Modul 226: Objektorientiert implementieren

Grundlagen zur Umsetzung eines objektorientierten Designs (OOD) mit Java anhand eines Fallbeispiels mit Repetitionsfragen und Antworten

Markus Ruggiero

Modul 226: Objektorientiert implementieren
Grundlagen zur Umsetzung eines objektorientierten Designs (OOD) mit Java anhand eines Fallbeispiels mit Repetitionsfragen und Antworten
Markus Ruggiero

Grafisches Konzept: december und juli, Wernetshausen
Satz und Layout: Mediengestaltung, Compendio Bildungsmedien AG, Zürich
Druck: Edubook AG, Merenschwand

Artikelnummer: 10067
Auflage: 3., überarbeitete Auflage 2012
Ausgabe: K1094
Sprache: DE
Code: ICT 070

Inhaltsverzeichnis

Vorwort

Liebe Leserin, lieber Leser

Vorweg schon einmal herzliche Gratulation! Sie haben sich für den Einsatz eines der aktuellsten Lehrmittel der Informatikausbildung entschlossen.

An wen richtet sich die Lernwelt «Informatik»?

Die Lernwelt «Informatik» ist ausgerichtet auf die gültigen Modulbeschreibungen für die Informatik-Grund- und -Weiterbildung. Mit diesem Grundlagenbuch wenden wir uns deshalb an Auszubildende und Unterrichtende

- einer Informatiklehre,
- der Informatikmittelschulen,
- der höheren Berufsbildung und
- von Ausbildungsgängen und Schulungen in der Erwachsenenbildung.

Dank zahlreicher Beispiele, Grafiken, Abbildungen und Übungen mit kommentierten Lösungen eignet sich die Lernwelt «Informatik» auch für das Selbststudium.

Wie Sie mit diesem Lehrmittel arbeiten

Dieses Arbeitsbuch bietet Ihnen mehr als nur einen Lerntext. Deshalb weisen unsere Bildungsmedien eine Reihe von Charakteristiken auf, die Ihnen Ihre Arbeit erleichtern:

- Das **Inhaltsverzeichnis** dient Ihnen als Orientierungshilfe und als Lernrepetition. Fragen Sie sich, was Sie von jedem Kapitel erwarten, und überprüfen Sie anschliessend an das Bearbeiten des Lerntextes, was Sie jetzt zu den einzelnen Teilen wissen.
- Wissen Sie gerne im Voraus, wofür Sie Ihre kostbare Zeit einsetzen? Kein Problem, lesen Sie die **Lernziele** vor der Lektüre des entsprechenden Teils. An gleicher Stelle finden Sie auch eine Auflistung der **Schlüsselbegriffe.**
- Die einzelnen Lerneinheiten werden durch eine **Zusammenfassung** abgeschlossen. Sie greift die wichtigsten Punkte des vorangegangenen Textes nochmals auf und stellt sie in den richtigen Zusammenhang.
- Nach dem Durcharbeiten der einzelnen Lerneinheiten können Sie anhand der **Repetitionsfragen** überprüfen, ob Sie das Gelernte verstanden haben. Die **Lösungen** zu diesen Repetitionsfragen finden Sie im Anhang des Buchs. Bitte beachten Sie, dass die Übungen nicht fortlaufend nummeriert sind; die Nummern dienen lediglich zum Auffinden der Lösung.
- Nutzen Sie das **Glossar;** schlagen Sie dort nach, wenn Sie einen Begriff nicht verstehen.
- Das **Stichwortverzeichnis** beschliesst das Lehrmittel. Sie können es benutzen, wenn Sie einzelne Abschnitte zu bestimmten Schlagwörtern nachlesen wollen.

Wer steht hinter der Lernwelt «Informatik»?

Die erfahrenen Lehrmittelentwickler von Compendio Bildungsmedien haben die Lernwelt «Informatik» zusammen mit ausgewiesenen Fachleuten und Kennern der Informatikausbildung konzipiert und realisiert.

Dank gebührt allen, die trotz grossem Zeitdruck mit Rat und Tat am Konzept und an der Ausarbeitung mitgewirkt haben.

In eigener Sache

Um den Text dieses Lehrbuchs möglichst einfach und verständlich zu halten, wurde bewusst auf die weibliche Form bei Substantiven wie z. B. Kundin, Anwenderinnen verzichtet.

Haben Sie Fragen oder Anregungen zu diesem Lehrmittel? Über unsere E-Mail-Adresse postfach@compendio.ch können Sie uns diese gerne mitteilen. Sind Ihnen Tipp- oder Druckfehler aufgefallen, danken wir Ihnen für einen entsprechenden Hinweis über die E-Mail-Adresse korrekturen@compendio.ch.

Wir wünschen Ihnen mit diesem Lehrmittel viel Spass und Erfolg.

Zürich, im August 2012

Andreas Ebner, Unternehmensleiter
Markus Ruggiero, Autor
Hans Egg, Fachlektor
Johannes Scheuring, Redaktor

Über dieses Lehrmittel

Inhalt und Aufbau dieses Lehrmittels

Dieses Lehrmittel führt Sie in die **objektorientierte Programmierung (OOP) mit Java** ein. In der modernen Softwareentwicklung ist die OOP der am häufigsten gewählte Ansatz zur Problemlösung. Ohne Kenntnisse über die objektorientierte Denkweise kommt heute kein Softwareentwickler mehr aus. Verschiedene Programmiersprachen unterstützen diese Art der Programmierung. Neben Java sind C++, C# von Microsoft und Objective-C von Apple weitverbreitet. Java ist eine mächtige, aber einfach zu erlernende Sprache, die streng auf Objektorientiertheit ausgerichtet ist.

Das vorliegende Lehrmittel ist wie folgt aufgebaut:

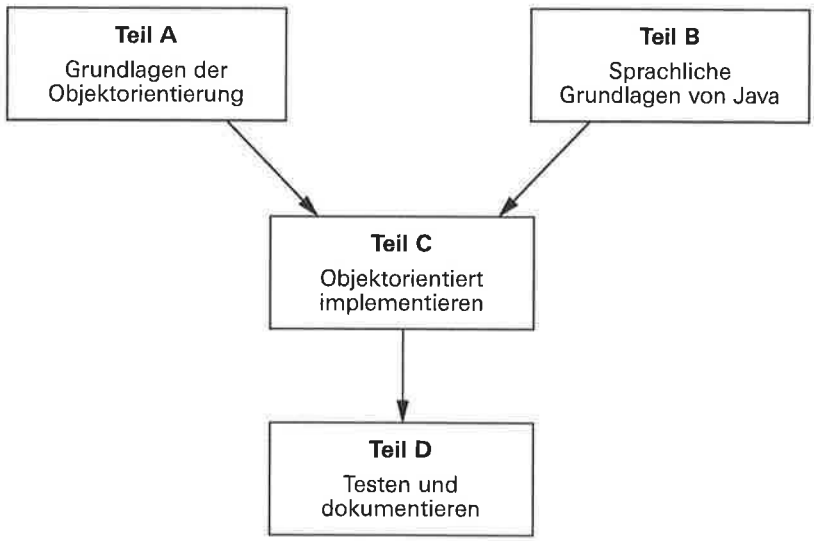
- In **Teil A** lernen Sie die Grundlagen der OOP kennen. Anhand eines einfachen Programms werden Sie an die objektorientierte Denkweise herangeführt und schliessen Bekanntschaft mit wichtigen objektorientierten Ideen. Nach einem kurzen Rückblick in die Geschichte der Programmierung werden wichtige Grundsätze und Konzepte der Objektorientierung näher vorgestellt. Dabei lernen Sie, was Klassen, Objekte, Methoden und Nachrichten sind und wie Vererbung und Polymorphismus funktionieren. Schliesslich erhalten Sie einen Einblick in die typischen Darstellungsformen, die im Rahmen des OOD[1] zur Anwendung kommen.
- **Teil B** widmet sich den sprachlichen Grundlagen von Java, wobei der Aufbau und die Syntax[2] von Java im Vordergrund stehen. Weiter lernen Sie verschiedene Datentypen von Java sowie deren Kontrollstrukturen für den Programmablauf kennen.
- **Teil C** demonstriert anhand eines Beispiels die Umsetzung eines OOD. Hier sehen Sie, wie OO-Konzepte praktisch angewendet und OO-Entwürfe mittels Java implementiert werden. Dabei lernen Sie, was ein Java-Paket ist und wie ein Java-Programm mit seiner Umgebung und seinen Benutzern kommunizieren kann. Zudem erfahren Sie, was ein Konstruktor ist und wie die offizielle Java-Dokumentation aufgebaut ist. Ein eigenes Kapitel ist den Werkzeugen gewidmet, die Sie für die Programmierung eines OO-Programmcodes verwenden können.
- **Teil D** zeigt auf, wie ein fertiggestelltes OO-Programm systematisch getestet und korrekt dokumentiert werden kann. Dabei lernen Sie den Unit-Test kennen und erfahren, wie Sie im Java-Programmcode Kommentare einfügen und automatisch zu einer Dokumentation aufbereiten können.
- Im **Anhang** finden Sie die Gesamtzusammenfassung, Antworten auf die Repetitionsfragen, den Programmcode der Klasse **ConsoleReader** und das Stichwortverzeichnis des Lehrmittels.

[1] Abk. für: objectoriented design. Engl. für: objektorientierter Entwurf.

[2] Fachbegriff für: Lehre vom (korrekten) Satzbau. Teil der Grammatik, der sich mit den Mustern und Regeln befasst, wie einzelne Sprachelemente zu grösseren funktionellen Einheiten zusammengefasst und deren Beziehungen untereinander formuliert werden müssen.

Der Aufbau dieses Lehrmittels lässt sich wie folgt veranschaulichen:

[0-1] Aufbau des Lehrmittels (Übersicht)



Dieses Lehrmittel liefert die Grundlage für den Erwerb folgender Kompetenzen:

1. Ein objektorientiertes Design nachvollziehen und durch technische Klassen ergänzen.
2. Dynamische und statische Strukturen zwischen Objekten bzw. Klassen mittels UML (Unified Modeling Language) darstellen.
3. Objektorientiertes Design implementieren.
4. Klassen systematisch prüfen (Unit-Test).
5. Klassen- und Systemdokumentation vervollständigen.

Technische und methodische Voraussetzungen

Bei allen wichtigen Betriebssystemen (Windows, Mac OS X, Linux, Unix) wird normalerweise auch die Java-Laufzeitumgebung geliefert und bei der Systeminstallation installiert. Als Integrierte Entwicklungsumgebung wird in diesem Lehrmittel Eclipse verwendet. Unter Windows muss die Java-Entwicklungsumgebung meistens nachinstalliert werden. Vergleichen Sie zur Installation von Java und Eclipse das Linkverzeichnis auf S. 11.

Für die Bearbeitung dieses Lehrmittels werden folgende Kenntnisse und Fähigkeiten vorausgesetzt:

Verlangte Kenntnisse/Fähigkeiten	Informationsquellen
Prozedurale Programmierung mit einer auf der C-Syntax basierenden Sprache (C/C++, JavaScript, PHP)	• I-CH-Modul 103 (C/C++) • I-CH-Modul 118 (C/C++) • I-CH-Modul 133 (PHP)

Folgende Fähigkeiten müssen Sie mitbringen, um das Lehrmittel erfolgreich zu nutzen:

- Sie können einen Computer mit grafischer Benutzeroberfläche problemlos bedienen (Windows, Mac OS X oder Linux).
- Sie können das Internet gezielt für die Suche von Informationen und Dokumentationen nutzen.
- Sie verstehen technische Fachbegriffe in englischer Sprache, da viele Informationen zur Java-Programmierung im Internet nur auf englischsprachigen Seiten vorliegen.

Eine Sprache lernt man nicht nur, indem man Bücher liest. Um eine Programmiersprache wie Java zu beherrschen, muss sie intensiv geübt werden. Dieses Lehrmittel stellt Ihnen verschiedene Beispiele zur Verfügung, um den theoretischen Lernstoff aktiv zu verarbeiten und nachzuvollziehen. Spielen Sie mit dem Code und probieren Sie eigene Ideen aus!

Nützliche Links

Thema	Hyperlink	Kurzbeschreibung
Java	http://www.oracle.com/technetwork/java/index.html	Java Haupteinstieg bei Oracle (in englischer Sprache)
	http://docs.oracle.com/javase/6/docs/api/	Dokumentation zu Version 6 (in englischer Sprache)
	http://docs.oracle.com/javase/tutorial/	Tutorials mit vielen Beispielen
OO-Konzepte und -Bücher	http://de.wikipedia.org/wiki/Objektorientierte_Programmierung	Überblick über die OOP
	http://www.galileocomputing.de/artikel/gp/artikelID-215	Einführung in die OOP
	http://www.dpunkt.de/java//index.html	Einführung in Java und OOP
	http://www.galileocomputing.de/openbook/javainsel6/	Kostenlose Online-Version des Buchs «Java ist auch eine Insel»
	http://www.mindviewinc.com/downloads/TIJ-3rd-edition4.0.zip	Online-Version des Buchs «Thinking in Java» von Bruce Eckels. (Download in HTML-Format)
Werkzeuge	http://www.eclipse.org	Einstiegseite der Entwicklungsumgebung Eclipse
	http://www.junit.org	Testframework JUnit (Download und Dokumentation)
	http://argouml.tigris.org/	CASE-Tool ArgoUML
Quellcode Fallbeispiel	http://www.compendio.ch/Bildungsmedien/Downloads.asp?Titel=2046	Kostenloser Download des Quellcodes zum Airportscheduler

Nützliche Literatur

Autor/Herausgeber	Titel	ISBN/Verlag	Auflage/Jahr
Eckels, Bruce	Thinking in Java (4th Edition)	978-0-1318-7248-6, Prentice Hall	Februar 2006
Kaintantzis, Nikolaos; Waldispühl, Candidus	Modul 103: Strukturiert programmieren nach Vorgabe	Compendio Bildungsmedien	3. Auflage, 2007
Ruggiero, Markus; Scheuring, Johannes	Modul 118: Aufgabenstellung analysieren und implementieren	Compendio Bildungsmedien	3. Auflage, 2010
Ullenboom, Christian	Java ist auch eine Insel	978-3-8362-1802-3, Galileo Computing	10. Auflage, 2012

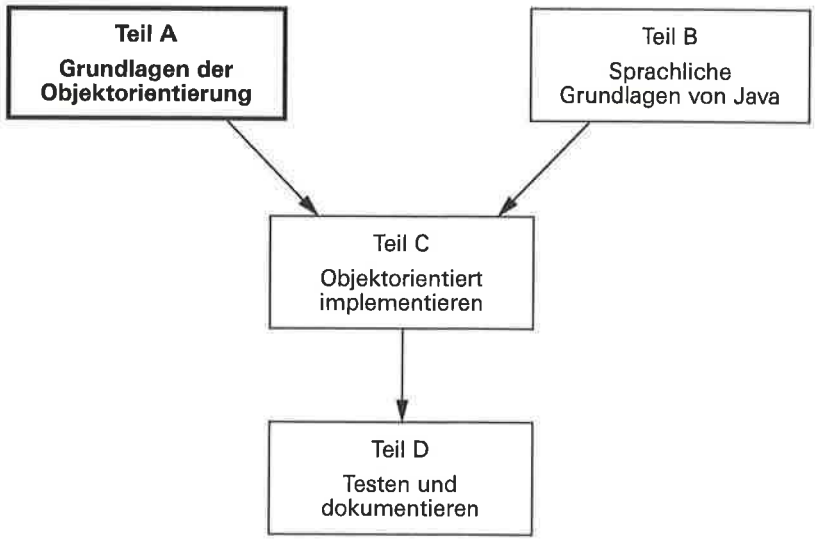
Teil A Grundlagen des objektorientierten Designs (OOD)

Einleitung, Lernziele und Schlüsselbegriffe

Einleitung

In diesem Teil lernen Sie die **Grundlagen des objektorientierten Designs (OOD) und der objektorientierten Programmierung (OOP)** kennen. Sie erfahren, welche Ideen dahinterstecken und wie sich die objektorientierte Programmierung von der klassischen strukturierten Programmierung unterscheidet. Erste Erfahrungen machen Sie mit einem einfachen Beispiel, das Sie auf spielerische Art an die Thematik und die Denkweise heranführt. Folgende Grafik zeigt, wo Sie sich innerhalb des Lehrmittels befinden:

Teil A im Gesamtzusammenhang



Lernziele und Lernschritte

	Lernziele	Lernschritte
☐	Sie können das Paradigma des objektorientierten Ansatzes beschreiben und anhand von Beispielen erläutern, welche prinzipiellen Unterschiede gegenüber dem funktionalen Ansatz bestehen.	• Einführung in die objektorientierte Programmierung • Probleme der klassischen prozeduralen Programmierung • Grundideen der objektorientierten Programmierung
☐	Sie können aufzeigen, wie durch Klassen und deren Attribute und Methoden die reale Welt im (vorgegebenen) Design abgebildet wird.	• Einführung in die objektorientierte Programmierung • Grundideen der objektorientierten Programmierung • Eigenschaften und Beziehungen
☐	Sie können die Beziehungstypen zwischen Klassen und das Konzept der Vererbung darlegen und aufzeigen, wie diese umgesetzt werden.	• Eigenschaften und Beziehungen Vererbung • Abstrakte Klassen
☐	Sie können an einem Codebeispiel den Effekt von Polymorphie aufzeigen.	• Polymorphismus
☐	Sie können aufzeigen, wie durch die Nutzung von Schnittstellen der Code unabhängig erstellt und getestet werden kann.	• Schnittstellen (Interfaces)
☐	Sie können die Notation des Klassen- und Objektdiagramms interpretieren und aufzeigen, wie sich diese unterscheiden.	• Die grafische Sprache UML • Klassendiagramme zeigen die statische Sicht • Objektdiagramme zeigen das Beziehungsgeflecht
☐	Sie können die Notation von Sequenz- und Kollaborationsdiagramm interpretieren und aufzeigen, wie sich damit Abläufe darstellen lassen.	• Notationen und Diagramme • Sequenzdiagramme zeigen den Interaktionsablauf

Schlüsselbegriffe

abstrakt, Accessor, Attribut, Ausnahme, Beziehung, Datenkapselung, Exception, Instanz, Instanzvariable, Interface, Klasse, Klassendiagramm, main()-Methode, new, Objekt, Objektdiagramm, Polymorphismus, Rolle, Sequenzdiagramm, UML, Vererbung

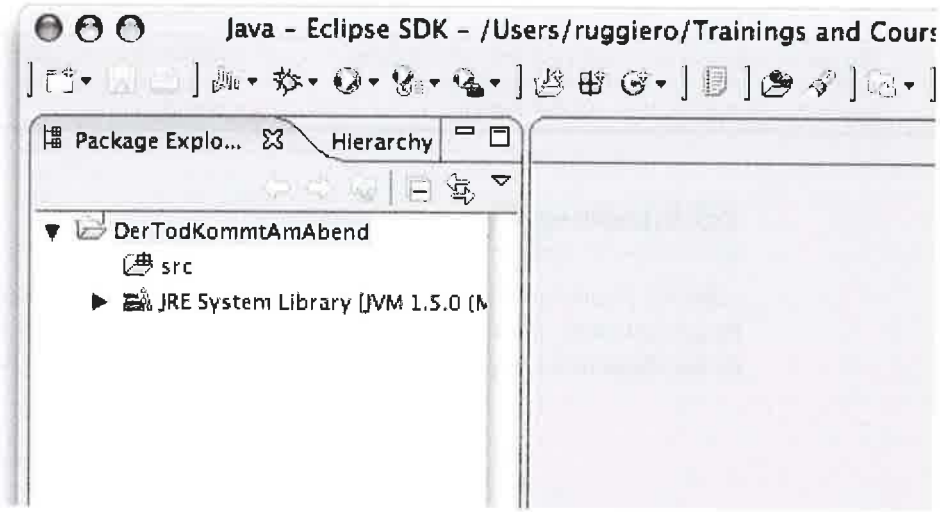
1 Einführung in die objektorientierte Programmierung

Es gibt diverse Programmiersprachen, die sich für die **objektorientierte Programmierung (OOP)** eignen. In diesem Lehrmittel wird die **Programmiersprache Java** vorgestellt, die in zahlreichen Anwendungsbereichen starke Verbreitung gefunden hat. In diesem Kapitel erhalten Sie eine Einführung in die OOP mit Java anhand eines Beispielprojekts.

1.1 Projekt eröffnen und Paket erstellen

In unserem Beispielprojekt wird ein Kriminalfilm gedreht, in dem mehrere Personen mit unterschiedlichen Rollen vorkommen. Das zugehörige Drehbuch erteilt Anweisungen, welcher Schauspieler wann was tut. Die Umsetzung dieser Anweisungen soll mittels Java erfolgen. Dabei werden Sie viel über die objektorientierte Programmierung lernen. Zunächst muss jedoch das entsprechende Projekt eröffnet werden. Dazu können Sie beispielsweise die **Entwicklungsumgebung Eclipse**[1] verwenden. Erstellen Sie darin ein **neues Java-Projekt** und nennen Sie es z. B. **DerTodKommtAmAbend.**

[1-1] Krimiprojekt in Eclipse eröffnen



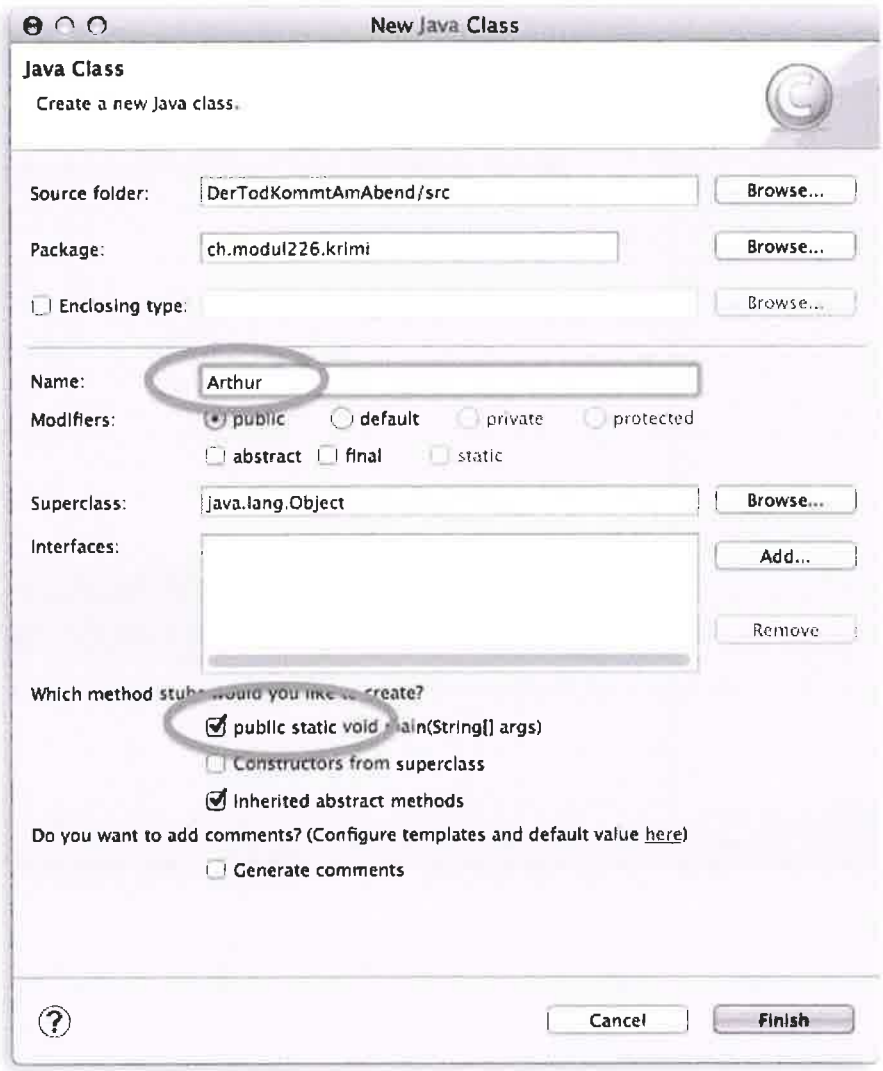
Nach der Eröffnung des Beispielprojekts erstellen Sie ein erstes **Paket** und nennen dieses **ch.modul226.krimi.** OOP-Pakete dienen zur Organisation des Programmcodes bzw. Quelltextes. Vergleichen Sie dazu auch das Kapitel 9, S. 98.

[1] Eine Einführung in die Arbeit mit Eclipse finden Sie in Kapitel 13.1, S. 125.

1.2 Produzent definieren und Programm starten

Jedes Filmprojekt hat einen Produzenten, der für die Organisation und die finanziellen Aspekte zuständig ist. Auch in unserem Krimi gibt es einen Produzenten namens Arthur. Die Klasse **Arthur** beschreibt, wer dieser Produzent ist:

[1-2] Neue Klasse erstellen

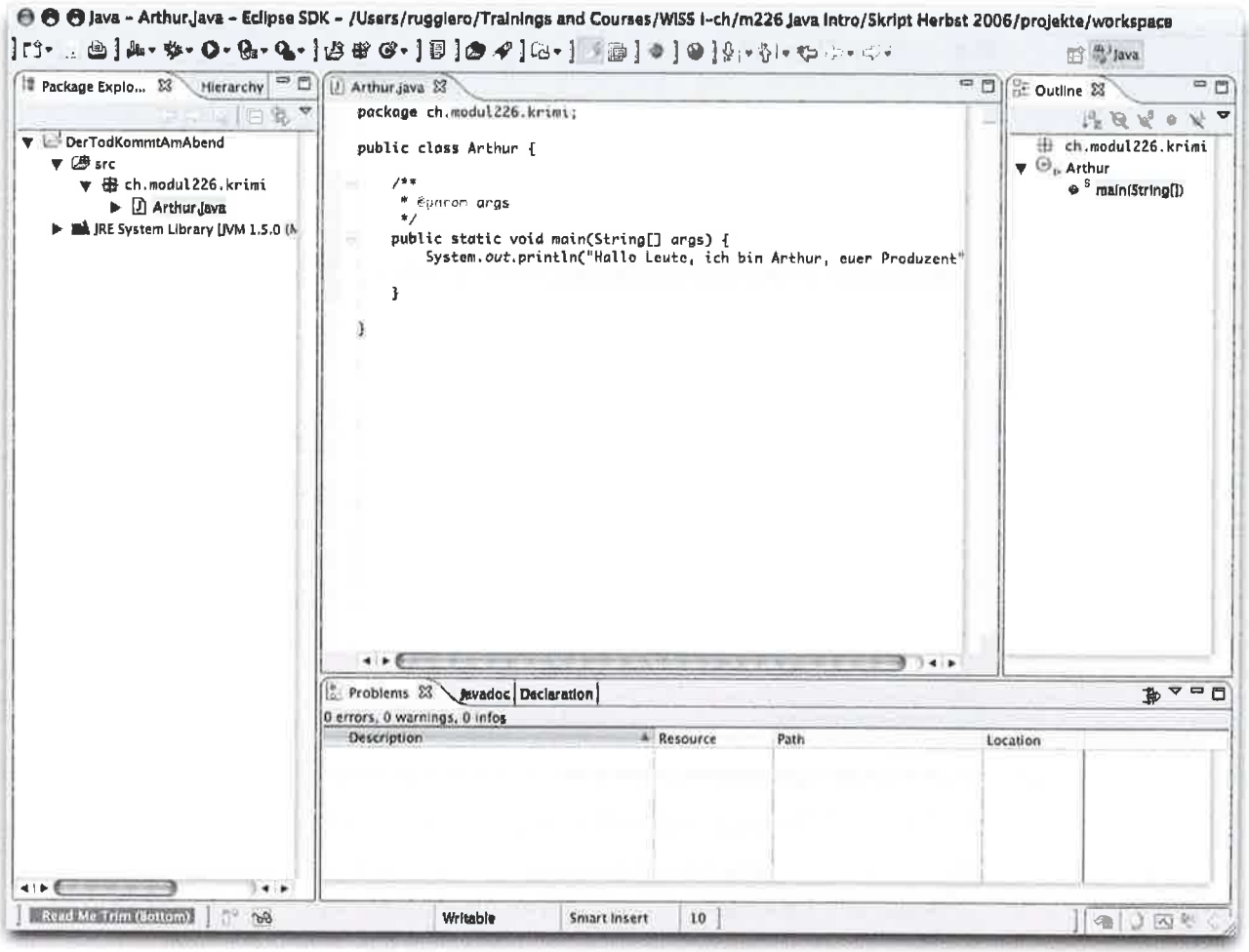


Die Klasse **Arthur** gehört in unserem Beispielprojekt zum Paket **ch.modul226.krimi.** Die **main()-Methode** dient in Java als Einstiegspunkt in ein Programm. Aktivieren Sie diese Methode, indem Sie die entsprechende Checkbox anklicken. Kurz darauf zeigt Eclipse im linken Bereich die neu erstellte Klasse an. Nach einem Doppelklick darauf öffnet sich ein neues Arbeitsfenster. Ergänzen Sie nun die main()-Methode um folgende Zeile:

```
System.out.println("Hallo Leute, ich bin Arthur, euer Produzent");
```

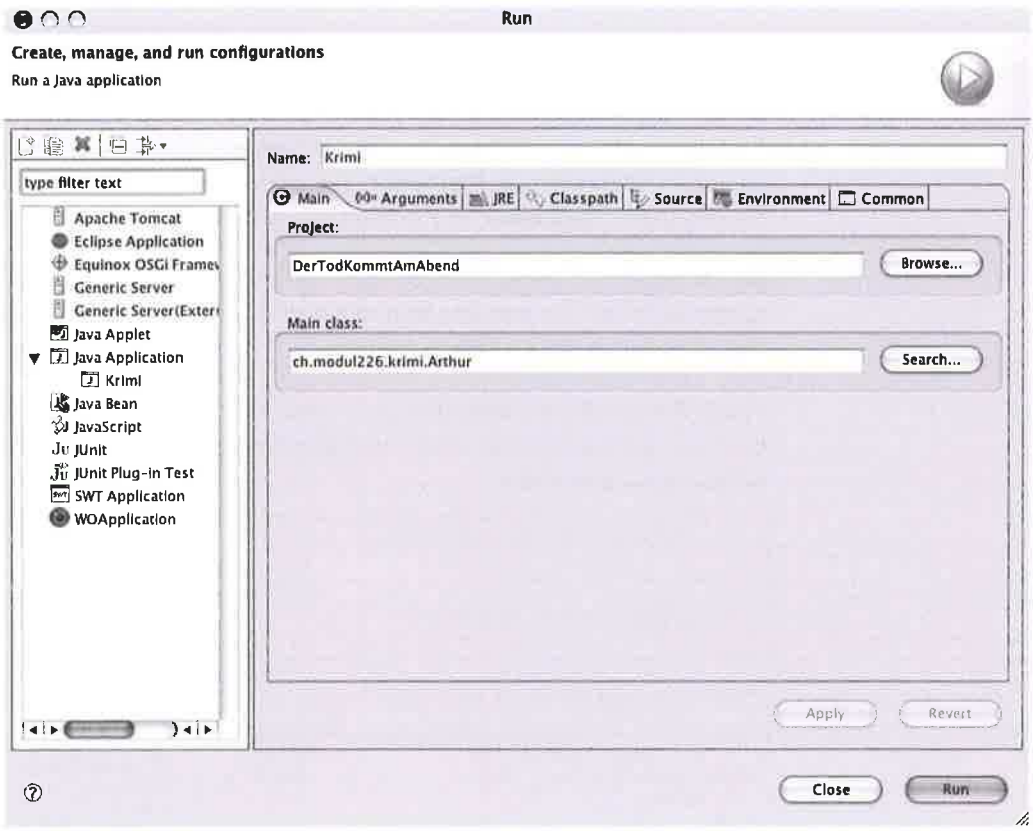
Mit der eingefügten Zeile gibt der Produzent eine Begrüssung aus. Nun sieht Ihre Arbeitsumgebung so aus:

[1-3] Eclipse-Arbeitsumgebung mit Quelltext der Klasse Arthur



Zum Test können Sie das Programm schon einmal laufen lassen, indem Sie in Eclipse eine sogenannte **Run-Konfiguration** erstellen. Wählen Sie dazu im Menü **Run** den Punkt **Run...** Ein Doppelklick im darauf folgenden Dialogfenster auf **Java Application** führt zu einem weiteren Dialogfenster:

[1-4] Run-Konfiguration erstellen



Sie nennen die Run-Konfiguration für unser Beispielprojekt «Krimi». Die Felder «Project» und «Main class» sollten bereits richtig angezeigt werden. Drücken Sie **[Apply],** um die neue Konfiguration zu speichern, und danach **[Run],** um das Programm zu starten. In der Folge erscheint im unteren Bildschirmbereich die Begrüssung von Arthur. Damit ist unser Projekt bereit.

1.3 Regisseur definieren

Als Nächstes brauchen Sie einen Regisseur für unseren Film. In der realen Welt schaltet Produzent **Arthur** zu diesem Zweck ein Stelleninserat in der internationalen Filmpresse (z. B. «Regisseur gesucht») und führt darin auf, welche Aufgaben der Regisseur erfüllen muss und welche Eigenschaften dafür notwendig sind. In Java müssen Sie eine neue Klasse erstellen, um die Stellenbeschreibung zu «übersetzen».

1.3.1 Die Klasse Regisseur beschreibt den Job des Regisseurs

Sie können sich jede **Klasse** als eine Art Stellenbeschreibung vorstellen, wobei die Aufgaben bzw. Tätigkeiten als **Methoden** dargestellt werden. Für unser Beispielprojekt erstellen Sie die Klasse **Regisseur** im Paket **ch.modul226.krimi.**

Da der Regisseur seine Anweisungen vom Produzenten bekommt, muss er keine main()-Methode haben. Sie erinnern sich, die main()-Methode dient zum Starten des Programms, dies wird in der Klasse **Arthur** erledigt.

Fügen Sie die beiden Methoden `go()` und `stop()` ein, damit der Regisseur bereits etwas tun kann. Sie werden bald sehen, was damit möglich ist. Die vollständige Klasse **Regisseur** sollte nun so aussehen:

```
001: package ch.modul226.krimi;
002:
003: public class Regisseur {
004:
004:     public void go() {
006:         System.out.println("Klappe, die erste, Action...");
007:     }
008:
009:     public void stop() {
010:         System.out.println("Alles im Kasten. Szene gestorben.");
011:     }
012: }
```

Schauen Sie sich den obigen Code genau an:

- Auf Zeile 1 legen Sie fest, dass die Klasse **Regisseur** im **Paket ch.modul226.krimi** eingeordnet wird. Wenn alles richtig läuft, erledigt Eclipse das für Sie automatisch.
- Zeile 3 sagt dem Java-Compiler, dass Sie hier die Klasse **Regisseur** definieren.
- Die beiden Methoden `go()` auf den Zeilen 4 bis 7 und `stop()` auf den Zeilen 9 bis 11 sagen, dass der Regisseur die Tätigkeiten `go` und `stop` ausführen kann. Wenn der Regisseur den Auftrag `go` bekommt, soll er eine Filmszene starten, bei `stop` ist die Aufnahme erledigt. Selbstverständlich kann ein echter Regisseur noch viele weitere Dinge tun. Sie werden dem Regisseur im Verlauf dieses Projekts noch weitere Tricks beibringen. Vorerst begnügen Sie sich mit dieser einfachen Stellenbeschreibung.

Hinweis

▷ Das Schlüsselwort `public` regelt den Zugriff. In Java können einzelne Programmteile «versteckt» werden. Genaueres dazu erfahren Sie in Kapitel 8.3, S. 80.

1.3.2 Der Regisseur wird angestellt

Auf das Stelleninserat melden sich viele Personen, die den Job als Regisseur haben wollen. Nach mehreren Interviews entscheidet sich Arthur für David und stellt ihn ein. Übertragen auf die Java-Sprache bedeutet dies: Es wird eine **Instanz** der Klasse **Regisseur** erzeugt. Zu diesem Zweck wird mit dem **Operator** `new` ein Objekt des Typs `Regisseur` erstellt. Für unser Beispielprojekt fügen Sie in der Klasse **Arthur.java** folgende Zeile ein:

```
Regisseur david = new Regisseur();
```

Die main()-Methode sieht nun wie folgt aus:

```
public static void main(String[] args) {
    System.out.println("Hallo Leute, ich bin Arthur, euer Produzent");

    Regisseur david = new Regisseur();
}
```

Im obigen Codeabschnitt wird eine **Variable** `david` von Typ `Regisseur` deklariert. Mittels `new Regisseur()` wird ein neuer Regisseur erzeugt. In der Variablen `david` wird durch die Zuweisung `=` eine Referenz auf diesen Regisseur festgehalten.

1.3.3 Klappe und Action

Produzent Arthur will Regisseur David rasch etwas zu tun geben und lässt ihn erste Probeaufnahmen machen. Fügen Sie dazu unterhalb von `new Regisseur()` diese Zeilen ein:

```
david.go();
david.stop();
```

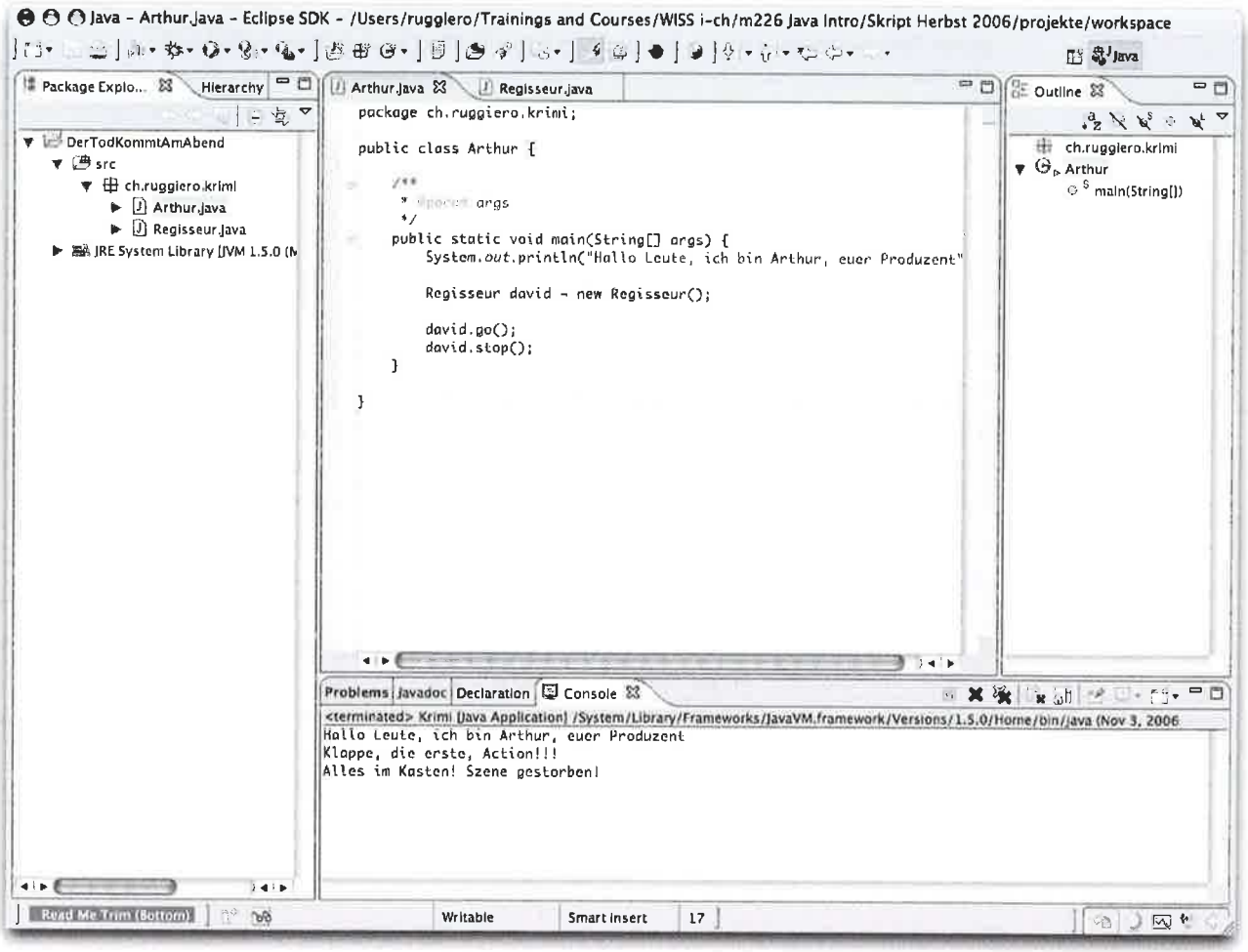
Die main()-Methode sieht nun wie folgt aus:

```
public static void main(String[] args) {
    System.out.println("Hallo Leute, ich bin Arthur, euer Produzent");

    Regisseur david = new Regisseur();

    david.go();
    david.stop();
}
```

Nun lassen Sie das Programm laufen, indem Sie in Eclipse den grünen Startknopf links oben in der Toolbar anklicken.

Die zuvor erstellte Run-Konfiguration wird automatisch übernommen. Sie sehen: Es ist einfach, David eine **Anweisung** zu geben.

[1-5] Der Krimi läuft bereits: Code und die Ausgabe des Programms



1.4 Mörder und Opfer definieren

In unserem Beispielkrimi gibt es einen Mörder und ein Opfer. Leider hat der Mörder Pech und wird von Zeugen beobachtet. Zudem schreitet die Polizei bald ein. Doch alles schön der Reihe nach: Zuerst brauchen Sie die Rollenbeschreibungen, danach müssen Sie die Rollen besetzen und schliesslich kann der Regisseur die Schauspieler handeln lassen.

1.4.1 Was macht der Mörder?

Der Mörder soll sein Opfer mit mehreren Messerstichen tödlich verletzen, dabei sadistisch grinsen und danach wegrennen. Erstellen Sie dazu in unserem Projekt folgende Klasse **Moerder.java:**

```
package ch.modul226.krimi;


public class Moerder {


    public void zustechen() {
        System.out.println("Mörder: sticht heftig zu");
    }


    public void sadistischGrinsen() {
        System.out.println("Mörder: Ha. Ha. Ha. Ha.");
    }


    public void wegrennen() {
        System.out.println("Mörder: rennt weg");
    }


}
```

Hinweis

▷ Beachten Sie die Schreibweise: Klassen- und Dateinamen dürfen keine Sonderzeichen und Umlaute enthalten.

1.4.2 Was macht das Opfer?

Das Opfer Johnny soll nach Hilfe rufen, schreien, stöhnen und sterben können. Erstellen Sie dazu folgende Klasse **Opfer.java:**

```
package ch.modul226.krimi;

public class Opfer {

    public void hilfeRufen() {
        System.out.println("Opfer: Hilfe, Hilfe.");
    }

    public void schreien() {
        System.out.println("Opfer: Oh, Ah, HILFE, AUA.");
    }

    public void stoehnen() {
        System.out.println("Opfer: stöhn, AAAAHHH, Aaahh, aaahh, ahhhhhhh");
    }

    public void sterben() {
        System.out.println("Opfer: Es geht zu Ende, ich sterbe.....");
    }
}
```

1.4.3 Klappe und Action

Für die weiteren Ereignisse in unserem Krimiprojekt engagiert Regisseur David den Mörder Bill und lässt diesen mehrmals zustechen. Während das Opfer Johnny erfolglos um Hilfe schreit und einen qualvollen Tod stirbt, rennt Bill sadistisch grinsend vom Tatort weg. Fügen Sie diese Ereignisse in **Regisseur.java** in der Methode `go()` wie folgt ein:

```
Moerder bill = new Moerder();
Opfer johnny = new Opfer();

johnny.hilfeRufen();
bill.zustechen();
johnny.schreien();
bill.sadistischGrinsen();

for (int i = 0; i < 5; i++) {
    bill.zustechen();
}

johnny.stoehnen();
bill.sadistischGrinsen();
johnny.sterben();
bill.wegrennen();
```

Im obigen Programmablauf engagiert der Regisseur einen Mörder und ein Opfer und gibt ihnen folgende Anweisungen:

- Johnny, das Opfer, ruft um Hilfe, doch niemand scheint ihn zu hören.
- Mörder Bill sticht ein erstes Mal zu und Johnny schreit.
- Bill grinst sadistisch und sticht erneut zu.
- Der arme Johnny kann nur noch stöhnen und stirbt langsam, während Bill weiter sadistisch grinst.
- Nachdem Johnny gestorben ist, rennt Bill, der Mörder, weg.

1.5 Zeugen definieren

Unser Beispielprogramm geht weiter: Jemand hat die Hilferufe von Johnny gehört und den Mörder Bill beim Wegrennen beobachtet. Es gibt also einen Zeugen des Mordes. Die Rolle des Zeugen besteht darin, zweckdienliche Aussagen zum Geschehen zu machen, damit der Mörder überführt werden kann. Die Klasse **Zeuge.java** wird wie folgt definiert:

```
package ch.modul226.krimi;


public class Zeuge {


    public void aussagen() {
        System.out.println("Zeuge: Der Mörder ist gross und blond.");
    }
}
```

Für das Engagement des ersten Zeugen muss in der go()-Methode der Klasse **Regisseur** folgender Eintrag vorgenommen werden:

```
Zeuge einZeuge = new Zeuge();
```

In unserem Beispielprojekt ist noch ein weiterer Zeuge zugegen, der eine Aussage zum Vorfall machen kann. Der folgende Eintrag sorgt für einen zweiten Zeugen:

```
Zeuge zweiterZeuge = new Zeuge();
```

Regisseur David testet nun die Aussagen der beiden Zeugen wie folgt:

```
einZeuge.aussagen();
zweiterZeuge.aussagen();
```

Was kommt dabei heraus? Beide Zeugen machen dieselbe Aussage, weil die Aussage in der Klasse fest codiert ist. Eigentlich sollte die Klasse aber nur definieren, was ein Zeuge tun kann und wie er dies kann. Wenn Sie den Text der Zeugenaussage direkt in die Klasse schreiben, gehen Sie einen Schritt zu weit.

Wie können Sie erreichen, dass jeder Zeuge eine eigenständige Aussage macht? Damit sich jeder Zeuge seine Aussage merken kann, müssen Sie ihm ein Gedächtnis verschaffen und in der entsprechenden Klasse eine **Instanzvariable** definieren. Als idealen Datentyp dafür bietet Java den **String** an. Ergänzen Sie die Klasse **Zeuge** um eine Zeile und ändern Sie die Methode `aussagen()` wie folgt:

```
package ch.modul226.krimi;

public class Zeuge {

    private String aussage = "Ich habe keine Ahnung.";

    public void setAussage(String text) {
        aussage = text;
    }

    public void aussagen() {
        System.out.println("Zeuge: Der Mörder ist gross und blond.");
        System.out.println(aussage);
    }
}
```

Hinweis

▷ Im obigen Programmcode taucht neben dem Schlüsselwort `public` das Schlüsselwort `private` auf. Dieses sorgt dafür, dass die Variable `aussage` ausserhalb der Klasse unsichtbar ist. Näheres dazu erfahren Sie in Kapitel 8.3, S. 80.

Bevor Sie sich den Code genauer ansehen, lassen Sie das Programm erneut laufen. Welche Aussage erhalten Sie? Beide Zeugen haben keine Ahnung. Das ist zwar nichts Neues, trotzdem sind Sie nun einen Schritt weiter. Denn nun kann Regisseur David jedem einzelnen Zeugen einen eigenen Text geben. Erweitern Sie dazu die Methode `go()` in der Klasse **Regisseur** um die beiden fett gedruckten Zeilen:

```
Zeuge einZeuge = new Zeuge();
einZeuge.setAussage("Er ist dorthin gerannt");

Zeuge zweiterZeuge = new Zeuge();
zweiterZeuge.setAussage("Er ist gross und blond.");
```

In der Klasse **Zeuge** haben Sie die Variable `aussage` vom Datentyp `string` definiert und ihr den Wert «Ich habe keine Ahnung» zugewiesen. Die Methode `setAussage()` nimmt als Eingabeparameter einen String entgegen und weist diesen der Variablen `aussage` zu. Damit kann der Regisseur jedem Zeugen einen separaten Text zuordnen. Die Klasse **Zeuge** besagt nur noch, dass jeder Zeuge eine Variable haben soll, die einen String speichern kann. Mit dem Aufruf von `setAusssage()` kann diese Variable mit Inhalt gefüllt werden. Die Methode `aussagen()` greift ebenfalls auf diese Variable zu. Indem Sie die Variable `aussage` bei der Deklaration definieren `("Ich habe keine Ahnung")`, hat jeder Zeuge etwas zu sagen, auch wenn der Regisseur vergisst, eigene Texte zu vergeben.

1.6 Kommissar definieren

Nun wird es Zeit, dass die Polizei eingreift. Regisseur David schreibt den Job des verantwortlichen Polizisten aus und stellt Hercule als Kommissar ein.

1.6.1 Die Klasse Polizist beschreibt den Job des Polizisten

Erstellen Sie eine Klasse **Polizist** mit folgenden drei Methoden:

```
package ch.modul226.krimi;

public class Polizist {

    public void zeugeBefragen(Zeuge z) {
        z.aussagen();
    }

    public void verfolgen() {
        System.out.println("Warte nur, gleich habe ich dich...");
    }

    public void verhaften() {
        System.out.println("Sie sind verhaftet. Jede Aussage kann gegen Sie verwendet werden.");
        System.out.println("Sie haben das Recht, die Aussage zu verweigern.");
    }
}
```

Beim Aufruf der Methode `zeugeBefragen()` übergibt diese eine Referenz auf ein Objekt **Zeuge**. Beim ersten Aufruf handelt es sich um den Zeugen **einZeuge,** beim zweiten Aufruf wird auf **zweiterZeuge** referenziert.

1.6.2 Was macht der Kommissar?

Kommissar Hercule kann verfolgen, verhaften und Zeugen befragen. Schauen Sie sich nun die Methode `zeugeBefragen()` etwas genauer an. Sie verlangt einen Eingabewert vom Typ `Zeuge`. Zeuge? Ist das ein Datentyp? Das ist doch eine Klasse.

Stimmt! In Java gilt: Klassen sind Datentypen. Eine Klasse beschreibt, wie sich Objekte verhalten und welche Eigenschaften sie haben. Wenn David mit dem Finger auf einen Schauspieler zeigt und dem Kommissar sagt, er soll den Zeugen befragen, dann muss dieser ein Zeuge sein, sonst klappt der Aufruf von `zeugeBefragen()` nicht.

Nun soll der Regisseur David den Kommissar Hercule damit beauftragen, die Zeugen zu befragen. Ändern Sie dazu die Methode `go()` in der Klasse **Regisseur** wie folgt ab:

```
einZeuge.aussagen();
zweiterZeuge.aussagen();
Polizist hercule = new Polizist();
hercule.zeugeBefragen(einZeuge);
hercule.zeugeBefragen(zweiterZeuge);
```

Im obigen Codeabschnitt engagiert Regisseur David Kommissar Hercule und gibt diesem den Auftrag, die beiden Zeugen einzeln zu befragen. Dabei zeigt er zuerst auf den ersten Zeugen und danach auf den zweiten Zeugen.

1.6.3 Klappe und Action

Regisseur David gibt letzte Anweisungen für die Mordszene in unserem Beispielprojekt. Dazu gehören eine kurze Verfolgungsjagd und die anschliessende Verhaftung des Mörders Bill. Fügen Sie dazu in der Methode `go()` folgende Aktionen ein:

```
hercule.verfolgen();
bill.wegrennen();
hercule.verfolgen();
hercule.verhaften();
```

Die **objektorientierte Programmierung (OOP)** geht von der Idee aus, dass «Dinge der realen Welt» bestimmte **Eigenschaften** besitzen und ein bestimmtes **Verhalten** zeigen. Wenn Software einen Teil der Realität abbilden soll, muss sie ebenfalls Eigenschaften, Verhaltensweisen und Kombinationen davon darstellen können.

Klassen beschreiben solche Eigenschaften und Verhalten. **Objekte** sind konkrete **Instanzen** von Klassen. Objekte kommunizieren miteinander, indem sie sich **Nachrichten** senden und auf Nachrichten reagieren. Von aussen werden Objekte nur durch ihr Verhalten wahrgenommen. Ihre Eigenschaften zeigen sie nur als Antwort auf eine Frage. Mit anderen Worten: Das «Innenleben eines Objekts» ist von aussen **nicht sichtbar.**

Repetitionsfragen

1 Was verstehen Sie unter einem Objekt im Zusammenhang mit der OOP? Antworten Sie in einem Satz.

2 Was verstehen Sie unter einer Klasse im Zusammenhang mit der OOP? Antworten Sie in einem Satz.

2 Grundsätze der objektorientierten Programmierung

In diesem Kapitel wird aufgezeigt, wie es zur objektorientierten Programmierung kam und welche Grundsätze bei der objektorientierten Programmierung massgebend sind.

2.1 Probleme der prozeduralen Programmierung

In den 1960er-Jahren wurden die Computerprogramme immer umfangreicher und komplexer. Bald einmal ergaben sich dadurch grosse Probleme, die in den 1970er-Jahren zu einer regelrechten **Softwarekrise** führten. Bei der Analyse dieser Softewarekrise wurden hauptsächlich die nachfolgend beschriebenen Problembereiche und Ursachen identifiziert.

2.1.1 Mangelhafte Produktivität

Die meisten Programmierer schrieben praktisch den gesamten Code in jedem Programm neu. Oft lassen sich zwar Codeblöcke aus bestehenden Programmen wiederverwenden, indem sie im Texteditor kopiert und in das neue Programm übernommen werden. In vielen Fällen muss der übernommene Code aber mehr oder weniger angepasst, d. h. verändert werden. **Funktionsbibliotheken,** wie sie etwa vom Betriebssystem zur Verfügung gestellt werden, bieten zwar eine gewisse Wiederverwendbarkeit, allerdings ist diese i. d. R. stark eingeschränkt, weil die meisten **Routinen** spezialisiert sind.

2.1.2 Kosten der Programmpflege

Nur ein kleiner Teil der Programmierarbeit bestand in der Entwicklung neuer Programme. Der weitaus grössere Teil bestand in der **Überarbeitung bestehender Programme** bzw. in der Anpassung an neue Anforderungen. Grundsätzlich ist dies ein gutes Zeichen, denn Anwendungsprogramme, die intensiv genutzt werden, haben **neue Benutzerbedürfnisse** zur Folge. Demgegenüber verschwinden Anwendungen, die wenig oder gar nicht genutzt werden, früher oder später vom Markt oder sie werden ersetzt. Ständige Anpassungen gehören daher zu jedem guten Programm, führen aber auch zu **hohen Unterhaltskosten**.

2.1.3 Unterschiedliche Methoden, Modelle und Begriffe

Ein weiteres Problem waren die Verwendung **unterschiedlicher Vorgehensmethoden und Modelle** für verschiedene Entwicklungsphasen (Analyse, Design, Implementation) sowie der Gebrauch unterschiedlicher **Fachbegriffe.** Während ein Kunde bestimmte Probleme gelöst bzw. Geschäftsprozesse unterstützt haben möchte, wird die Problembeschreibung vom Entwickler über mehrere Stufen hinweg abstrahiert und wieder konkretisiert. Dabei spricht der Entwickler von Datenfluss-Diagrammen und Funktionsgraphen oder sogar in kryptischer Programmiersprache und der Kunde wird von Beginn an «abgehängt». Ob am Schluss dann das Programm vorliegt, das der Kunde wollte, ist so nicht sichergestellt.

2.1.4 Top-down-Ansatz

Benutzerorientierte, interaktive Programme sind mittels Top-down-Ansatz nur schwer oder gar nicht umzusetzen, weil sie keinen **Einstiegspunkt** (Top) haben. Die Zerlegung eines Problems in immer kleinere Teilprobleme kann zudem den **Blick auf Zusammenhänge** verhindern und zu **funktionalen Ähnlichkeiten** mit anderen Applikationsteilen führen. Der

Top-down-Ansatz verstärkt zudem die Tendenz, dass eine Spezifikation den ganzen Code durchdringt. Dadurch wird die Wiederverwendbarkeit des Codes stark eingeschränkt.

2.1.5 Modularisierung

Zusammengehöriger Code wurde in **Modulen** zusammengefasst. Eine Modularisierung hat den Vorteil, dass bei neuen Anforderungen ggf. nur das betroffene Modul geändert werden muss. Zudem können einzelne Module in anderen Projekten wiederverwendet werden. In der Praxis ist es nun aber oft so, dass ein vorgefertigtes Modul nicht immer genau das macht, was der Anwender wünscht. Um dies zu erreichen, müssen entweder die Anforderungen oder die Module angepasst werden. Entweder müssen die Anwender also Abstriche machen oder die Wiederverwendbarkeit der Module wird aufgegeben. Module müssen daher gleichzeitig «geschlossen» und «offen» sein: **Geschlossen** heisst, dass ein Modul funktioniert und verwendet werden kann, ohne dass der zugehörige Code offengelegt und bearbeitet werden muss. **Offen** heisst, dass ein Modul erweitert werden kann, ohne dass der geprüfte bzw. getestete Code verändert wird. Dies ist beispielsweise bei **Funktionsbibliotheken** meistens nicht der Fall.

2.2 Ursachenanalyse und Lösungsansatz

Die Ursachen für die Probleme «Mangelhafte Produktivität» und «Kosten der Programmpflege» lagen in der Art und Weise, wie in der klassischen prozeduralen Programmierung Anwendungen entworfen wurden: Gestartet wurde mit einer Beschreibung, was das Programm alles machen bzw. können soll. Das Lastenheft wurde mit einem Top-down-Ansatz kombiniert, der die gewünschten Funktionen in immer kleinere Einheiten aufteilt. Dieser Ansatz ist zwar bei **statischen Programmen** brauchbar, bei **dynamischen Programmen** aber nicht[1]. Er führt u. a. dazu, dass die gleiche «Funktionalität» im Programm mehrfach vorkommt und gleiche oder ähnliche Konstruktionen an vielen Stellen auftauchen:

```
switch(mailtyp) {
    case PHONE: ….
    case FAX: ….
    ….
}
```

Wenn nun zusätzliche Funktionen (z. B. ein E-Mail) gewünscht werden, müssen eventuell Dutzende von Stellen im Programmcode angepasst werden. Dies ist nicht nur aufwendig, sondern auch eine fehleranfällige Arbeit.

Um die Probleme der klassischen Programmierung zu vermeiden, entstand bereits Ende der 1960er-Jahre die Idee der **objektorientierten Programmierung.** Damals reichte allerdings die Rechenkapazität der erhältlichen Computer noch nicht für eine produktive Umsetzung dieser Idee. Dies änderte sich erst zu Beginn der 1990er-Jahre.

[1] Unter einem statischen Programm wird eine Software verstanden, die immer gleichartige Berechnungen durchführen muss, weil sich die Rahmenbedingungen nicht verändern (beispielsweise zur Berechnung der Flugbahn von Satelliten). Dynamische Programme sind dagegen flexibel und interaktiv, d. h., sie reagieren auf veränderliche Rahmenbedingungen und orientieren sich am Benutzerverhalten bzw. an den Eingaben der Anwender.

2.3 Grundideen der objektorientierten Programmierung

2.3.1 Daten und Funktionen

Bereits in der prozeduralen Programmierung wird zwischen **Daten** und **Operationen** unterschieden, die auf die Daten einwirken. Während Daten beständig bzw. langlebig sind, «existieren» Operationen nur so lange, wie sie für eine bestimmte Aufgabe **(Prozedur bzw. Funktion)** benötigt werden. Operationen haben also keinen bleibenden Zustand. Moderne Programmiersprachen unterstützen diese Unterscheidung und stellen geeignete **Konstruktionen für Daten und Funktionen** zur Verfügung. Bei der OOP wird die gleiche Unterscheidung getroffen, mit **Klassen** und **Objekten** aber eine weitere **Abstraktionsebene** hinzugefügt: Daten und Operationen werden zu modularen Einheiten zusammengefasst. Als Programmierer bilden Sie mit diesen Einheiten ein **Beziehungsgeflecht,** um die gewünschte Funktionalität des Programms zu realisieren.

2.3.2 Objekte

Objekte der objektorientierten Programmierung sind ähnlich wie Objekte der realen Welt. Jedes Objekt hat bestimmte **Eigenschaften** (Daten), definierte **Fähigkeiten** (Operationen) und kann ein bestimmtes **Verhalten** (Funktionen bzw. Prozeduren) zeigen. Diese Nähe zu Objekten der realen Welt macht die OOP so mächtig. Nehmen Sie beispielsweise das Objekt Getränkeflasche. Dieses kann unterschiedliche Eigenschaften (z. B. leer, voll, offen, geschlossen, aus Plastik, aus Glas) sowie verschiedene Fähigkeiten (z. B. stapelbar, verschliessbar, lichtdurchlässig, temperaturempfindlich) haben und dadurch ein spezifisches Verhalten aufweisen.

2.3.3 Aussen- und Innensicht

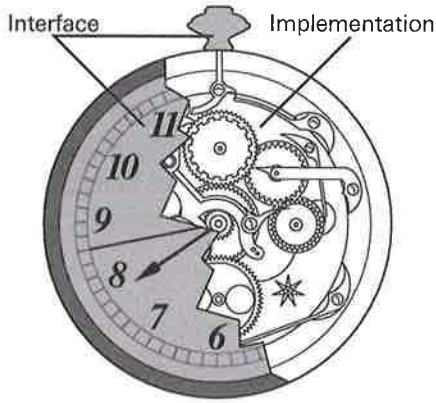
Wenn Sie ein Anwendungsprogramm entwickeln möchten, müssen Sie in der Lage sein, **Abstraktionen** vorzunehmen und in Form von **Programmcode** auszudrücken. Nehmen Sie z. B. einen Datensatz, der die Eigenschaften einer Person beschreibt. Dieser Datensatz ist natürlich nicht mit der betreffenden Person gleichzusetzen, erlaubt aber eine eindeutige Identifikation. Der Datensatz ist eine Abstraktion dieser Person.

Verschiedene Konstruktionen in prozeduralen Programmiersprachen helfen Ihnen dabei, solche Abstraktionen vorzunehmen. Von C her kennen Sie beispielsweise Funktionen und den Datentyp `struct`, mit denen Sie bestimmte Daten zusammenfassen können. Solche Konstruktionen erlauben Abstraktionen, unterscheiden aber weiterhin zwischen **Daten** und **Funktionen,** wobei die Funktionen auf die Daten einwirken. Vergleichen Sie dazu den folgenden Codeabschnitt:

```
// eine Datenstruktur in C, die Personen beschreibt
struct Person {
    char name[30];
    char vorname[30];
    int alter;
};
```

Mit Variablen von Typ `struct Person` oder mithilfe von Funktionen können Sie arbeiten, ohne sich um deren **inneren Aufbau** zu kümmern. Sie können also Funktionen erstellen, ohne sich darum zu kümmern, wer und in welchem Zusammenhang sie später anwenden wird. Als Programmierer können Sie also zwischen der **Aussensicht** (dies ist das Interface) und der **Innensicht** (dies ist die Implementation) unterscheiden.

[2-1] Interface und Implementation



2.3.4 Verhalten von Objekten

Stellen Sie sich vor, dass Sie die Datenstruktur `struct Person` vorliegen haben. Wenn Sie nun eine Person wiederverwenden möchten, «verstecken» Sie die **Datenstruktur** und erstellen davon unabhängige **Funktionen,** die mit dieser Datenstruktur arbeiten können. Somit muss ein (ggf. anderer) Programmierer nicht wissen, dass es in dieser Datenstruktur beispielsweise eine Variable `alter` vom Datentyp `int` gibt, weil er einfach die Funktion `alter()` aufrufen kann. Als Programmierer steht Ihnen quasi ein **Interface** für die Arbeit mit Personen zur Verfügung. Sie müssen lediglich die gewünschte Funktion aufrufen und als Eingabeparameter die Variable der aktuellen Person übergeben. Dadurch wird der «innere Aufbau» von `struct Person` unwichtig und Sie können sich vollständig darauf konzentrieren, was die Funktion macht.

Wie die Daten organisiert sind, kann Ihnen egal sein. Weshalb soll sich ein Programmierer noch um die Daten kümmern, wenn alle Zugriffe durch die Funktionen erledigt werden? Und warum sollen die Daten noch von Funktion zu Funktion übergeben werden? Dadurch, dass die Daten als «interne Details» betrachtet werden, wird die Programmierarbeit stark vereinfacht und Sie können sich ganz auf das **Verhalten der Objekte** fokussieren.

Analog dazu ist ein **Objekt** mehr als eine Ansammlung von Funktionen. Es ist vielmehr ein **Bündel von Verhaltensweisen,** die durch gemeinsam genutzte Daten ermöglicht werden. Um eine Funktion anwenden zu können, müssen Sie zuerst ein Objekt mit einer internen Datenstruktur erstellen. Danach sagen Sie dem Objekt, was es machen soll. Dabei denken Sie an Handlungen, die das Objekt ausführen soll, und nicht mehr in Funktionsaufrufen. Dieser Übergang vom **Denken in Funktionen und Datenstrukturen** hin zum **Denken in Verhaltensweisen von Objekten** ist die Basis der objektorientierten Programmierung und Java ist eine Programmiersprache, die diese Denkweise unterstützt.

2.3.5 Klassen und Instanzen

Klassen sind die «Baupläne für Objekte». Sie beschreiben, welche Daten und Verhalten ein Objekt aufweisen kann. In unserem einführenden Beispielkrimi haben Sie die Klasse als eine Art von **Stellenbeschreibung** kennengelernt. Hier wird definiert, was der Regisseur, der Mörder, das Opfer oder die Zeugen alles machen können. Doch erst die konkreten Objekte (z. B. Regisseur David oder Polizist Hercule) sind als sogenannte **Instanzen** in der Lage, in Aktion treten und zu interagieren.

2.3.6 Methoden und Nachrichten

Im Krimi gibt Regisseur David den Schauspielern Anweisungen für bestimmte Handlungen. In der objektorientierten Programmierung spricht man in diesem Zusammenhang von **Nachrichten,** d. h., ein Objekt schickt einem anderen Objekt eine bestimmte Nachricht. Wenn das Empfängerobjekt die Nachricht versteht, reagiert es, indem es eine Funktion bzw. in Java eine **Methode** auslöst. Weil die Programmiersprache Java auf die Syntax von C aufbaut, sieht es so aus, als ob eine Funktion aufgerufen wird, wenn Regisseur David dem Mörder Bill die Anweisung `sadistischGrinsen` gibt. Sie werden in Kapitel 3.2, S. 37 sehen, dass die Unterscheidung zwischen Nachricht und Methode Sinn macht. In anderen OO-Sprachen wird diese Trennung auch in der Syntax sichtbar.

2.4 Eigenschaften und Beziehungen

Objekte zeigen Verhalten. Damit Objekte zueinander in eine Beziehung treten können, müssen sie sich gegenseitig kennen und ansprechen können. In diesem Kapitel erfahren Sie, wie solche Beziehungen realisiert werden und funktionieren.

2.4.1 Attribute

Attribute sind Daten, die Eigenschaften von Objekten definieren. Jedes Auto hat beispielsweise eine bestimmte Farbe und eine bestimmte Anzahl von Türen. Farbe und Anzahl von Türen sind also Attribute des Objekts Auto, wobei jedes Auto verschiedene Ausprägungen (Werte) dieser Attribute aufweisen kann (z. B. rot, blau oder grau bzw. 3, 4 oder 5 Türen).

In der OO-Programmierung können Werte von Attributen mithilfe von **Instanzvariablen** gespeichert werden. Der Name der Instanzvariablen entspricht normalerweise dem Namen des Attributs. Sie haben aber in Kapitel 2.3.3, S. 31 die Unterscheidung zwischen Innen- und Aussensicht bei der OOP kennengelernt. Von aussen ist nicht erkennbar, wie Objekte die Werte ihrer Attribute speichern. Autos haben das Attribut «Farbe», Menschen haben das Attribut «Alter». Wie und wo die Werte für die Farbe und das Alter gespeichert sind, ist nicht wichtig, solange Sie ein Auto-Objekt nach seiner Farbe und ein Mensch-Objekt nach seinem Alter fragen können. Vielleicht ist das Alter eines Menschen nirgendwo gespeichert, sondern wird bei Bedarf anhand des Jahrgangs berechnet. Trotzdem besitzen Mensch-Objekte die Eigenschaft «Alter».

2.4.2 Beziehungen

Damit Objekte miteinander kommunizieren können, müssen sie sich gegenseitig kennen, d. h. eine **Beziehung** zueinander haben. Im Rahmen der OOP können zwischen Objekten folgende **Beziehungsarten** unterschieden werden:

- Die allgemeine Beziehung «Ich kenne dich» wird **Assoziation** genannt. Dazu gehören etwa die Beziehungen zwischen dem Regisseur und seinen Schauspielern.
- Enger bzw. stärker sind Beziehungen der Art «Du gehörst zu mir». Eine solche Beziehung nennt man **Aggregation.** Ein Auto hat beispielsweise Räder, d. h., Räder sind Bestandteile eines Autos bzw. Räder gehören zum Auto. Wichtig ist, dass die Räder zwar zum Auto gehören, aber auch ohne Auto existieren. Räder lassen sich abmontieren und an einem anderen Auto wieder neu montieren.
- Die **Komposition** ist die stärkste Beziehungsart zwischen Objekten. Sie besagt «Du gehörst zu mir und ohne mich gibt es dich gar nicht». Ein Haus hat beispielsweise Räume und diese Räume gehören zum Haus. Gleichzeitig ist es nicht möglich, einen Raum vom Haus zu lösen. Mit anderen Worten: Der Raum existiert nur als Bestandteil des Hauses.

Hinweis

▷ Jede Komposition ist in abgeschwächter Form eine Aggregation, und jede Aggregation ist auch eine Assoziation. Es ist wichtig, dass Sie diese drei Beziehungstypen unterscheiden können. Allerdings ist es nicht so, dass jede OO-Programmiersprache für jede Beziehungsart eine eigene Syntax bereitstellt. Bei Java müssen Sie solche Abhängigkeiten selber programmieren. Andere OO-Sprachen kennen spezielle Anweisungen, um die Stärke der Beziehung zwischen Objekten auszudrücken.

Sie haben in Eclipse ein neues Projekt, das Paket **ch.modul226.krimi** und danach die Klasse Arthur erstellt. Arthur ist der Produzent des Krimis und hat eine **main()-Methode.** Die main()-Methode bildet den Einstieg für die gesamte Applikation.

Eine **Klasse** kann als Stellenbeschreibung aufgefasst werden, in der die Tätigkeiten als Methoden dargestellt sind. Mit `new` erhalten Sie ein Objekt, dem Sie Anweisungen geben können. Dieses Objekt lässt sich in einer Variablen festhalten. Als Datentyp für eine solche Variable wird der Klassenname verwendet.

Im Rahmen des Krimiprojekts beschreibt eine Klasse, wie sich ein Schauspier zu verhalten hat. Die Klasse kann wie ein Datentyp verwendet werden. **Objekte** sind konkrete voneinander klar unterscheidbare Instanzen dieser Klassen. Objekte können miteinander kommunizieren und dabei sogar Referenzen auf andere Objekte weitergeben.

Attribute sind Eigenschaften von Objekten. Die Klasse beschreibt die Eigenschaft, jedes Objekt hat aber seinen eigenen individuellen Wert für das Attribut. Objekte müssen miteinander in Beziehung stehen, damit sie kommunizieren können. Je nach Stärke der Bindung spricht man von allgemeiner **Assoziation** («ich kenne dich»), **Aggregation** («du gehörst zu mir») oder **Komposition** («dich gibt es ohne mich gar nicht»).

Repetitionsfragen

3 Umreissen Sie in einem Satz, wofür die main()-Methode benötigt wird.

4 Wie können Sie in einem Projekt mit Eclipse ein neues Paket erzeugen?

5 Inwiefern kann die Klasse **Regisseur** mit einem Stelleninserat verglichen werden?

6 Wie lautet der Java-Code zur Definition einer Klasse?

7 Wie erzeugen Sie im Java-Programm ein neues Objekt? Machen Sie ein Beispiel.

8 Was bedeutet die Anweisung `david.go();`?

9 Dürfen Sie in einem Klassennamen Umlaute verwenden? Begründen Sie Ihre Antwort.

10 Wenn Sie in der Klasse **Zeuge** eine Aussage hinterlegen, sagen alle Zeugen das Gleiche. Wie erreichen Sie, dass verschiedene Zeugen unterschiedliche Aussagen machen können?

11 In einem Programmcode finden Sie die folgenden Zeilen. Erklären Sie, was hier passiert.

```
Polizist hercule = new Polizist();

hercule.zeugeBefragen(einZeuge);

hercule.zeugeBefragen(zweiterZeuge);
```

12 Begründen Sie in einem Satz, warum Objekte untereinander Beziehungen haben müssen.

13 Spezifizieren Sie folgende Beziehungstypen:

- Ein Flugzeug hat Propeller.
- Ein Flugzeug steht am Gate 17A.
- Ein Flugzeug verfügt über Passagierkabinen.

3 Konzepte der objektorientierten Programmierung

Nachdem Sie erfahren haben, was Klassen und Objekte sind, lernen Sie in diesem Kapitel wichtige Konzepte der OO-Programmierung kennen.

3.1 Datenkapselung

Sie haben bereits erfahren, dass Objekte Daten und Verhaltensweisen umfassen. In der objektorientierten Programmierung gibt es ein Konzept, das besagt, dass das «Innenleben» eines Objekts und damit seine Daten nur für das Objekt selbst wichtig ist, nicht aber für die «Aussenwelt». Dieses Konzept ist auch unter der Bezeichnung **Datenkapselung**[1] bekannt.

3.1.1 Kapselung im Krimiablauf

Auch wenn es Ihnen vielleicht nicht bewusst ist: In unserem Krimiprojekt haben Sie das Konzept der Datenkapselung bereits kennengelernt. Es wurden Zeugen angelegt, die sich einen Text merken müssen, damit sie auf Befehl des Regisseurs ihre Aussagen machen können. Was würden Sie machen, wenn Sie in einem Film die Rolle eines Zeugen spielen müssten? Vielleicht schreiben Sie den Text auf einen Zettel und stecken diesen in Ihre Jackentasche. Wenn der Aufruf des Regisseurs kommt, nehmen Sie den Zettel hervor und sagen den Text auf. Wenn nun der Regisseur verlangt, dass Sie ohne Jacke spielen sollen, stecken Sie den Zettel mit der Zeugenaussage vielleicht in Ihre Hosentasche. Für Sie ist es also wichtig, dass Sie einen Spickzettel haben und jederzeit wissen, wo dieser steckt. Für den Regisseur ist hingegen entscheidend, dass Sie im richtigen Moment den richtigen Text aufsagen. Wie Sie das machen und wie Sie sich darauf vorbereiten, interessiert ihn nicht. Er wird auch nicht in Ihre Tasche greifen, um nachzusehen, ob Sie den richtigen Zettel dabeihaben, sondern er wird Sie nur nach Ihrer Zeugenaussage fragen.

3.1.2 Kapselung in der Programmierung

Um eine Datenkapselung zu programmieren, sind Mechanismen notwendig, die einem Objekt einen bestimmten Wert zuordnen oder das Objekt nach einem bestimmten Wert fragen. Solche Mechanismen werden in Java mittels **Methoden** realisiert. Normalerweise brauchen Sie dafür zwei Methoden; eine Methode, um den Wert zu «setzen», und eine Methode, um den Wert abzufragen. Diese Methoden nennt man **Zugriffsmethoden**[2]. Weil Java als Namenskonvention für solche Methoden das Präfix[3] `set` bzw. `get` vorsieht und Java eine weitverbreitete Programmiersprache ist, werden diese Zugriffsmethoden auch **setter** oder **getter** genannt.

Hinweis

▷ In Kapitel 2.1.5, S. 30 haben Sie bereits erfahren, dass Module gleichzeitig «offen» und «geschlossen» sein sollten. Geschlossen heisst, dass das Modul existiert, funktioniert und verwendet werden kann, ohne dass der Code offengelegt und bearbeitet werden muss. Diese Forderung wird mit dem Einsatz der Zugriffsmethoden erfüllt.

[1] Engl. Fachbegriff: Data Hiding.

[2] Engl. Fachbegriff: Accessor Methods.

[3] Unselbstständiger, vorangehender Bestandteil eines Worts. Hier: Element, das dem Methodennamen vorangesetzt wird.

3.2 Vererbung

Mittels **Vererbung** können Eigenschaften und Verhaltensweisen einer Klasse auf andere Klassen bzw. Objekte übertragen werden. Voraussetzung für Vererbungen ist eine so genannte **«Ist-ein-Beziehung»** zwischen diesen Klassen bzw. Objekten.

Beispiele:

- Ein Auto ist ein Fahrzeug.
- Ein Mensch ist ein Säugetier.
- Der Mond ist ein Planet.

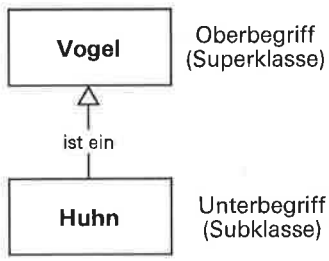
Hinter einer Ist-ein-Beziehung stecken die Begriffe **Generalisierung** und **Spezialisierung.** Solche «Verallgemeinerungen» und «Differenzierungen» treffen Sie in der realen Welt immer wieder an. Folgende Tabelle enthält einige Beispiele dazu:

Generalisierung	Spezialisierung
Computer	• Laptop • Desktop-Rechner • iPad
Fahrzeug	• Auto • Velo • Motorrad
Flugzeug	• Düsenjet • Helikopter • Segelflugzeug
Lebensmittel	• Brot • Butter • Milch
Vogel	• Huhn • Adler • Taube • Pinguin • Strauss

Das Verhältnis zwischen Spezialisierung und Generalisierung kann als «Ist-ein-Beziehung» bezeichnet werden. Ein Huhn ist z. B. ein Vogel und Milch ist ein Lebensmittel. Umgekehrt gilt: Nicht jeder Vogel ist ein Huhn und nicht jedes Lebensmittel ist Milch. Vogel und Lebensmittel sind also **Oberbegriffe** und Huhn und Milch stellen **Unterbegriffe** dar. Beide Begriffe werden in der objektorientierten Programmierung als Klassen dargestellt, wobei der Oberbegriff einer **Superklasse** und der Unterbegriff einer **Subklasse** entspricht.

«Ist-ein-Beziehungen» können im Rahmen der objektorientierten Programmierung mithilfe der grafischen Modelliersprache **UML**[1] wie folgt dargestellt werden:

[3-1] Darstellung der Vererbung (Beispiel)

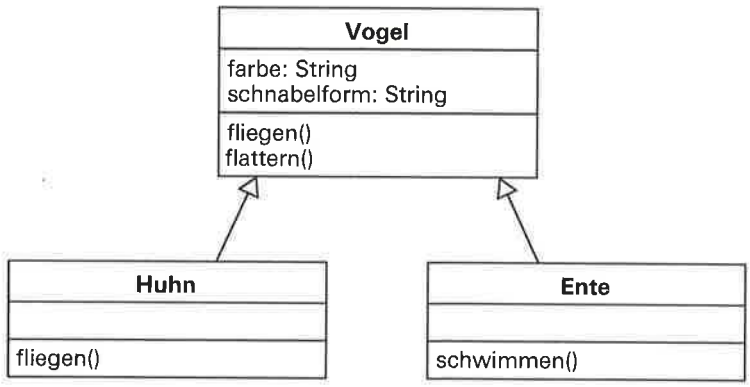


[1] Vergleichen Sie zu UML das Kapitel 4, S. 46.

Wie Sie sehen, werden «Ist-ein-Beziehungen» in UML durch eine Linie mit dem offenen Pfeilsymbol dargestellt. Weil das Huhn vom Vogel abgeleitet werden kann, spricht man anstelle von Subklassen auch von **abgeleiteten Klassen.** Es gibt Eigenschaften und Verhaltensweisen, die für alle Vögel gelten. In der Superklasse **Vogel** werden alle Eigenschaften und Verhaltensweisen zusammengefasst, die einen Vogel ausmachen. Da Hühner und Geier auch Vögel sind, zeigen sie die gleichen Vogel-Eigenschaften und -Verhaltensweisen auf. Zusätzlich können Sie den Subklassen spezielle Eigenschaften und Verhaltensweisen zuordnen, die nur für Hühner bzw. nur für Geier gelten (z. B. pickt Körner, fliegt kaum und tief -> typisch Huhn; isst Aas und Fleisch, fliegt viel und hoch -> typisch Geier).

In der folgenden Grafik sehen Sie oben die Superklasse **Vogel** und unten die davon abgeleiteten Subklassen **Huhn** und **Ente.**

[3-2] Vererbung zwischen Vogel, Huhn und Ente



Die Klasse **Vogel** definiert zwei Attribute[1]; die Farbe des Gefieders und die Form des Schnabels. Damit hat jeder Vogel – egal ob Huhn oder Ente – eine Farbe und eine Schnabelform. In der Klasse **Vogel** gibt es zudem zwei Methoden. Jeder Vogel kann daher im Prinzip fliegen und flattern.

[3-3] Das Huhn Jakob

[3-4] Die Ente Ede

[1] In Wirklichkeit haben Vögel weitere Attribute.

Wenn Sie sich nun die Klasse **Huhn** anschauen, sehen Sie, dass dort ebenfalls eine Methode `fliegen()` definiert ist. Was soll das? Bei Hühnern kann man doch nicht von «fliegen» sprechen, denn meist schaffen sie es nur ein paar wenige Meter weit. Aus diesem Grund bekommt das Huhn eine eigene fliegen()-Methode mit der Ausprägung «ziemlich schlecht». Enten können dagegen ganz passabel fliegen. Aus diesem Grund reicht bei Enten die fliegen()-Methode der Klasse **Vogel** völlig aus. Gleichzeitig können Enten auch gut schwimmen. Deshalb bekommt die Klasse **Ente** eine eigene Methode `schwimmen()`. Demgegenüber weist die Klasse **Vogel** keine schwimmen()-Methode auf, weil dieses Verhalten für Vögel eher untypisch ist.

Wenn Sie nun ein Enten-Objekt erstellen (z. B. den Erpel Ede), dann bekommt Ede alle Eigenschaften der Klasse **Vogel** und zusätzlich alle Attribute der Klasse **Ente**. Das gilt auch für die Methoden, die das Verhalten von Ede beschreiben. Das Huhn Jakob vereinigt alle Attribute und Methoden der Klasse **Vogel** und der Klasse **Huhn**. Man sagt in diesem Zusammenhang auch: Die Klassen **Ente** und **Huhn** erben von der Klasse **Vogel.**

Folgende Punkte gilt es bei **Vererbungen** zu beachten:

- Die Vererbung ist eine Beziehungsform zwischen Klassen. Die Klasse **Huhn** erbt von der Klasse **Vogel** und nicht das Objekt **Jakob.** Jakob ist ein Huhn, das alle Eigenschaften und Verhaltensweisen von Vogel und Huhn in sich vereint. Jakob erbt nicht.
- Normalerweise werden Attribute und Methoden vererbt. Eine solche Vererbung kann mittels Zugriffsrechten gesteuert werden.[1]
- Durch Vererbung gehen prinzipiell keine Fähigkeiten verloren. Bestehende Fähigkeiten können aber überschrieben werden. Wenn z. B. Vögel fliegen können, kann auch jedes Objekt jeder davon abgeleiteten Klasse fliegen. In einer Unterklasse kann höchstens die Art und Weise des Fliegens angepasst werden. Wenn Sie beispielsweise an einen Emu oder Strauss denken und die Klasse **Laufvogel** erzeugen, müssen Sie die Methode `fliegen()` bei der Implementation entsprechend anpassen.
- Grundsätzlich können Sie einer Unterklasse in Java beliebige neue Eigenschaften bzw. Attribute und Verhaltensweisen bzw. Methoden hinzufügen.

3.3 Polymorphismus

Im Kontext der Vererbung treffen Sie immer wieder mal auf den Begriff **Polymorphismus.** Dieser Begriff stammt aus dem Griechischen und bedeutet so viel wie Viel- oder Mehrgestaltigkeit. Was dieser Begriff in der objektorientierten Programmierung bedeutet, soll anhand der Klassen **Huhn** und **Ente** verdeutlicht werden. Vergleichen Sie dazu folgenden Codeabschnitt:

[1] Vergleichen Sie dazu die Kapitel 8.3, S. 80 und 8.6, S. 88.

```
// wir erstellen ein Array für Vögel
Vogel[] vieleVoegel = new Vogel[3];


// Wir füllen das Array mit verschiedenen Vögeln
vieleVoegel[0] = new Vogel();
vieleVoegel[1] = new Ente();
vieleVoegel[2] = new Huhn();


// wir lassen jeden Vogel fliegen
for (short i = 0; i < vieleVoegel.length; i++) {
    vieleVoegel[i].fliegen();
}
```

Sehen Sie sich im obigen Code die fett markierte Zeile innerhalb der for-Schleife genauer an und verfolgen Sie, was hier passiert.

Schleifendurchgang	Aktion
i == 0	Der erste Eintrag aus dem Array wird geholt. Dies ist ein Objekt vom Typ Vogel. Wenn dieser Vogel fliegen gelassen wird, ruft Java die Methode `fliegen()` der Klasse **Vogel** auf. Das ist zu erwarten.
i == 1	Der zweite Eintrag aus dem Array wird geholt. Dies ist ein Objekt vom Typ Ente. Objekte der Klasse **Ente** erben die Methode `fliegen()` von der Klasse **Vogel.** Ente ist ein Vogel, daher wird von Java auch die Methode `fliegen()` der Klasse **Vogel** aufgerufen.
i == 2	Der dritte Eintrag wird geholt. Dies ist ein Huhn. Welche Methode wird nun ausgeführt? Fliegt das Huhn wie ein Vogel, da Vögel im Array stehen? Nein. Obwohl das Array Vögel enthält[1], weiss jedes Objekt, was es wirklich ist. Das dritte Objekt im Array ist effektiv ein Huhn und ein Huhn hat seine eigene fliegen()-Methode. Deshalb wird der dritte Vogel wie ein Huhn fliegen, obwohl keine Typumwandlung vorliegt. Nirgends wird der dritte Vogel als Huhn angesprochen.

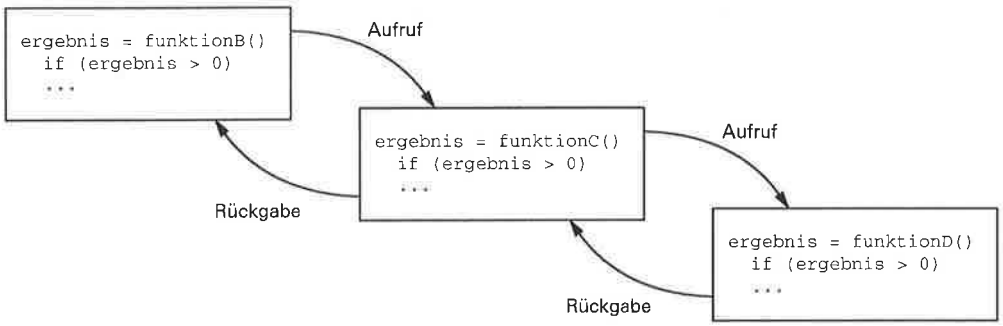
[1] Vergleichen Sie dazu die Deklaration des Arrays in der ersten Zeile des Codebeispiels.

Die Vogel-Objekte passen ihre Form und ihr Verhalten korrekt an, obwohl alle als Objekt der gleichen Klasse **Vogel** angesprochen werden. Vogel-Objekte sind also verwandlungsfähig und können mehrgestaltig sein. In Kapitel 1.4, S. 22 haben Sie erfahren, dass eine Nachricht wie der Aufruf einer Funktion bzw. Methode aussieht. Hier haben Sie nun ein Beispiel, in dem der Unterschied zwischen Nachricht und Methode klar sichtbar wird. Die Nachricht `fliegen()` wird an das Vogel-Objekt geschickt. Erst zur Laufzeit des Programms wird festgestellt, welche Methode als Reaktion auf diese Nachricht aufgerufen wird, diejenige der Klasse **Vogel** oder diejenige der Klasse **Huhn.** Mit anderen Worten: Durch Trennung von Nachricht und Methode wird Polymorphismus erst möglich.

3.4 Exceptions

In der prozeduralen Programmierung werden auftretende Fehler in einem Programm meist durch entsprechende Rückgabewerte und darauf aufbauende Fallunterscheidungen behandelt. Diese **klassische Fehlerbehandlung** sieht wie folgt aus:

[3-5] Klassische Fehlerbehandlung



Was passiert hier? Funktion A ruft Funktion B, diese ruft Funktion C und diese wiederum Funktion D auf. Jede Funktion gibt jeweils einen Ergebniswert an die aufrufende Funktion zurück. Nun stellen Sie sich vor, dass in Funktion D etwas Unvorhergesehenes passiert, sodass der zurückgegebene Wert in Funktion C nicht brauchbar ist. In diesem Fall muss Funktion C irgendwie die aufrufende Funktion B informieren, dass der Rückgabewert nicht brauchbar ist, und am Ende muss auch Funktion A diese Information bekommen.

In dieser Situation ergeben sich verschiedene Probleme. Wie soll die aufgerufene Funktion die Ungültigkeit des Ergebnisses mitteilen? Wenn das ein numerischer Wert ist, besteht vielleicht die Möglichkeit, 0 zurückzugeben oder einen anderen Wert, der kein gültiges Ergebnis ist. In C haben Sie oft **Zeiger** als Rückgabewert. Dort finden Sie den Code, der den Rückgabezeiger auf den Wert NULL testet. Aber was, wenn NULL ein gültiger Rückgabewert ist? Zudem müssen Sie in ihrem Code an vielen Stellen den Rückgabewert auf seine Gültigkeit testen, bevor das Programm weiterarbeiten darf. Das sieht dann vielleicht so aus:

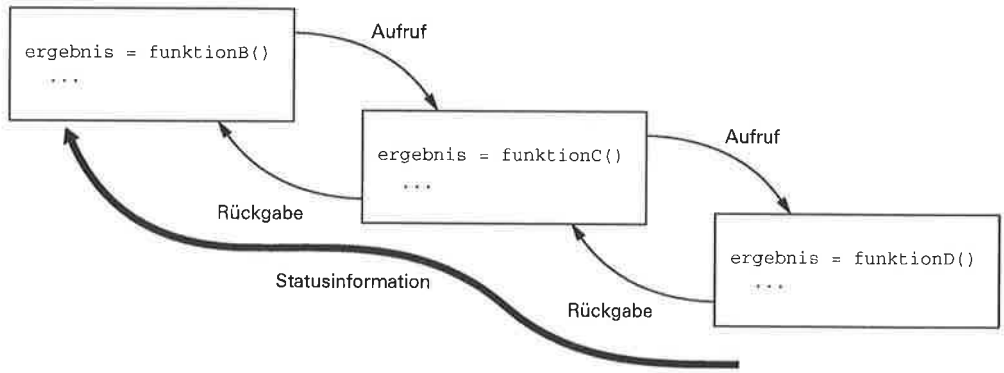
```
int ergebnis = funktionA();
if (ergebnis > 0) {  // alles in Ordnung;
    int zahl = funktionB(ergebnis);
    if (zahl >= 0)  { // alles in Ordnung
    ...
```

In solchen **Fehlerabfragen** droht der eigentliche Code der Funktion zu verschwinden. Übersichtlich kann man einen solchen Code auf jeden Fall nicht mehr nennen. Ausserdem gilt: Fehler treten nur in Ausnahmefällen auf und die Fehlerbehandlung ist daher meistens überflüssig. Idealerweise sollte nur der reguläre Ablauf programmiert werden, ohne dass sich der Programmierer um Fehler kümmern muss – ausser, wenn wirklich etwas schiefgeht. Genau das können Sie in Java und in vielen anderen objektorientierten Programmiersprachen realisieren. Im folgenden Codeabschnitt sehen Sie, wie dies geschieht:

```
try {
    ergebnis = funktionA();
    zahl = funktionB(ergebnis);
    // und viele weitere Anweisungen
}
catch (Exception ex) {
    // hier Fehler behandeln
}
```

Mithilfe einer **try**[1]**-catch**[2]**-Konstruktion** wird der reguläre Programmablauf von der Fehlerbehandlung getrennt. Im **try-Block** stehen sämtliche Anweisungen ohne Fehlerbehandlung. Hier wird so programmiert, als könnten keine Probleme auftreten. Im Anschluss an den try-Block werden alle Fehler abgefangen, die innerhalb des try-Ablaufs auftreten können. Genauer gesagt werden sogenannte **Exceptions**[3] abgefangen, denn Fehler im Programmablauf sollen Ausnahmen sein und bleiben. In der folgenden Abbildung können Sie erkennen, dass die Fehlerbehandlung ausserhalb des regulären Programmablaufs über einen **zusätzlichen Rückgabekanal** stattfindet:

[3-6] Rückgabekanal für Ausnahmen



Sobald irgendwo innerhalb des mit `try` eingefassten Ablaufs ein Problem auftritt, wird der Ablauf an dieser Stelle abgebrochen und es werden keine weiteren Anweisungen aus dem try-Block mehr ausgeführt. Dabei macht es keinen Unterschied, wo das Problem auftritt, direkt in einer Anweisung im try-Block oder beliebig tief verschachtelt in einem Aufruf einer Methode. Damit try-catch möglich ist, muss die gewählte Programmiersprache Exceptions unterstützen. Exceptions sind **Objekte, die «geworfen» werden.** Stellen Sie sich darunter einen Ball vor, der von Methode zu Methode geworfen wird. Irgendjemand muss den Ball auffangen, sonst fällt er zu Boden. Genauso verhält es sich mit Exceptions. Wo ein Problem auftritt, wird jeweils ein Exception-Objekt erstellt und geworfen. Das Exception-Objekt kann so Informationen über das Problem (z. B. eine Fehlermeldung) transportieren. Die aufrufende Methode kann dieses Objekt abfangen und auswerten oder weiterfliegen lassen. Vielleicht fängt es die aufrufende Methode. Als letzte Möglichkeit kann die Applikation selbst die Exception fangen. Exceptions können aber auch ignoriert werden. Je nach Typ der Exception kann dies zum (kontrollierten) Absturz des Programms führen oder die geworfene Exception wird komplett ignoriert und hat keine Konsequenzen.

Hinweis

▷ Normalerweise stehen in jeder OO-Sprache, die Exceptions unterstützt, verschiedene **Exception-Klassen** zur Verfügung. Zudem ist es möglich, eigene Exceptions zu definieren.

3.5 Schnittstellen (Interfaces)

Eine **Schnittstelle** bzw. ein **Interface** kann als Versprechen aufgefasst werden, sich auf eine bestimmte Art und Weise zu verhalten. Schauen Sie sich anhand des Krimiprojekts an, was dies bedeutet. Die meisten Mitwirkenden sind Schauspieler wie etwa der Mörder,

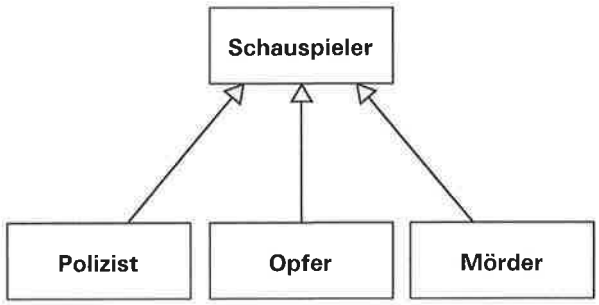
[1] Engl.: to try = versuchen.

[2] Engl.: to catch = (auf)fangen.

[3] Engl. für: Ausnahmen.

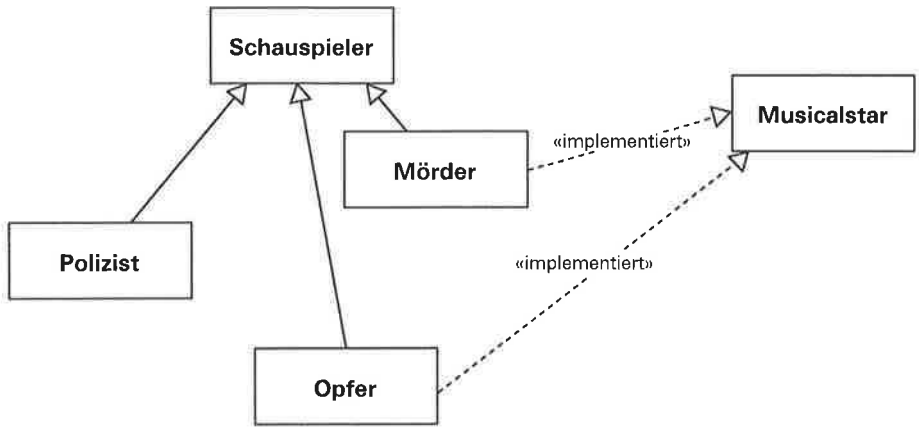
das Opfer, die Zeugen und der Polizist. Wenn Sie nun den Krimi ausbauen möchten, kann es sinnvoll sein, diese Klassen in einer gemeinsamen Oberklasse **Schauspieler** zusammenzufassen. Dies kann mittels Vererbung geschehen und beispielsweise so aussehen:

[3-7] Einfache Vererbung



Weil Musicals im Trend liegen, möchte unser Produzent Arthur das Krimiprojekt zu einem «mörderischen» Musical ausbauen und die Schauspieler singen und tanzen lassen. Entsprechend leiten Sie aus der Klasse **Schauspieler** eine neue Klasse ab (z. B. Musicalstar). Nun eignet sich aber nicht jeder Schauspieler zum Musicalstar, der gleichzeitig auch ein guter Sänger und Tänzer ist. Einige Rollen im Musical setzen aber gerade diese Fähigkeiten voraus. Aus diesem Grund müssen die Klassen **Mörder** und **Opfer** sowohl von der Klasse **Schauspieler** als auch von der Klasse **Musicalstar** erben können. Das ist in Java aber nicht möglich. In solchen Situationen kommen Interfaces ins Spiel. Im Klassendiagramm sieht dies dann wie folgt aus:

[3-8] Interface Musicalstar



Beachten Sie im Zusammenhang mit **Interfaces** folgende Punkte:

- Ein Interface sieht zwar wie eine Mehrfachvererbung aus, ist es aber nicht.
- Interfaces sind immer `abstract` und `public`. Näheres über diese Begriffe erfahren Sie in Kapitel 3.6, S. 44. Eine nichtabstrakte Klasse, die ein Interface implementiert, muss alle Methoden des Interface implementieren. Wenn eine Klasse nicht alle Methoden implementiert, ist sie nicht vollständig und muss abstrakt sein.
- Ein Interface kann andere Interfaces erweitern. Vererbungen sind daher nicht nur bei Klassen, sondern auch bei Interfaces möglich. So kann z. B. das Interface **Musicalstar** vom Interface **Taenzer** abgeleitet sein.
- Ein Interface enthält keinen eigenen Code. Es sagt nur aus, dass eine Verhaltensweise bzw. eine Methode (in Java) vorhanden ist, gibt aber nicht an, wie diese realisiert wird.

Beispiel

Ein MP3-Player hat einen USB-Anschluss, über den er mit Musik geladen werden kann. Wenn Sie den MP3-Player an einen Computer anschliessen, «sieht» dieser keinen MP3-Player, sondern eine USB-Festplatte oder einen USB-Speicherstick. In der realen Welt ist der MP3-Player keine USB-Festplatte. Er kann sich allerdings so verhalten, d. h., er implementiert das Interface für USB-Festplatten.

3.6 Abstrakte Klassen

In Kapitel 3.2, S. 37 haben Sie anhand von Vögeln, Hühnern und Enten das Konzept der Vererbung kennengelernt. Hühner und Enten sind zwar Vögel, doch es ist sinnlos, von der Klasse **Vogel** Objekte zu erzeugen. Um das **Instanziieren**[1] von Vogel-Objekten zu verhindern, verwenden Sie bei der Klassendeklaration in Java das Schlüsselwort `abstract`:

```
public abstract class Vogel {
  ....
```

Nun verhindert der Java-Compiler, dass Sie `new Vogel()` machen können. Selbstverständlich ändert das nichts daran, dass sowohl Hühner als auch Enten Vögel sind.

3.6.1 Abstrakte Methoden

Voraussetzung für **Polymorphismus** ist eine **gemeinsame Methode in der Oberklasse.** Diese Methode muss in allen abgeleiteten Unterklassen überschrieben werden. Nehmen Sie z. B. an, Sie haben in der Klasse **Vogel** die Methode `fliegen()` vorliegen. Damit kann jeder Vogel fliegen (was für Polymorphismus notwendig ist), aber jede Vogelart hat ihre eigene Art zu fliegen. Sie müssen also erzwingen, dass jede von Vogel abgeleitete Klasse ihre eigene fliegen()-Version hat.

In solchen Fällen können Sie die Methode in der Oberklasse als `abstract` kennzeichnen. Die betreffende Methode enthält dann keinen Code. Eine davon abgeleitete Klasse muss diese Methode implementieren und einen Code liefern, sonst ist sie unvollständig. Eine **Klasse mit abstrakten Methoden** ist selber auch abstrakt und muss daher ausdrücklich mit dem Schlüsselwort `abstract` versehen werden.

3.6.2 Abstrakte Klassen und Interfaces

In einem Interface geben Sie an, welche Methoden eine Klasse implementieren muss. Es ist der Klasse überlassen, ob sie dies auch tut oder nicht. Werden nicht alle Methoden implementiert, ist die Klasse automatisch abstrakt. Erst wenn in einer davon abgeleiteten Klasse alle Methoden des Interface implementiert sind, liegt eine konkrete Klasse vor, von der Objekte instanziiert werden können.

[1] Instanziieren bedeutet: eine Instanz erzeugen.

Vermeiden Sie es, direkt auf Instanzvariablen von Objekten zuzugreifen. Erstellen Sie dafür jeweils eine set()- und eine get()-Methode, die den Zugriff vermitteln. Diesen Vorgang nennt man **Datenkapselung.** Benutzer von Objekten müssen sich dann nicht darum kümmern, wie und wo deren Daten intern abgelegt sind. Als Programmierer sind Sie frei, den internen Aufbau von Klassen beliebig zu wählen und zu ändern, ohne dass das Konsequenzen für andere Teile der Applikation hat.

Vererbung ist der Ausdruck einer «Ist-ein-Beziehung» zwischen Klassen. Ein Huhn ist ein Vogel. Die Klasse **Huhn** erbt von der Klasse **Vogel.** Huhn ist die **Spezialisierung** (Unterklasse bzw. abgeleitete Klasse), Vogel die **Generalisierung** (Oberklasse). Die Vererbung in der objektorientierten Programmierung erlaubt es, komplexe Sachverhalte einfach und übersichtlich darzustellen.

Interfaces sind eine elegante Option, um die in Java nicht erlaubte Mehrfachvererbung zu umgehen. Klassen versprechen, bestimmte Methoden zu implementieren. Eine Klasse ist nicht verpflichtet, für jede Methode einen Methodenrumpf mit Code zu liefern. Eine solche unvollständige Klasse ist abstrakt. Von **abstrakten Klassen** können keine Objekte erzeugt werden. Klassen, die von abstrakten Klassen abgeleitet werden, müssen die fehlenden Codeteile (Implementierung der Methodenrümpfe) liefern. In einer Vererbungshierarchie können beliebig viele Oberklassen abstrakt sein. Nur die letzte Unterklasse muss alle Methodenrümpfe implementieren. Eine Klasse kann als abstrakt deklariert werden, auch wenn sie vollständig ist. Dadurch wird verhindert, dass sinnlose Objekte erzeugt werden.

Mit dem Mechanismus der **Ausnahmebehandlung (Exception)** besteht die Möglichkeit, den eigentlichen Code und die Behandlung von Fehlern klar zu trennen. Dadurch wird der Code übersichtlicher.

Repetitionsfragen

14	Warum sollten Sie nicht direkt auf Instanzvariablen eines anderen Objekts zugreifen?
15	Wie heissen die beiden Methoden, mit denen Sie auf die Eigenschaften von Objekten zugreifen können?
16	Wozu dienen Objekte der Klasse **Exception?** Antworten Sie in einem Satz.
17	Erläutern Sie anhand eines Beispiels den Begriff «Polymorphismus».
18	Beschreiben Sie den Begriff «Interface».

4 Notationen und Diagramme

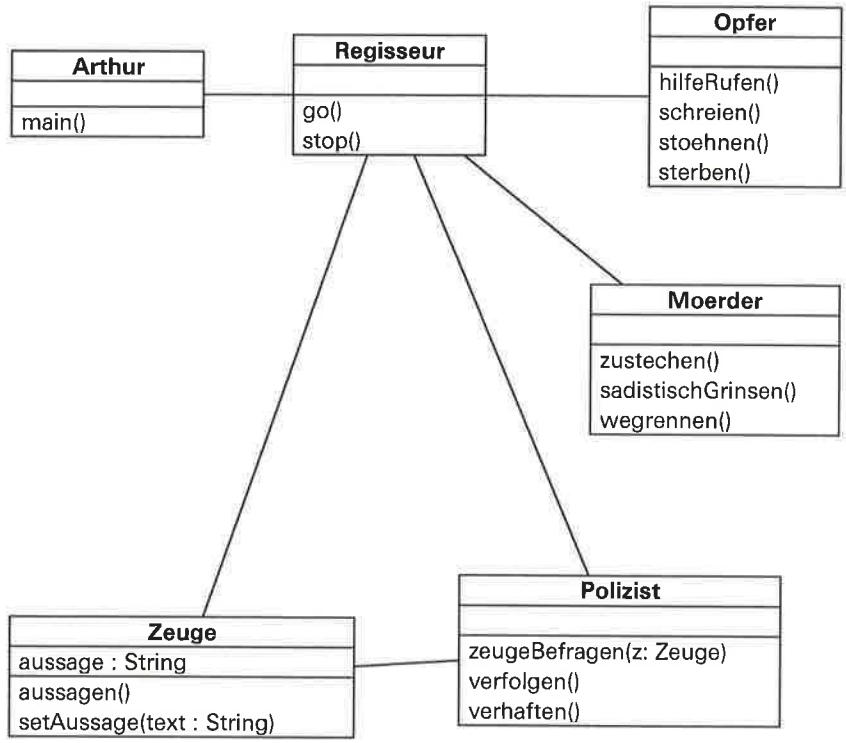
Vermutlich haben Sie bereits grafische Darstellungen von Programmen und Abläufen angetroffen. Von der strukturierten bzw. prozeduralen Programmierung her kennen Sie beispielsweise **Struktogramme** oder **Programmablaufpläne (PAP).**

UML[1] ist eine international standardisierte Modellierungs- und Darstellungssprache, die v. a. im Rahmen des OOD grosse Verbreitung gefunden hat. Bei der Entwicklung von UML wurde darauf geachtet, dass **Darstellungsformen,** die unterschiedliche Sichtweisen und Informationen der objektorientierten Programmierung bzw. Analyse widerspiegeln, als konsistente und sich ergänzende Diagramme abgebildet werden. UML kennt zahlreiche normierte Diagramme zur Darstellung statischer und dynamischer Beziehungen zwischen Klassen und Objekten. In diesem Kapitel werden Sie anhand unseres Krimiprojekts ein paar typische **UML-Diagramme** kennenlernen. Dabei beschränken wir uns auf die für uns wichtigen Klassen-, Objekt- und Sequenzdiagramme.

4.1 Klassendiagramme zeigen die statische Sicht

In unserem Krimiprojekt kommen verschiedene Rollen vor, die durch Klassen dargestellt werden. Die statischen Zusammenhänge zwischen den einzelnen Rollen können in einem **Klassendiagramm** aufgezeigt werden:

[4-1] Klassendiagramm (Beispiel)



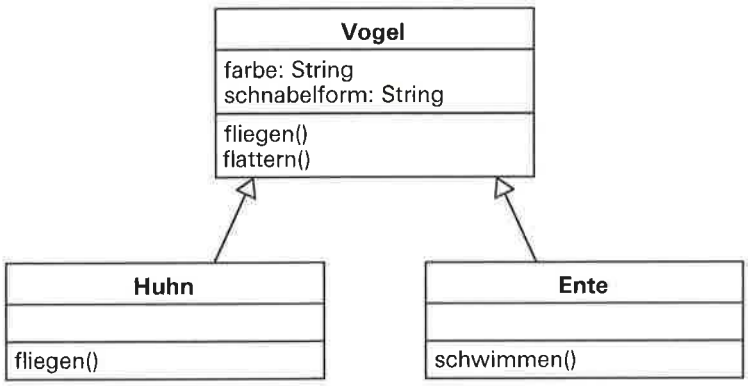
Im obigen Klassendiagramm wird jede Klasse durch ein dreigeteiltes Rechteck dargestellt. Im oberen Teil steht der **Klassenname,** im mittleren Teil werden die **Instanzvariablen** ausgewiesen und im unteren Teil sind die **Methoden** zu finden.

[1] Abk. für: Unified Modeling Language.

Die Klasse **Zeuge** hat beispielsweise die Instanzvariable `aussage` vom Typ `String` und die beiden Methoden `aussagen()` und `setAussage()`. Die Linien zwischen Klassen deuten auf Beziehungen zwischen diesen hin. Genauer gesagt besteht eine Beziehung zwischen den Objekten dieser Klassen, d. h. Objekte der Klasse **Regisseur** «kennen» Objekte der Klassen **Opfer, Moerder, Polizist und Zeuge.** Polizisten-Objekte kennen auch Zeugen-Objekte, allerdings gibt es keine Beziehung zwischen Mörder und Zeugen. Mit anderen Worten: Ein Klassendiagramm zeigt auf, welche Klassen es gibt und zwischen welchen Objekten dieser Klassen Beziehungen bestehen können. Es sagt aber nichts über die **Beziehungsarten** aus oder darüber, ob sich immer alle Objekte von Klassen mit einer Beziehung kennen. So müssen z. B. nicht immer alle Polizisten alle Zeugen kennen, obwohl zwischen diesen beiden Klassen eine Beziehung vorliegt.

Wie Sie bereits in Kapitel 3.2, S. 37 gesehen haben, werden im Klassendiagramm auch **Vererbungen** dargestellt. Hier sehen Sie noch einmal die entsprechenden Beziehungen zwischen den Klassen **Vogel, Huhn** und **Ente.**

[4-2] Klassendiagramm mit Vererbung (Beispiel)

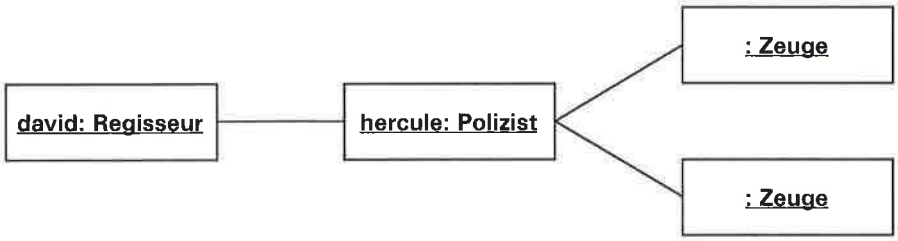


Die **Vererbungsbeziehung** wird durch eine Linie mit offenem Dreieckspfeil symbolisiert, wobei der Pfeil immer auf die Oberklasse zeigt. Im obigen Klassendiagramm sehen Sie, dass sowohl die Klasse **Huhn** als auch die Klasse **Ente** von der Klasse **Vogel** erben.

4.2 Objektdiagramme zeigen das Beziehungsgeflecht

Im Gegensatz zum Klassendiagramm werden im **Objektdiagramm** die Objekte einzeln gezeichnet, die miteinander interagieren. Auch dazu ein Beispiel aus unserem Krimiprojekt:

[4-3] Objektdiagramm (Beispiel)



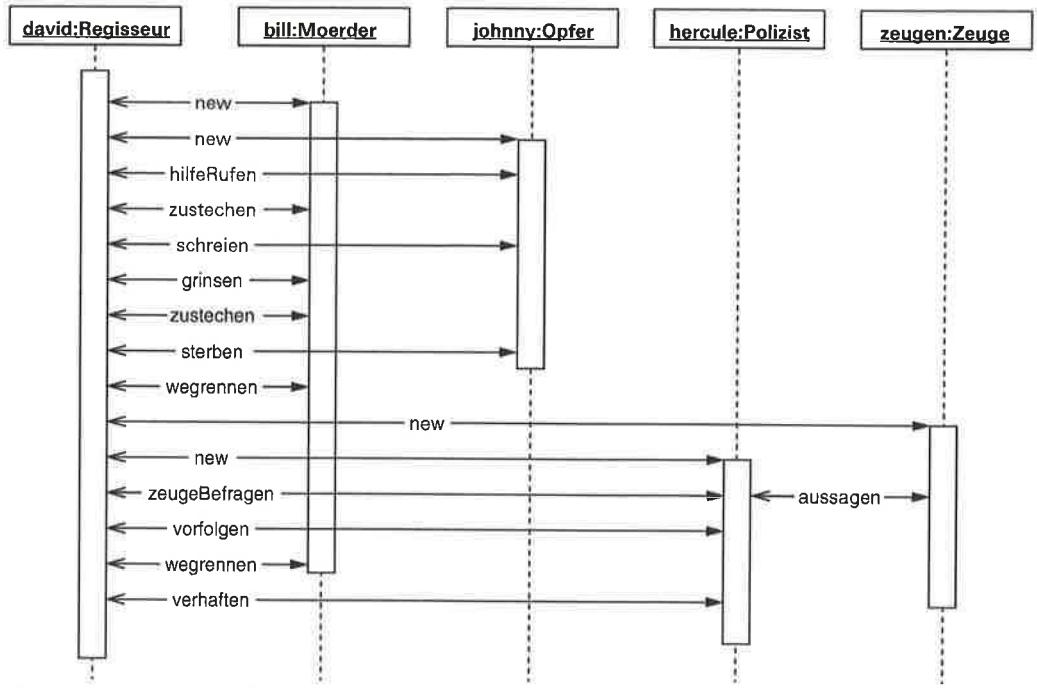
Hier sehen Sie, dass das Objekt **David** vom Typ `Regisseur` mit Objekt **Hercule** vom Typ `Polizist` kommuniziert. Polizist Hercule befragt die beiden Zeugen-Objekte. Zur besseren Unterscheidung zwischen Klasse und Objekt muss der **Objektname** immer unterstrichen dargestellt werden. Der Objektname beinhaltet einen **Datentyp,** der der Klasse des Objekts

entspricht und durch einen Doppelpunkt getrennt dem Objektnamen angehängt wird. Da in unserem Krimiprojekt die beiden Zeugen keinen Namen haben, werden sie im obigen Beispieldiagramm weggelassen. Der Datentyp bzw. die Klasse muss jedoch auch in solchen Fällen aufgeführt werden. Entsprechend beginnt der Objektname hier jeweils mit einem Doppelpunkt.

4.3 Sequenzdiagramme zeigen den Interaktionsablauf

In einem **Sequenzdiagramm** wird der zeitliche Ablauf der Aktionen von Objekten grafisch dargestellt. Dabei werden die Objekte gleich wie im Objektdiagramm beschriftet, d. h. unterstrichen und entsprechend dem Aufbau «Bezeichnung-Doppelpunkt-Datentyp». Das Sequenzdiagramm für unser Krimiprojekt kann beispielsweise wie folgt aussehen:

[4-4] Sequenzdiagramm (Beispiel)



Im Objektdiagramm wird für jedes Objekt eine eigene «Lebenslinie» gezeichnet und die zeitliche Abfolge von oben nach unten gelesen. Hier verlangt beispielsweise Regisseur David vom Polizisten Hercule, dass er Zeugen befragt. Hercule reagiert darauf, indem er vom Zeugen eine Aussage verlangt. Danach verlangt David, dass Hercule die Verfolgung aufnimmt.

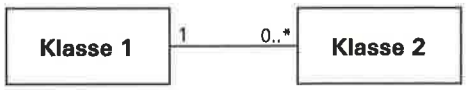
Klassendiagramme und **Objektdiagramme** sind grafische Darstellungen von Klassen bzw. Objekten der OOP. Sie haben Ähnlichkeit mit einem Entity-Relationship-Diagramm (ERD) und zeigen auf, welche Klassen bzw. Objekte sich kennen. Klassendiagramme und Objektdiagramme stellen sozusagen die statische Sicht der Beziehungen zwischen Klassen (inkl. Vererbungen) bzw. Objekten dar.

Demgegenüber zeigen **Sequenzdiagramme** eine dynamische Sicht und veranschaulichen den zeitlichen Ablauf eines Programms. Sie zeigen auf, in welcher Abfolge welches Objekt mit welchem anderen Objekt interagiert, wobei die zeitliche Abfolge von oben nach unten gelesen wird und jedes Objekt eine eigene Lebenslinie hat.

Repetitionsfragen

19 Was stellt ein Klassendiagramm dar? Antworten Sie in einem Satz.

20 Erklären Sie das folgende Klassendiagramm:



21 Was ist ein Sequenzdiagramm? Antworten Sie in einem Satz.

22 Wie wird in einem Klassendiagramm eine Vererbung dargestellt?

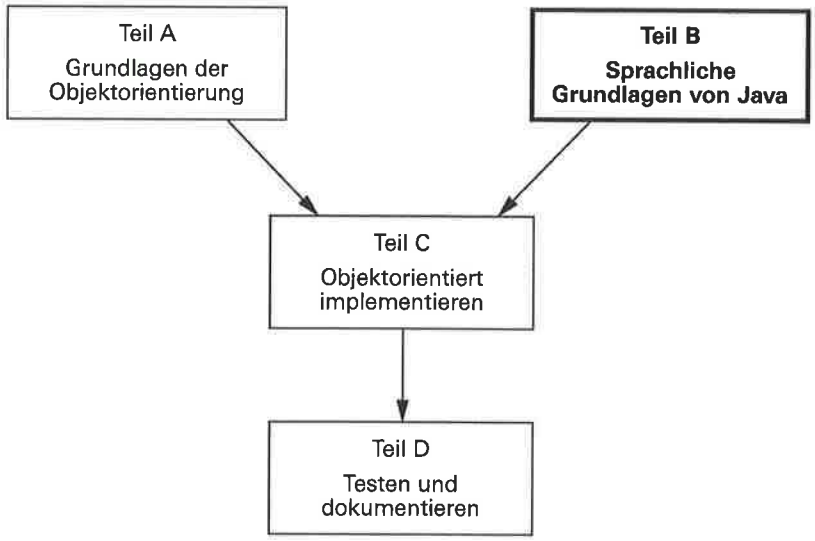
Teil B Sprachliche Grundlagen von Java

Einleitung, Lernziele und Schlüsselbegriffe

Einleitung

Java ist eine Sprache, die sich von der **Syntax** her stark an C orientiert. Wenn Sie sich die grundlegenden Kenntnisse dafür aneignen möchten, empfehlen wir Ihnen die Lehrmittel zu den Modulen 103 und 118.[1] In Teil B dieses Lehrmittels werden wichtige **prozedurale Elemente der Programmiersprache Java** behandelt. Objektorientierte Elemente lernen Sie in Teil C kennen. Folgende Grafik zeigt, wo Sie sich innerhalb des Lehrmittels befinden:

Teil B im Gesamtzusammenhang



Lernziele und Lernschritte

	Lernziele	Lernschritte
☐	Sie können erläutern, wie Klassenmodelle mit einer objektorientierten Programmiersprache umgesetzt werden können.	• Aufbau eines Java-Programms • Datentypen • Collections • Kontrollstrukturen • Funktionen und Methoden

Schlüsselbegriffe

Array, autoboxing, boolean, by reference, by value, byte, char, double, final, for, if-else, import, int, Klasse, long, main()-Methode, Methode, Paket, short, Signatur, String, switch-case, Typumwandlung, Unicode, Vergleichsoperator, while, Wrapper

[1] Vergleichen Sie dazu das Literaturverzeichnis auf S. 11.

5 Aufbau und Datentypen

In diesem Kapitel können Sie nachvollziehen, wie ein einfaches «Hallo-Welt-Programm» in Java erstellt wird. Dabei lernen Sie, wie diese Programmiersprache aufgebaut ist, welche Datentypen sie verwendet und wie die Programmierumgebung von Java aussieht.

5.1 Hallo Welt und die virtuelle Java-Maschine

Zum Einstieg in die Programmierung mit Java wollen Sie der Welt «Hallo!» sagen. Dazu muss auf dem Rechner ein aktuelles **JDK**[1] installiert werden. Dieses Tool ist die minimale Entwicklungsumgebung, die auf einem Rechner installiert sein muss, um Java-Programme zu erstellen. Um besser verfolgen zu können, wie Java funktioniert, arbeiten wir mit der **Java-Umgebung auf der Kommandozeile** des Betriebssystems.

5.1.1 Über die Kommandozeile einsteigen

Programme werden normalerweise mithilfe eines **Texteditors** erstellt und danach durch den **Compiler** in eine ausführbare Version umgewandelt. Damit Sie sehen, wie das in Java funktioniert, arbeiten Sie zunächst nur mit der **Kommandozeile.** Erstellen Sie zuerst einen Arbeitsordner, in den Sie Ihre Java-Programme ablegen möchten. Öffnen Sie danach eine Kommandozeile (unter Windows eine DOS-Box, unter Unix und Linux ein Konsolenfenster) und bestimmen Sie diesen Ordner als aktuelles **Arbeitsverzeichnis.**

[5-1] Das Java-Logo

5.1.2 Programmcode erfassen

Erstellen Sie mit dem **Texteditor**[2] eine Datei **HalloWelt.java** mit folgendem Inhalt:

```
package ch.modul226;

public class HalloWelt {
    public static void main(String[] args) {
       System.out.println("Hallo Welt, Java ist alles andere als kalter Kaffee.");
    }
}
```

Eine Java-Anwendung besteht aus mindestens einer **Klasse.** Klassen können als kleine (Teil-)Programme aufgefasst werden und oft bilden mehrere Klassen eine Anwendung.

[1] Abk. für: Java Development Kit.

[2] Es kommt nicht darauf an, welchen Texteditor Sie wählen. Dieser muss aber eine reine Textdatei erzeugen.

Die Klasse **HalloWelt** im obigen Codeabschnitt hat eine main()-Methode. Es handelt sich also um den **Einstiegspunkt** des Programms. Mittels Aufruf der **Methode** `System.out.println()` wird auf dem Bildschirm die Zeile `Hallo Welt, Java ist alles andere als kalter Kaffee.` ausgegeben.

5.1.3 Byte-Code erzeugen

Um den Programmcode (auch: Sourcecode, Quellcode oder Quelltext) in ein ausführbares Programm zu bringen, müssen Sie den **Java-Compiler** aufrufen. Dieser verwandelt den erfassten Programmcode in einen sogenannten **Byte-Code,** der vom Computer «gelesen und verstanden» werden kann. Sie können den Java-Compiler wie folgt aufrufen:[1]

```
javac -d . HalloWelt.java
```

Die Option `-d .` gibt an, wo die Klassendateien abgelegt werden sollen. Dabei markiert der Punkt das aktuelle **Arbeitsverzeichnis.** Beachten Sie die genaue Schreibweise mit zwei Leerschlägen und einem einzelnen Punkt!

Der **Kompiliervorgang** sollte kommentarlos abgeschlossen werden. Werden Fehler gemeldet, liegt das wahrscheinlich daran, dass der Programmcode nicht richtig erfasst wurde. Vergleichen Sie Ihren Sourcecode genau mit dem obigen Programmcode. Sind alle Klammern vorhanden? Haben Sie kein **Semikolon**[2] vergessen? Stimmt die Gross- und Kleinschreibung?

Mit dem Befehl `dir` unter Windows oder `ls` unter Linux bzw. Unix können Sie nun sehen, was der Java-Compiler gemacht hat: Im Arbeitsverzeichnis finden Sie den Ordner **ch,** darin den Ordner **modul226** und darin die Datei **HalloWelt.class.** Ein Dateibrowser zeigt die **Ordnerstruktur** wie folgt an:

[5-2] Ordnerstruktur (Ansicht Dateisystem)

Was hat der Java-Compiler genau gemacht? Im Sourcefile steht zuoberst die Zeile `package ch.modul226;` Daraus wurde die obige Ordnerstruktur erstellt und auf der untersten Ebene steht die Datei **HalloWelt.class** mit dem ausführbaren Java-Programm.

5.1.4 Programm ausführen

Wenn Sie z. B. in C/C++ ein Programm schreiben und kompilieren, wird eine **ausführbare Datei** erzeugt, die direkt im aktuellen Betriebssystem gestartet werden kann. Diese Datei enthält Maschinenbefehle für den Prozessor des jeweiligen Rechners. Eine unter Windows erstellte EXE-Datei können Sie deshalb auf einem Mac-OS-X-Rechner nicht ausführen und eine Mac-OS-X-Programmdatei lässt sich nicht unter Linux starten. Dies ist bei Java anders: Der Java-Compiler erzeugt mit dem Byte-Code quasi die Maschinensprache für

[1] Wenn Sie ein JDK installiert haben, dieser Befehl aber nicht funktioniert, haben Sie vermutlich die Pfadvariable nicht korrekt gesetzt. Vergleichen Sie dazu bitte den Anhang auf S. 161.

[2] Fremdwort für: Strichpunkt.

einen Java-Prozessor, den es als Hardware nicht gibt.[1] Das Java-Programm läuft stattdessen in einer sogenannten **virtuellen Java-Maschine,** die den Byte-Code erst während der Laufzeit des Programms in Befehle für den Prozessor des Rechners verwandelt.

Um ein Java-Programm zu starten, müssen Sie also zuerst die virtuelle Maschine aufrufen und dieser danach das Programm wie folgt zu Ausführung übergeben:

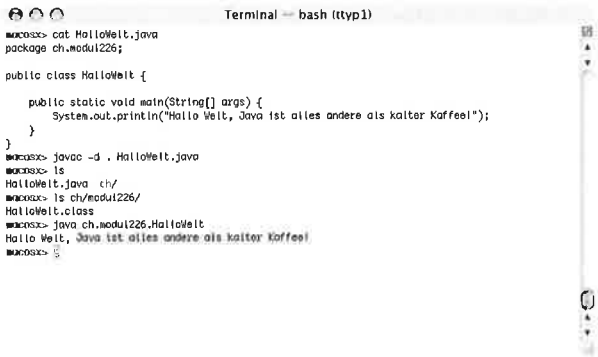
```
java ch.modul226.HalloWelt
```

Hinweis

▷ Beachten Sie, die virtuelle Maschine interessiert sich nicht für Dateien. Sie benötigt den Paketpfad und den Namen des Programms. Die virtuelle Maschine trennt den Pfad bei den Punkten auf und sucht sich die entsprechende Ordnerstruktur. Beachten Sie, dass der Name des Programms kein **.class** hat.

Nachfolgend sehen Sie ein Beispiel, wie der **Kompiliervorgang eines Java-Programms** in der Konsole eines Unix-Systems aussieht:

[5-3] Kompiliervorgang in der Unix-Konsole



5.2 Aufbau eines Java-Programms

Anhand des Beispielprogramms HalloWelt werden Sie im Folgenden die **Strukturen eines Java-Programms** näher kennenlernen.

5.2.1 Struktur einer Klassendeklaration

Sehen Sie sich den Quelltext des Programms HalloWelt in Kapitel 5.1.2, S. 53 noch einmal genau an. In Java müssen alle ausführbaren Codezeilen innerhalb einer **Klassendefinition** stehen:

```
public class HalloWelt {
    // hier steht der Inhalt der Klassendefinition
}
```

Das Schlüsselwort `class` sagt, dass es sich bei **HalloWelt** um eine Klasse handelt. Alle Klassen müssen vorerst als `public` deklariert werden. Beachten Sie, dass eine «öffentliche» Klasse immer in einer Datei stehen muss, die den gleichen Namen wie die Klasse

[1] Einschränkung: Es gibt zwar keine Java-PCs, aber spezielle Prozessoren (z. B. für Mobiltelefone), die Java-Byte-Code verarbeiten können.

trägt. Bei Java muss die Dateierweiterung zwingend **.java** lauten. Die Klasse **HalloWelt** muss demnach in der Datei **HalloWelt.java** stehen.

Vor der **Klassendeklaration** steht das Schlüsselwort `package` und dahinter, in welchem Paket der Programmcode «verstaut» werden soll. Innerhalb des Klassenkörpers können Sie beliebige Variablen und Funktionen deklarieren, die gemeinsam die Eigenschaften und das Verhalten der Objekte dieser Klasse ausmachen.

5.2.2 Gesamtstruktur eines Java-Programms

Wie Sie im Rahmen des Krimiprojekts gesehen haben, besteht ein Java-Programm aus einer **Abfolge von mehreren Klassen und Objekten,** die gemeinsam die Funktionalität des Programms ausmachen. Während der Laufzeit des Programms werden ständig neue Objekte erzeugt, die untereinander kommunizieren bzw. Nachrichten austauschen. Nicht mehr benötigte Objekte werden von der Laufzeitumgebung automatisch wieder entsorgt.

5.2.3 Struktur der main()-Methode

Jedes Programm muss einen **Einstiegspunkt** haben. Bei Java wird dieser – ähnlich wie bei C/C++ – durch die **main()-Methode** gekennzeichnet. In einem typischen Java-Programm beinhaltet die main()-Methode nur wenige Zeilen Programmcode. Vergleichen Sie dazu den folgenden Ausschnitt aus dem Programm **HalloWelt:**

```
public static void main(String[] args) {
  System.out.println("Hallo Welt, Java ist alles andere als kalter Kaffee.");
}
```

Ein Java-Programm kann nicht nur einen, sondern mehrere (verschiedene) Einstiegspunkte haben. Entsprechend muss der Programmcode in mehreren Klassen eine main()-Methode aufweisen. Beim Starten des Programms müssen Sie dann jeweils die Klasse angeben, in der sich die main()-Methode befindet, die als Einstieg verwendet werden soll.

5.3 Datentypen

Anders als PHP ist Java eine streng typisierte Programmiersprache. Das heisst, dass alles immer einen eindeutig definierten **Datentyp** haben muss. Java kennt mehrere **primitive Datentypen**[1], die im Folgenden kurz beschrieben werden.

5.3.1 Ganzzahlen

Folgende **Datentypen für ganze Zahlen** gibt es in Java:

Datentyp	Grösse	Beschreibung	Wertebereich
`byte`	8 bit.	Damit lassen sich die Zahlen von –128 bis +127 darstellen.	-2^7 bis 2^7-1
`short`	16 bit.	Damit lassen sich die Zahlen von –32 768 bis +32 767 darstellen.	-2^{15} bis $2^{15}-1$
`int`	32 bit.	Damit lassen sich die Zahlen von –2 147 483 648 bis +2 147 483 647 darstellen (beinhaltet insgesamt mehr als vier Milliarden Werte).	-2^{31} bis $2^{31}-1$
`long`	64 bit.	Damit lassen sich Zahlen von –9 223 372 036 854 775 808 bis +9 223 372 036 854 775 807 darstellen.	-2^{63} bis $2^{63}-1$

[1] Diese heissen primitiv, weil sie keine weiteren Eigenschaften haben. Klassen hingegen sind sogenannte komplexe Datentypen. Klassen definieren Eigenschaften und Verhaltensweisen.

Beispiel

```
int zahl;
zahl = 17;
long nummer = 2348739271;
```

5.3.2 Kommazahlen

Für **Kommazahlen** stehen Ihnen zwei Datentypen zur Verfügung, `float` und `double`. Beide reichen für die meisten Anwendungen problemlos aus. Der Typ `float` wird intern mit 32 Bit abgespeichert, für `double` verwendet Java sogar 64 Bit Speicherplatz.

Beispiel

```
float temp;
double pi = 3.1415926;
```

5.3.3 Logische Werte

Logische Werte werden beispielsweise für Schleifenkonstruktionen und Verzweigungen verwendet.[1] Für die logischen Werte «wahr» und «falsch» gibt es den Datentyp `boolean`. Dieser kann nur die Werte `true` oder `false` annehmen. `true` und `false` sind Schlüsselwörter in Java.

Beispiel

```
boolean fertig;
fertig = true;
```

Hinweis

▷ In C/C++ sind logische Werte numerisch. Daher können dort die Zahlen 1 und 0 anstatt `true` und `false` verwendet werden. Dies ist in Java nicht möglich. Boolean-Werte sind hier keine numerischen Werte.

5.3.4 Unicode-Zeichen

Java arbeitet konsequent mit **Unicode-Zeichen.** Dieser genormte Zeichensatz umfasst mehr als 65 000 Zeichen. Davon sind viele noch ungenutzt. Trotzdem reicht es, um alle Schriftzeichen dieser Welt aufzunehmen. Unicode-Zeichen müssen jeweils zwischen einfachen Hochkommas geschrieben werden. Unicode-Werte umfassen den Bereich von `'\u0000'` bis `'\uffff'` und benötigen 16 Bit.

Beispiel

```
char zeichenA = 'A';
char uUmlaut = '\u00e4';  // ä
```

5.3.5 Zeichenketten

Für **Zeichenketten** gibt es in Java keinen primitiven Datentyp. Java hat dafür die Klasse **java.lang.String** vorgesehen. Java-Strings verwenden intern immer Unicode, können aber auch in andere Codierungen umgewandelt werden. Zeichenketten können entweder direkt eingegeben (wie bei C/C++) oder zugewiesen werden.

[1] Vergleichen Sie dazu das Kapitel 6, S. 63.

Beispiel

- `String wort = "Hallo";`
- `wort = "Hallo";  // Kurzschreibweise von wort = new String("Hallo")`

Strings können zudem verknüpft werden. Vergleichen Sie dazu folgendes Beispiel:

```
String satz = "Guten Tag. ";
satz = satz + "Wie geht es Ihnen?";  // ergibt "Guten Tag. Wie geht es Ihnen?"
```

5.3.6 Arrays

In C/C++ ist es die Aufgabe des Programmierers, die **Länge eines Arrays** zu verwalten und im Auge zu behalten. In Java sind **Arrays** dagegen Objekte.

Beispiel

- `int[] vieleZahlen = {1, 2, 4, 18, 99, -34, 2395, -77, 0, 12356};`
- `int[] zahlenReihe;`
  `zahlenReihe = new int[10];`

Beachten Sie, dass eckige Klammern beim Datentyp (auf der linken Seite des Gleichheitszeichens) notwendig sind, um Java zu sagen, dass es sich um ein **Array** handelt. Der Ausdruck auf der rechten Seite des Gleichheitszeichens erzeugt jeweils ein **neues Array-Objekt.** In beiden Fällen wird ein Array erzeugt, das 10 int-Werte aufnehmen kann. Im ersten Fall wird das Array mit 10 int-Werten initialisiert. Im zweiten Fall wird zuerst eine Variable erzeugt, die ein Array referenzieren kann. Durch die darauf folgende Anweisung wird ein Array-Objekt erstellt und die Referenz darauf der Variablen `zahlenReihe` zugewiesen. So können Sie Arrays erstellen, ohne bereits im Voraus deren Länge kennen zu müssen.

Arrays können ihre Länge nicht verändern, kennen diese jedoch. Dies macht sich die Java-Laufzeitumgebung zunutze, um zu prüfen, ob der **Array-Index** gültig ist. Dadurch kann z. B. eine for-Schleife programmiert werden, ohne dass über die Grenzen des Arrays hinaus in den Speicher geschrieben wird. Werden die Grenzen eines Arrays überschritten, erzeugt das Programm einen **Laufzeitfehler.** Mithilfe folgender Anweisung greifen Sie direkt auf die **Array-Länge** zu, um die richtige Anzahl der Schleifendurchgänge zu ermitteln:

```
for (short index = 0; index < zahlenReihe.length; index++) {
    zahlenReihe[index] = index;
}
```

5.3.7 Konstanten

Konstanten sind Variablen, deren Wert nicht mehr geändert werden kann, wenn er einmal definiert worden ist. Java verlangt für solche Konstanten das Schlüsselwort `final`:

```
final String GRUSSFORMEL = "Mit freundlichen Grüssen";
```

Im obigen Codeabschnitt wurde ein String mit einem fest zugeordneten Wert deklariert. Aufgrund des Schlüsselworts `final` kann dieser Wert im Programmablauf nicht mehr geändert werden.

Hinweis

▷ Beachten Sie, dass Variablen bzw. Konstanten in Java kleingeschrieben werden. In der C-Programmierung hat es sich eingebürgert, Konstanten vollständig in Grossbuchstaben zu schreiben. Viele Java-Programmierer haben dies so übernommen.

5.3.8 Typumwandlungen

In Java hat jede Variable und jeder Ausdruck einen eindeutig definierten **Datentyp.** Wenn Variablen mittels Gleichheitszeichen einander zugewiesen werden, werden die Datentypen automatisch umgewandelt, falls dies notwendig und ohne Datenverlust machbar ist. So ist es z. B. möglich, anstelle von long-Werten int-Werte oder noch kürzere Ganzzahlwerte zu verwenden. Allerdings können Sie den Wert einer long-Variablen nur einer int-Variablen zuweisen, wenn der entsprechende Wert in einer int-Variablen auch effektiv Platz hat.

Hinweis

▷ 10.0 ist nicht dasselbe wie die Zahl 10. Das eine ist ein Wert von Typ `double` oder `float`, das andere ist ein ganzzahliger Wert. Ganzzahlen werden automatisch zu `float` oder `double` erweitert, double-Werte können jedoch nicht einer int-Variablen zugewiesen werden.

Im folgenden Codeabschnitt sehen Sie, dass eine **Typenumwandlung** auch erzwungen werden kann, indem der gewünschte Datentyp in runden Klammern vor den Ausdruck geschrieben wird:

```
byte b = 397;      // nicht erlaubt, Wert ist zu gross, passt nicht in byte
long c = b;        // b ist ein byte-Wert, hat problemlos in long Platz
double d = c;      // c ist ein long-Wert und wird automatisch erweitert

double e = 3.75;
int f = e;         // nicht möglich, da Kommastellen verloren gehen würden
int g = (int)e;    // Ok, int wird vom Programmierer ausdrücklich erzwungen
```

Hinweis

▷ Im letzten Beispiel werden die Kommastellen abgeschnitten, d. h., Java rundet nicht.

5.3.9 Wrapper-Klassen

Primitive Datentypen besitzen keine Eigenschaften und stellen daher keine Objekte dar. Damit Sie in der OOP auf solche Datentypen verzichten können, gibt es zu jedem primitiven Datentyp eine **Wrapper**[1]**-Klasse.** Der primitive Datentyp wird hier sozusagen von einer Klasse «umhüllt».

Primitiver Datentyp	Wrapper-Klasse («Hüllklasse»)
`byte`	Byte
`short`	Short
`int`	Integer
`long`	Long
`boolean`	Boolean
`char`	Character
`float`	Float
`double`	Double
`void`	Void

Sie sehen: Die Wrapper-Klassen heissen fast gleich wie die primitiven Datentypen, werden jedoch nach Java-Konvention mit einem **grossen Anfangsbuchstaben** geschrieben.

[1] Abgeleitet von «to wrap»: engl. für: «umhüllen».

Jede Wrapper-Klasse hat einen **Konstruktor,** der einen primitiven Datentyp als Parameter verlangt. Ausserdem hat jede Wrapper-Klasse **Methoden,** die den Wert als primitiven Datentyp zurückgeben. Hier ein Beispiel dazu mit der Klasse **Integer:**

```
int zahl = 17;                                          // primitiver Datentyp

Integer zahlObj = new Integer(zahl);                    // ein Integer-Objekt
                                                        // wird erzeugt

int primitiverIntWert = zahlObj.intValue();             // der primitive int-Wert
                                                        // wird verlangt

double primitiverDoubleWert = zahlObj.doubleValue()     // der Wert als primitiver
                                                        // double-Datentyp wird
                                                        // zurückgegeben
```

Weil Klassen und primitive Datentypen nicht dasselbe sind, müssen Sie bei **Zuweisungen** aufpassen. In Java bis und mit Version 1.4 gilt:

```
Integer intObj = 17;
```

Diese Zuweisung ist nicht erlaubt, da 17 kein Objekt ist, die Variable `intObj` aber ein Objekt vom Typ `Integer` referenzieren muss. Auch der Weg vom Objekt zum primitiven Datentyp ist nicht direkt erlaubt:

```
Double einDoubleObj = new Double(3.14159);
double doubleWert = einDoubleObj;          // nicht erlaubt in Java bis V1.4 .
```

Rechnen mit Objekten ist in Java aufwendig, da die Rechenwerte jeweils unter Aufruf der entsprechenden Methoden in primitive Werte umgesetzt werden müssen. Aus diesem Grund wurden ab Version 5 das sogenannte **Autoboxing** und **Auto-Unboxing** eingeführt. Dadurch können Wrapper-Objekte und primitive Datentypen einander ohne Typumwandlung zugewiesen werden und Java sorgt «hinter den Kulissen» für die korrekte Umwandlung eines Objekts zum primitiven Datentyp und umgekehrt. Was in den obigen Beispielen nicht funktioniert, ist ab Java 5 also erlaubt. Dazu folgendes Beispiel:

```
Double einDoubleObjekt = 17.5;  // ein Double-Objekt wird erstellt
                                // mit dem primitiven Wert 17.5
int ergebnis = intObj + 5;      // das Integer-Objekt wird von Java automatisch
                                // "ausgepackt", um die Berechnung
                                // ausführen zu können
```

5.3.10 Zeiger

In C/C++ sind Zeiger zwar eine mächtige Konstruktion, aber auch eine ständige Quelle von Programmierfehlern, die meist nur schwer zu finden sind. In Java gibt es **keine Zeiger.** Weil in Java mit Ausnahme der primitiven Datentypen alles ein Objekt ist, werden auch keine Zeiger benötigt. Wenn Sie eine Variable mit einer Klasse als Datentyp deklarieren, wird eine Variable für ein Objekt erstellt. Dabei wird nicht das Objekt selbst erstellt und in der Variablen abgelegt, sondern lediglich eine Referenz auf ein Objekt bereitgestellt. Die Variable ist also nichts anderes als ein Zeiger auf das Objekt. Entsprechend kann bei Java jede Variable eines nichtprimitiven Datentyps als **interner Zeiger** aufgefasst werden.

Hinweis

▷ Wenn Sie einen Zeiger auf einen primitiven Datentyp benötigen, müssen Sie auf die entsprechende Wrapper-Klasse ausweichen und mit Objekten arbeiten. Beachten Sie dazu auch das Kapitel 7.2, S. 70, in dem die Parameterübergabe bei Funktionen besprochen wird.

5.4 Collections

Collections sind Java-Klassen und Java-Interfaces, die gemeinsame Eigenschaften haben und sich zur Aufnahme mehrerer Objekte des gleichen Typs eignen. Dazu gehören etwa Arrays oder verschiedene Listen (sortiert, unsortiert, eindeutig, mehrdeutig etc.), die eine variable Anzahl von Objekten zulassen. Im folgenden Codeabschnitt wird mit der Klasse **ArrayList** eine Collection-Klasse vorgestellt, die eine Art **dynamisches Array** darstellt und im Paket **java.util** zu finden ist:

```
// ein ArrayList-Objekt wird erzeugt
ArrayList<Integer> zahlen = new ArrayList<Integer>();
// das erste Element erhält den Wert 17
zahlen.add(new Integer(17));
// weitere Anweisungen gleicher Art
// das 4. Element wird abgerufen (Zählung beginnt bei 0.)
Integer wert = zahlen.get(3);
// Anzahl Elemente abfragen
System.out.println("Es sind " + zahlen.count() + " Elemente im Array");
// Der Wert des dritten Elements wird neu gesetzt
Integer dreissig = new Integer(30);
zahlen.set(2, dreissig);
```

Wie Sie erkennen können, funktioniert die ArrayList ähnlich wie ein normales Array, kann ihre Grösse aber dynamisch anpassen. Interessant ist auch, dass bei Collections eine **vereinfachte for-Schleife** angewendet werden kann. Hier sehen Sie ein Beispiel dazu:

```
ArrayList<String> liste = ...;
// ... liste wird mit String-Werten gefüllt
for (String wort : liste) {
    System.out.println(wort);
}
```

Der **Java-Compiler** übersetzt den Quelltext in einen Byte-Code, der von der **virtuellen Java-Maschine** interpretiert wird.

Java-Programme bestehen meist aus vielen Klassen. Jede Klasse kann als **Einstieg in das Programm** dienen, wenn in der Klasse eine **main()-Methode** erstellt wird.

Java unterscheidet ähnliche Datentypen wie prozedurale Programmiersprachen:

- Für **Ganzzahlen** werden die Datentypen `byte`, `short`, `int` und `long` verwendet.
- Für **Zahlen mit Kommastellen** werden die Datentypen `float` und `double` verwendet.
- **Logische Werte** haben den Datentyp `boolean` mit den möglichen Werten `true` und `false`.
- Für **Zeichenketten** kann die Klasse **String** verwendet werden.

Arrays sind in Java echte Objekte, die ihre Länge kennen. Wie C nimmt Java automatische **Typumwandlungen** vor, sofern dabei keine Informationen verloren gehen. Eine Typumwandlung kann erzwungen werden, indem der gewünschte Typ in runden Klammern vor den Ausdruck gesetzt wird. Zusätzlich zu **primitiven Datentypen** stellt Java sogenannte **Wrapper-Klassen** zur Verfügung, die eine objektorientierte Darstellung der Datentypen erlauben. Um noch mehr Flexibilität zu ermöglichen, bietet das Paket **java.util** sogenannte **Collections** an, die mehrere Werte oder Objekte zusammenhalten können.

Repetitionsfragen

23 Nennen Sie mindestens vier ganzzahlige primitive Datentypen.

24 Welchen Datentyp hat die Variable `zahl` in der folgenden Zeile?

```
zahl = 3.14159;
```

25 Worin unterscheiden sich logische Werte in C und in Java?

26 Was ist der Unterschied zwischen `'a'` und `"a"` in Java?

27 Kann man Strings in Java addieren? Wenn ja, was passiert in diesem Fall? Wenn nein, warum nicht?

28 Funktioniert folgender Code? Begründen Sie Ihre Antwort und machen Sie ggf. einen Korrekturvorschlag.

```
int[] zahlenreihe;
zahlenreihe[1] = 17;
```

29 Charakterisieren Sie Wrapper-Klassen in einem Satz.

30 Beschreiben Sie in einem Satz, wie «Zeiger» in Java implementiert werden können.

31 Wie wird der Java-Quellcode «zum Laufen» gebracht? Antworten Sie möglichst knapp.

32 Beschreiben Sie Funktion und Nutzen der Java Virtual Machine in ein bis zwei Sätzen.

33 Wie muss die Deklaration der main()-Methode aussehen?

34 Nennen Sie zwei zentrale Unterschiede zwischen Arrays in Java gegenüber Arrays in C.

35 Nachfolgend sehen Sie die ersten Zeilen der Datei **Rechner.java.** Was ist hier falsch?

```
package ch.modul226;
private int nummer;
public class Rechner {
...
```

6 Kontrollstrukturen

Kontrollstrukturen werden zur **Steuerung des Programmablaufs** benötigt. Nachfolgend lernen Sie die **Kontrollstrukturen von Java** näher kennen. Da Java auf der Syntax von C basiert, sollte Ihnen dabei vieles vertraut vorkommen.

6.1 Vergleichsoperatoren und logische Verknüpfungen

Mithilfe von **Vergleichsoperatoren** können Sie Ausdrücke bilden, die eindeutig wahr (`true`) oder falsch (`false`) sind und als Ergebnis den primitiven Datentyp `boolean` liefern. Java kennt alle von C her bekannten Vergleichsoperatoren. Diese werden in der folgenden Tabelle aufgeführt:

Operator	Bedeutung
==	gleich
!=	ungleich
<	kleiner als
<=	kleiner oder gleich
>	grösser als
>=	grösser oder gleich
!	nicht

Hinweis

▷ Beachten Sie, dass der **Test auf Gleichheit** mit dem doppelten Gleichheitszeichen gemacht werden muss. Ein einfaches Gleichheitszeichen entspricht einer Zuweisung.

Mithilfe von **Verknüpfungsoperatoren** lassen sich verschiedene Vergleiche zusammenfassen. Nachfolgend werden die logischen Verknüpfungsoperatoren von Java aufgeführt:

Operator	Bedeutung
&&	logisches UND
||	logisches ODER

Für die folgenden Beispiele und Erläuterungen gehen wir von diesen Programmzeilen aus:

- `int zahl1 = 17;`
- `int zahl2 = -5;`

Vergleich	Ergebnis	Begründung
`zahl1 >= 20`	`false`	17 ist nicht grösser oder gleich 20.
`zahl1 != zahl2`	`true`	Die beiden Zahlen sind nicht gleich.
`(zahl1 > 0) && (zahl1 > zahl2)`	`true`	17 ist grösser als 0 und gleichzeitig ist 17 grösser als –5.
`(8 < zahl1) && (zahl1 < 20)`	`true`	17 liegt zwischen 8 und 20.
`(zahl1 > 50 && zahl1 < 100) || zahl2 < 0`	`true`	True, obwohl 17 nicht zwischen 50 und 100 liegt, ist der Ausdruck erfüllt, weil –5 kleiner als 0 ist.

Vergleichsoperatoren werden von Java immer von links nach rechts gelesen – ausser wenn Sie **Klammern zur Gruppierung** verwenden. Sobald das Ergebnis des Ausdrucks eindeutig ist, beendet Java die weitere Auswertung. Als Programmierer können Sie sich also nicht

darauf verlassen, dass immer alle Teilausdrücke ausgewertet werden. Sehen Sie sich beispielsweise folgenden Ausdruck an:

```
(a == 17) || (b > 0)
```

Wenn a gleich 17 ist, wird b > 0 nie ausgewertet, da bereits nach dem ersten Test klar ist, dass der vollständige Ausdruck erfüllt wird.

Nun sehen Sie sich diesen Ausdruck an:

```
(a == 17) || rechne(3, 88)
```

Die Funktion `rechne()` wird nicht ausgeführt, wenn a gleich 17 ist. Je nachdem, was die Funktion `rechne()` alles machen muss, kann es Problemen geben, wenn diese Funktion nicht immer ausgeführt wird. Da Java also bei Bedarf eine Abkürzung nimmt, bezeichnet man die Operatoren auch als **Shortcut-Operator**[1].

6.2 Einfache Selektion

Wenn der Programmablauf aufgrund eines Werts oder einer Bedingung geändert werden muss, sieht die **Struktur einer einfachen Selektion bzw. Verzweigung** wie folgt aus:

```
if ( Bedingung ) {
    // Codeblock 1
}
else {
    // Codeblock 2
}
```

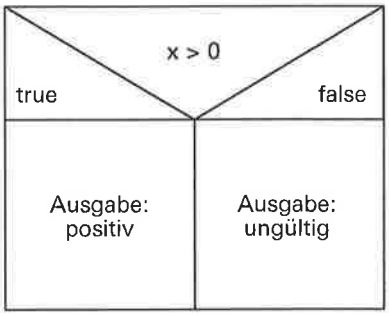
Hier ein Beispiel für eine **einfache Selektion:**

```
if (x > 0) {
    System.out.println("Eingabe ist positiv");
}
else {
    System.out.println("Ungueltige Eingabe");
}
```

Die Bedingung muss während der Laufzeit des Programms eindeutig als wahr oder falsch erkennbar sein. Wenn die Bedingung erfüllt ist, wird der Codeblock 1 ausgeführt, wenn die Bedingung nicht zutrifft, arbeitet das Programm den Codeblock 2 ab. Sie dürfen den else-Abschnitt ersatzlos weglassen. Die geschweiften Klammern umfassen die beiden Codeblöcke. Damit markieren Sie mehrere zusammengehörende Programmzeilen. Wenn der Codeblock aus nur einer Zeile besteht, dürfen Sie auch die Klammern weglassen. Es ist jedoch zu empfehlen, die Klammern trotzdem zu setzen. Die **Verzweigung** kann als **Struktogramm** wie folgt dargestellt werden:

[1] Engl. für: Abkürzungsoperator bzw. abgekürzter Operator.

[6-1] Struktogramm der Verzweigung



Möglicherweise kennen Sie von der Programmiersprache C her bereits die abgekürzte Schreibweise der **if/else-Verzweigung:**

```
int absolutBetrag = ( a < 0 ? -a : a);
```

Dies ist gleichbedeutend mit:

```
int absolutBetrag;
if (a < 0) {
    absolutBetrag = -a;
}
else {
    absolutBetrag = a;
}
```

Sie sehen eine massive Verkürzung des Codes. Diese Konstruktion heisst **konditionaler** oder **ternärer**[1] **Operator** und wird idealerweise bei selektiven Zuweisungen verwendet, wie im obigen Beispiel deutlich zu sehen ist: Wenn a kleiner als null ist, verwende –a für die Zuweisung, sonst a. Ein weiteres Beispiel dazu:

```
System.out.println("Guten Tag Herr " + (name == null ? "Unbekannt" : name) );
```

Wenn ein Name gegeben ist, wird die Begrüssung mit Name ausgegeben, ansonsten wird die Person mit «Unbekannt» angesprochen.

6.3 Mehrfachselektion

Von der Programmiersprache C her kennen Sie auch die **switch/case-Anweisung.** Diese **Mehrfachselektion** erlaubt es, wie ein Stufenschalter aus mehreren Abläufen zu wählen.

```
switch ( Selektor ) {
    case wert1 : Codeblock; break;
    case wert2 : Codeblock; break;
    ...
    default : Codeblock;
}
```

Hier kann als «Schalter» ein ganzzahliger Wert stehen, d. h., es gibt keine Zwischenstufen. `Selektor` ist ein Ausdruck, der einen ganzzahligen Wert oder ein Zeichen ergibt. Der Wert wird nun von oben nach unten mit den gegebenen Werten `wert1`, `wert2` etc. verglichen. Bei einer Übereinstimmung wird der dort stehende Codeblock ausgeführt. Wird keine Übereinstimmung gefunden, wird der Codeblock bei `default` ausgeführt.

[1] Dreiwertig (wörtl).

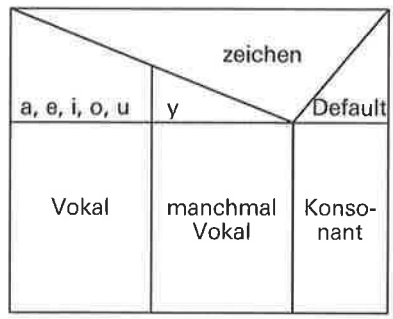
Alle Codeblöcke können aus einer Anweisung oder aus mehreren Anweisungen bestehen. Interessanterweise müssen diese nicht mit geschweiften Klammern zusammengefasst werden. Die **break-Anweisung** nach den Codeblöcken sorgt dafür, dass das Programm aus den Klammerblöcken ausbricht und die Programmausführung unterhalb der schliessenden Klammer weitergeht. Fehlt das `break`, wird der nachfolgende Codeblock ebenfalls ausgeführt. Sehen Sie sich dazu das folgende Beispiel an:

```
char zeichen;
// irgendwoher wird ein Zeichen geholt, z.B. durch Benutzereingabe
switch (zeichen) {
    case 'a' :
    case 'e' :
    case 'i' :
    case 'o' :
    case 'u' : System.out.println("Vokal"); break;
    case 'y' : System.out.println("Manchmal als Vokal gesprochen"); break;
    default : System.out.println("Konsonant");
}
```

Hier haben die Fälle `a`, `e`, `i` und `o` keinen Codeblock und kein `break`. Wenn einer dieser Buchstaben oder ein `u` in der Variablen `zeichen` steht, «fällt» die Ausführung durch bis zum `u`. Dort wird dann ausgegeben, dass es sich um einen Vokal handelt.

Weitere Ausgaben erfolgen nicht, da der Codeblock bei `u` mit `break` abgeschlossen ist. Wenn `zeichen` ein `y` ist, erfolgt die Ausgabe «Manchmal als Vokal ausgesprochen». Danach wird ebenfalls wegen der break-Anweisung nach der switch/case-Konstruktion weitergemacht. In allen anderen Fällen landet die Ausführung bei der Marke `default`. Die **Mehrfachselektion** kann **als Struktogramm** wie folgt dargestellt werden:

[6-2] Struktogramm der Mehrfachselektion



6.4 Schleifen

Müssen Teile des Programmcodes mehrmals hintereinander ausgeführt werden, kommen **Schleifen** zum Einsatz. Auch dafür verwendet Java die aus C bekannten Konstrukte.

6.4.1 While-Schleife

Die **while-Schleife** ist eine kopfgesteuerte Schleife mit folgender Struktur:

```
while ( Bedingung ) {
    // Anweisungen
}
```

Hier werden die Anweisungen ausgeführt, solange die Bedingung erfüllt ist. Folglich wird vor jedem Durchgang getestet, ob die Schleifenbedingung (immer noch) erfüllt ist. Dabei gilt auch, dass die geschweiften Klammern mehrere Anweisungen zusammenfassen.

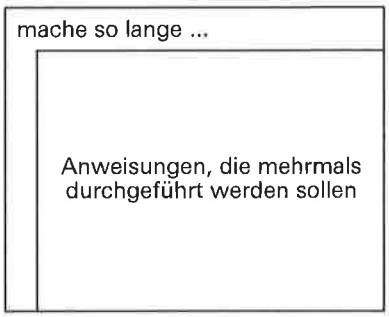
Wenn nur eine Anweisung im Schleifenkörper steht, dürfen die Klammern weggelassen werden. Es empfiehlt sich jedoch, die Klammern immer zu setzen, denn je nach Bedingung kann es sein, dass bei der while-Schleife der Anweisungsblock nie ausgeführt wird.

Hinweis

▷ Nach der schliessenden runden Klammer darf kein Semikolon stehen, weil der Schleifenkörper dadurch als eigenständiger Codeblock von der Schleife getrennt wird. Auch wenn dies technisch zulässig ist, ist es wahrscheinlich nicht das, was Sie beabsichtigen.

Die **kopfgesteuerte Schleife** kann **als Struktogramm** wie folgt dargestellt werden:

[6-3] Struktogramm der kopfgesteuerten Schleife



6.4.2 Do/while-Schleife

Die **do/while-Schleife** ist eine fussgesteuerte Schleife mit folgender Struktur:

```
do {
    // Anweisungen
} while ( Bedingung );
```

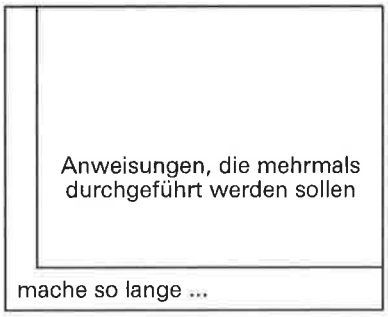
Bei einer do/while-Schleife wird auf jeden Fall ein Durchgang gemacht und erst danach geprüft, ob ein weiterer Durchgang notwendig ist. Es wird also nach jedem Durchlauf getestet, ob die Bedingung für einen erneuten Durchgang weiterhin erfüllt ist.

Hinweis

▷ Vor `while` darf kein Semikolon stehen, nach der Bedingung muss eines stehen.

Die **fussgesteuerte Schleife** kann **als Struktogramm** wie folgt dargestellt werden:

[6-4] Struktogramm der fussgesteuerten Schleife



6.4.3 For-Schleife

Die **for-Schleife** ist ebenfalls eine kopfgesteuerte Schleife mit folgender Struktur:

```
for ( Initialisierung ; Bedingung ; Schritt ) {
    // Anweisungen
}
```

Im Initialisierungsbereich wird alles vorbereitet, was für die Durchführung der for-Schleife benötigt wird. Meist werden hier Zählervariablen definiert. Die for-Schleife wird so lange abgearbeitet bzw. durchlaufen, wie die Bedingung erfüllt wird. Nach jedem Durchgang werden die Schritt-Anweisungen ausgeführt. Dabei wird die Zählervariable normalerweise um einen Schritt erhöht bzw. weitergezählt. Sehen Sie sich dazu folgendes Beispiel an:

```
for (short index = 0; index < 10; index++) {
    System.out.println("Durchgang Nummer " + index);
}
```

Hinweis

▷ Weitere Beispiele für die for-Schleife finden Sie in Kapitel 13, S. 125.

6.4.4 Erweiterte for-Schleife

Seit Java 5 gibt es eine **erweiterte for-Schleife,** die speziell dafür geeignet ist, die Elemente eines Arrays oder eines Collection-Objekts durchzugehen. In C gibt es diese Konstruktion nicht. In PHP wird ein solcher Schleifentyp mit `foreach` gekennzeichnet. Die allgemeine Struktur dieser Konstruktion sieht wie folgt aus:

```
for ( Typ einElement : Array ) {
    // Anweisungen
}
```

Der erste Bestandteil vor dem Doppelpunkt deklariert eine Variable für ein Element. Der zweite Bestandteil nach dem Doppelpunkt steht für das Array, dessen Elemente durchgegangen werden. Java geht alle Elemente des Arrays durch und setzt das gerade aktuelle Array-Element in die vor dem Doppelpunkt deklarierte Variable. Vergleichen Sie dazu die folgende Tabelle. Links sehen Sie die «klassische» for-Schleife und rechts die erweiterte Version. In beiden Fällen werden alle Elemente des Arrays ausgegeben:

Klassische for-Schleife	Vereinfachte for-Schleife

`int[] zahlen = {1, 4, -3, 0, -1};`
`for (short i = 0; i < zahlen.length; i++) {`
`  System.out.println(zahlen[i]);`
`}`
	`int[] zahlen = {1, 4, -3, 0, -1};`
	`for (int zahl : zahlen ) {`
	`  System.out.println(zahl);`
	`}`

Hinweis

▷ Die besonderen Vorzüge der erweiterten for-Schleife kommen erst im Zusammenhang mit Collections richtig zum Tragen. Vergleichen Sie dazu das Kapitel 5.4, S. 61.

Zur Steuerung des Programmablaufs kennt Java die von anderen Programmiersprachen her bekannten **Kontrollstrukturen:**

- **Verzweigungen** können mithilfe von if/else- und switch/case-Anweisungen realisiert werden.
- Für **Schleifen** stehen Ihnen die while-, do/while- und for-Schleifen zur Verfügung. Seit Java 5 gibt es eine interessante Erweiterung der for-Schleife, die besonders geeignet ist, über **Arrays** zu iterieren.

Repetitionsfragen

36 Wie lautet der Operator für eine logische UND-Verknüpfung?

37 Was bedeutet folgende Codezeile?

```
a = (a >0 ? a : -a);
```

38 Worin besteht in Java der Unterschied zwischen dem einfachen Gleichheitszeichen (=) und dem doppelten Gleichheitszeichen (==)?

39 Wie oft werden eine kopfgesteuerte Schleife und eine fussgesteuerte Schleife mindestens durchlaufen?

7 Funktionen und Methoden

Was in C/C++ als eine **Funktion** bezeichnet wird, nennt man in der OOP eine **Methode.** Nachfolgend wird aufgezeigt, wie Methoden bei Java typischerweise eingesetzt werden.

7.1 Methode erstellen

Eine **Methode** wird in Java gleich aufgebaut wie eine Funktion in C. Sehen Sie sich dazu das folgende Beispiel an:

```
int addieren(int zahl1, int zahl2) {
    int ergebnis = zahl1 + zahl2;
    return ergebnis;
}
```

Hier wird eine Methode `addieren` definiert, die zwei int-Werte als Eingabeparameter benötigt. Die int-Werte werden addiert und das Ergebnis – ebenfalls vom Datentyp `int` – an den aufrufenden Code zurückgegeben. Diese Methode kann z. B. wie folgt verwendet werden:

```
System.out.println("Das Ergebnis ist " + addieren(17, 4) );
```

Im obigen Beispiel lautet die Ausgabe: `Das Ergebnis ist 21.`

Als **Datentypen für Methoden** kommen alle einfachen Datentypen von Java infrage. Beachten Sie, dass in Java jede Klasse einen Datentyp darstellt, unabhängig davon, ob sie vom Programmierer erstellt wurde oder es sich um ein Klasse der Java-Laufzeitumgebung handelt. Aus diesem Grund ist der folgende Code absolut korrekt und sicher:

```
void sagen(String einSatz) {
    System.out.println("ich sage den Satz: " + einSatz);
}
```

Im obigen Beispiel hat die Methode `sagen()` einen Eingabewert vom Typ `String`.

Hinweise

▷ Beachten Sie, dass Java **keine Prototypen** kennt. Entsprechend müssen bei Methoden keine Prototypen aufgeschrieben werden.

▷ Weil der Java-Compiler immer zuerst die komplette Quelldatei liest, bevor er den Code übersetzt, müssen **Methoden und Variablen** vor dem ersten Aufruf **nicht bekannt** gemacht werden. Dies bedeutet eine grosse Vereinfachung gegenüber C/C++.

7.2 Parameter übergeben

Vielleicht haben Sie im Zusammenhang mit der Übergabe von Parametern die Begriffe **Wertaufruf** («pass by value») und **Referenzaufruf** («pass by reference») schon einmal angetroffen.

7.2.1 Pass by value

Dies ist bei Java die normale Art und Weise, um Werte an eine Methode zu übergeben. Dabei wird der zu übergebende Wert in eine lokale Variable in der Methode kopiert. Dort kann der Wert ausgelesen werden. Da es sich um eine Kopie des Originals handelt, darf

die Methode diesen Wert verändern, ohne dass der Originalwert davon betroffen wird. Alle primitiven Datentypen werden in Java «by value» übergeben.

7.2.2 Pass by reference

Diese Art und Weise der Parameterübergabe muss in C mittels Zeigern realisiert werden. Statt den Wert selbst zu übergeben, wird eine Referenz auf diesen Wert an die Methode weitergegeben. Wenn die Methode über diesen Verweis auf den Wert zugreift, kann sie den Originalwert verändern.

In Java werden **alle Objekte immer «by reference»** übergeben. Entsprechend werden nie die Objekte selbst, sondern nur Verweise darauf an die Methode weitergereicht.

In Kapitel 5.3, S. 56 haben Sie erfahren, dass Java keine Zeiger bietet. Dies bedeutet, dass **primitive Datentypen nicht «by reference»** übergeben werden können. Sie müssen in diesem Fall mit einer **Wrapper-Klasse** arbeiten.

7.3 Methodensignatur und überladene Methode

In Java und in anderen objektorientierten Programmiersprachen ist es möglich, mehrere Methoden mit dem gleichen Namen zu definieren. Zur Unterscheidung fügt der Compiler dem Methodennamen die Datentypen der Eingabeparameter hinzu. Diese Kombination aus Bezeichnung und Parameter wird auch **Signatur einer Methode** genannt.

In der OOP wird die Möglichkeit oft genutzt, Methoden mit demselben Namen gleichzeitig zu verwenden. Der Compiler kann dann anhand der Signatur erkennen, welche Methode gemeint ist. Wenn Sie diesen Mechanismus im Programmcode einsetzen, spricht man von einer **überladenen Methode.**

Hinweis

▷ Beachten Sie, dass der Rückgabedatentyp einer Methode nicht zur Signatur gehört.

Der Compiler kann beispielsweise folgende **drei rechne()-Methoden** unterscheiden:

```
int rechne (int zahl1, int zahl2)              // Signatur ist rechne_int_int
int rechne (int zahl1, double zahl2)           // Signatur ist rechne_int_double
int rechne (int zahl1, int zahl2, double zahl3) // Signatur ist rechne_int_int_double
```

Wird nun eine Methode `rechne()` aufgerufen, kann der Compiler anhand der Signatur erkennen, welche rechne()-Methode gemeint ist. Dabei wählt der Compiler immer diejenige Methode aus, die am besten passt.

Methode 1

```
int ergebnis = rechne(17, 4);
```

Hier wird die erste Methode aufgerufen, weil die Parameter 17 und 4 jeweils als int-Werte interpretiert werden können. Möglich wäre zwar auch die zweite Methode, dort muss der Compiler die Zahl 4 jedoch in 4.0 umwandeln.

Methode 2

Wenn Sie die Zahl 4 als double-Zahl schreiben, wird die zweite Methode aufgerufen:

```
int ergebnis = rechne(17, 4.);
```

Methode 3

Für folgende Signatur gibt es keine unmittelbar passende Methode:

```
int ergebnis = rechne(17, 4, -2);
```

Durch Anpassung der Zahl –2 in die Zahl –2.0 kann der Compiler den Aufruf passend zur dritten Methode machen.

Analysieren Sie folgenden Aufruf. Was geschieht in dieser Situation?

```
void machWas(String eingabe, int wert)
int machWas(String x, int y)
```

Hier haben beide Methoden die Signatur `machWas_String_int`. Aus diesem Grund kann sie der Compiler nicht unterscheiden. Er wird also reklamieren und das Programm nicht übersetzen.

Die objektorientierte Programmiersprache Java verwendet praktisch die **gleiche Syntax** wie die prozedurale Programmiersprache C.

Funktionen heissen in Java **Methoden** und brauchen keinen **Prototyp.** Damit entfällt eine grosse Quelle von Aufwand und möglichen Fehlern. Primitive Datentypen werden in Java immer **«by value»** an Methoden übergeben, Objekte immer **«by reference».**

Repetitionsfragen

40 Erklären Sie die Begriffe `pass by reference` und `pass by value`.

41 Was wird in Java unter der Signatur einer Methode verstanden?

42 Welche Methode wird im folgenden Fall aufgerufen? Warum?

```
void doit(String a, Double b){...};
void doit(String a, Integer b){...};
...
doit("Hallo Welt", 3);
```

43 Welche Methode wird im folgenden Fall aufgerufen? Warum?

```
void doit(String a, int b){...};
void doit(String a, Integer b){...};
...
doit("Hallo Welt", 3);
```

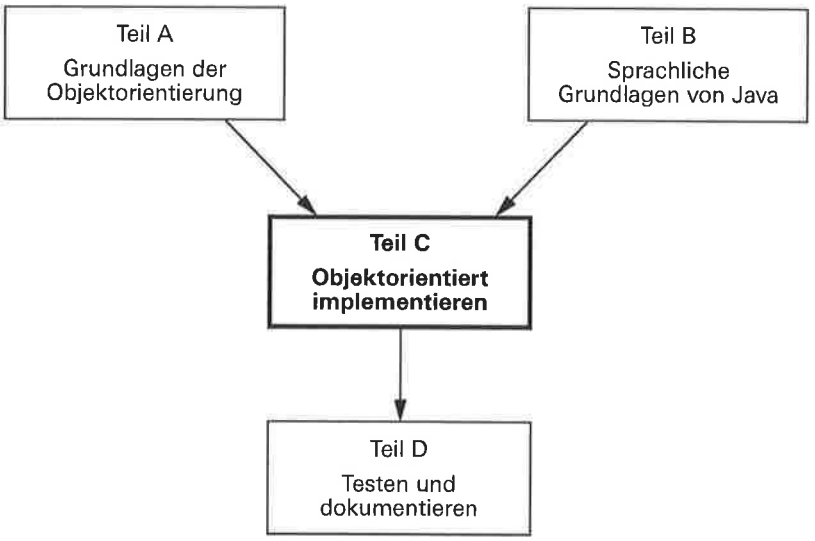
Teil C OOD in Java implementieren

Einleitung, Lernziele und Schlüsselbegriffe

Einleitung

In diesem Teil des Lehrmittels erfahren Sie, wie objektorientierte Konzepte und Entwürfe mithilfe der Programmiersprache Java realisiert werden können. Dabei lernen Sie **weitere Sprachelemente von Java** und **typische Werkzeuge für die OO-Entwicklung** kennen und können Sie mitverfolgen, wie ein kleines Beispielprojekt objektorientiert implementiert wird. Folgende Grafik zeigt, wo Sie sich innerhalb des Lehrmittels befinden:

Teil C im Gesamtzusammenhang



Lernziele und Lernschritte

	Lernziele	Lernschritte
☐	Sie können die Beziehungstypen zwischen Klassen und das Konzept der Vererbung beschreiben und aufzeigen, wie diese umgesetzt werden.	• Eine Klasse in Java deklarieren • Vererbung implementieren • Zugriffsmodifikatoren
☐	Sie können an einem Codebeispiel den Effekt von Polymorphie aufzeigen.	• Vererbung implementieren
☐	Sie kennen die Notation von Schnittstellen und Paketen und können aufzeigen, wie sich diese auf die Implementation auswirken.	• Interfaces und abstrakte Methoden implementieren • Pakete in Java • Schnittstellen zur Aussenwelt
☐	Sie können den Programmcode vollständig und korrekt dokumentieren, um daraus die API-Spezifikation abzuleiten.	• Wo finde ich die Java-Dokumentation? • Wie ist die Java-Dokumentation aufgebaut? • Wie wird die Klasse String dokumentiert?
☐	Sie können die Notation von Schnittstellen und Paketen interpretieren und aufzeigen, wie sich diese auf die Implementation auswirken.	• Schnittstellen zur Aussenwelt • Pakete in Java • Objektorientierte Anwendung realisieren
☐	Sie können aufzeigen, wie durch die Nutzung von Schnittstellen der Code unabhängig erstellt und getestet werden kann.	• Schnittstellen zur Aussenwelt • Objektorientierte Anwendung realisieren
☐	Sie können die grundlegenden Funktionen eines CASE-Tools beschreiben und erläutern, mit welchen Funktionen die Implementation unterstützt wird.	• Arbeiten mit Eclipse • CASE-Tools
☐	Sie können die Struktur einer Systemdokumentation und ihre Bedeutung für Wartung und Nachvollziehbarkeit darlegen.	• Java-Dokumentation

Schlüsselbegriffe

Accessor, ArgoUML, CASE, catch, Eclipse, Exception, extends, getter, I/O, import, InputStream, instanceof, Instanzvariable, java.*, Klasse, Klassenmethode, Klassenvariable, OutputStream, package, Paket, private, protected, public, Reader, setter, static, super, this, throw, throws, try, Writer

8 Sprachkonzepte und -elemente implementieren

Sie haben bereits erfahren, dass eine **Klasse** einem Bauplan für **Objekte** entspricht. Alle Objekte, die nach dem gleichen Bauplan gefertigt werden, sind **Instanzen derselben Klasse.** Sie zeigen das gleiche Verhalten und haben die gleichen Datenstrukturen, aber nicht die gleichen Daten. Im einleitenden Krimiprojekt ist **David** beispielsweise eine Instanz der Klasse **Regisseur,** d. h., es gibt nur einen Regisseur namens David, es kann aber weitere Regisseure geben, die das Gleiche können wie er. In diesem Kapitel wird die Deklaration von Klassen und Objekten vertieft und Sie erfahren mehr über die korrekte Implementation zentraler Sprachkonzepte und -elemente von Java anhand neuer Anwendungsbeispiele.

8.1 Klassen deklarieren

Hier noch einmal die **Deklaration einer Klasse** aus unserem Krimiprojekt:

```
public class Regisseur {
    // hier kommt dann der ganze Inhalt der Klasse
}
```

Hinweis

▷ Den Schlüsselbegriff `public` können Sie im Moment ignorieren. In Kapitel 8.3, S. 80 wird näher darauf eingegangen.

Für die Klassendeklaration benötigen Sie das Schlüsselwort `class` und den Namen der Klasse. Beachten Sie in Bezug auf den **Klassennamen** folgende Regeln:

- Ein Klassename sollte keine Tätigkeit oder Eigenschaft beschreiben, sondern ein «Ding». Objekte haben Eigenschaften und tun etwas.
- Ein Klassenname muss mit einem Grossbuchstaben beginnen.
- Erstellen Sie für jede Klasse eine eigene Datei.[1] Der Dateiname muss mit dem Namen der Klasse übereinstimmen (inkl. Gross-/Kleinschreibung) und die Erweiterung **.java** aufweisen.
- Ein Klassenname muss mit einem Buchstaben oder Unterstrich beginnen, darf aber auch Ziffern und das $-Zeichen enthalten. Verwenden Sie keine Umlaute, auch wenn das manchmal funktioniert.
- Trennen Sie mehrteilige Klassennamen optisch durch den sogenannten Camel-Case, indem Sie die Anfangsbuchstaben der Teilwörter jeweils grossschreiben.

Hier sehen Sie einige **gültige Klassennamen:**

Gültige Klassennamen
• Regisseur • WasserFahrzeug • EditingContex • GrafikContext3D

[1] Genau genommen stimmt diese Regel nicht ganz. Es können mehrere Klassendeklarationen in derselben Datei stehen, aber nur eine davon darf mit «public» markiert sein. Der Dateiname muss dann genau mit dem Namen dieser Klasse übereinstimmen.

In der folgenden Tabelle sind einige Beispiele für **ungültige Klassennamen** aufgeführt:

Ungültige Klassennamen	Begründung
`33erTaxi`	Beginnt mit einer Ziffer
`Dreiunddreissiger Taxi`	Beinhaltet einen Leerschlag
`Text-Format`	Minuszeichen ist nicht erlaubt
`Haus.Besitzer`	Punkt ist Sonderzeichen und daher nicht erlaubt

Hier noch einige gültige, aber **schlecht gewählte Klassennamen:**

Gültige, aber ungeschickte Klassennamen	Begründung	Besserer Klassenname
`Zweit_Meinung`	Beinhaltet einen Unterstrich	ZweitMeinung (Camel-Case)
`eisenbahn`	Beginnt mit Kleinbuchstaben	Eisenbahn
`GültigeWerte`	Beinhaltet Umlaute	GueltigeWerte
`Starten`	Beschreibt eine Tätigkeit	Start

8.2 Variablen und Methoden implementieren

Erinnern Sie sich noch, wie sich die Zeugen im Krimi ihre Aussagen merken können?

8.2.1 Instanzvariablen deklarieren

In der Klasse **Zeuge** werden Variablen deklariert. Hier noch einmal der entsprechende Codeausschnitt:

```
public class Zeuge {
    private String aussage = "Ich habe keine Ahnung.";
...
```

Hinweis

▷ Den Schlüsselbegriff `private` können Sie im Moment ignorieren. In Kapitel 8.3, S. 80 wird näher darauf eingegangen.

Jedes Objekt, das von der Klasse **Zeuge** erstellt wird, hat die Variable `aussage` vom Typ `String`. Auf diese Weise können Sie in jeder Klasse beliebige Variablen deklarieren. Diese sind danach bei allen Objekten der entsprechenden Klasse vorhanden und jedes Objekt verfügt über einen eigenen, unabhängigen Speicherplatz dafür. Solche Variablen werden **Instanzvariablen** genannt.

Wenn Sie das obige Beispiel genauer betrachten, sehen Sie, dass die Variable `aussage` bei der Deklaration initialisiert wird. Jeder Zeuge hat daher zu Beginn den gleichen Wert in der Variable `aussage`.

8.2.2 Auf Instanzvariablen zugreifen

Sie können eine Instanzvariable wie jede andere Variable verwenden. Beim **Zeugen** sieht der entsprechende Code etwa wie folgt aus:

```
public void setAussage(String text) {
    aussage = text;
}
```

Die Methode `setAussage()` erhält beim Aufruf einen `String` im Übergabeparameter `text`. In der Folge wird dieser `String` der Instanzvariablen `aussage` zugewiesen. Denken Sie daran, dass nicht der String selbst übergeben wird, sondern nur eine Referenz auf das entsprechende Objekt.[1] Wenn Sie auf eine Instanzvariable eines anderen Objekts zugreifen wollen, müssen Sie eine **Objektreferenz** mit einem Punkt voranstellen.

Hinweis

▷ Beachten Sie, dass diese Art des Zugriffs nicht immer möglich ist. Näheres über die Zugriffsarten und deren Anpassung finden Sie in Kapitel 8.6, S. 88.

8.2.3 Schlüsselwort «this» verwenden

Es ist nicht immer sofort erkennbar, ob es sich bei einer Variablen um eine **Instanzvariable** oder eine **lokale Variable** handelt. In den meisten Fällen ist dies auch kein Problem. Manchmal möchte oder muss man den Unterschied allerdings ausdrücklich klarmachen.

Der folgende Codeabschnitt beinhaltet eine Abwandlung der Methode `setAussage()`.

```
public void setAussage(String aussage) {
    this.aussage = aussage;
}
```

Erkennen Sie den Unterschied? Der **Übergabeparameter** heisst hier nicht mehr `text`, sondern `aussage`. Da es sich dabei um eine lokale Variable handelt, ist dieser Name korrekt und drückt den Zweck des Parameters bestens aus. Problematisch ist nun, dass innerhalb der Methode `setAussage()` zwei Variablen mit dem Namen `aussage` vorkommen: der Parameter und die Instanzvariable. In dieser Situation können Sie mit dem Schlüsselwort `this`[2] klarmachen, welche Variable gemeint ist. Bei `this.aussage` handelt es sich also um eine Referenz auf die Variable `aussage` des aktuellen Objekts. Damit wird der Wert des Übergabeparameters `aussage` der Instanzvariablen `aussage` zugewiesen. Auch wenn es Ihnen seltsam vorkommt, wenn für zwei verschiedene Variablen derselbe Name verwendet wird: In Java handelt es sich dabei um eine übliche Konstruktion, die oft vorkommt.

8.2.4 Klassenvariablen deklarieren

Nehmen Sie einmal an, Objekte einer Klasse möchten wissen, wie viele es davon gibt. Sie könnten sich z. B. Folgendes ausdenken:

```
public class Person {
    private int anzahl = 0;

    public Person() {
        anzahl++;
    }

    public int soVieleLeute() {
        return anzahl;
    }
...
```

In der Klasse **Person** gibt es eine Instanzvariable `anzahl`, die am Anfang 0 entspricht und im Konstruktor jeweils um 1 erhöht wird. Mit der Methode `soVieleLeute()` kann die Anzahl der Personen abgerufen werden. Das Problem hierbei ist, dass jede Person ihre

[1] Engl. Fachbegriff: pass by reference.

[2] «This» bedeutet hier: «dieses (Objekt) selbst» und verweist auf das eigene Objekt.

eigene Instanzvariable `anzahl` hat und so die Anzahl Personen gar nicht gezählt werden kann. Sie brauchen also eine Variable, die unabhängig von einzelnen Personen quasi über den Objekten steht. Dies erreichen Sie mit dem Schlüsselwort `static` vor der Variablendeklaration. Eine solche Variable nennt man **Klassenvariable.**

```
public class Person {
    private static int anzahl = 0;

...
```

Hier haben Sie eine Variable `anzahl`, die in der Klasse deklariert ist, deren Speicherplatz aber unabhängig von Objekten existiert. Die Variable gibt es sogar dann, wenn überhaupt keine Objekte vorhanden sind. Nun funktioniert das Zählen der Personen wie gewünscht. So wie Sie bei Instanzvariablen mit `this` das aktuelle Objekt referenzieren, können Sie den Klassenvariablen den **Klassennamen** voranstellen. In unserem Fall ist das nicht unbedingt nötig, da der Zugriff aus der Klasse erfolgt, in der die Variable deklariert ist. Aber in neueren Java-Versionen wird der Compiler reklamieren, dass man statisch auf eine static-Variable zugreifen soll. Der korrekte Code muss dann so aussehen:

```
public class Person {
    private static int anzahl = 0;

    public Person() {
        Person.anzahl++;
    }

    public int soVieleLeute() {
        return Person.anzahl;
    }
...
```

8.2.5 Klassenmethoden implementieren

Bisher waren alle Methoden, die Sie erstellt oder verwendet haben, sogenannte **Instanzmethoden.** Um eine solche Methode aufrufen zu können, müssen Sie ein Objekt haben, das Sie ansprechen konnten. Da Methoden in erster Linie dazu dienen, Objekten ein bestimmtes Verhalten beizubringen, ist das in Ordnung so.

Manchmal brauchen Sie aber eine von Objekten losgelöste Methode. In Java können Sie dazu das Schlüsselwort `static` verwenden. Wie bei Klassenvariablen muss der Zugriff auf eine solche Methode über den Klassennamen erfolgen. Sie werden in Kapitel 11.2, S. 110 eine Klasse kennenlernen, die einfache Methoden zur Eingabe von Zahlen und Strings über die Konsole zur Verfügung stellt. Diese Methoden sind alle statisch. In der Klasse **ConsoleReader** gibt es beispielsweise eine Methode mit folgender Signatur:

```
public static int readInteger() {
    ...
```

Sie sehen das Schlüsselwort `static`. Ansprechen können Sie diese Methode wie folgt:

```
int eingabe = ConsoleReader.readInteger();
```

Sie müssen also den Klassennamen vor den Methodenaufruf setzen, da Sie kein Objekt haben, das die Methode ausführen kann.

8.3 Datenkapselung implementieren

Sie haben bereits erfahren, dass Objekte Daten und Verhaltensweisen umfassen. In der objektorientierten Programmierung gibt es ein wichtiges Konzept: Es besagt, dass das «Innenleben» eines Objekts (und somit seine Daten) nur für das Objekt selbst wichtig ist, nicht aber für die Aussenwelt. Dieses Konzept ist unter der Bezeichnung **Kapselung** bzw. unter dem englischen Ausdruck **Data Hiding** bekannt.

8.3.1 Kapselung realisieren

Obwohl Sie im Prinzip von aussen direkt auf die Instanzvariablen von Objekten zugreifen können, sollten Sie ein Objekt immer nach der gewünschten Information fragen. Aussenstehende müssen schliesslich nicht wissen, wie diese Information intern gespeichert ist. Und als Programmierer möchten Sie die Art der Speicherung innerhalb von Objekten ggf. ändern (beispielsweise, um eine Performanceverbesserung zu erreichen), ohne dass die «Benutzer» der Objekte davon betroffen sind.

Folgendes Beispiel soll dies verdeutlichen:

```
public class Person {

    private String name;

    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return this.name;
    }
}
```

Sie sehen hier die Klasse **Person,** die einen einzigen Namen hat. Diesen Namen merkt sich die Person in einer Instanzvariablen, der das Schlüsselwort `private` vorangestellt wird. Die Variable `name` kann hier nur innerhalb der Klasse **Person** «eingesehen» werden und ist für die anderen Klassen unsichtbar. Dafür liegen zwei weitere Methoden mit dem Schlüsselwort `public` vor, denen `set` bzw. `get` vorangestellt ist. Dies bedeutet, dass jede Java-Programmzeile in jeder anderen Klasse diese Methoden einsehen und ansprechen kann. Mehr über die Begriffe `public` und `private` erfahren Sie in Kapitel 8.6, S. 88.

Wenn Sie nun einer Person einen Namen geben möchten, können Sie die Methode `setName()` verwenden. Wenn Sie von einer Person den Namen abfragen möchten, rufen Sie die Methode `getName()` auf. Falls eine logische Eigenschaft abgefragt werden soll, die nur `true` oder `false` sein kann, verwenden Sie nicht `get`, sondern setzen das englische Wort `is` voran. Vergleichen Sie dazu folgenden Beispielcode:

```
Person hans = new Person();
hans.setName("Hans");
System.out.println("Die Person heisst " + hans.getName() );
while (hans.isSingle()) {
    hans.goOutForParty();
...
```

Sie können nun das «Innenleben» der Klasse **Person** beliebig verändern und z. B. den Namen einer Person nicht mehr in einer Instanzvariablen, sondern in einer Datenbank abspeichern. Der obige Code für die Namensgebung und Namensausgabe der Person kann dabei unverändert bestehen bleiben.

8.3.2 Zugriffsmethode definieren

Instanzvariablen sollten wenn immer möglich `private` sein. Für Daten, die man von extern setzen oder abfragen können soll, muss eine entsprechende **Zugriffsmethode** vorliegen. Nennen Sie die Zugriffsmethode immer so, dass die Bedeutung der Information klar wird. Eine **Methode zur Übergabe eines (neuen) Objektwerts** muss `set` vor dem Namen der Information haben und wird **setter** bezeichnet. Eine **Methode zur Abfrage eines Objektwerts** muss das Wort `get` vor dem Namen der Information haben und wird oft als **getter** bezeichnet. In der folgenden Tabelle finden Sie einige Beispiele dafür:

Information	Mögliche Instanzvariable	Setter	Getter
Farbe eines Autos	`farbe`	`setFarbe(Farbe farbe)`	`getFarbe()`
Preis eines Artikels	`preis`	`setPreis(double preis)`	`getPreis()`
Geburtsdatum einer Person	`gebDatum`	`setGebDatum(Date datum)`	`getGebDatum()`
Alter einer Person	–	–	`getAlter()`

Beachten Sie, dass es sinnlos ist, einer Person ein bestimmtes Alter zuzuordnen und das Alter einer Person in einer Instanzvariablen zu speichern. Besser wäre es, in einer Instanzvariablen das Geburtsdatum zu hinterlegen. Wird eine Person nach ihrem Alter gefragt, kann dieses anhand des Geburtsdatums berechnet werden. Von aussen her ist jedoch nicht ersichtlich, was hinter dem Aufruf der Methode `getAlter()` steckt.

Hinweis

▷ Selbstverständlich darf innerhalb eines Objekts direkt auf private Variablen zugegriffen werden. Die Methoden `set()`, `get()` und `is()` sind nur notwendig, wenn von aussen auf eine Instanzvariable zugegriffen wird.

8.4 Objekte implementieren

8.4.1 Neues Objekt erzeugen und Konstruktor erstellen

Wie man Objekte korrekt erzeugt, haben Sie bereits beim Krimibeispiel gesehen. Neue Objekte werden mit dem Schlüsselwort `new` erstellt. Hier einige Beispiele dazu:

```
Double kommaZahl = new Double(23.785);
String name = new String("Meier");  // hier gibt es die Abkürzung: name = "Meier";
Person einePerson = new Person();
```

Was passiert hier eigentlich genau? Und wozu dienen die runden Klammern? Jemand hat einmal gesagt: «If you want a new house, you call the constructor.» Wenn Sie ein Haus bauen möchten, engagieren Sie einen Baumeister bzw. einen Konstrukteur. Als Java-Programmierer wissen Sie, dass man eine **Methode aufrufen**[1] kann. Funktionen bzw. Methoden müssen in Programmiersprachen mit ähnlicher Syntax wie C/C++ runde Klammern aufweisen. Obige Codezeile besagt: Java: Bereite ein neues Objekt vor und verwende dafür die Methode `Person()`.

Die Methode `Person()` wird **Konstruktor** genannt. Sie unterscheidet sich von anderen Methoden dadurch, dass sie den gleichen Namen hat wie die Klasse, inklusive Gross- und Kleinschreibung. Ein Konstruktor ist eine Methode, die u. a. den Speicherplatz für Objekte bereitstellt, Instanzvariablen initialisiert und weitere Java-interne Verwaltungsaufgaben bei

[1] Engl.: to call.

der Erstellung von Objekten übernimmt. Ein Konstruktor ist also eine Methode fast wie jede andere Methode. Allerdings gibt es Bedingungen, die eine Methode erfüllen muss, damit Java sie als Konstruktor erkennt.

Die **wichtigsten Regeln für Konstruktoren** lauten:

- Der Methodenname ist mit dem Klassennamen identisch.
- Es gibt keinen Rückgabedatentyp, nicht einmal das Schlüsselwort `void`.

In vielen Klassen ist kein Konstruktor-Code vorhanden. Für solche Fälle stellt Ihnen Java standardmässig einen sogenannten **Defaultkonstruktor** zur Verfügung. Im Krimiprojekt wurde beispielsweise für die Klasse **Regisseur** kein Konstruktor angegeben. Wäre der Defaultkonstruktor sichtbar, würde er wie folgt aussehen:

```
public Regisseur() {
    // hier ist irgendwelcher Code von Java für
    // die Konstruktion der Regisseur-Objekte
}
```

Sie sehen, der Defaultkonstruktor ist eine Methode ohne Eingabeparameter.

In komplexeren Java-Programmen müssen Sie in den meisten Klassen **eigene Konstruktoren definieren** – oft sogar mehrere in derselben Klasse. Wenn Sie keinen Konstruktor angeben, erhalten Sie von Java automatisch den Defaultkonstruktor. Wenn Sie einen oder mehrere eigene Konstruktoren erstellen, streicht Java den Defaultkonstruktor. Falls Sie also zusätzlich zum eigenen Konstruktor den Defaultkonstruktor verwenden möchten, müssen Sie diesen selbst definieren.

Innerhalb einer Klasse können Sie grundsätzlich beliebig viele Konstruktoren erstellen, solange diese sich in der Signatur unterscheiden. Sie haben diesen Sachverhalt bereits in unserem Krimibeispiel in der Klasse **Zeuge** gesehen:

```
public Zeuge(String text) {
    super();
    aussage = text;
}
```

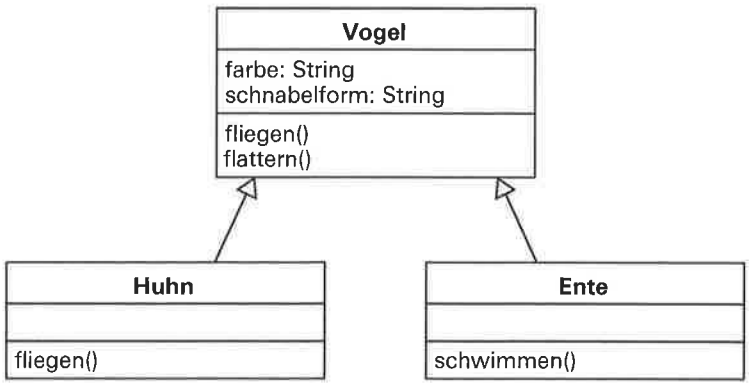
Hier sehen Sie einen Konstruktor `Zeuge` mit dem Parameter `String text`.

Was hat es eigentlich mit der Methode `super()` auf sich? Im Moment müssen Sie sich einfach merken, dass Ihr Konstruktor nicht alles macht, was Java bei der Erstellung eines neuen Objekts benötigt. Damit das Java-Laufzeitsystem Objekte vollständig erstellen kann, muss zuerst die Methode `super()` im zugehörigen Konstruktor aufgerufen werden.

8.5 Vererbung implementieren

Grundlage für die nachfolgenden Erläuterungen zur **Umsetzung einer Vererbung** bildet das folgende Klassendiagramm:

[8-1] Klassendiagramm mit einfacher Vererbungsstruktur



«Ist-ein-Beziehungen» werden in Java durch das Schlüsselwort `extends` angegeben. Hier ein Ausschnitt aus der Klasse **Vogel:**

```
public class Vogel {

    private String farbe;
    private String schnabelform;

    public void fliegen() {
        System.out.println("Vogel fliegt");
    }

    public void flattern() {
        System.out.println("Vogel flattert");
    }

    // weitere Methoden ...
}
```

... und hier der Code für die Klassen **Huhn** und **Ente:**

```
public class Huhn extends Vogel {

    public void fliegen() {
        System.out.println("Huhn versucht zu fliegen");
    }
}

public class Ente extends Vogel {

    public void schwimmen() {
        System.out.println("Ente schwimmt");
    }
}
```

Sie können eine Vererbung über beliebig viele Stufen implementieren. In unserem Fall sind Vögel eine **Unterklasse** der Warmblüter und diese erben wiederum von den Wirbeltieren. In Java hat jede Klasse eine **Oberklasse.** Wenn Sie keine Oberklasse angeben, erbt eine Klasse automatisch von der Klasse **Object.** Object ist also die einzige Klasse in Java, die selbst keine Oberklasse mehr hat. Die Klasse **Object** hat ein paar wenige Methoden, die an alle Klassen vererbt werden.

Hinweise

▷ Unter- und Oberklasse müssen nicht aus demselben Paket stammen.

▷ In Java gibt es keine Mehrfachvererbung, d. h., eine Klasse kann nur von der direkten Oberklasse erben. Mehrfachvererbung gibt es z. B. in der Sprache C++ und führt dort schnell zu komplexen, unübersichtlichen Vererbungsstrukturen.

▷ In Java wird die Mehrfachvererbung durch Interfaces implementiert.

8.5.1 Was wird vererbt?

Anhand des obigen Beispiels wird klar, dass **Attribute** vererbt werden: Hühner haben die gleichen Attribute wie Vögel. Ebenfalls ist ersichtlich, dass **Methoden** vererbt werden. Hühner und Enten können flattern. Generell gilt, dass Attribute und Methoden vererbt werden, sofern diese nicht mit dem Schlüsselwort `private` bezeichnet sind. Dies gilt auch für statische Attribute und Methoden.

Nicht vererbt werden Konstruktoren. Das ist auch leicht einzusehen. Ein **Konstruktor** dient dazu, eine Klasse zu erstellen. Daher benötigt jede Klasse ihren spezifischen Konstruktor, entweder den von Java standardmässig gelieferten Defaultkonstruktor oder einen selbst geschriebenen Konstruktor.

8.5.2 Methoden und Attribute überladen

Sie können in einer Unterklasse eine Methode schreiben, die die gleiche Signatur hat wie eine geerbte Methode der Oberklasse. Ein Beispiel ist die Methode `fliegen()` bei den Hühnern. Die fliegen()-Methoden beim Vogel und beim Huhn haben dieselbe Signatur. In diesem Fall «übersteuert» die Methode in Huhn diejenige in Vogel. Man spricht in diesem Zusammenhang auch vom **Überladen**[1] **einer Methode.**

Dasselbe kann mit Attributen gemacht werden. Sie können in der Klasse **Huhn** ein Attribut `farbe` mit dem Datentyp `String` definieren. Dies ist allerdings nicht zu empfehlen, da Sie möglicherweise Probleme bekommen. Wenn Sie beispielsweise das Huhn Jakob als Huhn ansprechen und nach seiner Farbe fragen, erhalten Sie den Wert, der im Attribut `farbe` in der Klasse **Huhn** gespeichert ist. Sie können das Objekt **Jakob** aber auch einfach als Vogel ansprechen, weil jedes Huhn auch ein Vogel ist. In diesem Fall bekommen Sie den entsprechenden Wert aus dem Attribut der Klasse **Vogel.**

Vergleichen Sie dazu das folgende Beispiel:

```
Vogel jakob = new Huhn();  // ein Huhn wird als Vogel instantiiert
jakob.farbe = "rot";
```

Obwohl hier ein Huhn erstellt wird, wird dieses «nur» als Vogel angesprochen. Dadurch wird die farbe-Variable der Klasse **Vogel** gesetzt und nicht diejenige der Klasse **Huhn.**

8.5.3 Auf Methoden und Attribute der Oberklasse zugreifen

Mit dem Schlüsselwort `super` können Sie in einer Klasse auf **Attribute und Methoden der direkten Oberklasse** zugreifen. Dies ist in folgenden Fällen notwendig:

[1] Engl. Fachbegriff: Overriding.

Zugriff auf überschriebene Methode

Oft gibt es in einer Klasse eine Methode, die zwar fast alles macht, was in der Unterklasse benötigt wird, aber eben nur fast. In diesem Fall können Sie die Methode in der Unterklasse überschreiben und mit `super` auf die Methode der Oberklasse zugreifen. In der Klasse **Huhn** kann dies bei der Methode `fliegen()` vielleicht so aussehen:

```
public void fliegen() {
    super.fliegen();
    if (distanz > 10 || hoehe > 4) {
        System.out.println("Huhn kommt nicht weiter und stürzt ab");
    }
}
```

Mit `super.fliegen()` greift die Methode `fliegen()` in der Klasse **Huhn** auf die Methode `fliegen()` in der Klasse **Vogel** zurück. Das Huhn hebt zwar wie ein Vogel ab, stürzt nach zehn Metern Flugdistanz oder bei einer Flughöhe von über vier Metern aber ab.

Zugriff auf Konstruktor der Oberklasse

In Kapitel 8.4.1, S. 81 haben Sie die Anweisung `super();` kennengelernt. Konstruktoren werden nicht vererbt. Trotzdem müssen auch die Oberklassen korrekt konstruiert werden. Mit `super();` können Sie den **Defaultkonstruktor** der Oberklasse aufrufen. Wenn die Oberklasse bereits einen Defaultkonstruktor hat, übernimmt Java diese Aufgabe und Sie können auf den Aufruf von `super()` verzichten.

Mit `super()` können Sie auch **Parameter übergeben.** In der Folge wird der Konstruktor mit den passenden Werten aufgerufen. Dazu folgendes Beispiel:

```
public class Oberklasse {

    private int einWert;

    // Konstruktor mit einem Parameter
    public Oberklasse(int wert) {
        this.einWert = wert;
    }
}

public class Unterklasse extends Oberklasse {

    String einText;

    public Unterklasse (int zahl, String wort) {
        super(zahl);
        this.einText = wort;
    }
}
```

Hier hat die Oberklasse eine Instanzvariable, die direkt im Konstruktor gesetzt wird. Die Unterklasse hat einen Konstruktor, der eine Zahl und ein Wort als Parameter verwendet. Während das Wort in eine Instanzvariable der Unterklasse abgelegt wird, wird die Zahl an den Konstruktor der Oberklasse weitergegeben. Mit der Anweisung `super(zahl)` wird der Konstruktor der Oberklasse mit einem Parameter aufgerufen. In der Oberklasse kann der Aufruf von `super()` entfallen, da die Oberklasse direkt von der Klasse **Object** abgeleitet wird und Java automatisch den Defaultkonstruktor aufruft.

8.5.4 Typumwandlung

Weil aufgrund der Vererbung ein Huhn auch ein Vogel ist, kann in Java jedes Huhn-Objekt auch als Vogel-Objekt angesprochen werden. Folgende Beispiele für Typumwandlungen basieren auf den deklarierten Klassen **Vogel, Huhn** und **Ente:**

```
Huhn jakob = new Huhn();              // ein Huhn ist ein Huhn

Vogel einVogel = jakob;               // korrekt, Jakob ist ein Vogel
Vogel nochEinVogel = (Vogel)jakob;    // ausdrückliche Typumwandlung
                                      // korrekt, weil Jakob ein Vogel ist

Ente ede = new Vogel();               // falsch, ein Vogel ist keine Ente

Vogel nochEinVogel = new Vogel();     // korrekt
Ente ede = (Ente)nochEinVogel;        // falsch, ein Vogel ist keine Ente
                                      // das kann auch nicht erzwungen werden

einVogel.fliegen();                   // korrekt, jeder Vogel kann fliegen
jakob.fliegen();                      // korrekt, Jakob ist ein Huhn und Hühner
                                      // können fliegen

Vogel ede = new Ente();               // ede ist eine Ente, und Enten sind Vögel
ede.schwimmen();                      // falsch, ede wird als Vogel angesprochen,
                                      // Vögel können nicht schwimmen
((Ente)ede).schwimmen();              // korrekt, ede wird als Ente angesprochen,
                                      // Enten können schwimmen
```

Um eine **Typumwandlung** zu erreichen, muss der gewünschte Typ in runden Klammern vor die Variable geschrieben werden.

8.5.5 Polymorphismus

Im folgenden Codeabschnitt wird **Polymorphismus** angewendet. Können Sie ihn finden und beschreiben?

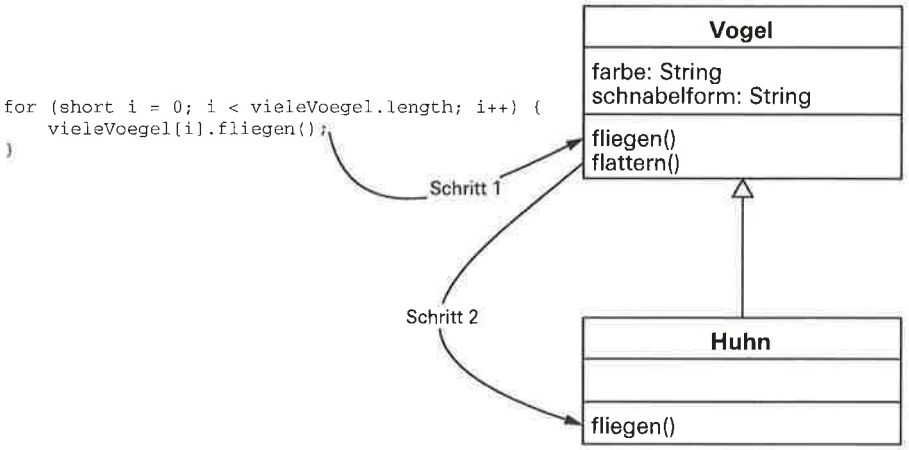
```
// wir erstellen ein Array für Vögel
Vogel[] vieleVoegel = new Vogel[3];

// Wir füllen das Array mit verschiedenen Vögeln
vieleVoegel[0] = new Vogel();
vieleVoegel[1] = new Ente();
vieleVoegel[2] = new Huhn();

// wir lassen jeden Vogel fliegen
for (short i = 0; i < vieleVoegel.length; i++) {
    vieleVoegel[i].fliegen();
}
```

Vögel können fliegen. Daher funktioniert der Aufruf innerhalb der for-Schleife. Wenn Sie die Vögel im Array nun aber nicht fliegen, sondern schwimmen lassen möchten? In diesem Fall hilft Polymorphismus nicht weiter, denn Vogel-Objekte können nicht schwimmen. Enten sind zwar Vögel, die schwimmen können. Da sie hier aber als Vogel angesprochen werden, können sie auch nicht schwimmen, weil eben Vögel nicht schwimmen können.

Wie findet Java also die richtige Methode? Java sucht zuerst nach der Methode in der Klasse, die bei der Referenz auf das Objekt verwendet wird. Im Beispiel mit dem Fliegen sucht Java in der Klasse **Vogel** nach der Methode `fliegen()`. Nun fragt Java das Objekt, was es denn wirklich ist, und das Objekt antwortet «Ich bin ein Huhn». Java schaut nun in der Klasse **Huhn** nach, ob eine «bessere» fliegen()-Methode vorliegt. Dies ist hier der Fall. Java ruft daher die fliegen()-Methode der Klasse **Huhn** auf.

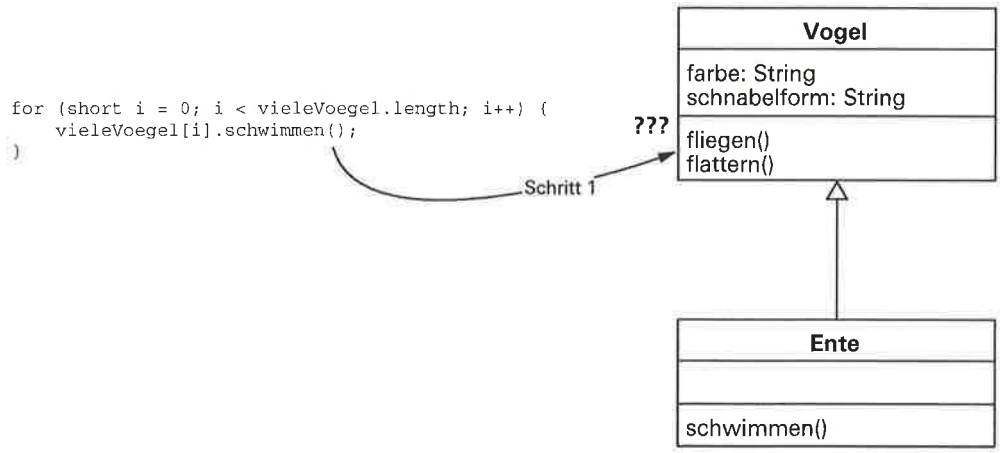
Dieser Vorgang lässt sich wie folgt verdeutlichen:

[8-2] Java ruft die passende Methode auf



Wenn Sie dasselbe mit der Methode `schwimmen()` machen, passiert Folgendes: Java sucht in der Klasse **Vogel** nach einer schwimmen()-Methode. Dort aber gibt es diese Methode nicht. Java entscheidet daher, dass Vögel nicht schwimmen können, und bricht mit einer Fehlermeldung ab. Java sucht also nicht in Unterklassen weiter, ob sie eventuell doch schwimmen könnten. Folgende Grafik veranschaulicht diesen Sachverhalt:

[8-3] Java findet keine passende Methode



Wie können Sie Vögel trotzdem schwimmen lassen? Zumindest diejenigen Vogel-Objekte, die schwimmen können? Der Operator `instanceof` erlaubt es, ein Objekt zu fragen, ob es von einem bestimmten Typ ist. Vergleichen Sie dazu das folgende Beispiel:

```
// wir lassen jeden Vogel schwimmen
for (short i = 0; i < vieleVoegel.length; i++) {
    if (vieleVoegel[i] instanceof Ente) {
        ((Ente)vieleVoegel[i]).schwimmen();
    }
}
```

Hier wird der **instanceof-Operator** verwendet, um zu testen, ob es sich beim aktuellen Vogel um eine Ente handelt. Zudem muss der Vogel ausdrücklich als Ente angesprochen werden, damit Java die schwimmen()-Methode auch wirklich findet.

Hinweis

▷ Denken Sie daran: Vögel können nicht schwimmen, auch wenn der Vogel zufälligerweise eine Ente ist. Nur «richtige» Enten können schwimmen.

8.5.6 Die Klasse Object

In Java erbt jede Klasse von einer anderen Klasse. Erinnern Sie sich noch?

```
public class HalloWelt {
    ...
}
```

Dies funktioniert auch ohne explizite Vererbung, denn am Ende der Vererbungshierarchie erbt jede Klasse von der Klasse **Object.** Diese Klasse wird von Java in der Hierarchie immer zuoberst eingesetzt und muss nicht explizit angegeben werden. Sie hätten aber auch Folgendes schreiben können:

```
public class HalloWelt extends Object {
    ...
}
```

Die Klasse **Object** umfasst Methoden, die an alle Objekte jeder Java-Anwendung vererbt werden. In der folgenden Tabelle[1] werden die wichtigsten Methoden kurz beschrieben, die automatisch vererbt werden:

Methode	Funktionsbeschreibung
`boolean equals(Object anderesObj)`	Vergleicht das aktuelle Objekt mit einem anderen Objekt. Defaultmässig sind Objekte gleich, wenn ihre Referenzen identisch sind. Diese Methode ist z. B. bei der Klasse **String** überschrieben, sodass Gleichheit als Übereinstimmung der Zeichenketten gedeutet wird.
`Class getClass()`	Jedes Objekt kann eine Aussage über seine Klasse machen. Diese Methode liefert ein Objekt vom Typ `Class`.
`String toString()`	Mit dieser Methode kann ein Objekt eine textliche Darstellung von sich selbst machen. Die Defaultausgabe lautet: `<Klassenname>@<Adresse im Speicher>`. Überschreiben Sie diese Methode, um für Ihre Objekte aussagekräftige Anzeigen zu bekommen.

8.6 Zugriffsmodifikatoren implementieren

Zugriffsmodifikatoren regeln, wer welche Zugriffsrechte auf eine Variable oder eine Methode hat. Dabei wird unterschieden, woher der Zugriff erfolgt. Denkbar sind beispielsweise folgende Möglichkeiten:

- Aus der gleichen Klasse
- Aus einer anderen Klasse im selben Paket
- Aus einer abgeleiteten Klasse

In Kapitel 8.3, S. 80 haben Sie bereits die Zugriffsmodifikatoren `private` und `public` im Zusammenhang mit der Vererbung kennengelernt. **Attribute** und **Methoden,** die mit dem Schlüsselwort `private` gekennzeichnet sind, können nicht vererbt werden. Hier ist der Zugriff nicht einmal aus einer abgeleiteten Klasse, sondern nur aus der eigenen Klasse möglich. Daneben gibt es mit `protected` und `default` zwei weitere Zugriffsmodifikatoren in Java. Das erste Schlüsselwort erlaubt den Zugriff bis auf Stufe Unterklasse, das zweite wird von Java automatisch verwendet, wenn kein Schlüsselwort angegeben wird.

[1] Diese Liste ist nicht vollständig. Sehen Sie bitte in der Java-Dokumentation nach, was die Klasse **Object** noch alles zu bieten hat.

Die folgende Tabelle stellt die **Zugriffsrechte** zusammen:

Schlüsselwort	Klasse	Paket	Unterklasse	Alle andern
`public`	Ja	Ja	Ja	Ja
`protected`	Ja	Ja	Ja	Nein
`(keine Angabe)`	Ja	Ja	Nein	Nein
`private`	Ja	Nein	Nein	Nein

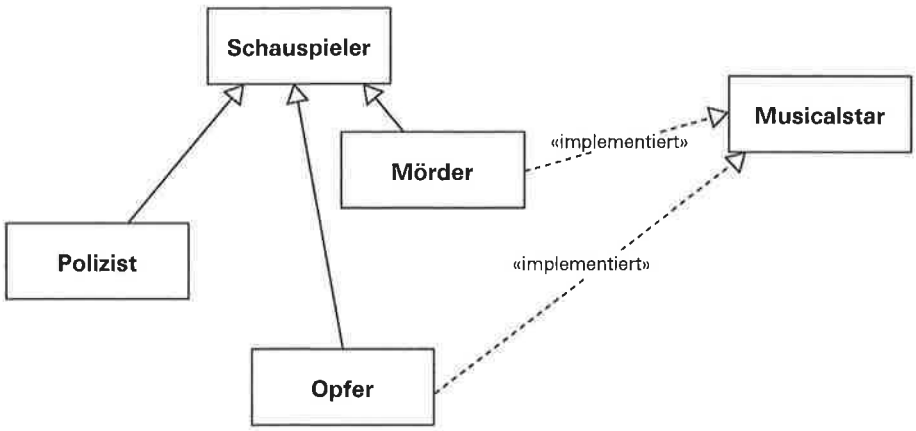
Die vier Spalten in der Tabelle sagen Folgendes aus:

- Die Spalte «Klasse» zeigt auf, ob eine Klasse Zugriff hat. Dies ist immer so, d. h., jede Klasse hat Zugriff auf ihre eigenen Attribute und Methoden.
- Die Spalte «Paket» gibt an, ob eine abgeleitete oder eine nichtabgeleitete Klasse im gleichen Paket Zugriff hat.
- Die Spalte «Unterklasse» zeigt auf, ob eine abgeleitete Klasse in einem anderen Paket Zugriff hat.
- Die Spalte «Alle anderen» gibt an, ob eine beliebige andere Klasse in einem beliebigen anderen Paket Zugriff hat.

8.7 Interfaces und abstrakte Methoden implementieren

Wie Sie in Kapitel 3.5, S. 42 erfahren haben, können Sie mithilfe von Interfaces in Java die fehlende Mehrfachvererbung realisieren. Ein **Interface** kann als Versprechen aufgefasst werden, bestimmte Methoden zu implementieren, und sieht wie eine Mehrfachvererbung aus, ist aber keine:

[8-4] Interface Musicalstar



Ein Interface sagt aus, dass eine Methode bzw. ein Verhalten vorhanden ist, gibt aber nicht an, wie diese realisiert wird. Entsprechend enthalten Interfaces auch keinen Code. Nachfolgend sehen Sie beispielhaft die **Deklaration des Interface Musicalstar:**

```
package ch.modul226.krimi;

public interface Musicalstar {

    public void tanzen();
    public void singen();
}
```

Sie sehen: Ein Interface sieht aus wie eine Klasse, verwendet jedoch das Schlüsselwort `interface`. Wenn Sie sich die zugehörigen Methoden anschauen, fällt auf, dass es nur

Methodenköpfe sind. Keine der Methoden hat einen **Methodenrumpf.** Nicht einmal geschweifte Klammern sind vorhanden.

Wenn Sie nun die Rolle des Mörders zum Musicalstar ausbauen möchten, müssen Sie die Deklaration der Klasse **Moerder** wie folgt ergänzen:

```
package ch.modul226.krimi;

public class Moerder extends Schauspieler implements Musicalstar {

    public void tanzen() {
        System.out.println("Mörder: Und sie tanzen einen Tango...");
    }

    public void singen() {
        System.out.println("Mörder: Hinweis. Ich singe nun.");
    }

    public void zustechen() {
        System.out.println("Mörder: sticht heftig zu");
    }

    public void sadistischGrinsen() {
        System.out.println("Mörder: Ha. Ha. Ha. Ha.");
    }

    public void wegrennnen() {
        System.out.println("Mörder: rennt weg");
    }

}
```

Das Schlüsselwort `implements` verspricht, dass die Klasse **Moerder** den Programmcode für die Methoden `singen()` und `tanzen()` liefert. Eine Klasse kann mehr als ein Interface implementieren. Damit ist eine hohe Flexibilität möglich. Interessant ist, dass alle Mörderobjekte nicht nur Schauspieler sind, sondern auch Musicalstars. Sie können also einen Mörder auch als Musicalstar ansprechen:

```
Moerder fredTheKiller = new Moerder();
Musicalstar taenzer = (Musicalstar)fredTheKiller;
```

Beachten Sie im Zusammenhang mit Interfaces folgende **Regeln:**

- Interfaces sind immer abstrakt und publik, auch wenn diese Schlüsselwörter fehlen.
- Ein Interface kann mehrere andere Interfaces erweitern.
- Eine nicht abstrakte Klasse, die ein Interface implementiert, muss alle Methoden des Interfaces implementieren.
- Interfaces sparen keine Tipparbeit.
- Interfaces enthalten keinen Code.[1]

[1] Sie können auch Klassen erstellen, die keinen oder nur teilweise Code enthalten.

8.8 Abstrakte Klassen implementieren

Hühner und Enten sind Vögel, aber es macht eigentlich keinen Sinn, im Programmcode `new Vogel()` aufzurufen. Dies können Sie verhindern, indem Sie bei der Deklaration der Klasse **Vogel** das Schlüsselwort `abstract` wie folgt verwenden:

```
public abstract class Vogel {
    ...
}
```

Java verhindert nun, dass Vogel-Objekte erstellt werden können. Das ändert aber nichts daran, dass Hühner und Enten Vögel sind und als solche angesprochen werden können.

Manchmal möchte man in einer Oberklasse eine Methode haben, die in allen Unterklassen überschrieben werden muss. Dazu ein Beispiel: Die Klasse **Vogel** besitzt die Methode `fliegen()`. Wenn nun aber jede Vogelart unterschiedlich fliegt, macht es keinen Sinn, die Methode `fliegen()` in der Klasse **Vogel** vollständig zu definieren. Trotzdem sollen alle Vögel die Methode `fliegen()` haben. In diesem Fall kennzeichnen Sie die Methode `fliegen()` in der Klasse **Vogel** als `abstract` und geben ihr keinen Methodenrumpf:

```
public abstract class Vogel {

    public abstract void fliegen();
    ...
}
```

Jede Klasse, die von **Vogel** erbt, muss die Methode `fliegen()` implementieren. Wenn sie dies nicht tut, ist sie automatisch abstrakt und davon abgeleitete Klassen müssen den Code zu `fliegen()` liefern. Entsprechend ist die Klasse **Vogel** nicht vollständig definiert.

Hinweis

▷ Auch wenn eine Klasse automatisch abstrakt wird, muss sie mit dem Schlüsselwort `abstract` versehen werden, da der Compiler ansonsten einen Fehler meldet. Auf diese Weise stellt Java sicher, dass sich der Entwickler darüber im Klaren ist, dass die Klasse noch nicht vollständig ist.

8.9 Exception-Handling implementieren

In Kapitel 3.4, S. 40 haben Sie das Konzept der **Fehlerbehandlung** mithilfe von Exceptions kennengelernt. Hier erfahren Sie, wie der Umgang mit Exceptions in Java realisiert wird.

8.9.1 Programmablauf und Fehlerbehandlung trennen

Kann ein Methodenaufruf zu einer Exception führen, verwenden Sie einen sogenannten **try-catch-Block** nach folgendem Muster:

```
try {
    ergebnis = funktionA();
    zahl = funktionB(ergebnis);
    // und viele weitere Anweisungen
}
catch (Exception ex) {
    // hier Fehler behandeln
}
```

Sie haben erfahren, dass Sie mit dieser Konstruktion den normalen Programmablauf von der Fehlerbehandlung trennen können. Im **try-Block** stehen alle Anweisungen ohne spezielle Fehlerbehandlungen. Der **catch-Block** gehört zum vorangehenden try-Block, sieht ähnlich aus wie eine Methodendeklaration und wird mit einem Parameter aufgerufen, der im obigen Beispiel vom Typ `Exception` ist und den Namen `ex` trägt. Exception ist eine Klasse, die von Java zur Verfügung gestellt wird und sich im Paket **java.lang** befindet. Die wichtigsten beiden Methoden der Klasse **Exception** sehen wie folgt aus:

Methode	Beschreibung
`String getMessage()`	Liefert einen Klartext der Fehlermeldung.
`void printStackTrace()`	Liefert eine detaillierte Liste aller Methodenaufrufe, die zu diesem Problem geführt haben, inklusive Zeilennummern und Klassen- und Methodendetails dazu.

Java kennt und stellt viele Exceptions zur Verfügung. Sie können bei Bedarf aber auch **eigene Exceptions** definieren. Hier einige Exceptions aus **java.lang:**

Exception	Beschreibung
`ClassNotFoundException`	Eine Klasse kann von der Java-Laufzeitumgebung nicht gefunden werden.
`IOException`	Es ist ein Fehler bei einer Ein- oder Ausgabe aufgetreten.
`RuntimeException`	Während des Programmablaufs ist ein allgemeiner Fehler aufgetreten. Davon gibt es viele abgeleitete Exceptions.

RuntimeException	Beschreibung
`ArithmeticException`	Es ist ein Berechnungsfehler aufgetreten (z. B. Division durch null).
`ClassCastException`	Es wurde eine ungültige bzw. unerlaubte Klassentypumwandlung versucht. Beispiel: Ein Objekt vom Typ Huhn wird als Ente angesprochen. Dadurch wird eine «Ist-ein-Beziehung» verletzt.
`NullPointerException`	Es wurde auf eine Objektreferenz zugegriffen, deren Wert null ist.
`ArrayIndexOutOfBoundException`	Bei einem Array wurde auf ein Element ausserhalb des Arrays zugegriffen. Beispiel: Sie greifen bei einem Array mit 9 Elementen auf das 10. Element zu.

8.9.2 Fehlerbehandlung programmieren

Sie können alle Exceptions mittels catch-Block abfangen und darin eine entsprechende **Reaktion bzw. Fehlerbehandlung** programmieren. Vergleichen Sie dazu den folgenden Beispielcode:

```
int zahl1 = ConsoleReader.reatInteger("Bitte erste Zahl eingeben");
int zahl2 = ConsoleReader.reatInteger("Bitte zweite Zahl eingeben");
try {
    int ergebnis = zahl1 / zahl2;
    System.out.println("Das Ergebnis lautet " + ergebnis);
    System.out.println("Besten Dank für den Rechenauftrag");
}
catch (ArithmeticException ex) {
    System.out.println("Es ist ein Fehler bei der Berechnung aufgetreten");
    System.out.println(ex.getMessage());
}
```

Hier verlangt das Programm vom Benutzer zwei Zahlen und versucht danach, die erste Zahl durch die zweite Zahl zu dividieren. Wenn die Division funktioniert, gibt das Programm das Ergebnis aus und bedankt sich beim Benutzer. Wenn bei der Division Fehler auftreten (z. B. bei einer Division durch null), wird die weitere Verarbeitung des try-Blocks abgebrochen und der catch-Block ausgeführt. In der Fehlermeldung gibt das Programm danach den **Klartext der Exception** aus.

Wenn **mehrere Exceptions** auftreten können, auf die Sie unterschiedlich reagieren möchten, notieren Sie einfach mehrere catch-Blöcke hintereinander.

```
try {
    // was auch immer, hier gibt
    // es möglicherweise Exceptions
}
catch (ArrayIndexOutOfBoundsException e1) {
    // wird ausgeführt, wenn Arraygrenzen überschritten werden
}
catch (IOException e2) {
    // wird ausgeführt, wenn Ein-Ausgabe-Fehler auftreten
}
```

8.9.3 Exception wird nicht abgefangen

Was passiert, wenn eine Exception auftritt, aber nicht abgefangen wird? In diesem Fall fängt die **Java-Laufzeitumgebung** die Exception ab. Wenn es sich dabei um eine RuntimeException handelt, ruft die Laufzeitumgebung von Java die Methode `getMessage()` der RuntimeException auf und beendet das Programm. Wenn es sich um eine Exception handelt, erwartet Java, dass sich die Anwendung darum kümmert, und ignoriert eine solche Exception, wenn sie nicht abgefangen wird.

Im Beispiel in Kapitel 8.9.2, S. 92 wird ausdrücklich die Ausnahme `ArithmeticException` abgefangen. Stattdessen können Sie eine RuntimeException oder eine Exception wie folgt abfangen:
`ArithmeticException` ist eine `RuntimeException` ist eine `Exception`

Sie können dies ausprobieren, indem Sie ein Java-Programm starten und eine ungültige Klasse[1] angeben. Hier ein Beispiel dazu:

```
macosx> java GibtEsNicht
Exception in thread "main" java.lang.NoClassDefFoundError: GibtEsNicht
```

Und hier ein Beispiel für eine nicht abgefangene Division durch null:

```
macosx> cat Division.java
public class Division {
    public static void main(String[] args) {
        int ergebnis = 17 / 0;
    }
}
macosx> javac Division.java
macosx> java Division
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Division.main(Division.java:3)
```

Hier ist in der Methode `main()` der Klasse **Division** in der Datei **Division.java** eine Division durch null aufgetreten (`/ by zero`).

8.9.4 Exception selber auslösen

Bisher haben Sie gesehen, wie eine Exception abgefangen und ausgewertet werden kann. Sie können eine Exception auch selbst auslösen bzw. «werfen»[2]. Schauen Sie sich dazu den folgenden Beispielcode an:

[1] Zum Beispiel eine Klasse, die es nicht gibt oder die keine main()-Methode hat.

[2] Engl.: to throw.

```
try {
    int wert = ConsoleReader.readInteger("Zahl zwischen 0 und 100 eingeben");
    if (wert < 0 || wert > 100) {
        throw new Exception("Ungueltiger Wert");
    }
}
catch(Exception ex) {
    System.out.println(ex.getMessage());
}
```

Hier werfen Sie eine neue Exception und geben dem Konstruktor als Fehlermeldung einen String mit. Die Methode `getMessage()` liefert diesen Text. Im obigen Beispiel wird die Exception in der gleichen Methode «geworfen» und «gefangen». Wenn eine Exception aus einer Methode hinausgeworfen und erst in einer aufrufenden Methode gefangen werden soll, müssen Sie dies im **Methodenkopf** ausdrücklich deklarieren:

```
public int eingabe() throws Exception {
    int wert = ConsoleReader.readInteger("Zahl zwischen 0 und 100 eingeben");
    if (wert < 0 || wert > 100) {
        throw new Exception("Ungueltiger Wert");
    }
    return wert;
}
```

Hinweis

▷ Wird eine Exception von java.lang.RuntimeException abgeleitet, müssen Sie diese nicht angeben, da **RuntimeExceptions** auch von der Java-Laufzeitumgebung geworfen werden können. In diesem Fall gibt es keinen Methodenkopf, in dem Sie die Exception aufführen können.

8.9.5 Eigene Exceptions definieren

In vielen Fällen reicht es, eine Standardexception zu werfen und bei der Konstruktion einen Fehlertext mitzugeben. Manchmal empfiehlt es sich aber auch, eine eigene **Exception-Klasse** zu definieren. Dies soll anhand eines Beispiels verdeutlicht werden. Schauen Sie sich dazu folgenden Codeabschnitt an:

```
public class UngueltigerWertException extends Exception {
    public UngueltigerWertException(String meldung) {
        super(meldung);
    }
}
```

Hier wurde eine separate Exception-Klasse deklariert. Diese Klasse können Sie einsetzen, indem Sie zunächst ein Objekt der eigenen Exception «werfen»:

```
public int eingabe() throws UngueltigerWertException {
    int wert = ConsoleReader.readInteger("Zahl zwischen 0 und 100 eingeben");
    if (wert < 0 || wert > 100) {
        throw new UngueltigerWertException("Ungueltiger Wert");
    }
    return wert;
}
```

... und danach die Klasse **UngueltigerWertException** «fangen»:

```
try {
    int wert = eingabe();
}
catch(UngueltigerWertException ex) {
    System.out.println(ex.getMessage());
}
```

Da es sich bei Exception-Klassen um reguläre Klassen handelt, können Sie diese mit beliebigen **Funktionen** ausstatten.

Mehrere Exceptions stehen untereinander oft in einer **«Ist-ein-Beziehung»**. Es gilt z. B. `ArrayIndexOutOfBoundsException` ist eine `IndexOutOfBoundsException` ist eine `RuntimeException` ist eine `Exception`. Exceptions müssen daher im catch-Block von spezifisch zu generell abgefangen werden. Wichtig ist dabei der throws-Teil in einem Methodenkopf:

```
double division() throws Exception {
    ...
}
```

Wenn innerhalb der division()-Methode eine Division durch null versucht wird, wirft Java eine **ArithmeticException.** Diese wird allerdings im obigen Beispiel nur an die aufrufende Methode weitergeleitet und kann von dieser im catch-Block nicht abgefangen werden.

Sie haben in diesem Kapitel gelernt, wie man **Klassen** definiert. Jede Klasse hat einen Namen, der mit einem Grossbuchstaben beginnen sollte. Innerhalb der Klassendefinition können Sie beliebige Methoden erstellen. In jeder Klasse kann es sogenannte Instanzvariablen geben. Dabei handelt es sich um Variablen, die in jedem Objekt gleich heissen, aber jedes Objekt hat dafür seinen eigenen Speicherplatz. Mit dem Schlüsselwort `this` können Sie Instanzvariablen des eigenen Objekts ausdrücklich referenzieren.

In jeder Klasse kann es auch **Variablen** geben, die unabhängig von Objekten existieren. Diese Variablen werden durch das Schlüsselwort `static` markiert. Für eine Klassenvariable gibt es genau einen Speicherplatz. Klassenvariablen sprechen Sie an, indem Sie den Klassennamen vor den Variablennamen setzen. Genauso, wie es Klassenvariablen gibt, gibt es auch Klassenmethoden. Diese werden ebenfalls durch das Schlüsselwort `static` markiert und müssen daher auch wie die Klassenvariablen mit dem Namen der Klasse angesprochen werden.

Der **Konstruktor** ist eine spezielle Methode in jeder Klasse. Sie können den von Java standardmässig gelieferten Defaultkonstruktor verwenden oder einen oder mehrere eigene Konstruktoren erstellen. Konstruktoren sind (fast) normale Methoden, die überladen werden können.

Vermeiden Sie es, direkt auf Instanzvariablen von Objekten zuzugreifen. Erstellen Sie dafür jeweils eine set()- und eine get()-Methode. Dies nennt man **Datenkapselung.** Als Benutzer von Objekten müssen Sie sich so nicht darum kümmern, wie und wo deren Daten intern abgelegt sind. Als Programmierer sind Sie nun frei, den internen Aufbau von Klassen beliebig zu wählen und auch zu ändern, ohne dass das Konsequenzen für andere Teile der Applikation hat.

Das Schlüsselwort `extends` wird benutzt, um **Vererbung** zu implementieren. Mit `super` erhalten Sie eine Referenz auf die Oberklasse und mit `super()` können Sie einen Konstruktor der Oberklasse aufrufen. Hierbei sind auch Übergabeparameter möglich.

Interfaces sind eine elegante Möglichkeit, die in Java nicht erlaubte Mehrfachvererbung zu umgehen. Klassen versprechen dabei, bestimmte Methoden zu implementieren. Eine Klasse ist nicht verpflichtet, für jede Methode einen Methodenrumpf mit Code zu liefern. Eine solche unvollständige Klasse nennt man abstrakt.

Von **abstrakten Klassen** können keine Objekte erzeugt werden. Klassen, die von abstrakten Klassen abgeleitet werden, müssen die fehlenden Codeteile liefern (Implementierung der Methodenrümpfe). In einer Vererbungshierarchie können beliebig viele Oberklassen abstrakt sein. Nur die letzte Unterklasse muss alle Methodenrümpfe implementieren. Eine Klasse kann, auch wenn sie vollständig ist, als abstrakt deklariert werden. Damit verhindert man, dass Objekte erzeugt werden, die sinnlos sind.

Repetitionsfragen

44 Beurteilen Sie, ob die unten aufgeführten Klassennamen gültig, schlecht gewählt oder ungültig sind, und begründen Sie jedes negative Urteil.

Klassenname	gültig	schlecht gewählt	ungültig
Grafik			
Mühle			
Diesisteinlangername			
Waerme Pumpe			
radioAufnahme			

45 Beschreiben Sie in ein bis zwei Sätzen, was eine Instanzvariable ist.

46 Was bedeutet das Schlüsselwort `this`?

47 Was bedeutet `static` in der Variablendeklaration `private static int anzahl = 0;`?

48 Was ist der Defaultkonstruktor einer Klasse?

49 Wofür brauchen Sie die Methode `super()`?

50 Welches Symbol verwenden Sie in einem Klassendiagramm zur Darstellung einer «Ist-ein-Beziehung» in Java? Machen Sie ein Beispiel.

51 Wie implementieren Sie eine «Ist-ein-Beziehung» in Java? Machen Sie ein Beispiel.

52 Ergänzen Sie die Zugriffsrechte in der folgenden Tabelle:

Schlüsselwort	Klasse	Paket	Unterklasse	Alle andern
`public`				
`protected`				
`(default/keine Angabe)`				
`private`				

53 Wie greifen Sie auf eine überschriebene Methode in der Oberklasse zu?

4 Gehen Sie von den Klassen Vogel, Huhn, Ente aus. Welche Codezeilen sind korrekt? Begründen Sie Ihre Antwort.

```
Huhn jakob = new Huhn();
Vogel einVogel = jakob;
Vogel nochEinVogel = (Vogel)jakob;
Ente ede = new Vogel();
Vogel nochEinVogel = new Vogel();
Ente ede = (Ente)nochEinVogel;
einVogel.fliegen();
jakob.fliegen();
Vogel ede = new Ente();
ede.schwimmen();
((Ente)ede).schwimmen();
```

55 Wie können Sie zur Laufzeit eines Programms prüfen, ob ein Objekt von einer bestimmten Klasse ist?

56 Hat jede Klasse eine Oberklasse?

57 Wie definieren Sie ein Interface?

58 Wie implementieren Sie ein Interface?

59 Was bedeutet das Wort abstract bei `public abstract class Vogel {...` und welche Konsequenzen hat es?

60 Wie definieren Sie eine abstrakte Methode? Machen Sie ein Beispiel.

61 Wie lösen Sie eine Exception aus?

62 Wie fangen Sie eine Exception?

63 Erläutern Sie die Aussage «If you want a new house, you call the constructor».

64 Exceptions sind Objekte. Nennen Sie die zwei wichtigen Methoden von Exceptions.

65 Was bedeutet es, wenn bei einem Methodenkopf `throws xyz` steht?

66 Wozu dient die Klasse **ConsoleReader?**

9 Pakete in Java

Sie haben bereits in Kapitel 1.1, S. 16 gesehen, dass der Java-Sourcecode in sogenannten **Paketen** zusammenfasst wird. In diesem Kapitel erfahren Sie, wie Pakete organisiert und angesprochen werden. Ausserdem lernen Sie wichtige Java-Pakete näher kennen.

9.1 Was sind Pakete und wie werden sie benannt?

Sie können sich Pakete wie **die Verzeichnis- und Ordnerstruktur in einem Dateisystem** vorstellen. Pakete können sich innerhalb von anderen Paketen befinden und in jedem Paket können Klassen definiert werden. Dabei gilt: Wenn sich die Klassen in verschiedenen Paketen befinden, können Sie den **gleichen Klassennamen** mehrmals verwenden – genauso, wie Sie für verschiedene Dateien denselben Dateinamen verwenden können, solange sich die Dateien in unterschiedlichen Verzeichnissen bzw. Ordnern auf einer Festplatte befinden.

Paketnamen sollten in Kleinbuchstaben geschrieben werden, können aber im Grunde genommen frei gewählt werden. Wenn Sie jedoch Pakete oder Klassen unterschiedlicher Projekte mit anderen Entwicklern austauschen, besteht die Gefahr, dass diese gleichen Namen haben. Dies kann zu Problemen führen. Daher empfiehlt es sich, den Paketnamen mit dem Domainnamen in umgekehrter Schreibweise zu beginnen und die einzelnen Unterpakete jeweils durch einen Punkt voneinander zu trennen.

Beispiel:

Für das Krimiprojekt wurde die Domain **modul226.ch** verwendet. Entsprechend werden die Pakete dieses Projekts **ch.modul226.krimi** genannt. Es handelt sich also um das Paket **krimi** innerhalb des Pakets **modul226** innerhalb des Pakets **ch**.

9.2 Klassen in Paketen ablegen und ansprechen

Wenn Sie eine Klasse in ein Paket legen möchten, müssen Sie als erste Zeile (noch vor der Klassendefinition) eine **Paketangabe** machen. In unserem Krimi finden Sie daher in allen Klassen folgende Zeile:

```
package ch.modul226.krimi;
```

Das Schlüsselwort `package` weist darauf hin, in welchem Paket sich die Klasse befindet. Genauso, wie eine Datei **Zeuge.java** heissen kann, aber erst mit der vollständigen Pfadangabe eindeutig ist[1], heisst auch die Klasse **Zeuge** eigentlich **ch.modul226.krimi.Zeuge.** Wenn Sie sich im Dateisystem im richtigen Ordner befinden, reicht die Angabe des Dateinamens, um die betreffende Datei aufzurufen. Bei den Klassen in Paketen verhält es sich ebenso. Java sucht eine Klasse ohne Paketangabe automatisch im gleichen Paket.

Da sich die Krimiklassen alle im gleichen Paket befinden, reicht die Angabe des Klassennamens, um alle Klassen anzusprechen. In der Klasse **Regisseur** finden Sie z. B. folgende Zeile:

```
Moerder bill = new Moerder();
```

[1] Beispiel: **C:\Projekte\Modul226\ch\modul226\krimi\Zeuge.java.**

Eigentlich müsste diese Zeile vollständig ausgeschrieben wie folgt lauten:

```
ch.modul226.krimi.Moerder bill = new ch.modul226.krimi.Moerder();
```

9.2.1 Auf Klassen mittels Paketpfadangabe zugreifen

Wenn Sie eine Klasse in einem anderen Paket ansprechen möchten, müssen Sie den **Paketpfad vor dem Klassennamen** setzen. Nehmen Sie beispielsweise an, Sie haben eine Klasse **ConsoleReader** im Paket **ch.modul226.utils.** Diese Klasse hat nicht direkt mit dem Krimi zu tun, sondern wird nur für die Konsoleneingabe verwendet. Aus diesem Grund ist es sinnvoll, **ConsoleReader** in ein separates Paket zu «legen». Wenn Sie nun in der Klasse **Regisseur** vom Benutzer eine Eingabe verlangen, müssen Sie dies wie folgt tun:

```
int eingabe = ch.modul226.utils.ConsoleReader.readInteger();
```

Sie rufen die Klassenmethode `readInteger()` in der Klasse **ConsoleReader** auf, die sich im Paket **ch.modul226.utils** befindet.

9.2.2 Auf Klassen durch import-Angabe zugreifen

Wenn Sie immer wieder auf Klassen in einem anderen Paket zugreifen müssen, kann eine vollständige Pfadangabe ziemlich aufwendig werden. Zudem ist ein Programmcode mit langen Pfadangaben schwer lesbar. Sie können dies umgehen, indem Sie vor der Klassendefinition eine **import-Anweisung** einfügen. Um z. B. in der Klasse **Regisseur** die Klasse **ConsoleReader** aus dem Paket **utils** bekannt zu machen, schreiben Sie Folgendes:

```
package ch.modul226.krimi;

import ch.modul226.utils.*;

public class Regisseur {
...
    int eingabe = ConsoleReader.readInteger();
...
```

Mit der Angabe `import ch.modul226.utils.*;` weisen Sie Java an, alle (*)-Klassen im Paket **utils** bekannt zu machen. Nun können Sie einfach auf die Klasse **ConsoleReader** zugreifen. Diese Klasse wird zuerst im aktuellen Paket **ch.modul226.krimi** gesucht, dort aber nicht gefunden. Nun sucht Java die Klasse in einem Paketpfad, der durch `import` angegeben worden ist, und findet die Klasse im Paket **ch.modul226.utils.**

9.3 Klassenpfad

Wenn Sie Klassen in Pakete ablegen, erzeugt der Java-Compiler beim Kompilieren eine entsprechende Ordnerstruktur im Dateisystem. Wo aber sind die diversen Klassen der Java-Laufzeitumgebung abgelegt? Wo ist die Struktur für das Paket **java.lang,** in dem sich auch die Klasse **String** befindet? Und wie findet die Java-Laufzeitumgebung die Klassen im Dateisystem? Die Antwort lautet: über den Klassenpfad. Der **Klassenpfad** ist eine Liste von Verzeichnissen, in denen Java nach Klassen sucht. Der Pfad zu den Systemklassen ist in Java «eingebaut», d. h., Java sucht diese Klassen immer an genau vorgegebenen Stellen im Installationsverzeichnis.

Erinnern Sie sich noch an das Programm **HalloWelt.java?** Der **Kompilierbefehl** auf der Kommandozeile lautet dort wie folgt:

```
javac -d . HalloWelt.java
```

Die Option `-d .` sagt dem Compiler, dass er die Ordnerstruktur im aktuellen Verzeich ablegen soll. Wenn Sie ohne diese Option kompilieren, erstellt der Compiler keine Ordne struktur, sondern legt die erzeugte Klassendatei im **Defaultverzeichnis** ab. Vergleichen Sie dazu das Kapitel 9.4, S. 101.

Wenn Sie ein Java-Programm laufen lassen möchten, sucht Java im Klassenpfad nach der angegebenen Klasse. Dabei wird das aktuelle Arbeitsverzeichnis automatisch dem Systemklassenpfad vorangestellt. Sehen Sie, was passiert, wenn Sie Pakete verwenden:

[9-1] Versuche mit dem Klassenpfad

```
Terminal — bash (ttyp2)
modul226> ls
HalloWelt.java
modul226> cat HalloWelt.java
A package ch.modul226;

  public class HalloWelt {

      public static void main(String[] args) {
          System.out.println("Hallo Welt, Java ist alles andere als kalter Kaffee!");
      }
  }
  modul226> javac HalloWelt.java
B modul226> ls
  HalloWelt.class  HalloWelt.java
C modul226> java ch.modul226.Hallo
  Exception in thread "main" java.lang.NoClassDefFoundError: ch/modul226/Hallo
D modul226> javac -d /tmp HalloWelt.java
  modul226> ls /tmp/ch//modul226/
  HalloWelt.class
E modul226> java -classpath /tmp ch.modul226.HalloWelt
  Hallo Welt, Java ist alles andere als kalter Kaffee!
  modul226>
  modul226>
```

A] Package **ch.modul226:** In der Klasse steht eine Paketangabe.

B] Kompilieren ohne `-d .` erzeugt ein Klassenfile im Arbeitsverzeichnis.

C] Aufruf von Java mit dem Klassennamen führt zu einer Fehlermeldung. Die Klasse wird nicht gefunden.

D] Mit `-d /tmp` wird vom Compiler verlangt, dass er die kompilierte Klassendatei in der korrekten Verzeichnisstruktur ablegt. Diese soll im tmp-Verzeichnis liegen.

E] Mit der Option `-classpath /tmp` wird mitgeteilt, wo die eigenen Klassen zu finden sind. Hier soll Java im Verzeichnis **/tmp** suchen.

Hinweise

▷ Der vollständige Name einer Klasse setzt sich aus dem Klassenpfad, dem Paket und dem Klassennamen zusammen.

▷ Passen Paketangabe und Klassenpfad nicht zusammen, kann ein Programm nicht kompiliert werden.

Wenn Sie kein Paket anlegen bzw. angeben, legt Java die erstellten Klassen automatisch in ein sogenanntes **Defaultpaket.** Dieses Paket hat keinen Namen und kann daher auch nicht angesprochen werden. Für einfache Projekte reicht es, wenn Sie auf Pakete verzichten und alle Klassen in das Defaultpaket legen. Allerdings wird seit der Java-Version 5 ausdrücklich davon abgeraten, d. h., in künftigen Java-Versionen wird es das Defaultpaket möglicherweise nicht mehr geben. Am besten gewöhnen Sie sich deshalb daran, immer alle Klassen in Paketen abzulegen.

9.4 Wichtige Java-Pakete

Mit den String- und Wrapper-Klassen haben Sie bereits Klassen des Java-Laufzeitsystems kennengelernt, die sich in Java-Paketen befinden. Insgesamt gibt es mehr als hundert Pakete im Java-Laufzeitsystem mit Tausenden von Klassen. Nachfolgend werden ein paar wichtige **Java-Pakete** näher erläutert.

Hinweis

▷ Die Paketanfänge **java** und **javax** sind reserviert. Sie können keine eigenen Klassen in Paketpfade stellen, die mit einem dieser Namen beginnen.

9.4.1 Das Paket java.lang

Im Paket **java.lang** befinden sich sämtliche Basisklassen von Java. Ohne diese Basisklassen würde Java gar nicht funktionieren, weil z. B. keine Datentypen zur Verfügung stehen. Das Paket **java.lang** wird deshalb vom Java-Compiler als einziges Paket automatisch eingebunden, d. h., Sie müssen keine Anweisung `import java.lang.*;` einfügen. In der folgenden Tabelle finden Sie einige Klassen aus diesem Paket:

Klasse	Beschreibung
String	Zeichenketten
Byte, Boolean, Integer, Double etc.	Die Wrapper-Klassen für die primitiven Datentypen
Runtime	Jede laufende Java-Anwendung hat Zugriff auf eine Instanz der Klasse Runtime. Diese Klasse erlaubt die Interaktion der Applikation mit der Laufzeitumgebung
System	Diese Klasse stellt unter anderem Möglichkeiten zur Ein- und Ausgabe bereit

9.4.2 Das Paket java.math

Das Paket **java.math** enthält wenige Klassen, die speziell für mathematische Aufgaben benötigt werden. In der folgenden Tabelle werden einige Klassen aus diesem Paket aufgeführt:

Klasse	Beschreibung
BigDecimal	Eine Klasse, die es erlaubt, mit Kommazahlen mit beliebig vielen Stellen beliebig genau zu rechnen
BigInteger	Beliebig grosse Ganzzahlen, bei Bedarf mit Tausenden von Stellen

9.4.3 Das Paket java.util

Das Paket **java.util** enthält zahlreiche Klassen, die für alle möglichen Anwendungen nützlich sind. In der folgenden Tabelle finden Sie einige Klassen aus diesem Paket:

Klasse	Beschreibung
Date	Datum und Zeit. Berücksichtigt lokale Spracheinstellungen
Calendar, GregorianCalendar	Kalender mit Monat, Jahr, Schaltjahr etc.
Timezone	Zeitzonen, wird im Zusammenhang mit Zeit und Kalender benötigt
Currency	Währung. Berücksichtigt lokale Einstellungen
Collections: List, ArrayList, Set, Map etc.	Verschiedenste Klassen, um Objekte zusammenzufassen
Enumeration	Hilfsklasse, um durch eine Collection zu iterieren
Random	Zufallszahlengenerator

9.4.4 Das Paket java.io

Das Paket **java.io** enthält Klassen für die Ein- und Ausgabe sowie zum Lesen, Schreiben und Manipulieren von Dateien. In der folgenden Tabelle werden einige Klassen aus diesem Paket aufgeführt:

Klasse	Beschreibung
File	Stellt eine Datei dar
FileReader, FileWriter	Einfache Klassen zum Lesen und Schreiben von Textdateien
InputStream, OutputStream	Basis von vielen anderen Klassen, die einen einfachen Strom von Bytes zum Lesen und Schreiben darstellen

9.4.5 Das Paket javax.swing

Das Paket **javax.swing** enthält viele Unterpakete mit Klassen für die grafische Darstellung auf dem Bildschirm. Die Klassen in javax.swing werden für grafische Benutzerschnittstellen verwendet. Hier einige Klassen aus diesem Paket:

Klasse	Beschreibung
ImageIcon	Dient zur Darstellung von Icons
ButtonGroup	Dient zur Darstellung von Auswahlknöpfen (Single Choice)
JButton	Dient zur Darstellung von Schaltflächen
JMenu, JMenuBar	Dient zur Darstellung von Auswahlmenüs (Single Choice)
JWindow	Dient als Basis zur Darstellung eines Dialogfensters für Benutzereingaben

In Java werden die Klassen in Paketen zusammengefasst. **Pakete** können andere Pakete enthalten. Jede Klasse ist so erst durch Ihren Namen und den kompletten **Paketpfad** vollständig definiert. Java empfiehlt nachdrücklich, dass Sie Ihren Code ebenfalls in Paketen organisieren, auch wenn dies in den aktuellen Versionen von Java noch optional ist.

Das Java-Laufzeitsystem selbst stellt Hunderte von Klassen zur Verfügung, die in den Paketen **java.*** und **javax.*** zusammengefasst sind. Das wichtigste Paket ist **java.lang.** Es enthält die Sprachbasis der Programmiersprache Java.

Repetitionsfragen

67 Wozu brauchen Sie das Schlüsselwort `package` und wie setzen Sie es ein?

68 In einem Quelltext lautet die 1. Zeile: `package ch.meier.kooleapp;`
Was bedeutet das?

69 Wie können Sie in Java auf Klassen in anderen Paketen zugreifen? Zeigen Sie zwei Optionen auf.

70 Welches Standardpaket wird von Java immer automatisch eingebunden und muss nicht importiert werden?

71 Was bedeutet die Zeile `import java.math.Random;`?

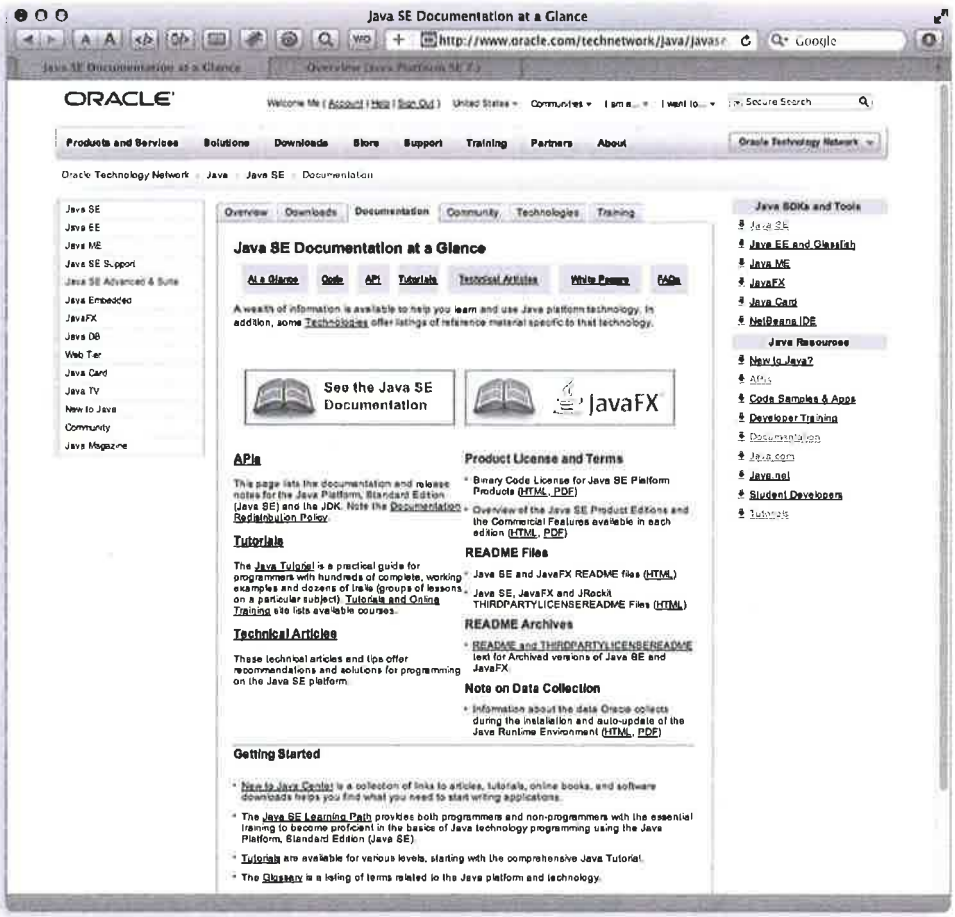
10 Java-Dokumentation

Sie haben in Kapitel 9, S. 98 gesehen, dass die Java-Laufzeitumgebung zahlreiche Klassen zur Verfügung stellt. Die Sprache und die Laufzeitumgebung von Java sind gut dokumentiert. In diesem Kapitel können Sie sich mit der **Java-Dokumentation** näher vertraut machen.

10.1 Wo finde ich die Java-Dokumentation?

Die gesamte Java-Dokumentation ist **online** zu finden. Zudem können Sie die wichtigsten Dateien auf Ihren lokalen Rechner herunterladen. So haben Sie die Dokumentation jederzeit auch ohne Internetverbindung **offline** zur Verfügung. Wenn Sie im Browser die URL **http://www.oracle.com/technetwork/java/javase/documentation/index.html** eingeben, gelangen Sie zu folgender Einstiegsseite:

[10-1] Einstiegsseite der Java-Dokumentation



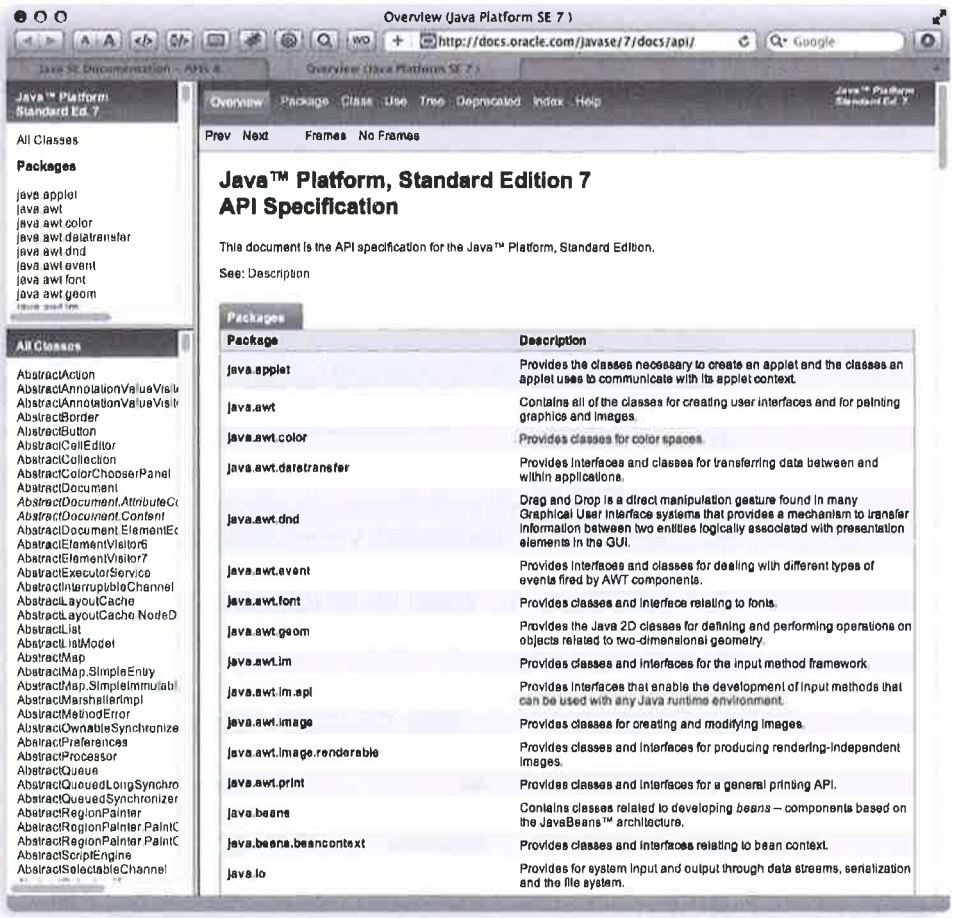
Folgende **Navigationspunkte** sind für uns besonders interessant:

Navigation	URL	Inhalt
API Specification	http://docs.oracle.com/javase/7/docs/api/	Beschreibung aller Java-Pakete und -Klassen
Tutorials	http://docs.oracle.com/javase/tutorial/index.html	Schritt-für-Schritt-Anleitungen für diverse Anwendungsfälle, wie beispielsweise Klassen einzusetzen und typische Programmierprobleme mit Java zu lösen sind

10.2 Wie ist die Java-Dokumentation aufgebaut?

Kein Softwareentwickler weiss immer alles auswendig. Ein Entwickler sollte aber wissen, wo er welche Informationen findet, die für die Programmierung notwendig sind. Die **API-Spezifikation** wird bei der Java-Programmierung am meisten gebraucht, weil sie wichtige Informationen für die korrekte Verwendung von Klassen und Methoden bereitstellt.

[10-2] API-Spezifikation von Java

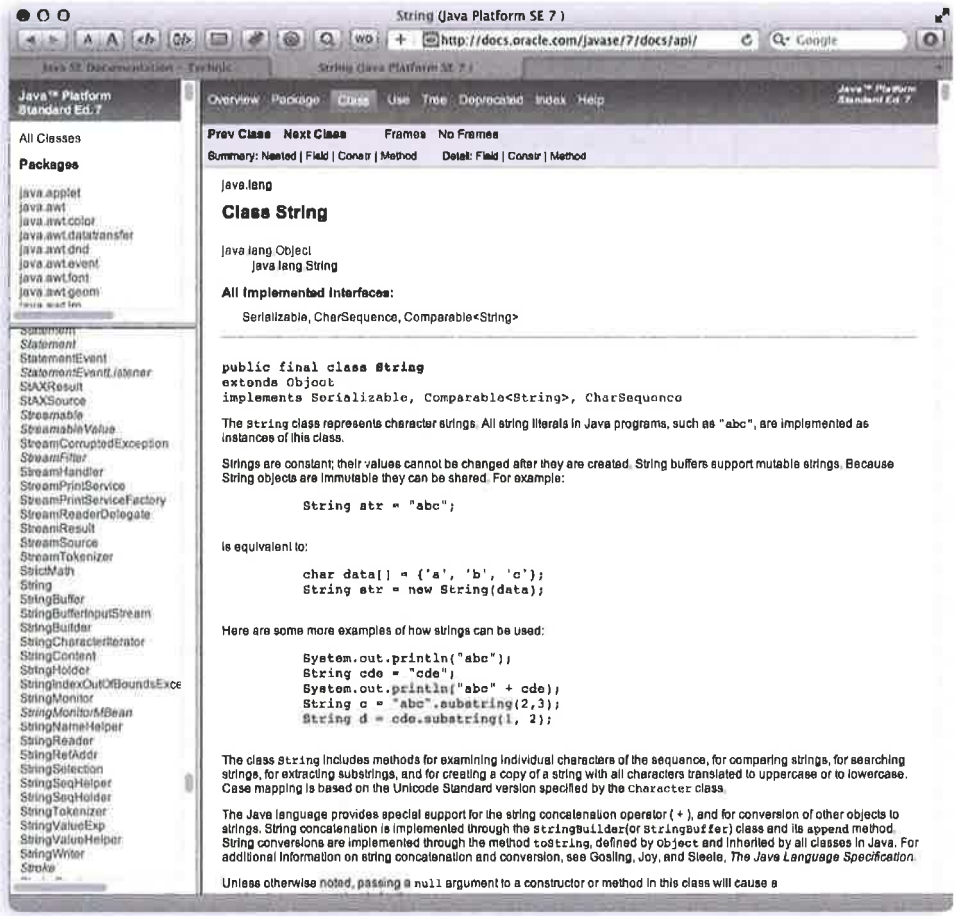


Nach dem Aufruf der API-Spezifikation sehen Sie ein **dreiteiliges Anzeige- und Auswahlfenster.** Im linken oberen Teilfenster finden Sie eine Liste aller Pakete in der Java-Laufzeitumgebung. Im linken unteren Teilfenster finden Sie eine Liste aller Klassen. Wenn Sie links oben ein Paket auswählen, werden in der Liste nur noch die Klassen aus diesem Paket angezeigt. Im grossen Anzeigebereich wird die eigentliche Java-Dokumentation gezeigt.

10.3 Wie wird die Klasse String dokumentiert?

Nachdem Sie in der linken unteren Liste die Klasse **String** ausgewählt haben, können Sie im grossen Anzeigebereich die **Dokumentation zur String-Klasse** einsehen:

[10-3] Dokumentation für java.lang.String



Zuoberst sehen Sie folgende Navigationsleiste:

[10-4] Navigation in der Dokumentation

Hier können Sie jederzeit zu verschiedenen weiteren Dokumentationen im Zusammenhang mit der aktuell angezeigten Dokumentation wechseln. In der folgenden Tabelle werden die einzelnen **Navigationspunkte** aufgeführt:

Navigation	Beschreibung
Overview	Wechsel zur Gesamtübersicht
Package	Wechsel zur Dokumentation des Pakets, in dem sich die Klasse befindet
Class	Dokumentation zu einer Klasse
Use	Gesamtübersicht über alle Klasse und Methoden, die die Klasse **String** verwenden
Tree	Übersicht der Verschachtelung und Vererbung aller Klassen
Deprecated	Liste aller überholten Klassen und Methoden. Eine als `deprecated` gekennzeichnete Klasse oder Methode ist zwar noch vorhanden, allerdings wird von deren Benutzung abgeraten, weil es sie in einer künftigen Java-Version nicht mehr geben wird
Index	Alphabetisches Stichwortverzeichnis
Help	Hilfe zur Dokumentation

Darunter folgen **allgemeine Informationen** zur angezeigten Klasse (hier: Klasse **String**).

[10-5] Allgemeine Klasseninformationen

java.lang
Class String

java.lang.Object
  └ java.lang.String

All Implemented Interfaces:
Serializable, CharSequence, Comparable<String>

Hier sehen Sie den Namen und das Paket der Klasse, die Vererbungshierarchie und weitere Details wie z. B. die implementierten Schnittstellen. Je nach Klasse finden Sie zusätzliche Rubriken. Der folgende Teil der Dokumentation bietet eine detaillierte Beschreibung der Klasse (inkl. Deklaration und Verwendung). Je nach Klasse ist die **Klassenbeschreibung** mehr oder weniger umfangreich. Hier eine Beschreibung der Klasse String als Beispiel:

[10-6] Klassenbeschreibung (Deklaration und Verwendung)

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

is equivalent to:

```
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

Im Anschluss daran werden die **Instanzvariablen, Konstruktoren und Methoden** der angezeigten Klasse beschrieben. Nachfolgend sehen Sie beispielhaft die Beschreibung der Methode `indexOf()` (hier: der Klasse **String**).

[10-7] Methodenbeschreibung

indexOf

```
public int indexOf(int ch)
```

Returns the index within this string of the first occurrence of the specified character. If a character with value `ch` occurs in the character sequence represented by this `String` object, then the index (in Unicode code units) of the first such occurrence is returned. For values of `ch` in the range from 0 to 0xFFFF (inclusive), this is the smallest value *k* such that:

```
this.charAt(k) == ch
```

is true. For other values of `ch`, it is the smallest value *k* such that:

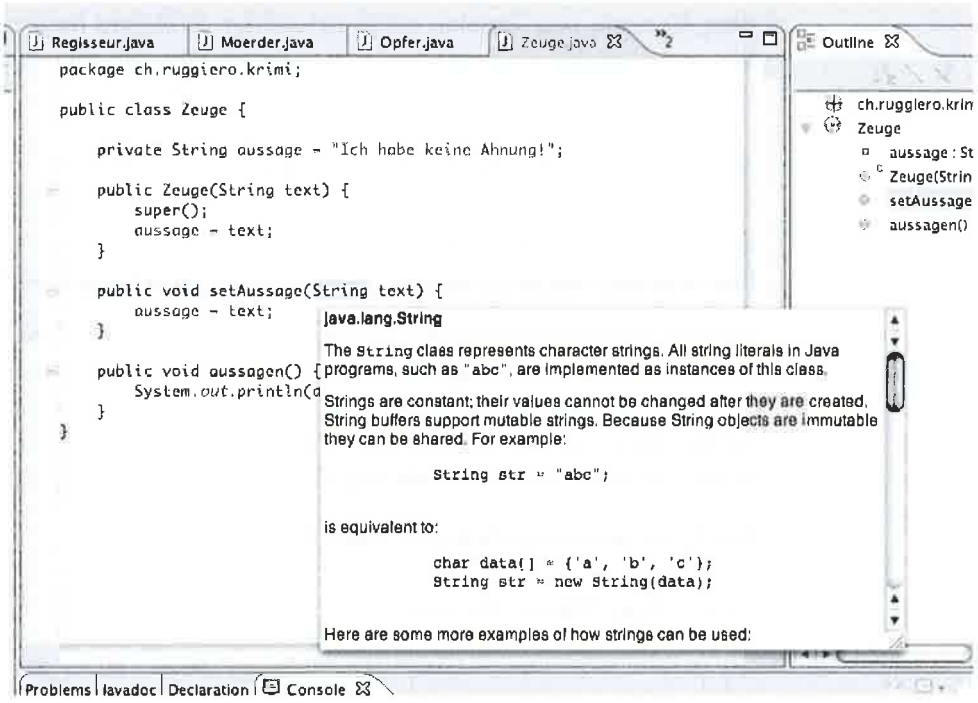
```
this.codePointAt(k) == ch
```

is true. In either case, if no such character occurs in this string, then `-1` is returned.

Parameters:
`ch` - a character (Unicode code point).
Returns:
the index of the first occurrence of the character in the character sequence represented by this object, or `-1` if the character does not occur.

Die Entwicklungsumgebung **Eclipse**[1] kann direkt auf die API-Dokumentation zugreifen und diese im **Editorfenster** anzeigen. Auf diese Weise können Sie die einzelnen Dokumente entsprechend Ihren Bedürfnissen bearbeiten und pflegen bzw. weiterentwickeln.

[10-8] API-Dokumentation in Eclipse (Beispiel)



Für **Java** ist eine ausführliche **Online-Dokumentation** verfügbar, die bei Bedarf auch lokal installiert und **offline** eingesehen werden kann. Das Entwicklungstool **Eclipse** kann direkt auf die Java-Dokumentation zugreifen und Programmierer bei ihrer Arbeit unterstützen.

Repetitionsfragen

72	Sie möchten die Java-Dokumentation der Klasse **Runtime** lesen. Wie machen Sie das?
73	Wie finden Sie heraus, in welchem Paket sich die Klasse **JFrame** befindet?
74	Was bedeutet es, wenn eine Java-Funktionalität als «deprecated» gekennzeichnet wird?
75	Wo finden Sie Hilfe, wenn Sie mit der Java-Dokumentation nicht zurechtkommen?

[1] Vergleichen Sie dazu das Kapitel 12, S. 115.

11 Schnittstellen zur Aussenwelt

Ein Java-Programm muss mit den **Anwendern** und anderen **Applikationen** interagieren können. Voraussetzung dafür sind geeignete Schnittstellen zur Aussenwelt. Dazu gehören sowohl Konsolen- und grafische Benutzerschnittstellen als auch Schnittstellen zu Dateien und Netzwerkverbindungen. In diesem Kapitel werden **Ein- und Ausgabenschnittstellen über Konsole und Dateien** näher beleuchtet. Einleitend dazu wird ausgeführt, was die Ein- und Ausgabe von Daten im Rahmen der objektorientierten Programmierung bedeutet.

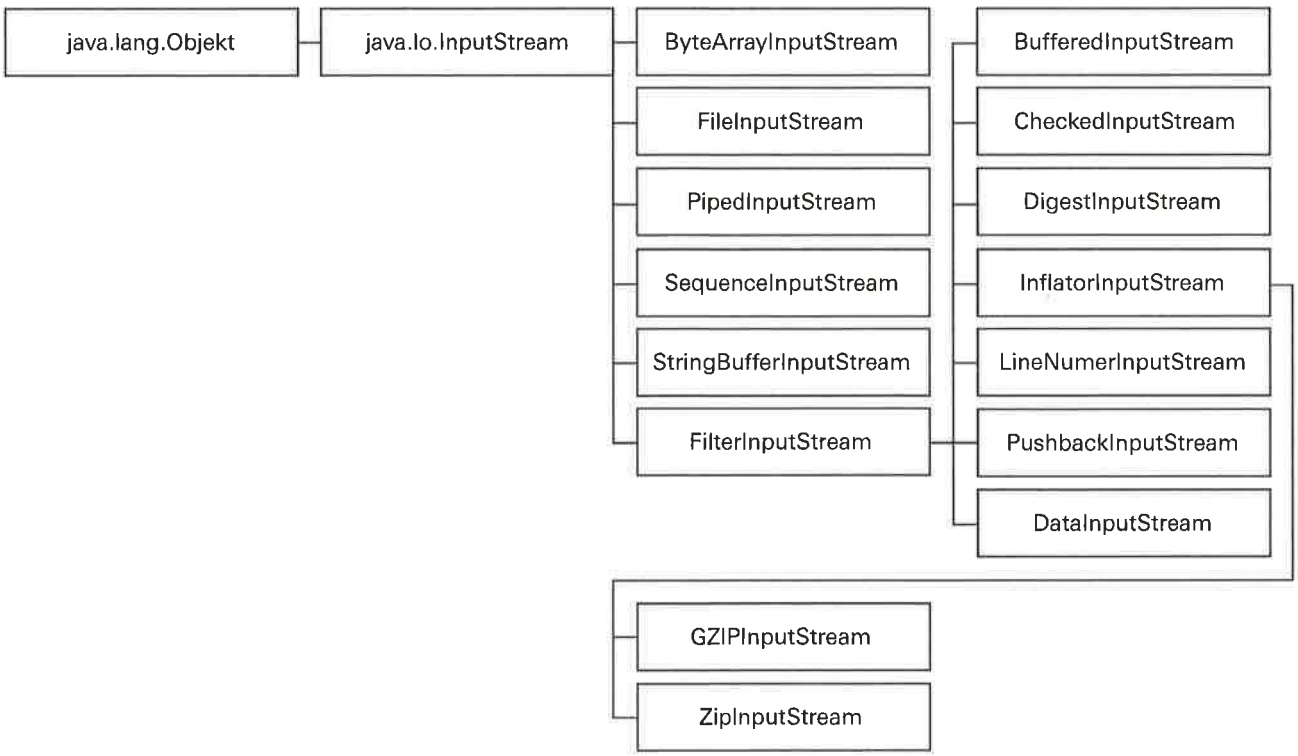
11.1 Objektorientierte Ein- und Ausgabe

Java behandelt die **Ein- und Ausgabe von Daten** nach objektorientierten Prinzipien und betrachtet grundsätzlich alle Ein- und Ausgaben als **Datenstrom in Form von Bytes**[1].

Entsprechend kommen für die Datenein- und -ausgabe geeignete Klassen zum Einsatz. Die beiden Klassen **java.io.InputStream** und **java.io.OutputStream** werden sowohl für die Ein- bzw. Ausgabe auf der Konsole als auch zum Lesen bzw. Schreiben von Dateien sowie für den Datentransfer via Netzwerk benötigt. Die nachfolgenden Diagramme sollen Ihnen einen Überblick über die Gliederung dieser beiden Klassen verschaffen.

Das Paket **java.io** beinhaltet folgende **Input-Klassen:**

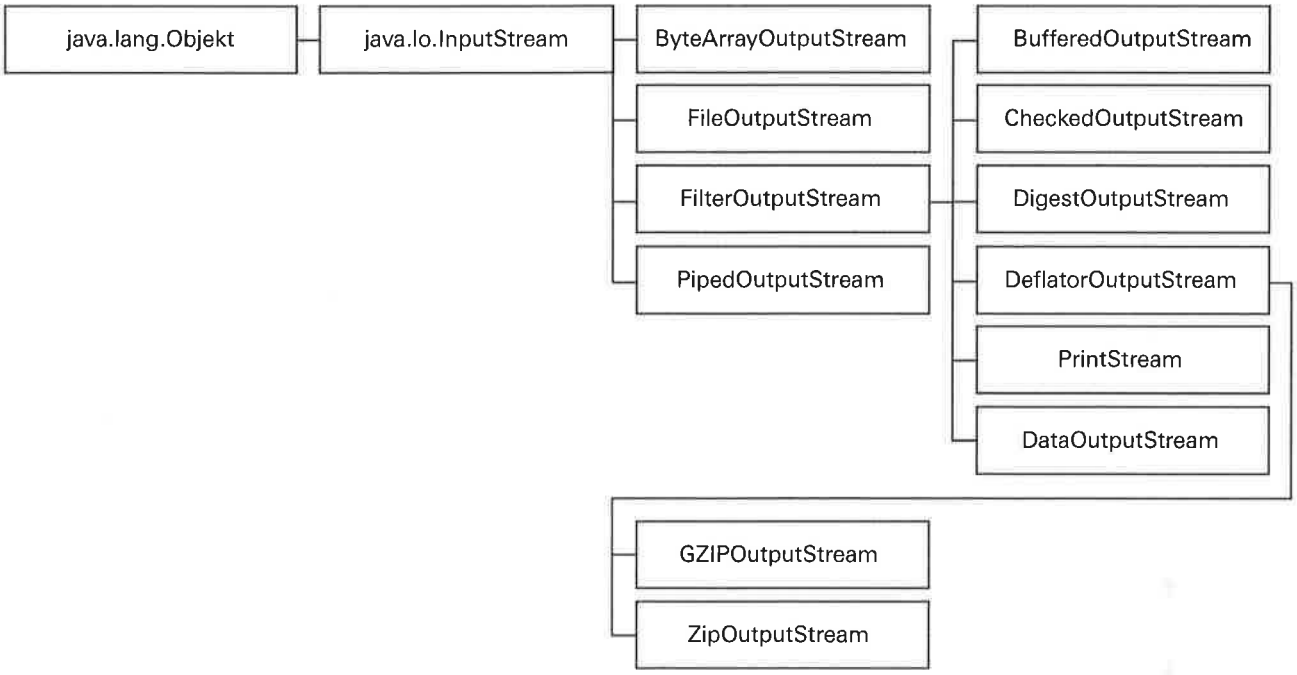
[11-1] Input-Klassen bei Java



[1] Fachbegriff: ByteStream. Engl. für: Datenstrom bzw. -abfolge (wörtl.). Hier: Kommunikationskanal.

... und folgende **Output-Klassen:**

[11-2] Output-Klassen bei Java



Je nach **Art der Ein- bzw. Ausgabe** werden unterschiedliche Klassen als Schnittstellen benötigt. Massgeblich sind dabei folgende Fragen:

- Handelt es sich um Textdateien oder um binäre Dateien?
- Findet ein linearer oder ein wahlfreier Zugriff statt?
- Handelt es sich um Objekte oder «nur» um Daten?
- Woher kommen bzw. wohin gehen die Bytes?
- Ist eine Filterung der Daten notwendig?

Wenn Sie beispielsweise **Objekte** ein- oder auslesen wollen, verwenden Sie die Klassen **ObjektInputStream** und **ObjectOutputStream.**

Als **Ziel und Quelle der Ein- und Ausgabedaten** können Dateien, Strings, Arrays oder Netzwerkverbindungen verwendet werden. Während des Datentransfers können Sie die Daten bei Bedarf filtern oder puffern (zwischenspeichern). So lassen sich z. B. grössere Datenmengen zusammenfassen, Prüfsummen berechnen und Daten komprimieren oder dekomprimieren[1]. Dadurch müssen nicht alle Bytes einzeln übertragen werden und der Datentransfer kann verkürzt bzw. Datenqualität und -sicherheit können erhöht werden.

[1] Zum Beispiel mittels ZIP-Komprimierung.

In der Klasse **InputStream** finden Sie folgende **Methoden:**

- Zum Lesen von Daten: `read()`
- Zum Überspringen von Daten: `skip()`
- Zum Überprüfen, ob Daten im Eingabekanal vorhanden sind: `available()`
- Zum Schliessen des Eingabekanals: `close()`

In der Klasse **OutputStream** finden Sie folgende **Methoden:**

- Zum Schreiben von Daten: `write()`
- Zum Leeren («Spülen») des Ausgabekanals: `flush()`
- Zum Schliessen des Ausgabekanals: `close()`

Wenn Sie Texte lesen oder schreiben wollen, können Sie die Klassen **Reader** bzw. **Writer** oder davon abgeleitete Klassen verwenden. Alle diese Klassen «verstehen» Unicode und können als Ziel und als Quelle sowohl Strings und Arrays als auch Netzwerkverbindungen und Dateien verwenden.

Beachten Sie, dass viele dieser Klassen mit **Exceptions** um sich werfen, da bei der Ein- und Ausgabe von Daten oft Probleme auftreten können. So kommt es beispielsweise immer wieder vor, dass Dateien nicht lesbar oder Netzwerkverbindungen nicht verfügbar bzw. unterbrochen sind. Daher sind oft **try-catch-Blöcke** notwendig.

11.2 Ein- und Ausgabe über die Konsole

Während einer laufenden Java-Anwendung erzeugt das Runtime-System automatisch Byte-Streams, die in der Klasse **System** über **statische Variablen** zur Verfügung stehen:

- **System.in:** Standardeingabe, normalerweise mit der Tastatur verbunden
- **System.out:** Standardausgabe, normalerweise mit dem Bildschirm verbunden
- **System.err:** Standardausgabe für Fehlermeldungen über den Bildschirm

11.2.1 Standardausgabe

Den Ausgabestream **System.out** haben Sie bereits kennengelernt. Auf die gleiche Art und Weise kann auch der Fehlerstream **System.err** eingesetzt werden:

```
System.out.println("Hallo Welt");
```

11.2.2 Konsoleneingabe

Um von der Konsole einlesen zu können, brauchen Sie einen **Reader,** d. h. ein Objekt aus einer Reader-Klasse. Damit unterschiedliche Datentypen richtig eingelesen werden, ist ein zusätzlicher Aufwand nötig. Im Rahmen dieses Lehrmittels wurde für diesen Zweck die Klasse **ConsoleReader** mit statischen Methoden entwickelt.

Hinweis

▷ Sie finden den vollständigen Code dieser Klasse im Anhang, S. 172. Für Lern- und Ausbildungszwecke können Sie sich diese vorgefertigte Schnittstelle herunterladen. Vergleichen Sie dazu das Linkverzeichnis auf S. 11.

Nachfolgend wird ein Ausschnitt aus der Klasse **ConsoleReader** abgebildet und besprochen:

```
package utils;
import java.io.*;

public abstract class ConsoleReader {

    private static BufferedReader reader = new BufferedReader(
                    new InputStreamReader(System.in));

    private ConsoleReader() {
        super();
    }
    public static int readInteger(String prompt)
    {
        int value = 0;
        while (true)
        {
            try
            {
                System.out.print(prompt + " _>");
                value = Integer.valueOf(reader.readLine()).intValue();
                break;
            }

            catch (IOException ex)
            {
                System.out.println("*** Read Error ***");
                System.out.println("*** " + ex.getMessage());
                System.exit(1);
            }
            catch (NumberFormatException ex)
            {
                System.out.println("*** Conversion Error ***, try again");
            }
        }
        return value;
    }
```

Wie Sie sehen können, enthält diese Klasse ausschliesslich **statische Methoden.** Und um die Erzeugung (Instanziierung) von Objekten zu unterbinden, ist der Defaultkonstruktor ausdrücklich aufgeführt und auf `private` gesetzt. Die Anweisung `new ConsoleReader()` ist auf diese Weise gar nicht möglich.

Weiter gibt es ein statisches Reader-Objekt. Mit der Klasse **BufferedReader** können nicht nur einzelne Zeichen, sondern ganze Zeilen eingelesen werden. Das BufferedReader-Objekt wird via **InputStream** mit Bytes versorgt, die über den Standardkanal **System.in** kommen. Auf dieses Reader-Objekt kann nun zugegriffen werden. Sehen Sie sich dazu die Methode `readInteger()` an. Sie dient dazu, die Eingabe in einen int-Wert umzuwandeln. Gleichzeitig kann sie eine Eingabeaufforderung ausgeben, deren Text im Eingabeparameter mitgegeben wird. Weil sowohl das Lesen als auch die Umwandlung der Eingabe in einen int-Wert fehlschlagen kann, muss mit einem **try-catch-Block** gearbeitet werden.

Der «eigentliche» Lesevorgang läuft wie folgt ab:

1. Zuerst wird mittels `reader.readln()` eine Zeile eingelesen.
2. Die eingelesene Zeile wird der Methode `valueOf()` der Klasse **Integer** übergeben.
3. Die Klasse **Integer** versucht, ein Integer-Objekt zu konstruieren, dessen int-Wert mit dem Aufruf von `intValue()` in die Variable `value` geschrieben wird. Das läuft in einer while(true)-Schleife so lange, bis es klappt.

4. In den **catch-Blöcken** werden folgende Fehler abgefangen:
   - `IOException`: Wenn bei der Tastatureingabe ein grundsätzliches Problem auftritt, wird das Programm mit einer detaillierten Fehlermeldung abgebrochen.
   - `NumberFormatException`: Wenn die Umwandlung einer Eingabe in einen int-Wert nicht klappt, erhält der Benutzer eine Aufforderung zur erneuten Dateneingabe.
5. Am Ende wird der eingelesene int-Wert zurückgegeben.

Die Klasse **ConsoleReader** enthält weitere Methoden für das Einlesen von float-, char- und anderen Werten, die nach dem gleichen Schema aufgebaut sind. Mit ihrer Hilfe können Sie nun **sichere Konsoleneingaben** machen:

```
import utils.ConsoleReader;
...
int eingabe = ConsoleReader.readInteger("Bitte eine ganze Zahl eingeben");
```

11.3 Ein- und Ausgabe über Dateien

Für die Ein- und Ausgabe über Dateien stehen dieselben Klassen zur Verfügung wie für die Ein- und Ausgabe über die Konsole. Zusätzlich gibt es Klassen und Methoden, um mit Dateien selbst zu arbeiten.

11.3.1 Datentyp File

Objekte der Klasse **File** beschreiben die Ein- und Ausgabe über Dateien und unterstützen deren Bearbeitung. Für die Erzeugung einer neuen Ein- oder Ausgabedatei können Sie ein entsprechendes Objekt wie folgt erzeugen:

```
File einFile = new File("brief.txt");
```

Hier übergeben Sie dem Konstruktor einen **Dateinamen**. Wenn Sie zusätzlich einen Pfad angeben möchten, versehen Sie den Konstruktor mit folgenden Parametern:

```
File einAnderesFile =  new File ("/home/meier", "brief.txt");
```

Somit erstellen Sie eine Beschreibung für die Datei **/home/meier/brief.txt.** Nun können Sie testen, ob es die Datei tatsächlich gibt:

```
if (einFile.exists())...
```

... und ob es sich um eine **Datei** handelt:

```
if (einFile.isFile())...
```

... oder ob es sich um ein **Verzeichnis** handelt:

```
if (einFile.isDirectory())...
```

Weiter haben Sie folgende Bearbeitungsmöglichkeiten:

- Datei umbenennen: `renameTo()`
- Datei löschen: `delete()`
- Neues Verzeichnis erstellen: `mkdir()`
- Verzeichnisinhalt auflisten: `list()`

11.3.2 Dateien lesen

Um aus Dateien zu lesen, brauchen Sie einen **InputStream.**

- Für zeichenweises Lesen («Byte für Byte») bietet sich `FileInputStream` an.
- Für zeilenweises Einlesen von Unicode-Textdateien bietet sich `FileReader` an.

Vergleichen Sie dazu folgenden Beispielcode:

```
BufferedReader reader;
String line;

File dataFile = new File("myFile.txt");
try {
       reader = new BufferedReader( new FileReader(dataFile) );
       while ( (line = reader.readLine()) .= null ) {
       ...
       }
       reader.close();
}
catch (IOException e) {
       ...
}
```

Für zeilenweises Lesen aus einer Textdatei wird hier ein BufferedReader-Objekt verwendet. Das FileReader-Objekt liest einzelne Bytes, die als vollständige Zeilen zusammengefasst erhalten bleiben sollen. Der try-catch-Block ist dazu da, um Lesefehler abzufangen.

11.3.3 Dateien schreiben

Um in Dateien zu schreiben, brauchen Sie einen **OutputStream.**

- Für zeichenweises Schreiben («Byte für Byte») bietet sich `FileOutputStream` an.
- Für zeilenweises Schreiben von Unicode-Textdateien bietet sich `FileWriter` an.

Vergleichen Sie dazu folgenden Beispielcode:

```
String filename = "myFile.txt";
File theFile = new File(filename);
try {
    if (theFile.isFile()) {
    theFile.delete();
    }
    // String line = ...
    BufferedWriter  writer = new BufferedWriter(new FileWriter(filename));
    writer.write(line, 0, line.length());
    writer.newLine();
    ...
    writer.flush();
    writer.close();
}
catch (IOException ex) {
    System.out.println("*** I/O-ERROR could not create file " + ex.getMessage());
}
```

Hier werden Zeilen in eine Textdatei geschrieben. Falls diese Datei bereits existiert, soll sie gelöscht und neu erstellt werden.

In Java ist die Ein- und Ausgabe objektorientiert. Im Paket **java.io** gibt es verschiedenste Klassen, um Bytes oder (Unicode-)Texte einzulesen oder auszugeben. Für **Textdaten** bieten sich Reader- und Writer-Klassen an. Für **binäre Daten** verwendet man InputStream- und OutputStream-Klassen.

Als Quelle und Ziel von Ein- bzw. Ausgaben kommen Strings, Arrays, Dateien und Netzwerkverbindungen infrage. Grundsätzlich sind alle Ein- und Ausgabedaten ein Strom von Bytes (Klassen **InputStream** und **OutputStream**). Während der Arbeit mit einem Stream können Sie die Bytes filtern und puffern. Auf diese Weise kann der Datentransfer vereinfacht und beschleunigt werden. Die Klasse **File** beschreibt eine Datei und bietet Methoden, um Dateien zu bearbeiten.

Repetitionsfragen

76 Wozu dient die Klasse **File?** Antworten Sie in einem Satz.

77 Wozu dient ein `BufferedReader`?

78 Was bedeutet `System.out`?

79 Worin besteht die Aufgabe der Writer-Klassen?

12 Mit Entwicklungs- und CASE-Tools arbeiten

In diesem Kapitel können Sie einen Blick auf «Werkzeuge» werfen, die Ihnen dabei helfen, eine OO-Anwendung zu realisieren.

12.1 Mit Eclipse arbeiten

Die «beste» Umgebung für die Entwicklung einer OO-Anwendung gibt es nicht. Es gibt nur für bestimmte Zwecke mehr oder weniger geeignete **Entwicklungsumgebungen.** Nachfolgend wird **Eclipse** als Entwicklungstool vorgestellt und eingesetzt. Es handelt sich um das wohl am meisten eingesetzte Werkzeug für die Entwicklung von Java-Programmen.

12.1.1 Was ist Eclipse?

Eclipse ist ein **Open-Source-Werkzeug,** das ursprünglich von IBM entwickelt wurde und seit einigen Jahren von der Open Source Community gepflegt und weiterentwickelt wird. Eigentlich ist Eclipse eine **Runtime-Umgebung für Java-Programme.** Diese Umgebung ist besonders darauf ausgelegt, Entwicklungswerkzeuge für die Java-Programmierung zu erstellen. Wenn man von Eclipse spricht, meint man daher in den meisten Fällen die Eclipse-Entwicklungsumgebung für Java-Programme. Eclipse ist aber so aufgebaut, dass es durch **Plug-ins**[1] um beliebige Funktionalitäten erweitert werden kann. Neben Plug-ins zur Unterstützung des Programmierers gibt es Plug-ins für C und C++, für PHP und sogar für grafische Werkzeuge, um beispielsweise UML-Diagramme zu erstellen und daraus den Java oder C++-Code zu generieren.

12.1.2 Eclipse installieren

Eclipse kann einfach auf den verschiedensten Plattformen installiert werden. Es läuft unter Microsoft Windows, Apple Mac OS X, Linux und verschiedenen Unix-Betriebssystemen. Laden Sie die von Ihrem Betriebssystem benötigte Version von der **Eclipse-Downloadsite** herunter. Nachdem Sie das heruntergeladene Archiv entpackt haben, können Sie bereits mit der Arbeit beginnen. Eine Installation, wie Sie es vielleicht von Windows-Programmen her kennen, ist nicht nötig.

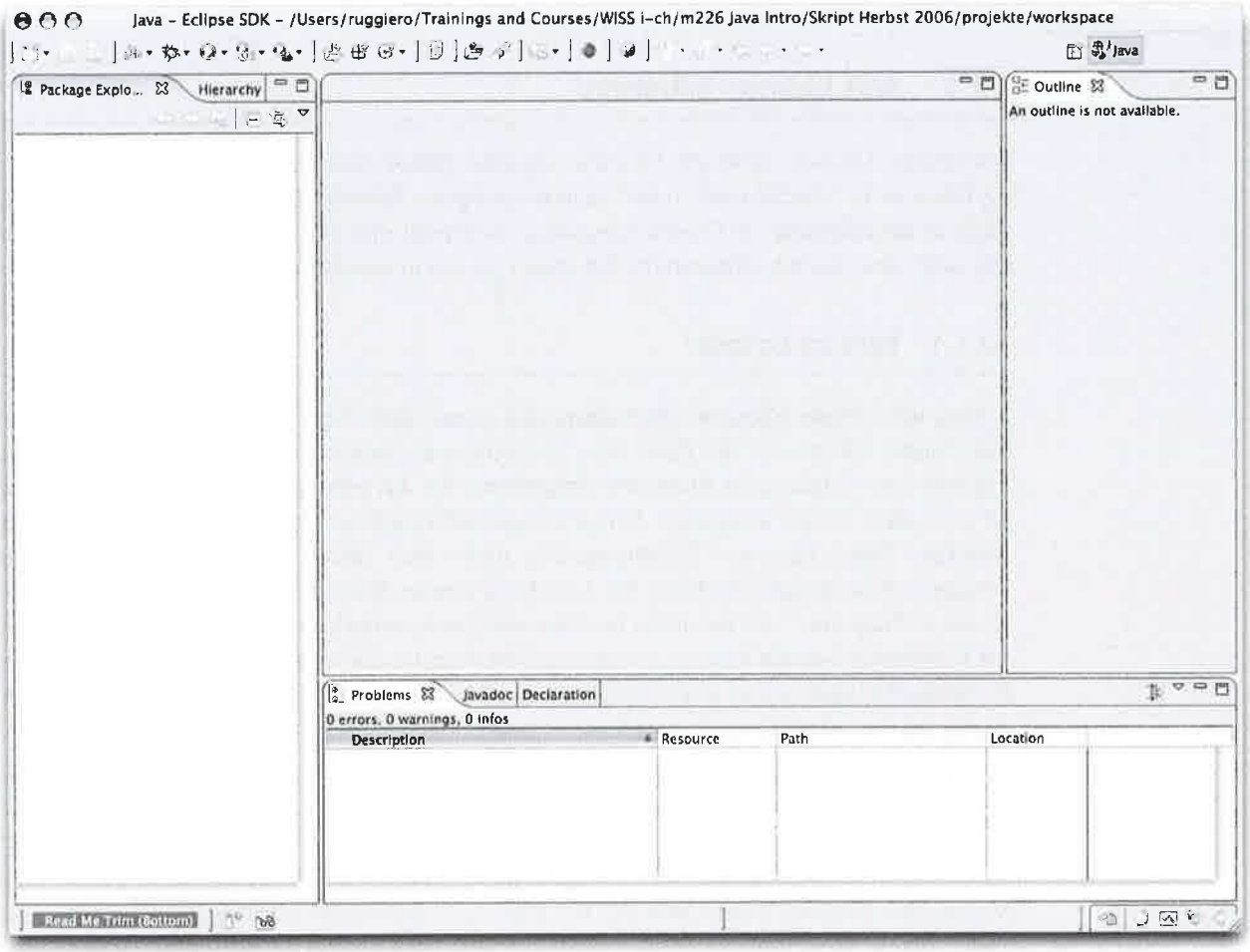
12.1.3 Handhabung von Eclipse

Im folgenden Abschnitt wird der Umgang mit Eclipse anhand eines einfachen Beispiels aufgezeigt. Eine ausführliche Online-Dokumentation dazu finden Sie auf der Website von Eclipse. Ausserdem bietet das **Help-Menü** detaillierte Schritt-für-Schritt-Anleitungen mit Beispielen. Wenn Sie noch nie mit Eclipse gearbeitet haben, sollten Sie sich unbedingt Zeit nehmen, um die **grundlegenden Konzepte von Eclipse** zu studieren.

[1] Softwaremodul, das von einer Anwendung während der Laufzeit erkannt und eingebunden werden kann.

Wenn Sie Eclipse das erste Mal starten, erscheint folgende Bildschirmmaske:

[12-1] Das leere Arbeitsfenster von Eclipse[1]



Das Arbeitsfenster von Eclipse ist in vier Bereiche gegliedert: Links finden Sie den **Package-Explorer,** in der Mitte den **Arbeitsbereich für den Programmcode.** rechts den **Bereich zur Darstellung der Klassenübersicht** und unten einen **Infobereich** mit mehreren Registern.

Neues Projekt erstellen

Erstellen Sie als Erstes ein neues Projekt, indem Sie mit der rechten Maustaste in den Bereich des **Package Explorer** klicken.

[1] Unter Windows und Linux liegt der Menübalken innerhalb des Dialogfensters, unter Mac OS X finden Sie den Menübalken ausserhalb des Dialogfensters am oberen Bildschirmrand.

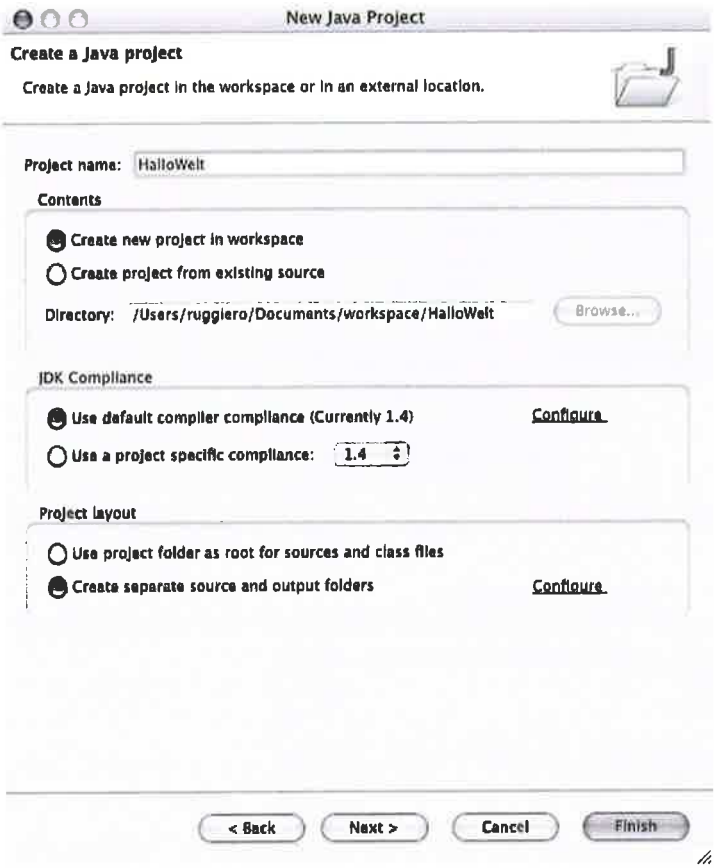
[12-2] Neues Projekt erstellen

Im darauf folgenden Dialogfenster wählen Sie den Projekttyp **«Java Project»**.

[12-3] Projekttyp auswählen

Mit **«Next >»** erreichen Sie folgendes Dialogfenster:

[12-4] Projekt benennen

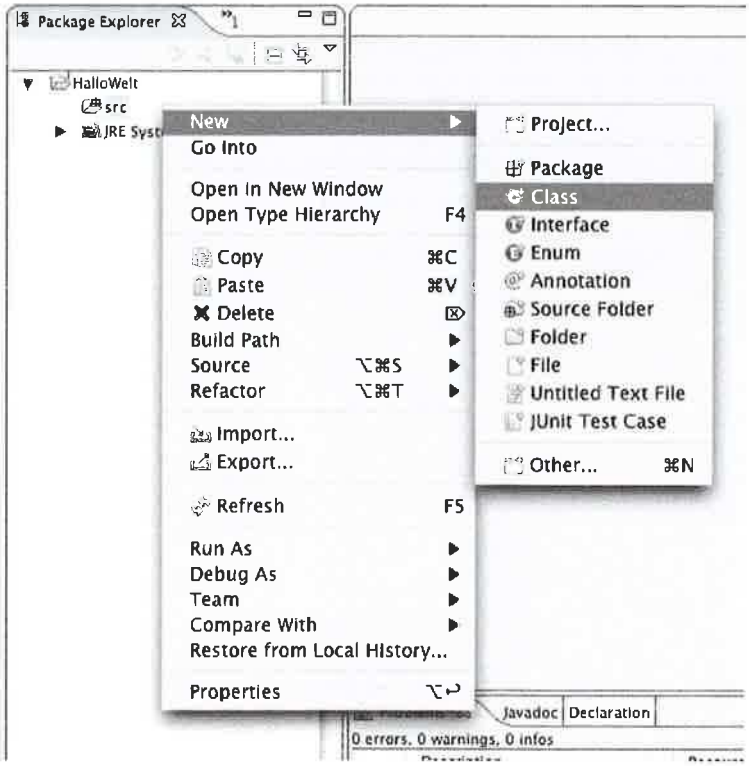


Hier können Sie dem Projekt einen Namen geben und erste Konfigurationen vornehmen. Klicken Sie danach auf **Finish.** Kurz darauf wird das neue Projekt im Package Explorer angezeigt.

Sourcedatei erstellen

Klicken Sie nun mit der rechten Maustaste auf den **Sourceordner,** um eine neue Java-Klasse zu erstellen.

[12-5] Neue Klasse erstellen

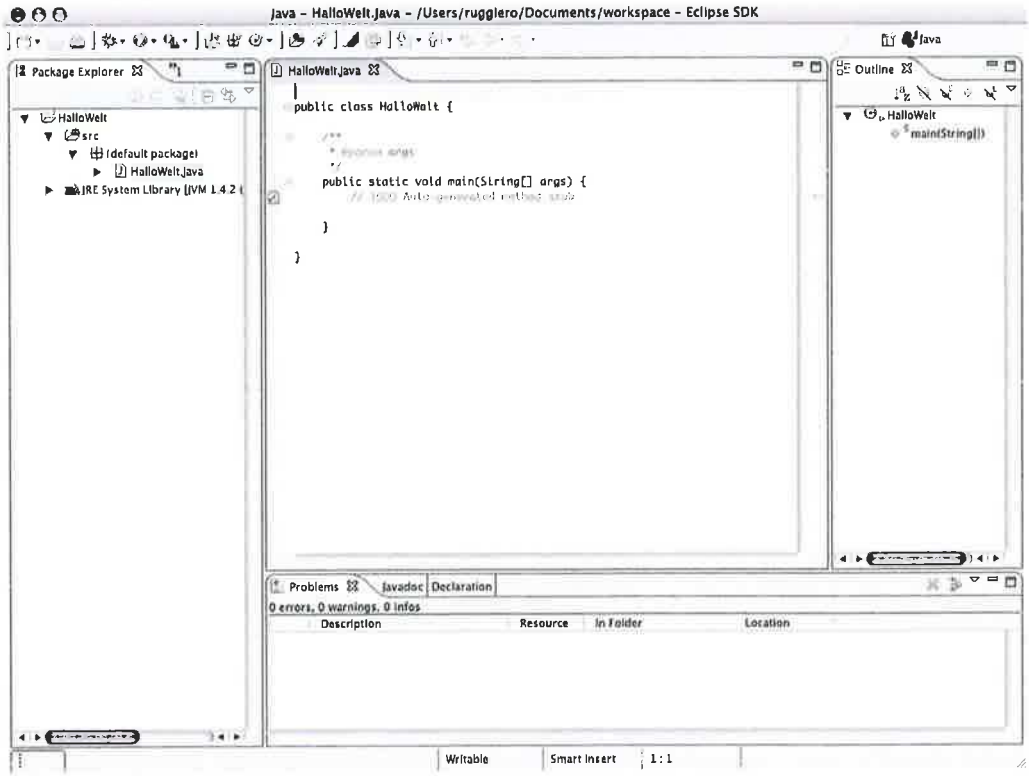


Geben Sie der Klasse einen **Namen.** Wenn Sie die erste Klasse eines Projekts definieren, brauchen Sie die **main()-Funktion.** Aktivieren Sie dazu die entsprechende Checkbox. Sie können für den Moment die im oberen Bereich des Dialogfensters angezeigte Warnung ignorieren.

[12-6] Eigenschaften der neuen Klasse definieren

Nach der Bestätigung der **Klasseneigenschaften** sollte ein vorbereiteter Sourcecode für die neue Klasse angezeigt werden, der eine noch leere main()-Funktion enthält.

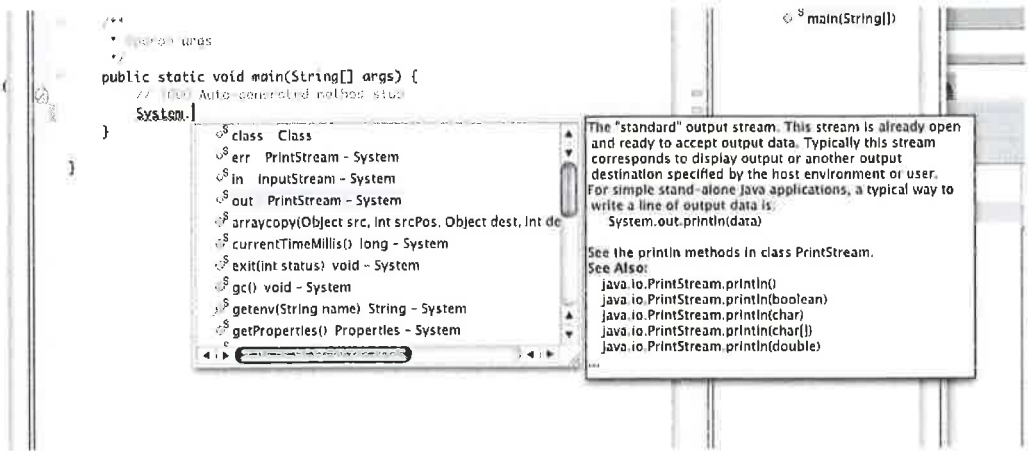
[12-7] Vorbereiteter Sourcecode (Beispiel)



Sourcecode schreiben

Beginnen Sie nun, den Programmcode für Ihre Klasse zu erfassen. Sie werden sehen, dass Ihnen Eclipse während des Schreibens viele Hilfestellungen gibt. **Syntaxfehler** werden unmittelbar angezeigt und mithilfe der Tastenkombination **[Ctrl]+[Leerschlag]** können Sie **Hilfestellungen und Informationen** zu den jeweiligen Klassen, Methoden und Variablen aufrufen.

[12-8] Direkthilfe im Editor aufrufen



Geben Sie doch einmal folgenden Code ein:

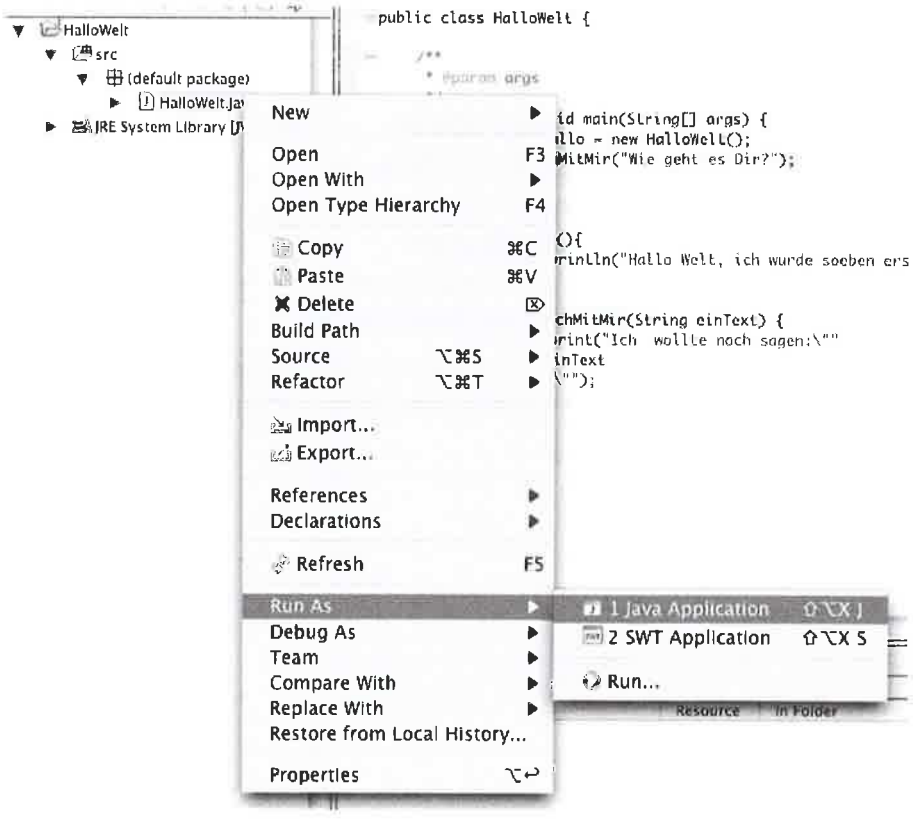
```
public class HalloWelt {
    /**
    * @param args
    */
    public static void main(String[] args) {
        HalloWelt hallo = new HalloWelt();
        hallo.sprichMitMir("Wie geht es Dir?");
    }

    public HalloWelt(){
        System.out.println("Hallo Welt, ich wurde soeben erstellt");
    }

    public void sprichMitMir(String einText) {
        System.out.print("Ich wollte noch sagen:\""
                         + einText
                         + "\"");
    }
}
```

Im rechten Teil des Arbeitsfensters erscheint nun eine Übersicht über die bearbeitete Klasse und mittels Klick auf die Einträge können Sie schnell an eine gewünschte Stelle im Programmcode springen.

Lauffähiges Programm erstellen

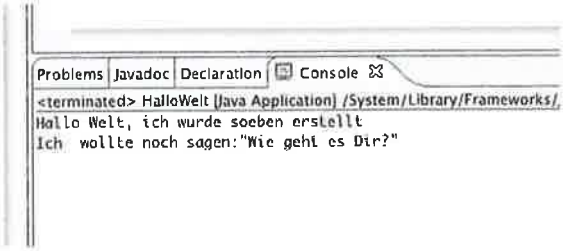
Wenn Sie in Eclipse eine Datei speichern, wird sie sogleich kompiliert. Auf diese Weise haben Sie immer ein lauffähiges Programm, das Sie starten können, indem Sie mit der rechten Maustaste im Package **Explorer** die betreffende Datei anklicken und danach die Option **Run As -> Java Application** auswählen.

[12-9] Java-Programm laufen lassen



Im unteren Bereich des Arbeitsfensters erscheint kurz darauf die Konsole, in der die Ausgabe des Programms zu sehen ist.

[12-10] Ausgabe des Programms



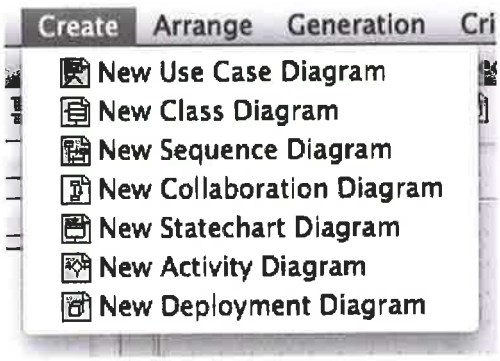
Gratulation! Sie haben mit Eclipse Ihre erste, einfache Java-Applikation erstellt.

12.2 Mit CASE-Tools arbeiten

CASE[1]**-Tools** unterstützen den Programmierer bei der Entwicklung von Applikationen und sind als Open-Source-Werkzeuge gratis erhältlich, können aber auch kommerziell erworben werden. Wesentliche Unterschiede zeigen sich meist im funktionalen Umfang und in der Unterstützung durch den Hersteller bzw. Lieferanten. Der Einsatz von CASE-Tools ist nicht unbedingt einfach, d. h., Sie müssen mit einem gewissen Einarbeitungsaufwand rechnen. **ArgoUML** ist ein Open-Source-CASE-Tool, das folgende Funktionalitäten unterstützt:

- Sie können zwischen verschiedenen Diagrammtypen auswählen:

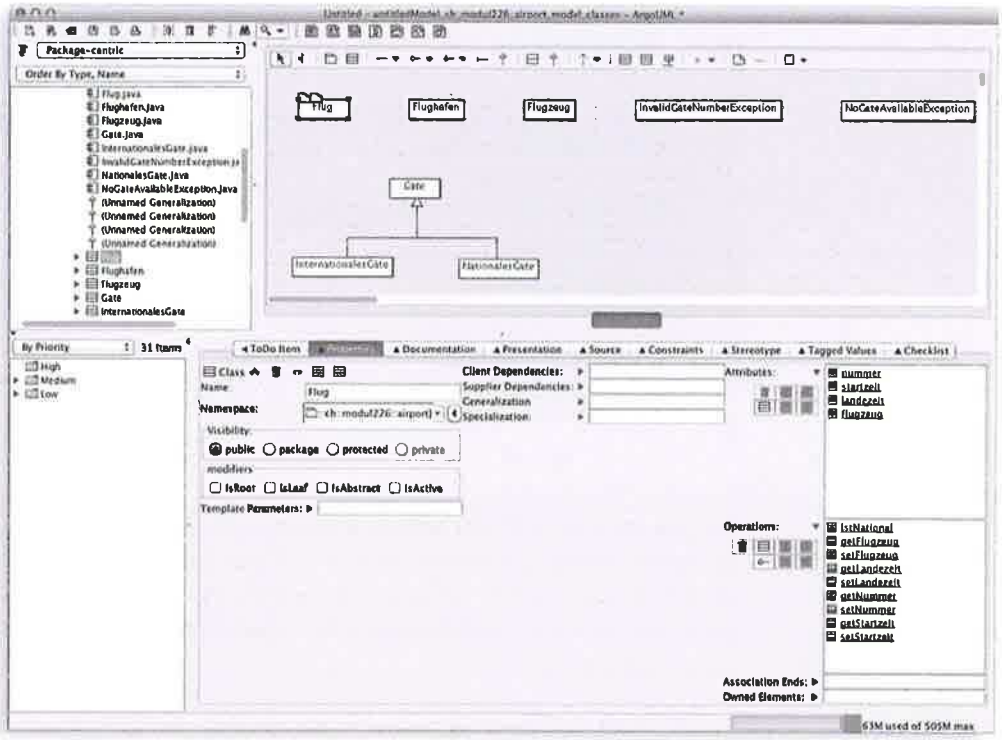
[12-11] Diagrammtyp auswählen



- Sie können Daten und Informationen direkt im jeweiligen Diagramm erfassen und beispielsweise alle Attribute und Methoden im Klassendiagramm definieren:

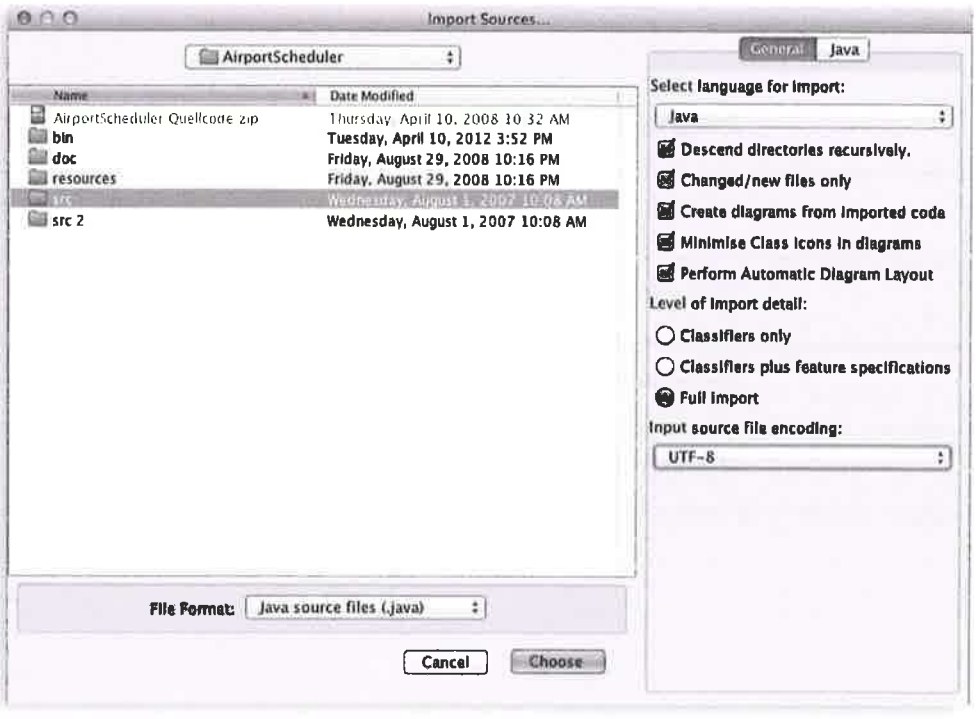
[1] Abk. für: Computer Assisted Software Engineering. Engl für: computerunterstützte Softwareentwicklung.

[12-12] Klassendiagramm definieren



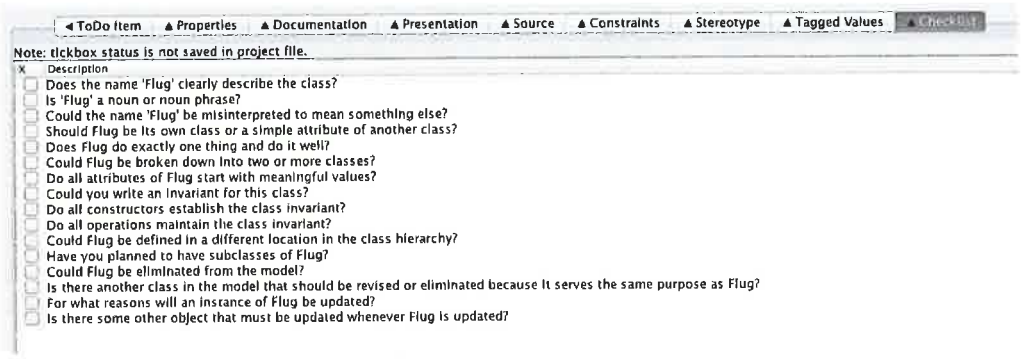
- Sie können den Programmcode der Methoden direkt erfassen.
- Sie können den gesamten Quelltext in Form von Dateien auf Knopfdruck exportieren. Diese sind danach in Eclipse kompilierbar.
- Umgekehrt können Sie auch Java-Quelltextdateien einlesen und daraus ein Klassendiagramm erstellen. Man nennt diesen Prozess auch **«Reverse Engineering».**

[12-13] Reverse Engineering bei ArgoUML



Gute CASE-Tools unterstützen Sie als Entwickler dabei, Fehler zu vermeiden, und stellen zu allen Objekten **Checklisten** zur Verfügung. So geht nichts vergessen.

[12-14] Checkliste zur Vermeidung von Programmierfehlern



Eclipse ist eine **OOP-Arbeitsumgebung,** die Entwickler bei der Programmierung und Kompilierung des Quellcodes unterstützt. Eclipse kann dem Programmierer viel Tipparbeit abnehmen und helfen, Fehler zu vermeiden. Weil die Java-Dokumentation in Eclipse eingebunden ist, kann der Programmierer jederzeit nachsehen, wie eine Klasse oder Methode genau einzusetzen ist.

CASE-Tools erlauben es dem Programmierer, verschiedene **Diagrammtypen** wie z. B. Klassen- oder Sequenzdiagramme mit dem Code zu verbinden. Mit dem Werkzeug ArgoUML kann beispielsweise aus Diagrammen direkt Programmcode erzeugt werden. Umgekehrt kann mittels Reverse Engineering aus bestehendem Programmcode direkt ein Klassendiagramm generiert werden. Zwischen einzelnen Diagrammen eines OOD darf es keine Widersprüche geben. ArgoUML stellt die Widerspruchsfreiheit sicher und warnt den Entwickler, wenn in einem Diagramm etwas fehlt oder mit einem anderen Diagramm nicht zusammenpasst.

Beide Werkzeuge sind sehr mächtig und brauchen eine gewisse Einarbeitungszeit.

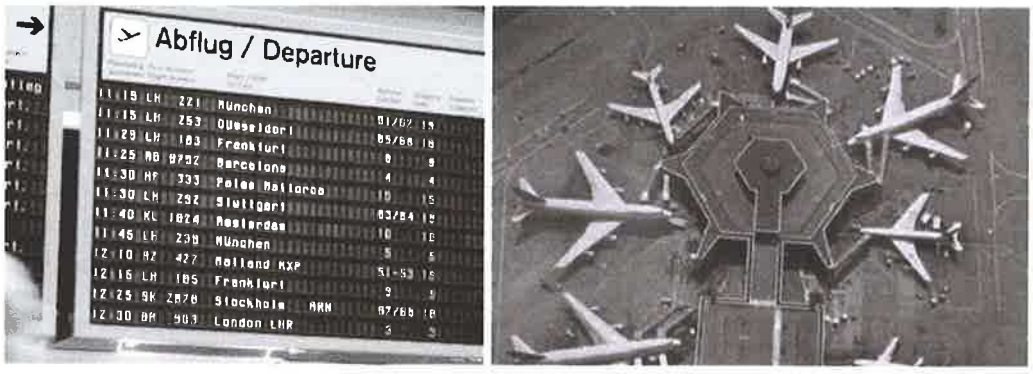
Repetitionsfragen

80	Wie lautet die wichtigste Tastenkombination von Eclipse und wozu dient sie?
81	Was verstehen Sie bei der OOP unter «Reverse Engineering»?
82	Wofür steht CASE?
83	Wie erstellen Sie in Eclipse eine neue Klasse?

13 Objektorientierte Anwendung realisieren

In diesem Kapitel können Sie das bisher erworbene Wissen über OOD und OOP anwenden und eine komplexere Anwendung entwickeln, die den Betrieb eines Flughafens simuliert. Diese Anwendung soll den Namen **Airportscheduler**[1] tragen.

[13-1] Anzeigetafel für Abflüge (links), Flugzeuge an den Gates (rechts)



Der Flughafen betreibt zehn **Gates**[2], wobei zwischen nationalen und internationalen Gates unterschieden wird. Die **nationalen Gates** sind alle gleich gross, während es bei den **internationalen Gates** zwei verschiedene Grössen gibt. Jedes Gate hat eine Nummer. Ein Gate ist entweder frei oder belegt.

Aufgrund der jeweiligen **Flugnummer** kann unterschieden werden, ob es sich um einen nationalen Flug (Flugnummer < 1000) oder internationalen Flug (Flugnummer >= 1000) handelt. Internationale Flüge können nur an einem internationalen Gate anlegen, nationale Flüge sind an nationalen und internationalen Gates möglich. Jede Flugnummer hat eine vorgegebene **Lande- und Abflugzeit.**

Auf dem Flughafen landen verschiedene Flugzeugtypen. **Propellerflugzeuge** werden für nationale und internationale Flüge eingesetzt, **Jets** dagegen nur für internationale Flüge. Jedes Flugzeug stellt eine bestimmte **Passagierkapazität** zur Verfügung. Für Flugzeuge mit 200 und mehr Passagieren wird ein grosses Gate benötigt. Ausserdem benötigt jeder **Flugzeugtyp** eine bestimmte Zeit zum Nachtanken.

13.1 Alle wichtigen Klassen finden

Versuchen Sie als Erstes, die **Klassen** zu identifizieren, die für das Projekt Airportscheduler benötigt werden. Dabei stellt sich jeweils die Frage, welche Klassen unbedingt benötigt werden. Weil Klassen die Eigenschaften und das Verhalten von Dingen bzw. Objekten beschreiben, wird schnell klar, wonach Sie suchen müssen. Welche **Objekte** kommen in der Aufgabenstellung vor? Vermutlich kommen Sie zu folgender Aufstellung:

- Flughafen
- Gate
- Flug
- Flugzeug

[1] Engl. für: Flughafen-Terminplaner (wörtl.).

[2] Andockstationen für Flugzeuge bzw. Ankunfts- oder Abflugstore für Passagiere.

In einer komplexeren Aufgabenstellung kann es sinnvoll sein, mit einem Leuchtstift die «Dingwörter» anzustreichen. Wovon ist immer wieder die Rede? Welche Dinge werden immer wieder erwähnt? Es ist nicht immer klar, ob solch ein «Ding» eine Klasse werden soll oder nicht. In diesen Fällen müssen Sie sich überlegen, ob und welche Eigenschaften das betreffende Ding hat. Liegen eindeutig spezifizierbare Eigenschaften vor, haben Sie vermutlich eine Klasse identifiziert. Es ist auch nicht immer möglich oder sinnvoll, bereits zu Beginn eines Entwicklungsprojekts alle benötigten Klassen zu identifizieren. Manchmal ist vielleicht die Aufgabenstellung so offen gehalten, dass Sie eigene Ideen zur Erweiterung des Programms einbringen können.

13.2 Klassendiagramm erstellen

Nachdem Sie alle wichtigen Klassen gefunden haben, machen Sie sich Gedanken über die **Beziehungen** und das **Zusammenspiel** zwischen den Objekten dieser Klassen. So gibt es beispielsweise zwischen Flughafen und Gate folgende Beziehungen:

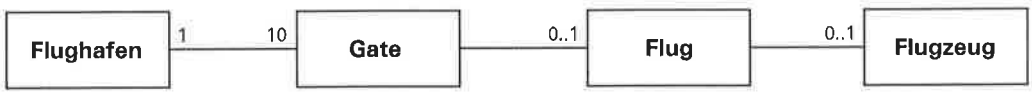
- Ein Flughafen hat Gates.
- Gates gehören zu einem Flughafen.

Daneben gibt es weitere Beziehungen wie etwa:

- Ein Flug wird mit einem Flugzeug durchgeführt.
- Ein Flugzeug belegt ein Gate.
- Ein Flug braucht ein Gate.

Ein erstes, einfaches **Klassendiagramm** könnte etwa wie folgt aussehen:

[13-2] Klassendiagramm (Beispiel)



Hinweis

▷ Denken Sie daran: Klassen sollen die Realität abbilden. Die Beziehung zwischen Gate und Flugzeug wird z. B. nicht eingezeichnet, da ein Flug mit unterschiedlichen Flugzeugen durchgeführt werden kann und Passagiere normalerweise einen Flug buchen und nicht ein Flugzeug. Entsprechend steht auf der Bordkarte auch: Flug LX1225 Einsteigen Gate 26.

13.2.1 Klasse Flughafen

Bei der Definition der Klasse **Flughafen** müssen Sie folgende Fragen beantworten:

- Welche Eigenschaften hat ein Flughafen? Der Flughafen hat einen Namen und zehn Gates.
- Welche Tätigkeiten sind für den Flughafen wichtig? Der Flughafen soll einen angemeldeten Flug landen lassen, indem er ein geeignetes freies Gate bestimmt, und einen Flug wieder starten lassen. Zudem soll der Flughafen Auskunft über den aktuellen Zustand der Gates geben können, d. h., er muss «seine Gates kennen».

Mit diesen Informationen können Sie das Klassendiagramm wie folgt erweitern:

[13-3] Klasse Flughafen

Flughafen
gates: Liste von Gates name: String
landen(flug: Flug) starten(gateNummer: integer) gateStatus(gateNummer: integer)

Hinweis

▷ Um die Klassenbeschreibung nicht unnötig «aufzublähen», werden einfache Zugriffsmethoden normalerweise nicht aufgeführt.

13.2.2 Klasse Gate

Bei der Definition der Klasse **Gate** müssen Sie folgende Fragen beantworten:

- Welche Eigenschaften hat ein Gate? Ein Gate hat eine Nummer, eine Grösse und einen Typ (national oder international).
- Welche Tätigkeiten muss ein Gate ausführen können? Es muss Auskunft über seinen Typ und seine Grösse geben können, denn der Flughafen braucht diese Informationen, wenn ein geeignetes Gate für eine Landung gesucht wird. Weiter muss es einen Flug und das entsprechende Flugzeug andocken und wieder freigeben können. Möglicherweise muss es auch Informationen über den Flug und das Flugzeug an den Flughafen weitergeben können. Zum jetzigen Zeitpunkt können Sie noch nicht alle Tätigkeiten des Gates vollständig angeben. Es ist gut möglich, dass es im Verlauf des Projekts bzw. der Programmierung noch Verschiebungen und Änderungen geben wird.

Je nach den Antworten kann die entsprechende Klasse beispielsweise so aussehen:

[13-4] Klasse Gate

Gate
typ: integer {0=national, 1=international} groesse: integer {0=klein, 1=gross} nummer: integer flug: Flug
landen(flug: Flug) starten() istFrei(): boolean getGroesse(): integer getTyp(): integer getFlug(): Flug getFlugzeug(): Flugzeug

Hinweis

▷ Beachten Sie, dass eine Methode `istFrei()` definiert werden soll, die darüber Auskunft gibt, ob ein Gate frei oder belegt ist. Dabei werden die get()-Methoden explizit aufgeführt, da sie für den reibungslosen Flughafenbetrieb eine gewisse Bedeutung haben.

13.2.3 Klasse Flug

Bei der Definition der Klasse **Flug** müssen Sie folgende Fragen beantworten:

- Welche Eigenschaften hat ein Flug? Jeder Flug hat eine Flugnummer, eine Lande- sowie eine Startzeit und setzt ein Flugzeug voraus.
- Welche Tätigkeiten muss ein Flug ausführen können? Ein Flug muss über sich und über das Flugzeug Auskunft geben können.

Je nach den Antworten kann die entsprechende Klasse beispielsweise so aussehen:

[13-5] Klasse Flug

Flug
nummer: integer startzeit: String landezeit: String flugzeug: Flugzeug
getFlugzeug(): Flugzeug getFlugnummer(): integer getStartzeit(): String getLandzeit(): String istNational(): boolean

Die Zugriffsmethoden werden hier explizit aufgeführt. Dabei ist v. a. die Methode `istNational()` interessant, die den Wert `true` oder `false` zurückliefert. Diese Information steckt bereits in der **Flugnummer**[1]. Damit Sie jederzeit die Nummerierung für nationale und internationale Flüge ändern können, ohne andere Klassen verändern zu müssen, die diese Information abfragen, empfiehlt sich eine spezialisierte Methode.

Vielleicht fragen Sie sich, weshalb die beiden get()-Methoden für die Start- und Landezeit einen String zurückgeben? Um die Anwendung möglichst einfach zu halten, soll keine Uhr in das Programm eingebaut werden, um die **Flugzeiten** auszuwerten. Stattdessen speichern Sie die Start- und Landezeiten am besten als Strings. Dieser Datentyp erlaubt es Ihnen, ggf. auch Einträge wie z. B. «heute», «samstags» oder «um Mitternacht» zu machen.

13.2.4 Klasse Flugzeug

Bei der Definition der Klasse **Flugzeug** sind folgende Fragen zu beantworten:

- Welche Eigenschaften hat ein Flugzeug? Ein Flugzeug hat eine Passagierkapazität und einen Typ (Propeller oder Jet). Zudem benötigt es eine gewisse Nachtankzeit am Gate.
- Welche Tätigkeiten muss ein Flugzeug ausführen können? Das Flugzeug muss über sich Auskunft geben können.

[1] Erinnern Sie sich? Bei Flügen mit einer Nummer unter 1000 handelt es sich um nationale Flüge. Bei Flügen mit einer Nummer von 1000 oder darüber handelt es sich um internationale Flüge.

Je nach den Antworten kann die entsprechende Klasse beispielsweise so aussehen:

[13-6] Klasse Flugzeug

Flugzeug
typ: integer {0=prop, 1=jet} nachtankzeit: String kapazitaet: integer

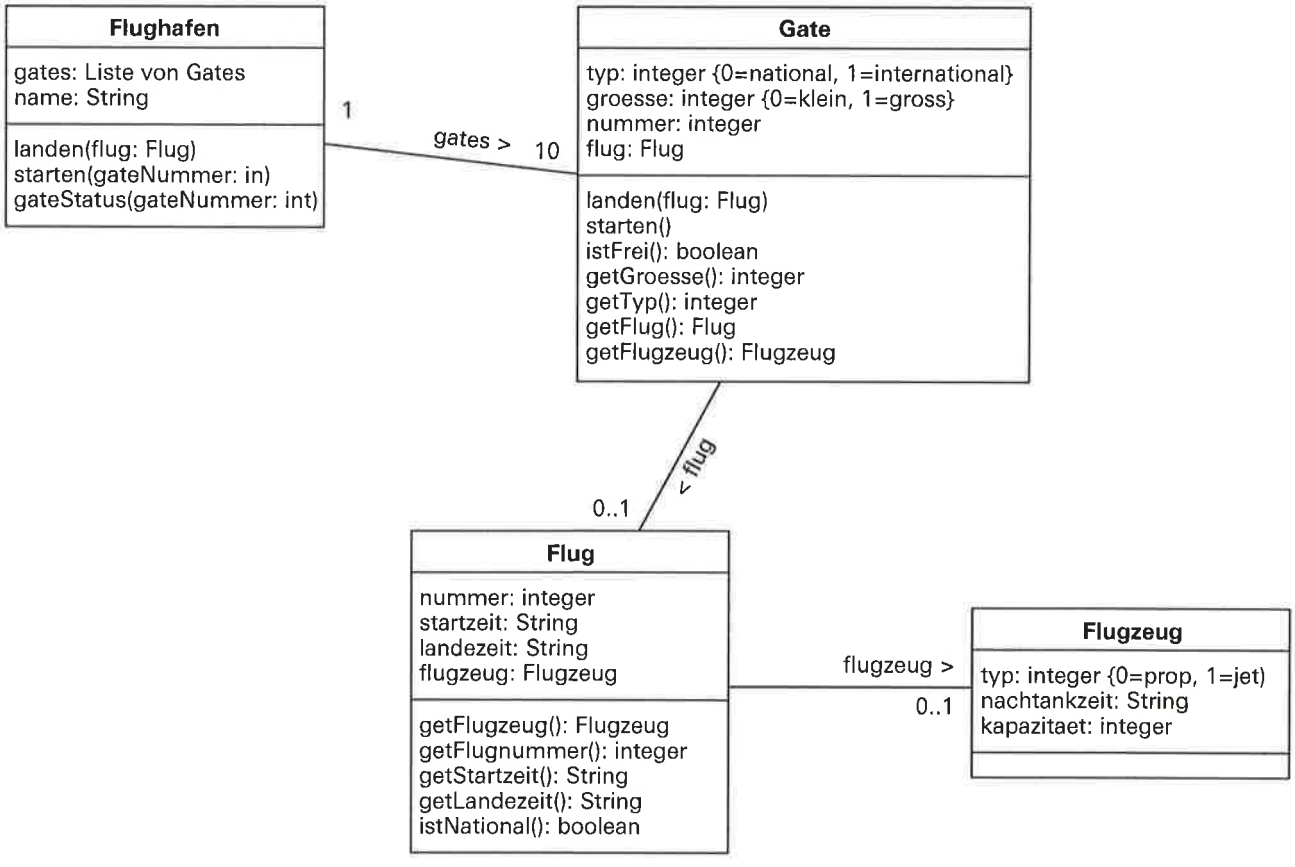
Hinweis

▷ Da ein Flugzeug ausser den normalen Zugriffsmethoden keine speziellen Methoden hat, können Sie in der UML-Darstellung den Methodenbereich auch leer lassen.

13.2.5 Vollständiges Klassendiagramm

Nun können Sie das **Klassendiagramm** wie folgt komplettieren:

[13-7] Vollständiges Klassendiagramm



Wie Sie sehen können, sind die Beziehungslinien zwischen den Klassen mit zusätzlichen Informationen versehen (gates, flug, flugzeug). Es handelt sich um **Rollenbezeichnungen,** die Sie im Java-Code wiederfinden werden. Die kleinen Pfeile geben die **Leserichtung** an.

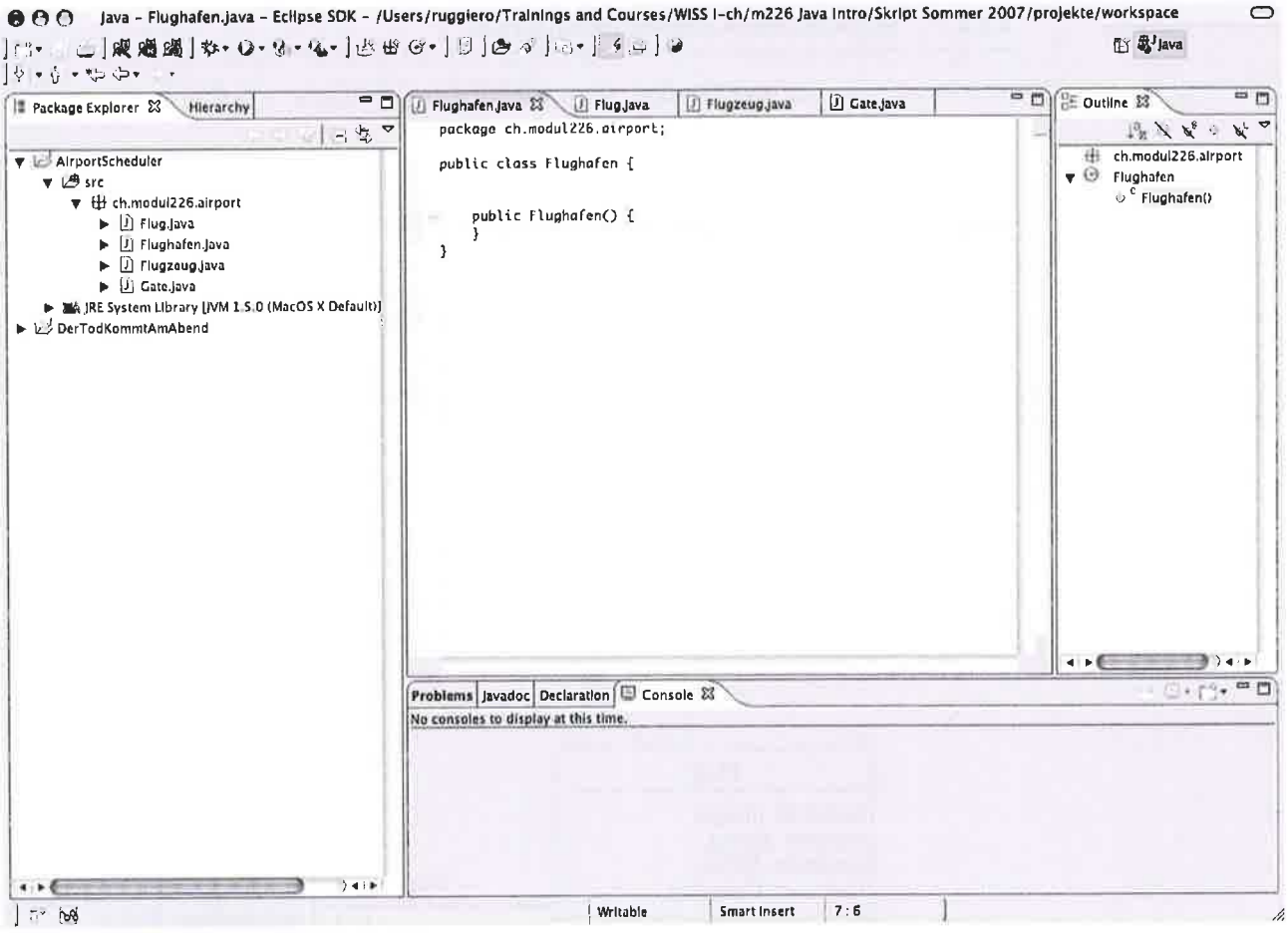
13.3 OOD implementieren

Nun soll das obige Klassendiagramm «übersetzt» und eine erste Version der Anwendung **Airportscheduler** realisiert werden.

13.3.1 Projekt in Eclipse anlegen

Zunächst legen Sie in Eclipse ein neues Projekt mit der Bezeichnung **AirportScheduler** an und erstellen das zugehörige Paket **ch.modul226.airport.** Darin definieren Sie die Klassen **Flughafen, Gate, Flug** und **Flugzeug.** Keine dieser Klassen braucht eine main()-Methode. Das Projekt könnte nun etwa so aussehen:

[13-8] Neues Projekt «AirportScheduler»



13.3.2 Flughafen bauen

Nun realisieren Sie den Flughafen, indem Sie in der Klasse **Flughafen** die fett markierten Codezeilen einfügen. Diese Klasse sieht nun wie folgt aus:

```
package ch.modul226.airport;

public class Flughafen {

    private Gate[] gates;
    private String name;

    public Flughafen() {
        this.name = "Gatwick";
        this.gates = new Gate[10];
    }

    public int landen(Flug flug) {
    }

    public void starten(int gateNummer) {
    }

    public void gateStatus(int gateNummer) {
    }
}
```

Dem Klassendiagramm entnehmen Sie die beiden Instanzvariablen `gates` und `name`. Für die Liste der zehn Gates verwenden Sie ein Array. Sie benötigen drei Methoden und zusätzlich einen Konstruktor. Darin geben Sie dem Flughafen einen Namen und erstellen das Array für die Gates. Beachten Sie, dass Sie damit ein Array mit zehn Elementen erstellt haben, aber noch keine Gates vorhanden sind. Dazu brauchen Sie erst die Klasse **Gate.**

13.3.3 Klasse Gate realisieren

Sie können die Klasse **Gate** wie die Klasse **Flughafen** realisieren, indem Sie die nötigen Informationen über Instanzvariablen und Methoden aus dem Klassendiagramm ablesen. Allerdings haben Sie dann folgendes Problem: Wenn das Gate frei ist, soll die Methode `istFrei()` den Wert `true` zurückgeben. Bei belegtem Gate soll sie `false` zurückgeben. Doch was heisst frei und belegt genau?

Wenn das Gate **belegt** ist, soll die Instanzvariable `flug` eine Referenz auf das entsprechende Flugobjekt enthalten. Wenn das Gate dagegen **frei** ist, soll diese Instanzvariable nichts referenzieren. «Nichts» wird in Java durch das Schüsselwort `null` angegeben. Ergo können Sie die Methode `istFrei()` wie folgt realisieren:

```
public boolean istFrei() {
    return (this.flug == null);
}
```

Wenn `this.flug` gleich `null` ist, wird `true` zurückgegeben. Wenn ein Flug vorhanden ist, entspricht die Referenz `this.flug` nicht null und die Methode gibt `false` zurück.

Ein weiterer Stolperstein stellt die Methode `getFlugzeug()` dar. Weil das Gate nichts von einem Flugzeug «weiss», müssen Sie den Flug anfragen, ob er das Flugzeug «liefert»:

```
public Flugzeug getFlugzeug() {
    if (this.flug == null) {
        return null;
                        }
    return flug.getFlugzeug();
}
```

Damit ein Flug überhaupt abgefragt werden kann, muss er vorhanden sein. Dies wird im obigen Code in der ersten Zeile überprüft. Wenn kein Flug vorhanden ist, kann es auch kein Flugzeug geben und die Methode gibt den Wert `null` zurück. Nun meldet Eclipse aber, dass es beim Flug gar keine Methode `getFlugzeug()` gibt. Sie müssen sich daher als Nächstes unbedingt um die Klasse **Flug** kümmern.

13.3.4 Klasse Flug realisieren

Implementieren Sie die vier Instanzvariablen und die zugehörigen Zugriffsmethoden `setter()` und `getter()`. Auch in dieser Klasse haben Sie mit `istNational()` eine spezielle Zugriffsmethode. In den Anforderungen steht, dass nationale Flüge Nummern unter 1000 aufweisen. Daher können Sie `istNational()` wie folgt implementieren:

```
public boolean istNational() {
    return (nummer < 1000);
}
```

Nachdem das Grundgerüst für die Klassen steht, beginnt die Feinarbeit. Gehen Sie nun zurück zum Konstruktor des Flughafens. Der Flughafen besitzt noch keine Gates.

13.3.5 Gates konstruieren

Um ein Gate zu erzeugen, müssen Sie dessen Konstruktor mit `new` aufrufen. Jedes Gate lässt sich durch Typ, Grösse und Nummer charakterisieren. Am besten hinterlegen Sie diese Informationen direkt bei der Erzeugung eines Gates. Das lässt sich einfach erledigen, indem Sie dem Gate einen geeigneten **Konstruktor** geben.

```
public Gate(int nummer, int typ, int groesse) {
super();
    this.typ = typ;
    this.groesse = groesse;
    this.nummer = nummer;
}
```

Obiger Konstruktor verlangt alle drei Informationen über Gates und legt sie in der Instanzvariablen ab. Vergessen Sie nicht den Aufruf von `super()` als Erstes im Konstruktor. Danach können Sie die Gates in den «Flughafen» einbauen, indem Sie im Konstruktor des Flughafens folgende Zeilen hinschreiben:

```
// vier nationale Gates
gates[0] = new Gate(1, 0, 0);
gates[1] = new Gate(2, 0, 0);
gates[2] = new Gate(3, 0, 0);
gates[3] = new Gate(4, 0, 0);

// 2 kleine internationale Gates
gates[4] = new Gate(5, 1, 0);
gates[5] = new Gate(6, 1, 0);

// vier grosse internationale Gates
gates[6] = new Gate(7, 1, 1);
gates[7] = new Gate(8, 1, 1);
gates[8] = new Gate(9, 1, 1);
gates[9] = new Gate(10, 1, 1);
```

Vielleicht werden Sie sich nun zu Recht fragen: Wie erinnert man sich als Programmierer daran, dass mit 0 ein kleines Gate und mit 1 ein internationales Gate gemeint ist? Manchmal bedeutet 0 auch ein nationales und 1 ein grosses Gate. Allein vom Code her lässt sich dieser Sachverhalt nicht ableiten. Obiger Code ist daher missverständlich.

Besser wäre es, anstelle der Ziffern 0 und 1 **Konstanten** mit einem aussagekräftigen Namen zu definieren. Am meisten Sinn macht es wohl, solche Konstanten in der Klasse **Gate** zu definieren, da sie **Werte für die Attribute dieser Klasse** darstellen. Nun wissen Sie aber, dass Sie nur auf Instanzvariablen zugreifen können, wenn ein Objekt vorliegt. Weil die Konstanten von Objekten unabhängig sind, müssen sie mit dem Schlüsselwort `static` versehen werden. Aus diesem Grund fügen Sie in der Klasse **Gate** direkt vor den Instanzvariablen folgende Codezeilen ein:

```
public static final int KLEIN = 0;
public static final int GROSS = 1;
public static final int NATIONAL = 0;
public static final int INTERNATIONAL = 1;
```

Es hat sich in Java eingebürgert, **Konstantennamen** in Grossbuchstaben zu schreiben. Aus Gründen der Übersichtlichkeit und Kompatibilität empfiehlt es sich, diese Konvention einzuhalten. Entsprechend sieht der Code für die Erstellung von zehn Gates wie folgt aus:

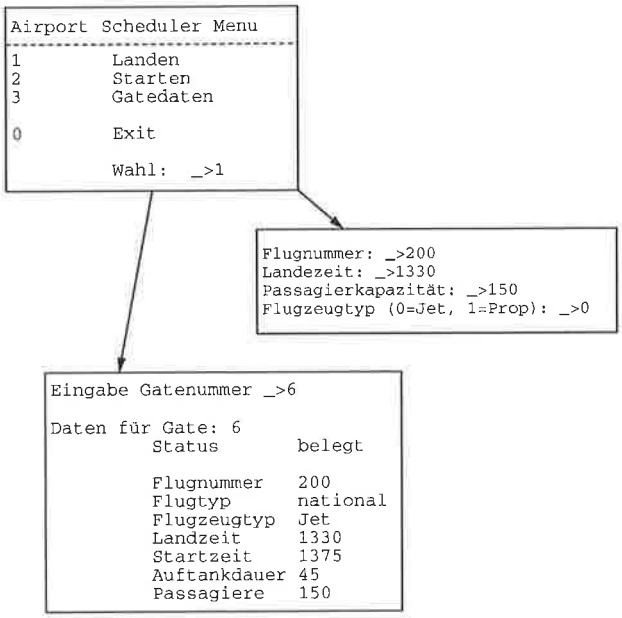
```
// vier nationale Gates
gates[0] = new Gate(1, Gate.NATIONAL, Gate.KLEIN);
gates[1] = new Gate(2, Gate.NATIONAL, Gate.KLEIN);
gates[2] = new Gate(3, Gate.NATIONAL, Gate.KLEIN);
gates[3] = new Gate(4, Gate.NATIONAL, Gate.KLEIN);

// zwei kleine internationale Gates
gates[4] = new Gate(5, Gate.INTERNATIONAL, Gate.KLEIN);
gates[5] = new Gate(6, Gate.INTERNATIONAL, Gate.KLEIN);

// vier grosse internationale Gates
gates[6] = new Gate(7, Gate.INTERNATIONAL, Gate.GROSS);
gates[7] = new Gate(8, Gate.INTERNATIONAL, Gate.GROSS);
gates[8] = new Gate(9, Gate.INTERNATIONAL, Gate.GROSS);
gates[9] = new Gate(10, Gate.INTERNATIONAL, Gate.GROSS);
```

Vielleicht ist Ihnen aufgefallen, dass die **Gatenummer** jeweils um eine Nummer grösser ist als die Position des Gates im Array. Der Grund: In einem Java-Array beginnt die Zählung immer bei 0. In einem Flughafen beginnt die Zählung der Gates dagegen normalerweise bei 1. Ein weiterer Makel: Nationale Gates gibt es nur in einer einzigen Grösse. Dennoch müssen Sie die Grösse der Gates bei deren Konstruktion angeben. Diese «Unschönheit» werden später in Kapitel 13.4, S. 139 beseitigt.

13.3.6 Benutzerschnittstelle realisieren

Mit der Konstruktion der Gates sind Sie schon weit fortgeschritten. Noch haben Sie aber keine lauffähige Anwendung und es gibt auch noch keine Möglichkeit, die Anwendung zu bedienen. Als Schnittstelle für die Benutzereingaben soll eine reine **Konsolenoberfläche** realisiert werden. Hier ein Vorschlag, wie diese aussehen könnte:

[13-9] Benutzerschnittstelle via Konsoleneingabe



Im Hauptmenü werden – wenn nötig – vom Benutzer bestimmte Eingaben verlangt. Auch der Code zur Darstellung der Menüs und der Dateneingabe muss in einer oder mehrerer Klassen untergebracht werden. Sie können z. B. das Hauptmenü in der Klasse **Flughafen** und die Dateneingabe für einen Flug in der Klasse **Flug** programmieren. Dies funktioniert und Sie hätten wohl bald eine lauffähige Applikation.

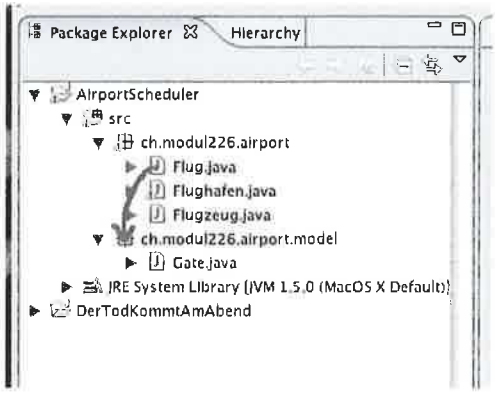
Allerdings können dadurch auch **Probleme** entstehen: Was machen Sie etwa, wenn später eine grafische Benutzerschnittstelle gewünscht wird? Oder wenn Flugdaten nicht mehr von Hand eingegeben, sondern automatisch aus einer Datei oder Datenbank eingelesen werden sollen? Oder wenn die Datenausgabe in eine PDF-Datei umgeleitet werden soll? Weil der **Programmcode für die Benutzerschnittstelle über die gesamte Anwendung verteilt** ist, müssen Sie dann in zahlreichen Klassen Änderungen vornehmen. Betroffen wären eventuell auch Klassen, die bisher problemlos funktioniert haben. Jedes Mal, wenn Sie Änderungen am Programmcode vornehmen, besteht die Gefahr, dass etwas nicht mehr korrekt läuft. Es kann aber auch passieren, dass Sie nicht überall die notwendigen Änderungen vornehmen. Wenn aus irgendwelchen Gründen eine notwendige Anpassung vergessen geht, herrscht schnell Chaos in der Applikation.

Der Code für Benutzerschnittstellen sollte daher vom Code der restlichen Applikation getrennt werden. Zu diesem Zweck erstellen Sie die Pakete **ch.modul226.airport.model** und **ch.modul226.airport.view.** Im **model-Paket** bringen Sie alle Klassen unter, die mit der Logik des Programms zu tun haben[1]. In das **view-Paket** stecken Sie alles, was die Benutzeroberfläche und -interaktion betrifft.

Die Klassen **Flughafen, Gate, Flug** und **Flugzeug** stellen also die **Geschäftsklassen** dar. Ziehen Sie nun diese vier Klassen in der Paketansicht von Eclipse einzeln mit der Maus in das neue model-Paket. Eclipse fragt kurz nach und passt danach alle Paketreferenzen im gesamten Sourcecode automatisch der neuen Struktur an.

[1] Man nennt diese Klassen auch Geschäftsklassen.

[13-10] Klassen in Paket verschieben



Danach sind alle Modellklassen im richtigen Paket.

13.3.7 Hauptmenü erstellen und Einstieg definieren

Für das **Hauptmenü** und den **Einstieg in die Applikation** definieren Sie eine Klasse. Diese soll **Hauptmenue** heissen, zum view-Paket gehören und die main()-Methode enthalten.

Nachdem Sie in Eclipse die Klasse **Hauptmenue** erstellt haben, brauchen Sie als Nächstes eine Methode, die das Hauptmenü ausgibt. Ausserdem brauchen Sie am Anfang einige Leerzeilen, um die Darstellung auf dem Bildschirm übersichtlicher zu gestalten. Der zugehörige Programmcode könnte wie folgt aussehen:

```
private void anzeigen() {
    System.out.println();
    System.out.println();
    System.out.println();
    System.out.println("Airport Scheduler Hauptmenü");
    System.out.println("===========================");
    System.out.println();
    System.out.println("1 Landen");
    System.out.println("2 Starten");
    System.out.println("3 Gatedaten");
    System.out.println();
    System.out.println("4 Ende");
}
```

Weiter brauchen Sie eine Methode, die die vom Benutzer getroffene Wahl abfragt und die passende Aktion ausführt. Dabei greifen Sie auf die Klasse **ConsoleReader** zurück, die Sie bereits in Kapitel 11.2, S. 110 kennengelernt haben. Zu diesem Zweck erstellen Sie im Projekt das Paket **utils** und legen Sie darin den Quelltext der Klasse **ConsoleReader** ab, indem Sie die Datei vom Dateisystem in Eclipse mit der Maus ins Paket **utils** ziehen. Danach erstellen Sie die Methode `aktion()`, die vom Benutzer mittels ConsoleReader die gewünschte Menüwahl abfragt.

```
public boolean aktion() {
    while (true) {
        anzeigen();

        int eingabe = ConsoleReader.readInteger("Ihre Wahl");
        switch (eingabe) {
            case 1:
                landen();
                return true;
            case 2:
                starten();
                return true;
            case 3:
                gateDatenAnzeigen();
                return true;
            case 4:
                return false;
            default:
                System.out.println("Ungueltige Wahl");
        }
    }
}
```

Mit obiger Methode wird immer genau eine **Menüwahl** verarbeitet. Wenn der Benutzer «Beenden» wählt, gibt die Methode den Wert `false` zurück, in allen anderen Fällen ist der Rückgabewert `true`. Mittels Schleife können Sie die Methode so lange aufrufen, bis der Rückgabewert `false` ist. Danach endet die Schleife und somit auch das Programm. Nachfolgend sehen Sie die main()-Methode mit der Menüschleife.

```
public static void main(String[] args) {
    Hauptmenue menue = new Hauptmenue();
    while (menue.aktion()) {
        // einfach immer wieder anzeigen
    }
    System.out.println();
    System.out.println("Besten Dank und auf Wiedersehen.")
}
```

Noch können Sie die neue Applikation nicht laufen lassen, da in der **Aktionsmethode** drei Methoden aufgerufen werden, die es noch nicht gibt. Erstellen Sie diese drei Methoden vorerst als «Dummies», in denen einfach ausgegeben wird, dass die aktuelle Methode gerade dran ist:

```
private void landen() {
    System.out.println("Landen...");
}

private void starten() {
    System.out.println("starten...");
}

private void gateDatenAnzeigen() {
    System.out.println("Gate Daten anzeigen...");
}
```

Nun können Sie die **Steuerung des Hauptmenüs** testen. Starten Sie dazu die Applikation, indem Sie in der Klasse **Hauptmenue** die main()-Methode auswählen. Wenn alles richtig funktioniert, wird in der Folge das Hauptmenü angezeigt. Hier können Sie die einzelnen Menüpunkte anwählen und das Programm wieder beenden.

13.3.8 Menüsteuerung und Flughafen verbinden

Beim Start der Applikation sollte auch der Flughafen «gebaut» werden. Fügen Sie dazu in der Datei **Hauptmenue.java** unterhalb der package-Anweisung folgende Zeile ein:

```
import ch.modul226.airport.model.*;
```

Auf diese Weise machen Sie die **model-Klassen** bekannt, die ja in einem anderen Paket liegen. Nun müssen Sie innerhalb der Klasse **Hauptmenue** noch eine Instanzvariable und einen Konstruktor definieren. In der Folge wird der Flughafen konstruiert, wenn das Hauptmenü erstellt wird.

```
private Flughafen flughafen;

public Hauptmenue() {
    flughafen = new Flughafen();
}
```

13.3.9 Ein Flugzeug landet

Wenn ein Benutzer im Hauptmenü den Menüpunkt **Landen** auswählt, soll das Programm die aktuellen Daten für den Flug und das Flugzeug abfragen und ein dazu passendes Gate suchen. In der Klasse **Hauptmenue** haben Sie bereits die Methode `landen()` vorbereitet, die korrekt aufgerufen wird. Nun sollen zwei weitere Methoden erstellt werden: Eine Methode soll nach den Flugdaten fragen und ein Flugobjekt erstellen, eine andere Methode soll die Eingabe der Flugzeugdaten verlangen.

Flug und Flugzeug erstellen

Diese Methode erstellt ein Flugobjekt und fragt den Benutzer nach den entsprechenden Angaben. Danach wird die Methode `flugzeugDatenBeschaffen()` aufgerufen, die anhand der Benutzerangaben ein Flugzeug erstellt. Abschliessend werden der Flug und das Flugzeug miteinander verbunden und der «fertige» Flug an den Aufrufer der Methode zurückgegeben. Fügen Sie in der Klasse **Hauptmenue** folgende Methoden ein:

```
private Flug flugDatenBeschaffen() {
    Flug flug = new Flug();
    System.out.println();
    System.out.println();
    System.out.println("Bitte Flugdaten eingeben");
    System.out.println("-------------------------");
    flug.setNummer(ConsoleReader.readInteger("Flugnummer"));
    flug.setLandezeit(ConsoleReader.readString("Landezeit"));
    flug.setStartzeit(ConsoleReader.readString("Startzeit"));

    flug.setFlugzeug(flugzeugDatenBeschaffen());
    return flug;
}
private Flugzeug flugzeugDatenBeschaffen() {
    Flugzeug flugzeug = new Flugzeug();
    flugzeug.setTyp(ConsoleReader.readInteger("Flugzeugtyp (0=Prop, 1=Jet)"));
    flugzeug.setKapazitaet(ConsoleReader.readInteger("Passagierzahl"));
    flugzeug.setNachtankzeit(ConsoleReader.readString("Nachtankzeit"));
    return flugzeug;
}
```

Beide Methoden sind `private`, da sie nur innerhalb der Klasse **Hauptmenue** verwendet werden.

Flugzeug landen lassen

Die Methode `landen()` in Hauptmenue kann etwa so aussehen:

```
private void landen() {
    System.out.println("\n\n\nLanden...\n\n");
    Flug flug = flugDatenBeschaffen();
    int gateNummer = flughafen.landen(flug);
    if (gateNummer >= 0) {
       System.out.println("Maschine ist an Gate " + gateNummer + " gelandet");
    }
    else {
      System.out.println("ERROR Kein freies Gate gefunden, Flug muss umgeleitet
werden");
    }
}
```

Beachten Sie, dass die Methode `landen()` in der Klasse **Flughafen** neuerdings die **Gatenummer** zurückgibt. Die bisherige Dummymethode hatte keinen Rückgabewert. An diesem Beispiel können Sie die Trennung zwischen Geschäftslogik und Darstellung gut erkennen. Die Datenbeschaffung ist Aufgabe der Benutzerschnittstelle, der Landevorgang bzw. das Auffinden eines passenden Gates und das Andocken an diesem Gate haben dagegen nichts mit der Benutzeroberfläche zu tun. Dies ist eine Aufgabe der **Geschäftslogik im model-Paket.** In unserem Beispielprojekt übergibt die landen()-Methode im Hauptmenü den Flug an den Flughafen zur weiteren Bearbeitung.

Die Methode `landen()` sieht nun wie folgt aus:

```
public int landen(Flug flug) {
    for (short i = 0; i < 10; i++) {      // Schleife über alle Gates
        if (gates[i].istFrei()) {         // nur freie Gates
            if (flug.istNational()) {     // nationaler Flug passt immer
                gates[i].landen(flug);
                return i + 1;
            }
            else {                        // internationaler Flug
              if (flug.getFlugzeug().istGrossraum()) {    // Grossraumflugzeug
                    if (gates[i].getGroesse() == Gate.GROSS) {
                        gates[i].landen(flug);
                        return i+1;       // Gate ist gross->OK, passt
                    }
                }
                else {                    // Kein Grossraum
                    gates[i].landen(flug);
                    return i+1;           // Gate passt immer
                }
            }
        }
    }
    return -1;
}
```

In der obigen Schleife wird jedes Gate angeschaut. Wenn ein Gate frei ist, wird überprüft, ob der Flug zum Gate passt. Ein **nationaler Flug** passt zu jedem Gate. Bei einem internationalen Flug muss nachgesehen werden, ob das Flugzeug gross oder klein ist. Ein kleines Flugzeug passt immer, ein grosses Flugzeug braucht ein genügend grosses Gate. Der eigentliche Landevorgang bzw. das Andocken wird durch das Gate erledigt. Wenn die for-Schleife über alle Gates gelaufen ist und kein passendes Gate gefunden wurde, wird der Wert `-1` zurückgegeben.

Fügen Sie in der Klasse **Gate** in der Methode `landen()` folgende Zeile hinzu:

```
public void landen(Flug flug) {
    this.flug = flug;
}
```

Testen Sie die neue Applikation aus und lassen Sie ein Flugzeug landen. Funktioniert alles?

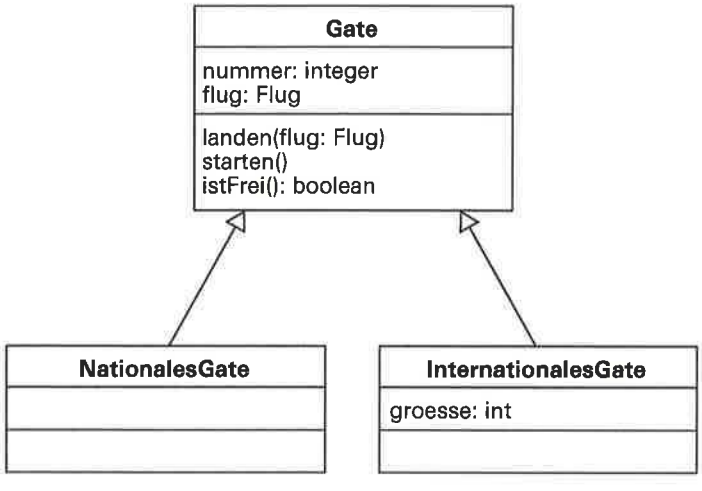
Hinweis

▷ Funktionen wie «Gate-Informationen anzeigen» oder «Flugzeug starten» werden an dieser Stelle nicht weiter vertieft. Sie können den vollständigen Code des Beispielprojekts via Internet herunterladen. Vergleichen Sie dazu auch das Linkverzeichnis auf S. 11.

13.4 Vererbung implementieren

Erinnern Sie sich noch an die Erstellung der Gates? Während nationale Gates immer «gross genug» sind, werden bei internationalen Gates zwei Grössen unterschieden. Nationale und internationale Gates können daher als Spezialfälle der Klasse **Gates** aufgefasst und die Mechanismen **Vererbung** und **Polymorphismus** angewendet werden. Das entsprechende Klassendiagramm sieht wie folgt aus:

[13-11] Vererbung bei den Gates



Die Klasse **Gate** bekommt einen Konstruktor mit einem einzigen Parameter. Neben Grösse und Typ entfallen auch zugehörigen die Instanzvariablen und Zugriffsmethoden. Zudem werden die vier statischen finalen Variablen für Grösse und Typ ersatzlos gestrichen. Damit ist die Klasse **Gate** viel einfacher geworden:

```
package ch.modul226.airport.model;

public class Gate {

    // Instanzvariablen
    private int nummer;
    private Flug flug;

    public Gate(int nummer) {
        this.nummer = nummer;
    }

    public void landen(Flug flug) {
        this.flug = flug;
    }

    public void starten() {
        this.flug = null;
    }

    public boolean istFrei() {
        return (this.flug == null);
    }

    public int getFlugnummer() {
        return nummer;
    }

    public Flug getFlug() {
        return flug;
    }

    public Flugzeug getFlugzeug() {
        if (this.flug == null) {
            return null;
        }
        return flug.getFlugzeug();
    }
}
```

Wie sieht die Klasse **NationalesGate** aus? Hier gibt es nicht mehr viel Code:

```
package ch.modul226.airport.model;

public class NationalesGate extends Gate {

    public NationalesGate(int nummer) {
        super(nummer);
    }
}
```

Hier brauchen Sie nur noch einen einfachen Konstruktor mit einem einzigen Parameter, der **Gatenummer.** Diese wird an den Konstruktor der Oberklasse **Gate** weitergeleitet. Die Klasse **NationalesGate** hat weder eigene Instanzvariablen noch eigene Methoden.

Auch der Programmcode für internationale Gates sieht nun einfach und übersichtlich aus:

```
package ch.modul226.airport.model;

public class InternationalesGate extends Gate {

    // Konstanten
    public static final int KLEIN = 0;
    public static final int GROSS = 1;

    private int groesse;

    public InternationalesGate(int nummer, int groesse) {
        super(nummer);
        this.groesse = groesse;
    }

    public int getGroesse() {
        return groesse;
    }
}
```

Weil bei den internationalen Gates weiterhin zwischen zwei verschiedenen Grössen unterschieden wird, finden Sie die beiden Konstanten an dieser Stelle. Der Konstruktor hat zwei Parameter, die **Gatenummer** und die **Gategrösse.** Während die Nummer an den Konstruktor der Oberklasse **Gate** weitergereicht wird, wird die Grösse in einer eigenen Instanzvariablen abgelegt. Damit die Grösse eines Gates abgefragt werden kann, brauchen Sie noch eine entsprechende Zugriffsmethode. Das ist schon alles.

Welche Änderungen braucht es sonst noch? Aufgrund der Vererbung benötigen Sie die bisher verwendeten Konstanten für Gate-Typ nicht mehr. Deshalb müssen die Gates nun auf eine etwas andere Art und Weise erstellt werden. Hier der entsprechende Ausschnitt der Klasse **Flughafen,** in dem die Gates «gebaut» werden:

```
// vier nationale Gates
gates[0] = new NationalesGate(1);
gates[1] = new NationalesGate(2);
gates[2] = new NationalesGate(3);
gates[3] = new NationalesGate(4);

// zwei kleine internationale Gates
gates[4] = new InternationalesGate(5, InternationalesGate.KLEIN);
gates[5] = new InternationalesGate(6, InternationalesGate.KLEIN);

// vier grosse internationale Gates
gates[6] = new InternationalesGate(7, InternationalesGate.GROSS);
gates[7] = new InternationalesGate(8, InternationalesGate.GROSS);
gates[8] = new InternationalesGate(9, InternationalesGate.GROSS);
gates[9] = new InternationalesGate(10, InternationalesGate.GROSS);
```

Eine weitere Stelle in der Klasse **Flughafen** muss angepasst werden: Wird ein passendes Gate gesucht, muss die Abfrage der Gategrösse auf internationale Gates eingeschränkt werden. Zu diesem Zweck erweitern Sie die Methode `landen()` um eine Zeile, in der die Grösse eines Gates abgefragt wird:

```
if (gates[i].getGroesse() == Gate.GROSS) {
```

... und ergänzen die obige Zeile wie folgt:

```
if (gates[i] instanceof InternationalesGate
    && ((InternationalesGate )gates[i]).getGroesse() == InternationalesGate.GROSS) {
```

Auf diese Weise darf die Grösse eines Gates nur dann abgefragt werden, wenn es sich um ein internationales Gate handelt.

13.5 Exception-Handling implementieren

Es kann nicht davon ausgegangen werden, dass auf dem Flughafen **immer freie Gates** vorhanden sind. Zudem ist es denkbar, dass die freien Gates für einen angemeldeten Flug ungeeignet sind. Der normale Programmablauf sieht daher wie folgt aus: Wenn ein Flug angemeldet wird, muss zuerst ein freies und passendes Gate gesucht werden. Erst danach kann der Flug am Gate angemeldet und das Flugzeug am Gate angedockt werden.

Wenn kein passendes Gate vorhanden ist, kann dieser Ablauf nicht durchgeführt werden. Dieser Fall wurde bisher durch eine **negative Gatenummer** signalisiert. Eine Gatenummer kann aber nicht negativ sein. Die Gatenummer wird insofern «missbraucht», als ihr zwei unterschiedliche Bedeutungen zugewiesen werden:

- Positiv: Es handelt sich um eine gültige Gatenummer.
- Negativ: Es handelt sich nicht um eine gültige Gatenummer, sondern um ein Problem.

Eine solche Vermischung der Bedeutung ist problematisch und sollte vermieden werden. Aus diesem Grund soll dieser Fall durch eine eigene **Exception** behandelt werden. Diese definieren Sie als «normale» Klasse, die von **java.lang.Exception** abgeleitet wird:

```
package ch.modul226.airport.model;

public class NoGateAvailableException extends Exception {

    public NoGateAvailableException() {
        super("SORRY - Kein freies Gate gefunden");
    }
}
```

In der obigen Klasse **Exception** hat es einen Konstruktor, der einen String entgegennimmt. Die Fehlerbehandlung wird mit einem **Defaultkonstruktor** versehen, der den Text `"SORRY - Kein freies Gate gefunden"` als Nachricht verwendet.

13.5.1 Exception erzeugen und werfen

Ändern Sie den Code der Methoden `landen()` und `freiesGateSuchen()` in der Klasse **Flughafen** folgendermassen ab:

```
public int landen(Flug flug) throws NoGateAvailableException {
    Gate gate = freiesGateSuchen(flug);
    gate.landen(flug);
    return gate.getGateNummer();
}

private Gate freiesGateSuchen(Flug flug) throws NoGateAvailableException {
    for (short i = 0; i < gates.length; i++) {
        if (gates[i].istFrei()) {
            if (flug.istNational()) {
                return gates[i];
            }
            else {
                if (gates[i].getGroesse() == InternationalesGate.GROSS
                        || .flug.getFlugzeug().istGrossraum()) {
                    return gates[i];
                }
            }
        }
    }
    throw new NoGateAvailableException();
}
```

Schauen Sie die letzte Zeile genauer an. Hier wird mit `new` regulär ein neues Exception-Objekt erzeugt. Dabei wird der Defaultkonstruktor verwendet. Mit der Anweisung `throw` wird die erzeugte Exception geworfen. Der Code selbst ist recht einfach. Es wird die Liste aller Gates durchgegangen und ein passendes freies Gate gesucht. Wenn kein Gate gefunden wird, wirft die letzte Zeile die Exception. Im Kopf der Methode `freiesGateSuchen()` wird mit `throws NoGateAvailableException` angezeigt, dass diese Methode möglicherweise eine `NoGateAvailableException` wirft. Java verlangt, dass Sie alle möglichen Exceptions hier angeben.

Betrachten Sie nun die Methode `landen()`. Hier wird `freiesGateSuchen()` aufgerufen. Allerdings finden Sie keinen Code für die Fehlerbehandlung. Obwohl eine NoGateAvailable-Exception auftreten kann, wird so getan, als ob immer alles korrekt abläuft. Soll sich doch die aufrufende Benutzerschnittstelle darum kümmern. Also deklarieren Sie im Kopf der Methode `landen()`, dass diese eine NoGateAvailable-Exception werfen kann.

Nun müssen Sie die Methode `landen()` nur noch im Hauptmenü anpassen. Auch diese wird dadurch einfacher und übersichtlicher:

```
private void landen() {
    System.out.println("\n\n\nLanden...\n\n");
    Flug flug = flugDatenBeschaffen();
    try {
        int gateNummer = flughafen.landen(flug);
       System.out.println("Maschine ist an Gate " + gateNummer + " gelandet");
    }
    catch(NoGateAvailableException ex) {
        System.out.println(ex.getMessage());
    }
}
```

Sie versuchen einfach einen Landevorgang. Kein Problembehandlungscode stört die Übersicht. Falls irgendwo innerhalb des try-Blocks etwas schiefgeht, wird eine Exception geworfen. Sie fangen diese ab und geben deren Meldungstext aus. Es kann hier nur eine NoGateAvailableException auftreten.

13.5.2 Standard-Exception abfangen

Es gibt eine weitere Stelle im Flughafenprojekt, an der Sie Exceptions gut einsetzen können: Wenn der Benutzer eine Gatenummer eingibt, um den Status dieses Gates abzufragen, muss die Nummer im gültigen Bereich liegen. Definieren Sie dazu eine separate Exception `InvalidGateNumberException` nach dem bereits bekannten Muster:

```
package ch.modul226.airport.model;

public class InvalidGateNumberException extends Exception {

    public InvalidGateNumber() {
        super("SORRY - Diese Gatenummer gibt es nicht");
    }
}
```

In der Klasse **Flughafen** gibt es die Methode `getGate()`. Ändern Sie diese wie folgt:

```
public Gate getGate(int gateNummer) throws InvalidGateNumberException{
    try {
        return gates[gateNummer];
    }
    catch (ArrayIndexOutOfBoundsException ex) {
        throw new InvalidGateNumberException();
    }
}
```

Das Programm versucht hier, mit der vom Benutzer eingegebenen Nummer auf den Gate-Array zuzugreifen. Solange der Benutzer alle Gates am Flughafen kennt und korrekt angibt, geht das gut. Wenn er aber eine Gatenummer wählt, die es nicht gibt, wird Java eine ArrayIndexOutOfBands-Exception werfen. Diese fangen Sie ab und werfen stattdessen die eigene Exception. In der Klasse **Hauptmenue** können Sie die Methode `gateDatenAnzeigen()` entsprechend anpassen:

```
private void gateDatenAnzeigen() {
    System.out.println("\n\n\nGate Daten anzeigen...\n\n");
    int gateNummer = ConsoleReader.readInteger("Gatenummer eingeben");
    try {
        Gate gate = flughafen.getGate(gateNummer);
        if (gate.istFrei()) {
            System.out.println("Gate ist frei\n\n\n");
            return;
        }
        if (gate instanceof NationalesGate) {
            System.out.println("Nationales Gate");
            System.out.println("---------------");
        }
        else {
            System.out.println("Internationales Gate");
            System.out.println("--------------------");
            System.out.println(
                gate.getGroesse() == InternationalesGate.GROSS
                ? "Grosses Gate"
                : "kleines Gate");
        }
        System.out.println("Flugnummer: " + gate.getFlug().getNummer());
        System.out.println("Flugzeugtyp"
            + (gate.getFlugzeug().getTyp() == Flugzeug.JET ? "Jet" : "prop"));

    }
    catch (InvalidGateNumberException ex) {
        System.out.println(ex.getMessage());
    }
}
```

Beim Aufruf von `flughafen.getGate()` tun Sie, als würde dies immer klappen. Falls einmal eine ungültige Gatenummer eingegeben wird, bricht Java den weiteren Programmablauf sofort ab und macht beim catch-Block weiter.

Bei der **Programmierung einer objektorientierten Anwendung** definieren Sie zunächst die benötigten **Klassen** und **Objekte,** indem Sie nach «Dingen» suchen, die in der Aufgabenstellung erwähnt werden und in unterschiedlichen Beziehungen zueinander stehen. Dabei lautet die zentrale Frage: «Wovon ist die Rede?»

Danach definieren Sie die **Eigenschaften** und **Verhaltensweisen** der ermittelten Klassen bzw. Objekte. Dabei versuchen Sie, die **Geschäftsklassen** und **Benutzerschnittstellen** getrennt zu halten. Dadurch kann eine textorientierte Benutzerschnittstelle bei Bedarf durch eine grafische Benutzerschnittstelle ersetzt werden. Mittels **Vererbung** und **Exception-Handling** kann der Programmcode vereinfacht und der Aufbau der Anwendung klarer gemacht werden. Dies ist vor allem im Hinblick auf die Pflege und Weiterentwicklung des Programms von grosser Bedeutung.

Repetitionsfragen

84 Stellen Sie zwei hilfreiche Fragen zum Auffinden von Klassen.

85 Wie unterscheiden sich in unserem Fallbeispiel nationale und internationale Flüge?

86 Beschreiben Sie folgenden Ausschnitt aus einem Klassendiagramm möglichst bündig.

Flughafen
gates: Liste von Gates name: String
landen(flug: Flug) starten(gateNummer: integer) gateStatus(gateNummer: integer)

87 Zur Klasse **Flug** gehört eine Methode, die aussagt, ob es sich um einen nationalen oder internationalen Flug handelt. Weshalb? Die entsprechende Information ist doch bereits in der Flugnummer ersichtlich. Ist diese Methode überhaupt nötig?

88 Welchen Datentyp verwenden Sie für die Liste der Gates eines Flughafens?

89 In der Klasse **Gate** finden Sie folgende Methode. Erklären Sie, was da vor sich geht.

```
public boolean istFrei() {
return this.flug == null;
}
```

90 Was würde passieren, wenn in der folgenden Methode aus der Klasse **Gate** die markierten Zeilen nicht vorhanden wären?

```
public Flugzeug getFlugzeug() {
if (this.flug == null)          <<<
    return null;                <<<
return flug.getFlugzeug();
}
```

91 Wie definieren Sie in Java eine Konstante?

92 Die Konstanten der Klasse **Gate** weisen zusätzlich das Schlüsselwort `static` aus. Was bedeutet dies und wie greifen Sie auf eine solche Konstante zu?

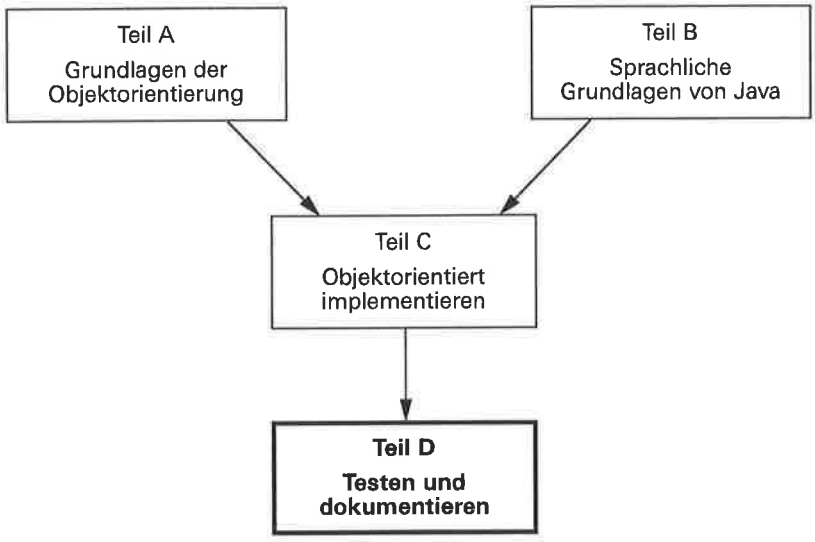
Teil D Implementation testen und dokumentieren

Einleitung, Lernziele und Schlüsselbegriffe

Einleitung

Dieser Teil des Lehrmittels zeigt Ihnen auf, wie Sie eine fertig entwickelte **OO-Anwendung systematisch testen und dokumentieren** können. In der folgenden Grafik sehen Sie, wo Sie sich innerhalb des Lehrmittels befinden:

Teil D im Gesamtzusammenhang



Lernziele und Lernschritte

	Lernziele	Lernschritte
☐	Sie können die grundlegenden Schritte erläutern, die bei einem Unit-Test durchlaufen werden müssen, und aufzeigen, welchen Beitrag diese zu einem qualitativ guten Ergebnis leisten.	• Testprojekt für JUnit erstellen • Testklasse Rechnertest erstellen • JUnit-Tests ausführen und bewerten
☐	Sie sind in der Lage, zu einer Klasse Testfälle und Grenzwerte festzulegen und diese in einer Testklasse zu implementieren.	• Testprojekt für JUnit erstellen • Testklasse Rechnertest erstellen • JUnit-Tests ausführen und bewerten
☐	Sie sind in der Lage, den Programmcode vollständig und korrekt zu dokumentieren, um daraus die API-Spezifikation abzuleiten.	• JavaDoc-Kommentare einfügen • Dokumentation automatisch erstellen

Schlüsselbegriffe

@param, @return, assert, assertEquals, Framework, JavaDoc, JUnit, TestCase, Unit-Test

14 Unit-Test mit JUnit

Nicht immer funktioniert ein Programm wie gewünscht. Ein Compiler findet zwar Syntaxfehler, doch logische Fehler zeigen sich erst dann, wenn das Programm läuft und sich nicht wie erwartet verhält. OO-Programme mit ihren zahlreichen, oft kleinen Klassen lassen sich sehr gut testen. Da Klassen idealerweise in sich abgeschlossen sind und über **Interfaces** angesprochen werden[1], können sie problemlos einzeln getestet werden. Eine korrekt angelegte Klasse verhält sich richtig. Wenn Sie an einer Klasse Änderungen vornehmen, müssen Sie sicherstellen, dass sich ihr Verhalten nicht geändert hat.

Klassen und Methoden sind die kleinsten Einheiten in einem OO-Programm. Wenn Sie auf dieser Ebene testen, spricht man von einem **Unit-Test.** Für die Durchführung von Unit-Tests im Rahmen der OOP mit Java ist das **Klassenframework**[2] **JUnit** verfügbar. In einer Installation von **Eclipse** ist JUnit bereits enthalten. Nachfolgend wird der Umgang mit JUnit anhand eines einfachen Testprojekts vorgestellt.

14.1 Testprojekt für JUnit erstellen

Sie erstellen in Eclipse ein neues Projekt und nennen es JUnitTest. Darin definieren Sie das Package **modul226.rechner** und darin wiederum die Klasse **TemperaturRechner.** Fügen Sie danach folgenden Code in die Klasse ein:

```
package modul226.rechner;

public class TemperaturRechner {

    public static void main(String[] args) {
        TemperaturRechner rechner = new TemperaturRechner();
        System.out.println(rechner.convert(37.5));

    }

    public double convert(double celsius) {
        double ergebnis = (celsius * 9 / 5) + 32;
        return ergebnis;
    }
}
```

Wenn Sie dieses Miniprogramm laufen lassen, sollten Sie das Ergebnis 99.5 erhalten. Das Resultat entspricht dem Wert in Fahrenheit von 37.5 Grad Celsius.

[1] Via setter()- und getter()-Methoden sowie über Nachrichten.

[2] Ein Klassenframework ist eine Sammlung von Klassen, die zusammen eine oder mehrere klar definierte Aufgabe(n) erledigen. Ein Klassenframework ist also mehr als eine Klassenbibliothek.

14.2 Testklasse Rechnertest erstellen

Sie wollen nun für die convert()-Methode einen JUnit-Test schreiben. Erstellen Sie dazu im Projekt **JUnitTest** ein Package **testing** und darin folgende Klasse:

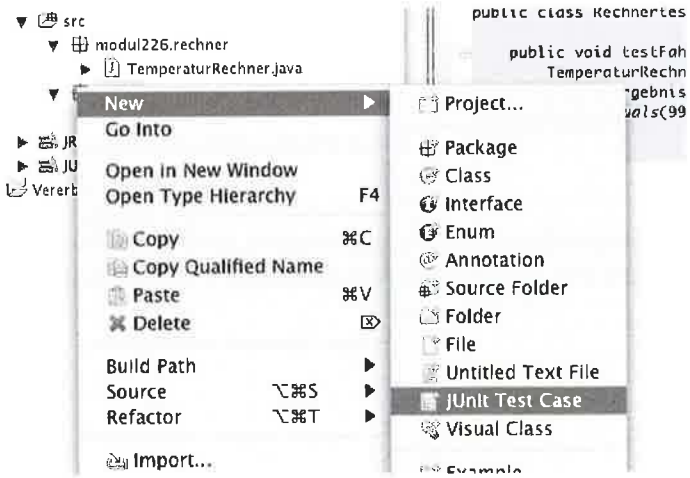
```
package testing;

import modul226.rechner.TemperaturRechner;
import junit.framework.TestCase;

public class Rechnertest extends TestCase {

    public void testConvert() {
        TemperaturRechner rechner = new TemperaturRechner();
        double ergebnis = rechner.convert(37.5);
        assertEquals(99.5, ergebnis, 0.0001);
    }
}
```

Wenn Sie mit JUnit testen möchten, müssen Sie für jeden **Testfall** eine eigene **Methode** erstellen. Diese Methoden müssen in einer Klasse sein, die Sie von **junit.framework.TestCase** ableiten. Das JUnit-Framework sucht dann in dieser Klasse alle Methoden, deren Namen mit `test` beginnt. Aus diesem Grund müssen Sie alle Testfälle nach folgendem Schema benennen:

```
public void testXyz()
```

Für den Platzhalter `Xyz` setzen Sie jeweils einen aussagekräftigen Namen ein. Wenn Sie mit Eclipse arbeiten, steht Ihnen ein **Wizard**[1] zur Verfügung, mit dem Sie solche Testklassen erstellen können:

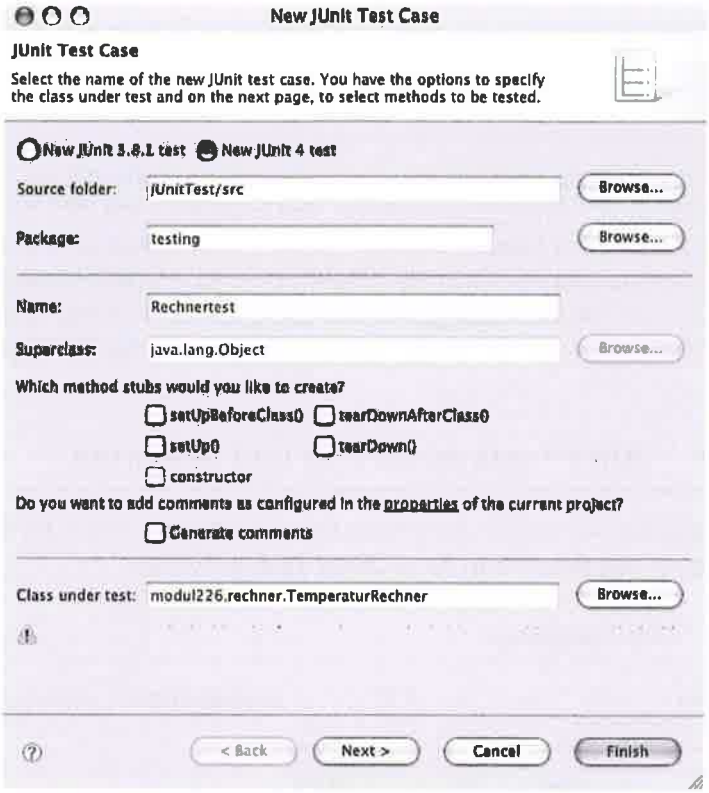
[14-1] Wizard für die Erstellung von Testklassen starten



[1] Engl. für: Zauberer, Assistent.

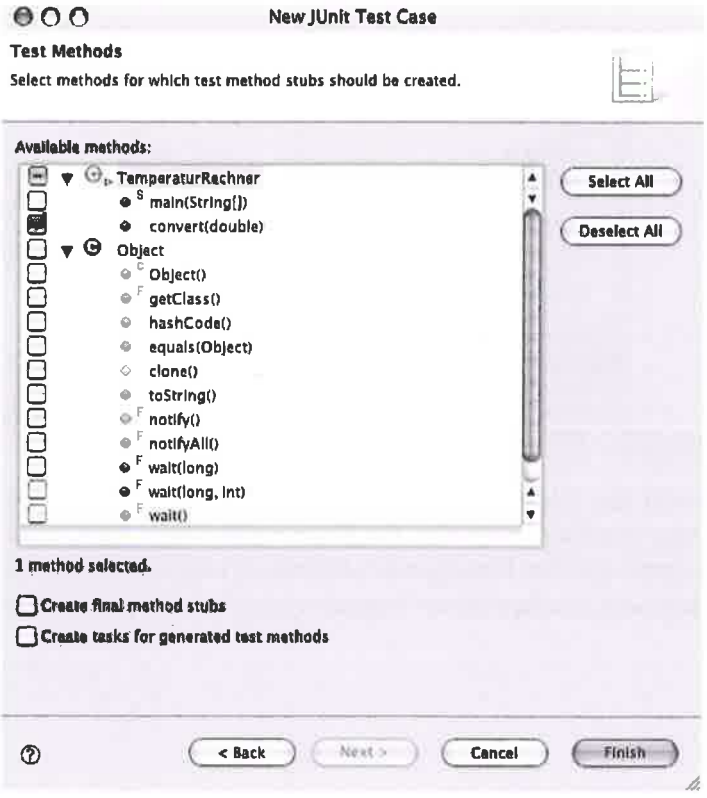
Nach dem Start des Wizard werden Angaben zum neuen Testfall erwartet:

[14-2] Neuen Testfall erstellen



Nach einem Klick auf **«Next >»** erscheint folgendes Dialogfenster:

[14-3] Testfallmethode bestimmen



Hier können Sie alle benötigten **Testfallmethoden** erstellen. Danach müssen Sie nur noch den **Testcode** in die vorbereitete Methode `testConvert()` einfügen. Vergleichen Sie dazu den folgenden Beispielcode:

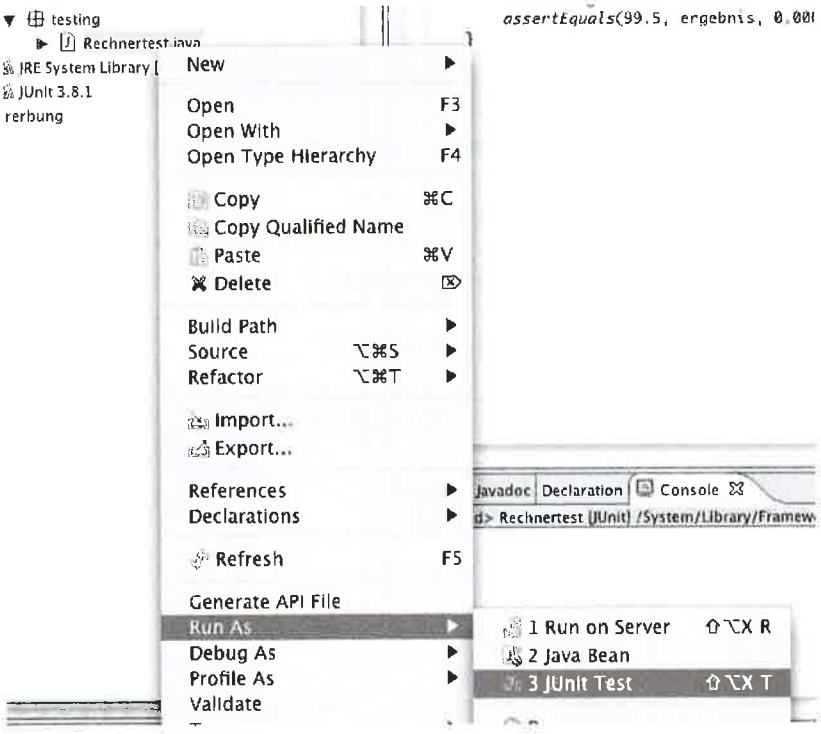
```
public void testConvert() {
    TemperaturRechner rechner = new TemperaturRechner();
    double ergebnis = rechner.convert(37.5);
    assertEquals(99.5, ergebnis, 0.0001);
}
```

Im obigen Beispiel erstellen Sie einen neuen Temperaturrechner und lassen diesen 37.5 Grad Celsius in Fahrenheit umrechnen. Das Ergebnis sollte 99.5 betragen. Weil es sich um einen double-Wert handelt, können interne Rundungsfehler auftreten, sodass evtl. nicht genau 99.5, sondern vielleicht 99.4999999 oder 99.5000001 herauskommen. Die Methode `assert()` ist von **TestCase** geerbt und prüft, ob das erwartete und das effektive Ergebnis übereinstimmen. Der dritte Parameter dient dazu, eine Rundungsgenauigkeit anzugeben.

14.3 JUnit-Tests ausführen und bewerten

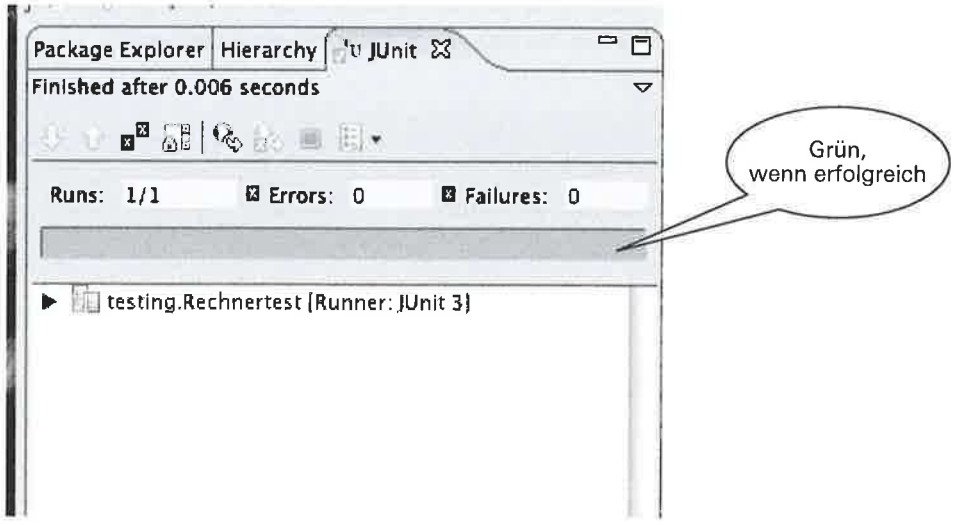
In Eclipse können Sie die JUnit-Tests sofort ausführen, indem Sie im Kontextmenü für die Testklasse den Befehl **Run As -> JUnit Test** auswählen.

[14-4] Unit-Test starten



Nun sucht das JUnit-Framework alle Methoden in der Rechnertestklasse, die mit `test` beginnen, und führt diese aus. Die Tests werden vom Framework jeweils mit `assert()` ausgewertet und das Testergebnis am Schluss in einem Eclipse-View grafisch dargestellt. Ein erfolgreich durchgeführter Testlauf wird mit einem **grünen Balken** dargestellt:

[14-5] Unit-Test ohne Fehler (grüner Balken)

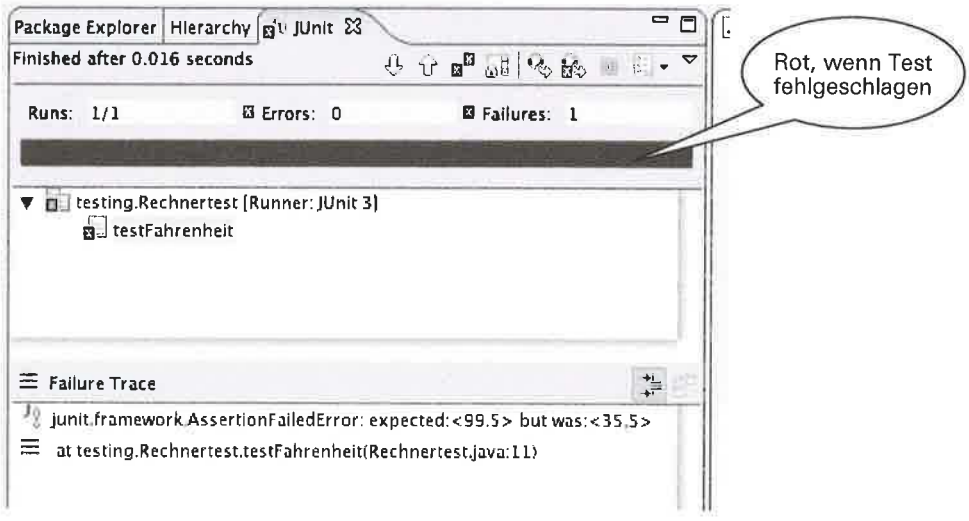


Sie möchten nun im Temperaturumrechner einen Fehler einbauen. Dazu ändern Sie die Formel folgendermassen ab:

```
public double convert(double celsius) {
    double ergebnis = (celsius * 9 / 5) - 32;
    return ergebnis;
}
```

Beachten Sie das Minus-Zeichen bei 32. Richtig wäre +. Diese Berechnung wird mit Sicherheit nicht das korrekte Ergebnis liefern. Führen Sie nun den Test ein zweites Mal durch, indem Sie erneut den Befehl **Run As -> JUnit Test** anklicken. Beobachten Sie, was nun passiert.

[14-6] Unit-Test fehlgeschlagen (roter Balken)



Ein fehlerhafter Testlauf wird mit einem **roten Balken** dargestellt. Im oberen Bereich des Dialogfensters sehen Sie, welcher Test durchgeführt wurde, während im unteren Bereich aufgezeigt wird, warum der Test fehlgeschlagen ist. Hier wurde der Wert 99.5 erwartet, der Test hat aber als Ergebnis 35.5 geliefert. Das Ergebnis ist also falsch und der Programmcode muss einen Fehler enthalten.

Hinweis

▷ Weiterführende Informationen über JUnit finden Sie auf der entsprechenden Homepage. Vergleichen Sie dazu auch das Linkverzeichnis auf S. 11.

Unit-Tests erlauben es Ihnen, die kleinsten Einheiten eines objektorientierten Programms, Klassen und Methoden, zu testen. Für professionelle Tests steht ihnen dazu das Java-Framework **JUnit** zur Verfügung. In Eclipse ist die Unterstützung für JUnit bereits eingebaut.

Repetitionsfragen

93	Was ist ein Unit-Test?
94	Wenn Sie mit JUnit arbeiten wollen, müssen Sie eine spezielle Klasse aus dem JUnit-Framework verwenden. Welche?
95	Gibt es eine Namenskonvention für Tests?

15 Implementation dokumentieren

Ein fertiggestelltes Programm muss systematisch dokumentiert werden, damit es ggf. auch von anderen Personen gepflegt und weiterentwickelt werden kann. Bei Java hält sich der Aufwand dafür in Grenzen, da die gesamte **Dokumentation direkt im Java-Code erfasst und automatisch aufbereitet** werden kann. In diesem Kapitel erfahren Sie, wie dies gemacht werden kann.

15.1 JavaDoc-Kommentare einfügen

Sie haben die **Schreibweise für Kommentare in Java** bereits kennengelernt:

- **Zeilenkommentar:** `// dies ist ein Zeilenkommentar`
- **Blockkommentar:** `/* dies ist ein Blockkommentar */`

Hier lernen Sie eine weitere Schreibweise kennen, um Kommentare in den Programmcode einzufügen. Der sogenannte **JavaDoc-Kommentar** entspricht einem Blockkommentar, beginnt aber mit einem zweiten Stern. JavaDoc-Kommentare werden bei der Erstellung einer **Programmdokumentation** automatisch berücksichtigt und können um **spezielle Markierungen** ergänzt werden, beispielsweise, um Zwischentitel zu kennzeichnen.

Sehen Sie sich dazu das folgende Beispiel an.

```
package ch.modul226.airport.model;

/**
 * Diese Klasse verwaltet einen einfachen Flughafen mit verschiedenen Gates
 *
 * @author ruggiero
 * @version V1.0
 */
public class Flughafen {

    /**
     * Die Liste der Gates
     */
    private Gate[] gates;

    /**
     * Name des Flughafens
     */
    private String name;

    /**
     * Defaultkonstruktor für den Flughafen
     */
    public Flughafen() {
        this.name = "Gatwick";
        this.gates = new Gate[10];

        // vier nationale Gates
        gates[0] = new NationalesGate(1);
        gates[1] = new NationalesGate(2);
        gates[2] = new NationalesGate(3);
        gates[3] = new NationalesGate(4);

        // 2 kleine internationale Gates
        gates[4] = new InternationalesGate(5, InternationalesGate.KLEIN);
        gates[5] = new InternationalesGate(6, InternationalesGate.KLEIN);

        // vier grosse internationale Gates
        gates[6] = new InternationalesGate(7, InternationalesGate.GROSS);
        gates[7] = new InternationalesGate(8, InternationalesGate.GROSS);
        gates[8] = new InternationalesGate(9, InternationalesGate.GROSS);
        gates[9] = new InternationalesGate(10, InternationalesGate.GROSS);
```

```
}

/**
 * Menueaktion landen
 *
 * @param flug der Flug, der sich zur Landung angemeldet hat
 * @return Nummer des Gates, an welchem der Flug angedockt hat
 * @throws NoGateAvailableException
 */
public int landen(Flug flug) throws NoGateAvailableException {
    Gate gate = freiesGateSuchen(flug);
    gate.landen(flug);
    return gate.getGateNummer();
}

/**
 * Sucht für den gemeldeten Flug ein passendes freies Gate
 *
 * Die Methode geht einfach die Liste der Gates durch und gibt das
 * erste freie passende Gate zurueck. Es kann daher sein, dass ein kleines
 * Flugzeug an einem grossen Gate andockt, weil dieses in der Liste zuerst
 * gefunden wird, obwohl auch ein freies kleines Gate vorhanden waere
 *
 * @param flug der zu landende Flug
 * @return das gefundene passende Gate
 * @throws NoGateAvailableException
 */
private Gate freiesGateSuchen(Flug flug) throws NoGateAvailableException{
    for (short i = 0; i < gates.length; i++) {
        if (gates[i].istFrei()) {
            if (flug.istNational()) {
```

Im obigen Beispiel sehen Sie vor der Klassendeklaration zuoberst einen Kommentar für die ganze Klasse. Weitere Kommentare stehen jeweils direkt vor den Variablen sowie vor allen Methoden.

Eclipse hilft Ihnen auch bei der Erfassung eines JavaDoc-Kommentars. Wenn Sie einen Kommentar mit `/**<enter>` beginnen, bekommen Sie ein komplettes Raster für den Kommentar, passend zur Stelle im Programmcode. Wenn Sie innerhalb eines solchen Kommentarblocks eine neue Zeile beginnen, wird am Anfang automatisch der korrekte Abstand und Stern für Kommentare eingefügt.

Innerhalb eines JavaDoc-Kommentars können Sie einfache **HTML-Auszeichnungen** wie `<b>...</b>` usw. verwenden. JavaDoc kennt zudem einige spezielle **Anweisungen**. Hier eine Übersicht der wichtigsten Anweisungen:

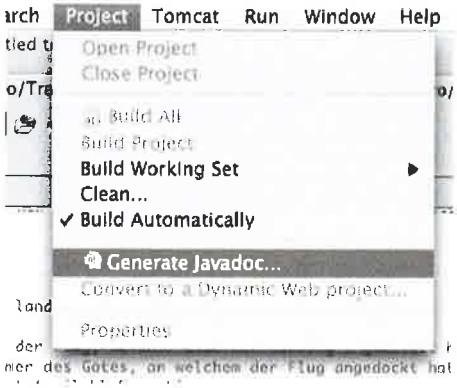
Anweisung	Bedeutung
`@author`	Name des Autors
`@see`	Verweis «siehe auch ...»
`@version`	Version
`@since`	Angabe, seit welcher Version diese Klasse/Methode/Variable in der Software enthalten ist
`@param`	Eingabeparameter bei einer Methode
`@return`	Beschreibung des Rückgabewerts
`@throws`	Beschreibung allfälliger Exceptions

15.2 Java-Dokumentation automatisch erstellen

Das **Java Development Kit (JDK)** beinhaltet ein Tool, das alle JavaDoc-Kommentare aus einem Projekt herauslesen und zu einer **HTML-Dokumentation** aufbereiten kann.

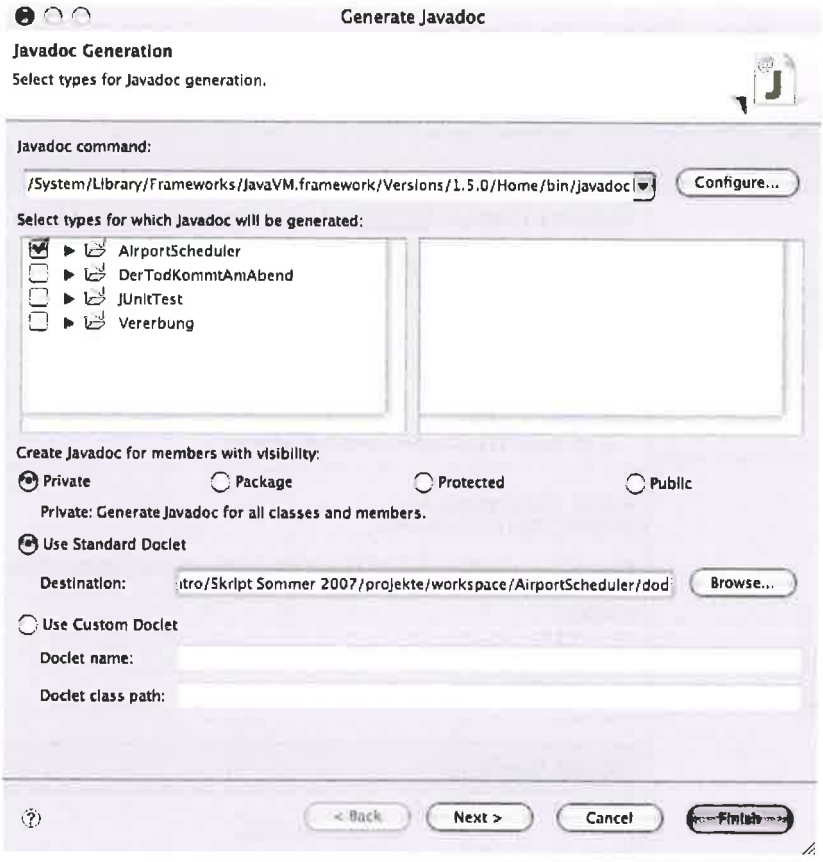
Eclipse integriert diese Funktionalität, die Sie im **Projektmenü** über den Befehl **JavaDoc erstellen** anwählen können:

[15-1] JavaDoc erstellen



Eclipse zeigt nun das folgende Dialogfenster an:

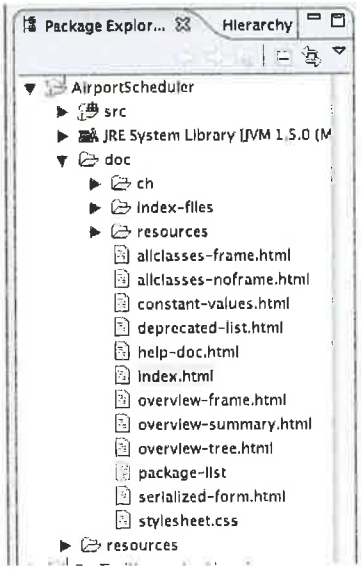
[15-2] JavaDoc definieren



Hier können Sie bestimmen, welche **Elemente in die Dokumentation aufgenommen** werden sollen. In unserem Fall sollen auch Elemente dokumentiert werden, die `private` sind.

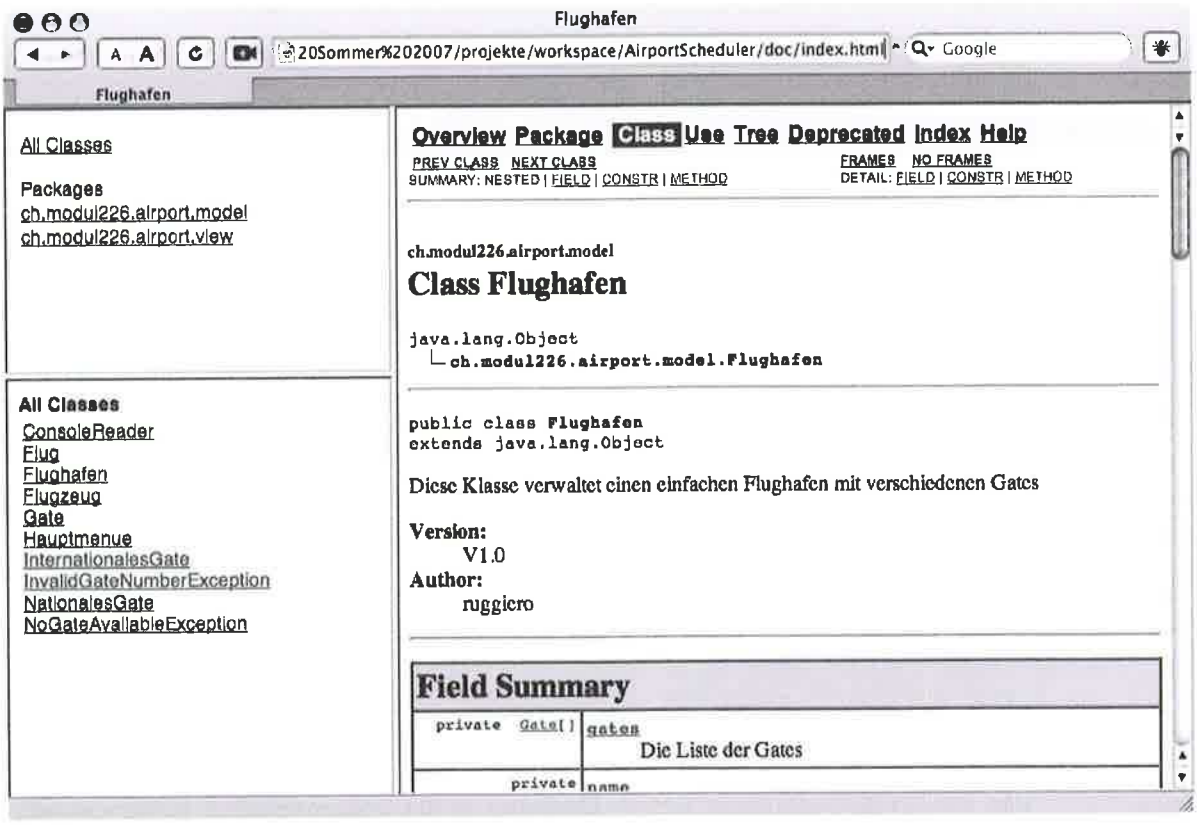
Weiter müssen Sie angeben, wo die Java-Dokumente abgelegt werden. In unserem Fall haben Sie im Projektordner einen Unterordner **doc** angelegt und geben diesen als Ziel an. Nach einem kurzen Moment finden Sie im doc-Verzeichnis die generierte Dokumentation:

[15-3] Verzeichnis für die Dokumentation festlegen



Wenn Sie die Datei **index.html** öffnen, haben Sie eine umfassende Dokumentation Ihres Projekts vorliegen.

[15-4] Fertige Dokumentation



Standardmässig verwendet JavaDoc die gleichen Templates wie die Java-Dokumentation von Oracle. Sie können aber auch eigene **Templates** erstellen (z. B. mit einem Firmenlogo) und diese verwenden.

Die nächste Abbildung verdeutlicht, wie Sie mithilfe der JavaDoc-Anweisungen `@param`, `@return` und `@throw` die **Formatierung der Dokumente** steuern können:

[15-5] Detail aus der Dokumentation

Method Detail

landen

```
public int landen(ch.modul226.airport.model.Flug flug)
           throws ch.modul226.airport.model.NoGateAvailableException
```

Menuaktion landen

Parameters:
`flug` - der Flug, welcher sich zur Landung angemeldet hat
Returns:
Nummer des Gates, an welchem der Flug angedockt hat
Throws:
`ch.modul226.airport.model.NoGateAvailableException`

---

freiesGateSuchen

```
private ch.modul226.airport.model.Gate freiesGateSuchen(ch.modul226.airport.model.Flug flug)
                                                 throws ch.modul226.airport.model.NoGateAvailableException
```

Sucht fuer den gemeldeten Flug ein passendes freies Gate Die Methode geht einfach die Liste der Gates durch und gibt das erste freie passende Gate zurueck. Es kann daher sein, dass ein kleines Flugzeug an einem grossen Gate andockt, weil dieses in der Liste zuerst gefunden wird, obwohl auch ein freies kleines Gate vorhanden waere

Parameters:
`flug` - der zu landende Flug
Returns:
das gefundene passende Gate
Throws:
`ch.modul226.airport.model.NoGateAvailableException`

Ein fertiggestelltes Programm muss systematisch dokumentiert werden, damit es gepflegt und weiterentwickelt werden kann. Bei Java hält sich der Aufwand dafür in Grenzen, da sich die gesamte **Dokumentation im Java-Code erfassen** und **automatisch aufbereiten** lässt.

Mithilfe einfacher **Tags und Markierungen** lassen sich **Kommentare** direkt im Quellcode kennzeichnen und formatieren. **JDK** beinhaltet zudem ein Werkzeug **Javadoc,** das solche Kommentare auslesen und daraus automatisch eine umfassende Programmdokumentation erstellen kann.

Repetitionsfragen

96 Wie lautet das JavaDoc-Kommentarzeichen?

97 Nennen Sie die wichtigsten beiden JavaDoc-Schlüsselwörter, die für die Dokumentation einer Methode gebraucht werden.

98 Wo fügen Sie im Quelltext JavaDoc-Kommentare ein?

99 Wozu verwenden Sie das Tag `@throws`?

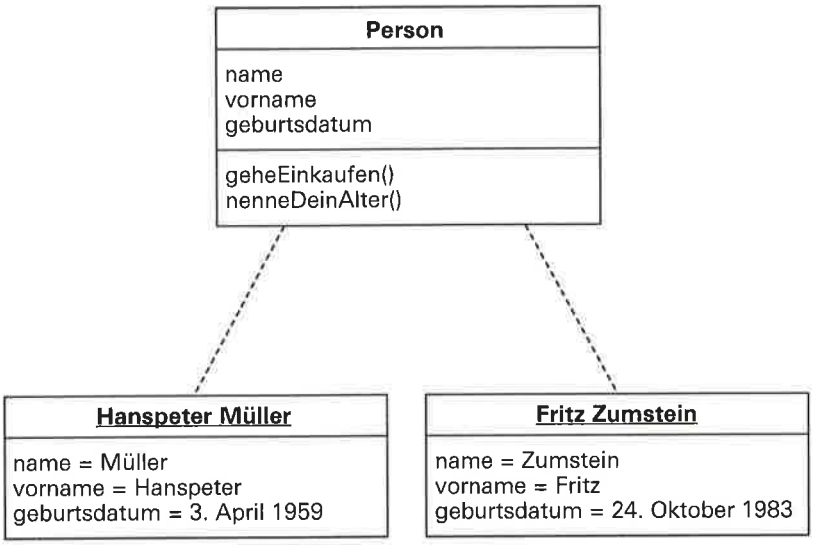
Teil E Anhang

Gesamtzusammenfassung

Java ist eine objektorientierte Programmiersprache, die auf fast allen Plattformen läuft. Der **Java-Compiler** javac übersetzt den Quellcode in einen sogenannten Byte-Code, der von der virtuellen Java-Maschine interpretiert wird. Die **Syntax von Java** ist fast identisch mit derjenigen von C. Java kennt alle Standard-Datentypen von C und verwendet dieselben Ablauf- und Kontrollstrukturen wie if/else, switch/case oder Schleifen. Aus diesen Gründen ist es für C-Programmierer relativ einfach, Java zu erlernen.

Objekte stellen Dinge aus der realen Welt dar. Sie haben bestimmbare Eigenschaften, zeigen ein typisches Verhalten und lassen sich in einem objektorientierten Entwurf abbilden. Objekte mit gleichen Eigenschaften und gleichem Verhalten werden zu **Klassen** zusammengefasst.

Klassen und Objekte



In diesem Diagramm gibt es eine Klasse **Person.** Jede Person hat einen Namen, einen Vornamen und ein Geburtsdatum. Dies sind die **Eigenschaften** von Personen. Personen können auch etwas tun, sie haben ein Verhalten und reagieren auf Anweisungen. Die Personen in der obigen Grafik können beispielsweise einkaufen gehen und ihr Alter nennen. Hanspeter Müller und Fritz Zumstein sind konkrete Vertreter der Klasse **Person.** Es handelt sich um sogenannte **Objekte.** Objekte sind Instanzen von Klassen.

In der realen Welt hat man es immer mit Objekten zu tun, die interagieren. Die objektorientierte Programmierung versucht, diese Mechanismen abzubilden. Dabei gibt es nicht nur einfache Klassen, sondern ganze **Klassenhierarchien.** Denken Sie nur an die Hierarchie in der Zoologie. Tiere unterteilt man in Wirbeltiere und Wirbellose. Die Wirbeltiere kann man weiter unterteilen in Säuger und eierlegende und die eierlegenden wiederum in Vögel, Fische und so weiter. Zwischen diesen Tiergruppierungen besteht eine spezielle Beziehung: Ein Säuger ist ein Wirbeltier und ein Wirbeltier ist ein Tier. Man spricht in diesem Zusammenhang von einer **Spezialisierung** bzw. **Generalisierung.** Eine «Ist-ein-Beziehung» kann in der objektorientierten Programmierung als **Vererbung** implementiert werden. Dabei werden Klassen in Java folgendermassen definiert:

```
public class Huhn extends Vogel {
    ...
}
```

Im obigen Beispiel wird das Schlüsselwort `extends` für die Angabe der Vererbung verwendet.

Ein weiteres wichtiges Konzept in der OO-Programmierung ist die **Datenkapselung** (Data Hiding). Diese sorgt dafür, dass das «Innenleben» einer Klasse von der «Aussenwelt» losgelöst wird. Indem der Zugriff auf die Attribute und Eigenschaften von Objekten über eine definierte Schnittstelle erfolgt, kann eine einmal erstellte Klasse intern beliebig geändert werden, ohne dass andere Klassen in ihrer Zusammenarbeit gestört werden. Objekte einer Klasse müssen ja nicht wissen, wie andere Objekte anderer Klassen intern aufgebaut sind. Wenn eine Schnittstelle bekannt und gleichbleibend ist, lassen sich verschiedene Bereiche einer Anwendung voneinander trennen. So lassen sich getestete und funktionierende Anwendungsteile bei Änderungen in anderen Anwendungsteilen weiterverwenden, ohne dass Sie diese neu testen müssen. Auch Programmierfehler lassen sich auf diese Weise stark einschränken und einfacher beheben.

In Java wird Datenkapselung durch die Schlüsselwörter `public`, `protected` und `private` sowohl für Methoden als auch für Attribute realisiert. Es wird empfohlen, Attribute `private` zu halten und mit geeigneten Zugriffsmethoden zu versehen. Ein weiterer Mechanismus in der objektorientierten Programmierung ist die Ausnahmebehandlung. Während eines Programmlaufs können immer wieder mehr oder weniger unvorhergesehene Probleme auftreten, die eine Weiterführung des Programmablaufs verhindern. Ein solches Problem könnte die Division durch null sein. In der klassischen prozeduralen Programmierung ist Code, der solche Ausnahmesituationen prüft, mit dem Code für den Normalfall eng verwoben. In der OO-Programmierung trennt man die beiden Zweige. Der Normalfall wird durchgespielt, als ob es keine Probleme geben könnte. Sollte dennoch im Programmablauf ein Problem auftreten, wird ein Ausnahmeobjekt erzeugt. Diese **«Exception»** kann an geeigneter Stelle abgefangen werden. Die **Java-Laufzeitumgebung** sorgt automatisch dafür, dass der angefangene Normalablauf abgebrochen wird. In Java wird das folgendermassen programmiert:

```
try {
    anweisung_1;
    anweisung_2;
    ...
    anweisung_n;
}
catch (Exception ex) {
    // informiere den Benutzer, korrigiere das Problem oder was auch immer
}
```

Im **try-Block** wird der vollständige Programmablauf programmiert, ohne auf mögliche Probleme Rücksicht zu nehmen. Wenn in einer Anweisung trotzdem ein Problem auftaucht, wird eine Exception geworfen, die dafür sorgt, dass der Ablauf sofort abgebrochen wird. Die geworfene Ausnahme kann im **catch-Block** gefangen werden. Hier kann das Programm auf das Problem reagieren und z. B. den Benutzer benachrichtigen oder einen Defaultwert irgendwo einsetzen. Exceptions können pauschal abgefangen werden (wie im Beispiel gezeigt) oder es werden nur gezielt einzelne Ausnahmen gefangen. Java kennt viele vordefinierte Exceptions, es ist jedoch auch problemlos möglich, eigene Exceptions zu definieren, zu werfen und zu fangen. Damit lassen sich auf elegante Art und Weise Probleme während der Laufzeit eines Programms behandeln.