A. Poetzsch-Heffter
Mitarbeit: J. Meyer, P. Müller
Aktualisiert: J. Knoop, M. Müller-Olm, U. Scheben, D. Keller, A. Thies

Einführung in die objektorientierte Programmierung

mathematik und informatik

FernUniversität in Hagen

Inhaltsverzeichnis

Vorwort

Die objektorientierte Programmierung modelliert und realisiert Software-Systeme als „Populationen“ kooperierender Objekte. Vom Prinzip her ist sie demnach eine Vorgehensweise, um Programme gemäß einem bestimmten Grundmodell zu entwerfen und zu strukturieren. In der Praxis ist sie allerdings eng mit dem Studium und der Verwendung objektorientierter Programmiersprachen verbunden: Zum einen gibt die programmiersprachliche Umsetzung den objektorientierten Konzepten konkrete und scharfe Konturen; zum anderen bilden die objektorientierten Sprachen eine unverzichtbare praktische Grundlage für die Implementierung objektorientierter Software.

Der Kurs stellt die konzeptionellen und programmiersprachlichen Aspekte so dar, dass sie sich gegenseitig befruchten. Jedes objektorientierte Konzept wird zunächst allgemein, d.h. unabhängig von einer Programmiersprache eingeführt. Anschließend wird ausführlich seine konkrete programmiersprachliche Umsetzung in Java erläutert. Zum Teil werden auch Realisierungsvarianten in anderen objektorientierten Sprachen vorgestellt. Aus praktischer Sicht ergibt sich damit insbesondere eine konzeptionell strukturierte Einführung in die Sprache und die Programmbibliothek von Java.

Inhaltlich ist der *Kurstext* in acht Kapitel eingeteilt. Die ersten drei Kapitel stellen die zentralen Konzepte und Sprachmittel vor. Kapitel 1 entwickelt das objektorientierte Grundmodell und vergleicht es mit den Modellen anderer Programmierparadigmen. Es fasst Grundbegriffe der Programmierung (insbesondere Variable, Wert, Typ, Ausdruck, Anweisung) zusammen und erläutert sie am Beispiel von Java. Anhand eines typischen Problems zeigt es Defizite der prozeduralen Programmierung auf und demonstriert, wie objektorientierte Techniken derartige Probleme bewältigen. Schließlich geht es nochmals genauer auf die Struktur des Kurses ein und ordnet ihn in verwandte und vertiefende Literatur ein. Kapitel 2 behandelt das Objekt- und Klassenkonzept und beschreibt, wie diese Konzepte in Java umgesetzt sind. Anschließend wird insbesondere auf Kapselungstechniken eingegangen. Kapitel 3 bietet eine detaillierte Einführung in Subtyping, Vererbung und Schnittstellenbildung und demonstriert, wie diese Konzepte in der Programmierung eingesetzt werden können und welche Schwierigkeiten mit ihrem Einsatz verbunden sind.

Die Kapitel 4 und 5 sind der Wiederverwendung von Klassen gewidmet.

Kapitel 4 geht zunächst auf solche Klassenhierarchien ein, deren Klassen nur in sehr eingeschränkter Form voneinander abhängen. Als Beispiele werden Bibliotheksklassen von Java herangezogen. Insbesondere wird auf die Verwendung von Stromklassen zur Ein- und Ausgabe eingegangen. Kapitel 5 behandelt eng kooperierende Klassen und sogenannte Programmgerüste (engl. Frameworks). Ausführlich wird in diesem Zusammenhang Javas Grundpaket zur Konstruktion graphischer Bedienoberflächen erläutert und seine Anwendung beschrieben.

In den Kapiteln 6 und 7 werden die spezifischen Aspekte der Realisierung paralleler und verteilter objektorientierter Programme behandelt. Kapitel 6 stellt dazu insbesondere das Thread-Konzept und die Synchronisationsmittel von Java vor. Kapitel 7 beschreibt die verteilte Programmierung mittels Sockets und entferntem Methodenaufruf. Schließlich bietet Kapitel 8 eine Zusammenfassung, behandelt Varianten bei der Realisierung objektorientierter Konzepte und gibt einen kurzen Ausblick.

Java: warum, woher, welches? Im Vergleich zu anderen objektorientierten Sprachen bietet die Abstützung und Konzentration auf Java wichtige Vorteile: Die meisten objektorientierten Konzepte lassen sich in Java relativ leicht realisieren. Java besitzt ein sauberes Typkonzept, was zum einen die Programmierung erleichtert und zum anderen eine gute Grundlage ist, um Subtyping zu behandeln. Java wird vermehrt in der industriellen Praxis eingesetzt und bietet außerdem eine hilfreiche Vorbereitung für das Erlernen der sehr verbreiteten, programmtechnisch aber komplexeren Sprache C++. Java ist eine Neuentwicklung und keine Erweiterung einer prozeduralen Sprache, so dass es relativ frei von Erblasten ist, die von den objektorientierten Aspekten ablenken (dies gilt z.B. nicht für Modula-3 und C++).

Ein weiterer Vorteil ist die freie Verfügbarkeit einfacher Entwicklungsumgebungen für Java-Programme und die umfangreiche, weitgehend standardisierte Klassenbibliothek. Aktuelle Informationen zu Java findet man auf der Web-Site der Firma Sun ausgehend von `http://java.sun.com`. Insbesondere kann man von dort die von Sun angebotene Software zur Entwicklung und Ausführung von Java-Programmen herunterladen oder bestellen. Die im vorliegenden Kurs beschriebenen Programme wurden mit den Versionen 1.2 und 1.3 des Java 2 Software Development Kits[1] (Standard Edition) entwickelt und getestet.

Hagen, Dezember 2001 *Arnd Poetzsch-Heffter*

[1] Java 2 SDK ist der neue Name für die früher als Java Development Kit, JDK, bezeichnete Entwicklungsumgebung.

Das Lehrgebiet Programmiersprachen und Softwarekonstruktion (Praktische Informatik V) wurde von Februar 2002 bis März 2003 von Dr. Jens Knoop und seit April 2003 von PD Dr. Markus Müller-Olm vertreten. Das unter Leitung von Professor Poetzsch-Heffter entwickelte Kursmaterial wurde für das Sommersemester 2003 von Dr. Jens Knoop und Ursula Scheben durchgesehen und an einigen Stellen ergänzt. Neben redaktionellen Nachbesserungen kamen dabei die folgenden Abschnitte neu zum Kurs hinzu: Der Unterabschnitt „Initialisierungsblöcke“ im Abschnitt 2.14, der Abschnitt 3.3.4 „Verstecken von Attributen und Klassenmethoden vs. Überschreiben von Instanzmethoden“ sowie der Abschnitt 3.3.5 „Auflösen von Methodenaufrufen im Kontext überschriebener und überladener Methoden“. Außerdem wurde am Ende jeder Kurseinheit ein Abschnitt mit Aufgaben und Wiederholungsfragen hinzugefügt. Für das Sommersemester 2004 wurde der Kurstext erneut durchgesehen und redaktionell überarbeitet.

Hagen, Dezember 2003 *Markus Müller-Olm*

Das ehemalige Lehrgebiet Programmiersprachen und Softwarekonstruktion wurde zum Lehrgebiet Programmiersysteme und wird seit Herbst 2004 von Professor Steimann geleitet. Das unter Leitung von Professor Poetzsch-Heffter entwickelte und von Professor Knoop und Professor Müller-Olm überarbeitete Kursmaterial wird für das Sommersemester 2007 von Dr. Ursula Scheben und Dr. Daniela Keller durchgesehen und erneut überarbeitet.

Der Kurstext wird u.a. an Java 5.0 angepasst. Dadurch ergeben sich an verschiedenen Stellen im Kurs Änderungen im Text und in den Programmbeispielen, u.a. werden Java Generics eingeführt. Die alten Übungsaufgaben, zu denen es keine Lösungsvorschläge gab, fallen weg und werden durch neue Beispiele mit Lösungen ersetzt. Desweiteren werden weniger relevante Abschnitte aus dem Kurs entfernt, andere erhalten mehr Gewicht. Entfernt wird z.B. das ehemalige Kapitel 2.1.7 „Klassen als Objekte“, während das bisherigen Kapitel 7 über verteilte, objektorientierte Systeme als grundlegend und wichtig eingestuft wird.

Hagen, Januar 2007 *Ursula Scheben*

Studierhinweise zur Kurseinheit 1

Diese Kurseinheit beschäftigt sich mit dem ersten Kapitel des Kurstextes. Sie sollten dieses Kapitel im Detail studieren und verstehen. Ausgenommen von dem Anspruch eines detaillierten Verständnisses ist der Abschnitt 1.2.2 „Deklarative Programmierung“; bei dessen Inhalt reicht ein Verstehen der erläuterten Prinzipien. Schwerpunkte bilden die Abschnitte 1.1 „Objektorientierung: Konzepte und Stärken“ und der Abschnitt 1.3.1 „Grundlegende Sprachmittel am Beispiel von Java“. Nehmen Sie sich die Zeit, die Sprachkonstrukte an kleinen, selbst entworfenen Beispielen im Rahmen dieser Kurseinheit zu üben! Lassen Sie sich von den am Ende des Kapitels gegebenen Aufgaben zu eigenen selbständigen Übungen anregen. Ein bloßes Durchlesen dieses Kapitels ist nicht ausreichend.

Die Erfahrung lehrt, dass es sehr hilfreich ist, sich bei der Beschäftigung mit einem Kurs dessen inhaltliche Struktur genau zu vergegenwärtigen und die Kursstruktur als Einordnungshilfe und Gedächtnisstütze zu verwenden. Versuchen Sie deshalb, sich den in Abschnitt 1.4 „Aufbau und thematische Einordnung“ erläuterten Aufbau des Kurses einzuprägen.

Lernziele:

- Grundlegende Konzepte der objektorientierten Programmierung.
- Basiskonzepte der objektorientierten Analyse.
- Unterschied zwischen objektorientierter Programmierung und anderen Programmierparadigmen.
- Sprachliche Grundlagen der Programmierung mit Java.
- Programmtechnische Fähigkeiten im Umgang mit Java.

Kapitel 1

Objektorientierung: Ein Einstieg

Dieses Kapitel hat zwei Schwerpunkte. Zum einen erläutert es den konzeptionellen Hintergrund der objektorientierten Programmierung und vergleicht ihn mit dem anderer Programmierparadigmen. Zum anderen vermittelt es die benötigten programmiersprachlichen Voraussetzungen. Insgesamt gibt es erste Antworten auf die folgenden Fragen:

1. Was sind die grundlegenden Konzepte und wo liegen die Stärken der objektorientierten Programmierung?

2. Wie unterscheidet sich objektorientierte Programmierung von anderen Programmierparadigmen?

3. Wie ist der Zusammenhang zwischen objektorientierten Konzepten und objektorientierten Programmiersprachen?

Jeder dieser Fragen ist ein Abschnitt gewidmet. Der dritte Abschnitt bietet darüber hinaus eine zusammenfassende Einführung in elementare Sprachkonzepte, wie sie in allen Programmiersprachen vorkommen, und erläutert ihre Umsetzung in Java. Abschnitt 1.4 beschreibt den Aufbau der folgenden Kapitel.

1.1 Objektorientierung: Konzepte und Stärken

Dieser Abschnitt bietet eine erste Einführung in objektorientierte Konzepte (Abschn. 1.1.1) und stellt die objektorientierte Programmierung als Antwort auf bestimmte softwaretechnische Anforderungen dar (Abschn. 1.1.2). Zunächst wollen wir allerdings kurz den Begriff „Objektorientierte Programmierung“ reflektieren. Dabei soll insbesondere deutlich werden, dass objektorientierte Programmierung mehr ist als die Programmierung in einer objektorientierten Programmiersprache.

Programmierung

Objektorientierte Programmierung: Was bedeutet das? Der Begriff „Programmierung“ wird mit unterschiedlicher Bedeutung verwendet. Im engeren Sinn ist das Aufschreiben eines Programms in einer gegebenen Programmiersprache gemeint: Wir sehen die Programmierer vor uns, die einen Programmtext in ihrem Rechner editieren. Im weiteren Sinn ist die Entwicklung und Realisierung von Programmen ausgehend von einem allgemeinen Softwareentwurf gemeint, d.h. einem Softwareentwurf, in dem noch keine programmiersprachspezifischen Entscheidungen getroffen sind. Programmierung in diesem Sinne beschäftigt sich also auch mit Konzepten und Techniken zur Überwindung der Kluft zwischen Softwareentwurf und Programmen. *In diesem Kurs wird Programmierung in dem weitergefassten Sinn verstanden*. Konzepte der Programmierung beeinflussen dementsprechend sowohl Programmiersprachen und -techniken als auch den Softwareentwurf und umgekehrt.

Obj.-or. Programmierung

Objektorientierte Programmierung ist demnach Programmentwicklung mit Hilfe objektorientierter Konzepte und Techniken. Dabei spielen naturgemäß programmiersprachliche Aspekte eine zentrale Rolle. Resultat einer objektorientierten Programmentwicklung sind in der Regel, aber nicht notwendig, Programme, die in einer objektorientierten Programmiersprache verfasst sind. Im Gesamtbild der Softwareentwicklung wird die objektorientierte Programmierung durch objektorientierte Techniken für Analyse, Entwurf und Testen ergänzt.

1.1.1 Gedankliche Konzepte der Objektorientierung

Die Objektorientierung bezieht ihre gedanklichen Grundlagen aus Vorgängen der realen Welt[1]. Vorgänge werden durch handelnde Individuen modelliert, die Aufträge erledigen und vergeben können. Dabei ist es zunächst unerheblich, ob die Individuen Personen, Institutionen, materielle Dinge oder abstrakte Gebilde sind. In der objektorientierten Programmierung werden die Individuen als *Objekte* bezeichnet. Dieser Abschnitt führt an einem kleinen Beispielszenario in die gedanklichen Konzepte der Objektorientierung ein und gibt eine erste Erläuterung der Begriffe „Nachricht“, „Methode“, „Klassifikation“ und „Vererbung“.

Objekt

Nachrichten und Methoden. Ein zentraler Aspekt der Objektorientierung ist die Trennung von Auftragserteilung und Auftragsdurchführung. Betrachten wir dazu ein kleines Beispiel: Wir nehmen an, dass ein Herr P. ein Buch kaufen möchte. Um das Buch zu besorgen, erteilt Herr P. einem Buchhändler

[1] Der gedankliche Hintergrund der deklarativen Programmierung stammt aus der Mathematik, die prozedurale Programmierung hat sich durch Abstraktion aus der maschinennahen Programmierung und aus mathematischen Berechnungsverfahren entwickelt; vgl. Abschnitt 1.2.

den Auftrag, das Buch zu beschaffen und es ihm zuzuschicken. Die Auftragsdurchführung liegt dann in der Hand des Buchhändlers. Genauer besehen passiert Folgendes:

1. Herr P. löst eine Aktion aus, indem er dem Buchhändler einen Auftrag gibt. Übersetzt in die Sprache der Objektorientierung heißt das, dass ein Senderobjekt, nämlich Herr P., einem Empfängerobjekt, nämlich dem Buchhändler, eine *Nachricht* schickt. Diese Nachricht besteht üblicherweise aus der Bezeichnung des Auftrags (Buch beschaffen und zuschicken) und weiteren Parametern (etwa dem Buchtitel). *Nachricht*

2. Der Buchhändler besitzt eine bestimmte *Methode*, wie er Herrn P.'s Auftrag durchführt (etwa: nachschauen, ob Buch am Lager, ansonsten billigsten Großhändler suchen, etc.). Diese Methode braucht Herr P. nicht zu kennen. Auch wird die Methode, wie das Buch beschafft wird, von Buchhändler zu Buchhändler im Allgemeinen verschieden sein. *Methode*

Die konzeptionelle Trennung von Auftragserteilung und Auftragsdurchführung, d.h. die Unterscheidung von Nachricht und Methode, führt zu einer klaren Aufgabenteilung: Der Auftraggeber muss sich jemanden suchen, der seinen Auftrag versteht und durchführen kann. Er weiß im Allgemeinen nicht, wie der Auftrag bearbeitet wird (Geheimnisprinzip, engl. Information Hiding). Der Auftragsempfänger ist für die Durchführung verantwortlich und besitzt dafür eine Methode.

Klassifikation und Vererbung. Die Klassifikation von Gegenständen und Begriffen durchzieht beinahe alle Bereiche unseres Lebens. Beispielsweise sind Händler und Geschäfte nach Branchen klassifiziert. Jede Branche ist dabei durch die Dienstleistungen charakterisiert, die ihre Händler erbringen: Buchhändler handeln mit Büchern, Lebensmittelhändler mit Lebensmitteln. Was im Geschäftsleben die Branchen sind, sind in der Sprache der Objektorientierung die Klassen. Eine *Klasse* legt die Methoden und Eigenschaften fest, die allen ihren Objekten gemeinsam sind. Klassen lassen sich hierarchisch organisieren. Abbildung 1.1 demonstriert eine solche hierarchische *Klassifikation* am Beispiel von Branchen. Dabei besitzen die übergeordneten Klassen nur *Klasse* *Klassifikation*

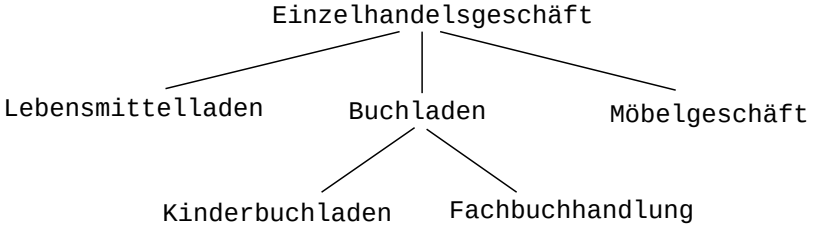

Abbildung 1.1: Klassifikation von Geschäften

Eigenschaften, die den untergeordneten Klassen bzw. ihren Objekten gemeinsam sind. Für die Beschreibung der Eigenschaften von Klassen und Objekten bringt die hierarchische Organisation zwei entscheidende Vorteile gegenüber einer unstrukturierten Menge von Klassen:

abstrakte Klassen

1. Es lassen sich *abstrakte* Klassen bilden; das sind Klassen, die nur dafür angelegt sind, Gemeinsamkeiten der untergeordneten Klassen zusammenzufassen. Jedes Objekt, das einer abstrakten Klasse zugerechnet wird, gehört auch zu einer der untergeordneten Klassen. Beispielsweise ist Einzelhandelsgeschäft eine abstrakte Klasse. Sie fasst die Eigenschaften zusammen, die allen Geschäften gemeinsam sind. Es gibt aber kein Geschäft, das nur ein Einzelhandelsgeschäft ist und keiner Branche zugeordnet werden kann. Anders ist es mit der Klasse Buchladen. Es gibt Buchläden, die zu keiner *spezielleren Klasse* gehören; d.h. die Klasse Buchladen ist nicht abstrakt.

Vererbung

2. Eigenschaften und Methoden, die mehreren Klassen gemeinsam sind, brauchen nur einmal bei der übergeordneten Klasse beschrieben zu werden und können von untergeordneten Klassen *geerbt* werden. Beispielsweise besitzt jedes Einzelhandelsgeschäft eine Auftragsabwicklung. Die Standardverfahren der Auftragsabwicklung, die bei allen Geschäften gleich sind, brauchen nur einmal bei der Klasse Einzelhandelsgeschäft beschrieben zu werden. Alle untergeordneten Klassen können diese Verfahren erben und an ihre speziellen Verhältnisse anpassen. Für derartige Spezialisierungen stellt die objektorientierte Programmierung bestimmte Techniken zur Verfügung.

Spezialisierung

Wie wir in den folgenden Kapiteln noch sehen werden, ist das Klassifizieren und Spezialisieren eine weitverbreitete Technik, um Wissen und Verfahren zu strukturieren. Die Nutzung dieser Techniken für die Programmierung ist ein zentraler Aspekt der Objektorientierung.

Nach dieser kurzen Skizze der gedanklichen Konzepte, die der Objektorientierung zugrunde liegen, werden wir uns im folgenden Abschnitt den softwaretechnischen Anforderungen zuwenden, zu deren Bewältigung die objektorientierte Programmierung angetreten ist.

1.1.2 Objektorientierung als Antwort auf softwaretechnische Anforderungen

Simula

Objektorientierte Programmierung ist mittlerweile älter als 30 Jahre. Bereits die Programmiersprache *Simula 67* besaß alle wesentlichen Eigenschaften für die objektorientierte Programmierung (siehe z.B. [Lam88]). Die Erfolgsgeschichte der objektorientierten Programmierung hat allerdings erst Anfang der achtziger Jahre richtig an Fahrt gewonnen, stark getrieben von der

Programmiersprache *Smalltalk* (siehe [GR89]) und der zugehörigen Entwicklungsumgebung. Es dauerte nochmals ein Jahrzehnt, bis die objektorientierte Programmierung auch in der kommerziellen Programmentwicklung nennenswerte Bedeutung bekommen hat. Mittlerweile ist Objektorientierung so populär geworden, dass sich viele Software-Produkte, Werkzeuge und Vorgehensmodelle schon aus Marketing-Gründen objektorientiert nennen – unnötig zu sagen, dass nicht überall, wo „objektorientiert" draufsteht, auch „objektorientiert" drin ist.

Smalltalk

Es ist schwer im Einzelnen zu klären, warum es solange gedauert hat, bis die objektorientierten Techniken breitere Beachtung erfahren haben, und warum sie nun so breite Resonanz finden. Sicherlich spielen bei dieser Entwicklung viele Aspekte eine Rolle – das geht beim Marketing los und macht bei ästhetischen Überlegungen nicht halt. Wir beschränken uns hier auf einen inhaltlichen Erklärungsversuch: Objektorientierte Konzepte sind kein Allheilmittel; sie leisten aber einen wichtigen Beitrag zur Lösung bestimmter *softwaretechnischer Probleme*. In dem Maße, in dem diese Problemklasse in Relation zu anderen softwaretechnischen Problemen an Bedeutung gewonnen hat, haben auch die objektorientierten Techniken an Bedeutung gewonnen und werden weiter an Bedeutung gewinnen.

Vier softwaretechnische Aufgabenstellungen stehen in einer sehr engen Beziehung zur Entwicklung objektorientierter Techniken und Sprachen:

1. softwaretechnische Simulation,
2. Konstruktion interaktiver, graphischer Bedienoberflächen,
3. Programm-Wiederverwendung und
4. verteilte Programmierung.

Nach einer kurzen Erläuterung dieser Bereiche werden wir ihre Gemeinsamkeiten untersuchen.

1. Simulation: Grob gesprochen lassen sich zwei Arten von Simulation unterscheiden: die Simulation *kontinuierlicher* Prozesse, bespielsweise die numerische Berechnung von Klimavorhersagen im Zusammenhang mit dem Treibhauseffekt, und die Simulation *diskreter* Vorgänge, beispielsweise die Simulation des Verkehrsflusses an einer Straßenkreuzung oder die virtuelle Besichtigung eines geplanten Gebäudes auf Basis eines Computermodells. Wir betrachten im Folgenden nur die diskrete Simulation. Softwaretechnisch sind dazu drei Aufgaben zu erledigen:

   1. Modellierung der statischen Komponenten des zugrunde liegenden Systems.
   2. Beschreibung der möglichen Dynamik des Systems.
   3. Test und Analyse von Abläufen des Systems.

Für das Beispiel der Simulation des Verkehrsflusses an einer Straßenkreuzung heißt das: Die Straßenkreuzung mit Fahrspuren, Bürgersteigen, Übergängen usw. muss modelliert werden; Busse, Autos und Fahrräder müssen mit ihren Abmessungen und Bewegungsmöglichkeiten beschrieben werden (Aufgabe 1). Die Dynamik der Objekte dieses Modells muss festgelegt werden, d.h. die möglichen Ampelstellungen, das Erzeugen neuer Fahrzeuge an den Zufahrten zur Kreuzung und die Bewegungsparameter der Fahrzeuge (Aufgabe 2). Schließlich muss eine Umgebung geschaffen werden, mit der unterschiedliche Abläufe auf der Kreuzung gesteuert, getestet und analysiert werden können (Aufgabe 3).

2. **Graphische Bedienoberflächen:** Interaktive, graphische Bedienoberflächen ermöglichen die nicht-sequentielle, interaktive Steuerung von Anwendungsprogrammen über direkt manipulierbare, graphische Bedienelemente wie Schaltflächen, Auswahlmenüs und Eingabefenster. Der Konstruktion graphischer Bedienoberflächen liegen eine ergonomische und zwei softwaretechnische Fragestellungen zugrunde:

   1. Wie muss eine Oberfläche gestaltet werden, um der Modellvorstellung des Benutzers von der gesteuerten Anwendung gerecht zu werden und eine leichte Bedienbarkeit zu ermöglichen?
   2. Der Benutzer möchte quasi-parallel arbeiten, z.B. in einem Fenster eine Eingabe beginnen, bevor er diese beendet, eine andere Eingabe berichtigen und eine Information in einem anderen Fenster erfragen, dann ggf. eine Anwendung starten und ohne auf deren Ende zu warten, mit der erstgenannten Eingabe fortfahren. Ein derartiges Verhalten wird von einem sequentiellen Programmiermodell nicht unterstützt: Wie sieht ein gutes Programmiermodell dafür aus?
   3. Das Verhalten einer Oberflächenkomponente ergibt sich zum Großteil aus der Standardfunktionalität für die betreffende Komponentenart und nur zum geringen Teil aus Funktionalität, die spezifisch für die Komponente programmiert wurde. Beispielsweise braucht zu einer Schaltfläche nur programmiert zu werden, was bei einem Mausklick getan werden soll; die Zuordnung von Mausklicks zur Schaltfläche, die Verwaltung und das Weiterreichen von Mausbewegungen und anderen Ereignissen steht bereits als Standardfunktionalität zur Verfügung. Wie lässt sich diese Standardfunktionalität in Form von Oberflächenbausteinen so zur Verfügung stellen, dass sie programmtechnisch gut, sicher und flexibel handhabbar ist?

3. Wiederverwendung von Programmen Zwei Hauptprobleme stehen bei der Wiederverwendung von Programmen im Mittelpunkt:

1. Wie finde ich zu einer gegebenen Aufgabenstellung einen Programmbaustein mit Eigenschaften, die den gewünschten möglichst nahe kommen?
2. Wie müssen Programme strukturiert und parametrisiert sein, um sich für Wiederverwendung zu eignen?

Wesentliche Voraussetzung zur Lösung des ersten Problems ist die Spezifikation der Eigenschaften der Programmbausteine, insbesondere derjenigen Eigenschaften, die an den Schnittstellen, d.h. für den Benutzer sichtbar sind. Das zweite Problem rührt im Wesentlichen daher, dass man selten zu einer gegebenen Aufgabenstellung einen fertigen, passenden Programmbaustein findet. Programme müsssen deshalb gut anpassbar und leicht erweiterbar sein, um sich für Wiederverwendung zu eignen. Derartige Anpassungen sollten möglich sein, ohne den Programmtext der verwendeten Bausteine manipulieren zu müssen.

4. Verteilte Programmierung Die Programmierung verteilter Anwendungen – oft kurz als verteilte Programmierung bezeichnet – soll es ermöglichen, dass Programme, die auf unterschiedlichen Rechnern laufen, miteinander kommunizieren und kooperieren können und dass Daten und Programmteile über digitale Netze automatisch verteilt bzw. beschafft werden können. Demzufolge benötigt die verteilte Programmierung ein Programmiermodell,

- in dem räumliche Verteilung von Daten und Programmteilen dargestellt werden kann,
- in dem Parallelität und Kommunikation in natürlicher Weise beschrieben werden können und
- das eine geeignete Partitionierung von Daten und Programmen in übertragbare Teile unterstützt.

In der Einleitung zu diesem Abschnitt wurde der Erfolg objektorientierter Techniken teilweise damit erklärt, dass sie sich besser als andere Techniken eignen, um die skizzierten softwaretechnischen Aufgabenstellungen zu bewältigen. Schrittweise wollen wir im Folgenden untersuchen, woher die bessere Eignung für diese Aufgaben kommt. Dazu stellen wir zunächst einmal gemeinsame Anforderungen zusammen. Diese Anforderungen dienen uns in den kommenden Abschnitten als Grundlage für die Diskussion unterschiedlicher Programmiermodelle und insbesondere, um die spezifischen Aspekte des objektorientierten Programmiermodells herauszuarbeiten.

Frage. Welche Anforderungen treten in mehreren der skizzierten softwaretechnischen Aufgabenstellungen auf?

Die Aufgabenstellungen sind zwar in vielen Aspekten sehr unterschiedlich; aber jede von ihnen stellt zumindest zwei der folgenden drei konzeptionellen Anforderungen:

1. Sie verlangt ein inhärent paralleles Ausführungsmodell, mit dem insbesondere Bezüge zur realen Welt modelliert werden können. („Inhärent parallel" bedeutet, dass Parallelität eine Eigenschaft des grundlegenden Ausführungsmodells ist und nicht erst nachträglich hinzugefügt werden muss.)

2. Die Strukturierung der Programme in kooperierende Programmteile mit klar definierten Schnittstellen spielt eine zentrale Rolle.

3. Anpassbarkeit, Klassifikation und Spezialisierung von Programmteilen ist eine wichtige Eigenschaft und sollte möglich sein, ohne bestehende Programmtexte manipulieren zu müssen.

In der Simulation ermöglicht ein paralleles Ausführungsmodell eine größere Nähe zwischen dem simulierten Teil der realen Welt und dem simulierenden Softwaresystem. Dabei sollten Programme so strukturiert sein, dass diejenigen Daten und Aktionen, die zu einem Objekt der realen Welt gehören, innerhalb des Programms zu einer Einheit zusammengefasst sind. Darüber hinaus sind Klassifikationshierarchien bei der Modellierung sehr hilfreich: Im obigen Simulationsbeispiel ließen sich dann die Eigenschaften aller Fahrzeuge gemeinsam beschreiben; die Eigenschaften speziellerer Fahrzeugtypen (Autos, Busse, Fahrräder) könnte man dann durch Verfeinerung der Fahrzeugeigenschaften beschreiben.

Wie bereits skizziert liegt auch interaktiven, graphischen Bedienoberflächen ein paralleles Ausführungsmodell zugrunde. Der Bezug zur realen Welt ergibt sich hier aus der Interaktion mit dem Benutzer. Anpassbarkeit, Klassifikation und die Möglichkeit der Spezialisierung von Programmteilen sind bei Bedienoberflächenbaukästen besonders wichtig. Sie müssen eine komplexe und mächtige Standardfunktionalität bieten, um dem Programmierer Arbeit zu sparen. Sie können andererseits aber nur unfertige Oberflächenkomponenten bereitstellen, die erst durch Spezialisierung ihre dedizierte, für den speziellen Anwendungsfall benötigte Funktionalität erhalten.

Die unterschiedlichen Formen der Wiederverwendung von Programmen werden wir in späteren Kapiteln näher analysieren. Im Allgemeinen stehen bei der Wiederverwendung die Anforderungen 2 und 3 im Vordergrund. Wenn man den Begriff „Wiederverwendung" weiter fasst und z.B. auch dynamisches Laden von Programmkomponenten über eine Netzinfrastruktur oder sogar das Nutzen im Netz verfügbarer Dienste als Wiederverwendung

begreift, spielen auch Aspekte der verteilten Programmierung eine wichtige Rolle für die Wiederverwendung.

Bei der verteilten Programmierung ist ein paralleles Ausführungsmodell gefordert, dessen Bezugspunkte in der realen Welt sich durch die räumliche Verteilung der kooperierenden Programmteile ergeben. Darüber hinaus bildet Anforderung 2 eine Grundvoraussetzung für die verteilte Programmierung; es muss klar definiert sein, wer kooperieren kann und wie die Kommunikation im Einzelnen aussieht.

Zusammenfassung. Die Entwicklung objektorientierter Konzepte und Sprachen war und ist eng verknüpft mit der Erforschung recht unterschiedlicher softwaretechnischer Aufgabenstellungen (das ist eine Beobachtung). Aus diesen Aufgabenstellungen resultieren bestimmte Anforderungen an die Programmierung (das ist ein Analyseergebnis). Die Konzepte und Techniken der objektorientierten Programmierung sind im Allgemeinen besser als andere Programmierparadigmen geeignet, diese Anforderungen zu bewältigen (dies ist – immer noch – eine Behauptung). Ein Ziel dieses Kurses ist es, diese Behauptung argumentativ zu untermauern und dabei zu zeigen, wie die bessere Eignung erreicht werden soll und dass dafür auch ein gewisser Preis zu zahlen ist. Als Grundlage für diese Diskussion bietet der nächste Abschnitt eine kurze Übersicht über andere, konkurrierende Programmierparadigmen.

1.2 Paradigmen der Programmierung

Eine Softwareentwicklerin, die ausschließlich funktionale Programme entwickelt hat, geht anders an eine softwaretechnische Problemstellung heran als ein Softwareentwickler, der nur mit Pascal gearbeitet hat. Sie benutzt andere Konzepte und Techniken, um Informationen zu organisieren und zu repräsentieren als ihr Berufskollege. Sie stützt ihre Entscheidungen auf andere Theorien und andere Standards. Engverzahnte Gebäude aus Konzepten, Vorgehensweisen, Techniken, Theorien und Standards fasst Thomas Kuhn (vgl. [Kuh76]) unter dem Begriff „Paradigma“ zusammen. Er benutzt den Begriff im Rahmen einer allgemeinen Untersuchung wissenschaftlichen Fortschritts. Wir verwenden diesen Begriff hier im übertragenen Sinne für den Bereich der Programmierung. Wie aus der Erläuterung zu ersehen ist, können sich unterschiedliche Paradigmen durchaus gegenseitig ergänzen. Dies gilt insbesondere auch für die Paradigmen der Programmierung.

Paradigma

Wir unterscheiden drei Programmierparadigmen: die prozedurale, deklarative und objektorientierte Programmierung. Dabei betrachten wir die prozedurale Programmierung als eine Erweiterung der imperativen Programmierung und begreifen die funktionale Programmierung als Spezialfall der deklarativen Programmierung. Um beurteilen zu können, was das Spezifi-

sche an der objektorientierten Programmierung ist und ob bzw. warum sie zur Lösung der im letzten Abschnitt erläuterten Aufgabenstellungen besonders geeignet ist, stellen wir in diesem Abschnitt die wesentlichen Eigenschaften alternativer Programmierparadigmen vor. Damit soll insbesondere auch in einer größeren Breite illustriert werden, was wir unter Programmierung verstehen, welche Konzepte, Techniken und Modelle dabei eine Rolle spielen und wie sie sich unterscheiden.

Eine zentrale Aufgabe der Softwareentwicklung besteht darin, allgemeine informationsverarbeitende Prozesse (z.B. Verwaltungsprozesse in Unternehmen, Steuerungsprozesse in Kraftanlagen oder Fahrzeugen, Berechnungsprozesse zur Lösung von Differentialgleichungen, Übersetzungsprozesse für Programme) so zu modellieren, dass sie von Rechenmaschinen verarbeitet werden können. Die Modellierung besteht im Wesentlichen aus der Modellierung der Informationen und der Modellierung der Verarbeitung.

Aufgabe der Programmierung ist es, die im Rahmen des Softwareentwurfs entwickelten Modelle soweit zu verfeinern, dass sie mittels Programmiersprachen formalisiert und damit auf Rechenmaschinen ausgeführt werden können. Die Programmierparadigmen unterscheiden sich dadurch, wie die Informationen und deren Verarbeitung modelliert werden und wie das Zusammenspiel von Informationen und Verarbeitung aussieht. Um die Vorstellung der unterschiedlichen Konzepte prägnant zu halten, werden wir ihre Darstellung angemessen vereinfachen. In der Praxis finden sich selbstverständlich Mischformen dieser Konzepte. Modulkonzepte lassen wir zunächst unberücksichtigt, da sie für alle Paradigmen existieren und deshalb zur Unterscheidung nicht wesentlich beitragen.

Die folgenden Abschnitte erläutern, wie Informationen und deren Verarbeitung in den unterschiedlichen Programmierparadigmen modelliert und beschrieben werden.

1.2.1 Prozedurale Programmierung

In der prozeduralen Programmierung ist die Modellierung der Informationen von der Modellierung der Verarbeitung klar getrennt. Informationen werden im Wesentlichen durch Grunddaten (ganze Zahlen, boolesche Werte, Zeichen, Zeichenreihen usw.) modelliert, die in Variablen gespeichert werden. Variablen lassen sich üblicherweise zu Feldern (Arrays) oder Verbunden (Records) organisieren, um komplexere Datenstrukturen zu realisieren. Darüber hinaus kann man mit Referenzen/Zeigern rechnen, die auf Variable verweisen.

Zustand

Die Verarbeitung wird modelliert als eine Folge von globalen Zustandsübergängen, wobei sich der globale *Zustand* aus den Zuständen der Variablen zusammensetzt (in der Praxis gehören auch die vom Programm bearbeiteten Dateien zum Zustand). Bei jedem Zustandsübergang wird der Inhalt einer oder einiger weniger Variablen verändert. Möglicherweise können bei einem

Zustandsübergang auch Variablen erzeugt oder entfernt werden. Ein Zustandsübergang wird durch eine elementare Anweisung beschrieben (z.B. in Pascal durch eine Zuweisung, `x:=4`; eine Allokation, `new(p)`; eine Deallokation, `dispose(p)`, oder eine Lese- bzw. Schreiboperation); d.h. es wird explizit vorgeschrieben, wie der Zustand zu ändern ist (daher auch der Name *imperative Programmierung*). Folgen von Zustandsübergängen werden durch zusammengesetzte Anweisungen[2] beschrieben. Zusammengesetzte Anweisungen können auch unendliche, d.h. nichtterminierende Übergangsfolgen beschreiben; dies ist zum Beispiel wichtig zur Programmierung von Systemen, die bis zu einem unbestimmten Zeitpunkt ununterbrochen Dienste anbieten sollen, wie z.B. Betriebssysteme, Netzserver oder die Event-Behandlung von graphischen Bedienoberflächen.

Prozedur

Eine *Prozedur* ist eine benannte, parametrisierte Anweisung, die meistens zur Erledigung einer bestimmten Aufgabe bzw. Teilaufgabe dient; Prozeduren, die Ergebnisse liefern (zusätzlich zur möglichen Veränderung, Erzeugung oder Entfernung von Variablen), werden *Funktionsprozeduren* genannt. Eine Prozedur kann zur Erledigung ihrer Aufgaben andere Prozeduren aufrufen. Jeder Programmstelle mit einem Prozeduraufruf ist eindeutig eine auszuführende Prozedur zugeordnet;[3] z.B. wird im folgenden Programmfragment in Zeile (2) bei jedem Schleifendurchlauf die Prozedur `proz007` aufgerufen:

```
(1)  while( not abbruchbedingung ) {
         ...
(2)      proz007(...);
         ...
     }
```

Wie wir sehen werden, gibt es in der objektorientierten Programmierung keine derartige eindeutige Zuordnung von Aufrufstelle zu Prozedur.

!

Das Grundmodell der Verarbeitung in prozeduralen Programmen ist die Zustandsänderung. Bei terminierenden Programmen wird ein Eingabezustand in einen Ausgabezustand transformiert, bei nichtterminierenden Programmen wird ein initialer Zustand schrittweise verändert. Die gesamte Zustandsänderung wird in einzelne Teiländerungen zerlegt, die von Prozeduren ausgeführt werden:

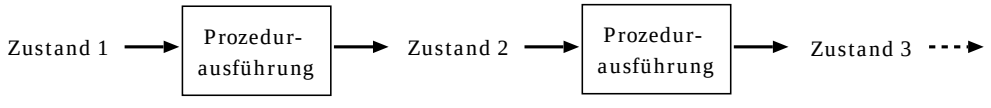

Abbildung 1.2: Schrittweise Änderung globaler Zustände

[2] Z.B. bedingte Anweisung, Schleifen etc.

[3] Wir sehen an dieser Stelle von Prozedurparametern und Prozedurvariablen ab.

Damit lässt sich das Grundmodell der prozeduralen Programmierung so zusammenfassen: Modelliere die Informationen durch (im Wesentlichen globale) Zustände über Variablen und Daten und modelliere die Verarbeitung als schrittweise Änderung des Zustands. Zustandsänderungen werden durch Anweisungen beschrieben. Mehrfach verwendete Anweisungsfolgen lassen sich zu Prozeduren zusammenfassen. Prozeduren bilden demnach das zentrale Strukturierungsmittel prozeduraler Programme. Sie strukturieren aber nur die Verarbeitung und nicht die Modellierung der Information.

Frage. Was sind die Schwächen der prozeduralen Programmierung in Hinblick auf die drei in Abschn. 1.1.2 skizzierten Anforderungen?

Die prozedurale Programmierung basiert auf einem sequentiellen Ausführungsmodell. Um Parallelität ausdrücken zu können, muss das Grundmodell erweitert werden, beispielsweise indem die parallele Ausführung von Anweisungen oder Prozeduren unterstützt wird oder indem zusätzliche Sprachelemente zur Verfügung gestellt werden (z.B. Prozesse).

Das Grundmodell der prozeduralen Programmierung erlaubt in natürlicher Weise die Strukturierung der Verarbeitung. Es bietet aber wenig Möglichkeiten, um Teile des Zustands mit den auf ihm operierenden Prozeduren zusammenzufassen und die Kapselung von Daten zu erreichen. Dieser Schwäche wird mit Modulkonzepten begegnet. Die üblichen Modulkonzepte gestatten es, Typen, Variablen und Prozeduren zusammenzufassen, also die Programmtexte zu strukturieren. Mit Modulen kann man aber nicht rechnen: Sie können nicht als Parameter beim Prozeduraufruf übergeben oder Variablen zugewiesen werden; insbesondere ist es normalerweise nicht möglich, während der Programmausführung Kopien von Modulen zu erzeugen.

Prozedurale Programmierung wird meist im Zusammenhang mit strenger Typisierung behandelt. Typkonzepte verbessern zwar die statische Überprüfbarkeit von Programmen, erschweren aber deren Anpassbarkeit. Die Typen der Parameter einer Prozedur `p` sind fest vorgegeben. Wird ein Typ erweitert oder modifiziert, entsteht ein neuer Typ, dessen Elemente von `p` nicht mehr akzeptiert werden, selbst wenn die Änderungen keinen Einfluss auf die Bearbeitung hätten. In diesem Bereich hat man durch Prozedurparameter, generische Prozeduren und generische Module Verbesserungen erzielt.

Die prozedurale Programmierung ermöglicht eine Modellierung informationsverarbeitender Prozesse mit relativ einfachen, effizient zu implementierenden Mitteln. Diese Einfachheit ist ihre Stärke. Ihr Grundmodell bedarf aber einiger Erweiterungen, um den im letzten Abschnitt skizzierten Anforderungen gerecht zu werden. Aus der Summe dieser Erweiterungen resultiert dann allerdings ein komplexes Programmiermodell. Eine sprachliche Umsetzung führt zu sehr umfangreichen Programmiersprachen (typisches Beispiel hierfür ist die Sprache Ada).

1.2.2 Deklarative Programmierung

Die deklarative Programmierung hat das Ziel, mathematische Beschreibungsmittel für die Programmierung nutzbar zu machen. Damit sollen vor allem zwei Schwächen der imperativen bzw. prozeduralen Programmierung überwunden werden. Zum einen soll der Umgang mit komplexeren Daten wie Listen, Bäumen, Funktionen und Relationen erleichtert werden, zum anderen soll das oft fehleranfällige Arbeiten mit Variablen überwunden werden. Die deklarative Programmierung verzichtet im Wesentlichen auf den Zustandsbegriff. Ein deklaratives Programm ist nicht mehr eine Verarbeitungsvorschrift, sondern eine Spezifikation der gewünschten Programmergebnisse mit speziellen mathematischen Beschreibungsmitteln. Im Mittelpunkt steht also die Modellierung von Informationen, ihren Beziehungen und Eigenschaften. Die Verarbeitung der Informationen geschieht in deklarativen Programmiermodellen zum Großteil implizit.

Die deklarative Programmierung kennt mehrere Ausprägungen, die sich im Wesentlichen durch die verwendeten mathematischen Beschreibungsmittel unterscheiden. Wir stellen hier die funktionale und logische Programmierung kurz vor, um die obigen allgemeinen Ausführungen zu illustrieren.

Funktionale Programmierung. Die funktionale Programmierung betrachtet ein Programm als eine partielle Funktion von Eingabe- auf Ausgabedaten. Die Ausführung eines funktionalen Programms entspricht der Anwendung der Funktion auf eine Eingabe.

Ein funktionales Programm besteht im Wesentlichen aus Deklarationen von Datentypen und Funktionen, wobei Parameter und Ergebnisse von Funktionen selbst wieder Funktionen sein können (solche Funktionen nennt man *Funktionen höherer Ordnung*). Da funktionale Programmierung keine Variablen, keine Zeiger und keine Schleifen kennt, kommt der Rekursion zur Definition von Datenstrukturen und Funktionen eine zentrale Rolle zu. Mit einem kleinen funktionalen Programm, das prüft, ob ein Binärbaum Unterbaum eines anderen Binärbaums ist, wollen wir diese Art der Programmierung kurz illustrieren.

Ein Binärbaum ist entweder ein Blatt oder eine Astgabel mit zwei Teilbäumen. Ein Binärbaum `a` ist Unterbaum von einem Blatt, wenn `a` ein Blatt ist; ein Binärbaum ist Unterbaum von einem zusammengesetzten Binärbaum mit Teilbäumen `b1` und `b2`, wenn er (1) gleich dem zusammengesetzten Binärbaum ist oder (2) Unterbaum von `b1` ist oder (3) Unterbaum von `b2` ist. Formuliert in der Syntax der funktionalen Programmiersprache Gofer ergibt sich aus dieser Definition folgendes kompakte Programm:

```
data  BinBaum = Blatt  |  AstGabel  BinBaum BinBaum

istUnterbaum :: BinBaum -> BinBaum -> Bool
```

```
istUnterbaum a Blatt    =  ( a == Blatt )
istUnterbaum a (AstGabel b1 b2) =
                              ( a == (AstGabel b1 b2) )
                           || ( istUnterbaum a b1 )
                           || ( istUnterbaum a b2 )
```

Das Schlüsselwort `data` leitet eine Datentypdeklaration ein, hier die Deklaration des Typs `BinBaum` mit den beiden Alternativen `Blatt` und `AstGabel`. Die Funktion `istUnterbaum` nimmt zwei Werte vom Typ `BinBaum` als Eingabe und liefert einen booleschen Wert als Ergebnis; `istUnterbaum` angewendet auf einen Binärbaum `a` und ein Blatt liefert `true`, wenn `a` ein Blatt ist; `istUnterbaum` angewendet auf eine Astgabel liefert `true`, wenn einer der drei angegebenen Fälle erfüllt ist.

Die funktionale Programmierung ermöglicht es also insbesondere, komplexe Datenstrukturen (im Beispiel Binärbäume) direkt, d.h. ohne Repräsentation mittels verzeigerten Variablen, zu deklarieren und zu benutzen. Besondere Stärken der funktionalen Programmierung sind das Programmieren mit Funktionen höherer Ordnung, ausgefeilte, parametrische Typsysteme (parametrischer Polymorphismus) und flexible Modularisierungskonzepte.

Logische Programmierung. Die logische Programmierung betrachtet ein Programm als eine Ansammlung von Fakten und Folgerungsbeziehungen, an die Anfragen gestellt werden können. Die Ausführung eines logischen Programms sucht Antworten auf solche Anfragen.

Die logische Programmierung bedient sich einer Sprache der formalen Logik, um Fakten und ihre Zusammenhänge zu beschreiben. Beispielsweise könnten wir die Tatsache, dass Sokrates ein Mensch ist, dadurch ausdrücken, dass ein Prädikat `istMensch` für Sokrates gilt. Die Aussage, dass alle Menschen sterblich sind, könnten wir als Folgerung formulieren, indem wir postulieren, dass jeder, der das Prädikat `istMensch` erfüllt, auch das Prädikat `sterblich` erfüllen soll. In der Syntax der Programmiersprache PROLOG erhält man damit folgendes Programm:

```
istMensch( sokrates ).
sterblich(X) :- istMensch(X).
```

Die Ausführung solcher Programme wird über Anfragen ausgelöst. Auf die Anfrage `sterblich( sokrates )?` würde die Programmausführung mit „ja“ antworten. Auf eine Anfrage `sterblich( X )?` sucht die Programmausführung nach allen Termen, die das Prädikat `sterblich` erfüllen. Auf der Basis unseres Programms kann sie dies nur für den Term `sokrates` ableiten. Eine Anfrage `sterblich( kohl )?` würde mit „nein“ beantwortet werden, da die Sterblichkeit von Kohl aus den angegebenen Fakten nicht abgeleitet werden kann.

Zwei weitere Ausprägungen der deklarativen Programmierung seien zumindest erwähnt: die Programmierung mit Constraints und die relationale Programmierung im Zusammenhang mit Datenbanken. Constraint-Programmierung kann als eine Verallgemeinerung der logischen Programmierung begriffen werden, bei der der Zusammenhang von Daten nicht nur mittels Folgerungsbeziehungen formuliert werden kann. In der relationalen Datenbankprogrammierung werden Fakten mittels endlicher Relationen formuliert und gespeichert. Eine Datenbank ist im Wesentlichen eine Menge solcher Relationen (formal gesehen also eine Variable vom Typ „Menge von Relationen“). Mit Abfragesprachen, die auf den Relationenkalkül abgestützt sind, lässt sich der Inhalt von Datenbanken abfragen und modifizieren. Die relationale Datenbankprogrammierung ist sicherlich die ökonomisch mit Abstand bedeutendste Variante der deklarativen Programmierung.

Ein Schwerpunkt der Forschung im Bereich deklarativer Programmierung zielt auf die Integration der unterschiedlichen Formen deklarativer Programmierung ab. So wurde die logische Programmierung zum allgemeinen Constraint-Lösen erweitert, es wurden mehrere Ansätze erarbeitet, funktionale und logische Programmierung zu kombinieren, und es wurden sogenannte deduktive Datenbanken entwickelt, die Techniken der logischen Programmierung für relationale Datenbanken nutzbar machen. Dabei gestaltet sich die Integration der mathematischen Beschreibungskonzepte meist relativ einfach. Probleme macht die Integration der oft impliziten und recht unterschiedlichen Ausführungsmodelle und ihrer Implementierungen.

Frage. Was sind die Schwächen der deklarativen Programmierung in Hinblick auf die drei in Abschn. 1.1.2 skizzierten Anforderungen?

In der deklarativen Programmierung spielen die Ausführungsmodelle eine untergeordnete Rolle. Deshalb ist das Grundmodell der deklarativen Programmierung wenig geeignet, parallele Prozesse der realen Welt zu modellieren, bei denen räumlich verteilte Objekte eine Rolle spielen, die im Laufe der Ausführung erzeugt werden, in Beziehung zu anderen Objekten treten, ihren Zustand ändern und wieder verschwinden. Modellierung und Realisierung von verteilten Prozessen wird in der deklarativen Programmierung vielfach durch spezielle Konstrukte erreicht, zum Beispiel durch Einführung expliziter Kommunikationskanäle, über die Ströme von Daten ausgetauscht werden können. Ähnliches gilt für die Beschreibung nichtterminierender Prozesse oder zeitlich verzahnter Interaktionen zwischen Programmteilen bzw. zwischen dem Programm und dem Benutzer.

Die deklarative Programmierung eignet sich gut für eine hierarchische Strukturierung von Programmen. Für eine Strukturierung in kooperierende Programmteile müssen zwei Nachteile überwunden werden:

1. Deklarative Programmierung geht implizit davon aus, alle Daten „zen-

tral“ verfügbar zu haben, und bietet wenig Hilfestellung, Daten auf Programmteile zu verteilen.

2. Da Programmteile nicht über die Änderung an Datenstrukturen kommunizieren können, müssen tendenziell mehr Daten ausgetauscht werden als in imperativen Programmiermodellen.

In der Literatur geht man davon aus, dass sich deklarative Programme leichter modifizieren lassen als prozedurale Programme. Zwei Argumente sprechen für diese These: Deklarative Programmierung vermeidet von vornherein einige Fehlerquellen der prozeduralen Programmierung (Seiteneffekte, Zeigerprogrammierung, Speicherverwaltung); Schnittstelleneigenschaften deklarativer Programme lassen sich einfacher beschreiben. Andererseits gelten die Anmerkungen zur Anpassbarkeit prozeduraler Programme entsprechend auch für deklarative Programme.

Die deklarative Programmierung stellt für spezifische softwaretechnische Aufgabenstellungen (z.B. Datenbanken) sehr mächtige Programmierkonzepte, -techniken und -werkzeuge zur Verfügung. Sie ermöglicht in vielen Fällen eine sehr kompakte Formulierung von Programmen und eignet sich durch ihre Nähe zu mathematischen Beschreibungsmitteln gut für die Programmentwicklung aus formalen Spezifikationen.

1.2.3 Objektorientierte Programmierung: Das Grundmodell

Objekte haben einen Zustand

Die objektorientierte Programmierung betrachtet eine Programmausführung als ein System kooperierender Objekte. Objekte haben einen eigenen lokalen Zustand. Sie haben eine gewisse Lebensdauer, d.h. sie existieren vor der Programmausführung oder werden während der Programmausführung erzeugt und leben, bis sie gelöscht werden bzw. bis die Programmausführung endet. Objekte empfangen und bearbeiten Nachrichten. Bei der Bearbeitung von Nachrichten kann ein Objekt seinen Zustand ändern, Nachrichten an andere Objekte verschicken, neue Objekte erzeugen und existierende Objekte löschen. Objekte sind grundsätzlich selbständige Ausführungseinheiten, die unabhängig voneinander und parallel arbeiten können[4].

Objekte bearbeiten Nachrichten

Identität

Objekte verhalten sich in mehreren Aspekten wie Gegenstände der materiellen Welt (bei Gegenständen denke man etwa an Autos, Lampen, Telefone, Lebewesen etc.). Insbesondere haben sie eine *Identität*: Ein Objekt kann nicht an zwei Orten gleichzeitig sein; es kann sich ändern, bleibt dabei aber dasselbe Objekt (man denke beispielsweise daran, dass ein Auto umlackiert werden kann, ohne dass sich dabei seine Identität ändert, oder dass bei einem Menschen im Laufe seines Lebens fast alle Zellen ausgetauscht werden, die Identität des Menschen davon aber unberührt bleibt). Objekte im Sinne der

[4]Dieses Modell wird in vielen, in der Praxis eingesetzten Programmiersprachen nicht umgesetzt. Siehe auch Absatz „Relativierung inhärenter Parallelität“ auf Seite 20

objektorientierten Programmierung unterscheiden sich also von üblichen mathematischen Objekten wie Zahlen, Funktionen, Mengen, usw. (Zahlen haben keine Lebensdauer, keinen „Aufenthaltsort“ und keinen Zustand.)

Im Folgenden werden wir das oben skizzierte *objektorientierte Grundmodell* näher erläutern und den Zusammenhang zu den drei in Abschn. 1.1.2 skizzierten Anforderungen diskutieren.

obj.-orient. Grundmodell

Modellierung der realen Welt. Jedes der unterschiedlichen Programmierparadigmen hat sich aus einem speziellen technischen und gedanklichen Hintergrund heraus entwickelt. Die prozedurale Programmierung ist im Wesentlichen aus einer Abstraktion des Rechenmodells entstanden, das heutigen Computern zugrunde liegt. Die deklarative Programmierung basiert auf klassischen Beschreibungsmitteln der Mathematik. Ein grundlegendes Ziel der objektorientierten Programmierung ist es, eine möglichst gute softwaretechnische Modellierung der realen Welt zu unterstützen und damit insbesondere eine gute Integration von realer und softwaretechnischer Welt zu ermöglichen. Die softwaretechnisch realisierte Welt wird oft als *virtuelle* Welt bezeichnet. Das folgende Zitat (vgl. [MMPN93], Kap. 1) gibt eine Idee davon, was mit Modellierung der realen Welt gemeint ist und welche Vorteile sie mit sich bringt:

> „The basic philosophy underlying object-oriented programming is to make the programs as far as possible reflect that part of the reality they are going to treat. It is then often easier to understand and to get an overview of what is described in programs. The reason is that human beings from the outset are used to and trained in the perception of what is going on in the real world. The closer it is possible to use this way of thinking in programming, the easier it is to write and understand programs.“

Anhand eines kleinen Beispiels wollen wir genauer studieren, wie eine Modellierung der realen Welt in einem Programm aussehen kann und was dafür benötigt wird:

Beispiel 1.2.1 *(Modellierung der Gerätesteuerung in einem Haushalt)*
Als fiktive Aufgabe wollen wir ein System konstruieren, dass es uns gestattet, alle elektrischen und elektronischen Geräte in einem Haus zentral von einem Rechner aus zu bedienen. Den Ausschnitt aus der realen Welt, der für eine Aufgabe relevant ist, nennen wir den Aufgabenbereich*. In unserem Fall besteht der Aufgabenbereich also u.a. aus Lampen, CD-Spielern, HiFi-Verstärkern, Telefonen etc. sowie aus den Räumen des Hauses. Geräte und Räume sind* Objekte der realen Welt *. Sie können sich in unterschiedlichen* Zuständen *befinden: Eine Lampe kann an- oder abgeschaltet sein, ein CD-Spieler kann leer sein oder eine CD geladen haben, der Verstärker kann in unterschiedlichen Lautstärken spielen, in einem Raum können sich verschiedene Geräte befinden. In einem objektorientierten Programm entspricht jedem (relevanten)*

Objekt der realen Welt ein Objekt in der virtuellen Welt des Programms. Jedes Objekt hat einen Zustand.

In der realen Welt geschieht die Inspektion und Änderung des Zustands der Objekte auf sehr unterschiedliche Weise. Inspektion: Man schaut nach, ob eine Lampe leuchtet, hört hin, ob der CD-Spieler läuft, sucht nach dem Telefon. Änderungen: Man schaltet Geräte an und aus, stellt sie lauter, legt CD's in den CD-Spieler, stellt neue Lampen auf. In der virtuellen Welt stellen die Objekte Operationen zur Verfügung, um ihren Zustand zu inspizieren und zu verändern. Beispielsweise wird es für Lampen eine Operation geben, mit der festgestellt werden kann, ob die Lampe leuchtet, und eine Operation, um sie an- bzw. auszuschalten. □

Durch die Aufteilung der Verarbeitungsprozesse auf mehrere Objekte ermöglicht das Grundmodell der objektorientierten Programmierung vom Konzept her insbesondere eine natürliche Behandlung von parallelen und verteilten Prozessen der realen Welt.

Relativierung inhärenter Parallelität Das Konzept der inhärenten Parallelität wird von den meisten objektorientierten Programmiersprachen allerdings **nicht** unterstützt. Methoden werden standardmäßig sequentiell ausgeführt, auch über Objektgrenzen hinweg. Parallelität muss in diesen Sprachen, ähnlich wie in imperativen Programmiersprachen, explizit durch Sprachkonstrukte eingeführt werden. In Java gibt es dafür das Konzept der Threads, das in Abschnitt 6.2 ausführlich vorgestellt wird.

Programmstruktur. Objekte bilden eine Einheit aus Daten und den auf ihnen definierten Operationen. Die Gesamtheit der Daten in einem objektorientierten Programm ist auf die einzelnen Objekte verteilt. Auch gibt es keine global arbeitenden Prozeduren bzw. Operationen[5]. Jede Operation gehört zu einem Objekt und lässt sich von außen nur über das Schicken einer Nachricht an dieses Objekt auslösen. Abbildung 1.3 skizziert dieses Modell in vereinfachter Form für zwei Objekte *obj1* und *obj2*. Beide Objekte haben objektlokale Variablen, sogenannte *Attribute*, z.B. hat *obj1* die Attribute `a1` und `a2`. Beide Objekte haben lokale Operationen, sogenannte *Methoden*: *obj1* hat die Methoden `m`, `m1`, `m2`; *obj2* hat die Methoden `m` und `n` (auf die Angabe der Methodenrümpfe wurde verzichtet). Die Kommunikation zwischen Objekten ist in Abb. 1.3 (wie auch im Folgenden) durch einen gepunkteten Pfeil dargestellt: Objekt *obj1* schickt *obj2* die Nachricht `m` mit den Parametern `1618` und `"SS2007"`. Objekt *obj2* führt daraufhin seine Methode `m` aus und liefert (möglicherweise) ein Ergebnis zurück.

Attribut

[5]Einige Programmiersprachen weichen von diesem Grundkonzept ab und bieten sogenannte *Klassenmethoden* an (siehe auch Abschnitt 2.1.4.2). Klassenmethoden ähneln Prozeduren der prozeduralen Programmierung und können unabhängig von einem Objekt aufgerufen werden.

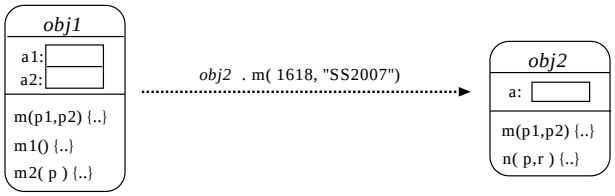

Abbildung 1.3: Kommunizierende Objekte

Das Grundmodell der objektorientierten Programmierung geht demnach davon aus, dass die zu verarbeitenden Informationen auf Objekte verteilt sind. Die Verarbeitung der Information geschieht entweder objektlokal oder durch Nachrichtenkommunikation zwischen Objekten. Dabei werden zur Beschreibung der objektlokalen Verarbeitung meist prozedurale Techniken verwendet.

Jedes Objekt hat eine klar festgelegte Schnittstelle, die seine Eigenschaften beschreibt. Diese Eigenschaften bestehen aus den Nachrichten, die es „versteht“, d.h. für die es eine passende Methode besitzt und aus seinen von außen direkt zugreifbaren Attributen. Idealerweise werden statt direkt zugreifbarer Attribute Zugriffsmethoden verwendet, über die die Attribute gelesen und geschrieben werden können. Diese inhärente Schnittstellenbildung bringt zwei zentrale Fähigkeiten der objektorientierten Programmierung mit sich: Datenkapselung und Klassifizierung. Wenn auf den Zustand eines Objekts von anderen Objekten nur über Methoden zugegriffen wird, hat ein Objekt die vollständige Kontrolle über seine Daten, sofern es nicht von außen Referenzen auf andere Objekte übergeben bekommt, die einen Teil seines Zustandes ausmachen. [6]. Insbesondere kann es die Konsistenz zwischen Attributen gewährleisten und verbergen, welche Daten es in Attributen hält und welche Daten es erst auf Anforderung berechnet.

Objekte können nach ihrer Schnittstelle klassifiziert werden. Beispielsweise könnten die Objekte *obj1* und *obj2* aus Abb. 1.3 zu der Klasse der Objekte zusammengefasst werden, die die Nachricht `m` (mit zwei Parametern) verstehen. Typisches Beispiel wäre eine Klasse von Objekten, die alle eine Methode `drucken` besitzen, um sich selbst auszudrucken. Solche Klassifikationen kann man insbesondere nutzen, um Mengen von Objekten hierarchisch zu strukturieren. Betrachten wir als Beispiel die Personengruppen an einer Universität, skizziert in Abb. 1.4: Jedes Objekt der Klasse Person hat eine Methode, um Namen und Geburtsdatum zu erfragen. Jeder Student kann darüber hinaus nach Matrikelnummer und Semesterzahl befragt werden; bei den Angestellten kommen stattdessen Angaben über das Arbeitsverhältnis und die Zuordnung zu Untergliederungen der Universität hinzu.

[6] Der Zustand solcher Objekte kann dann über weiterhin bestehende Referenzen von außen ebenfalls verändert werden. Auf diese Problematik wird im weiteren Verlauf des Kurses noch eingegangen.

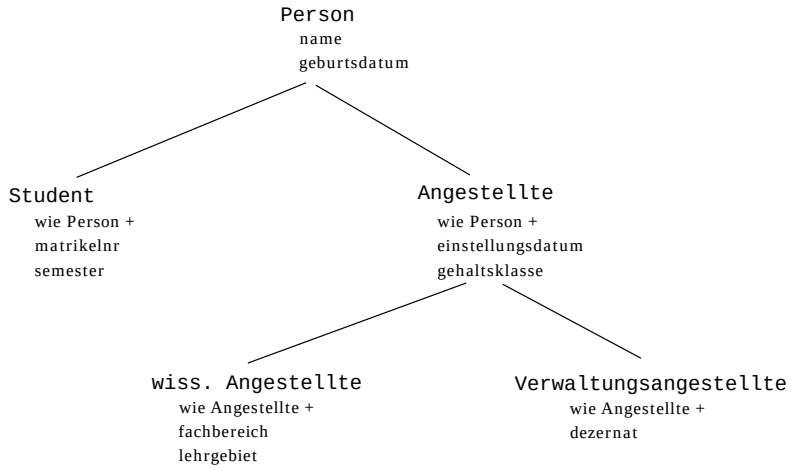

Abbildung 1.4: Klassifikation der Personen an einer Universität

Wie wir im Folgenden sehen werden, lässt sich eine Klassifikation wie in Abb. 1.4 in objektorientierten Programmen direkt modellieren. Für das obige Beispiel heißt das, dass man die Eigenschaften, die allen Personen gemeinsam sind, nur einmal zu beschreiben braucht und sie an alle untergeordneten Personenklassen „vererben“ kann.

Erweiterung von Programmen. Das Nachrichtenmodell in der objektorientierten Programmierung bringt wesentliche Vorteile für eine gute Erweiterbarkeit und Wiederverwendbarkeit von Programmen mit sich. Betrachten wir dazu ein Programm, das Behälter für druckbare Objekte implementiert, d.h. für Objekte, die die Nachricht `drucken` verstehen. Die Behälterobjekte sollen eine Methode `alle_drucken` besitzen, die allen Objekten im Behälter die Nachricht `drucken` schickt. Der Programmtext der Methode `alle_drucken` braucht nicht geändert zu werden, wenn das Programm neue druckbare Objekte mit anderen Eigenschaften unterstützen soll, da jedes druckbare Objekt seine eigene, spezifische Methode zum Drucken besitzt und damit auf die Nachricht `drucken` vom Behälterobjekt reagieren kann.

Um zu sehen, dass derartige Erweiterungseigenschaften nicht selbstverständlich sind, betrachten wir das gleiche Problem im Rahmen der prozeduralen Programmierung. Dazu gehen wir von einem Pascal-Programmfragment aus, das die Daten von Studenten und Angestellten druckt, d.h. die Personen fungieren hier als druckbare Objekte. Als Behälter wählen wir ein Feld mit Elementen vom Typ `Druckbar`. Da die spezifischen Druckprozeduren der Personenarten in Pascal nicht den Personen zugeordnet werden können, muss explizit eine Fallunterscheidung programmiert werden, die für jede Personenart einen Fall mit dem entsprechenden Prozeduraufruf vorsieht:

```
const anz = 100;
```

```
type Student   =  record   ...  end;
     WissAng   =  record   ...  end;
     VerwAng   =  record   ...  end;

     ObjektArt =  ( stud, wiss, verw );
     Druckbar  =  record case  art: ObjektArt  of
                     stud: ( s: Student );
                     wiss: ( w: WissAng );
                     verw: ( v: VerwAng )
                  end;
     Behaelter =  Array [1 .. anz] of Druckbar;

procedure  Student_drucken( s: Student );
  begin  ...  end;

procedure  WissAng_drucken( w: WissAng );
  begin  ...  end;

procedure  VerwAng_drucken( v: VerwAng );
  begin  ...  end;

procedure  alle_drucken( b: Behaelter );
   var e     : Druckbar;
       index : Integer;
begin
   ...
   for index := 1 to anz do
   begin
      e := b[index];
      case  e.art  of
         stud: Student_drucken( e.s );
         wiss: WissAng_drucken( e.w );
         verw: VerwAng_drucken( e.v )
      end
   end  { for-Schleife }
end;
```

Die Entscheidung darüber, welche spezifische Druckprozedur auszuführen ist, wird in der Prozedur `alle_drucken` getroffen. Damit zieht jede Erweiterung des Typs `Druckbar` um neue Objektarten eine Veränderung der Fallunterscheidung in der Prozedur `alle_drucken` nach sich. (Insbesondere muss die Prozedur `alle_drucken` jedes Mal neu übersetzt werden.) In der objektorientierten Programmierung wird diese Fallunterscheidung implizit vom Nachrichtenmechanismus übernommen. Durch den Nachrichtenmechanismus wird die Bindung zwischen der Anforderung eines Dienstes (Verschicken einer Nachricht) und dem ausführenden Programmteil (Methode)

dynamisches / statisches Binden

erst zur Programmlaufzeit getroffen (dynamisch), beim Prozeduraufruf zur Übersetzungszeit (statisch).

Ein weiterer zentraler Aspekt des objektorientierten Grundmodells für die Erweiterung von Programmen resultiert aus der Bündelung von Daten und Operationen zu Objekten mit klar definierten Schnittstellen (siehe Paragraph „Programmstruktur“ auf Seite 20). Ein Objekt beziehungsweise seine Beschreibung lässt sich einfach erweitern und insbesondere leicht spezialisieren. Dazu fügt man zusätzliche Attribute und/oder Methoden hinzu bzw. passt existierende Methoden an neue Erfordernisse an. Insgesamt erhält man Objekte mit einer umfangreicheren Schnittstelle, sodass die erweiterten Objekte auch an allen Stellen eingesetzt werden können, an denen die alten Objekte zulässig waren. Auf diese Möglichkeit wird im Zusammenhang mit *Subtyping* genauer eingegangen (s. Kapitel 3). Der Nachrichtenmechanismus mit seiner dynamischen Bindung garantiert dabei, dass die angepassten Methoden der neuen Objekte ausgeführt werden. Wie wir in späteren Kapiteln genauer untersuchen werden, ermöglicht diese Art der Programmerweiterung eine elegante Spezialisierung existierender Objekte unter Wiederverwendung des Programmcodes der alten Objekte. Diese Art der Wiederverwendung nennt man *Vererbung*: Die spezialisierten Objekte erben den Programmcode der existierenden Objekte.

Subtyping

1.3 Programmiersprachlicher Hintergrund

Dieser Abschnitt stellt den programmiersprachlichen Hintergrund bereit, auf den wir uns bei der Behandlung objektorientierter Sprachkonzepte in den folgenden Kapiteln stützen werden. Er gliedert sich in drei Teile:

1. Zusammenfassung grundlegender Sprachkonzepte von imperativen und objektorientierten Sprachen.
2. Objektorientierte Programmierung mit Java.
3. Überblick über existierende objektorientierte Sprachen.

Der erste Teil bietet darüber hinaus eine Einführung in die Basisdatentypen, Kontrollstrukturen und deren Syntax in Java.

1.3.1 Grundlegende Sprachmittel am Beispiel von Java

Bei den meisten objektorientierten Programmiersprachen werden objektlokale Berechnungen mit imperativen Sprachmitteln beschrieben. Im Folgenden sollen die in den späteren Kapiteln benötigten Sprachmittel systematisch zusammengefasst werden, um eine begriffliche und programmtechnische Grundlage zu schaffen. Begleitend werden wir zeigen, wie diese Sprachmittel in Java umgesetzt sind. Dabei werden wir uns auf eine knapp gehaltene

Einführung beschränken, die aber alle wesentlichen Aspekte anspricht. Eine detaillierte Darstellung findet sich in [GJS96].

Der Abschnitt besteht aus drei Teilen. Er erläutert zunächst den Unterschied zwischen Objekten und Werten. Dann geht er näher auf Werte, Typen und Variablen ein; in diesem Zusammenhang beschreibt er die Basisdatentypen von Java und definiert, was Ausdrücke sind und wie sie in Java ausgewertet werden. Der letzte Teil behandelt Anweisungen und ihre Ausführung. Ziel dieses Abschnitts ist es u.a., den Leser in die Lage zu versetzen, selbständig imperative Programme in Java zu schreiben.

1.3.1.1 Objekte und Werte: Eine begriffliche Abgrenzung

Objekt vs. Wert

Begrifflich unterscheiden wir zwischen *Objekten* und *Werten* (engl. *objects* und *values*). Prototypisch für Objekte sind materielle Gegenstände (Autos, Lebewesen etc.). Prototypische Werte sind Zahlen, Buchstaben, Mengen und (mathematische) Funktionen. Die Begriffe „Objekt“ und „Wert“ sind fundamentaler Natur und lassen sich nicht mittels anderer Begriffe definieren. Wir werden deshalb versuchen, sie durch charakteristische Eigenschaften voneinander abzugrenzen:

1. Zustand: Objekte haben einen veränderbaren Zustand (ein Auto kann eine Beule bekommen; ein Mensch eine neue Frisur; eine Mülltonne kann geleert werden). Werte sind abstrakt und können nicht verändert werden (es macht keinen Sinn, die Zahl 37 verändern zu wollen; entnehme ich einer Menge ein Element, erhalte ich eine andere Menge).

2. Identität: Objekte besitzen eine Identität, die vom Zustand unabhängig ist. Objekte können sich also völlig gleichen, ohne identisch zu sein (man denke etwa an zwei baugleiche Autos). Insbesondere kann man durch Klonen/Kopieren eines Objekts *obj* ein anderes Objekt erzeugen, das *obj* in allen Eigenschaften gleicht, aber nicht mit ihm identisch ist. Zukünftige Änderungen des Zustands von *obj* haben dann keinen Einfluss auf den Zustand der Kopie.

3. Lebensdauer: Objekte besitzen eine Lebensdauer; insbesondere gibt es Operationen, um Objekte zu erzeugen, ggf. auch um sie zu löschen. Werte besitzen keine beschränkte Lebensdauer, sondern existieren quasi ewig.

4. Aufenthaltsort: Objekten kann man üblicherweise einen Aufenthaltsort, beschrieben durch eine Adresse, zuordnen. Werte lassen sich nicht lokalisieren.

5. Verhalten: Objekte reagieren auf Nachrichten und weisen dabei ein zustandsabhängiges Verhalten auf. Werte besitzen kein „Eigenleben“; auf

ihnen operieren Funktionen, die Eingabewerte zu Ergebniswerten in Beziehung setzen.

Der konzeptionell relativ klare Unterschied zwischen Objekten und Werten wird bei vielen programmiersprachlichen Realisierungen nur zum Teil beibehalten. Zur Vereinheitlichung behandelt man Werte häufig wie Objekte. So werden z.B. in Smalltalk ganze Zahlen als Objekte modelliert. Zahl-Objekte besitzen einen unveränderlichen Zustand, der dem Wert der Zahl entspricht, eine Lebensdauer bis zum Ende der Programmlaufzeit und Methoden, die den arithmetischen Operationen entsprechen. Zahl-Objekte werden als identisch betrachtet, wenn sie denselben Wert repräsentieren. In Java wird eine ähnliche Konstruktion für Zeichenreihen verwendet und um zahlwertige Datentypen als Subtypen des ausgezeichneten Typs `Object` behandeln zu können (vgl. Unterabschn. 3.2.2.1). Andererseits lassen sich Objekte auch durch Werte formal modellieren, beispielsweise als ein Paar bestehend aus einem Objektbezeichner (Bezeichner sind Werte) und einem Wert, der den Zustand repräsentiert.

1.3.1.2 Objektreferenzen, Werte, Felder, Typen und Variablen

Dieser Unterabschnitt behandelt den Unterschied zwischen Objekten und Objektreferenzen, beschreibt die Werte und Typen von Java, deren Operationen sowie die Bildung und Auswertung von Ausdrücken.

Objekte und Objektreferenzen. Die Beschreibung und programmtechnische Deklaration von Objekten wird ausführlich in Kap. 2 behandelt. Um präzise erklären zu können, was Variablen in Java enthalten können, müssen wir allerdings schon hier den Unterschied zwischen Objekten und Objektreferenzen erläutern. Abbildung 1.5 zeigt das Objekt *obj* und die Variablen `a`, `b`, `i` und `flag`. Wie in der Abbildung zu sehen, stellen wir Objekte als Rechtecke mit runden Ecken und Variablen als Rechtecke mit rechtwinkligen Ecken dar. Die Variablen `a` und `b` enthalten eine *Referenz* auf *obj*, die graphisch jeweils durch einen Pfeil gezeichnet sind. Während wir Objektreferenzen durch Pfeile repräsentieren, benutzen wir bei anderen Werten die übliche Darstellung. So enthält die int-Variable `i` den Wert 1998 und die boolesche Variable `flag` den Wert `true`.

Objektreferenz

Am besten stellt man sich eine Referenz als eine (abstrakte) Adresse für ein Objekt vor. Konzeptionell spielt es dabei keine Rolle, ob wir unter Adresse eine abstrakte Programmadresse, eine Speicheradresse auf dem lokalen Rechner oder eine Adresse auf einem entfernten Rechner meinen. Lokale Referenzen sind das gleiche wie *Zeiger* in der imperativen Programmierung.

Zeiger

Werte, Typen und Variablen in Java. Ein Datentyp beschreibt eine Menge von Werten zusammen mit den darauf definierten Operationen. Java stellt die

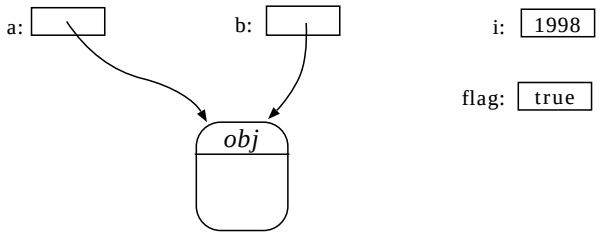

Abbildung 1.5: Referenziertes Objekt und Variablen

vordefinierten Basisdatentypen `byte`, `short`, `int`, `long`, `float`, `double`, `char` und `boolean` zur Verfügung. Wertebereich und der jeweilige Speicherbedarf der Basisdatentypen sind in Abb. 1.6 zusammengestellt.

Basisdatentypen

Typname	Wertebereich und Notation	Speicherbedarf
`byte`	-128 bis 127	1 Byte
`short`	-32768 bis 32767	2 Byte
`int`	-2147483648 bis 2147483647	4 Byte
`long`	-9223372036854775808L bis 9223372036854775807L	8 Byte
`float`	im Bereich ±3.402823E+38F jeweils 6-7 signifikante Stellen	4 Byte
`double`	im Bereich ±1.797693E+308 jeweils 15 signifikante Stellen	8 Byte
`char`	65.536 Unicode-Zeichen, Notationsbeispiele: `'a','+','\n','\'','\u0022'`	2 Byte
`boolean`	`true`, `false`	1 Byte

Abbildung 1.6: Basisdatentypen von Java

Konstante

Abbildung 1.6 zeigt auch, wie die *Konstanten* der Basisdatentypen in Java geschrieben werden. Die Konstanten der Typen `byte`, `short` und `int` werden notationell nicht unterschieden. Die Konstanten des Typs `long` haben ein „`L`“ als Postfix. Bei Gleitkommazahlen kann die Exponentenangabe entfallen. Unicode-Zeichen lassen sich grundsätzlich durch `'\u`*xxxx*`'` in Java ausdrücken, wobei *x* für eine Hexadezimalziffer steht; jedes ASCII-Zeichen *a* lässt sich aber auch direkt als `'`*a*`'` notieren; darüber hinaus bezeichnet `'\t'` das Tabulatorzeichen, `'\n'` das Neue-Zeile-Zeichen, `'\''` das Apostroph, `'\"'` das Anführungszeichen und `'\\'` den Backslash.)

Werte in Java

Ein *Wert* in Java ist entweder

- ein Element eines der Basisdatentypen,
- eine Objektreferenz oder

- die spezielle Referenz `null` , die auf kein Objekt verweist.

Typen in Java

Jeder Wert in Java hat einen *Typ*. Beispielsweise hat der durch die Konstante `true` bezeichnete Wert den Typ `boolean`; die Konstanten `'c'` und `'\n'` bezeichnen Werte vom Typ `char`. Der Typ charakterisiert die Operationen, die auf den Werten des Typs zulässig sind. Außer den vordefinierten Basisdatentypen gibt es in Java Typen für Objekte. Die Typisierung von Objekten behandeln wir in den Kapiteln 2 und 3. Für die hier zusammengestellten Grundlagen ist es nur wichtig, dass jedes Objekt in Java einen Typ hat und dass nicht zwischen dem Typ eines Objekts *obj* und dem Typ der Referenz auf *obj* unterschieden wird[7]. In Java ist vom Programmkontext her immer klar, wann ein Objekt und wann eine Objektreferenz gemeint ist. Deshalb werden auch wir, wie in vielen anderen Texten über Java üblich, ab Kap. 3 nicht mehr zwischen Objekten und Objektreferenzen unterscheiden, um die Sprechweise zu vereinfachen. In diesem Kapitel werden wir weiterhin präzise formulieren; in Kap. 2 werden wir auf den Unterschied durch einen Klammerzusatz aufmerksam machen, wo dies ohne Umstände möglich ist.

Variablen

Variablen sind Speicher für Werte. In Java sind Variablen typisiert. Variablen können nur Werte speichern, die zu ihrem Typ gehören. Eine *Variablendeklaration* legt den Typ und Namen der Variablen fest. Folgendes Programmfragment deklariert die Variablen `i`, `flag`, `a`, `b`, `s1` und `s2`:

```
int  i;
boolean flag;
Object  a;
Object  b;
String  s1, s2;
```

Die Variable `i` kann Zahlen vom Typ `int` speichern; `flag` kann boolesche Werte speichern; `a` und `b` können Referenzen auf Objekte vom Typ `Object`, und `s1` und `s2` können Referenzen auf Objekte vom Typ `String` speichern. Wie wir in Kap. 2 sehen werden, sind `Object` und `String` vordefinierte Objekttypen.

Felder. *Felder* (engl. *arrays*[8]) sind in Java Objekte; dementsprechend werden sie dynamisch, d.h. zur Laufzeit, erzeugt und ihre Referenzen können an Variablen zugewiesen und als Parameter an Methoden übergeben werden. Ist T ein beliebiger Typ in Java, dann bezeichnet T`[]` den Typ der Felder mit *Komponententyp* T. Den Operator `[]` nennt man einen *Typkonstruktor*, da er aus einem beliebigen Typ einen neuen Typ konstruiert.

Komponententyp

Typkonstruktor

[7] In C++ werden Objekte und Objektreferenzen typmäßig unterschieden.

[8] Zur Beachtung: „Arrays“ sind nicht mit „fields“ zu verwechseln. In der engl. Java-Literatur wird das Wort „field“ für die in einer Klasse deklarierten Attribute verwendet.

Jedes Feld(-Objekt) besitzt ein unveränderliches Attribut `length` vom Typ `int` und eine bestimmte Anzahl von Attributen vom Komponententyp. Die Attribute vom Komponententyp nennen wir im Folgenden die *Feldelemente*. Die Anzahl der Feldelemente wird bei der *Erzeugung* des Feldes festgelegt und im Attribut `length` gespeichert. Felder sind in Java also immer eindimensional. Allerdings können die Feldelemente andere Felder referenzieren, sodass mehrdimensionale Felder als Felder von Feldern realisiert werden können.

Feldelement

Felder als Objekte zu realisieren hat den Vorteil, dass sich Felder auf diese Weise besser in objektorientierte Klassenhierarchien einbetten lassen (Genaueres dazu in Kap. 3). Außerdem gewinnt man an Flexibilität, wenn man zwei- bzw. mehrdimensionale Felder realisieren möchte. In zweidimensionalen Feldern, wie man sie beispielsweise in Pascal deklarieren kann, müssen alle Spalten bzw. alle Zeilen die gleiche Länge haben. Dies gilt nicht bei der Java-Realisierung mittels Referenzen. Wenn die von einem Feld referenzierten Felder nicht alle gleich lang sind, spricht man auch von *Ragged Arrays*.

Auch kann man mehrfach auftretende „Spalten“ bzw. „Zeilen“ durch ein mehrfach referenziertes Feld realisieren (vgl. Abbildung 1.8). Der Preis für die größere Flexibilität ist im Allgemeinen größerer Speicherbedarf, höhere Zugriffszeiten sowie eine etwas gestiegene Programmierkomplexität und damit Fehleranfälligkeit.

Folgendes Programmfragment deklariert Variablen für zwei eindimensionale Felder `vor` und `Fliegen` und ein zweidimensionales Feld `satz`:

```
char[] vor;
char[] Fliegen;
char[][] satz;
```

(Um syntaktische Kompatibilität zur Sprache C zu erreichen, darf der Typkonstruktor `[]` auch erst nach dem deklarierten Namen geschrieben werden, also `char vor[]` statt `char[]vor`.) Obige Variablendeklarationen legen lediglich fest, dass die Variablen `vor` und `Fliegen` jeweils eine Referenz auf ein Feld (beliebiger Länge) enthalten können, deren Elemente vom Typ `char` sind und dass `satz` eine Referenz auf ein Feld mit Komponententyp `char[]` enthalten kann; d.h. die Feldelemente können (Referenzen auf) char-Felder speichern. Die Anzahl der Feldelemente wird in diesen Deklarationen noch nicht festgelegt. Wie eine solche Festlegung erfolgt, wird im Rahmen von Zuweisungen (Abbildung 1.7) erläutert.

Operationen, Zuweisungen und Auswertung in Java. Java stellt drei Arten von Operationen zur Verfügung:

1. Operationen, die auf allen Typen definiert sind. Dazu gehören die Gleichheitsoperation == und die Ungleichheitsoperation != mit booleschem Ergebnis sowie die *Zuweisungsoperation* = .

Zuweisung

2. Operationen, die nur für Objektreferenzen bzw. Objekte definiert sind (Methodenaufruf, Objekterzeugung, Typtest von Objekten); diese werden wir in Kap. 2 behandeln.

3. Operationen zum Rechnen mit den Werten der Basisdatentypen; diese sind in Abb. 1.9 zusammengefasst.

Zuweisungen In Java (wie in C) ist die Zuweisung eine Operation mit Seiteneffekt: Sie weist der Variablen auf der linken Seite des Zuweisungszeichens „=“ den Wert des Ausdrucks der rechten Seite zu und liefert diesen Wert als Ergebnis der gesamten Zuweisung zurück. Dazu betrachten wir diverse Beispiele aufbauend auf den Variablendeklarationen von S. 28 und 29; der Ergebniswert ist teilweise hinter dem *Kommentarzeichen* `//` angegeben, das den Rest der Zeile als Kommentar kennzeichnet:

Kommentarzeichen

```
(1)  i = 4;              // nach Auswertung: i==4   Ergebnis: 4
(2)  flag = (5 != 3);    // nach Ausw.: flag==true  Ergebnis: true
(3)  flag = (5 != (i=3));   //  nach Ausw.: i==3,  flag==true
                            //  Ergebnis: true
(4)  a = (b = null);     /* nach Ausw.:  a==null, b==null
                            Ergebnis: null */
(5)  vor = new char[3];
(6)  vor[0] = 'v';
(7)  vor[1] = 'o';
(8)  vor[2] = 'r';
(9)  char[] Fliegen = {'F','l','i','e','g','e','n'};
(10) char[][] satz = { Fliegen,
                {'f','l','i','e','g','e','n'},
                vor,
                Fliegen };
```

Abbildung 1.7: Zuweisungen

Kommentarklammern

Die mit (4) markierte Zeile demonstriert außerdem, dass in Java auch die *Kommentarklammern* `/*` und `*/` benutzt werden können.

Die Zeilen (5) - (10) zeigen alle im Zusammenhang mit Feldern relevanten Operationen. In Zeile (5) wird die Variable `vor` mit einem Feld der Länge 3 initialisiert. In den Zeilen (6)-(8) werden die drei Feldelemente initialisiert; man beachte, dass die Feldelemente von 0 bis $length - 1$ indiziert werden. In Zeile (9) wird die Variable `Fliegen` deklariert und mit einem Feld der Länge 7 initialisiert; die Länge berechnet der Übersetzer dabei aus der Länge der angegebenen Liste von Ausdrücken (in dem Beispiel ist jeder Ausdruck eine char-Konstante). Das Initialisieren von Arrays in dieser Form mit geschweiften Klammern ohne explizite Angabe der Arraylänge ist nur bei gleichzeitiger

Deklaration der Variable gestattet. Die schon auf S. 29 gesehene Deklaration darf erst hier erstmalig auftreten. In Zeile (10) wird die Variable `satz` deklariert und mit einem vierelementigen Feld initialisiert,

- deren erstes und viertes Element mit dem in Zeile (9) erzeugten Feld initialisiert wird,
- deren zweites Element mit einem neu erzeugten Feld initialisiert wird
- und deren drittes Element mit dem von `vor` referenzierten Feld initialisiert wird.

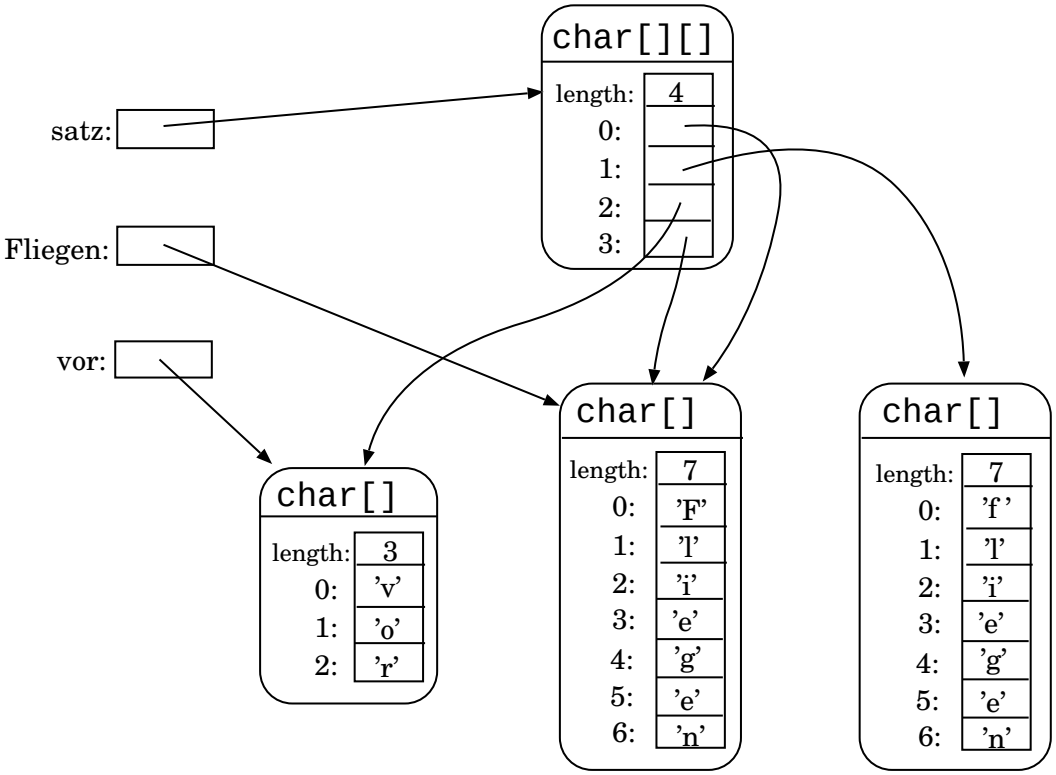

Abbildung 1.8: Die Feldobjekte zum Beispiel

Das resultierende *Objektgeflecht* ist in Abb. 1.8 dargestellt.

Operationen auf Basisdatentypen Die Operationen auf Basisdatentypen sind in der folgenden Tabelle zusammengefasst.

Operator	Argumenttypen	Ergebnistyp	Beschreibung
`+, -, *, /, %`	`int` × `int`	`int`	ganzzahlige Addition, etc.
`+, -, *, /, %`	`long` × `long`	`long`	ganzzahlige Addition, etc., wobei `%` den Rest bei ganzzahliger Division liefert
`+, -, *, /`	`float` × `float`	`float`	Gleitkomma-Addition, etc.
`+, -, *, /`	`double` × `double`	`double`	Gleitkomma-Addition, etc.
`-`	*zahltyp*	*zahltyp*	arithmetische Negation
`<, <=, >, >=`	*zahltyp* × *zahltyp*	`boolean`	kleiner, kleiner-gleich, etc., wobei *zahltyp* für `int`, `long`, `float` oder `double` steht
`!`	`boolean`	`boolean`	logisches Komplement
`&, |, ^`	`boolean` × `boolean`	`boolean`	logische Operationen Und, Oder und ausschließendes Oder (xor)
`&&, ||`	`boolean` × `boolean`	`boolean`	nicht-strikte Und-/Oder-Operation, dh. rechter Operand wird ggf. nicht ausgewertet
`_?_:_`	`boolean` × *typ* × *typ*	*typ*	bedingter Ausdruck (Bedeutung auf Seite 33 erläutert)

Abbildung 1.9: Operationen der Basisdatentypen

Die obige Abbildung stellt nicht alle Operationen in Java dar: Der `+`-Operator im Zusammenhang mit Objekten des Typs `String` wird in Kap. 2 erläutert; Operationen, die auf der Bitdarstellung von Zahlen arbeiten, sowie bestimmte Formen der Zuweisung und Operatoren zum Inkrementieren und Dekrementieren haben wir der Kürze halber weggelassen.

Ausdrücke

Ausdrücke und deren Auswertung Ein *Ausdruck* (engl. *expression*) ist eine Variable, eine Konstante oder eine Operation angewendet auf Ausdrücke. Wie üblich werden Klammern verwendet, um die Reihenfolge der anzuwendenden Operationen eindeutig zum Ausdruck zu bringen. Außerdem werden, um Klammern einzusparen, gewisse Vorrangregeln beachtet. Z. B. hat `*` Vorrang vor `+` („Punktrechnung vor Strichrechnung“) und alle arithmetischen und logischen Operatoren haben Vorrang vor dem Zuweisungsoperator `=`. Jeder Ausdruck in Java besitzt einen Typ, der sich bei Variablen aus deren Deklaration ablesen lässt, der sich bei Konstanten aus der Notation ergibt und der bei Operationen derem Ergebnistyp entspricht. Grundsätzlich gilt, dass die Typen der Operanden genau den Argumenttypen der Operationen entsprechen müssen. Um derartige Typgleichheit erzielen zu können, bietet Java die Möglichkeit, Werte eines Typs in Werte eines anderen Typs

zu konvertieren. Häufig spricht man anstatt von *Typkonvertierung* auch von Typecasts oder einfach nur von Casts (vom engl. to cast).

Typkonvertierung (cast)

In Java werden Typkonvertierungen dadurch notiert, dass man den Namen des gewünschten Ergebnistyps, in Klammern gesetzt, dem Wert bzw. Ausdruck voran stellt. Beispielsweise bezeichnet `(long)3` den Wert drei vom Typ `long`, ist also gleichbedeutend mit `3L`. Java unterstützt insbesondere die Typkonvertierung zwischen allen Zahltypen, wobei der Typ `char` als Zahltyp mit Wertebereich 0 bis 65.536 betrachtet wird. Vergrößert sich bei der Typkonvertierung der Wertebereich, z.B. von `short` nach `long`, bleibt der Zahlwert unverändert. Andernfalls, z.B. von `int` nach `byte`, führt die Typkonvertierung im Allgemeinen zu einer Verstümmelung des Wertes. Um die Lesbarkeit der Programme zu erhöhen, werden bestimmte Konvertierungen in Java implizit vollzogen: Z.B. werden alle ganzzahligen Typen, wo nötig, in ganzzahlige Typen mit größerem Wertebereich konvertiert und ganzzahlige Typen werden, wo nötig, in Gleitkommatypen konvertiert. Die folgenden beiden Beispiele demonstrieren diese implizite Typkonvertierung, links jeweils der Java-Ausdruck, bei dem implizit konvertiert wird, rechts ist die Konvertierung explizit angegeben:

```
585888 * 3L                ((long) 585888) * 3L
3.6 + 45L                  3.6 + ((double) 45L)
```

Man beachte, dass auch implizite Konvertierungen zur Verstümmelung der Werte führen können (beispielsweise bei der Konvertierung von großen long-Werten nach `float`.

Auswertung von Ausdrücken

Die *Auswertung* (engl. *evaluation*) eines Ausdrucks ist über dessen Aufbau definiert. Ist der Ausdruck eine Konstante, liefert die Auswertung den Wert der Konstanten. Ist der Ausdruck eine Variable, liefert die Auswertung den Wert, der in der Variablen gespeichert ist. Besteht der Ausdruck aus einer Operation angewendet auf Unterausdrücke, gilt grundsätzlich, dass zuerst die Unterausdrücke von links nach rechts ausgewertet werden und dann die Operation auf die Ergebnisse angewandt wird (*strikte* Auswertung). Abweichend davon wird bei den booleschen Operationen `&&` und `||` der rechte Operand nicht mehr ausgewertet, wenn die Auswertung des linken Operanden `false` bzw. `true` ergibt, da in diesen Fällen das Ergebnis des gesamten Ausdrucks bereits feststeht (*nicht-strikte* Auswertung). Entsprechendes gilt für den bedingten Ausdruck: Zur Auswertung von

```
B ? A1 : A2
```

werte zunächst den booleschen Ausdruck `B` aus. Wenn er `true` ergibt, werte `A1` aus und liefere dessen Ergebnis, ansonsten `A2`.

normale und abrupte Terminierung der Auswertung

Die Auswertung eines Ausdrucks kann in Java auf zwei Arten terminieren: *normal* mit dem üblichen Ergebnis oder *abrupt*, wenn bei der Ausführung einer Operation innerhalb des Ausdrucks ein Fehler auftritt – z.B. Division durch null. Im Fehlerfall wird die Ausführung des gesamten umfassenden

Ausdrucks sofort beendet und eine Referenz auf ein Objekt zurückgeliefert, das Informationen über den Fehler bereitstellt (vgl. den folgenden Abschnitt über Kontrollstrukturen sowie die Kapitel 2 und 4).

1.3.1.3 Anweisungen, Blöcke und deren Ausführung

Anweisungen dienen dazu, den Kontrollfluss von Programmen zu definieren. Während Ausdrücke ausge*wert*et werden – üblicherweise mit Rückgabe eines Ergebniswertes –, spricht man bei Anweisungen von *Ausführung* (engl. *execution*). Dieser Unterabschnitt stellt die wichtigsten Anweisungen von Java vor und zeigt, wie aus Deklarationen und Anweisungen Blöcke gebildet werden können. Wir unterscheiden *elementare* und *zusammengesetzte* Anweisungen.

Die zentrale elementare Anweisung in Java ist ein Ausdruck. Wie oben erläutert, können durch Ausdrücke in Java sowohl Zuweisungen, als auch Methodenaufrufe beschrieben werden. Insofern stellen Java-Ausdrücke[9] eine syntaktische Verallgemeinerung der üblichen elementaren Anweisungsformen „Zuweisung“ und „Prozeduraufruf“ dar. Ausdrücke, die als Anweisung fungieren, werden mit einem Semikolon abgeschlossen; dazu drei Beispiele:

Ausdrücke als Anweisung

```
i = 3;
flag = ( i >= 2 );
a.toString();
```

Der int-Variablen `i` wird 3 zugewiesen. Der booleschen Variable `flag` wird das Ergebnis des Vergleichs „`i` größer gleich `2`“ zugewiesen. Für das von der Variablen `a` referenzierte Objekt wird die Methode `toString` aufgerufen (gleichbedeutend dazu: Dem von der Variablen `a` referenzierten Objekt wird die Nachricht `toString` geschickt); genauer werden wir auf den Methodenaufruf in Kap. 2 eingehen.

Blöcke

Eine in geschweifte Klammern eingeschlossene Sequenz von Variablendeklarationen und Anweisungen heißt in Java ein *Block* oder *Anweisungsblock*. Ein Block ist eine zusammengesetzte Anweisung. Das folgende Beispiel demonstriert insbesondere, dass Variablendeklarationen und Anweisungen gemischt werden dürfen:

```
{
   int i;
   Object a;
   i = 3;
   boolean flag;
   flag = ( i >= 2 );
   a.toString();
}
```

[9] Gleiches gilt für Ausdrücke in C oder C++.

Variablendeklarationen lassen sich mit einer nachfolgenden Zuweisung zusammenfassen. Beispielsweise könnte man die Deklaration von `flag` mit der Zuweisung wie folgt verschmelzen:

```
boolean flag = ( i >= 2 );.
```

Variablendeklarationen in Blöcken sind ab der Deklarationsstelle bis zum Ende des Blockes gültig.

Klassische Kontrollstrukturen. *Bedingte Anweisungen* gibt es in Java in den Formen:

`if (` *boolescher_Ausdruck* `)` *Anweisung*
`if (` *boolescher_Ausdruck* `)` *Anweisung1* `else` *Anweisung2*

wobei die Zweige der if-Anweisung selbstverständlich wieder zusammengesetzte Anweisungen sein können. Bei der ersten Form wird *Anweisung* nur ausgeführt, wenn der boolsche Ausdruck zu `true` auswertet. Im zweiten Fall wird *Anweisung1* ausgeführt, wenn der boolsche Ausdruck zu `true` auswertet, andernfalls wird *Anweisung2* ausgeführt. Im folgenden Beispiel wird die erste Form der `if`-Anweisung in Zeile 2 verwendet und die zweite Form in Zeile 4.

bedingte Anweisung

```
(1)   int  n = 4;
(2)   if ((n % 2) == 0)
        System.out.println ("n ist durch 2 teilbar");
(3)   n = n + 2;
(4)   if ((n % 3) == 0)
        System.out.println ("n ist durch 3 teilbar");
      else
        System.out.println ("n ist NICHT durch 3 teilbar");
```

Dabei gibt `System.out.println(...)` den als Parameter übergebenen Wert auf der Standardausgabe an der aktuellen Position aus und macht anschließend noch einen Zeilenvorschub.

Für das Programmieren von Schleifen bietet Java die while- und die do-Anweisung sowie zwei Formen der for-Anweisung mit folgender Syntax:

`while (` *boolescher_Ausdruck* `)` *Anweisung*
`do` *Anweisung* `while (` *boolescher_Ausdruck* `) ;`
`for (` *Init-Ausdruck* `;` *boolescher_Ausdruck* `;` *Ausdruck* `)` *Anweisung*
`for (` *Variablendeklaration* `:` *Ausdruck* `)` *Anweisung*

Bei der while-Anweisung wird zuerst der boolesche Ausdruck ausgewertet; liefert er den Wert *true*, wird die Anweisung im Rumpf der Schleife ausgeführt und die Ausführung der while-Anweisung beginnt von neuem. Andernfalls wird die Ausführung nach der while-Anweisung fortgesetzt.

Schleifen-Anweisung

Die do-Anweisung unterscheidet sich nur dadurch, dass der Schleifenrumpf auf jeden Fall einmal vor der ersten Auswertung des booleschen Ausdrucks ausgeführt wird. Die do-Anweisung wird beendet, sobald der boolesche Ausdruck zu `false` auswertet.

Die Ausführung der for-Schleife in der ersten Form kann mit Hilfe der while-Schleife erklärt werden; dazu betrachten wir ein kleines Programmfragment, mit dem die Fakultät berechnet werden kann (bei Eingaben größer als 20 tritt allerdings ein Überlauf auf, sodass das Programm nur im Bereich 0 bis 20 korrekt arbeitet):

```
(1)   int  n;      // Eingabeparameter
(2)   long result; // Ergebnis
(3)   int  i;      // Schleifenvariable

(4)   n =  19;     // initialisieren mit Wert von 0 bis 20
(5)   result = 1;
(6)   for( i = 2; i <= n; i = i + 1 )  result = result * i;
(7)   // result enthaelt  fac(n)
```

Äquivalent zu der for-Schleife in Zeile (6) ist folgendes Fragment:

```
i = 2;
while( i <= n ) {
   result =  result * i;
   i = i + 1;
}
```

Die zweite Form der `for`-Anweisung, auch *for-each* genannt, erlaubt das einfache Iterieren über Felder oder Objekte, die vom Typ `java.lang.Iterable` sind. Wir betrachten hier zunächst nur das Iterieren über Felder. Statt über den Index der Feldelemente zu iterieren wie in der ersten Form, wird in der zweiten Form über die Feldelemente selbst iteriert.

```
String[] satz = { "Wenn", "Fliegen", "hinter", "Fliegen",
                  "fliegen", ",", "fliegen", "Fliegen",
                  "Fliegen", "nach", "." };
System.out.println();

// Lesen Sie: Fuer jedes (for each) 'wort' in 'satz'
for( String wort : satz ) {
  System.out.print(wort);
  System.out.print(" ");
}
System.out.println();
```

Ähnlich `System.out.println(...)` gibt `System.out.print(...)` den als Parameter übergebenen Wert auf der Standardausgabe an der aktuellen Position aus. Allerdings wird kein Zeilenvorschub vorgenommen.

Dasselbe Verhalten kann wie folgt mit der ersten Form der `for`-Schleife erreicht werden:

```
String[] satz = { "Wenn", "Fliegen", "hinter", "Fliegen",
                  "fliegen", ",", "fliegen", "Fliegen",
                  "Fliegen", "nach", "." };
System.out.println();
for( int i = 0; i < satz.length; i = i + 1 ) {
  System.out.print(satz[i]);
  System.out.print(" ");
}
System.out.println();
```

switch-Anweisung

Für die effiziente Implementierung von Fallunterscheidungen bietet Java die switch-Anweisung. Diese soll hier nur exemplarisch erläutert werden. Nehmen wir an, dass wir den Wert einer int-Variablen `n`, von der wir annehmen, dass sie nur Werte zwischen 0 und 11 annimmt, als Zahlwort ausgeben wollen, also für 0 das Wort „null“ usw. Ein typischer Fall für die switch-Anweisung:

```
switch ( n ) {
case 0:  System.out.print("null");
         break;
case 1:  System.out.print("eins");
         break;
...
case 11: System.out.print("elf");
         break;
default: System.out.print("n nicht zwischen 0 und 11");
}
```

Zur Ausführung der switch-Anweisung wird zunächst der ganzzahlige Ausdruck hinter dem Schlüsselwort `switch` ausgewertet, in obigem Fall wird also der Wert von `n` genommen. Sofern kein Fall für diesen Wert angegeben ist, wird die Ausführung nach dem Schlüsselwort `default` fortgesetzt. Andernfalls wird die Ausführung bei dem entsprechenden Fall fortgesetzt. Die break-Anweisungen im obigen Beispiel beenden die Ausführung der einzelnen Fälle und brechen die switch-Anweisung ab; die Ausführung wird dann mit der Anweisung fortgesetzt, die der switch-Anweisung folgt. Fehlten die break-Anweisungen im Beispiel, würden alle Zahlworte mit Werten größer gleich `n` sowie die Meldung des default-Falls ausgegeben werden. Im Allgemeinen dient die break-Anweisung dazu, die umfassende Anweisung direkt

break-Anweisung

zu verlassen; insbesondere kann sie auch zum Herausspringen aus Schleifen benutzt werden.

return-Anweisung

Entsprechend kann man mittels der return-Anweisung die Ausführung eines Methodenrumpfs beenden. Syntaktisch tritt die return-Anweisung in zwei Varianten auf, je nachdem ob die zugehörige Methode ein Ergebnis zurückliefert oder nicht:

```
return ;    // in Methoden ohne Ergebnis und in Konstruktoren
return Ausdruck ;  // Wert des Ausdrucks liefert das Ergebnis
```

normale und abrupte Terminierung der Ausführung

Abfangen von Ausnahmen. Ebenso wie die Auswertung von Ausdrücken kann auch die Ausführung einer Anweisung normal oder abrupt terminieren. Bisher haben wir uns nur mit normaler Ausführung beschäftigt. Wie geht die Programmausführung aber weiter, wenn die Auswertung eines Ausdrucks oder die Ausführung einer Anweisung abrupt terminiert? Wird dann die Programmausführung vollständig abgebrochen? Die Antwort auf diese Fragen hängt von der betrachteten Programmiersprache ab. In Java gibt es spezielle Sprachkonstrukte, um abrupte Terminierung und damit *Ausnahmesituationen* zu behandeln: Mit der try-Anweisung kann der Programmierer aufgetretene Ausnahmen kontrollieren, mit der throw-Anweisung kann er selber eine abrupte Terminierung herbeiführen und damit eine Ausnahmebehandlung anstoßen.

try-Anweisung

Eine *try-Anweisung* dient dazu, Ausnahmen, die in einem Block auftreten, abzufangen und je nach dem Typ der Ausnahme zu behandeln. Die try-Anweisung hat folgende syntaktische Form:

```
try
  try-Block
catch ( AusnahmeTyp   Bezeichner )   catch-Block_1
  ...
catch ( AusnahmeTyp   Bezeichner )   catch-Block_N
finally  finally-Block
```

Die finally-Klausel ist optional; die try-Anweisung muss allerdings immer entweder mindestens eine catch-Klausel oder die finally-Klausel enthalten. Bevor wir die Bedeutung der try-Anweisung genauer erläutern, betrachten wir ein kleines Beispiel. In der in Java vordefinierten Klasse `Integer` gibt es eine Methode `parseInt`, die eine Zeichenreihe als Parameter bekommt (genauer: die Methode bekommt eine Referenz auf ein String-Objekt als Parameter). Stellt die Zeichenreihe eine int-Konstante dar, terminiert die Methode normal und liefert den entsprechenden int-Wert als Ergebnis. Andernfalls terminiert sie abrupt und liefert ein Ausnahmeobjekt vom Typ `NumberFormatException`.

Abbildung 1.10 zeigt an einem einfachen Beispiel, wie eine solche Ausnahme behandelt werden kann.

```
int m;
String str = "007L";
try {
  m = Integer.parseInt( str );
}
catch ( NumberFormatException e ) {
  System.out.println("str keine int-Konstante");
  m = 0;
}
System.out.println( m );
```

Abbildung 1.10: Programmfragment zur Behandlung einer Ausnahme

Da `parseInt` den Postfix „L“ beim eingegebenen Parameter nicht akzeptiert, wird die Ausführung des Methodenaufrufs innerhalb des try-Blocks abrupt terminieren und eine Ausnahme vom Typ `NumberFormatException` erzeugen. Dies führt zur Ausführung der angegebenen catch-Klausel. Danach terminiert die gesamte try-Anweisung normal, sodass die Ausführung mit dem Aufruf von `println` in der letzten Zeile fortgesetzt wird.

Programme werden schnell unübersichtlich, wenn für jede elementare Anweisung, die möglicherweise eine Ausnahme erzeugt, eine eigene try-Anweisung programmiert wird. Stilistisch bessere Programme erhält man, wenn man die Ausnahmebehandlung am Ende größerer Programmteile zusammenfasst. Dies soll mit folgendem Programmfragment illustriert werden, das seine beiden Argumente in int-Werte umwandelt und deren Quotienten berechnet:

```
String[] args = ...;

int m, n, ergebnis = 0;
try{
     m = Integer.parseInt( args[0] );
     n = Integer.parseInt( args[1] );
     ergebnis =  m/n ;
} catch ( IndexOutOfBoundsException e ) {
  System.out.println("argf falsch initialisiert");
} catch ( NumberFormatException e ) {
  System.out.println(
    "Element in argf keine int-Konstante");
} catch ( ArithmeticException e ) {
  System.out.println("Divison durch null");
}
System.out.println( ergebnis );
...
```

Eine Ausnahme vom Typ `IndexOutOfBoundsException` tritt bei Feldzugriffen mit zu großem oder zu kleinem Index auf. Genau wie die NumberFormat-Ausnahme kann solch ein falscher Feldzugriff innerhalb des try-Blocks zweimal vorkommen. Die Ausnahme vom Typ `ArithmeticException` fängt die mögliche Division durch null in der letzten Zeile des try-Blocks ab.

Im Allgemeinen ist die Ausführungssemantik für try-Anweisungen relativ komplex, da auch in den catch-Blöcken und dem finally-Block Ausnahmen auftreten können. Der Einfachheit halber gehen wir aber davon aus, dass die Ausführung der catch-Blöcke und des finally-Blocks normal terminiert. Unter dieser Annahme lässt sich die Ausführung einer try-Anweisung wie folgt zusammenfassen. Führe zunächst den try-Block aus:

- Terminiert seine Ausführung normal, führe den finally-Block aus; die Ausführung der gesamten try-Anweisung terminiert in diesem Fall normal.
- Terminiert seine Ausführung abrupt mit einer Ausnahme *Exc*, suche nach der ersten zu *Exc* passenden catch-Klausel:
  - Wenn es eine passende catch-Klausel gibt, führe den zugehörigen Block aus und danach den finally-Block; die Ausführung der gesamten try-Anweisung terminiert in diesem Fall normal, d.h. die im try-Block aufgetretene Ausnahme wurde abgefangen und die Ausführung wird hinter der try-Anweisung fortgesetzt.
  - Wenn es keine passende catch-Klausel gibt, führe den finally-Block aus; die Ausführung der gesamten try-Anweisung terminiert in diesem Fall abrupt mit der Ausnahme *Exc*. Die Fortsetzung der Ausführung hängt dann vom Kontext der try-Anweisung ab: Ist sie in einer anderen try-Anweisung enthalten, übernimmt diese die Behandlung der Ausnahme *Exc*; gibt es keine umfassende try-Anweisung, terminiert die umfassende Methode abrupt mit der Ausnahme *Exc*.

Dabei *passt* eine `catch`-Klausel zu einer geworfenen Ausnahme, wenn die Klasse, zu der das erzeugte Ausnahmeobjekt gehört, identisch mit oder eine untergeordnete Klasse der Klasse ist, die zur Deklaration der zu fangenden Ausnahmen innerhalb der `catch`-Klausel verwendet wurde[10].

Die in den Beispielen vorkommenden Klassen `ArithmeticException`, `NumberFormatException` und `IndexOutOfBoundsException` sind alle untergeordnete Klassen von `RuntimeException`, die selbst wiederum der Klasse `Exception` untergeordnet ist. Die in Java vordefinierte Klassenhierarchie für Ausnahmetypen wird in Abschnitt 4.2.1 genauer eingeführt.

[10] In späteren Kapiteln, insbesondere in Abschnitt 4.2.2, werden wir hierauf noch genauer eingehen.

Als letzte Anweisung behandeln wir die *throw-Anweisung*. Sie hat syntaktisch die Form: *throw-Anweisung*

`throw` *Ausdruck* `;`

Die throw-Anweisung terminiert immer abrupt. Der Ausdruck muss zu einem Ausnahmeobjekt auswerten. Als Beispiel nehmen wir an, dass wir eine Ausnahmebehandlung für Überläufe bei int-Additionen entwickeln möchten (in Java werden solche Überläufe normalerweise nicht behandelt).

Dazu deklarieren wir eine Ausnahmeklasse `Ueberlauf`. Alle selbstdefinierten Ausnahmeklassen müssen in Java direkt oder indirekt der Klasse `Exception` untergeordnet sein. Daher definieren wir die Klasse `Ueberlauf` wie folgt:

```
class Ueberlauf extends Exception {}
```

`class Ueberlauf` leitet dabei die Deklaration der Klasse `Ueberlauf` ein. `extends Exception` bedeutet, dass `Ueberlauf` als eine `Exception` untergeordnete Klasse deklariert wird. `Ueberlauf` erbt dadurch alle Methoden und Attribute von Exception. Die Deklaration eigener Ausnahmetypen wird in Kap. 4 genauer behandelt. Der in Zeile 8 der Abbildung 1.11 vorkommende Ausdruck `new Ueberlauf()` erzeugt ein Ausnahmeobjekt vom Typ `Ueberlauf`. Um die Anwendung der throw-Anweisung zu demonstrieren, haben wir im obigen Programmfragment die Division durch eine Addition im Typ `long` ersetzt. Abbildung 1.11 zeigt das Resultat.

```
(1)  String[] argf = {"2147483640", "3450336"};
(2)  long maxint = 2147483647L;
(3)  try {
(4)    int m, n, ergebnis ;
(5)    m = Integer.parseInt( argf[0] );
(6)    n = Integer.parseInt( argf[1] );
(7)    long aux = (long)m + (long)n;
(8)    if( aux > maxint ) throw  new Ueberlauf();
(9)    ergebnis =  (int)aux ;
(10) } catch ( IndexOutOfBoundsException e ) {
(11)   System.out.println("argf falsch initialisiert");
(12) } catch ( NumberFormatException e ) {
(13)   System.out.println(
(14)     "Element in argf keine int-Konstante");
(15) } catch ( Ueberlauf e ) {
(16)   System.out.println("Ueberlauf aufgetreten");
(17) }
```

Abbildung 1.11: Demonstration der throw-Anweisung

Insgesamt kann der Programmierer durch geeigneten Einsatz der try- und der throw-Anweisung verbunden mit der Definition eigener Ausnahmetypen eine flexible Ausnahmebehandlung realisieren.

1.3.2 Objektorientierte Programmierung mit Java

Dieser Abschnitt führt in die programmiersprachliche Realisierung objektorientierter Konzepte ein. Anhand eines kleinen Beispiels demonstriert er, wie die zentralen Aspekte des objektorientierten Grundmodells, das in Abschn. 1.2.3 vorgestellt wurde, mit Hilfe von Java umgesetzt werden können. Insgesamt verfolgt dieser Abschnitt die folgenden Ziele:

1. Das objektorientierte Grundmodell soll zusammengefasst und konkretisiert werden.

2. Die Benutzung objektorientierter Konzepte im Rahmen der Programmierung mit Java soll demonstriert werden.

3. Die Einführung in die Sprache Java soll fortgesetzt werden.

Das verwendete Programmierbeispiel lehnt sich an die Personenstruktur von Abb. 1.4 an (s. S. 22). Wir definieren Personen- und Studenten-Objekte und statten sie zur Illustration mit einfachen Methoden aus. Wir zeigen, wie das Erzeugen von Objekten realisiert und wie mit Objekten programmiert werden kann.

1.3.2.1 Objekte, Klassen, Methoden, Konstruktoren

Objekte besitzen einen Zustand und Methoden, mit denen sie auf Nachrichten reagieren (vgl. Abb. 1.3, S. 21). Programmtechnisch wird der Zustand durch mehrere Attribute realisiert, also objektlokale Variablen. Solche objektlokalen Variablen werden in Java im Rahmen von Klassen deklariert. Bei der Erzeugung eines Objekts werden diese Variablen dann initialisiert. Im Gegensatz zu z.B. C++ speichern Attribute in Java Objektreferenzen (Zeiger auf Objekte) und nicht die Objekte selbst.

Personen-Objekte mit den in Abb. 1.4 angegebenen Attributen `name` und `geburtsdatum`[11] werden dann spezifiziert durch eine Klasse `Person`:

```
class Person {
  String name;
  int    geburtsdatum; /* in der Form JJJJMMTT */
};
```

Objekte des Typs[12] `Person` kann man durch Aufruf des *default-Konstruktors* `new Person()` erzeugen. Durch diesen Aufruf wird ein Speicherbereich für das neue Objekt alloziert und die Attribute des Objekts

[11] Achtung: in diesem Java-Beispiel kann man direkt auf diese Attribute zugreifen. Bei Abb. 1.4 wurden statt dessen Zugriffsmethoden angenommen, was man in Java auch realisieren könnte, aber hier das Beispiel zu unübersichtlich macht.

[12] Klassen sind in Java spezielle Typen.

mit Standardwerten initialisiert. Möchte man die Initialisierungswerte selbst festlegen, muss man der Klassendeklaration einen selbstgeschriebenen *Konstruktor* hinzufügen:

Konstruktor

```
class Person {
  String name;
  int    geburtsdatum; /* in der Form JJJJMMTT */

  Person( String n, int gd ) {
    name = n;
    geburtsdatum = gd;
  }
};
```

Der Name eines Konstruktors muss mit dem Namen der Klasse übereinstimmen, zu der er gehört und kann Parameter besitzen, die zur Initialisierung der Attribute verwendet werden können (s.o.). Näheres zu Konstruktoren finden Sie in Abschnitt 2.1.2.

Wir statten Personen-Objekte nun mit zwei Methoden aus: Erhält eine Person die Nachricht `drucken`, soll sie ihren Namen und ihr Geburtsdatum drucken; erhält eine Person die Nachricht `hat_geburtstag` mit einem Datum als Parameter, soll sie prüfen, ob sie an dem Datum Geburtstag hat (Datumsangaben werden als int-Zahl im Format JJJJMMTT codiert). Die bisherige Typdeklaration von `Person` wird also erweitert um die Methoden `drucken` und `hat_geburtstag`:

```
class Person {
  String name;
  int  geburtsdatum; /* in der Form JJJJMMTT */

  Person( String n, int gd ) {
    name = n;
    geburtsdatum = gd;
  }
  void  drucken() {
    System.out.println("Name: "+ this.name);
    System.out.println("Geburtsdatum: "+ this.geburtsdatum);
  }
  boolean hat_geburtstag ( int datum ) {
    return  (this.geburtsdatum%10000) == (datum%10000);
  }
}
```

Die Methode `drucken` druckt die beiden Attribute `name` und `geburtsdatum` des Personen-Objekts aus, auf dem sie aufgerufen wird.

Dieses Personen-Objekt wird der Methode bei ihrem Aufruf als impliziter Parameter mitgegeben. Dieser Parameter wird häufig als *this-Objekt* oder *self-Objekt* bezeichnet. Die Methode `hat_geburtstag` vergleicht die Monats- und Tagesangabe des Geburtsdatums des entsprechenden Personen-Objekts mit derjenigen des übergebenen Datums.

this-Objekt

1.3.2.2 Spezialisierung und Vererbung

Gemäß Abb. 1.4, S. 22, ist ein Student eine Person mit zwei zusätzlichen Attributen. Studenten sind also spezielle Personen. Es wäre demnach wünschenswert, die Implementierung von Studenten-Objekten durch geeignete Erweiterung und Modifikation der Implementierung von Personen-Objekten zu erhalten. Da die Methode `hat_geburtstag` auch für Studenten-Objekte korrekt funktioniert, können wir sie übernehmen.

Objektorientierte Programmiersprachen wie Java unterstützen es, Klassen wie gewünscht zu erweitern. Beispielsweise ist die folgende Klasse `Student` eine Erweiterung der Klasse `Person`: `Student extends Person`. Sie erbt alle Attribute und Methoden von `Person` – im Beispiel die Attribute `name` und `geburtsdatum` und die Methode `hat_geburtstag`. Die Methoden und Konstruktoren der Klasse, von der geerbt wird, können mit dem Schlüsselwort `super` angesprochen werden:

```
class Student extends Person {
  int  matrikelnr;
  int  semester;

  Student( String n, int gd, int mnr, int sem ) {
    super( n, gd );
    matrikelnr = mnr;
    semester = sem;
  }
  void  drucken() {
    super.drucken();
    System.out.println( "Matrikelnr: " +  matrikelnr );
    System.out.println( "Semesterzahl: " + semester );
  }
}
```

Anhand des obigen Beispiels lassen sich die Aspekte erkennen, die für die Vererbung wichtig sind. Der speziellere Typ (hier `Student`) besitzt alle Attribute des allgemeineren Typs (hier `Person`). Er kann auf die gleichen Nachrichten reagieren (hier auf `drucken` und `hat_geburtstag`), ggf. auch auf zusätzliche Nachrichten (hier nicht demonstriert). Er kann also die Attribute und die Typen der Nachrichten vom allgemeineren Typ erben. Er kann auch

die Methoden erben, wie wir am Beispiel von `hat_geburtstag` gesehen haben. In vielen Fällen reichen die Methoden des allgemeineren Typs aber nicht aus, wie das Beispiel von `drucken` zeigt. Sie müssen durch eine speziellere Variante ersetzt werden. In solchen Fällen spricht man von *Überschreiben* der Methode des allgemeineren Typs. Allerdings ist es häufig sinnvoll die überschriebene Methode zur Implementierung der überschreibenden Methode heranzuziehen. In unserem Beispiel wird in der Methode `drucken` der Klasse `Student` durch den Aufruf von

Überschreiben

```
super.drucken();
```

die Methode `drucken` der Klasse `Person` aufgerufen und dazu genutzt, die Attribute `name` und `geburtsdatum` auszudrucken. Der Aufruf von

```
super( n, gd );
```

im Konstruktor von `Student` bewirkt, dass der Konstruktor der Klasse `Person` aufgerufen wird.

1.3.2.3 Subtyping und dynamisches Binden

Im Prinzip können Studenten-Objekte an allen Programmstellen verwendet werden, an denen Personen-Objekte zulässig sind; denn sie unterstützen alle Operationen, die man auf Personen-Objekte anwenden kann, wie Attributzugriff und Methodenaufruf. In der objektorientierten Programmierung gestattet man es daher, Objekte von spezielleren Typen überall dort zu verwenden, wo Objekte von allgemeineren Typen zulässig sind. Ist S ein speziellerer Typ als T, so wird S auch als *Subtyp* von T bezeichnet; in diesem Sinne ist `Student` ein Subtyp von `Person`.

Subtyp

Der entscheidende Vorteil von Subtyping ist, dass wir einen Algorithmus, der für einen Typ formuliert ist, für Objekte aller Subtypen verwenden können. Um dies an unserem Beispiel zu demonstrieren, greifen wir eine Variante des Problems aus Abschn. 1.2.3, S. 23, auf: Wir wollen alle Elemente eines Felds mit Komponenten vom Typ `Person` drucken. Das folgende Programmfragment zeigt, wie einfach das mit objektorientierten Techniken geht:

```
int i;
Person[] pf = new Person[3];
pf[0] = new Person( "Meyer", 19631007 );
pf[1] = new Student( "Mueller", 19641223, 6758475, 5 );
pf[2] = new Student( "Planck", 18580423, 3454545, 47 );

for( i = 0; i < 3; i = i + 1 ) {
  pf[i].drucken();
}
```

In der dritten bis fünften Zeile wird das Feld mit einem Personen-Objekt und zwei Studenten-Objekten initialisiert. In der for-Schleife braucht nur die Druckmethode für jedes Element aufgerufen zu werden. Ist das Element eine

Person, wird die Methode `drucken` der Klasse `Person` ausgeführt; ist das Element ein Student, wird die Methode `drucken` der Klasse `Student` ausgeführt. Drei Aspekte sind dabei bemerkenswert:

1. Eine Fallunterscheidung an der Aufrufstelle, wie sie in der Pascal-Version von S. 23 nötig war, kann entfallen.

2. Die Schleife braucht nicht geändert zu werden, wenn das Programm um neue Personenarten, z.B. Angestellte, erweitert wird.

3. Der Aufruf `pf[i].drucken()` führt im Allgemeinen zur Ausführung unterschiedlicher Methoden.

Der letzte Punkt bedeutet, dass die Zuordnung von Aufrufstelle und ausgeführter Methode nicht mehr vom Übersetzer vorgenommen werden kann, sondern erst zur Laufzeit erfolgen kann. Da man alle Dinge, die zur Laufzeit geschehen als *dynamisch* bezeichnet, spricht man von *dynamischer Bindung* der Methoden. Alle Dinge, die zur Übersetzungszeit behandelt werden können, werden als *statisch* bezeichnet.

dynamische Bindung

1.3.3 Aufbau eines Java-Programms

Aus einer vereinfachten Sicht eines Programmierers besteht ein Java-Programm aus einer oder mehreren von ihm entwickelten Klassen[13], von denen eine Klasse eine sogenannte *main-Methode* besitzen muss, die den Programmeinstiegspunkt festlegt:

```
class Main {
  public static void main( String[] args )
    Anweisungsblock
}
```

Die Klasse mit dem Programmeinstiegspunkt muss in einer Datei enthalten sein, die die Dateiendung `.java` besitzt. Am Besten gibt man dieser Datei den Namen der Klasse mit dem Programmeinstiegspunkt, in unserem Fall also `Main.java`.

Die `main`-Methode hat stets den oben beschriebenen Aufbau: sie muss den Namen `main` besitzen, darf keine Werte zurückliefern (angegeben durch `void`) und nur einen Parameter vom Typ `String[]` besitzen. Dieser Parameter dient dazu, Argumente aufzunehmen, die dem Java-Programm beim Aufruf mitgegeben werden (s.u.).

[13] Im Absatz „Objektorientierte Programme“ von Abschnitt 2.1.2 „Klassen beschreiben Objekte“ wird eine differenziertere Sicht auf ein Java-Programm beschrieben, die auch Bibliotheksklassen einbezieht.

Bei Verwendung des J2SE Development Kits (vgl. Einleitung [14]) kann man die Klasse `Main` mit dem Kommando[15]

```
javac Main.java
```

übersetzen. Die Übersetzung liefert eine Datei `Main.class`, die man mit

```
java Main
```

ausführen kann. Beim Aufruf ist auf die korrekte Groß-/Kleinschreibung des Klassennamens zu achten.

Wie bereits oben erwähnt, können einem Programmaufruf Argumente, auch Programmparameter genannt, mitgegeben werden. Diese sind durch Leerzeichen voneinander zu trennen. Der folgende Aufruf demonstriert dieses Vorgehen:

```
java Main Hello World!
```

Auf solche Argumente kann man im Programm über den Parameter `args` der Methode `main` zugreifen: `args[0]` liefert das erste Argument, `args[1]` das zweite usw. Dies demonstriert das folgende vollständige Java-Programm, das die ersten beiden Argumente und die Zeichenreihe „Na endlich, das erste Programm mit Argumenten laeuft!“ ausgibt:

```
class Main {
  public static void main( String[] args )
  {
    if (args.length > 1) {
       System.out.print(args[0]);
       System.out.print(" ");
       System.out.print(args[1]);
       System.out.println();
    }
    System.out.println
      ("Na endlich, das erste Programm mit Argumenten laeuft!");
  }
}
```

Ein etwas größeres Programm besteht aus den uns bereits bekannten Klassen `Person` und `Student` sowie der Klasse `Test`, die die `main`-Methode enthält.

[14] J2SE Development Kit ist der ab Version 5.0 verwendete Name für die Entwicklungsumgebung, der die früher verwendeten Namen Java 2 Software Development Kit und Java Development Kit ersetzt.

[15] In Windows können solche Kommandos über die Windows-Eingabeaufforderung eingegeben werden.

```
class Person {
  String name;
  int  geburtsdatum; /* in der Form JJJJMMTT */

  Person( String n, int gd ) {
    name = n;
    geburtsdatum = gd;
  }
  void  drucken() {
    System.out.println("Name: "+ this.name);
    System.out.println("Geburtsdatum: "+ this.geburtsdatum);
  }
  boolean hat_geburtstag ( int datum ) {
    return  (this.geburtsdatum%10000) == (datum%10000);
  }
}

class Student extends Person {
  int  matrikelnr;
  int  semester;

  Student( String n, int gd, int mnr, int sem ) {
    super( n, gd );
    matrikelnr = mnr;
    semester = sem;
  }
  void  drucken() {
    super.drucken();
    System.out.println( "Matrikelnr: " +  matrikelnr );
    System.out.println( "Semesterzahl: " + semester );
  }
}

class Test {
  public static void main( String[] args ) {
    int i;
    Person[] pf = new Person[3];
    pf[0] = new Person( "Meyer", 19631007 );
    pf[1] = new Student( "Mueller", 19641223, 6758475, 5 );
    pf[2] = new Student( "Planck", 18580423, 3454545, 47 );

    for( i = 0; i < 3; i = i + 1 ) {
      pf[i].drucken();
    }
  }
}
```

1.3.4 Objektorientierte Sprachen im Überblick

Objektorientierte Sprachen unterscheiden sich darin, wie sie das objektorientierte Grundmodell durch Sprachkonstrukte unterstützen und wie sie die objektorientierten Konzepte mit anderen Konzepten verbinden. Mittlerweile gibt es etliche Sprachen, die objektorientierte Aspekte unterstützen. Viele von ihnen sind durch Erweiterung existierender Sprachen entstanden, beispielsweise C++ und Objective C (als Erweiterung von C), CLOS (als Erweiterung von Common Lisp), Ada95, Object Pascal[16] und Modula-3. Andere Sprachen wurden speziell für die objektorientierte Programmierung entwickelt, beispielsweise Simula, Smalltalk, Eiffel, BETA und Java.

An dieser Stelle des Kurses fehlen noch die Voraussetzungen, um die Unterschiede zwischen objektorientierten Sprachen detailliert zu behandeln. Andererseits ist es wichtig, einen Überblick über die Variationsbreite bei der sprachlichen Umsetzung objektorientierter Konzepte zu besitzen, bevor man sich auf eine bestimmte Realisierung – in unserem Fall auf Java – einlässt. Denn nur die Kenntnis der unterschiedlichen Umsetzungsmöglichkeiten erlaubt es, zwischen den allgemeinen Konzepten und einer bestimmten Realisierung zu trennen. Deshalb werden schon hier Gemeinsamkeiten und Unterschiede bei der sprachlichen Umsetzung übersichtsartig zusammengestellt, auch wenn die verwendeten Begriffe erst im Laufe des Kurses genauer erläutert werden.

Gemeinsamkeiten. In fast allen objektorientierten Sprachen wird ein Objekt als ein Verbund von Variablen realisiert, d.h. so, wie wir es in Abschn. 1.3.2 erläutert haben. Allerdings werden die Methoden implementierungstechnisch in den meisten Fällen nicht den Objekten zugeordnet, sondern den Klassen. Eine verbreitete Gemeinsamkeit objektorientierter Sprachen ist die *synchrone* Kommunikation, d.h. der Aufrufer einer Methode kann erst fortfahren, wenn die Methode terminiert hat. Mit dem Vokabular des objektorientierten Grundmodells formuliert, heißt das, dass Senderobjekte nach dem Verschicken einer Nachricht warten müssen, bis die zugehörige Methode vom Empfänger bearbeitet wurde; erst nach der Bearbeitung und ggf. nach Empfang eines Ergebnisses kann der Sender seine Ausführung fortsetzen. Das objektorientierte Grundmodell ließe auch andere Kommunikationsarten zu (vgl. Kap. 7). Beispiele dafür findet man vor allem bei objektorientierten Sprachen und Systemen zur Realisierung verteilter Anwendungen.

[16] Object Pascal liegt u.a. der verbreiteten Programmentwicklungsumgebung Delphi zugrunde.

Unterschiede. Die Unterschiede zwischen objektorientierten Programmiersprachen sind zum Teil erheblich. Die folgende Liste fasst die wichtigsten Unterscheidungskriterien zusammen:

- Objektbeschreibung: In den meisten Sprachen werden Objekte durch sogenannte Klassendeklarationen beschrieben. Andere Sprachen verzichten auf Klassen und bieten Konstrukte an, mit denen man existierende Objekte während der Programmlaufzeit klonen kann und dann Attribute und Methoden hinzufügen bzw. entfernen kann. Derartige Sprachen nennt man *prototypbasiert* (Beispiel: die Sprache Self).

- Reflexion: Sprachen unterscheiden sich darin, ob Klassen und Methoden auch als Objekte realisiert sind, d.h. einen Zustand besitzen und Empfänger und Sender von Nachrichten sein können. Beispielsweise sind in Smalltalk Klassen Objekte, und in BETA sind Methoden als Objekte modelliert.

- Typsysteme: Die Typsysteme objektorientierter Sprachen sind sehr verschieden. Am einen Ende der Skala stehen untypisierte Sprachen (z.B. Smalltalk), am anderen Ende Sprachen mit strenger Typprüfung, Subtyping und Generizität (z.B. Java und Eiffel).

- Vererbung: Auch bei der Vererbung von Programmteilen zwischen Klassen gibt es wichtige Unterschiede. In vielen Sprachen, z.B. in Java oder C#, kann eine Klasse nur von *einer* anderen Klasse erben (Einfachvererbung), andere Sprachen ermöglichen Mehrfachvererbung (z.B. CLOS und C++). Meist sind Vererbung und Subtyping eng aneinander gekoppelt. Es gibt aber auch Sprachen, die diese Konzepte sauber voneinander trennen (beispielsweise Sather). Variationen gibt es auch dabei, was für Programmteile vererbt werden können.

- Spezialisierung: Objektorientierte Sprachen bieten Sprachkonstrukte an, um geerbte Methoden zu spezialisieren. Diese Sprachkonstrukte unterscheiden sich zum Teil erheblich voneinander.

- Kapselung: Kapselungskonstrukte sollen Teile der Implementierung vor dem Benutzer verbergen. Die meisten OO-Sprachen unterstützen Kapselung in der einen oder anderen Weise. Allerdings gibt es auch Sprachen, die Kapselung nur auf Modulebene und nicht für Klassen unterstützen (z.B. BETA).

- Strukturierungskonstrukte: Ähnlich der uneinheitlichen Situation bei der Kapselung gibt es eine Vielfalt von Lösungen zur Strukturierung einer Menge von Klassen. Einige Sprachen sehen dafür Modulkonzepte vor (z.B. Modula-3), andere benutzen die Schachtelung von Klassen zur Strukturierung.

- Implementierungsaspekte: Ein wichtiger Implementierungsaspekt objektorientierter Programmiersprachen ist die Frage, wer den Speicher für die Objekte verwaltet. Beispielsweise ist in C++ der Programmierer dafür zuständig, während Java eine automatische Speicherverwaltung anbietet. Ein anderer Aspekt ist die Ausrichtung der Sprache auf die Implementierungstechnik. Sprachen, die für Interpretation entworfen sind, sind meist besser für dynamisches Laden von Klassen und ähnliche Techniken geeignet als Sprachen, die für Übersetzung in Maschinensprache ausgelegt sind.

Die Liste erhebt keinen Anspruch auf Vollständigkeit. Beispielsweise unterscheiden sich objektorientierte Sprachen auch in den Mechanismen zur dynamischen Bindung, in der Unterstützung von Parallelität und Verteilung sowie in den Möglichkeiten zum dynamischen Laden und Verändern von Programmen zur Laufzeit.

1.4 Aufbau und thematische Einordnung des Kurses

Dieser Abschnitt erläutert den Aufbau des Kurses und bietet eine kurze Übersicht über andere einführende Bücher im thematischen Umfeld.

Aufbau. Dieser Kurs lässt sich von den objektorientierten Konzepten leiten und nimmt diese als Ausgangspunkt für die Behandlung der sprachlichen Realisierung. Dieses Vorgehen spiegelt sich in der Struktur des Kurses wider. Jedes der sechs Hauptkapitel stellt einen zentralen Aspekt objektorientierter Programmierung vor:

- Kapitel 2: Objekte, Klassen, Kapselung. Dieses Kapitel erläutert, was Objekte sind, wie man sie mit Klassen beschreiben kann, wozu Kapselung dient, wie Klassen strukturiert werden können und welche Beziehungen es zwischen Klassen gibt.

- Kapitel 3: Vererbung und Subtyping. Ausgehend vom allgemeinen Konzept der Klassifikation werden die zentralen Begriffe Vererbung und Subtyping erläutert und ihre programmiersprachliche Realisierung beschrieben.

- Kapitel 4: Bausteine für objektorientierte Programme. Dieses Kapitel behandelt die Nutzung objektorientierter Techniken für die Entwicklung wiederverwendbarer Programmbausteine. In diesem Zusammenhang werden auch typische Klassen aus der Java-Bibliothek vorgestellt.

- Kapitel 5: Objektorientierte Programmgerüste. Oft benötigt man ein komplexes Zusammenspiel mehrerer erweiterbarer Klassen, um eine softwaretechnische Aufgabenstellung zu lösen. Programmgerüste bilden die Basis, mit der häufig vorkommende, ähnlich gelagerte Aufgabenstellungen bewältigt werden können, beispielsweise die Realisierung graphischer Bedienoberflächen. In diesem Kapitel erläutern wir das Abstract Window Toolkit von Java als Beispiel für ein objektorientiertes Programmgerüst.

- Kapitel 6: Parallelität in objektorientierten Programmen. Wie bereits auf Seite 20 erwähnt, unterstützen gängige objektorientierte Programmiersprachen i.A. keine inhärente Parallelität. Statt dessen muss Parallelität explizit programmiert werden. Dieses Kapitel behandelt die Realisierung von expliziter Parallelität in objektorientierten Programmen anhand des Thread-Konzepts von Java und führt in das zugehörige objektorientierte Monitorkonzept ein.

- Kapitel 7: Verteilte Programmierung mit Objekten. Die Realisierung verteilter Anwendungen wird in der Praxis immer bedeutsamer. Die objektorientierte Programmierung leistet dazu einen wichtigen Beitrag. Dieses Kapitel stellt die dafür entwickelten Techniken und deren sprachliche Unterstützung vor. Dabei werden wir uns auf Anwendungen im Internet konzentrieren.

Diese Struktur lässt sich wie folgt zusammenfassen: Kapitel 2 und 3 stellen die Grundkonzepte sequentieller objektorientierter Programmierung vor. Kapitel 4 und 5 zeigen, wie diese Konzepte für die Entwicklung von Programmbibliotheken und wiederverwendbaren Programmgerüsten genutzt werden können. Kapitel 6 und 7 behandeln die quasi-parallele objektorientierte Programmierung. Am Ende des Kurses bringt Kapitel 8 eine Zusammenfassung, behandelt Varianten bei der Realisierung objektorientierter Konzepte und bietet einen kurzen Ausblick.

Die behandelten Konzepte, Techniken und Sprachkonstrukte werden mit Hilfe von Beispielen illustriert. Um das Zusammenwirken der Techniken über die Kapitelgrenzen hinweg zu demonstrieren, wird schrittweise ein rudimentärer Internet-Browser entwickelt und besprochen.

Einordnung in die Literatur. Erlaubt man sich eine vereinfachende Betrachtungsweise, dann kann man die Literatur zur Objektorientierung in vier Bereiche einteilen. Dieser Absatz stellt die Bereiche kurz vor und gibt entsprechende Literaturangaben.

Der erste Bereich beschäftigt sich mit objektorientiertem Software-Engineering, insbesondere mit der objektorientierten Analyse und dem objektorientierten Entwurf. Zentrale Fragestellungen sind in diesem Zusammen-

hang: Wie entwickelt man ausgehend von einer geeigneten Anforderungsanalyse einen objektorientierten Systementwurf? Wie können objektorientierte Modelle und Techniken dafür nutzbringend eingesetzt werden? Eine gute und übersichtliche Einführung in dieses Gebiet bietet [HS97]. Insbesondere enthält das Buch ein ausführliches, annotiertes Literaturverzeichnis und eine Übersicht über einschlägige Konferenzen und Zeitschriften zur Objektorientierung.

Als zweiten Bereich betrachten wir die objektorientierte Programmierung, d.h. die Realisierung objektorientierter Programme ausgehend von einem allgemeinen oder objektorientierten Softwareentwurf. Sofern das nicht von Anfang an geschehen ist, muss dazu der Entwurf in Richtung auf eine objektorientierte Programmstruktur hin verfeinert werden. Ergebnis einer solchen Verfeinerung ist typischerweise eine Beschreibung, die angibt, aus welchen Klassen das zu entwickelnde System aufgebaut sein soll, welche existierenden Klassen dafür wiederverwendet werden können und welche Klassen neu zu implementieren sind. Dieser so verfeinerte Entwurf ist dann – meist unter Verwendung objektorientierter Programmiersprachen – zu codieren. Für den ersten Schritt benötigt man die Kenntnis objektorientierter Programmierkonzepte, für den zweiten Schritt ist die Beherrschung objektorientierter Programmiersprachen vonnöten. Eine exemplarisch gehaltene Darstellung des Zusammenhangs zwischen objektorientiertem Entwurf und objektorientierter Programmierung bietet [Cox86].

Der vorliegende Kurstext erläutert die Konzepte objektorientierter Programmierung und deren Umsetzung in der Sprache Java. Ähnlich gelagert ist das Buch [Mey00] von B. Meyer. Es führt die Konzepte allerdings anhand der Programmiersprache Eiffel ein und geht darüberhinaus auch auf Aspekte des objektorientierten Entwurfs ein. Desweiteren bietet das Buch [Bud01] von T. Budd eine sehr lesenswerte Einführung in verschiedene Aspekte objektorientierter Programmierung. Dabei wird bewußt mit Beispielen in unterschiedlichen Sprachen gearbeitet. Beide Bücher verzichten aber auf eine Behandlung von Programmgerüsten und von paralleler bzw. verteilter Programmierung.

In einem dritten Bereich der Objektorientierung siedeln wir Bücher an, die sich im Wesentlichen nur mit einer objektorientierten Programmiersprache beschäftigen. Selbstverständlich ist die Grenze zum zweiten Bereich fließend. Bücher, die auch konzeptionell interessante Aspekte enthalten, sind beispielsweise zu Beta [MMPN93], zu C++ [Lip91] und zu Java [HC97, HC98].

Orthogonal zu den drei genannten Bereichen liegen Arbeiten zur theoretischen Fundierung der Objektorientierung. Das Buch [AC96] entwickelt eine Theorie für Objekte ausgehend von verschiedenen Kalkülen und liefert damit zentrale programmiersprachliche Grundlagen. Eine gute Übersicht über den Stand der Technik von formalen Entwicklungsmethoden für objektorientierte Software bietet das Buch [GK96].

Selbsttestaufgaben

Aufgabe 1: Der intelligente Kühlschrank

Ein intelligenter Kühlschrank kommuniziert mit seiner Umwelt und mit sich selbst.

- Er testet regelmäßig seine Komponenten auf Funktionstüchtigkeit. Bei Defekten sendet er eine Nachricht an den Reparaturdienst und / oder informiert seinen Besitzer.
- Er prüft ständig, welche der Nahrungsmittel und sonstigen Gegenstände, die sein Besitzer in ihm aufheben will, vorhanden sind.
- Wenn ein Nahrungsmittel zu Ende geht, sendet er eine Nachricht an das Geschäft, das dieses Nahrungsmittel verkauft.
- Der Hausroboter sendet Nachrichten an den Kühlschrank, wenn eine Lieferung angekommen ist.

Entwickeln Sie ein objektorientiertes Modell eines intelligenten Kühlschranks, indem Sie die Gegenstände seiner Welt angeben und die Methoden, die der Kühlschrank und diese Gegenstände besitzen müssen, um zu kommunizieren und ihre Aufgaben zu erledigen.

Aufgabe 2: Erste Schritte in Java

Java-Programme werden normalerweise von einer sogenannten *virtuellen Maschine* ausgeführt. Ein Java-Programm wird mit einem Übersetzer in einen Zwischencode (Bytecode) übersetzt. Eine virtuelle Maschine ist ein Programm, das Bytecode ausführen kann. Für verschiedene Rechnerarchitekturen und Betriebssysteme existieren virtuelle Maschinen, die diesen Zwischencode ausführen können. Der Bytecode ist unabhängig von der jeweiligen Rechnerumgebung und kann so auf jedem Rechner ausgeführt werden, für den eine virtuelle Maschine existiert.

Im Folgenden sollen Sie das Java Software Development Kit installieren und kleine Java-Programme erstellen und ausführen, um sich mit der Syntax der Sprache, der Bedienung des Übersetzers und der virtuellen Maschine vertraut zu machen.

a) Installation der Java-Entwicklungsumgebung (Erfahrung)

Installieren Sie die Java-Entwicklungsumgebung JDK 5.0. Einen Link zum Downloaden der Installationssoftware können Sie auf unseren Kurswebseiten finden. Dort finden Sie auch Hinweise zur Installation, ebenso wie einen Link zum PC-Tutorial, das u.A. wertvolle Hinweise zur Installation des JDK enthält, zum Setzen von Umgebungsvariablen unter Windows etc.

b) Das erste Java-Programm **(Erfahrung)**

1. Legen Sie ein Verzeichnis an, in dem Sie Ihre Java-Klassen ablegen wollen.

2. Erstellen Sie mit einem Editor Ihres Systems eine Datei namens `Start.java`, die folgenden Inhalt hat:

```
public class Start {
  public static void main(String[] args) {
    System.out.println("Mein erstes Java-Programm");
  }
}
```

3. Erzeugen Sie mit dem Java-Übersetzer `javac` aus dem eben erstellten Programmtext eine `.class`-Datei: '`javac Start.java`'. Die .class Datei enthält den oben beschriebenen Bytecode für die Klasse `Start`.

4. Jede `.class`-Datei, deren zugehörige Klasse eine `main`-Methode enthält, kann mit der virtuellen Maschine von Java ausgeführt werden: Geben Sie den Befehl '`java Start`' ein[17]. Der Code des eben beschriebenen Programms wird jetzt ausgeführt.

5. Ergänzen Sie jetzt Ihr Start-Programm in der folgenden Art und Weise:

```
public class MeinStart {
  public static void main(String[] args) {
    String Nachname;
    String Vorname;
    Vorname = args[0];
    Nachname = args[1];
    System.out.println("Mein erstes Java-Programm");
    System.out.println("geschrieben von " + Vorname +
                       " " + Nachname );
  }
}
```

   Speichern Sie es unter `MeinStart.java` ab. Erzeugen Sie wieder eine `.class`-Datei. Geben Sie den Befehl

   '`java MeinStart MeinVorname MeinNachname`' ein.

[17] Beim Starten wird die Dateiendung `.class` weggelassen.

Aufgabe 3: Implizite Typkonvertierungen

Nennen Sie im folgenden Java-Programm alle Stellen, an denen implizite Typkonvertierungen vorkommen und begründen Sie Ihre Aussage.

```
class Main {
  public static void main( String[] args ) {
(1)  int i = 10;
(2)  long l = 55567843L;
(3)  byte by = 15;
(4)  boolean b = true;
(5)  double d = 1.25;
(6)  l = i;
(7)  d = l;
(8)  by = i;
(9)  l = l + by;
(10) by = by - b;
(11) d = (l / i) * 20;
  }
}
```

Das Programm enthält auch Zuweisungen und Operationen, die nicht erlaubt sind. Nennen Sie auch solche Stellen und begründen Sie Ihre Aussage.

Aufgabe 4: Schleifen

Erstellen Sie ein einfaches Java-Programm, das alle beim Programmaufruf mitgegebenen Programmparameter auf der Standardausgabe jeweils in einer eigenen Zeile ausgibt. Verwenden Sie zur Ausgabe alle im Kurstext eingeführten Schleifenkonstrukte.

Aufgabe 5: Kontrollstrukturen

Schreiben Sie ein Java-Programm, das die ersten zwei Programmparameter in Integerzahlen umwandelt und von beiden Zahlen den größten gemeinsamen Teiler bestimmt. Prüfen Sie dann, ob der ermittelte größte gemeinsame Teiler einer der Zahlen von 1 bis 4 ist. Verwenden Sie für diesen Test eine `switch`-Anweisung und geben Sie in dem jeweils zutreffenden Fall eine entsprechende Meldung aus. Verwenden sie den `default`-Fall, wenn der größte gemeinsame Teiler größer als 4 ist.

Aufgabe 6: Sortieren eines Feldes

Gegeben sei das folgende noch mit einigen syntaktischen Fehlern behaftete Programmfragment, das wir in dieser Aufgabe korrigieren und vervollständigen wollen.

```
class Sortieren  {
  public static void main (string[] args) {
    // Ein double Feld erzeugen, das genauso gross ist wie das
    // args-Feld
    doubble[] feld = new double[args.length}

    // alle Zahlen, die in args als Strings
    // vorliegen, in double-Werte umwandeln
    // und in das Feld feld eintragen
    For{int i == 0; i < args.length, i = i + 1 {
      Feld[i] = Double.parseDouble(args(i));
    }

    // Hier Programmcode zum Sortieren einfuegen

    // Hier Programmcode zur Bestimmung und
    // Ausgabe des groessten Elements einfuegen

    // den Inhalt den Feldes feld ausgeben
    for(int i := 0; i < args.length; i = i + 1) {
      System.out.println[i + ". " + feld[i]];
    ]
  }
}
```

Die main-Methode eines Programms bekommt beim Aufruf des Programms angegebene Parameter im Feld `args` vom Typ String übergeben.
Beispiel: `java Sortieren 1.2 3.56 2.9 -23.4 3.1415926`
Die Korrektur der syntaktischen Fehler vorausgesetzt, wandelt das obige Programm die der `main`-Methode als String-Feld übergebenen Parameter mit der Methode Double.parseDouble() in ein Feld mit double-Werten um. Diese werden dann ausgegeben.

Aufgaben:

1. Das obige Programm enthält noch einige Syntaxfehler. Markieren Sie zunächst alle Fehlerstellen möglichst genau zusammen mit einer (knappen) Fehlerbeschreibung. Korrigieren Sie anschließend die erkannten Syntaxfehler und überprüfen Sie, ob Sie tatsächlich alle Fehler gefunden haben und Ihr korrigiertes Programm vom Java-Übersetzer akzeptiert wird. Korrigieren Sie ggf. die noch verbliebenen Fehler, bevor Sie mit der zweiten Teilaufgabe und der Erweiterung des Programms beginnen[18].

[18]Eine Umleitung der Fehlermeldungen des Java-Compilers von der Bildschirmausgabe in eine Datei mit Namen `error` können Sie durch folgenden Aufruf erreichen: `javac -Xstdout error Sortieren.java`

2. Erweitern Sie das korrigierte Programm an den gekennzeichneten Stellen um Programmcode, der das Feld mit den als Parameter übergebenen double-Werten absteigend sortiert und anschließend das größte Element bestimmt. Testen Sie Ihr Programm mit verschiedenen Eingaben.

Aufgabe 7: Ausnahmebehandlung (Brandschutzübung)

Um die jährliche Brandschutzübung an der FernUniversität für die Informatiker interessanter zu gestalten, hat die Hagener Feuerwehr das Motto der diesjährigen Übung mit einem Java-Programm codiert. Da für die Feuerwehr tägliche Alarme und Ausnahmesituationen die Regel sind, hat sie dabei reichlich Gebrauch von Exceptions gemacht.

1. Wie heißt das Motto, das von dem unten angegebenen Programm bei Ausführung ausgegeben wird?

2. Beschreiben Sie detailliert, was bei der Ausführung des Programms passiert, d.h. welche Exceptions auftreten und wo diese abgefangen und behandelt werden![19]

```
public class Alarm_Alarm {

  public static void main(String[] args) {
    try {
      try {
        int i = 7 % 5;
        if ( (i / (i % 2)) == 1) throw new Exception();
        System.out.println("leichtsinnig");
      }

      catch (Exception e) {
        System.out.println("man");
        try {
          if ( (7 % 6 / (7 % 6 % 2)) == 1) throw new Exception();
          System.out.println("leichtsinnig");
        }
        catch (Exception u) {
          System.out.println("spielt");
        }
      }

      System.out.println("nicht");
```

[19] Beachten Sie bei der Lösung, dass die Klasse `ArithmeticException` Subtyp der Klasse `Exception` ist.

```
      try {
        int i = true & false ? 0 : 1;
        switch (i) {
          case 1:
            System.out.println("mit");
          default:
            throw new Exception();
        }
      }
      catch (ArithmeticException e) {
        System.out.println("Streichhoelzern");
      }
      catch (Exception e) {
        System.out.println("Feuer");
      }

      finally {
        int i = false && true ? 0 : 2;
        switch (i) {
          case 1:
            System.out.println("mit");
          default:
            throw new Exception();
        }
      }
    }
    catch (ArithmeticException e) {
      System.out.println("Kerzen");
    }
    catch (Exception e) {
      System.out.println("");
    }
  }
}
```

Musterlösungen zu den Selbsttestaufgaben

Aufgabe 1: Der intelligente Kühlschrank

Die Kühlschrankwelt kann z.B. die folgenden Gegenstände enthalten:

1. den Kuehlschrankbesitzer
2. den Kuehlschrank
3. den Hausroboter
4. den Reparaturdienst
5. die Lieferanten: Lebensmittelhaendler, Optiker, Weinhaendler
6. die Bank

Zu diesen Gegenständen gehören die folgenden Methoden:

1. Methoden des Kuehlschranks:
   - Testen-auf-vorhandenen-Strom
   - Testen-auf-korrekte-Temperatur
   - Erhoehen-der-Temperatur
   - Verringern-der-Temperatur
   - Senden-einer-Nachricht-an-Kuehlschrankbesitzer
   - Entnehmen-von-Kuehlschrankinhalten
   - Hinzufuegen-von-Kuehlschrankinhalten
   - Ueberpruefen-des-Kuehlschrankinhalts
   - Speichern-des-Kuehlschrankinhalts
   - Nahrungsmittelbestellung-bei-Bedarf
   - Kontaktlinsenmittelbestellung-bei-Bedarf
   - Champagner-und-Weissweinbestellung-bei-Bedarf
   - Bezahlen-der-Lieferung
2. Methoden des Roboters:
   - Reagieren-auf-das-Tuerklingeln
   - Tueroeffnen
   - Lieferung-entgegennehmen
   - Beladen-des-Kuehlschranks

3. Methoden des Reparaturdienst
   - Reparaturwunsch-entgegennehmen
   - Reparatur-durchfuehren
   - Rechnung-senden
4. Methoden des Nahrungsmittelhaendlers
   - Lieferwunsch-entgegennehmen
   - Lieferung-ausfuehren
   - Rechnung-schreiben
5. Methoden des Optikers
   - Lieferwunsch-entgegennehmen
   - Lieferung-ausfuehren
   - Rechnung-schreiben
6. Methoden des Weinhaendlers
   - Lieferwunsch-entgegennehmen
   - Lieferung-ausfuehren
   - Rechnung-schreiben
7. Methoden der Bank
   - Kundenkonto-fuehren
   - Ueberweisungsauftrag-annehmen

Aufgabe 2: Erste Schritte in Java

Zu dieser Aufgabe gibt es keine Musterlösung.

Aufgabe 3: Implizite Typkonvertierungen

```
class Main {
  public static void main( String[] args ) {
(1)  int i = 10;
(2)  long l = 55567843L;
(3)  byte by = 15;
(4)  boolean b = true;
(5)  double d = 1.25;
(6)  l = i;
(7)  d = l;
```

```
(8)  by = i;
(9)  l = l + by;
(10) by = by - b;
(11) d = (l / i) * 20;
  }
}
```

Implizite Typkonvertierungen Folgende erweiternden impliziten Typkonvertierungen werden für primitive Datentypen vorgenommen:

`byte` → `short` → `int` → `long` → `float` → `double` und `char` → `int`

- In Zeile (6) wird eine implizite Typkonvertierung des in i enthaltenen int-Wertes in einen long-Wert vorgenommen.
- In Zeile (7) wird eine implizite Typkonvertierung des in l enthaltenen long-Wertes in einen double-Wert vorgenommen.
- In Zeile (9) wird vor Ausführung der Addition eine implizite Typkonvertierung des in by enthaltenen byte-Wertes in einen long-Wert vorgenommen.
- Um die Division in Zeile (11) durchführen zu können, wird zunächst der in i enthaltene int-Wert in einen long-Wert konvertiert. Das Ergebnis der Division ist vom Typ `long`. Um die anschließende Multiplikation mit 20 durchführen zu können, wird der Wert 20 in einen long-Wert konvertiert. Bei der Zuweisung des Ergebnisses an die Variable d wird das Ergebnis in einen Wert vom Typ `double` konvertiert.

Ungültige Zuweisungen / Operationen Obiges Programm enthält folgende Fehler:

- In Zeile (8) müsste ein int-Wert einem byte-Wert zugewiesen werden. Da int-Werte größer sein können als der größte byte-Wert, könnte bei der Zuweisung ein Genauigkeitsverlust auftreten. Der Compiler weist diese Zuweisung daher zurück.
- Werte vom Typ `byte` sind inkompatibel mit Werten vom Typ `boolean`. Daher ist die Subtraktion in Zeile (10) unzulässig.

Aufgabe 4: Schleifen

Folgendes einfache Java-Programm verwendet zunächst die `while`-Schleife zur Ausgabe aller beim Programmaufruf mitgegebenen Programmparameter, dann die do-Schleife und anschließend beide Varianten der `for`-Schleife. Bei der do-Schleife muss darauf geachtet werden, dass die Ausführung des Schleifenrumpfes verhindert wird, wenn überhaupt keine Argumente beim

Programmaufruf mitgegeben werden. Dies wird durch Voranstellen der `if`-Anweisung erreicht.

```
public class Schleifen {
  public static void main(String[] args) {
    //Mit while-Schleife
    int i = 0;
    while (i < args.length) {
      System.out.println(args[i]);
      i = i + 1;
    }
    //Mit do-Schleife
    i = 0;
    if (args.length > 0)
      do {
        System.out.println(args[i]);
        i = i + 1;
      } while (i < args.length);
    // Mit erster Variante der for-Schleife
    for (i = 0; i < args.length; i = i + 1)
      System.out.println(args[i]);
    // Mit zweiter Variante der for-Schleife
    for (String arg : args)
      System.out.println(arg);
  }
}
```

Aufgabe 5: Kontrollstrukturen

Folgendes Java-Programm löst die gestellte Aufgabe:

```
public class Ggt {
  public static void main(String[] args) {
    if (args.length <= 1) return;
    int z1, z2, ggt;
    z1 = Integer.parseInt(args[0]);
    z2 = Integer.parseInt(args[1]);

    // Mit der kleinsten uebergebenen Zahl als
    // Teiler starten.
    if (z1 <= z2) ggt = z1;
    else
      ggt = z2;

    // Solange ggt nicht beide Zahlen z1 und z2 teilt,
    // verringere ggt um eins. Die Schleife bricht
```

```
    // spaetestens bei ggt == 1 ab.
    while ( (z1 % ggt) != 0 || (z2 % ggt) != 0) {
      ggt = ggt - 1;
    }
    switch (ggt) {
      case 1: {
        System.out.println
            (" Der groesste gemeinsame Teiler ist 1.");
        break;
      }
      case 2: {
        System.out.println
            (" Der groesste gemeinsame Teiler ist 2.");
        break;
      }
      case 3: {
        System.out.println
            (" Der groesste gemeinsame Teiler ist 3.");
        break;
      }
      case 4: {
        System.out.println
            (" Der groesste gemeinsame Teiler ist 4.");
        break;
      }
      default:
        System.out.println
            (" Der groesste gemeinsame Teiler ist groesser als 4.");
    }
  }
}
```

Aufgabe 6: Sortieren eines Feldes

Teilaufgabe 1: Eliminieren von Syntaxfehlern

Die Syntaxfehler sind im Folgenden in #, die Fehlerbeschreibung in eine Folge von 3 Sternen eingeschlossen.

```
class Sortieren  {
  public static void main (#s#tring[] args) {
       *** Der vordefinierte Datentyp fuer Zeichenketten    ***
       *** ist String, nicht string.                        ***

    // Ein double Feld erzeugen, das genauso gross ist
    // wie das args-Feld
```

```
    doub#b#le[] feld = new double[args.length #}# #;#
       *** Der vordefinierte Datentyp fuer double-Werte     ***
       *** ist double, nicht doubble.                       ***
       *** Bei der Erzeugung eines Arrays wird dessen       ***
       *** Groesse in eckige Klammern [...] eingeschlossen. ***
       *** Jede Anweisung muss mit einem Semikolon abge-    ***
       *** schlossen werden.                                ***

    // Alle Zahlen, die in args als Strings vorliegen
    // in double-Werte umwandeln und in das Feld feld eintragen
    #F#or #{# int i #==# 0;i<args.length #,# i = i + 1 #}# {
       *** Eine for-Schleife beginnt mit dem Schluesselwort  ***
       *** for und nicht For. Zur Initialisierung einer      ***
       *** Laufvariablen der for-Schleife wird eine          ***
       *** Zuweisung (=) verwendet, kein Vergleich (==).     ***
       *** Die Angaben fuer die Laufvariable muessen in      ***
       *** runde statt in geschweifte Klammern eingeschlossen***
       *** werden. Die einzelnen Anweisungen innerhalb der   ***
       *** Klammern muessen durch ein Semikolon abgeschlossen***
       *** werden, nicht durch ein Komma.                    ***

      #F#eld[i] = Double.parseDouble(args #(# i #)# );
       *** Gross- und Kleinschreibung ist in Java relevant.  ***
       *** Die verwendete Variable fuer das Array, das die   ***
       *** double-Werte aufnehmen soll, ist weiter oben mit  ***
       *** dem Namen feld deklariert worden. Feld waere eine ***
       *** von feld verschiedene Variable.                   ***
       *** Ein Element eines Arrays wird ueber seinen Index  ***
       *** angesprochen, der in eckige Klammern gesetzt      ***
       *** werden muss. Runde Klammern sind falsch.          ***
    }

    // Hier Programmcode zum Sortieren einfuegen

    // Hier Programmcode zur Bestimmung und
    // Ausgabe des groessten Elements einfuegen

    // Den Inhalt des Feldes feld ausgeben
    for(int i #:=# 0;i < args.length;i = i + 1) {
       *** Eine Zuweisung erfolgt durch =, nicht durch      ***
       *** := wie in PASCAL.                                ***

      System.out.println #[# i + ". " + feld[i] #]# ;
       *** Aktuelle Parameter werden in runde Klammern      ***
       *** eingeschlossen.                                  ***

    #]#
       *** Statt der eckigen Klammer muss eine geschweifte  ***
       *** Klammer stehen.                                  ***
  }
}
```

Das syntaktisch korrekte Programmfragment sieht wie folgt aus:

```
class Sortieren {
  public static void main(String[] args) {
    // Ein double Feld erzeugen, das genauso gross ist
    // wie das args-Feld.
    double[] feld = new double[args.length];

    // Alle Zahlen, die in args als Strings vorliegen,
    // in double-Werte umwandeln und in das Feld feld eintragen.
    for (int i = 0; i < args.length; i = i + 1) {
      feld[i] = Double.parseDouble(args[i]);
    }

    // Hier Programmcode zum Sortieren einfuegen

    // Hier Programmcode zur Bestimmung und
    // Ausgabe des groessten Elements einfuegen

    // Den Inhalt des Feldes feld ausgeben
    for (int i = 0; i < args.length; i = i + 1) {
      System.out.println(i + ". " + feld[i]);
    }
  }
}
```

Teilaufgabe 2: Erweiterung des korrigierten Programms

Da das Feld absteigend sortiert wird, muss das größte Element das erste Element des Feldes sein und muss nicht erst durch Suchen ermittelt werden.

```
class Sortieren {
  public static void main(String[] args) {
    // Ein double Feld erzeugen, das genauso gross
    // ist wie das args-Feld
    double[] feld = new double[args.length];

    // Alle Zahlen, die in args als Strings vorliegen
    // in double-Werte umwandeln und in das Feld feld eintragen
    for (int i = 0; i < args.length; i = i + 1) {
      feld[i] = Double.parseDouble(args[i]);
    }

    // Feld absteigend sortieren
    for (int i = 0; i < args.length - 1; i = i + 1) {
      int max = i;
      for (int j = i + 1; j < args.length; j = j + 1) {
        if (feld[j] > feld[max]) {
```

```
          max = j;
        }
      }
      double h = feld[i];
      feld[i] = feld[max];
      feld[max] = h;
    }

    // Ausgabe des groessten uebergebenen Wertes.
    // Wir gehen bei dieser Loesung davon aus, dass
    // mindestens ein Programmparameter uebergeben wird.
    System.out.println("Das groesste Element ist " + feld[0]);

    // Den Inhalt des Feldes feld ausgeben
    for (int i = 0; i < args.length; i = i + 1) {
      System.out.println(i + ". " + feld[i]);
    }
  }
}
```

Aufgabe 7: Ausnahmebehandlung (Brandschutzübung)

1. Das Motto der diesjährigen Brandschutzübung lautet:

   ```
   man
   spielt
   nicht
   mit
   Feuer
   ```

2. Um dieses Ergebnis zu finden, muss man Schritt für Schritt die Ausführung des Programms nachvollziehen:

   1. 7%5 ergibt 2 und i%2 somit 0, d.h. es wird eine `ArithmeticException` wegen der Division durch null bei ((i/(i%2)) geworfen.
   2. Diese wird mit `catch(Exception e)` abgefangen und es wird `man` ausgegeben.
   3. Da ((7%6/(7%6%2))==1) wahr ist, wird eine Ausnahme vom Typ `Exception` erzeugt und bei `catch(Exception u)` abgefangen. Dann wird `spielt` ausgegeben.
   4. `nicht` wird ausgegeben.
   5. true&false?0:1 ergibt 1, somit wird nach `case 1:` `mit` ausgegeben.

6. Da der `case 1:` Fall keine `break`-Anweisung enthält, wird noch die Anweisung des default-Falles ausgegeben und mit `throw new Exception()` eine neue Ausnahme vom Typ `Exception` erzeugt.
7. Diese wird mit `catch(Exception e)` abgefangen und es wird `Feuer` ausgegeben.
8. Nun wird der `finally`-Teil ausgeführt. false&&true?0:2 ergibt 2 und im default-Fall wird eine Ausnahme vom Typ `Exception` geworfen. Diese wird wiederum mit `catch(Exception e)` abgefangen, jedoch wird nichts mehr ausgegeben.