Ralph Steyer

erfolgreich Java 7 programmieren

- Der grundlegende Einstieg in Java und die objektorientierte Programmierung
- Von einfachen Java-Beispielen bis zu fortgeschrittenen Java-Applikationen
- Inklusive der Behandlung wichtiger Tools im Java-Umfeld (JDK, Eclipse, NetBeans) und des Java-Standard-API

ALWAYS LEARNING PEARSON

erfolgreich Java 7 programmieren

Ralph Steyer

erfolgreich Java 7 programmieren

ADDISON-WESLEY

An imprint of Pearson

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.d-nb.de* abrufbar.

Die Informationen in diesem Buch werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht. Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt. Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter Sorgfalt vorgegangen. Trotzdem können Fehler nicht ausgeschlossen werden.
Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen. Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien. Die gewerbliche Nutzung der in diesem Produkt gezeigten Modelle und Arbeiten ist nicht zulässig.

Fast alle Hardware- und Softwarebezeichnungen und weitere Stichworte und sonstige Angaben, die in diesem Buch verwendet werden, sind als eingetragene Marken geschützt. Da es nicht möglich ist, in allen Fällen zeitnah zu ermitteln, ob ein Markenschutz besteht, wird das ® Symbol in diesem Buch nicht verwendet.

10 9 8 7 6 5 4 3 2 1

14 13 12

ISBN 978-3-8273-2658-4

© 2012 by Addison-Wesley Verlag,
ein Imprint der Pearson Deutschland GmbH,
Martin-Kollar-Straße 10–12, D-81829 München/Germany
Alle Rechte vorbehalten
Lektorat: Brigitte Bauer-Schiewek, bbauer@pearson.de
Anne Herklotz, aherklotz@pearson.de
Fachlektorat: Dirk Frischalowski
Korrektorat: Petra Kienle
Herstellung: Martha Kürzl-Harrison, mkuerzl@pearson.de
Coverkonzeption und -gestaltung: Thomas Arlt, tarlt@adesso21.net
Satz: Reemers Publishing Services GmbH, Krefeld, www.reemers.de
Druck und Verarbeitung: Drukarnia Dimograf, Bielsko-Biala

Printed in Poland

3 Einführung in die objektorientierte Programmierung mit Java

Java ist eine objektorientierte Programmiersprache und ohne Verständnis der grundlegenden Idee objektorientierter Programmierung kommen Sie in Java nicht zurecht. Dieses Kapitel führt Sie in die fundamentalen theoretischen Konzepte, Ideen und Methoden der objektorientierten Denkweise und vor allen Dingen deren konkrete Umsetzung in Java ein. Dabei besteht in jedem Buch zu Java das Dilemma, dass ein Programmierer für konkrete Praxis ebenfalls Kenntnisse über die Syntax einer Sprache benötigt. Manche Java-Bücher beginnen deshalb mit der Syntax statt der OO-Konzepte. Ich werte allerdings das Verständnis des OO-Konzepts als die elementare Basis für einen Einstieg höher und möchte damit beginnen. Das ist auch aus meiner eigenen Historie beim Lernen von Java motiviert, denn ich bin aus der prozeduralen auf die objektorientierte Programmierung umgestiegen und habe mich mit deren Konzepten lange schwer getan. Mit den syntaktischen Strukturen hingegen kam ich sofort zurecht, da diese in prozeduralen Sprachen wie Pascal oder C ähnlich bis gleich waren. Und Java entspricht von der Syntax her fast allen Sprachen, die zur C-Sprachfamilie zählen (etwa C, Perl, JavaScript, PHP etc.).

Info

Die Syntax mit Details wie Datentypen und Variablen, Operatoren, Literalen, verschiedenen Anweisungstypen (Kontrollflussanweisungen und Schleifen) etc. folgt im nächsten Kapitel, denn für ein voll funktionales Programm ist sie unabdingbar. In den Beispielen in dem Kapitel wird die Syntax nur benutzt, damit wir überhaupt etwas praktisch vorführen können. Allerdings werden die Beispiele grundsätzlich von der Syntax her einfach sein und nur die OO-Konzepte anschaulich verdeutlichen.

Auf den folgenden Seiten erfahren Sie also Genaues zum grundsätzlichen Programmaufbau bei Java (Klassenbildung) und zur praktischen Erzeugung von Objekten, zum Umgang mit Vererbung und Assoziationen sowie Modifizierern etc. und warum sich OOP durchgesetzt hat und wie sie sich von klassischer Programmierung abgrenzt.

3.1 Objektorientierte Programmiersprachen

Die Geschichte der Programmierung reicht von den theoretischen Ansätzen bis zum Beginn des 19. Jahrhunderts zurück. Allerdings begann die praktische Umsetzung in der heute darunter verstandenen Form erst nach den Arbeiten von Konrad Zuse und der Von-Neumann-Maschine um 1945. Seit dieser Zeit wurden eine große Anzahl verschiedener Programmiersprachen entwickelt, die sich indessen vielfach bestimmten Sprachfamilien zuordnen lassen. So werden die heute wichtigsten Programmiersprachen von der Syntax her meist zur C-Familie gezählt, die wiederum auf eine Sprache mit Namen Algol zurückgeht. Programmiersprachen werden aber nicht (nur) bezüglich der Syntax klassifiziert, sondern ebenfalls historisch in verschiedene Generationen eingeteilt:

1. Die historisch erste Generation ist die **Maschinensprache** mit all ihren Facetten. Maschinensprache ist wie besprochen explizit plattformabhängig bis hinunter auf die Prozessorebene.
2. Als zweite Generation wird **Assembler** gesehen. Assembler verwendet anstelle von numerischen Binärcodes symbolische Bezeichner (Mnemonic) für Anweisungen, die in genau einen Maschinenbefehl umgesetzt werden. Auch Assembler ist wie erwähnt plattformabhängig.
3. Mit der dritten Generation beginnen die **höheren Programmiersprachen**. Diese unterstützen erstmals Algorithmen und die Verwendung von Schlüsselwörtern, die der englischen Sprache entliehen sind und in Klartext Anweisungen formulieren. Höhere Programmiersprachen sind weitgehend anwendungsneutral und (im Quelltext) plattformunabhängig. Diese Generation der Programmiersprachen beginnt Mitte der fünfziger Jahre mit Fortran, Cobol und Algol-60. Später kamen unter anderem Pascal, Modula-2, C und Basic hinzu.
4. Die vierte Generation der Programmiersprachen bezeichnet **anwendungsbezogene (applikative) Sprachen**. Solche Programmiersprachen ergänzen Techniken der dritten Generation um Sprachmittel für relativ komplexe, anwendungsbezogene Operationen. Dies sind beispielsweise Zugriffe auf Datenbanken (zum Beispiel mit SQL) oder die Gestaltung von grafischen Benutzeroberflächen (GUI).
5. Als fünfte Generation der Programmiersprachen werden solche Sprachen gesehen, die das relativ neutrale Beschreiben von Sachverhalten und Problemen erlauben und den genauen Problemlösungsweg nicht exakt vorgeben. Im Rahmen der künstlichen Intelligenz setzt man diese Sprachen ein.

Wenn Sie sich nun die Sprachgenerationen ansehen, werden Sie sich vielleicht fragen, wo objektorientierte Sprachen einzusortieren sind? Die Antwort ist einfach – sie passen nicht in dieses Generationenmodell. Sie werden daher oft als eigenständige **OO-Generation** bezeichnet, die sich bis Anfang der 70er Jahre und teilweise noch weiter zurückverfolgen lässt. Sehr frühe Vertreter mit objektorientiertem Denkansatz sind Lisp oder Logo. In den 70er Jahren entstand als wichtigster Vertreter erster objektorientierter Spra-

chen Smalltalk, das sich in mehreren Zyklen entwickelte und als einer der geistigen Väter von Java gilt. In den 80er Jahren entstanden mit Objective C (ObjC), C++, Eiffel und ObjectPascal mehrere OO-Sprachen, die ein bestehendes, nicht objektorientiertes Konzept um objektorientierte Techniken erweiterten. Zum Beispiel wurde C um C++ erweitert und tritt seitdem meist als Paar auf (C/C++). Seit Anfang der 90er Jahre entstanden eine Reihe moderner, eigenständiger Programmiersprachen, die sich ausdrücklich als rein objektorientierte Programmiersprachen verstehen und mit prozeduralen Erblasten vollkommen brechen. Java zählt explizit dazu oder Sprachen unter dem .NET-Konzept von Microsoft wie C#. Diese Programmiersprachen unterstützen im Allgemeinen die wichtigsten Konzepte der objektorientierten Programmierung – allerdings in unterschiedlichem Maße.

3.1.1 Bessere Software-Qualität durch OOP

Das zentrale Problem bei der Entwicklung komplexer Software-Systeme ist, eine möglichst hohe Software-Qualität zu erreichen. Allgemein betrachtet man dabei sowohl die innere als auch die äußere Software-Qualität. Die innere Software-Qualität bezeichnet die Sicht des Entwicklers. OOP bietet durch die Möglichkeiten der Abstraktion, Hierarchiebildung, Kapselung von Interna, Wiederverwendbarkeit, Schnittstellenbildung und einige weitere Techniken hervorragende Ansätze zur Gewährleistung einer hohen inneren Software-Qualität. Die äußere Software-Qualität spiegelt die Sicht des Kunden wider. Dieser erwartet Dinge wie Korrektheit, Stabilität, Anwendungsfreundlichkeit oder Erweiterbarkeit einer Software. Ein Paradigma der OOP ist, dass eine hohe innere Software-Qualität zu einer hohen äußeren Software-Qualität führt. Und Java bestätigt dieses Paradigma eindrucksvoll.

3.1.2 Kernkonzepte der Objektorientierung

Als die theoretischen Kernkonzepte in der objektorientierten Software-Entwicklung werden in den meisten Abhandlungen folgende Punkte angeführt:

- **Objekte** dienen als Abstraktion eines realen Gegenstands oder Konzepts und verfügen über einen spezifischen Zustand und ein spezifisches Verhalten.
- **Klassen** dienen als Baupläne für Objekte. Sie repräsentieren die Objekttypen. Auf der anderen Seite fassen sie alle relevanten Eigenschaften und Verhaltensweisen der repräsentierten Objekttypen zusammen (sie klassifizieren diese).
- **Attribute** versteht man als Beschreibung objekt- und klassenbezogener Daten. Attribute sind die einzelnen Dinge, durch die sich ein Objekt von einem anderen unterscheidet. Man nennt Attribute meist die **Eigenschaften** eines Objekts.
- **Methoden** versteht man als Beschreibung der objekt- und klassenbezogenen **Funktionalität**.

- **Assoziationen**, **Kompositionen** und **Aggregationen** sind Mechanismen zum Ausdruck von Klassen- bzw. Objektbeziehungen, wobei diese Schlagwörter in Java allgemein und insbesondere in diesem Buch mehr theoretische, denn praktische Bedeutung haben. Eine Assoziation beschreibt in der Regel eine Beziehung zwischen zwei (meistens) oder mehr Klassen. Eine ganz besondere Assoziation, die wir konkret praktisch verwenden und nur unter dem Begriff weiter ausführen wollen, ist die **Vererbung**. Diese beschreibt einen Mechanismus zum Generalisieren und Spezialisieren von Objekttypen. In der objektorientierten Programmierung ist die Aggregation ebenso eine besondere Art der Assoziation – aber zwischen den Objekten selbst. Sie beschreibt eine sogenannte **schwache Beziehung** zwischen Objekten. Ein Objekt wird hier Teil eines anderen, ganzen Objekts gesehen. Es kann aber auch ohne das umgebende Objekt existieren. Im Fall einer Komposition ist ein Objekt auch Teil eines anderen, ganzen Objekts. Es kann jedoch nicht ohne das umgebende Objekt existieren.
- Die **Polymorphie** dient zur flexiblen Auswahl geeigneter Methoden[1] identischen Namens anhand der Argumentenliste bzw. der gesamten Methodenunterschrift.
- **Schnittstellen** beschreiben einen Mechanismen zur Strukturierung von Klassenbeziehungen. Sie definieren dazu eine Menge an Methoden, die eine implementierende (nichtabstrakte) Klasse bereitstellen muss.
- Über **abstrakte Klassen** kann man bestimmte Operationen in spezialisierten Klassen erzwingen.
- **Generische Klassen** dienen zur Darstellung von ganzen Klassenfamilien.

Wahrscheinlich werden diese Schlagwörter am Anfang für Sie kaum Bedeutung haben, aber wir werden diese mit Substanz füllen. Neben der Syntax sind diese Konzepte eben der wesentliche Teil dessen, was Sie zum erfolgreichen Lernen von Java beherrschen müssen und wir deshalb ins Zentrum des Buchs stellen. Zum großen Teil besprechen wir die Konzepte bereits in diesem Kapitel, zum Teil aber auch in den folgenden Kapiteln des Buchs.

3.1.3 Vertragsbasierte Programmierung

Nun soll Ihnen noch kurz theoretisch ein Verfahren vorgestellt werden, das über weite Strecken der OOP im Allgemeinen und Java im Besonderen zum Tragen kommt. In vielen Situationen werden sogenannte Verträge beziehungsweise Kontrakte zwischen dem Ersteller eines Codes und dessen Verwender geschlossen, entweder in eigenen Sprachkonstrukten oder durch geeignete Programmierung. Vertragsbasierte Programmierung oder Kontraktprogrammierung legt zwischen den aufrufenden und den aufgerufenen Stellen fest, welche Bedingungen vor der Verwendung und nach der Verwendung eines Codesegments gelten müssen. Bedingungen beziehen sich etwa

1 In manchen Sprachen hat man Polymorphie auch bei anderen Strukturen wie Operatoren – nicht aber in Java.

auf den Zustand des aufgerufenen Objekts, die übergebenen Parameter und die Ergebnisse. Insbesondere legt ein Kontrakt fest, welche Eingangs- und Ausgangsbedingungen bei einem Methodenaufruf sowie welche invarianten Eigenschaften im aufgerufenen Objekt gegeben sein müssen.

Nun wird Ihnen dieses Konzept vielleicht etwas abstrakt erscheinen. Aber ich möchte Sie insofern beruhigen und auf die konkreten praktischen Umsetzungen neugierig machen, dass diese Verträge die Programmierung in Java einfach, sicher und logisch machen. Und gerade in Java wird durch das Laufsystem und den Compiler streng auf die Einhaltung von Verträgen geachtet. Deshalb ist Java so logisch, sicher und stabil. Ein Programmierer, der eine bestimmte Verhaltensweise bei einem Objekt bereitstellen will, muss das garantieren und der Verwender kann zu 100% sicher sein, dass der Programmierer seine Versprechungen einhält.

Wir werden in der Folge meist erwähnen, wenn ein bestimmtes konkretes Verhalten Teil der vertragsbasierten Programmierung ist.

Info

3.2 Objekte und Klassen

In der OOP wird alles, was Sie jemals im Quelltext notieren, als Objekt beziehungsweise Klasse oder ein Objekt-/Klassenbestandteil (oder eine Spezialform davon) zu verstehen sein. Das gilt nicht für sogenannte **hybride Sprachen**, mit denen man zwar objektorientiert programmieren kann, die jedoch mit ihrer prozeduralen Vergangenheit nicht vollständig gebrochen haben. Als Beispiele seien C/C++, Delphi oder Visual Basic genannt. Dort finden Sie aufgrund der historischen Altlasten Techniken, die nicht in das OO-Konzept passen und die dort parallel zu OO-Techniken verwendet werden können. Sogenannte **objektbasierende Sprachen**[2] verwenden Objekte, realisieren aber die verbindlichen Konzepte der OOP nur teilweise. Aber versuchen wir erst einmal genauer zu klären, was Objekte sind.

3.2.1 Objekte

Lösen wir uns kurz von der Programmierung und betrachten die reale Welt. Objekte kommen ständig in unserer Umgebung vor. Ein Mensch bedient sich eines Objekts, um eine Aufgabe zu erledigen. Man weiß dabei in der Regel nicht genau, wie das Objekt im Inneren funktioniert, aber man kann es bedienen (weiß also um die verfügbaren Methoden, um es verwenden zu können) und weiß um die Eigenschaften des Objekts (seine Attribute) und wann welche Funktionalität zur Verfügung steht (sein Zustand). Die Reihe von Beispielen kann man beliebig lang erweitern, wir wollen es bei dem Auto, der Waschmaschine, einem Kugelschreiber oder der Kaffeemaschine belassen.

2 Beispielsweise JavaScript, das rein von der Syntax mit Java verwandt ist und im WWW die wichtigste Programmiertechnik im Browser ist,

Verfolgen wir unser Gedankenmodell mit einem ausgewählten realen Gegenstand weiter. Wenn Sie einen (realen) Kugelschreiber in die Hand nehmen, können Sie die **Eigenschaften** dieses (realen) Objekts beschreiben, seine Form, die Größe, die Farbe. Sie können alle Eigenschaften (Attribute) des Objekts aufführen, soweit es notwendig beziehungsweise sinnvoll ist. Und Sie können die aktiven Fähigkeiten des Stifts beschreiben, soweit dieses notwendig beziehungsweise sinnvoll ist (**Methoden**), etwa, dass Sie mit dem Stift schreiben können.

Dabei ist es offensichtlich, dass Sie mit dem Kugelschreiber nur dann schreiben können (eine aktive Fähigkeit), wenn die Mine vorher ausgefahren wurde. Und ebenso klar ist, dass eine Mine nur dann herausgedrückt werden kann, wenn sie zu diesem Zeitpunkt eingefahren ist und umgekehrt. Offensichtlich kann sich ein reales Objekt in einem spezifischen **Zustand** befinden und bestimmte Fähigkeiten in Abhängigkeit davon bereitstellen oder deaktivieren.

Ebenso ist ein Objekt in unserer Wahrnehmung immer nur die **Abstraktion** eines realen Gegenstands. Auch wenn das erst einmal sehr theoretisch klingt, bedeutet das nur, dass man immer selektiv die wirklich vorhandenen Eigenschaften und Methoden eines Objekts wahrnimmt. Dabei spricht man von Abstraktion, weil zur Lösung eines Problems normalerweise sämtliche Aspekte eines realen Elements weder benötigt werden noch überhaupt darstellbar oder wahrnehmbar sind. Irgendwo muss immer abstrahiert werden.

Nehmen wir uns wieder unser reales Referenzobjekt. Jeder Kugelschreiber hat ein spezifisches Gewicht, einen bestimmten inneren Aufbau aus Mine und Feder bis hin zu einer im Prinzip angebbaren Anzahl an Atomen, aus dem er besteht. Ebenso können Sie mit dem Kugelschreiber einen Nagel in die Wand schlagen oder darauf Musik machen. Aber weder interessieren Sie in der Regel diese gerade genannten Eigenschaften noch nutzen Sie diese Methoden, die über ein »natürliches« Verständnis des Objekts hinausgehen – wir abstrahieren den realen Gegenstand, indem wir ihn in der Wahrnehmung auf nützliche Dinge reduzieren.

Info

Die objektorientierte Denkweise im Computerumfeld entspricht vollkommen dem Abbild der realen Natur und den Objekten, die wir dort wahrnehmen. Die Objektorientierung versucht diese aus der realen Welt so natürlichen Vorstellungen in die EDV zu übertragen.

3.2.2 Objektorientierte Denkweise in der Benutzerführung

Wenn Sie sich den Desktop einer grafischen Betriebssystemoberfläche ansehen, repräsentieren alle dort zu findenden Symbole Objekte, die auf nützliche Dinge abstrahiert wurden. Ein Objekt wird also über ein Symbol (ein Icon) dargestellt und dem Anwender wird ein Menü (zum Beispiel ein Kontextmenü) bereitgestellt, das die Möglichkeiten des Objekts und seine Eigen-

schaften – möglicherweise abhängig von einem Zustand – zugänglich macht. Bei einer grafischen Benutzeroberfläche setzt man die Darstellung für so eine zustandsabhängige Verhaltensweise dadurch um, dass man zu einem bestimmten Zeitpunkt nicht verwendbare Fähigkeiten in einem Menü deaktiviert und grau darstellt. Die aktivierbaren Fähigkeiten und der Zugang zu den Eigenschaften werden normal dargestellt. Betrachten Sie nur einmal das Symbol des Papierkorbs auf Ihrem Desktop.

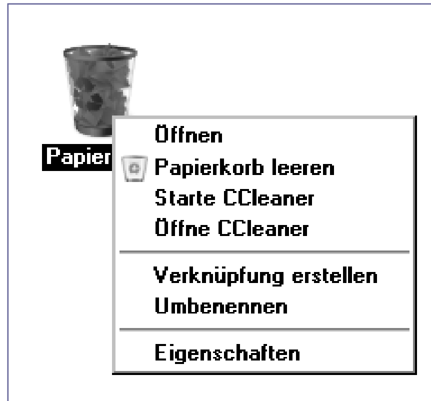


Abbildung 3.1
Objektsymbole in einer GUI samt Kontextmenü zum Bereitstellen der Fähigkeiten und Eigenschaften eines Objekts – der Papierkorb kann geleert werden.

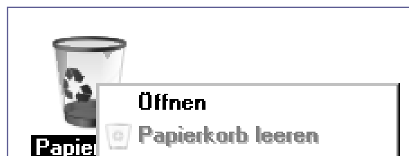


Abbildung 3.2
Der Zustand des Objekts sorgt dafür, dass die Methode zum Leeren nicht verfügbar ist.

3.2.3 Objektorientierte Denkweise in der Programmierung

Objektorientierung kann nicht nur auf Anwendungsebene in einer GUI umgesetzt werden, sondern ebenso in der Programmierung. Hier im Buch soll es ja explizit darum gehen. Unter einem Objekt stellt man sich in der EDV allgemein ein Softwaremodell vor, das ein Ding aus der realen Welt mit all seinen ***relevanten*** Eigenschaften und Verhaltensweisen beschreiben soll und das einem Anwender über ein Symbol oder einen Bezeichner zugänglich ist, zum Beispiel das Objekt Drucker, Bildschirm oder Tastatur oder ein Objekt aus der realen Welt, das EDV-technisch abgebildet werden soll. Beispiel wären ein Stift, ein Bankkonto, ein Bauernhof, ein Geldinstitut, ein Mensch. Auch Teile der Software selbst können ein Objekt sein, ein Browser beispielsweise oder ein Teil davon – zum Beispiel ein Frame. Oder Teile der Verzeichnisstruktur eines Rechners, zum Beispiel ein Ordner. Im Grunde ist in dem objektorientierten Denkansatz alles als Objekt zu verstehen, was sich eigenständig erfassen und ansprechen lässt.

Objektorientiert versus prozedural

In der althergebrachten Programmierung (ab der dritten Generation) programmiert man prozedural. Dies bedeutet die Umsetzung eines Problems in ein Programm durch eine Folge von Anweisungen, die in einer festgelegten Reihenfolge auszuführen sind. Einzelne zusammengehörende Anweisungen werden dabei maximal in kleinere Einheiten von Befehlsschritten zusammengefasst (sogenannte Unterprogramme oder Module). Ganz wichtig – die Datenebene und die Anweisungsebene können aufgetrennt werden. Das Problem prozeduraler Programmierung ist, dass Änderungen in der Datenebene Auswirkungen auf die unterschiedlichsten Programmsegmente haben können und die Wiederverwendbarkeit von Unterprogrammen sehr eingeschränkt ist, da oft Abhängigkeiten zu anderen Bestandteilen eines Programms existieren (beispielsweise zu globalen Variablen).

Objektorientierte Programmierung lässt sich im Vergleich zur prozeduralen Programmierung darüber abgrenzen beziehungsweise definieren, dass zusammengehörende Anweisungen und Daten eine **zusammengehörende, abgeschlossene und eigenständige Einheit** bilden – eben Objekte! OOP hebt die Trennung von Daten- und Anweisungsebene auf! Das kann als Kernsatz der OOP verstanden werden.

Java setzt den objektorientierten Ansatz konsequenter um als viele vergleichbare Sprachen. Das hat massive Konsequenzen. Es gibt in Java beispielsweise keine globalen Variablen. Diese würden außerhalb von Objekten existieren und das ist explizit unmöglich. Auch Strings und Arrays sind in Java Objekte. Ebenso werden Ereignisse, Ausnahmen oder Fehler durch spezifische Objekte repräsentiert.

Sogar ein Java-Programm selbst ist als Objekt zu verstehen. Es existiert im Hauptspeicher des Computers, solange das Programm läuft. In Java muss im Grunde nur eine Ausnahmesituation für die Aussage »Alles ist ein Objekt« beschrieben werden. Das sind sogenannte primitive Datentypen. Auf die hat Sun bei der Konzeption von Java nicht verzichtet, um prozedurale Programmierer nicht vollkommen vor den Kopf zu stoßen. Aber auch primitive Datentypen sind mittels sogenannter Wrapper-Klassen in das Konzept eingebunden. Damit werden für jeden primitiven Datentyp zugehörige Objekte bereitgestellt. Diese Details sollen allerdings zurückgestellt werden, denn primitive Datentypen zählen zu den syntaktischen Fragen, die später beantwortet werden.

Info

EDV-technisch sind Objekte bestimmte Bereiche im Hauptspeicher des Rechners, in denen zusammengehörige Informationen und Funktionalitäten gespeichert oder zugänglich gemacht werden.

Identifizieren Sie sich – Botschaften

Die Verwendung von Objekten in einer GUI erfolgt – wie wir besprochen haben – in der Regel über Kontextmenüs, indem ein Anwender zum Beispiel mit der rechten Maustaste auf ein Objekt klickt und im Kontextmenü

alle erlaubten Methoden und Eigenschaften angezeigt werden. Im Rahmen eines Quelltextes muss ein Zugriff auf Objekte natürlich anders erfolgen. Man braucht einen Namen für das Objekt oder zumindest einen Stellvertreterbegriff, der ein Objekt repräsentiert (der **Identifikator**). Egal, ob die Identität eines Objekts durch einen grafischen oder einen textnotierten Identifikator dargestellt wird – in der Regel ist der Identifikator eine Referenz (ein Zeiger beziehungsweise Pointer) auf das tatsächliche Objekt. Meist handelt es sich bei einer solchen Referenz technisch um eine Hauptspeicheradresse, aber es kann auch anderes umgesetzt werden. Das interessiert uns aus Sicht von Java aber grundsätzlich nicht. Wir müssen in Java nicht so tief in die Systemebene hinunter. Das ist ein zentraler Aspekt von Java und dessen virtueller Maschine.

Besonders wichtig ist im Zusammenhang mit der Verwendung von Objekten der Begriff der **Botschaften**. Damit man Objekte im Rahmen eines Quelltextes verwenden kann, tauschen sie sogenannte Botschaften (oder Nachrichten beziehungsweise Messages) aus. In der strengen OOP werden ausschließlich Botschaften für die Kommunikation zwischen Objekten verwendet. Andere Kommunikationswege gibt es nicht. Das Objekt, von dem man etwas will, erhält eine Aufforderung, eine bestimmte Methode auszuführen oder den Wert einer Eigenschaft zurückzugeben oder zu setzen. Das Zielobjekt versteht (hoffentlich) diese Aufforderung und reagiert entsprechend. Die genaue formale Schreibweise solcher Botschaften ist in den meisten OO-Programmiersprachen nach dem folgenden Schema aufgebaut:

```
Empfängerobjekt [Methode oder Eigenschaft]
```

Listing 3.1
Formale Verwendung eines Objekts über eine Botschaft

In den meisten OO-Sprachen trennt dabei ein Punkt die Bestandteile der Botschaft. Deshalb wird von der **Punktnotation** beziehungsweise **DOT-Notation** gesprochen. Eine Botschaft, dass ein Objekt eine bestimmte Methode bereitstellen soll, sieht also meist so aus:

```
Empfängerobjekt.Methodenname(Argument)
```

Listing 3.2
Eine Methode mit der Punktnotation verwenden

Das Argument stellt in dem Botschaftsausdruck einen Übergabeparameter für die Methode dar. Analoges gilt für die Verwendung von Objektattributen. In der Regel sieht das dann so aus:

```
Empfängerobjekt.Attributname
```

Listing 3.3
Eine Eigenschaft mit der Punktnotation verwenden

Ich denke, also bin ich

Aus programmiertechnischer Sicht ist es so, dass nicht nur der Programmierer weiß, was ein Objekt leistet. Auch das Objekt weiß es selbst und kann es nach außen dokumentieren. Es ist sich quasi seiner Existenz und seiner Fähigkeiten und Eigenschaften bewusst[3]. Das wird in Java-Tools beispielsweise sehr umfangreich eingesetzt und äußert sich darin, dass der Compiler beim Übersetzen bereits vieles von dem abfangen kann, was bei der Verwendung eines Objekts nicht erlaubt ist. Aber sogar schon vor der Kompilierung kann man diese Information nutzen. In geeigneten Entwicklungsumgebun-

3 Erinnern Sie sich an die Anekdote am Beginn des Buchs, wie ein Java-Programmierer einen Löwen fängt? Jeder Java-Löwe bringt seinen eigenen Käfig mit.

gen kann Ihnen ein Editor bereits Hilfe anbieten, indem er Ihnen bei einem Objekt alles anzeigt, was das Objekt leisten kann. Das haben Sie in den Beispielen mit Eclipse und NetBeans gesehen.

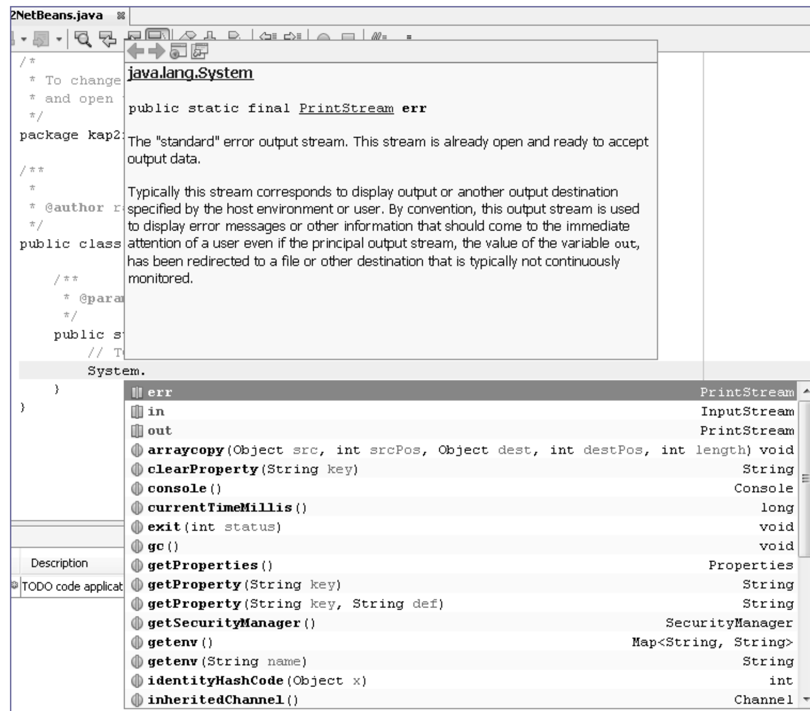


Abbildung 3.3 Ein Objekt gibt einer IDE gegenüber Auskunft, was es an Eigenschaften und Methoden bereitstellt.

Und umgekehrt gilt, dass man nur das von einem Objekt fordern kann, was da auftaucht, wobei auch IDEs gelegentlich fehlerhaft sind! Das ist dann aber kein Widerspruch und nicht auf die allgemeine objektorientierte Philosophie übertragbar.

In der OOP gibt es also eine Hilfe bzw. Kommunikation in zwei Richtungen. Der Programmierer sieht in einer geeigneten IDE sämtliche Eigenschaften und verfügbaren Methoden eines Objekts (samt dessen Zustand) und wird auf der anderen Seite sofort darauf hingewiesen, wenn er etwas vom Objekt anfordert, was dieses nicht bietet (falls zum Beispiel ein Schreibfehler bei einer Methode oder Eigenschaft gemacht wurde).

3.2.4 Klassen

Wenn nun Objekte die Basis der OOP sind und man in streng objektorientierten Sprachen wie Java ohne Objekte nichts machen kann – wie entstehen Objekte und was muss man als Programmierer konkret tun? Die Lösung lautet wie schon erwähnt **Klassen** (auch **Objekttyp** oder abstrakter Datentyp genannt) und darin enthaltene Konstruktoren. Diese müssen Sie schreiben und verwenden. Wenn man es genau betrachtet, besteht die gesamte

Java-Programmierung daraus, Klassen zu schreiben und darin Methoden und Eigenschaften zu definieren.

Info

Das Schreiben einer Klasse, aber auch die Notation der darin enthaltenen Eigenschaften/Attribute und Methoden nennt man die Deklaration.

Eine Klasse kann man sich wie angedeutet einmal als eine Gruppierung von ähnlichen Objekten vorstellen, die deren Klassifizierung ermöglicht. Eine Klasse ist also die Definition der gemeinsamen Attribute und Methoden sowie der Semantik für eine Menge von gleichartigen Objekten. Alle erzeugten Objekte einer Klasse werden dieser Definition entsprechen.

Die Eigenschaften und die Funktionalität der Objekte werden also in der Gruppierung gesammelt und für eine spätere Erzeugung von realen Objekten verwendet. Mit anderen Worten – Klassen sind so etwas wie Baupläne oder Rezepte, um mit deren Anleitung ein konkretes Objekt zu erzeugen. Ein aus einer bestimmten Klasse erzeugtes Objekt nennt man deren **Instanz**. Beachten Sie, dass der Zustand eines spezifischen Objekts natürlich nicht in einer Klassifizierung auftaucht. Allerdings kann man in einer Klasse einen **Anfangszustand** für jedes daraus erzeugte Objekt beschreiben.

UML und Diagrammdarstellungen in der OOP

Kommen wir zu einem kleinen Exkurs, der spätere Diagramme im Buch vorbereiten soll. Am Erstellen einer größeren Applikation sind nicht nur Programmierer beteiligt. Es gibt Leute, die Fachvorgaben machen, andere analysieren ein Problem (im Fall der Objektorientierung spricht man von der objektorientierten Analyse – **OOA**). Und dann gibt es Fachleute, die ein objektorientiertes Modell schaffen, das später konkret programmiert werden soll. Das nennt man das objektorientierte Design (**OOD**). Das Problem ist, dass diese verschiedenen Gruppen, die allesamt wichtig für ein Programmierprojekt sind, meist unterschiedliche Kenntnisse haben und oft über keine gemeinsame (Fach-)Sprache verfügen. Um eine gemeinsame Kommunikation zu gestatten, greift man deshalb in vielen professionellen Projekten auf eine Metasprache mit grafischen Modellen zurück, die die Strukturen und Abläufe in einer Applikation verdeutlichen soll.

In den meisten Modellen der OOP wird man zur grafischen Darstellung von Klassen mit ihrem Aufbau und ihren Beziehungen Diagramme verwenden, aber auch für die Darstellung von Objekten und ihren Beziehungen, für den Ablauf von Programmen und einiges mehr. Solche Hilfsmittel werden vor allem in Analyse- und Designphasen der Software-Entwicklung als flankierende oder gar unabdingbare Maßnahmen Verwendung finden. Aber diese grafischen Möglichkeiten können wir uns ebenso im Buch zunutze machen, um insbesondere Klassenstrukturen und Klassenabhängigkeiten zu visualisieren.

In der Praxis kommt in vielen Fällen eine neutrale Sprache mit Namen **Unified Modeling Language** (UML) zum Einsatz. UML bezeichnet eine vereinheitlichte Modellierungssprache, mit der allgemein Strukturen und Abläufe

in objektorientierten Softwaresystemen darzustellen sind. Die Sprache definiert eigene Bezeichner und Symbole für die meisten Begriffe in der OO sowie deren mögliche Beziehungen und Abläufe. In geeigneten Tools kann man mit UML OO-Modelle »zeichnen«, die sogar per Mausklick direkt in Grundgerüste (Skeletons oder Stubs) einer konkreten OO-Sprache wie Java überführt werden können.

Tipp

Es gibt eine ganze Anzahl an – oft freien bzw. kostenlosen – UML-Tools, auch als Plug-ins für Eclipse. Als Beispiele seien UML2 Tools, das Eclipse UML-Tool von Omondo, UModel, ArgoUML oder die Community Edition von Visual Paradigm for UML, die im Buch zusammen mit dem eigenständigen ArgoUML als Eclipse-Plug-in Verwendung findet, genannt. Allerdings ist UML für unsere Belange wirklich nur ein winziges Randthema und sowohl die zahlreichen Details zu UML als auch der Umgang mit den verschiedenen Tools werden hier nicht tiefer behandelt. Für Sie soll ein Diagramm nur gegebenenfalls die Struktur einer Klasse grafisch verdeutlichen.

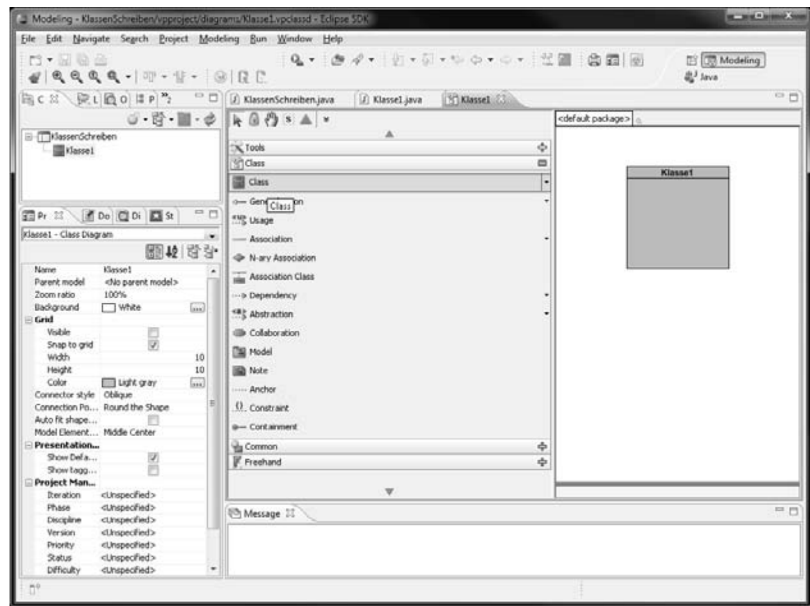


Abbildung 3.4
Ein UML-Plug-in in Eclipse

Info

UML 2.0 unterstützt eine ganze Reihe von Diagrammtypen. Wir werden im Buch nur gelegentlich Klassendiagramme zur grafischen Sicht auf ausgewählte, statische Aspekte des modellierten Software-Systems verwenden. Als Überschrift einer Klassendarstellung werden deren Name und darunter die Eigenschaften und darunter die Methoden der Klasse samt Sichtbarkeit und Typ angegeben. Statische Elemente werden in einem Klassendiagramm unterstrichen dargestellt und bei Attributen kann über ein Gleichheitszeichen ein zugewiesener Startwert angezeigt werden. Beachten Sie, dass kein Wert auf die exakte UML-Notation in den Diagrammen gelegt und davon abgewichen wird, wenn ein Sachverhalt durch eine mehr direkt an Java orientierte oder reduzierte Darstellung besser deutlich wird.

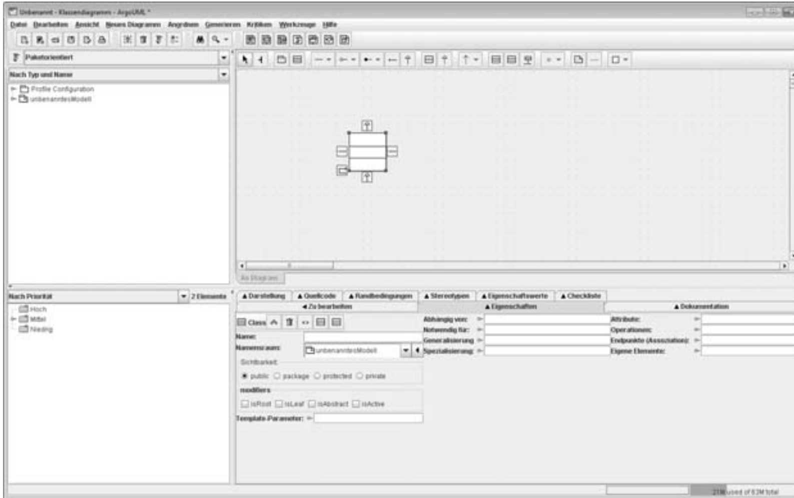


Abbildung 3.5
Das in Java geschriebene eigenständige ArgoUML

3.2.5 Konkret in Java Klassen schreiben

Sämtlicher Java-Code wird aus Klassen erzeugt. In Java kann man das deutlich erkennen. Jede Klasse im Java-Code beginnt mit dem Schlüsselwort `class`, dem höchstens Modifizierer vorangestellt werden können[4] und dem dann der Bezeichner der Klasse folgt.

Namensregeln

Wenn Sie Klassen und Objekte konstruieren oder auf fremde Klassen und Objekte zugreifen wollen, ist der Schlüssel für den Zugriff (meist) der Klassenname beziehungsweise der Bezeichner eines Objekts. Ebenso verwenden Sie Bezeichner für Variablen und Methoden. Die Namen sind in Java relativ frei zu wählen. Es gibt in Java einige wenige Regeln, die die Vergabe von Bezeichnern beschränken und sowohl für Klassen, aber auch die anderen Strukturen gelten:

- Bezeichner dürfen im Prinzip eine unbeschränkte Länge haben (bis auf technische Einschränkungen durch das Computersystem).
- Bezeichner dürfen nicht getrennt werden.
- Bezeichner in Java müssen mit einem Buchstaben oder einem der beiden Zeichen `_` oder `$` beginnen. Es dürfen weiter nur Unicode-Zeichen oberhalb des Hexadezimalwerts `00C0` (Grundbuchstaben und Zahlen sowie einige andere Sonderzeichen) verwendet werden. Das bedeutet mit anderen Worten, jeder (!) Buchstabe eines beliebigen (!) im Unicode abgebildeten Alphabets kann verwendet werden.

4 »Sonderklassen« wie Schnittstellen oder Enums sollen erst einmal außer Acht gelassen werden.

- Zwei Bezeichner sind nur dann identisch, wenn sie dieselbe Länge haben und jedes Zeichen bezüglich des Unicode-Werts identisch ist. Java unterscheidet deshalb Groß- und Kleinschreibung.
- Selbst definierte Bezeichner dürfen keinem Java-Schlüsselwort gleichen und sie sollten nicht mit Namen von Java-Paketen identisch sein.

Namenskonventionen

Es gibt nun in Java neben den zwingenden Regeln für Bezeichner ein paar unverbindliche, aber gängige Konventionen für Klassennamen, Objektnamen und andere Elemente. Man sollte in der Praxis möglichst sprechende Namen verwenden. Wenn Sie eine Klasse erstellen, die ein Fenster generiert, wäre `MeinFenster` ein sprechender Name für die Klasse und `meinFenster1`, `meinFenster2` und `meinFenster3` wären sprechende Namen für daraus erzeugte Objekte. Oder Sie verwenden eine englische Notation. Sehr oft werden Bezeichner gewählt, die eine Funktion genauer beschreiben. Wenn etwa eine Klasse `Button` zur Erzeugung einer OK-Schaltfläche verwendet wird, wäre `OKbtn` ein sinnvoller Name für die referenzierende Variable. Aber über diese recht schwammigen Regeln hinaus gibt es Konventionen, die ziemlich genau formuliert sind.

Da gibt es einmal Regeln für die **Groß- und Kleinschreibung**. Die Bezeichner von Klassen (und Schnittstellen oder Enums, was besondere Formen von Klassen sind) sollten immer mit einem Großbuchstaben beginnen und anschließend kleingeschrieben werden. Konstanten werden vollständig groß geschrieben. Die Bezeichner von Objekten, Variablen oder Methoden sollten immer mit einem Kleinbuchstaben beginnen und anschließend weiter kleingeschrieben werden. Wenn sich ein Bezeichner aus mehreren Wörtern zusammensetzt, dürfen diese nicht getrennt werden. Die jeweiligen Anfangsbuchstaben der zusammengesetzten Begriffe werden jedoch innerhalb des Gesamtbezeichners großgeschrieben (Camelnotation).

Darüber hinaus gibt es weitere Konventionen für verschiedene Syntaxstrukturen, die als so wichtig und selbstverständlich angesehen werden, dass IDEs wie Eclipse oder NetBeans bei vielen automatischen Vorgängen Ihnen gar keine Möglichkeiten bieten, von den Konventionen abzuweichen. Das werden wir sehen. In vielen professionellen Projekten werden zusätzlich zu den zwingenden Regeln und den offiziellen Konventionen ergänzende, projektverbindliche Regeln für Bezeichner aufgestellt.

Aber wie gesagt – Konventionen sind keine zwingenden Regeln. Sie müssen sich nicht an diese Konventionen halten. Sie sollten es aber dringend! Nur so kann jeder, der Ihren Quellcode liest, bereits über die Bezeichner Syntaxstrukturen eindeutig identifizieren. Außerdem passt dann die Logik Ihrer Bezeichner zu der überall sonst (in dem Java-API, aber auch anderen Projekten) verwendeten Logik. Wenn Sie diese Konventionen einhalten, ist unmittelbar klar, ob es sich bei den – vollkommen aus dem Syntaxzusammenhang gerissenen – Bezeichnern um eine Klasse oder ein Objekt oder auch

ein anderes Strukturelement handelt. Und Sie bekommen Unterstützung von Tools nur entsprechend der Konventionen.

Info

Um die Bedeutung von Konventionen und Regeln beim Kodieren in professionellen Projekten zu dramatisieren – wenn ich Projektleiter wäre und ein Programmierer würde die Konventionen nicht einhalten, wäre das für mich Sabotage und der Programmierer würde aus dem Projekt fliegen, auch wenn der Code funktional vollkommen in Ordnung wäre. Die Wartbarkeit und Verständlichkeit von Quellcode sind in einem Projekt auf längere Sicht mindestens genauso wichtig wie die Funktionalität.

Ein Beispiel zum Erstellen einer eigenen Klasse

Verdeutlichen wir uns das Erstellen einer eigenen Klasse mit einem konkreten Java-Beispiel. Dazu sollten Sie wieder in einem Verwaltungsverzeichnis Ihrer Projekte ein Projektverzeichnis anlegen, in meinem Fall *Kap3/KlassenSchreiben*.

Legen Sie dort eine Java-Datei mit dem Namen *Klasse1.java* an. Dazu können Sie zum Beispiel in Eclipse den im letzten Kapitel beschriebenen Weg verwenden.

Info

Machen wir uns noch einmal Gedanken um den Namen der Java-Datei. Vielleicht haben Sie schon einmal etwas von einem Zusammenhang zwischen dem Namen der *.java*-Datei und dem darin notierten Klassenbezeichner gehört. Immerhin haben wir die beiden Bezeichner bisher immer gleich gewählt. Und das war nicht zufällig.

Wenn eine Klasse als öffentlich deklariert wird (mit dem vor dem Schlüsselwort `class` platzierten Schlüsselwort `public`[5]), **muss** die *.java*-Datei einen Namensstamm haben, der identisch mit dem Bezeichner der Klasse ist. Andernfalls ist der Name frei. Das ist auch notwendig, denn Sie können in einer *.java*-Datei eine **beliebige Anzahl** an Klassendeklarationen notieren. Und dann kann nur eine davon einen Namen der *.java*-Datei erzwingen. das bedeutet umgekehrt, dass in einer *.java*-Datei nur eine der darin deklarierten Klassen öffentlich sein darf. Der Compiler erzeugt bei der Übersetzung für jede Klassendefinition eine eigene Bytecode-Datei mit dem Namen des Klassenbezeichners!

Sie sollten allerdings dringend von solchen Möglichkeiten Abstand nehmen und zwei Regeln einhalten:

1. Jede Klasse wird in einer eigenen Datei deklariert.
2. Der Name der deklarierten Klasse legt den Namen der *.java*-Datei fest (gleichgültig, ob öffentlich oder nicht).

Dies ist für die Wartbarkeit und Übersichtlichkeit immens von Vorteil.

5 Dazu kommen wir noch genauer.

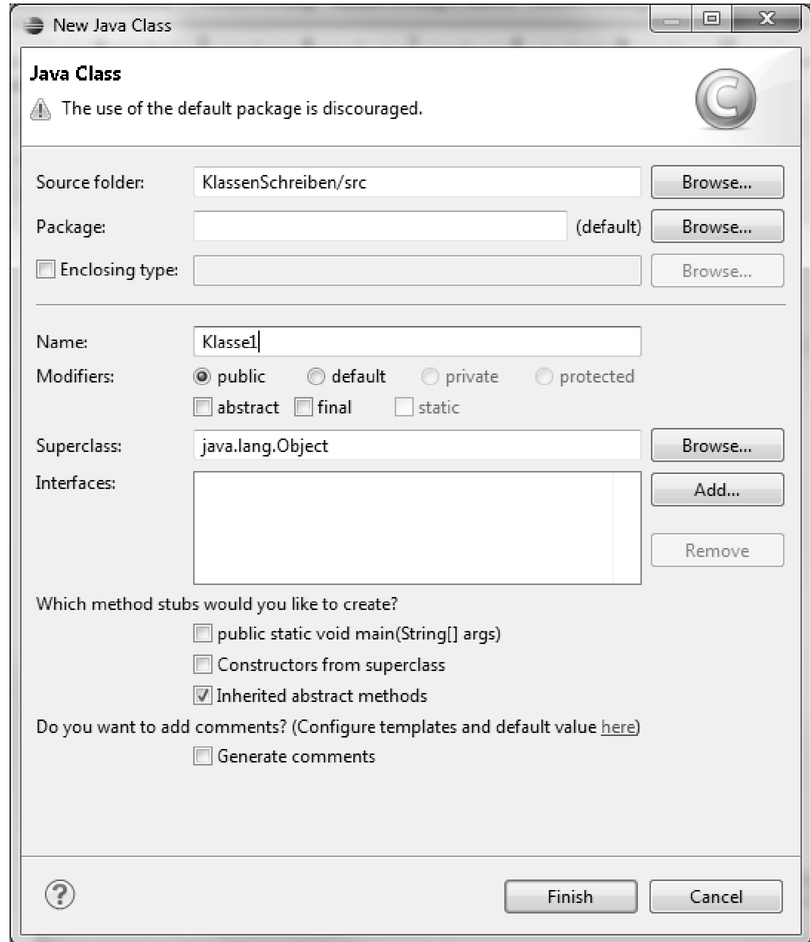


Abbildung 3.6 In Eclipse in einem Projekt eine neue Klasse erstellen

Zu Beginn werden Sie – wenn nicht irgendwelche Codestrukturen generiert wurden – eine leere Klassenschablone sehen:

Listing 3.4 Eine leere Klassenschablone

```
public class Klasse1 {
}
```

Dennoch besitzt diese Klasse bereits alle Kennzeichen einer Bauvorschrift für Objekte. Das Beispiel zeigt insbesondere die grundsätzliche Vorgehensweise, wie Sie in Java eine Klasse erstellen. Zu Beginn steht nach einem optionalen Modifizierer für die Zugänglichkeit der Klasse das Schlüsselwort `class`, gefolgt von einem Bezeichner für die Klasse. In den geschweiften Klammern (dem Block) wird der eigentliche Code der Klasse notiert.

Die Inhalte von Klassen

Bei der Spezifikation von Klassen können **Instanzelemente** und **Klassenelemente** deklariert werden. Das sind in beiden Fällen die Attribute und die Methoden, die ein Objekt besitzen soll und die relevanten spezifischen

Eigenschaften und Funktionalitäten charakterisieren. Aber die beiden Typen unterscheiden sich im Zeitpunkt ihrer Existenz und im möglichen Zugang.

- **Instanzattribute** und **Instanzmethoden** sind die Elemente, die erst in einer konkret erzeugten Instanz existieren und erst dort verwendet werden können. Sie gehören damit explizit zu einer konkreten Instanz, zu einem einzelnen Objekt. Andere Instanzen bekommen nicht mit, wenn und wie eine Instanz auf ihrem Instanzelement operiert. Ohne besondere Kennzeichnung sind die Attribute (Eigenschaften) und Methoden in Java immer Instanzelemente.
- **Klassenattribute** (auch **statische Attribute** genannt) sind Attribute, die zu einer Klasse selbst und nicht zu einem spezifischen Objekt gehören. Alle Instanzen einer Klasse sowie die Klasse selbst haben gemeinsam Zugriff auf statische Attribute, die zudem unabhängig davon existieren, ob es von der Klasse kein, ein oder mehrere Objekte gibt. Vollkommen analog sind **Klassenmethoden** (auch **statische Methoden** genannt) solche Methoden, die unabhängig von einem Objekt der Klasse ausgeführt werden können. Statische Attribute und Methoden ähneln in gewisser Weise globalen Variablen und globalen Funktionen in prozeduralen Programmiersprachen. Statische Klassenelemente sind jedoch im Unterschied zu diesen dem Namensraum der Klasse zugeordnet. Sie sind also auf die Klasse und ihre Instanzen beschränkt und existieren nicht in anderen Klassen und daraus erzeugten Objekten. Dafür existieren Klassenelemente grundsätzlich bereits bei Programmstart, wenn eine spezifische Klasse verwendet wird. Das gilt auch dann, wenn noch kein Objekt konstruiert wurde. In Java werden Klassenelemente mit dem vorangestellten Modifizierer `static` gekennzeichnet, was man sich ob der alternativen Bezeichnung als ***statische*** Attribute bzw. Methoden sicher gut merken kann.

Info

Grundsätzlich sollte die Verwendung von Klassenelementen mit Vorsicht geschehen, denn in der Regel ist es nicht sinnvoll, wenn spätere Änderungen klassenglobal erfolgen und sowohl die Klasse als auch alle Instanzen von einer Änderung betroffen sind. Klassenelemente werden wie erwähnt statische Elemente genannt und in Java entsprechend mit `static` gekennzeichnet. Diese Betrachtung zeigt, was sich technisch dahinter verbirgt. Während eine Instanzvariable in der Instanz aus der Klasse als **Kopie** der Klassendeklaration erzeugt wird und in der Instanz diese Kopie vollkommen ohne Auswirkungen auf die Klasse oder andere Instanzen der Klasse verändert werden kann, wird bei einer Klassenvariablen nur eine Referenz (ein Zeiger) auf den Speicherbereich in der Klasse in die Instanz kopiert. Der eigentliche Speicherbereich mit dem Wert bleibt fest (statisch). Dementsprechend wird sich jede Änderung in der Klasse in allen Instanzen auswirken.

Variablen

Im Zusammenhang mit Attributen/Eigenschaften muss man den Begriff der **Variablen** ansprechen. Auf Variablen unter Java gehen wir noch im Detail ein. Hier soll aber eine erste Erklärung folgen, soweit sie an der Stelle notwendig ist.

Eine Variable ist erst einmal ein benannter Speicherplatz im Hauptspeicher des Computers, der frei mit (passenden) Werten gefüllt werden kann. Diese Werte können über den Bezeichner wieder ausgelesen werden. Dazu hat eine Variable einen spezifischen **Typ** oder **Datentyp**, der bestimmt, welche Art von Wert darin gespeichert werden kann. Und eine Variable wird innerhalb einer Klasse deklariert.

Variablen versus Felder, Eigenschaften und Attribute

Doch wie hängen Variablen und die Begriffe Felder, Eigenschaften, Attribute zusammen? Aus Sicht der OOP ist eine Variable, die über das Objekt bzw. die Klasse nach außen gegeben und damit ***sichtbar*** ist, eine Eigenschaft bzw. ein Attribut eines Objekts. Attribut und Eigenschaft kann man im Sinn der objektorientierten Philosophie also synonym verwenden, sofern man nicht einen der beiden Bezeichner noch mit zusätzlichen Kriterien versieht, was etwa von Microsoft im .NET-Umfeld in Hinsicht auf die Verwendung des Begriffs »Eigenschaft« gemacht wird. Ist die Variable von außen nicht zugänglich, ist es keine Eigenschaft bzw. kein Attribut, sondern eben nur eine interne Variable, die für irgendwelche Algorithmen in der Klasse verwendet werden kann. Die sogenannte Sichtbarkeit bzw. Zugänglichkeit legt also in Java bzw. in der reinen Objektorientierung fest, ob eine Variable als Eigenschaft/Attribut zu verstehen ist oder nicht.

Info

Ob die Variable mit `static` gekennzeichnet ist oder nicht, spielt bei dieser Einordnung keine Rolle.

Oft redet man neutral von sogenannten **Feldern**. Damit ist man sozusagen auf der sicheren Seite, wenn man nicht bewusst die Eigenschaften/Attribute und Variablen trennen will. Ein Feld kann von außen sichtbar sein oder aber auch nicht.

Achtung

Beachten Sie, dass wie gesagt in manchen OO-Konzepten die Bezeichnung Eigenschaften speziell und enger als die Bezeichnung Attribut interpretiert wird und man damit gewisse syntaktische Besonderheiten verbindet. Allgemein versteht man in diesen Konzepten Eigenschaften oft so, dass diese nur mithilfe bestimmter Aktionen gelesen und geschrieben werden können. Eigenschaften erlauben über diese indirekten Zugriffe eine größere Kontrolle über die Werte der Attribute eines Objekts. Insbesondere lassen sich solche Eigenschaften gut über visuelle Tools darstellen und verändern, was in einigen IDEs ausgenutzt wird. Sollten Sie damit in Berührung kommen, müssen Sie gegebenenfalls aufpassen. Wir werden im Buch allerdings entsprechend der reinen OOP-Theorie Attribute einfach als die zugänglichen Eigenschaften eines Objekts verstehen.

Deklaration von Variablen und Eigenschaften

Betrachten wir noch einmal Variablen und Eigenschaften bzw. Attribute aus einer anderen Sicht. Ein Attribut, dessen Wert ein Objekt von einem anderen unterscheidet, bezeichnet aus Sicht des Quelltextes eine benannte Eigenschaft. Ein Attribut ist ein Datenelement, das in jedem Objekt einer Klasse

enthalten ist und in jedem dieser Objekte durch einen individuellen Wert repräsentiert ist. Im Quelltext wird jedes Attribut über einen Datentyp und einen Bezeichner repräsentiert. Die allgemeine Schreibweise sieht so aus:

```
attributName : datenTyp
```

Listing 3.5
Formale Beschreibung (UML) eines Attributs

Diese Form der Darstellung wird in der Regel im Rahmen eines UML-Klassendiagramms verwendet, während Java eine spezielle syntaktische Darstellung verwendet, die aber von der Aussage her identisch ist. Im Grunde ist diese Deklaration aber unabhängig davon, welche Sichtbarkeit bzw. Zugänglichkeit hier verwendet wird. Daher handelt es sich eigentlich um eine allgemeine Variablendeklaration.

In Java wird eine Variable immer so deklariert, dass zuerst der Datentyp und dann der Bezeichner für die Variable notiert wird. Allgemein wird in Java eine Eigenschaft einfach darüber definiert, dass in einer Klasse ein Typ der Eigenschaft festgelegt und dann ein Bezeichner vergeben wird und zusätzlich sichergestellt wird, dass die Zugänglichkeit von außen groß genug ist. Dazu regelt man die Sichtbarkeit mit vorangestelltem Modifizierer. Etwa so:

```
private int a;
Integer b;
public String meinText;
protected Tier obj;
java.awt.Frame fenster;
```

In der Regel weist man der Eigenschaft dann noch einen Wert zu, entweder direkt bei der Deklaration oder in einem späteren Schritt.

Info

Instanz- und Klasseneigenschaften haben in Java immer einen Vorgabewert. Er entspricht dem numerischen Nullwert, dem Leerstring oder im booleschen Fall dem Wert `false`. Beachten Sie, dass lokale Variablen dagegen keinen Vorgabewert besitzen.

Weitere Modifizierer wie `static` regeln zusätzliche Details.

Achtung

Die Definition einer Eigenschaft bzw. Variablen kann **an einer beliebigen Stelle** im Namensraum einer Klasse, aber auch in einer Methode (als **lokale Variable**) oder auch in einer Struktur wie einer Schleife (als **schleifenlokale Variable**) erfolgen. Für die bessere Lesbarkeit sollten Sie jedoch die Deklarationen in Ihren Klassen einheitlich und strukturiert vornehmen.

Lokale Variablen versus nichtlokale Variablen und Eigenschaften

Eben wurden bereits **lokale Variablen** angesprochen. Das sind Variablen, die nur in einem geschlossenen Bereich unterhalb des Gültigkeitsbereichs der Klasse gültig sind (meist innerhalb einer Methode, einer Schleife oder eines Blocks). Lokale Variablen verwenden keine Zugriffsmodifizierer (das nennt man auch Sichtbarkeitsmodifizierer) und können nicht als `static` definiert werden. Sie sind ja explizit nicht als Eigenschaften vorgesehen. Ebenso gilt es bei lokalen Variablen zu beachten, dass diese in Java **keinen** Defaultwert haben, während Klassen- und Instanzeigenschaften – wie gerade erwähnt – immer einen **Vorgabewert** haben.

Achtung

Vor der Verwendung einer lokalen Variablen muss deshalb zwingend ein Wert zugewiesen werden. Am besten machen Sie das direkt bei der Deklaration, aber Sie können dies auch später in einer zweiten Anweisung durchführen. Nur muss die Zuweisung eben vor dem ersten lesenden Zugriff auf die Variable erfolgen.

Als Modifizierer von lokal definierten Variablen ist in Java ausschließlich `final` erlaubt.

Konstanten

Mit dem Modifizierer `final` definieren Sie in Java eine **Konstante**, wobei Java strenggenommen gar keine Konstanten im üblichen Sinn besitzt. Mit dem Voranstellen des Schlüsselworts `final` wird gesagt, dass sich die Variable nicht mehr ändert[6]. Zusätzlich müssen Sie dieser Variablen einen Anfangswert zuweisen (der sich dann auch nie mehr ändert – sonst hätten wir ja keine Konstante). Sie sollten beachten, dass man in Java per Konvention festgelegt hat, dass eine Konstante immer vollständig groß geschrieben wird!

Info

Es gibt sowohl lokale Konstanten als auch Konstanten im Namensraum der Klasse. Letztere werden in der Regel als `static` deklariert, denn deren Wert wird sich ja in unterschiedlichen Instanzen nicht unterscheiden[7]. Beispiel:

Listing 3.6 Eine Klassenvariable als Konstante

```
final static int TEST = 42;
```

Statische Initialisierer

Wenn Sie statische Variablen deklarieren, können Sie die Initialisierung direkt mit angeben, so weit nichts Neues. Nur die Zuordnung von Werten für statische Variablen kann auch in einen **statischen Initialisierer** »ausgelagert« werden. Er beginnt mit dem Schlüsselwort `static` und dahinter folgt ein Block. Innerhalb der geschweiften Klammern können dann die gewünschten Initialisierungen vorgenommen werden. Etwa so:

Listing 3.7 Ein statischer Initialisierer

```
static int i;
static double j;
static String[] k;
static {
   i = 3;
   j = 3.1415;
   k = new String[]{"Hans", "Dampf"};
}
```

Ein Beispiel für eine Klasse mit Eigenschaften

Erweitern wir die bisher ziemlich leere Klasse `Klasse1` einmal um die Deklaration von zwei Attributen, die ganze Zahlen[8] repräsentieren sollen. Eines der Attribute soll ein Klassenelement darstellen, das zweite Attribut ein Instanzelement.

---

6 Was von der Aussagelogik her eigentlich ein Widerspruch zu der Bedeutung einer Variablen ist.
7 Das ist aber nicht zwingend – Sie können `final` auch vor Instanzvariablen notieren.
8 Sogenannte primitive Datentypen.

```
01 public class Klasse1 {
02   int attr1;
03   static int attr2;
04 }
```

Listing 3.8
Definition von zwei Eigenschaften/Attributen

In Zeile 2 sehen Sie die Definition eines Instanzattributs (da kein `static` davor) und in Zeile 3 wird ein Klassenattribut angelegt (mit `static`).

Die Methodendeklaration

Bei **Methoden** verhält es sich bezüglich der Verfügbarkeit ähnlich wie bei Attributen. Alle Objekte einer Klasse verfügen über die gleichen Methoden, die die Funktionalität der Objekte realisieren. Es gibt Methoden zur Ausführung irgendwelcher Aktionen, zur Abfrage von Informationen, zur Modifikation von Werten oder zur Änderung des Zustands eines Objekts. Das **Verhalten** eines Objekts ist also festgelegt durch eine Menge von Methoden, die im Allgemeinen auf den Daten des Objekts operieren. Die Spezifikation von Methoden erfolgt über

- einen Methodennamen,
- die Zugänglichkeit,
- eine Argumentliste,
- einen Rückgabetyp,
- gegebenenfalls vertragsbasierte Vereinbarungen (Regeln) zum Aufruf und
- die konkrete Implementierung.

Die Argumentliste besteht dabei aus Argumenten, für die sowohl ein Argumentname als auch ein Argumenttyp festgelegt werden. Wenn man die konkrete Implementierung nicht aufführt, redet man von der **Methodenunterschrift** bzw. **Methodensignatur**. Dies bezeichnet also neben dem Methodennamen samt Zugänglichkeit die Anzahl und Typen der Argumente einer Argumentliste sowie den optionalen Rückgabetyp. Die allgemeine Schreibweise einer Methodensignatur, die ebenfalls im Rahmen eines UML-Klassendiagramms Anwendung findet, sieht so aus:

```
methodenName(argName1:argTYP1, argName2:argTYP2, ...) :
  rückgabeTyp
```

Listing 3.9
Eine formale Beschreibung einer Methode

Info

Die mehrfach erwähnte Zugänglichkeit zu Attributen und Methoden kann in der OOP über einen Zugriffsschutz geregelt werden. Das wird dann in der formalen Beschreibung des Elements notiert. Die Details dieses Themas wollen wir aber gesondert behandeln.

Die Deklaration einer Methode können Sie an einer beliebigen Stelle im Namensraum einer Klasse vornehmen, aber nicht innerhalb einer anderen Methode[9]. Die schematische Deklaration einer Methode sieht konkret in Java generell folgendermaßen aus:

```
[<Modifiziererliste>] <Typ des Rückgabewerts> <NameMethode>
([<Parameterliste>]) [throws] [<ExceptionListe>]
```

Dabei ist alles in eckigen Klammern optional. Die Methodenunterschrift oder Methodensignatur besteht also aus mindestens dem Namen der Methode, dem Rückgabetyp und den Klammern.

Info

Die als letzte Angabe zu findende `ExceptionListe` steht für eine Liste mit sogenannten Ausnahmen oder Exceptions. Exceptions sind Bestandteil eines Java-Konzepts, wie auf nicht planbare Situationen reagiert werden kann. Sie gehören zum Konzept der vertragsbasierten Programmierung.

Die **Rückgabewerte** von Java-Methoden können von jedem erlaubten Datentyp sein, nicht nur primitive Datentypen, sondern auch komplexe Objekte. Eine Methode in Java muss immer einen Wert zurückgeben (und zwar genau den Datentyp, der in der Deklaration angegeben wurde), es sei denn, sie ist mit dem Schlüsselwort `void` deklariert worden.

Info

Das Schlüsselwort `void` bedeutet gerade, dass eine Methode keinen Rückgabewert hat.

return und die Rückgabewerte

Rückgabewerte werden in Java mit der Anweisung `return` zurückgegeben. Hinter dem Schlüsselwort steht der gewünschte Rückgabewert. Bei der `return`-Anweisung handelt es sich um eine **Sprunganweisung**, die beim Aufruf unmittelbar eine Methode verlässt. Bei einer als `void` deklarierten Methoden kann man die `return`-Anweisung auch verwenden – nur dann ohne nachgestellten Rückgabewert. Das ist etwa dann sinnvoll, wenn ein Rücksprung aus einer solchen Methode mit bestimmten Bedingungen gekoppelt wird, aber kein Ergebnis an den Aufrufer geben muss oder soll.

Die Parameter als lokale Variablen

Die optionalen **Parameter** einer Methode sind Variablen, die in einer Methodendeklaration in der Signatur eingeführt werden und dann über diesen Namen im Inneren der Methode als **lokale Variablen** zur Verfügung stehen. Bei der Deklaration werden in der Methodenunterschrift der Typ und ein Bezeichner für jeden Parameter angegeben. Mehrere Parameter werden durch Kommata getrennt. Beim Aufruf werden die Bezeichnernamen durch Werte des entsprechenden Typs ersetzt.

---

9 Das wären sogenannte **Closures** und die sind erst für Java 8 geplant. Solche Closures kennt man etwa aus JavaScript. Dort wurde dieses Konzept in neuen Versionen eingeführt und erwies sich als ein großer Wurf. Denn JavaScript kennt keine Zugriffsmodifizierer und damit kann man die Zugänglichkeit auf Funktionalitäten steuern.

Achtung

Beachten Sie, dass die Klammern unbedingt bei einer Methode auftauchen müssen. Auch wenn keine Parameter an die Methode übergeben werden. Dann bleiben die Klammern leer.

Das Beispiel für eine Klasse mit Methoden erweitern

Erweitern wir unsere Klasse `Klasse1` noch um zwei Methoden, wobei sowohl ein Klassenelement als auch ein Instanzelement verwendet werden soll.

Listing 3.10 Erweiterung der Klasse um zwei Methoden

```
01 public class Klasse1 {
02   int attr1;
03   static int attr2;
04   void init() {
05     attr1 = 2;
06     attr2 = 5;
07   }
08   static void ausgabe() {
09     System.out.println(attr2);
10   }
11 }
```

In den Zeilen 4 bis 7 wird eine Instanzmethode `init()` definiert, die den Wert beider Eigenschaften setzt.

Achtung

Beachten Sie, dass innerhalb einer Klasse Instanzmethoden immer sowohl auf Instanzelemente als auch Klassenelemente (sowohl Attribute als auch Methoden) in der Klasse zugreifen können, während Klassenmethoden immer nur auf die Klassenelemente zugreifen können. Das kann deswegen auch nicht möglich sein, weil zur »Lebenszeit« einer Klasse ja nicht unbedingt eine Instanz davon existieren muss und sich zudem die Werte von Instanzvariablen je nach Instanz unterscheiden können. Dies zu verstehen, ist grundlegend. Oder anders ausgedrückt – wenn Ihnen das klar ist, haben Sie Instanz- und Klassenelemente richtig verstanden, aber auch nur dann. Wir spielen ein Beispiel durch, das Ihnen (hoffentlich) hilft.

Von Zeile 8 bis 10 sehen Sie die Deklaration einer Klassenmethode `ausgabe()`, die den Wert des Klassenelements `attr2` ausgeben wird.

Info

Die Reihenfolge von Deklarationen in einer Klasse spielt in Java allgemein keine Rolle! Sie können zuerst die Variablen deklarieren und dann die Methoden oder umgekehrt. Sie können die einzelnen Deklarationsarten mischen. Es ist aber in der Regel übersichtlicher, wenn Sie zuerst die Variablen deklarieren und dann die Methoden. Eine gute IDE erlaubt aber sowohl eine gefilterte Ansicht auf die verschiedenen Typen einer Deklaration als auch die Umsortierung, obwohl dies für den Compiler uninteressant ist.

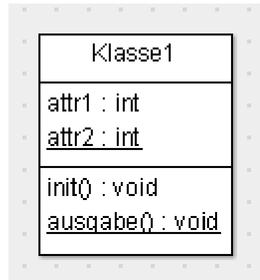


Abbildung 3.7 Das UML-Klassendiagramm zeigt die beiden Eigenschaften und die Methoden – beachten Sie die Unterstreichung von Klassenelementen.

Nun können Sie die gerade deklarierte Klasse verwenden. Nur wie?

Eine besondere Klasse – die Programmklasse und die Methode main()

Betrachten Sie das nachfolgende Listing von *KlassenSchreiben.java*:

Listing 3.11 Eine Klasse mit einer main()-Methode

```
01 public class KlassenSchreiben {
02   public static void main(String[] args) {
03     ;
04   }
05 }
```

Diesen Aufbau kennen Sie von unseren ersten Java-Beispielen. Im Inneren der Klasse `KlassenSchreiben` finden Sie eine **statische** Methode mit Namen `main()`. Diese Methodendeklaration zeichnet diese Klasse aus. Denn diese Klasse kann als **Java-Programm** aufgerufen werden. Oder noch genauer – jedes (normale) Java-Programm benötigt zwingend eine Klasse mit einer `main()`-Methode, die formal genau so aussehen muss wie in dem Beispiel.

Aber nicht jede Klasse enthält einen Einsprungspunkt für ein Programm, wie etwa unsere Klasse `Klasse1`. Ganz im Gegenteil, die meisten Klassen in Java werden nur Hilfsmittel sein, die im Rahmen eines Programms verwendet werden. Das gesamte Standard-API von Java mit seinen zig Tausenden von Klassen besteht aus solchen Klassen, die selbst kein Programm darstellen.

Gehen wir einmal den Ablauf eines (normalen) Programmablaufs unter Java durch:

1. Der Interpreter *java* wird aufgerufen und erhält als Übergabeparameter den Namen einer Klasse mit einer `main()`-Methode. Er startet die Java Virtual Machine, in der die Klasse geladen und deren Bytecode dann interpretiert wird.
2. Der Interpreter sucht in der Klasse nach einer Klassenmethode mit der Signatur `public static void main(String args[])`. Findet er diese Signatur nicht, bricht der Interpreter mit einer Fehlermeldung ab. Findet er sie, wird das Programm mit dem ersten Befehl innerhalb der `main()`-Methode gestartet. Die Methode `main()` kann ohne Objekt verwendet werden, denn sie ist ja explizit eine Klassenmethode. Als Startmethode ist das unabdingbar, denn damit wird quasi das Problem gelöst, ob zuerst

die Henne oder das Ei da sein müssen. Denn um überhaupt irgendeine Instanz zu erzeugen, muss erst einmal ein Programm laufen.

3. Alle Anweisungen in der `main()`-Methode werden der Reihe nach abgearbeitet[10]. Bei Bedarf lädt der Interpreter weitere Klassen dynamisch nach. Diese können zwar `main()`-Methoden enthalten, aber das wird fast nie der Fall sein. Und wenn diese nachgeladenen Klassen eine `main()`-Methode enthalten und sie diese wirklich aufrufen[11], wird das nur als »normaler« Aufruf einer Klassenmethode verstanden.
4. Nach der letzten Anweisung in der `main()`-Methode wird das Programm beendet.

Objektorientiert kann man das so ausdrücken: Der Interpreter erzeugt aus der Klasse, die das Programm beschreibt, ein Objekt im Hauptspeicher des Rechners, das das Programm repräsentiert und von dem aus alle anderen Objekte erzeugt und dann verwendet werden. Ist das Programm zu Ende, wird das das Programm repräsentierende Objekt wieder aus dem Hauptspeicher entfernt. Bei anderen Formen von Java-Applikationen wie Java-Applets oder Java-Servlets läuft der Vorgang zwar im Detail etwas anders, aber dennoch verwandt ab.

Sie haben also hier bereits einen Weg gesehen, wie aus einer Klasse (wenngleich einer sehr besonderen) ein Objekt (dasjenige, das das Programm selbst repräsentiert) entstehen kann. Im Allgemeinen reicht das nicht, denn wie gesagt – das ist nur das Verfahren, wie ein Programmlauf und das zugehörige Objekt in Beziehung gebracht werden.

Methoden mit variabler Argumentanzahl (Varargs)

In der Regel ist es bei der Deklaration einer Methode klar, wie viele Argumente sie haben muss. Allerdings kann es Fälle geben, in denen eine **variable Anzahl an Parametern** vorkommen kann. Vor Java 5 musste man dann entweder mehrere Methoden mit gleichem Namen und einer unterschiedlichen Anzahl an Parametern deklarieren[12] oder ein Array als Parameter angeben. In Java 5 wurde zusätzlich ein Konzept eingeführt, um Methoden mit variabler Anzahl an Parametern mit einem eigenständigen Token deklarieren zu können. Diese werden umgangssprachlich ***Varargs*** genannt. Es wird nach dem Datentyp des Parameters und vor dessen Bezeichner ein Token aus drei Punkten (`...`) notiert.

Info

Dieses Konzept kennt man bereits länger in einigen Konkurrenztechnologien. Vermutlich wurde es deswegen in Java eingeführt, um das Umsteigen von solchen Sprachen zu erleichtern. Technisch sehe ich keinen Grund, warum man Varargs braucht. Denn es wird schlicht und einfach ein Array übergeben – nur als Argument mit einer weiteren, speziellen Schreibweise formuliert.

10 In der bisherigen Version unseres Beispiels sehen Sie in Zeile 3 nur ein Semikolon. Das ist eine leere Anweisung, die rein gar nichts tut und gelegentlich als Platzhalter notiert wird. In Java ist so eine leere Anweisung vollkommen legitim.

11 Was man aber nie machen sollte und auch nie machen muss.

12 Das nennt man Überladen.

Im Inneren können Sie diesen Parameter dann als Array verwenden. Wir greifen etwas vor, aber im nächsten Beispiel soll die Methode `test()` als Vararg definiert werden und im Inneren geben wir mit einer Schleife alle Werte in dem Übergabearray aus (Projekt *Varargs*).

Listing 3.12 Varargs

```
01 public class Varargs {
02  void test(int... a) {
03    for (int i = 0; i < a.length; i++) {
04      System.out.println(a[i]);
05    }
06  }
07 }
```

Und so könnte man die Methode verwenden:

Listing 3.13 Aufruf einer Methode mit unterschiedlicher Anzahl an Parametern

```
01 public class NutzerVarargs {
02  public static void main(String[] args) {
03    Varargs obj = new Varargs();
04    obj.test(1,2,3);
05    obj.test(7,8);
06  }
07 }
```

Sie sehen in Zeile 4, dass die Methode mit drei Parametern aufgerufen wird, und in Zeile 5 mit zwei Parametern.

Achtung

Wenn Sie mit Varargs arbeiten wollen und zusätzliche Parameter in der Methode benötigen (etwa von einem anderen Typ – aber das ist nicht zwingend), so dürfen Sie das Vararg-Array nur als letzten Parameter notieren. Andernfalls kann der Compiler beim Aufruf der Methode nicht zuverlässig zuordnen, was ein normaler Parameter ist und was schon zum Vararg gezählt werden muss (insbesondere bei gleichen Datentypen).

3.2.6 Klassenelemente verwenden

Erweitern wir unser Programm *KlassenSchreiben.java* wie folgt:

Listing 3.14 Das Programm verwendet Elemente aus anderen Klassen.

```
01 public class KlassenSchreiben {
02   public static void main(String[] args) {
03     Klasse1.attr2=42;
04     Klasse1.ausgabe();
05   }
06 }
```

Sie sehen in Zeile 3, dass wir über den Namen der Klasse `Klasse1` den Wert des Klassenattributs `attr2` setzen und dann in Zeile 4 den Wert dieses Klassenattributs über die Klassenmethode `ausgabe()` – wieder mit dem vorangestellten Namen der Klasse `Klasse1` – ausgeben.

Sie sehen also hier, wie Sie auf Klassenelemente in fremden Klassen zugreifen können, wenn Sie Zugriff auf die fremde Klasse haben. Diese muss sich im gleichen Verzeichnis wie die zugreifende Klasse befinden oder allgemein im sogenannten Klassenpfad, über den wir uns später noch unterhalten werden.

Tipp

Wenn Sie eine Klassenmethode aufrufen wollen, die innerhalb der Klasse definiert ist, wo der Aufruf erfolgt, können Sie auf den vorangestellten Namen der Klasse verzichten.

Natürlich gibt es auch hier noch Details, die wir ausbauen müssen und werden. Erweitern wir das Beispiel noch ein wenig:

Listing 3.15 Das Programm verwendet ein Klassenelement aus der eigenen Klasse.

```
01 public class KlassenSchreiben {
02   static int attr3 = 7;
03   public static void main(String[] args) {
04     Klasse1.attr2 = 42;
05     Klasse1.ausgabe();
06     System.out.println(attr3);
07   }
08 }
```

In Zeile 2 deklarieren wir eine Klassenvariable *attr3* des Programms selbst. Auch das ist natürlich möglich. Wir könnten hier auch von einem Klassenattribut oder einer Klasseneigenschaft reden, denn im Prinzip kann man die Variable `attr3` von außen sehen und über die Klasse `KlassenSchreiben` aus einer anderen Klasse heraus verwenden. Nur handelt es sich ja hier um unserer Programm selbst und aus diesem heraus ist das als Variable zu sehen.

Beachten Sie nun die Zeile 6. Hier greifen wir auf die Klassenvariable `KlassenSchreiben.attr3` zu, aber **ohne** (!) die vorangestellte Klasse! In dem Fall ist das wie erwähnt möglich, weil wir uns bereits in der Klasse `KlassenSchreiben` befinden.

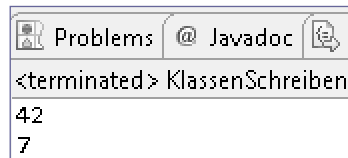


Abbildung 3.8 Die Ausgaben des Programms in der Console-View von Eclipse

3.2.7 Konstruktoren und Destruktoren – Objekte erzeugen und beseitigen

Wie jede Programmiersprache muss Java das Management des Hauptspeichers zur Laufzeit eines Programms bewerkstelligen. Java verfügt über ein ausgefeiltes **Speichermanagement**, das nicht nur die Allokierung von Speicher bei der Nutzung von Klassen wie bei der Erzeugung von Objekten und Variablen betrifft, sondern auch die Speicherfreigabe. So gut wie sämtliche Speicherverwaltung erfolgt automatisch oder mit einfachster Syntax.

Verallgemeinern wir nun die Möglichkeiten, wie wir eine Klasse nutzen können. Bisher verwenden wir ja nur die Klassenelemente. Das ist aber eher die Ausnahme. In der Regel verwendet man eine spezifische Instanz einer Klasse und deren Instanzelemente. Allgemein erzeugt man in der OOP Objekte mittels sogenannter **Konstruktoren** (engl. Constructors), von denen es in jeder (!) Klasse per Definition mindestens einen gibt (der Default-Kon-

struktor). Dies ist ein entscheidender Faktor der OOP und in Java essenzieller Bestandteil des Konzepts.

Bei Konstruktoren handelt es sich um sehr spezielle Methoden (deshalb wird oft von Konstruktormethoden gesprochen), deren einzige Aufgabe die Erstellung einer Instanz einer Klasse ist. Dabei wird ein Objekt initialisiert, bestimmte Eigenschaften der Instanz werden festgelegt, bei Bedarf werden notwendige Aufgaben ausgeführt und vor allem Speicher für die Instanz allokiert. Konstruktoren sind damit explizit ein Teil des Speichermanagements von Java. Wenn Sie mit einem Konstruktor eine neue Instanz einer Klasse erstellen, belegt das Laufzeitsystem von Java dafür einen Teil des Speichers, in dem die zu der Klasse beziehungsweise ihrer Instanz gehörenden Informationen abgelegt werden.

Und ein Konstruktor kann eine Referenz auf eine Instanz liefern, die einem Identifikator für das Objekt zugewiesen werden kann. Darüber ist das Objekt dann später zugänglich. Sie initiieren zwar damit für jedes neue Objekt eine Zuweisung von Speicherplatz, spezifizieren jedoch nicht die benötigte Größe des Speicherplatzes, sondern nur noch den Namen des benötigten Objekts, oder können gar anonym arbeiten.

Achtung

Eine elegante und vor allem praktisch anzuwendende Regel von Java ist, dass Konstruktoren in Java immer **den gleichen Bezeichner** wie die Klasse selbst haben müssen! Nun erinnern Sie sich, dass Klassennamen immer mit einem Großbuchstaben beginnen sollen, Methodennamen jedoch immer mit einem Kleinbuchstaben. Wenn Sie also einen Methodennamen sehen, der mit einem Großbuchstaben beginnt, kann es sich nur um eine Konstruktormethode handeln – solange Sie (oder der fremde Programmierer) diese Konventionen nicht gebrochen haben. In UML werden Konstruktoren mit *<<create>>* gekennzeichnet. Das macht recht deutlich, was deren Zweck ist.

Obwohl Konstruktoren Methoden sind, werden sie nicht wie gewöhnliche Methode aufgerufen (obgleich das prinzipiell möglich wäre – nur würde dann kein Objekt erzeugt). Konstruktoren werden in Java immer in Verbindung mit einem vorangestellten Schlüsselwort `new` eingesetzt. Das bedeutet ebenfalls, dass mit der `new`-Anweisung bereits der richtige Datentyp zugewiesen wurde, wenn sie eine Referenz auf einen zugewiesenen Speicherbereich zurückgibt. Der Default-Konstruktor unserer Klasse `Klasse1` wird damit so eingesetzt:

Listing 3.16 Aufruf des Default-Konstruktors der Klasse Klasse1

```
new Klasse1();
```

Mit dem Aufruf des Konstruktors wird ein Objekt vom Typ `Klasse1` erzeugt. Beachten Sie die Klammern. `Klasse1`**()** bezeichnet **nicht** den **Namen der Klasse**, sondern den (identischen) **Namen der Konstruktormethode**. Eine Klasse oder eine Variable kann nie ein Klammerpaar hinter dem Bezeichner stehen haben, eine Methode dagegen muss in Java immer (!) ein solches Klammerpaar dort stehen haben. Sie haben damit ein eindeutiges Unterscheidungskriterium (unabhängig von der Verwendung im Quelltext, die ebenso eindeutig ist), ob es sich um den Klassenbezeichner oder den Konstruktor handelt.

Damit Sie nun das so erzeugte Objekt nutzen können, wird meist[13] eine Referenz darauf einer Variablen zugewiesen. Über deren Namen kommen Sie dann an den Speicherplatz, in dem das Objekt abgelegt wird. Diese Variable muss für die Aufnahme eines ganz bestimmten Objekts eingerichtet werden. Sie bekommt dazu bei der Deklaration einen passenden Datentyp (dazu wird noch mehr folgen). In Java wird – wie schon ausgeführt – eine Variable so eingerichtet, dass zuerst der Datentyp notiert wird und dann der Bezeichner für die Variable. Das haben wir in der Klasse *Klasse1* bereits mit primitiven Datentypen gemacht.

Wenn eine Instanz einer Klasse erstellt wird, ist es nötig, dass ein Speicherbereich für verschiedene Informationen reserviert wird. Wenn Sie eine Variable für eine Instanz am Anfang einer Klasse deklarieren, dann sagen Sie dem Compiler damit lediglich, dass eine Variable eines bestimmten Namens in dieser Klasse verwendet wird. Eine Variable vom Typ einer Klasse ist in Java ein Alias für einen 32-Bit-Zeiger, eine **Referenz** auf einen Speicherbereich, in dem das konkrete Objekt abgelegt wird. Die Deklaration der Variablen ist dennoch eine gewöhnliche Variablendeklaration. Der Typ der Variablen ist vom Typ des Objekts, auf das damit referenziert werden soll. Die Syntax sieht folgendermaßen aus:

```
KlassenName Klasseninstanz;
```

Listing 3.17
Schema einer Deklaration

Info

Die Klasse wird auch als Typ bezeichnet. Und die so angelegte Variable ist eine Referenzvariable. Von daher redet man oft vom Referenztyp.

Wenn also eine Variable ein Objekt vom Typ der Klasse `Klasse1` aufnehmen (darauf verweisen) soll, muss die Deklaration der Variablen so erfolgen:

```
Klasse1 ob1;
```

Listing 3.18
Die Variablendeklaration

Der Bezeichner `ob1` ist der Name der Variablen, der das Objekt zugewiesen werden soll. Der gesamte Vorgang sieht also so aus:

```
Klasse1 ob1;
ob1 = new Klasse1();
```

Nun kann man die Erzeugung des Objekts und die Variablendeklaration zu einer Quellcodezeile zusammenfassen. Dies ist gängige Java-Praxis und es ist wichtig, dass Sie sich daran gewöhnen:

```
Klasse1 ob1 = new Klasse1();
```

Achtung

OOP-Einsteiger monieren jetzt oft, da würde ja auf beiden Seiten der Gleichung das Gleiche stehen. Das wären indessen zwei gravierende Denkfehler. Es handelt sich erst einmal nicht um eine Gleichung bzw. einen Vergleich, sondern eine Zuweisung. In dieser Zuweisung wird das, was auf der rechten Seite steht, dem, was auf der linken Seite steht, zugewiesen. Wichtiger im Moment ist jedoch, dass nicht das Gleiche auf beiden Sei-

13 Sie können ein Objekt auch anonym verwenden. Dazu kommen wir noch.

ten steht. Wir haben es ja gerade besprochen, dass mit den Klammern die Konstruktormethode bezeichnet wird und ohne die Klammern die Klasse.

Das Beispiel um die Erzeugung von Instanzen erweitern

Wenden wir uns der Vervollständigung des Beispiels zu. Die Klasse, die das Programm repräsentieren wird, soll nun so aussehen:

Listing 3.19
Erzeugen eines Objekts und Verwenden einer Instanzeigenschaft

```
01 public class KlassenSchreiben {
02   static int attr3 = 7;
03   public static void main(String[] args) {
04     Klasse1.attr2 = 42;
05     Klasse1.ausgabe();
06     System.out.println(attr3);
07     Klasse1 ob1 = new Klasse1();
08     System.out.println(ob1.attr1);
09   }
10 }
```

In der `main()`-Methode wird in Zeile 7 ein Objekt `ob1` der Klasse `Klasse1` erzeugt und dann der Wert der Instanzeigenschaft `attr1` in Zeile 8 über das Objekt `ob1` ausgegeben. Der Wert dieser Eigenschaft ist noch nicht explizit angegeben worden und hat den Vorgabewert 0[14].

Abbildung 3.9
Ausführen der neuen Version unseres Programms

Tipp

Wenn Sie die Objektinstanz einer Klasse nur einmal benötigen, um etwa den Wert einer Eigenschaft auszulesen oder eine Methode aufzurufen, muss man auf die Objektinstanz keine Referenz legen. Sie können mit dem Konstruktor die Instanz erzeugen und direkt per Punktnotation die gewünschte Eigenschaft oder Methode nachstellen, etwa so:

Listing 3.20
Die Eigenschaft attr1 wird direkt verwendet.

```
System.out.println(new Klasse1().attr1);
```

So etwas nennt man eine **anonyme Instanzierung**.

Weisen wir der Instanzvariablen `attr1` nun einen Wert zu und erzeugen wir ein zweites Objekt vom Typ `Klasse1`. Anschließend geben wir über beide Objekte den Wert von `attr1` aus.

Listing 3.21
Zwei Instanzen der Klasse Klasse1

```
01 public class KlassenSchreiben {
02   static int attr3 = 7;
03   public static void main(String[] args) {
04     Klasse1.attr2 = 42;
05     Klasse1.ausgabe();
```

14 Sie erinnern sich? Jede numerische Instanzeigenschaft hat den Nullwert als Vorgabewert.

```
06      System.out.println(attr3);
07      Klasse1 ob1 = new Klasse1();
08      ob1.attr1 = 666;
09      Klasse1 ob2 = new Klasse1();
10      System.out.println(ob1.attr1);
11      System.out.println(ob2.attr1);
12    }
13 }
```

Listing 3.21 (Forts.) Zwei Instanzen der Klasse Klasse1

Der Instanzvariablen `attr1` des Objekts `ob1` wird in Zeile 8 der Wert 666 zugewiesen. Dass dies funktioniert, sehen Sie an der Ausgabe von Zeile 10. Die Ausgabe der Instanzvariablen `attr1` des Objekts `ob2` zeigt jedoch, dass diese Zuweisung das zweite Objekt nicht tangiert. Instanzelemente sind explizit auf ihr jeweiliges Objekt beschränkt.

Abbildung 3.10 Die Instanzen sind vom gleichen Typ, aber streng getrennt.

Nun erweitern wir das Listing um folgende Zeilen:

```
...
12      System.out.println(ob1.attr2);
13      System.out.println(ob2.attr2);
14    }
15 }
```

Listing 3.22 Zugriff auf die Klasseneigenschaften

Die Eigenschaft `attr2` ist in der Klasse `Klasse1` als `static` deklariert. Damit können Sie darauf sowohl über die Klasse `Klasse1` als auch jede Instanz dieser Klasse zugreifen. Egal, wie der Zugriff erfolgt – es ist immer die gleiche Stelle, die Sie erreichen. Und jede Änderung – gleich, wie Sie das machen – wirkt sich sowohl in der Klasse als auch in sämtlichen Instanzen dieser Klasse aus.

Abbildung 3.11 Der Weg des Zugriffs auf die Klassenvariable ist egal – der Wert ist immer gleich.

Achtung

Der Zugriff auf ein Klassenelement sollte immer über die Klasse und nicht über eine Instanz der Klasse erfolgen – obgleich das möglich ist, wie das Beispiel und der zugehörende Screenshot beweisen. Allerdings sehen Sie auch, dass Eclipse gar nicht glücklich mit dieser Art des Zugriffs ist. Am Rand des Editorbereichs sehen Sie Warnhinweise. Wenn Sie diese anklicken, erhalten Sie den Hinweis »The static field Klasse1.attr2 should be accessed in a static way«, was die bessere Lösung für diese Situation genau beschreibt.

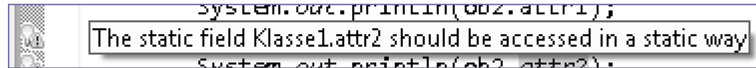


Abbildung 3.12 Klassenelemente sollten nur statisch referenziert werden.

Abschließend versuchen wir einen verbotenen Zugriffsweg – aus einer Klassenmethode soll auf eine Instanzvariable in der gleichen Klasse zugegriffen werden. Und das geht nicht. Betrachten Sie diese Anweisungen:

Listing 3.23 Fehlerhafter Zugriff auf die Instanzmethode

```
...
12      System.out.println(ob1.attr2);
13      System.out.println(ob2.attr2);
14    Klasse1.init();
15    }
16 }
```

Wenn Sie versuchen, die Instanzmethode `init()` über die Klasse `Klasse1` und nicht über eine spezifische Instanz der Klasse aufzurufen, wird in Eclipse ein Fehler angezeigt.

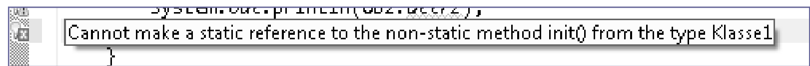


Abbildung 3.13 Eclipse macht deutlich, wo das Problem liegt.

Das ist keine einfache Warnung wie im Fall des Zugriffs auf eine Klassenvariable über eine Instanz, was man nicht machen *sollte*. Das ist schlicht und einfach Unsinn und der Compiler wird den Code nicht übersetzen! Sie können in einer Fabrik, in der Autos produziert werden, nicht bereits festlegen, welchen Weg die produzierten Autos zu fahren haben. Hingegen wird das gehen:

Listing 3.24 Zugriff auf die Instanzmethode init() über zwei verschiedene Objekte

```
...
06      System.out.println(attr3);
07      Klasse1 ob1 = new Klasse1();
08      ob1.init();
09      Klasse1 ob2 = new Klasse1();
10      ob2.init();
11      System.out.println(ob1.attr1);
12      System.out.println(ob2.attr1);
13      System.out.println(ob1.attr2);
14      System.out.println(ob2.attr2);
15      ob1.init();
16    }
17 }
```

In der Zeile 8 wird über das erste Objekt und in Zeile 10 über das zweite Objekt auf die Instanzmethode `init()` zugegriffen. Im ersten Fall wird das Objekt `ob1` initialisiert und im zweiten Fall das Objekt `ob2`.

Den Default-Konstruktor redefinieren

Sie können das Verhalten des Default-Konstruktors in einer Klasse redefinieren, indem Sie ihn in der Klasse neu hinschreiben. Im Inneren der Implementierung machen Sie dann etwas Sinnvolles. Etwa so:

```
class A {
  A() {
    System.out.println("Das Objekt wird erzeugt");
  }
  ...
 }
```

Listing 3.25
Schema für eine Redefinition des Default-Konstruktors

In der Regel wird man im Inneren des Konstruktors spezifische Eigenschaften einer konkreten Instanz initialisieren.

Zugriff auf das Objekt selbst und das Schlüsselwort this

In manchen Situationen möchte man in einer Klasse auf eine Instanz von sich selbst bzw. darin enthaltene Instanzelemente zugreifen. Nur wie geht das? Denn an dieser Stelle wird noch keine konkrete Instanz davon existieren und genauso kann selbstverständlich nicht bekannt sein, über welchen Bezeichner die konkrete Instanz später verfügbar gemacht wird. Es kann sogar sein, dass eine Instanz anonym eingesetzt wird. Das bedeutet, direkt an den Aufruf des Konstruktors mit vorangestelltem `new` wird ein Punkt notiert und eine verfügbare Methode oder Eigenschaft benutzt. Etwa so:

```
new javax.swing.JFrame().setVisible(true)
```

Listing 3.26
Ein Objekt wird erzeugt, aber es gibt keinen Bezeichner, über den man zugreifen kann.

In allen beschriebenen Situationen besteht das Dilemma, dass eine Klasseninstanz nicht über einen individuellen Bezeichner angesprochen werden kann. Aber es gibt in Java das Schlüsselwort `this` zum Zugriff auf die aktuelle Instanz einer Klasse aus der aktuellen Instanz der Klasse heraus.

Achtung

Das Schlüsselwort `this` kann nur in der Implementierung einer nichtstatischen Methode verwendet werden, denn Sie wollen ja direkt auf eine Instanz der aktuellen Klasse zugreifen und statische Methoden können – wie besprochen – nicht direkt auf Instanzelemente zugreifen.

Es gibt im Allgemeinen mehrere Situationen, die den Gebrauch von `this` rechtfertigen bzw. notwendig machen:

- Unabdingbar ist der Einsatz in dieser Situation: Es gibt in einer Klasse zwei Variablen mit gleichem Namen – eine gehört als Instanzvariable zum Namensraum der Klasse, die andere zum Namensraum einer spezifischen Instanzmethode in der Klasse[15]. Die Benutzung der Syntax `this.`*VariablenName* ermöglicht es, eindeutig auf diejenige Variable zuzugreifen, die zum Namensraum der Klasse gehört. Ohne `this` wird immer

15 Also eine lokale Variable.

die lokale Variable angesprochen. So eine doppelte Verwendung eines Bezeichners für eine Instanzvariable und eine lokale Variable in einer Methode kommt in der Praxis sehr oft vor, besonders im Zusammenhang mit Übergabewerten an eine Methode, wenn diese den Wert einer Instanzvariablen setzt[16]. Es ist also gängige Praxis, die Bezeichner von Variablen der Übergabewerte an Methoden identisch zu den Bezeichnern von Instanzvariablen zu wählen, um eine logische Zuordnung bereits im Quelltext erkennen zu können. Java stellt dafür Namensräume bereit, die eine Eindeutigkeit gewährleisten.

- In vielen Syntaxkonstruktionen wird `this` implizit verwendet. Sie notieren etwa in einer Instanzmethode den Namen einer anderen Instanzmethode der gleichen Klasse oder den Bezeichner einer Instanzvariable in der Klasse, die nicht lokal verdeckt wird. Dann können Sie auf `this` verzichten, denn implizit wird das System das gleiche Objekt als Ziel der Botschaft verwenden. Aber man notiert `this` oft dennoch explizit. Sie wollen dann rein von der Lesbarkeit ausdrücklich deutlich machen, dass Sie ein Instanzelement ansprechen. Mit dem Schlüsselwort ist die Notation eindeutiger.
- Der Parameter für eine Methode oder der Rückgabewert ist eine Objektinstanz der aktuellen Klasse.
- Man nutzt `this` auch zum anonymen Zugriff auf ein Objekt. Vor allem dann, wenn man die Funktionsweise von Konstruktoren modifiziert. Dort können Sie mit `this()` samt optionaler Parameter direkt einen anderen Konstruktor der gleichen Klassen aufrufen, wenn der Konstruktor überladen[17] ist.

Tipp

Bessere Java-Editoren und IDEs helfen mit einer Auswahlliste der erlaubten Methoden und Eigenschaften des über `this` referenzierten Objekts. Sie schreiben `this` und dann einen Punkt und der Editor bietet die gleiche Hilfe an, die beim Hinschreiben eines expliziten Objektnamens und eines Punkts angeboten würde.

Das Schlüsselwort `this` kann allgemein so verstanden werden, dass es im Quelltext an jeder Stelle verwendet werden kann, an der das Objekt der jeweiligen Klasse erscheinen kann, dessen Methode gerade aktiv ist, etwa als Argument einer Methode, als Ausgabewert oder in Form der Punktnotation zum Zugriff auf eine Instanzvariable. In vielen Anweisungen steckt das Schlüsselwort wie gesagt implizit drin, denn man kann oft darauf verzichten, wenn die Situation eindeutig ist.

Die nachfolgenden Beispiele zeigen die Verwendung von `this` in verschiedenen Varianten. Die Klasse `This1` zeigt, wie Sie innerhalb einer Methode auf eine Variable der Klasse zugreifen können. Beachten Sie, dass die Instanzvariable in der Klasse den gleichen Namen hat wie die als Parameter übergebene Variable. Ohne das vorangestellte `this` wären die Zuweisungen

16 Ein sogenannter Setter.
17 Wir müssen natürlich noch besprechen, was ***überladen*** bedeutet.

in der Methode `mMethode(int x, String y)` ***Selbstzuweisungen*** der lokalen Variablen. So bedeutet die Zeile `this.x = x;`, dass der Instanzvariablen `x` der Wert zugewiesen wird, der als Parameter der Methode übergeben wird.

Listing 3.27
Die Bezeichner von lokalen Variablen und Instanzvariablen sind identisch.

```
01 public class This1 {
02   int x = 0;
03   String y = "Vorbelegung";
04   void mMethode(int x, String y) {
05     this.x = x;
06     this.y = y;
07   }
08   void printwas() {
09     System.out.println("Wert der Instanzvariable y: " + y);
10     System.out.println("Wert der Instanzvariable x: " + x);
11   }
12   public static void main(String args[]) {
13     This1 ob1 = new This1();
14     ob1.printwas();
15     ob1.mMethode(666666, "Ihre TAN: ");
16     ob1.printwas();
17   }
18 }
```

In der `main()`-Methode wird zuerst ein Objekt der Klasse `This1` erstellt.

Achtung

Beachten Sie, dass wir hier ein Objekt der Klasse erstellen, in der wir uns gerade befinden! Ich habe schon mehrfach festgestellt, dass diese Logik Einsteigern Probleme macht. Es suggeriert ein wenig einen Münchhausen-Effekt. Aber das ist problemlos möglich und üblich.

Über das Objekt wird die Methode `printwas()` aufgerufen. Diese gibt die Werte der beiden Instanzvariablen `x` und `y` aus. Diese haben beim ersten Aufruf noch ihre Defaultwerte. In der nachfolgend aufgerufenen Methode `mMethode(int x, String y)` setzen wir den Wert der Instanzvariablen `x` in der Klasse auf *123456* und die Instanzvariable *y* bekommt den Wert *»Ihre TAN: »*. Diese neuen Werte geben wir dann über den erneuten Aufruf der Methode `printwas()` aus.

```
Problems  @ Javadoc  Declaration  Console
<terminated> This1 [Java Application] C:\Program Files\Ja
Wert der Instanzvariable y: Vorbelegung
Wert der Instanzvariable x: 0
Wert der Instanzvariable y: Ihre TAN:
Wert der Instanzvariable x: 666666
```

Abbildung 3.14
Zugriff auf Instanzvariablen über this

Erstellen wir noch ein Beispiel, das `this` verwendet. Hier nutzen wir `this`, um aus einem Konstruktor auf das erst zur Laufzeit erzeugte Objekt zugreifen zu können. In »diesem« Objekt initialisieren wir eine Instanzvariable.

Listing 3.28 Verwendung von this innerhalb einer Konstruktormethode

```
01 public class This2 {
02   int x;
03   static int y;
04   This2() {
05     y++;
06     this.x = y;
07   }
08   public static void main(String args[]) {
09     System.out.println("Id von dem Objekt: " + new This2().x);
10     System.out.println("Id von dem Objekt: " + new This2().x);
11     System.out.println("Id von dem Objekt: " + new This2().x);
12   }
13 }
```

In dem Beispiel verwenden wir eine Instanzvariable `x` und eine Klassenvariable `y`. Der Default-Konstruktor der Klasse `This2` wird in den Zeilen 4 bis 7 redefiniert. In der Zeile 5 wird die Klassenvariable `y` um den Wert *1* erhöht. Das bedeutet nicht mehr oder weniger, als dass wir uns die Anzahl der erzeugten Instanzen aus der Klasse merken. Und diesen Wert verwenden wir, um jeder Instanz einen spezifischen Wert als ID zuzuweisen. Dazu verwenden wir die Instanzvariable `x`, der wir in Zeile 6 über `this.x` den aktuellen Wert der Klassenvariable `y` zuweisen. Da das Objekt im Konstruktor klar ist, könnte auf `this` verzichtet werden. Aber so ist die Notation eindeutiger.

In der `main()`-Methode werden in den Zeilen eine Reihe von Objekten des Typs `This2` anonym erzeugt und dabei wird unmittelbar der Wert der ID des Objekts ausgegeben. Da jede Erzeugung eines Objekts dieses Typs die Klassenvariable erhöht, hat jede Instanz damit eine eindeutige ID.

Tipp

Dieses Verfahren zum Speichern der Anzahl bereits erzeugter Objekte hat eine allgemeine Bedeutung. Sie können damit etwa sicherstellen, dass nur eine bestimmte Anzahl von Instanzen aus einer Klasse erzeugt werden kann.

Info

Beachten Sie, dass wir uns nicht darum kümmern, wie lange die Objekte existieren (dazu finden Sie im nächsten Abschnitt Ausführungen). Es geht nur um den Zugriff auf die Instanz aus dem Konstruktor heraus via `this`.

Destruktoren und der Garbage Collector für die Speicherbereinigung

Nun sollte ein Programm nicht unbegrenzt Speicherplatz belegen. Der Speicher muss also von einem Programm so verwaltet werden, dass jeder Speicherbereich, der nicht mehr notwendig ist, wieder freigegeben wird. In Techniken wie C/C++ muss sich ein Programmierer manuell darum kümmern. Wird das in so einer Technik vergessen, wird der Speicher nicht freigegeben. Das Programm kann in einen ineffektiven oder gar instabilen Zustand laufen und sogar den Rechner mit ins Grab ziehen. In Java kann das nie passieren.

Das Gegenstück zu Konstruktoren sind in der OOP **Destruktoren**. Diese vernichten ein nicht mehr benötigtes Objekt, führen Aufräumarbeiten aus und geben Speicher frei, der von einem Objekt belegt wurde. Im Fall von Java

werden sie automatisch aufgerufen. Destruktoren sind damit ein weiteres explizites Mittel des Java-Speichermanagements. Oder genauer – Java besitzt gar keine expliziten Destruktoren, wenn man die selten verwendete Methode `finalize()` außer Acht lässt. Stattdessen gibt die Java-Laufzeitumgebung Speicherplatz immer automatisch mit einer Papierkorbfunktionalität frei. Diese wird Garbage Collection oder Garbage Collector genannt. Wenn es keine (starken)[18] Referenzen mehr auf ein Objekt gibt, werden diese beseitigt. Diese Funktionalität läuft **bei jedem Java-Programm** automatisch als Hintergrundprozess mit niedriger Priorität.

Info

Die explizite Speicherfreigabe ist in Java selten notwendig. Allerdings kann die Speicherbereinigung durch den Garbage Collector etwa mit der Methode `System.gc()` direkt manuell aufgerufen werden. Aber das bedeutet nicht, dass sie unmittelbar ausgeführt wird. Wie bei allen Multithreading-Prozessen wird der Zeitpunkt davon abhängen, ob der Prozessor gerade Zeit dafür hat. Falls höher priorisierte Prozesse CPU-Zeit brauchen, wird das dann auf keinen Fall unmittelbar erfolgen und keinerlei Vorteile gegenüber dem automatischen Speicherbereinigungsprozess bringen, mit hoher Wahrscheinlichkeit stattdessen sogar Nachteile.

3.2.8 Fremde Klassen verwenden

Bisher haben wir weitgehend selbst definierte Klassen verwendet. Das wird in der Praxis nicht so bleiben. Objektorientierung bezieht viel von seiner Leistungsfähigkeit daraus, dass Sie Klassen verwenden, die andere Programmierer bereitstellen. In Java haben Sie explizit eine umfangreiche Sammlung von Klassen mit vorgefertigten Funktionalitäten, die Sie ausführlich nutzen (das Java-Standard-API mit seinen Tausenden von Klassen). Wir haben schon darauf zurückgegriffen – selbst wenn das nicht explizit deutlich gemacht wurde.

Der Einstiegspunkt in die Suche nach Klassen

Der Einstiegspunkt für die Suche nach den entsprechenden Klassen des Standard-API durch das Java-System ist das Verzeichnis, das bei der Installation der JRE erstellt und entsprechend als Standardverzeichnis von Java in die Suchstrukturen des Java-Systems (den besagten Klassenpfad) eingetragen wurde. Damit werden die Klassen, die zum Standard-API gehören, (weitgehend) direkt gefunden.

Ein Beispiel, um eine Standardklasse zu verwenden

Betrachten Sie das nachfolgende Beispiel, in dem eine Klasse `Date` aus dem Java-Standard-API verwendet wird.

Listing 3.29 Verwendung der Klasse Date

```
01 public class Datumsausgaben {
02   public static void main(String[] args) {
03     java.util.Date d = new java.util.Date();
```

18 Es gibt auch sogenannte schwache Referenzen, die eine Beseitigung nicht behindern – aber das führt im Moment zu weit.

Listing 3.29 (Forts.) Verwendung der Klasse Date

```
04      System.out.println("Tag: " + d.getDay());
05      System.out.println("Monat: " + d.getMonth());
06      System.out.println("Jahr: " + d.getYear());
07      System.out.println("Stunden: " + d.getHours());
08      System.out.println("Minuten: " + d.getMinutes());
09      System.out.println("Sekunden: " + d.getSeconds());
10    }
11 }
```

Abbildung 3.15 Die Ausgabe des Programms

```
Tag: 5
Monat: 8
Jahr: 111
Stunden: 18
Minuten: 54
Sekunden: 25
```

In dem Beispiel wird über eine vom Java-API bereitgestellte Klasse `Date` das Systemdatum des Rechners abgegriffen. Das erzeugte Datumsobjekt stellt diverse Methoden bereit, die in dem Beispiel über das Objekt `d` genutzt werden.

Info

Beachten Sie, dass die zurückgegebenen Werte etwas von der deutschen Datumsnotation abweichen und die Methoden selbst als deprecated (veraltet) angezeigt werden, was aber hier keine Rolle spielt.

Und selbst die einfache Ausgabe in der Konsole nutzt eine Standardklasse des Java-API – `System`.

3.3 Pakete und die import-Anweisung

Java-Klassen werden allgemein in sogenannten **Paketen** organisiert. Pakete sind in Java – vereinfacht – die objektorientierte Abbildung von Verzeichnissen, in denen Klassen[19] bzw. deren Dateien einsortiert werden können. Und die virtuelle Abbildung der Verzeichnisstrukturen geht oft mit einer physikalischen Einsortierung der Dateien in entsprechende Verzeichnisse einher[20]. Pakete (engl. Packages) sind in Java also Gruppierungen von Klassen über eine baumartige Struktur. Sie sind das Java-Analogon zu Bibliotheken vieler anderer Computersprachen. Die Verzeichnisstrukturen haben im Java-Quelltext eine Darstellung über Bezeichner, die im Normalfall genau den physikalischen Verzeichnisnamen entsprechen. Der Zugriff auf Verzeichnisse und Unterverzeichnisse erfolgt gemäß der allgemeinen Regeln bei Objekten über die Punktnotation.

19 Und deren Spezialformen wie Schnittstellen oder Enums.

20 Obwohl das nicht zwingend ist. Man kann die Klassen auch in Datenbanken ablegen und die Paketstruktur über Datenbankstrukturen umsetzen. Das wurde auch schon durchgeführt, hat sich aber im Allgemeinen als nicht sonderlich performant erwiesen.

Betrachten wir etwa die Angabe `java.util`. Dabei steht der Bezeichner `java` für ein Verzeichnis gleichen Namens und `util` spezifiziert darin ein Unterverzeichnis. Also steht `java.util.Date` für die Klasse `Date` (diese wird über die Datei *Date.class* zur Verfügung gestellt) im Verzeichnis *util*, das sich wiederum in einem Verzeichnis *java* befindet. Wenn Sie sich die bisherigen Beispiele ansehen, erkennen Sie dort mehrfach bereits den Zugriff per Punktnotation auf Paketstrukturen und darin enthaltene Klassen.

Achtung

Die große Ähnlichkeit zum Dateisystem bzw. der meist sogar unmittelbare Bezug zwischen den Dateien und Verzeichnissen auf der einen Seite und den Klassen und Paketen auf der anderen Seite macht offensichtlich, dass sich in einem Paket nur immer genau eine Klasse eines Namens befinden kann. Wenn es in einem Projekt zwei verschiedene Klassen mit dem gleichen Namen geben soll, müssen diese zwingend in verschiedene Pakete einsortiert sein.

3.3.1 Die Zuordnung einer Klasse zu einem Paket und das Default-Paket

Jede Klasse in Java ist Bestandteil eines Pakets. Wenn Sie keine Angaben zu einem Paket im Quelltext angeben, werden die Klassen in der Datei dem **Default-Paket** zugeordnet, das normalerweise dem aktuellen Arbeitsverzeichnis Ihres Projekts entspricht. Dies nennt man das **anonyme Paket**. Die Klassen darin werden immer nur direkt über den Bezeichner angesprochen. Klassen in anderen Paketen hingegen werden vollqualifiziert angesprochen. Der vollqualifizierte Name einer Klasse besteht in Java immer aus dem Namen des Pakets, gefolgt von einem Punkt, eventuellen Unterpaketen, die wieder durch Punkte getrennt werden, bis hin zum eigentlichen Klassennamen. Um nun eine Klasse außerhalb des Default-Pakets verwenden zu können, muss einem Java-Tool wie dem Compiler gesagt werden, in welchem Paket (Verzeichnis) er sie suchen soll. Etwa so:

```
java.awt.Frame;
```

Das bedeutet, das Java-Tool sucht in dem Paket `java` nach dem Unterpaket `awt` (physisch in der Regel wie gesagt ein Unterverzeichnis des Verzeichnisses *java*) und dort die Klasse *Frame* (die physische Datei *Frame.class* im Verzeichnis *awt*). Das ist genau die Logik, die wir gerade beschrieben haben.

Achtung

Die Verwendung des Default-Pakets wird für das Speichern von eigenen Klassen allgemein nicht empfohlen. Das sagt Ihnen sogar Eclipse[21] oder NetBeans, wenn Sie eine Klasse erstellen und diesem Paket zuordnen wollen. Verboten ist es aber nicht und warum das nicht empfohlen wird, behandeln wir gleich.

21 The use of the default package is discouraged.

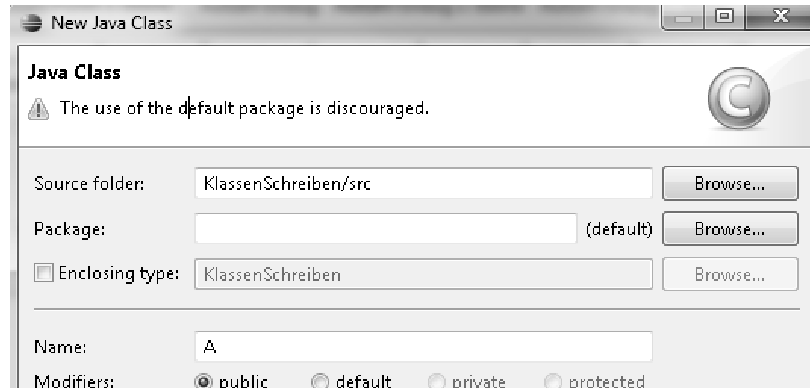


Abbildung 3.16 Das Default-Package ist kein guter Ort für Ihre Klassen.

Grundsätzliches zur Zuordnung zu einem Paket

Jede Java-Datei gehört ohne eine explizite Zuweisung im Quellcode automatisch zum Default-Paket. Die Zuordnung zu einem spezifischen Paket erfolgt mit einer Anweisung `package`. Wenn die `package`-Anweisung in einer Java-Datei fehlt, werden alle[22] Klassen in der Datei also dem anonymen Paket zugeordnet.

Eine Java-Datei wird einem anderen neuen Paket beziehungsweise einem bestehenden Paket zugeordnet, wenn Sie ganz am Anfang der Datei als erste gültige Anweisung (auch vor der ersten Klassendefinition, aber abgesehen von Kommentaren) das Schlüsselwort `package` und den Namen des Pakets, gefolgt von einem Semikolon, setzen.

Achtung

Es kann nur eine solche Anweisung in einer Java-Datei notiert werden.

Wenn der Compiler respektive eine IDE wie Eclipse eine entsprechende Verzeichnisstruktur nicht vorfindet, werden die Verzeichnisse angelegt. Das Verfahren setzt sich mit eventuellen Unterverzeichnissen fort. Anschließend können Sie wie gewohnt Ihre Klassen definieren.

Mit einer IDE Pakete erstellen und Java-Klassen zuordnen

Wie gerade erwähnt, kann man aus dem Java-Code heraus mit der Anweisung `package` die Klassen in einer Datei einem Paket hinzufügen. Bequemer geht das aber mit einer IDE wie NetBeans oder Eclipse. Spielen wir das mit Eclipse durch.

Erzeugen Sie zunächst ein neues Java-Projekt mit Namen *PaketeErstellen*. Nun klicken Sie im Package Explorer mit der rechten Maustaste auf das Projekt oder Sie wählen FILE. In beiden Fällen wählen Sie dann NEW -> PACKAGE.

22 Sie erinnern sich? Obwohl man es nicht machen sollte, kann man in einer *.java*-Datei beliebig viele Klassendeklarationen notieren, solange nur eine davon öffentlich ist.

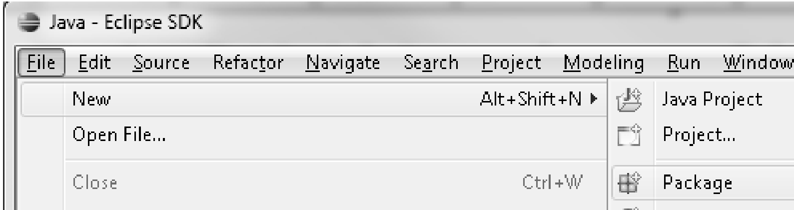


Abbildung 3.17
Ein neues Paket anlegen

Im folgenden Dialog geben Sie für den Namen *meinpaket* an.

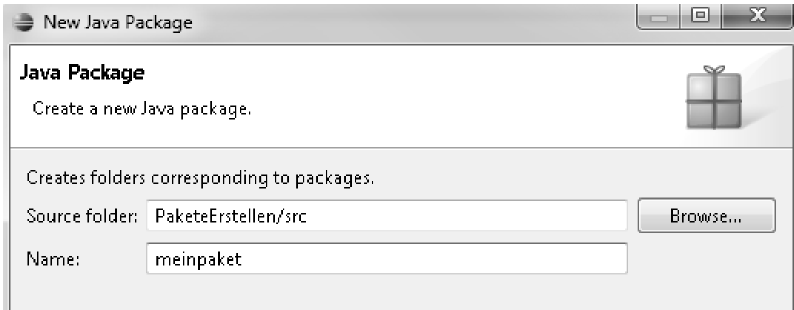


Abbildung 3.18
Der Name des Pakets wird angegeben.

Wenn Sie den Dialog mit Finish beenden, wird Eclipse ein Paket erzeugen. Im Package Explorer sehen Sie nachfolgend ein entsprechendes Symbol für ein (noch leeres) Paket unterhalb von *src*.

Abbildung 3.19
Das leere Paket wurde angelegt.

Jetzt lohnt sich ein Blick auf die physikalische Verzeichnisstruktur, wie sie etwa die Navigator-View (oder der Dateimanager des Betriebssystems) bietet.

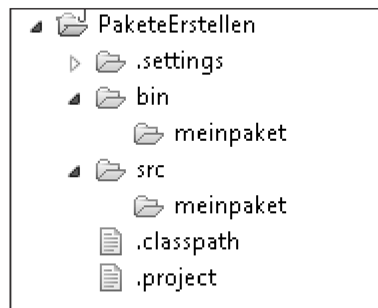
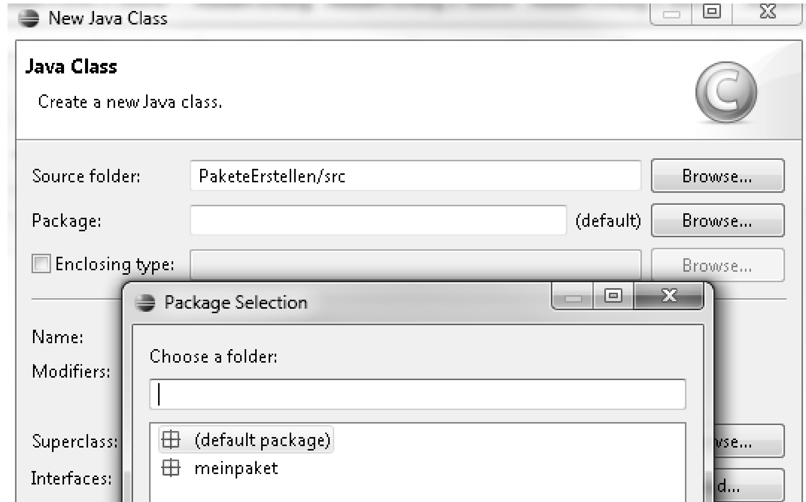


Abbildung 3.20
Die neu angelegten Unterverzeichnisse

Sowohl unter *src* als auch *bin* wurde ein Unterverzeichnis mit dem Namen des Pakets angelegt. Erstellen wir nun eine neue Klasse und fügen diese dem Paket hinzu. Wenn im Dialogfenster zum Anlegen einer Klasse im Eingabefeld ***Package*** kein Paket angegeben ist, können Sie auf `Browse...` klicken und Sie erhalten einen weiteren Dialog, in dem Sie das neu angelegte Paket auswählen können.

Abbildung 3.21
Hier können Sie alle im Projekt verfügbaren Pakete auswählen.



Geben Sie als Namen der Klasse `MeineKlasse` an und achten Sie darauf, dass der richtige Zielordner und das gewünschte Paket selektiert sind.

Als Resultat sollten Sie den folgenden Code bekommen:

Listing 3.30
Die Klasse gehört zum Paket meinpaket.

```
01 package meinpaket;
02   public class MeineKlasse {
03 }
```

Beachten Sie die `package`-Anweisung in Zeile 1, die Eclipse automatisch angelegt hat. Werfen wir wieder einen Blick in die Navigator-View, um die physikalische Verzeichnisebene zu sehen.

Sie werden sehen, dass in den jeweiligen Paketverzeichnissen die entsprechenden *.java*- bzw. *.class*-Dateien zu finden sind. Diese wurden von Eclipse automatisch richtig angelegt und einsortiert.

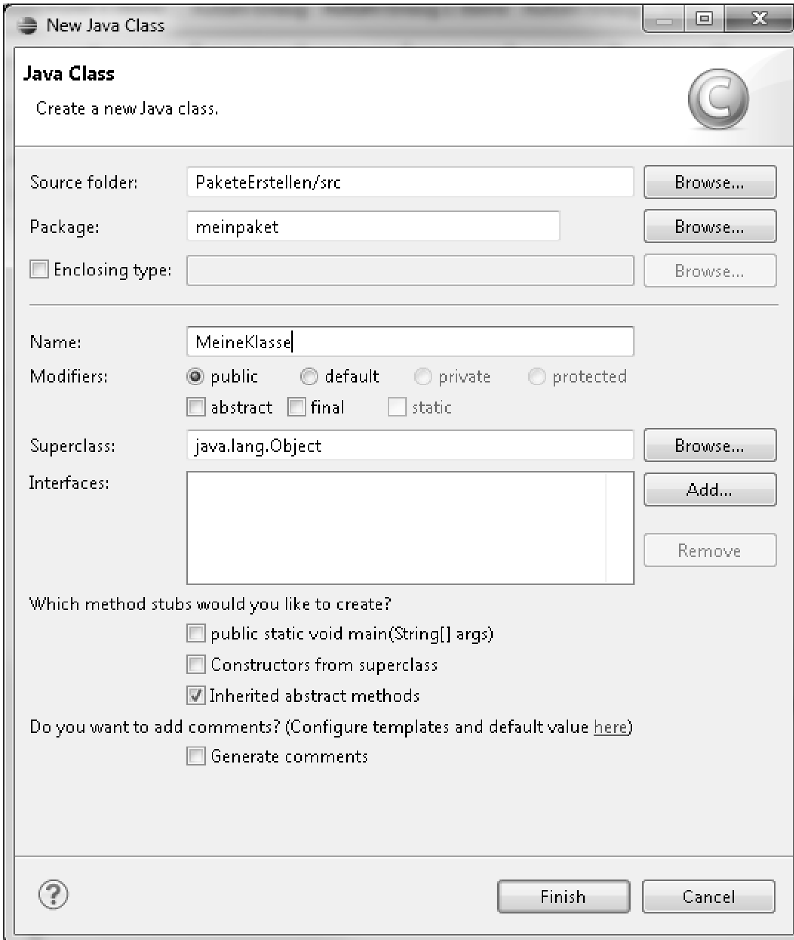
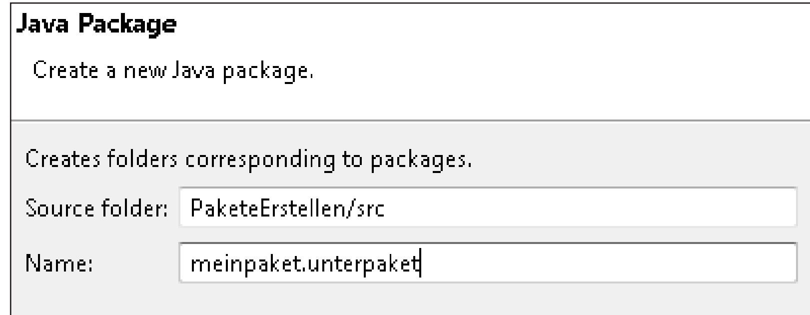


Abbildung 3.22
Die Klasse wird dem Paket hinzugefügt.

Abbildung 3.23
In den Paketverzeichnissen finden Sie die jeweiligen .java- bzw. .class-Dateien.

Um die Sache noch deutlicher zu machen, erzeugen wir ein weiteres Paket. Geben Sie dafür `meinpaket.unterpaket` an. Beachten Sie die Punktnotation.

Abbildung 3.24 Ein weiteres Paket wird angelegt.

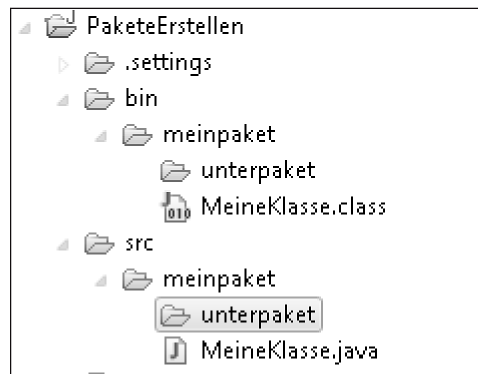


Im Package Explorer sehen Sie nach der Erstellung ein weiteres leeres Paket.

Abbildung 3.25 Im Projekt gibt es ein weiteres leeres Paket.

Werfen wir wieder einen Blick in die Navigator-View. Sie werden sehen, dass sich innerhalb des Verzeichnisses *meinpaket* ein leeres Unterverzeichnis *unterpaket* befindet.

Abbildung 3.26 Die neue Verzeichnisstruktur



Nun verschieben Sie im Package Explorer mit der Maus die Datei *MeineKlasse.java* in das Paket *meinpaket.unterpaket*. Eclipse wird Ihnen einen Dialog anbieten, ob die IDE bestehende Referenzen auf die Klasse im verwalteten

Bereich der Workbench bei der Verschiebung aktualisieren soll. Das ist ein nützliches Feature, um Ihre Projekte bei solchen Verschiebeaktionen konsistent zu halten. In der Regel werden Sie das bestätigen und das wollen wir jetzt tun, obwohl es nicht notwendig ist[23].

Achtung

Wenn Sie Dateien mit dem Windows Explorer oder einem anderen Dateimanagementprogramm des Betriebssystems verschieben, werden die Referenzen nicht aktualisiert und Ihre Projekte werden unter Umständen inkonsistent. Nutzen Sie deshalb unbedingt die Werkzeuge von Eclipse selbst oder NetBeans, wenn Sie damit arbeiten.

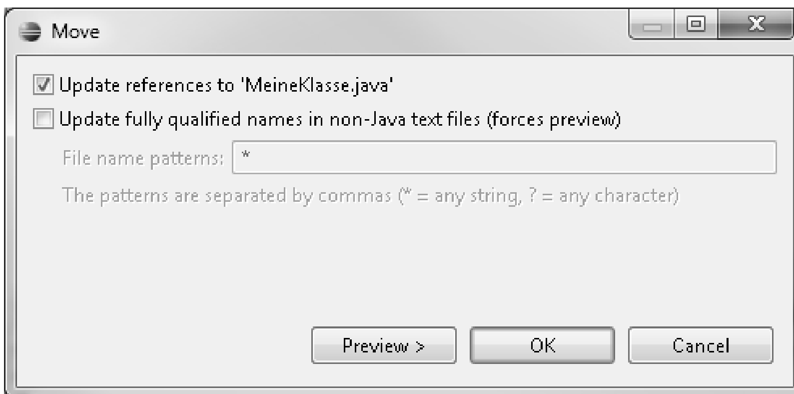


Abbildung 3.27 Aktualisieren von Referenzen auf die zu verschiebende Klasse

Wichtig ist das Resultat, das nach der Verschiebung herauskommt.

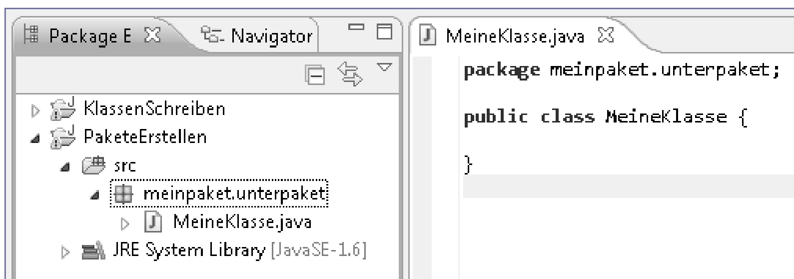


Abbildung 3.28 Nach dem Verschieben

Sie sehen im Package Explorer, dass das Paket `meinpaket.unterpaket` nun gefüllt ist, und vor allen Dingen wurde im Quelltext automatisch die package-Anweisung am Beginn der *.java*-Datei aktualisiert. Sie spiegelt exakt wider, wo sich die Datei befindet. In der Navigator-View sehen Sie, dass die *.class*-Datei automatisch mit verschoben wurde.

23 Es gibt ja keine Referenzen auf diese Klasse bisher – aber das schadet nichts.

Abbildung 3.29 Auch die .class-Datei wurde in das passende Unterverzeichnis verschoben.

Sie sollten anhand der langen und sehr ausführlichen Ausführungen erkannt haben, dass die `package`-Anweisung immer direkt mit dem Paket und damit dem Verzeichnis korrespondiert, in dem die jeweilige Datei zu finden ist.

Die Klasse in dem Paket verwenden

Lassen Sie uns nun unsere Beispielklasse mit etwas Substanz füllen und wie folgt erweitern:

Listing 3.31 Die Klasse hat nun einen sinnvollen Inhalt.

```
01 package meinpaket.unterpaket;
02 public class MeineKlasse {
03  public static int a = 42;
04 }
```

Das ist sicher nicht viel Funktionalität (nur eine initialisierte Klasseneigenschaft), aber darum soll es hier nicht gehen[24]. Wir wollen diese Klasse nun verwenden. Erzeugen Sie im Default-Paket des Projekts eine neue Klasse `NutzeKlasse` mit einer `main()`-Methode.

Die Klasse soll folgenden Inhalt haben:

Listing 3.32 Die Klasse in einem Paket verwenden

```
01 public class NutzeKlasse {
02   public static void main(String[] args) {
03     System.out.println(meinpaket.unterpaket.MeineKlasse.a);
04   }
05 }
```

In Zeile 3 können Sie erkennen, wie Sie die Klasse `MeineKlasse` und die darin enthaltene Klasseneigenschaft a verwenden können – mit dem vollqualifizierten Pfad entsprechend der Paketstruktur, ausgehend von dem Punkt, an dem Sie stehen – der Wurzel des Projekts. Das erscheint vollkommen analog einer Pfadangabe im Verzeichnissystem. Oder ist da ein Unterschied? Täuscht der Schein?

24 Zu viel Inhalt würde eventuell neue Fragen aufwerfen und von unserem Fokus – der Verwendung dieser Klasse – ablenken.

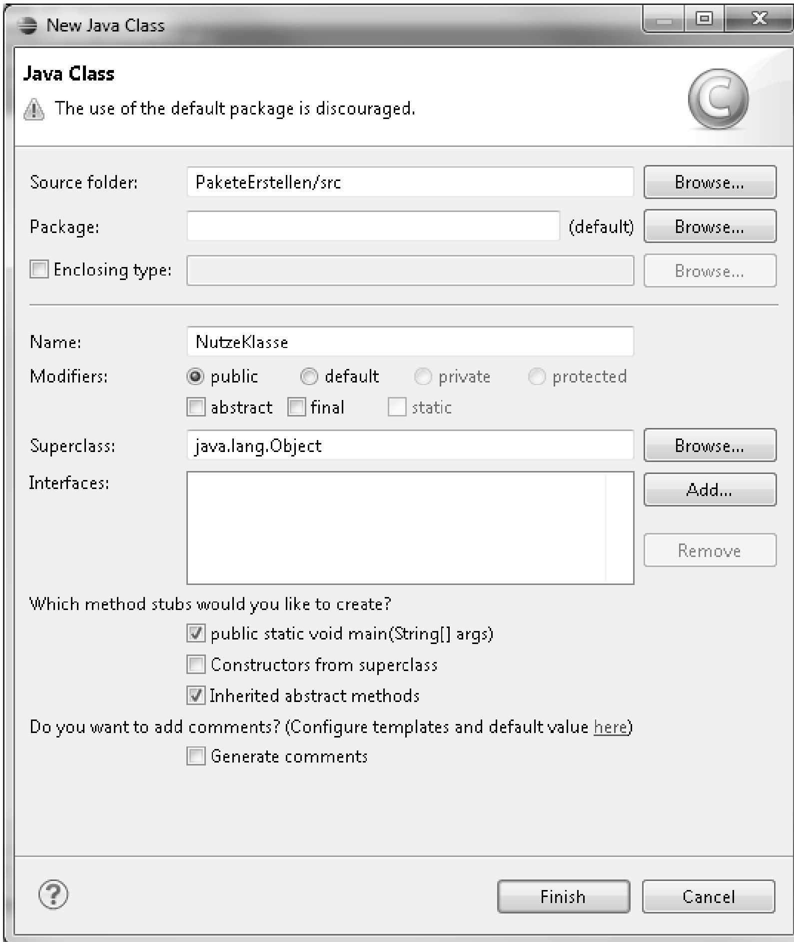


Abbildung 3.30
Anlegen einer Programmklasse im Projekt

Verschieben wir die Klasse `NutzeKlasse` einmal in das Paket `meinpaket`. Der Quelltext muss entsprechend die `package`-Anweisung `package meinpaket;` am Beginn enthalten.

Eclipse erweist sich Unter Umständen etwas störrisch, wenn Sie im Package Explorer eine Klasse in ein übergeordnetes Paket verschieben wollen und dieses leer ist. Verschieben Sie dann die Java-Datei in der Navigator-View. Das führt zwar zu einem Fehler in Eclipse, aber die Datei ist in dem entsprechenden Paket. Sie müssen dann nur noch manuell die `package`-Anweisung ergänzen. Das ist keine besondere Schwäche von Eclipse – normalerweise macht man solche Spielereien nicht, die wir hier rein zu Lernzwecken durchführen.

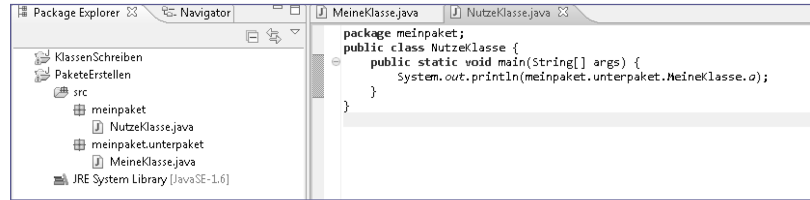


Abbildung 3.31 Das Programm wurde jetzt in das Paket meinpaket verschoben.

Doch was wird mit der vollqualifizierten Angabe in der Klasse `MeineKlasse` aus Zeile 3 passieren. Bleibt die gleich? Oder muss sie angepasst werden? Etwa so:

Listing 3.33 Eine mögliche vollqualifizierte Angabe

```
unterpaket.MeineKlasse.a
```

Wir stehen ja mit dem Programm schon im Verzeichnis *meinpaket* bzw. in dem entsprechenden Paket. Und dann kann man doch das Paket `meinpaket` weglassen – oder etwa nicht? Tatsächlich können Sie es **nicht** weglassen. Das **Projektverzeichnis** ist immer die Wurzel der Paketangaben.

3.3.2 Namenskonventionen und Standardpakete

Es gibt nun neben den üblichen Regeln für Bezeichner unter Java einige spezielle **Namenskonventionen** für Pakete. Die Standardklassen von Java[25] sind in eine Paketstruktur eingeteilt, die an das Domain-System von Internetnamen angelehnt ist. Denn dieses Namenssystem ist eindeutig. Grundsätzlich sollte jeder Anbieter von eigenen Java-Paketen diese nach seinem DNS-Namen strukturieren, nur in umgekehrter Reihenfolge, wie normalerweise die Domains verschachtelt werden. Wenn etwa eine Firma RJS mit der Internetadresse *www.rjs.de* Java-Pakete bereitstellt, sollte die Basispaketstruktur `de.rjs` lauten. Dies würde die Wurzel eines eigenen Namensraums darstellen. Eventuelle Unterpakete sollten darunter angeordnet werden. Selbst offiziell in dem Standard-API von Java enthaltene zusätzliche Pakete (etwa CORBA-Pakete – z.B. `org.omg.CORBA`) halten diese Konventionen ein. Einzige (offizielle) Ausnahmen sind die Klassen, die im Standardumfang einer Java-Installation enthalten sind. Diese beginnen mit `java` oder `javax`.

Die Einhaltung dieser Vorschläge zur Benennung von Paketstrukturen ist nicht zwingend. Sie können beliebige Namen für Ihre Pakete nehmen. Sie könnten sogar hingehen und ihre Java-Klassen in »offizielle« Pakete wie `sun.com` oder `microsoft.de` einsortieren. Es gibt in dem Sinn keine »Eigentümer« dieser Paketnamen bzw. Paketstrukturen, wie es mit den DNS-Namen selbst der Fall ist. Nur kann es bei Nichtbeachtung der Konventionen dazu führen, dass es in größeren Projekten oder bei der Zusammenarbeit mit anderen Projekten zu Namenskonflikten kommt. Und diese lassen sich eben leicht vermeiden. Wenn sich jeder Programmierer bzw. jede Firma oder Organisation an die Konventionen hält und die Pakete

25 Das API

entsprechend des eigenen DNS-Namens[26] benennt, werden alle Klassen in eigenen und weltweit eindeutigen Namensräumen verwaltet und diese Klassen können dann problemlos gemeinsam verwendet werden. Denn da sich in einem Paket nur immer genau eine Klasse eines Namens befinden kann, kann man durch getrennte und eindeutige Namensräume gewährleisten, dass es keine Konflikte gibt, wenn ein Klassenbezeichner in einem Projekt mehrfach vorkommt – solange die Klassen sich in unterschiedlichen Paketen befinden.

Bei der Auswahl der Namen für Pakete gelten ansonsten nur wieder die üblichen Regeln für Token. Grundsätzlich werden Paketbezeichner kleingeschrieben beziehungsweise nur die Anfangsbuchstaben ab dem zweiten Begriff (Camelnotaion), aber das ist nur eine Konvention. Allerdings müssen Sie beachten, dass das dann bei den Verzeichnisnamen so einzuhalten ist.

3.3.3 Die Suche nach Paketen

Nun wissen wir, wie Klassen in einem Projekt zu finden sind. Entweder befinden sie sich im gleichen Paket (Verzeichnis). Dann können Sie einfach direkt den Klassennamen angeben. Oder sie befinden sich in Paketen. Dann geben Sie mit Punktnotation den Namen des Pakets bzw. der Pakete und dann die Klasse an, etwa `java.awt.Frame`. Aber wo ist beispielsweise das Java-Standardverzeichnis *java* selbst zu finden? Wenn Sie aus einer Klasse heraus auf andere Klassen – sowohl Standardklassen des Java-API als auch selbst geschriebene Klassen – zugreifen, wird das aktuell verwendete JDK-Programm (Interpreter, Compiler, Debugger etc.) zuerst in dem aktuellen Verzeichnis danach suchen. Bei einem Element, das mit vorangestellten Verzeichnissen (Paketen) angegeben wird, wird dort vollkommen analog zuerst die Verzeichnisstruktur gesucht. Dabei werden sich die *.class*-Dateien in der Regel in *.jar*-Dateien befinden, die standardmäßig durchsucht werden.

Der Klassenpfad

Finden sich die gesuchten Klassen respektive die Verzeichnisstruktur nicht innerhalb des aktuellen Verzeichnisses, wird die Suche in der Verzeichnisstruktur des Java-API selbst fortgesetzt, die bei der Installation von Java angelegt und im Betriebssystem als **Klassenpfad** (Classpath) für Standardklassen registriert ist. Zusätzlich können Sie einen individuell über die Option *–classpath* angegebenen Pfad spezifizieren, der durchsucht werden soll.

26 Dass ein Java-Programmierer oder gar eine Firma oder Organisation keinen eigenen DNS-Namen besitzt, wird grundsätzlich ausgeschlossen und dürfte bei einer professionellen Arbeit mit Java auch nicht möglich sein. Sollte das doch der Fall sein, denken Sie sich einen fiktiven Namen aus, für den es noch keinen realen DNS-Namen gibt. Nur können Sie dann nie sicher sein, dass nicht irgendwann jemand diesen DNS-Namen doch verwendet und für eigene Java-Pakete einsetzt.

Tipp

In der Konsole können Sie sich bei Windows mit dem Befehl *set* die Umgebungsvariablen Ihres Betriebssystems anzeigen lassen. Früher wurde hier für Java eine globale Umgebungsvariable *Classpath* gesetzt, aber in modernen Versionen von Java macht man das unter Windows nicht mehr. Der Pfad wird im Hintergrund verwaltet (etwa in der Registry). In Eclipse können Sie den Klassenpfad für ein Projekt etwa über PROJECT -> PROPERTIES -> JAVA BUILD PATH einstellen. Wenn Sie das Dialogfenster betrachten, sehen Sie, dass Ihnen das komplette Java-Standard-API angezeigt wird und Sie zusätzliche Pakete hinzufügen können, die ebenfalls bei der Suche berücksichtigt werden sollen.

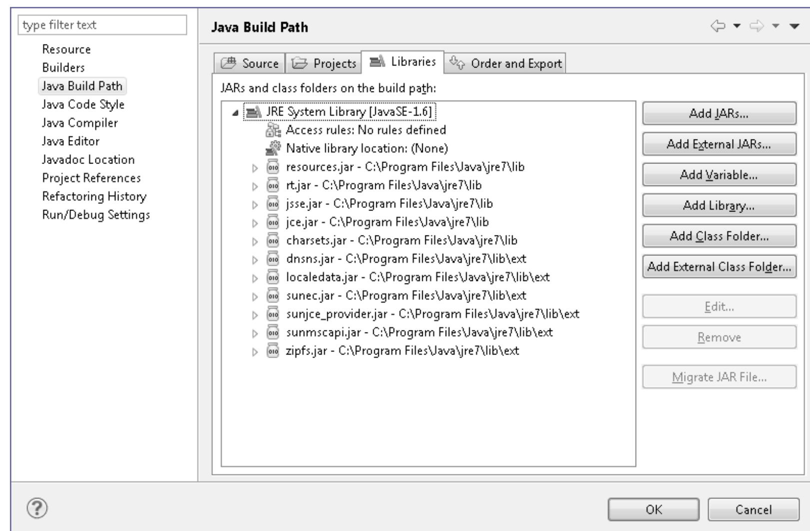


Abbildung 3.32 Der Klassenpfad in Eclipse

Die Durchforstung des Klassenpfads ist dann eine rein physische Suche nach *.class*-Dateien, die wie gesagt auch in gepackten Verzeichnissen (*.jar*-Dateien) durchgeführt wird. Genaugenommen wird das gesamte Java-Standard-API in *.jar*-Dateien bereitgestellt.

Wenn nun aber ein Verzeichnis zum Durchsuchen nach einer Klasse feststeht, werden dort eventuell vorhandene Unterverzeichnisse nicht (!) nach benötigten Klassen durchsucht. Deshalb muss bei der Verwendung von Paketen der vollqualifizierte Name einer Klasse angegeben werden.

Tipp

Expandieren Sie einmal im Package Explorer mit einem Doppelklick den Zweig *JRE System Library*. Dort finden Sie eine Datei *rt.jar* und die sollten Sie ebenso expandieren. Sie sehen danach eine große Anzahl von Standardpaketen. Expandieren Sie zum Beispiel weiter den Zweig *java.lang*. Dann sehen Sie die *.class*-Dateien, die den Kern von Java ausmachen.

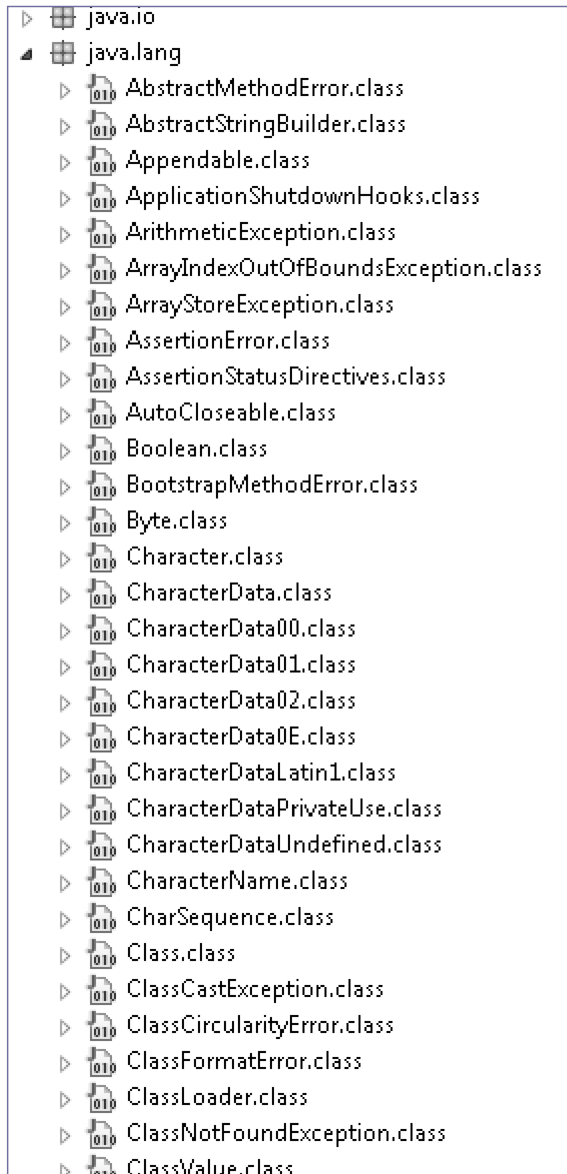


Abbildung 3.33
Das sind die Klassen in java.lang.

3.3.4 Importieren von Klassen

Wenn Sie eine Klasse in einem bestimmten Paket benötigen, können Sie die Klasse vollqualifiziert notieren, wie wir gesehen haben. Gegebenenfalls erweitern Sie direkt die Notation um einen gewünschten Methoden- beziehungsweise Variablennamen. Mit einer solchen vollqualifizierten Notation ist jedoch die Lesbarkeit des Quelltexts nicht gerade gut und vor allem der

Tippaufwand für sich wiederholende Quelltextpassagen sehr hoch. Dies macht eine einfachere und schnellere Technik notwendig.

Die import-Anweisung

Um eine Klasse eines fremden Verzeichnisses respektive Pakets direkt nutzen zu können, kann man die Klasse oder das ganze Paket vorher importieren. Das geschieht durch eine `import`-Zeile, die vor der Definition irgendeiner Klasse, aber nach einer `package`-Anweisung, in der *.java*-Datei stehen muss.

Sie können eine Klasse importieren, indem Sie die gesamte Punktnotation (das Paket, zu dem sie gehört, und die Klasse selbst) am Anfang einer Datei mit dem Schlüsselwort `import` angeben. Danach können Sie die Klasse direkt mit ihrem Namen ansprechen. In unserem letzten Beispiel würde es wie folgt aussehen:

Listing 3.34
Eine import-Anweisung

```
import meinpaket.unterpaket.MeineKlasse;
```

In diesen Fall können Sie später im Quelltext auf die Klasse `MeineKlasse` über ihren Namen zugreifen, ohne den vollqualifizierten Pfad notieren zu müssen, etwa so:

Listing 3.35
Der verkürzte Zugriff auf eine Eigenschaft in der Klasse MeineKlasse

```
MeineKlasse.meineEigenschaft;
```

So würde der vollständige, geänderte Code aussehen:

Listing 3.36
Importieren einer Klasse und der verkürzte Zugriff

```
01 package meinpaket;
02 import meinpaket.unterpaket.MeineKlasse;
03 public class NutzeKlasse {
04   public static void main(String[] args) {
05     System.out.println(MeineKlasse.meineKlasse);
06   }
07 }
```

In Zeile 2 sehen Sie den Import und in Zeile 5 den verkürzten Zugriff.

Import von mehreren Klassen

Es wird in der Regel in einer Quelltextdatei mehrfache `import`-Anweisungen geben. Sie müssen wie gesagt alle hinter der optionalen Anweisung `package` und vor der ersten Klassendefinition stehen. Wenn Sie mehrere Klassen aus einem Paket benötigen, arbeitet man sinnvollerweise mit Platzhaltern. Mittels einer solchen Wildcard – dem Stern `*` – kann ein ganzes Paket auf einmal importiert werden. Das geschieht wieder durch die `import`-Zeile, wobei nur der Klassenname durch den Stern ersetzt wird. Danach können Sie alle Klassen aus dem Paket direkt über ihren Namen ansprechen. Das Sternchen importiert jedoch keine (!) untergeordneten Pakete. Um also alle Klassen einer komplexen Pakethierarchie zu importieren, müssen Sie explizit auf jeder Hierarchieebene eine `import`-Anweisung erstellen.

Der `import`-Befehl kennt im Prinzip drei Varianten:

```
import <package>.<Klasse>;
import <package>.*;
import <package>;
```

Die erste Form wird dann eingesetzt, wenn man gezielt auf eine Klasse zugreifen möchte und nur genau diese Klasse aus einem Paket benötigt. Im Fall 2 besteht der Vorteil, dass alle Klassen aus dem eingebundenen Paket über den verkürzten Namen verfügbar sind. Die dritte Form dient dazu, mit möglichst wenigen Anweisungen ein Paket oder sogar mehrere Pakete zu importieren. Allerdings kommt jetzt wieder zum Tragen, dass keine untergeordneten Pakete importiert werden. Deshalb kann in dem Fall eine Klasse innerhalb des Quelltextes nicht direkt über ihren Namen angesprochen werden. Stattdessen muss vor dem Klassennamen das letzte Element aus dem Paketnamen per Punktnotation gesetzt werden. Dies ist sukzessive fortzusetzen, wenn es mehrere Paketebenen gibt, die nicht im `import`-Befehl angegeben wurden. In diesem Fall muss unter Umständen eine längere Pfadangabe die Klasse referenzieren. Diese Variante ist ziemlich unüblich.

Tipp

Viele bessere Java-Editoren bzw. IDEs unterstützen Anwender übrigens beim Import von Klassen und organisieren Importe teilweise sogar im Hintergrund. Gerade Eclipse und NetBeans bieten Ihnen da sehr viel Bequemlichkeit.

Statisches Importieren

Sie können in Java das Schlüsselwort `static` hinter dem `import` notieren. Das nennt sich dann statisches Importieren. Es erlaubt bei so referenzierten statischen Elementen im Code den Verzicht auf das Voranstellen der Klasse. Wenn Sie etwa in Ihrem Quellcode auf `Math.PI` zugreifen wollen und nur `import java.util.Math` notieren, müssen Sie im Quellcode `Math.PI` notieren. Bei einem statischen Import würden Sie `import static java.util.Math.PI` notieren und dann können Sie im Quellcode einfach `PI` schreiben, was gerade bei mathematischen Formeln eine kompaktere und besser lesbare Schreibweise gestattet.

import versus include

Eine `import`-Anweisung dient nur dazu, Java-Klassen im Quellcode über einen verkürzten Namen innerhalb der aktuellen Bezugsklasse zugänglich zu machen und damit den Code zu vereinfachen. Sie ist – trotz des irreführenden Namens – kein echter Import im Sinn einer `include`-Anweisung in C und bindet Klassen nicht wirklich in den Code ein, da das Importieren von Elementen in Java also kein echter Import in dem Sinn ist, dass das resultierende Programm alle angegebenen Klassen irgendwie verwalten muss, sondern nur eine Pfadangabe. Daher können Sie in eine Java-Klasse beliebig viele Pakete und Klassen importieren. Der resultierende Bytecode wird weder größer noch sonst ineffektiver. Nicht explizit benötigte Klassen werden vom Compiler wegoptimiert. Das nennt man **type import on demand**.

Das Paket java.lang

Es gibt ein Java-Paket aus dem Standard-API, das sich besonders auszeichnet. Sie müssen dieses nie explizit importieren, um die enthaltenen Klassen mit verkürzten Namen anzusprechen. Das ist das Paket `java.lang`. Dieses Paket wird vom Laufzeitsystem immer automatisch importiert und stellt die Basis des gesamten Java-Konzepts dar. Darin finden Sie so wichtige Klassen wie `Object` oder `String`, aber auch Wrapper-Klassen zur Konvertierung und die Klasse `System`[27] sind dort zu finden.

Mehrdeutigkeit von Klassen

Wenn Sie mehrere Klassen und/oder Pakete importieren, kann es bei dem verkürzten Namen zu Mehrdeutigkeiten kommen. Betrachten wir ein neues Projekt *Mehrdeutig* als Beispiel. Erzeugen Sie dort zwei Pakete `paket1` und `paket2` und darin jeweils eine Klasse mit Namen `A`.

Die Klasse `paket1.A` soll folgenden Inhalt haben:

Listing 3.37 Die erste Klasse A

```
01 package paket1;
02 public class A {
03   public static int var1=1;
04 }
```

Die Klasse `paket2.A` soll hingegen folgenden Inhalt haben:

Listing 3.38 Die zweite Klasse A

```
01 package paket1;
02 public class A {
03   public static int var1=100;
04 }
```

Es gibt nur einen kleinen Unterschied in den beiden Klassen, aber der genügt, um zu erkennen, welche Klasse der Compiler bei einer Mehrdeutigkeit eines Klassennamens verwenden würde. Im Default-Package erzeugen Sie die Klasse `NutzeKlassen` mit folgendem Inhalt:

Listing 3.39 Mehrdeutigkeit

```
01 import paket1.A;
02 import paket2.A;
03 public class NutzeKlassen {
04   public static void main(String[] args) {
05     System.out.println(A.var1);
06   }
07 }
```

Die Angabe in Zeile 5 wäre bei dem gleichzeitigen Import der beiden Klassen `A` nicht mehr zuverlässig aufzulösen. Zwar könnte man Regeln festlegen, dass die zuerst oder die zuletzt importierte Klasse Priorität hat, aber das wäre sowohl gefährlich als auch unlogisch, weil es sich ja nicht um echte Importe handelt. Wenn Sie den Code in Eclipse eingeben, werden Sie sehen, dass der Code so abgelehnt wird. Sie müssen auf eine der beiden `import`-Anweisungen verzichten und die nicht importierte Klasse gegebenenfalls vollqualifiziert ansprechen, etwa so:

27 Wir hatten schon oft `System.out.println()` verwendet. Eigentlich wäre `java.lang.System.out.println()` die vollständige Notation.

```
01 import paket1.A;
02 public class NutzeKlassen {
03   public static void main(String[] args) {
04     System.out.println(A.var1);
05     System.out.println(paket2.A.var1);
06   }
07 }
```

Listing 3.40
Das ist eindeutig.

3.3.5 Namensräume bzw. Gültigkeitsbereiche

Sowohl die Einsortierung von Klassen in verschiedene Pakete als auch in der Klasse selbst die Trennung des Bereichs innerhalb einer Methode und dem Bereich außerhalb gehören zu einem allgemeineren Konzept – den Namensräumen. Java stellt damit eine universell über verschiedene EDV-Techniken genutzte Möglichkeit zur Verfügung, mit der Namenskonflikte aufgelöst werden können, wenn es zwei identische Bezeichner gibt, im Fall vom Java im Quelltext, im Fall von einem Dateisystem bei Dateien. Man versteht unter einem **Namensraum** im Grunde nur einen Gültigkeitsbereich, in dem ein bestimmter Bezeichner benutzt werden kann und wo er eindeutig sein muss.

Namensräume sind in Java einer Hierarchie zugeordnet. Es gilt dabei die Regel, dass ein Bezeichner immer einen identischen Bezeichner in einem übergeordneten (umgebenden) Namensraum überdeckt. Außerdem trennt Java wie gesagt die Namensräume von lokalem und nichtlokalem Code. Die Hierarchie der Namensräume gliedert sich wie folgt:

- Ganz außen (aus Sicht der Mengenlehre zu sehen) steht der Namensraum des Pakets, zu dem die Klasse gehört.
- Danach folgt der Namensraum der Klasse. Dieser überdeckt den Namensraum des Pakets.
- Es folgen die Namensräume der einzelnen Methoden. Dabei überdecken die Bezeichner von Methodenparametern die Bezeichner von Elementen der Klasse.
- Innerhalb von Methoden gibt es unter Umständen noch weitere Namensräume in Form von geschachtelten Blöcken. Variablen, die innerhalb eines solchen geschachtelten Blocks deklariert werden, sind außerhalb des Blocks unsichtbar.

Um einen Bezeichner zuzuordnen, werden die Namensräume immer von innen nach außen aufgelöst. Wenn der Compiler einen Bezeichner vorfindet, wird er zuerst im lokalen Namensraum suchen. Sofern er dort nicht fündig wird, sucht er im übergeordneten Namensraum. Das Verfahren setzt sich analog bis zum ersten Treffer (gegebenenfalls bis zur obersten Ebene) fort. Es gelten immer nur die Vereinbarungen des Namensraums, in dem der Treffer erfolgt ist.

Copyright

Daten, Texte, Design und Grafiken dieses eBooks, sowie die eventuell angebotenen eBook-Zusatzdaten sind urheberrechtlich geschützt. Dieses eBook stellen wir lediglich als **persönliche Einzelplatz-Lizenz** zur Verfügung!

Jede andere Verwendung dieses eBooks oder zugehöriger Materialien und Informationen, einschließlich

- der Reproduktion,
- der Weitergabe,
- des Weitervertriebs,
- der Platzierung im Internet, in Intranets, in Extranets,
- der Veränderung,
- des Weiterverkaufs und
- der Veröffentlichung

bedarf der **schriftlichen Genehmigung** des Verlags. Insbesondere ist die Entfernung oder Änderung des vom Verlag vergebenen Passwortschutzes ausdrücklich untersagt!
Bei Fragen zu diesem Thema wenden Sie sich bitte an: info@pearson.de

Zusatzdaten

Möglicherweise liegt dem gedruckten Buch eine CD-ROM mit Zusatzdaten bei. Die Zurverfügungstellung dieser Daten auf unseren Websites ist eine freiwillige Leistung des Verlags. **Der Rechtsweg ist ausgeschlossen.**

Hinweis

Dieses und viele weitere eBooks können Sie rund um die Uhr und legal auf unserer Website herunterladen:

http://ebooks.pearson.de