Home context hergestellt. Es ist aber auch möglich, Blöcke mit Parametern zu versehen, die bei ihrer Auswertung an Objekte aus dem Kontext der Auswertung gebunden werden können. Den obigen Swap-Block könnte man also auch wie folgt definieren wollen:

```
193 swap := [ :x :y | | temp | temp := x. x := y. y := temp]
```

wobei die jeweils von einem Doppelpunkt eingeleiteten und vom Rest des Blocks durch einen senkrechten Strich abgetrennten Variablen x und y die *formalen Parameter* des Blocks sind und das — analog zu einer Methode — in Strichen eingeschlossene `temp` eine lokale Variable ist. Der Variable `swap` wie oben zugewiesen kann der Block mittels

```
194 swap value: a value: b
```

ausgewertet werden, wenn `a` und `b` Variablen aus dem Kontext der Auswertung sind.

Selbsttestaufgabe 4.2

(a) Versuchen Sie, durch Überlegen oder Ausprobieren herauszufinden, ob die Auswertung des Blocks `swap` aus Zeile 193 seinen (mutmaßlichen) Zweck erfüllt, also beispielsweise in Zeile 194 den Wert der Variablen `a` und `b` tauscht.

(b) Wie sieht das beim Parameterlosen `swap` aus Zeile 176 (in einem geeigneten Kontext) aus?

(c) Begründen Sie, warum es sinnvoll ist, die explizite Zuweisung an die formalen Parameter eines Blocks (im obigen Beispiel `x` und `y`) innerhalb des Blocks zu verbieten.

4.5 Kontrollstrukturen

Kontrollstrukturen regeln den Ablauf des Programms, also die Reihenfolge der Schritte, aus denen seine Ausführung besteht. Anders als in anderen Programmiersprachen gibt es in SMALLTALK nur zwei Kontrollstrukturen, nämlich die Sequenz und den dynamisch gebundenen Methodenaufruf; alle anderen, inklusive der Verzweigung und der Wiederholung (Schleife), müssen durch diese simuliert werden. Dies ist möglich, weil SMALLTALK Blöcke hat und weil in SMALLTALK (so wie in allen anderen objektorientierten Programmiersprachen) der Methodenaufruf variabler ausfällt als der gewöhnliche Prozedur- oder Funktionsaufruf, wie Sie ihn vielleicht von Sprachen wie PASCAL oder C her kennen: Er enthält, wie bereits in Abschnitt 4.3.2 angedeutet, eine *versteckte Fallunterscheidung* in Form des *dynamischen Bindens*.

4.5.1 Sequenz

Die Sequenz als Kontrollstruktur besagt lediglich, dass textuell aufeinanderfolgende Anweisungen eines Programms (einer Methode) auch zeitlich nacheinander ausgeführt werden. Die zeitliche Sequenz aufeinander folgender Anweisungen kann lediglich durch andere Kontrollstrukturen (in SMALLTALK nur durch den Methodenaufruf; s. u.) unterbrochen werden. Dies gilt auch für *parallele Ausführung*, die man sich wie die gleichzeitige Abarbeitung zweier sequentieller Programme auf denselben Objekten vorstellen kann (s. Kapitel 16).

4.5.2 Dynamisch gebundener Methodenaufruf

Wie bereits in Abschnitt 4.3.2 erläutert, verbirgt sich hinter dem Nachrichtenversand in SMALLTALK der Methodenaufruf. Wann immer ein Objekt eine Nachricht an ein Empfängerobjekt verschickt, wechselt der Kontrollfluss damit zum Empfängerobjekt, genauer zu der Methode des Empfängerobjekts, das zur Reaktion auf die Nachricht vorgesehen ist. Nach der Abarbeitung der Methode kehrt der Kontrollfluss an das sendende Objekt (genauer: zu der Methode, aus der die Nachricht versandt wurde) zurück und setzt seine Arbeit dort fort. Bei der Rückkehr wird auch das Ergebnis der Methode, (eine Referenz auf) ein Objekt, geliefert, das dann an der Stelle des Nachrichtenausdrucks, der den Methodenaufruf bewirkt hat, eingesetzt wird. Der genaue Mechanismus des dynamischen Bindens in SMALLTALK wird in Kapitel 12 von Lektion 2 untersucht.

4.6 Abgeleitete Kontrollstrukturen

Wenn Sie schon in anderen Programmiersprachen wie z. B. PASCAL, C oder JAVA programmiert haben, dann kennen Sie sicher *Schlüsselwörter* wie `if`, `else`, `for` und `while`. Diese Schlüsselwörter stehen für Kontrollstrukturen, feste Bestandteile der Sprache, die für die Steuerung des Ablaufs eines Programms durch den Programmierer vorgesehen sind. Man hat sich irgendwann einmal (im Zuge der Diskussion zur sog. *strukturierten Programmierung*) darauf festgelegt, dass jede Programmiersprache über die Kontrollstrukturen Sequenz, Verzweigung, Wiederholung (*Iteration*) und Aufruf verfügen sollte. Während die einfache Sequenz von Anweisungen während der Ausführung durch die lineare Folge der Anweisungen im Programmtext vorgegeben ist, sind für alle Abweichungen vom linearen Kontrollfluss, also für Verzweigung, Wiederholung und Aufruf, spezielle Flusssteuerungskonstrukte vorgesehen. Das *Goto* gehört übrigens nicht dazu; es gilt seit dem Aufkommen der strukturierten Programmierung als verpönt.

SMALLTALKs zwei Kontrollstrukturen

In SMALLTALK hat man die durch die Syntax der Sprache vorgesehenen Kontrollstrukturen auf die Sequenz und den Aufruf, letzteres ausgedrückt durch das Versenden einer Nachricht an ein Objekt, beschränkt. Alle anderen Kontrollstrukturen müssen mit den Mitteln der Sprache simuliert werden. Was zunächst wie eine erhebliche Einschränkung aussehen mag, erweist sich in der Praxis als gewichtiger Vorteil: Der Programmierer kann nämlich selbst, wenn ihm danach ist, neue Kontrollstrukturen einführen.

4.6.1 Verzweigung

Die einfache Verzweigung wird in SMALLTALK durch das Versenden eine Nachricht, die die bedingt auszuführenden Anweisungen in Form eines Blocks als Parameter enthält, an einen Wahrheitswert realisiert:

```
195 x > 5 ifTrue: [Transcript show: "x > 5"]
```

beispielsweise gibt genau dann die Meldung „x > 5“ auf der Konsole aus, wenn `x > 5` zu `true` auswertet.

Um zu verstehen, wie das funktioniert, sehen wir uns zunächst die Implementierung der Verzweigung mittels `ifTrue:` an. Sie wird durch eine Methode `ifTrue: aBlock` realisiert, die als Parameter `aBlock`, also eine Folge von Anweisungen, erhält, das sie, der Bedeutung von If entsprechend, entweder ausführt oder eben nicht ausführt. Diese Methode ist für die beiden Objekte `true` und `false` (genauer: für die beiden Objekte, die die Pseudovariablen `true` und `false` benennen; siehe Abschnitt 1.7) jeweils unterschiedlich implementiert:

```
196 ifTrue: aBlock
197    ^ aBlock value
```

für `true` und

```
198 ifTrue: aBlock
199    ^ nil
```

für `false`. Wenn also der Empfänger der Nachricht `ifTrue: aBlock` das Objekt `true` ist, dann bewirkt die Auswertung, dass der Block `aBlock` ausgeführt wird; ist das Objekt hingegen `false`, wird `aBlock` nicht ausgeführt, was genau der Bedeutung der Nachricht entspricht. So hat

```
200 1 = 1 ifTrue: [Transcript show: 'Eins bleibt Eins!!!'; cr]
```

genau den erwarteten Effekt. Aufgrund der *Continuation* bei Blöcken (s. Abschnitt 4.4.2) ist es möglich, aus einer Methode mittels `ifTrue:` bedingt zurückzukehren:

```
201 returnEarly
202    1 = 1 ifTrue: [^ true].
203    ^ false
```

liefert `true` zurück, da durch die Auswertung des Blocks `[^ true]` per `value` in Zeile 197 nicht der Block, sondern die Methode `returnEarly`, beendet wird. Wie man sieht, hat das zunächst etwas eigenwillig anmutende Konzept der Continuation in Kombination mit Return eine erhebliche programmierpraktische Relevanz.

Aus Symmetriegründen wird auch die Methode `ifFalse: aBlock` angeboten, die wie folgt implementiert ist:

```
204 ifFalse: aBlock
205   ^ nil
```

für `true` sowie

```
206 ifFalse: aBlock
207   ^ aBlock value
```

für `false`. Eigentlich viel zu banal, um es hinzuschreiben, aber da es nur ein einziges Mal gemacht werden muss und dabei so etwas Fundamentales wie die Verzweigung realisiert, kann man sich auch daran freuen.

Fallunterscheidung durch dynamisches Binden

Falls Sie dies dennoch für einen Budenzauber halten, dann haben Sie zumindest nicht völlig unrecht: Die Verzweigung ist nämlich gar nicht wirklich aus dem Sprachkern verschwunden, sie ist nur an einer Stelle versteckt, an der Sie sie vielleicht nicht vermuten: an der Stelle der Auswahl der Methode, die in Reaktion auf den Empfang einer Nachricht ausgeführt wird (also beim *dynamischen Binden*).

Selbsttestaufgabe 4.3

Überlegen Sie, wie Sie das aus anderen Sprachen bekannte If-then-else-Konstrukt in SMALLTALK realisieren würden und schreiben Sie die entsprechenden Methodendefinitionen auf.

Nach demselben Prinzip wie die einfache Fallunterscheidung vermeidet SMALLTALK übrigens auch solche, die das Auftreten von `nil` betreffen. So ist zum Beispiel die Methode `isNil` für `nil` als

```
208 isNil
209   ^ true
```

und für alle anderen Objekte als

```
210 isNil
211   ^ false
```

implementiert.

Selbsttestaufgabe 4.4

Überlegen Sie, wie Sie die logischen Operatoren `and:`, `or:` und `not` für `true` und `false` implementieren würden!

4.6.2 Wiederholung

Etwas weiter ausholen müssen wir für die Implementierung von Wiederholungen (Schleifen): Da das Abbruchkriterium von Schleifen immer wieder (bei jedem Schleifendurchlauf) ausgewertet werden muss, kann nicht einfach einmal eine Nachricht an (eine Variable mit Inhalt) `true` oder `false` gesendet werden. Vielmehr muss die Auswertung des Abbruchkriteriums selbst in einem Block stattfinden, der bei jedem Schleifendurchlauf neuerlich ausgewertet wird. Aber auch das ist kein Problem: Der Nachrichtenempfänger ist einfach ein Block, dessen Auswertung entweder `true` oder `false` zurückliefert; der Parameter der Nachricht ist dann der Block, der den Schleifenrumpf darstellt, wie in:

```
212 [ x < 5 ] whileTrue: [ x := x + 1 ]
```

`whileTrue:` ist dazu als Methode für Blöcke wie folgt implementiert (beachten Sie, dass Sie den Empfänger, einen Block, in der Methodendefinition nicht direkt sehen; er wird durch `self` repräsentiert und ist nicht mit dem Parameter `aBlock`, ebenfalls ein Block, zu verwechseln):

```
213 whileTrue: aBlock
214   self value
215     ifTrue: [
216       aBlock value.
217       self whileTrue: aBlock].
218   ^ nil
```

Simulation der Schleife durch Endrekursion

Wie man sieht, wird hier die Schleife durch eine sog. *Endrekursion* simuliert: `whileTrue:` ruft `whileTrue:` am Ende selbst wieder auf. Wegen der Performanz (oder möglicher Beschränkungen der Anzahl der Schleifendurchläufe durch die Größe des Aufrufstacks) braucht man sich dabei keine Sorgen zu machen: Da hinter dem rekursiven Aufruf nichts mehr passiert (deswegen ja Endrekursion), kann dieser vom Compiler in eine echte Schleife übersetzt werden. Für die Implementierung von `whileFalse:` (mit entsprechender Semantik) braucht übrigens nur das `ifTrue:` aus Zeile 215 durch `ifFalse:` ersetzt und der rekursive Aufruf entsprechend angepasst zu werden.

Für die ebenfalls aus anderen Sprachen bekannten For-Schleifen hat SMALLTALK eine andere elegante Lösung parat, auf die wir im nächsten Abschnitt eingehen werden.

4.6.3 Iteration

Wenn Sie If und While schon kennen, kennen Sie sicher auch For. Die klassische Form der For-Schleife verwendet eine Zählvariable, einen Anfangswert, ein Inkrement (das auch negativ, also ein Dekrement sein kann) sowie einen Endwert. Solche For-Schleifen gibt es in SMALLTALK auch:

```
219 5 to: 1 by: -2 do: [ :i | Transcript show: i printString]
```

beispielsweise gibt auf dem Transcript die Folge „531“ aus.

Zählschleife

Wir schauen uns den Ausdruck aus Zeile 219 einmal genauer an. Dem Objekt 5 wird offenbar eine Nachricht `to:by:do:` gesendet, wobei 5 der Startwert, der Parameter zu `to:`, 1, der Endwert, der zu `by:`, −2, das Inkrement und der zu `do:` ein Block ist. Der Block stellt offenbar, ähnlich wie bei der Realisierung der While-Schleife in SMALLTALK, den Schleifenrumpf dar; er hat einen Parameter `i`, der anscheinend als Zählvariable fungiert. Tatsächlich wird die Methode `to:by:do:` in SMALLTALK EXPRESS wie folgt implementiert:

```
220 to: stop by: step do: aBlock
221    | nextValue |
222    nextValue := self.
223    step = 0 ifTrue: [self error: 'step must be non-zero'].
224    step < 0
225       ifTrue: [[stop <= nextValue]
226             whileTrue:
227                [aBlock value: nextValue.
228                nextValue := nextValue + step]]
229       ifFalse: [[stop >= nextValue]
230             whileTrue:
231                [aBlock value: nextValue.
232                nextValue := nextValue + step]]
```

Hier interessiert uns aber vor allem eine Form der Iteration, die nicht einer einfachen Zählschleife entspricht, sondern über eine Menge von beliebigen Objekten geht. Solche Mengen sind uns ja schon begegnet, wenn auch nur in Gestalt von literalen Arrays.

For-each-Schleife

Anders als in vielen anderen Sprachen kann man in SMALLTALK über die Elemente eines Arrays direkt, also ohne die Verwendung einer Zählschleife, deren Laufvariable als Index in das Array dient, iterieren. So hat die Auswertung des Ausdrucks

```
233 #(5 3 1) do: [ :i | Transcript show: i printString]
```

exakt das gleiche Ergebnis wie die des Ausdrucks aus Zeile 219, nämlich die Ausgabe von „531“ auf dem Transcript. `i` ist aber diesmal keine Zählvariable, da hier nichts gezählt wird; es ist vielmehr eine Laufvariable, der der Reihe nach die Elemente des literalen Arrays `#(5 3 1)` zugewiesen werden. Dies müssen keine Zahlen sein:

```
234 #('Smalltalk' 'ist' 1.0 'klasse')
235    do: [ :i | Transcript show: i printString]
```

interne Iteration

funktioniert genauso. `do:` ersetzt also ganz offensichtlich das aus manchen anderen Sprachen (seit der Version 5.0 auch aus JAVA) bekannte For-each-Konstrukt. Wie wir gleich sehen werden, ist die Iteration, also das Fortschalten der Elemente und die Überprüfung der Abbruchbedingung, in der Collection, über die iteriert wird, implementiert, weswegen man das Verfahren auch **interne Iteration** nennt (in Abgrenzung von der herkömmlichen, **externen Iteration**, bei der die Laufvariable selbst gesetzt und abgefragt werden muss).

WIKIPEDIA

Implementierung der internen Iteration am Beispiel von `do:`

Die Implementierung der Kontrollstruktur erfolgt wiederum selbst in SMALLTALK und ist ziemlich einfach:

```
236 do: aBlock
```

```
237   1 to: self size do:
238      [:index | aBlock value: (self at: index)]
```

Dabei ist `to:do:` für Ganzahlen analog zu obigem `to:by:do` implementiert. Die Zählvariable `index` des Blocks von Zeile 238 läuft so von 1 bis zur Anzahl der indizierten Instanzvariablen des Empfängers von `do:` (im obigen Beispiel ein Array), die über den Aufruf von `size` auf dem Empfänger (repräsentiert durch `self`) abgefragt wird. Der Inhalt der indizierten Instanzvariablen des Empfängers wird dann der Reihe nach als Parameter mittels `value:` an den Block `aBlock` zur Auswertung geschickt.

4.6.4 Iterieren über :*n*-Beziehungen

Bei :1-Beziehungen schickt man häufig dem von der betreffenden Variable referenzierten Objekt eine Nachricht. Der Ausdruck

```
239 freund einladen
```

beispielsweise besagt, dass das Objekt, mit dem der Besitzer der Variable `freund` über diese Variable in :1-Beziehung steht, die Nachricht `einladen` erhalten soll. Wenn man dasselbe mit :*n*-Beziehungen machen möchte, so erreicht die Nachricht — bei gleicher Vorgehensweise — nicht die logisch in Beziehung stehenden Objekte, sondern das die Beziehung selbst repräsentierende *Zwischenobjekt* (das ja Wert der Variable ist), das mit dieser Nachricht jedoch nichts anfangen kann. Um die Nachricht stattdessen an alle durch das Zwischenobjekt referenzierten Objekte zu senden, schickt man dem Zwischenobjekt die Nachricht `do: aBlock`, wobei `aBlock` ein mit einem Parameter parametrisierter Block ist, der für jedes Element des Arrays genau einmal (mit dem Element als tatsächlichem Parameter) aufgerufen wird. Wenn also z. B. die Instanzvariable `freunde` heißt und eine :*n*-Beziehung ausdrückt, dann schreibt man statt Zeile 239

```
240 freunde do: [ :freund | freund einladen]
```

Man kann sich fragen, warum die SMALLTALK-Syntax nicht erlaubt, die Nachricht doch direkt an das Zwischenobjekt zu schicken, das die :*n*-Beziehung repräsentiert (also im gegebenen Beispiel `freunde einladen`), und dies dann intern so umsetzt, dass die Nachricht an alle Objekte geschickt wird. Der einfache Grund dafür wird gleich offenbar: Weil man häufig die Nachricht gar nicht an alle Objekte schicken will, sondern nur an ausgewählte, und weil dazu dann noch weitere Angaben notwendig sind, so dass im allgemeinen Fall nichts gewonnen wäre.

Iterationen mit zusätzlicher Funktion

Auf Basis von `do:` lassen sich nun zahlreiche weitere natürliche und äußerst praktische Kontrollstrukturen erzeugen. So ist wie im obigen Beispiel recht häufig eine Nachricht gar nicht an alle Elemente einer :*n*-Beziehung zu senden, sondern nur an solche, die bestimmte Kriterien erfüllen. Dazu ist es möglich, die Beziehung quasi im Vorübergehen einzuschränken und den Block dann nur auf der Einschränkung auszuführen:

```
241 (freunde select: [ :freund | freund eng == true])
              do: [ :freund | freund einladen]
```

Dabei ist die Methode `select:` wie folgt implementiert:

```
242 select: aBlock
243   | answer |
244   answer := self species new.
245   self do: [ :element |
246     (aBlock value: element)
247       ifTrue: [answer add: element]].
248   ^ answer
```

Zeile 244 müssen Sie hier noch nicht verstehen; der Rest sollte Ihnen aber inzwischen klar sein. `answer` ist eine *temporäre Variable*, die nur innerhalb der Methode Gültigkeit hat; ihr werden in der Do-Schleife mittels `add:` (einer Methode mit offensichtlicher Funktion) alle die Elemente des Empfängers hinzugefügt, für die der Parameterblock `aBlock` zu `true` auswertet.

Auf ähnlich einfache Weise lassen sich nahezu beliebig weitere Kontrollstrukturen realisieren. Zu `select:` komplementär ist beispielsweise die Methode `reject:`, die aus einer :*n*-Beziehung alle die Elemente entfernt, die eine genannte Bedingung nicht erfüllen:

```
249 reject: aBlock
250   ^ self select: [ :element |
251     (aBlock value: element) not]
```

Kaum zu glauben, dass mit so wenig Aufwand der Verwendung eine neue Kontrollstruktur hinzugefügt werden kann.

Eine weitere praktische Methode, die eine Sammlung von Objekten zurückgibt, über die dann (mittels `do:`) iteriert werden kann, ist `collect:`; sie sammelt all die Elemente, die die Auswertung des ihr als Parameter übergebenen Blocks auf den Elementen der ursprünglichen Sammlung zurückliefert, und ist wie folgt implementiert:

```
252 collect: aBlock
253   | answer |
254   answer := self species new.
255   self do: [ :element |
256     answer add: (aBlock value: element)].
257   ^ answer
```

Aber auch einzelne Elemente einer Beziehung lassen sich bestimmen: `detect:` mit einem Block `aBlock` als Parameter aufgerufen liefert z. B. aus einer Sammlung von Elementen das erste Element zurück, auf dem `aBlock` ausgewertet den Wahrheitswert `true` ergibt (wobei bei Fehlen eines solchen Elements ein Fehler geliefert wird). Das erlaubt z. B. den Ausdruck

```
258 (freunde detect: [ :freund | freund eng]) treffen
```

zu formulieren, der besagt, dass man sich mit dem ersten engen Freund, und nur mit dem, trifft. Für die Praxis wichtiger ist die Variante `detect: aBlock ifNone: exceptionBlock`, die `detect: aBlock` um einen (parameterlosen) Block ergänzt, dessen Wert bei Fehlen eines geeigneten Elements zurückgegeben wird.

Selbsttestaufgabe 4.5

Versuchen Sie, eine Implementierung der Methode `detect: aBlock` selbst zu entwerfen.

Kumulation

Eine ganze Klasse von immer wiederkehrenden Anweisungssequenzen zu ersetzen erlaubt schließlich die Methode `inject:into:`. Es ergänzt die Funktionsweise von `do:` darum, das Ergebnis der Auswertung eines Blocks in einem Iterationsschritt als ersten Parameter in die Auswertung des nächsten Schritts einzuspeisen. So lässt sich beispielsweise das öde Akkumulieren von Eigenschaften (inkl. der gern vergessenen Initialisierung des Akkumulators) elegant wie folgt ausdrücken:

```
259 freunde
260   inject: true
261   into: [ :einsam :freund | einsam and: [freund eng not]]
```

So einfach kann Programmieren sein!

Für Interessierte: Methoden wie `do:`, `collect:`, `select:` und `inject:` sind allesamt (als Funktionen höherer Ordnung) aus der funktionalen Programmierung bekannt. In die objektorientierte Programmierung mit Sprachen wie JAVA oder C# haben sie jedoch erst sehr spät Einzug gehalten.

5 Zusammenfassung der SMALLTALK-Syntax

Sicher ist Ihnen aufgefallen, dass uns bislang keine *Schlüsselwörter* in SMALLTALK begegnet sind (bis auf die sog. *Schlüsselwortnachrichten*, die aber frei wählbar und deswegen eben gerade keine Schlüsselwörter sind; vgl. Abschnitt 4.1.2). Der Grund hierfür ist einfach: Es gibt keine Schlüssel*wörter*, lediglich ein paar *Symbole mit spezieller Bedeutung*. Es sind dies:

Symbol	Bedeutung/Verwendung
`:=`	Zuweisung
`.`	Trennzeichen zwischen zwei Anweisungen sowie Dezimalpunkt für Gleitkommazahlen
`;`	Trennzeichen zum Kaskadieren von Nachrichten
`:`	Markierung von Parametern in Nachrichten und Blöcken
`( )`	Klammerung von Ausdrücken zur Festschreibung der Reihenfolge der Auswertung
`[ ]`	Bildung von Blöcken
`|`	Trennzeichen zwischen den Parametern eines Blocks und seinen Anweisungen
`| |`	Deklaration von temporären Variablen in Methoden (und Blöcken)
`" "`	Markierung von Kommentaren
`' '`	Markierung von String-Literalen
`$`	Markierung von Zeichenliteralen
`#`	Markierung von Symbol- und Array-Literalen
`^`	Rückgabe-Operator (Return)

Das ist alles! Die reservierten Namen `true`, `false`, `nil`, `self` und `super` sind die von *Pseudovariablen*; alle aus anderen Sprachen bekannten Schlüsselwörter sind als Methoden in SMALLTALK selbst definiert.

6 Lösungen zu den Selbsttestaufgaben

Selbsttestaufgabe 1.1 (Seite 16)

```
262 $a == $a ⇒ true
263 1 == 1 ⇒ true
264 1.0 == 1.0 ⇒ true
        "in PHARO, in anderen Systemen teilweise false"
265 1000000000 == 1000000000  ⇒ true
        "in PHARO, in anderen Systemen teilweise false"
266 #(1 2 3) == #(1 2 3) ⇒ true
        "in PHARO, in anderen Systemen teilweise false"
```

Es fällt auf, dass bei manchen Zahlen (syntaktisch) gleiche Literale identische Objekte repräsentieren, manche nicht. In PHARO ist das übrigens nur anders, weil der Compiler eine Art „*Literaloptimierung*“ betreibt: Zwei gleiche Literale, die gemeinsam übersetzt werden, liefern das identische Objekt.

Selbsttestaufgabe 1.2 (Seite 21)

In PHARO: Ein Objekt, welches offenbar das SMALLTALK-Image repräsentiert. Interessant ist dessen Instanzvariable `globals`, ein Verzeichnis, in dem die globalen Variablen des Systems eingetragen sind, so u. a. `Transcript` und `Smalltalk` selbst. Zugleich enthält es auch einen Eintrag für jede Klasse (so z. B. `Object`).

Selbsttestaufgabe 1.3 (Seite 22)

Das ist im Wesentlichen dieselbe Aufgabe wie Selbsttestaufgabe 1.1, also auch dieselbe Lösung.

Dies erklärt auf einfache Weise, warum der Test auf Identität (==) bei gleichen Literalen für verschiedene Arten von Objekten zu verschiedenen Ergebnissen führt: Er vergleicht einfach die systeminterne Repräsentation. Handelt es sich bei der linken und der rechten Seite um den Verweis auf dieselbe Stelle im Speicher, ergibt der Test `true`; handelt es sich bei der linken und der rechten Seite um denselben Wert, ergibt der Test ebenfalls `true`. In allen anderen Fällen ergibt er `false`

Selbsttestaufgabe 4.1 (Seite 49)

implizit: `g := e`, `f := 2`

explizit: `a := f`

Selbsttestaufgabe 4.2 (Seite 60)

(a) Da die bei der Auswertung des Blocks implizite Zuweisung `x := a` bzw. `y := b` Zeiger auf Objekte und nicht, wie beim *Call by reference*, Zeiger auf Variablen überträgt, ändert eine Änderung des Inhalts von `x` und `y` nicht auch den Wert von `a` und `b`. Siehe auch die Bemerkungen zu *Call by reference* vs. *Call by value* in Abschnitt 4.3.2.

(b) Das Problem ergibt sich beim parameterlose Block nicht, da hier direkt die Variablen aus dem *Home context* verwendet werden.

(c) Weil man naiverweise davon ausgehen könnte, dass die Zuweisung an formale Parametervariablen eben auch die tatsächlichen Parametervariablen betrifft. Es ist daher sinnvoll, die Zuweisung an formale Parameter grundsätzlich zu verbieten, um dem Programmierer die Enttäuschung zu ersparen, wenn er feststellen muss, dass es auf die tatsächlichen Parametervariablen keinen Effekt hat.

Selbsttestaufgabe 4.3 (Seite 63)

Einfach einen Block-Parameter anhängen:

für `true`:

```
267 ifTrue: wahrBlock else: falschBlock
268   ^ wahrBlock value
```

für `false`:

```
269 ifTrue: wahrBlock else: falschBlock
270   ^ falschBlock value
```

Selbsttestaufgabe 4.4 (Seite 63)

für `true`:

```
271 or: arg
272   ^ true

273 and: arg
274   ^ arg

275 not
276   ^ false
```

für `false`: analog

Selbsttestaufgabe 4.5 (Seite 68)

```
277 detect: einBlock
278    self do: [:elem | (einBlock value: elem) ifTrue: [^ elem]]
```