

Studienskript

GRUNDLAGEN DER OBJEKTORIENTIERTEN PROGRAMMIERUNG MIT

JAVA

IOBP01

iu

INTERNATIONALE
HOCHSCHULE

GRUNDLAGEN DER OBJEKTORIENTIERTEN PROGRAMMIERUNG MIT JAVA

IMPRESSIONUM

Herausgeber:
IU Internationale Hochschule GmbH
IU International University of Applied Sciences
Juri-Gagarin-Ring 152
D-99084 Erfurt

Postanschrift:
Albert-Proeller-Straße 15-19
D-86675 Buchdorf
media@iu.org
www.iu.de

IOBP01
Versionsnr.: 001-2023-1107
N. N.

© 2023 IU Internationale Hochschule GmbH
Dieses Lernskript ist urheberrechtlich geschützt. Alle Rechte vorbehalten.
Dieses Lernskript darf in jeglicher Form ohne vorherige schriftliche Genehmigung der
IU Internationale Hochschule GmbH (im Folgenden „IU“) nicht reproduziert und/oder
unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet wer-
den.
Die Autor:innen/Herausgeber:innen haben sich nach bestem Wissen und Gewissen
bemüht, die Urheber:innen und Quellen der verwendeten Abbildungen zu bestimmen.
Sollte es dennoch zu irrtümlichen Angaben gekommen sein, bitten wir um eine dement-
sprechende Nachricht.

INHALTSVERZEICHNIS

GRUNDLAGEN DER OBJEKTOIENTIERTEN PROGRAMMIERUNG MIT JAVA

Einleitung

Wegweiser durch das Studienskript	6
Basisliteratur	7
Weiterführende Literatur	8
Übergeordnete Lernziele	10

Lektion 1

Einführung in die objektorientierte Systementwicklung	11
1.1 Objektorientierung als Sichtweise auf komplexe Systeme	12
1.2 Das Objekt als Grundkonzept der Objektorientierung	13
1.3 Phasen im objektorientierten Entwicklungsprozess	14
1.4 Grundprinzip der objektorientierten Systementwicklung	16

Lektion 2

Einführung in die objektorientierte Modellierung	21
2.1 Strukturieren von Problemen mit Klassen	22
2.2 Identifizieren von Klassen	23
2.3 Attribute als Eigenschaften von Klassen	25
2.4 Methoden als Funktionen von Klassen	27
2.5 Beziehungen zwischen Klassen	29
2.6 Unified Modeling Language (UML)	32

Lektion 3

Programmieren von Klassen in Java	37
3.1 Einführung in die Programmiersprache Java	38
3.2 Grundelemente einer Klasse in Java	40
3.3 Attribute in Java	42
3.4 Methoden in Java	44
3.5 main-Methode: Startpunkt eines Java-Programms	50

Lektion 4

Java Sprachkonstrukte	55
4.1 Primitive Datentypen	56
4.2 Variablen	58
4.3 Operatoren und Ausdrücke	60
4.4 Kontrollstrukturen	65
4.5 Pakete und Sichtbarkeitsmodifikatoren	74

Lektion 5	
Vererbung	79
5.1 Modellierung von Vererbung im Klassendiagramm	81
5.2 Programmieren von Vererbung in Java	86
Lektion 6	
Wichtige objektorientierte Konzepte	95
6.1 Abstrakte Klassen	96
6.2 Polymorphie	100
6.3 Statische Attribute und Methoden	103
Lektion 7	
Konstruktoren zur Erzeugung von Objekten	107
7.1 Der Standard-Konstruktor	108
7.2 Überladen von Konstruktoren	111
Lektion 8	
Ausnahmebehandlung mit Exceptions	119
8.1 Typische Szenarien der Ausnahmebehandlung	120
8.2 Standard-Exceptions in Java	123
8.3 Definieren eigener Exceptions	128
Lektion 9	
Programmierschnittstellen mit Interfaces	131
9.1 Typische Szenarien für Programmierschnittstellen	132
9.2 Interfaces als Programmierschnittstellen in Java	135
Verzeichnisse	
Literaturverzeichnis	142
Abbildungsverzeichnis	143

EINLEITUNG

HERZLICH WILLKOMMEN

WEGWEISER DURCH DAS STUDIENSKRIPT

Dieses Studienskript bildet die Grundlage Ihres Kurses. Ergänzend zum Studienskript stehen Ihnen weitere Medien aus unserer Online-Bibliothek sowie Videos zur Verfügung, mit deren Hilfe Sie sich Ihren individuellen Lern-Mix zusammenstellen können. Auf diese Weise können Sie sich den Stoff in Ihrem eigenen Tempo aneignen und dabei auf lerntypspezifische Anforderungen Rücksicht nehmen.

Die Inhalte sind nach didaktischen Kriterien in Lektionen aufgeteilt, wobei jede Lektion aus mehreren Lernzyklen besteht. Jeder Lernzyklus enthält jeweils nur einen neuen inhaltlichen Schwerpunkt. So können Sie neuen Lernstoff schnell und effektiv zu Ihrem bereits vorhandenen Wissen hinzufügen.

In der IU Learn App befinden sich am Ende eines jeden Lernzyklus die Interactive Quizzes. Mithilfe dieser Fragen können Sie eigenständig und ohne jeden Druck überprüfen, ob Sie die neuen Inhalte schon verinnerlicht haben.

Sobald Sie eine Lektion komplett bearbeitet haben, können Sie Ihr Wissen auf der Lernplattform unter Beweis stellen. Über automatisch auswertbare Fragen erhalten Sie ein direktes Feedback zu Ihren Lernfortschritten. Die Wissenskontrolle gilt als bestanden, wenn Sie mindestens 80 % der Fragen richtig beantwortet haben. Sollte das einmal nicht auf Anhieb klappen, können Sie die Tests beliebig oft wiederholen.

Wenn Sie die Wissenskontrolle für sämtliche Lektionen gemeistert haben, führen Sie bitte die abschließende Evaluierung des Kurses durch.

Die IU Internationale Hochschule ist bestrebt, in ihren Skripten eine gendersensible und inklusive Sprache zu verwenden. Wir möchten jedoch hervorheben, dass auch in den Skripten, in denen das generische Maskulinum verwendet wird, immer Frauen und Männer, Inter- und Trans-Personen gemeint sind sowie auch jene, die sich keinem Geschlecht zuordnen wollen oder können.

BASISLITERATUR

Gamma, E. et al. (1995): *Design Patterns: Elements of Re-usuable Object-Oriented Software.*

Addison-Wesley Longman Publishing, Boston.

Gamma, E. et al. (2015): *Design Patterns: Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software.* mitp Verlag, Frechen.

Krüger, G./Stark, T. (2014): *Handbuch der Java-Programmierung.* 8. Auflage, Addison-Wesley, Weinheim. (Im Internet verfügbar).

Lahres, B./Rayman, G. (2006): *Praxisbuch Objektorientierung.* Galileo Computing. (Im Internet verfügbar).

Oestereich, B. (2013): *Analyse und Design mit der UML 2.5: Objektorientierte Softwareentwicklung.* 11. Auflage, Oldenbourg Wissenschaftsverlag, München.

Oracle (2017): *The Java Tutorials.* (Im Internet verfügbar).

Ullenboom, U. (2014): *Java ist auch eine Insel.* Galileo Computing. 11. Auflage. (Im Internet verfügbar).

WEITERFÜHRENDE LITERATUR

LEKTION 1

Johnson, R./Moses, D. R. (2008): *Objects-first vs. Structures-first Approaches to OO Programming Education: An Empirical Study*. In: Academy of Information and Management Sciences Journal, 11. Jg., Heft 2, S. 95–102.

LEKTION 2

Rajagopal, D./Thilakavalli, K. (2017): *A Study: UML for OOA and OOD*. In: International Journal of Knowledge Content Development & Technology, 7. Jg., Heft 2, S. 5–20.

LEKTION 3

Xing, C./Belkhouche, B. (2003): *On Pseudo Object-Oriented Programming Considered Harmful*. In: Communications of the ACM, 46. Jg., Heft 10, S. 115–117.

LEKTION 4

Ourosoff, N. (2002): *Primitive Types in Java Considered Harmful*. In: Communications of the ACM, 45. Jg., Heft 8, S. 105f.

LEKTION 5

Nasseri, E./Counsell S. (2010): *Java Method Calls in the Hierarchy—Uncovering Yet another Inheritance Foible*. In: CIT Journal of Computing and Information Technology, 18. Jg., Heft 2, S. 159–165.

LEKTION 6

Alkazemi, B. Y./Grami, G. M. (2012): *Utilizing BlueJ to Teach Polymorphism in an Advanced Object-Oriented Programming Course*. In: Journal of Information Technology Education: Innovations in Practice, 11. Jg., S. 271–282. (Im Internet verfügbar).

LEKTION 7

Bruno, R./Ferreira, P. (2018): *A Study on Garbage Collection Algorithms for Big Data Environments*. In: ACM Computing Surveys, 51. Jg., Heft 1, S. 1–35.

LEKTION 8

Rashkovits, R./Lavy, I. (2012): *Students' Understanding of Advanced Properties of Java Exceptions*. In: Journal of Information Technology Education: Innovations in Practice, 11. Jg., S. 327–352. (Im Internet verfügbar).

LEKTION 9

Sousa, B. L./Bigonha, M. A./Ferreira, K. A. (2018): *When GOF Design Patterns occur with God Class and Long Method Bad Smells? – An Empirical Analysis*. In: INFOCOMP: Journal of Computer Science, 17. Jg., Heft 1. (Im Internet verfügbar).

ÜBERGEORDNETE LERNZIELE

Der Kurs **Grundlagen der objektorientierten Programmierung mit Java** vermittelt Ihnen die grundlegenden Kompetenzen der objektorientierten Programmierung. Die dazugehörigen theoretischen Konzepte werden unmittelbar anhand der Programmiersprache Java gezeigt und geübt. Ziel dieses Kurses ist es u. a., dass Sie die Grundkonzepte der objektorientierten Modellierung und Programmierung kennen und voneinander abgrenzen können. Zudem werden die Grundkonzepte und -elemente der Programmiersprache Java erläutert und Sie können Erfahrungen in deren Verwendung sammeln. So sind Sie nach Abschluss dieses Kurses befähigt, selbstständig geeignete Lösungen für konkret beschriebene Probleme zu erstellen.

LEKTION 1

EINFÜHRUNG IN DIE OBJEKTORIENTIERTE SYSTEMENTWICKLUNG

LERNZIELE

Nach der Bearbeitung dieser Lektion werden Sie wissen, ...

- was unter dem Begriff Objektorientierung verstanden wird.
- was Objekte sind und woraus diese bestehen.
- aus welchen Phasen der objektorientierte Software-Entwicklungsprozess besteht.
- was die Grundprinzipien objektorientierter Software-Entwicklung sind.

1. EINFÜHRUNG IN DIE OBJEKTOIENTIERTE SYSTEMENTWICKLUNG

Aus der Praxis

Als Entwickler soll Herr Koch einen Online-Shop für den Verkauf von Medien aller Art (Bücher, Musik, Filme, Spiele) programmieren. Die Auftraggeberin Frau Lange erläutert ihm, welche Artikel verkauft werden sollen und welche speziellen Angebote sie sich ausgedacht hat. Nun soll es mit der Entwicklung losgehen und Herr Koch überlegt, wie er an diese Aufgabe am besten herangehen soll.

1.1 Objektorientierung als Sichtweise auf komplexe Systeme

Objektorientierung

Das ist ein Ansatz zur Unterstützung der Erstellung, Weiterentwicklung und Wartung komplexer betrieblicher IT-Systeme. Das IT-System wird dabei durch das Zusammenspiel komplexer Objekte beschrieben.

Unter der Bezeichnung **Objektorientierung** (kurz: OO) wurde in der Softwaretechnik ein Ansatz entwickelt, der insbesondere die Erstellung, Weiterentwicklung und Wartung von komplexen betrieblichen IT-Systemen unterstützt. Dabei dient die Objektorientierung als eine Sichtweise auf komplexe Systeme, bei der diese als ein Zusammenspiel von abstrakten oder realen Einheiten – eben den Objekten – aufgefasst werden.

Die „Objektorientierung“ ist somit keine Programmiersprache, keine Modellierungssprache, kein Betriebssystem, kein Programm, keine Anwendung, kein Vorgehensmodell, keine Entwicklungsmethode, keine Entwicklungsumgebung und keine Datenbank. Jedoch gibt es Sprachen, Methoden und Datenbanken, welche die objektorientierte Sichtweise gezielt unterstützen.

Die großflächige Verbreitung der Objektorientierung in der industriellen Praxis begann in den frühen 1990er Jahren. Ausschlaggebend dafür waren u. a. die sehr hohen Kosten für die Wartung und Weiterentwicklung von großen IT-Systemen. Typischerweise werden betriebliche Anwendungen kontinuierlich weiterentwickelt und den geschäftlichen und technischen Trends angepasst. So steigt gemeinsam mit dem Funktionsumfang von IT-Systemen der Umfang der programmierten Programmcodezeilen (Lines of Code, LOC). Betriebliche Informationssysteme bestehen oft aus mehreren Millionen Zeilen Programmcode, die von bis zu mehreren hundert Entwicklern gemeinsam erstellt wurden.

Die Objektorientierung reiht sich in die geschichtliche Entwicklung der Programmierparadigmen ein und ist heute die Grundlage fast aller Entwicklungsprojekte für betriebliche Informationssysteme. Tabelle 1 zeigt eine Übersicht über die geschichtliche Entwicklung von Programmierkonzepten, ihren typischen Elementen und deren Einsatzgebiete.

Tabelle 1: Übersicht über die geschichtliche Entwicklung von Programmierkonzepten

Programmierkonzept	Typische Elemente	Einsatzgebiete
Maschinencode	<i>1001001; 1 und 0 als einzige Ausdrucksmöglichkeit</i>	Nur noch historisch; heute keine Einsatzgebiete mehr
Assemblercode	<i>movb \$0x61; Prozessorbefehle und direktes Ansprechen von technischen Speicheradressen</i>	Steuerungen für elektrotechnische Geräte (Lüftungen, Motoren, Klimaanlagen), Reaktive Systeme (Sensorsysteme), hardwarenahe Programmierung
Imperative Programmierung	<i>WHILE, FOR, GOTO; Weiterentwicklung von Assemblercode, erlauben Schleifen und gezielte Sprünge</i>	Kleine Programme zur Lösung einfacher Aufgaben, in der Regel in speziellen Programmiersprachen
Strukturierte Programmierung	<i>Prozeduren; Strukturierung eines Programms in Prozeduren (Funktionen) und Unterprozeduren (Unterfunktionen)</i>	Einfache Webanwendungen, Technische Steuerkomponenten
Objektorientierte Programmierung	<i>Klassen, Objekte, Beziehungen; ein Programm besteht aus kooperierenden Objekten</i>	Große und komplexe industrielle Softwaresysteme
Komponentenbasierte Entwicklung	<i>Komponenten, Schnittstellen; Teile eines Systems werden als eine Komponente zusammengefasst, die ganz bestimmte Aufgaben erfüllt</i>	Wiederverwendung von bereits programmierten Funktionen; Einzelne Komponenten sind in der Regel objektorientiert programmiert.
Modellgetriebene Entwicklung	<i>Modelle, Code-Generatoren; Programmcode wird mit Hilfe von Code-Generatoren aus Software-Modellen automatisch erzeugt</i>	Wiederverwendung von einfacher Anpassbarkeit von Querschnittsfunktionen (z. B. Login-Mechanismen), der generierte Programmcode kann den Konzepten der Objektorientierung entsprechen

Quelle: erstellt im Auftrag der IU, 2013.

1.2 Das Objekt als Grundkonzept der Objektorientierung

Der Begriff **Objekt** ist direkt im Namen „Objektorientierung“ enthalten und bezeichnet einen Bestandteil eines Systems. Für die objektorientierte Programmierung bedeutet dies, dass ein Objekt Bestandteil eines Programms ist und auch, dass ein objektorientiert entwickeltes Programm nahezu ausschließlich aus Objekten besteht. Ein Objekt selber kann wiederum Attribute und Methoden enthalten. Mithilfe von **Attribut(en)** (auch: Eigenschaften) können in einem Objekt Informationen (z. B. aktuelle Werte von Variablen) gespeichert werden. Die Attribute eines Objekts werden vor einem direkten Zugriff von anderen

Objekt
Das ist ein Bestandteil eines Systems. Objektorientierte Systeme liegen zum Zeitpunkt ihrer Ausführung in Form von Objekten im Hauptspeicher eines Rechners.

Attribute

Das sind Elemente von Objekten, die zum Speichern von konkreten Werten verwendet werden.

Methode

Das sind Elemente von Objekten, mit denen Werte von Attributen erstellt, gelesen, verändert und Berechnungen durchgeführt werden können.

Objekten versteckt und damit geschützt. Das bedeutet, dass die aktuell in einem Objekt gespeicherten Werte nicht von anderen Objekten geändert werden können (**Prinzip der Datenkapselung**).

Das Lesen und Ändern von Attributen eines Objekts geschieht durch den Einsatz von Methoden eines Objekts. Mit den Methoden (auch: Funktionen, Operationen) bietet ein Objekt anderen Objekten des Systems die Möglichkeit, auf die eigenen Attribute zuzugreifen, sie auszulesen, zu verändern, zu aktualisieren oder Berechnungen durchzuführen. Ein Objekt greift auf ein anderes Objekt zu, indem es eine **Methode** auf dem Objekt aufruft.

Beispielszenario

Eine Uhr, wie in der folgenden Abbildung skizziert, soll als digitales Objekt im Sinne der Objektorientierung nachgebildet werden. Dazu wird zuerst ein Name für die Uhr festgelegt, in diesem Fall lautet der Name des Objekts „Digitale Uhr“. Als nächstes werden für die Uhr typische Attribute identifiziert, also Informationen, die benötigt werden, um das Objekt „Digitale Uhr“ zu beschreiben: die „aktuelle Uhrzeit“ sowie die „aktuelle Zeitzone“. Nun fehlen noch die Methoden des Objekts, also Funktionen, die das Objekt „Digitale Uhr“ bereitstellen muss, damit es gegenüber anderen Objekten seine Funktion erfüllen kann. Hier sind es „Uhrzeit einstellen“, „Uhrzeit ausgeben“ und „Zeitzone einstellen“.

Abbildung 1: Skizze einer Uhr

 Berlin	Name des Objektes:	Digitale Uhr
	Attribute des Objektes:	<ul style="list-style-type: none"> • Aktuelle Uhrzeit • Aktuelle Zeitzone
	Methoden des Objektes:	<ul style="list-style-type: none"> • Uhrzeit einstellen • Uhrzeit ausgeben • Zeitzone einstellen

Quelle: erstellt im Auftrag der IU, 2013.

1.3 Phasen im objektorientierten Entwicklungsprozess

Bevor mit der eigentlichen Programmierung von komplexen IT-Systemen begonnen werden kann, müssen alle Entwickler genau verstehen:

- was das System konkret können soll,
- welche Informationen in dem System gespeichert werden müssen
- und wer und für welchen Zweck jemand das System später benutzt.

Darüber hinaus müssen die grobe Struktur des Systems in Form einer Systemarchitektur durch den Systemarchitekten vorgegeben werden.

Bevor Herr Koch also nun mit der Programmierung des Online-Shops tatsächlich beginnen kann, muss er das Problem beziehungsweise die Aufgabenstellung analysieren und sich überlegen, aus welchen Objekten das System bestehen soll. Erst dann kann er mit dem zielgerichteten Programmieren beginnen. Das Durchlaufen dieser verschiedenen Phasen nennt man **Software-Entwicklungsprozess** (auch: **SW-Prozess**). Ziel eines SW-Entwicklungsprozesses ist ein lauffähiges System, das die Anforderungen der Auftraggeber hinsichtlich Funktionalität, Qualität und Entwicklungsaufwand zufriedenstellend erfüllt.

In einem objektorientierten Entwicklungsprozess lassen sich drei verschiedene Phasen unterscheiden:

- Objektorientierte Analyse (kurz: OOA)
- Objektorientiertes Design (kurz: OOD)
- Objektorientierte Programmierung (kurz: OOP)

Während der **objektorientierten Analyse** wird bestimmt, was ein System tun soll. Ziel der Analyse ist ein umfassendes Verständnis der fachlichen Zusammenhänge des zu bauenden Systems. Hierzu werden Objekte der realen Welt in einem fachlichen Modell nachgebildet und nur die für das System relevanten Eigenschaften (Attribute) berücksichtigt. Dieses Analysemmodell ist zum einen ein Mittel zur Kommunikation zwischen den Entwicklern und dem Auftraggeber beziehungsweise den Anwendern des Systems. Zum anderen bildet es den Ausgangspunkt für die nächste Phase: das objektorientierte Design.

Das **objektorientierte Design** bildet die Brücke zwischen der Analyse und der Implementierung. Hierzu wird auf Basis des Analysemmodells das Design des Systems festgelegt: Das fachliche Analysemmodell wird dahin gehend um technische Informationen erweitert, sodass ein Programmierer in der Lage ist, auf Basis des Designs den Programmcode zu implementieren. Dabei wird festgelegt, welche Arten von Objekten es geben soll, welche Attribute und Methoden sie haben und auf welche Art welche Objekte miteinander kooperieren. Für komplexe Systeme kann diese Aufgabe nur von erfahrenen Systemarchitekten durchgeführt werden. Die Qualität des Designs hat dabei in der Regel maßgeblichen Einfluss auf die Qualität des programmierten Systems sowie auf alle zukünftigen Aktivitäten zur Erweiterung und Anpassung des Systems.

Bei den Aktivitäten der **objektorientierten Programmierung** wird ein Systemdesign in funktionierenden Programmcode übersetzt. Dazu programmiert der Entwickler unter Einhaltung der im Design getroffenen Vorgaben den Programmcode. Analyse und Design sind notwendige Vorarbeiten, das eigentliche IT-System jedoch ergibt sich ausschließlich durch die implementierten **Programmcodes**. Wenn sich also die Programmierer nicht an die Vorgaben halten, kann es passieren, dass Designentscheidungen nicht so umgesetzt werden und das System sich letztendlich anders verhält als es vorgesehen ist. Andererseits pflanzen sich Fehler aus der Analyse und dem Design im Programmcode fort, falls sie der Entwickler nicht rechtzeitig erkennt.

Beispiel

Software-Entwicklungsprozess

Dieser Prozess besteht aus Aktivitäten mit dem Ziel Software zu erstellen, weiterzuentwickeln oder zu warten. Dieser Prozess ist in mehrere Phasen unterteilt.

Objektorientiertes Design

Das ist eine Phase im objektorientierten SW-Prozess. Ziel ist es, aufbauend auf dem Analysemmodell das technische Design des Systems zu erstellen. Ergebnis ist das Design des Systems.

Objektorientierte Programmierung

Dies ist eine Phase im objektorientierten SW-Prozess. Ziel ist die Erstellung von lauffähigem Programmcode auf Basis des Designs. Ergebnis ist das fertige Programm.

Für die Umsetzung des Online-Shops von Herrn Koch bedeutet es, dass er sich zunächst einmal intensiv mit den Wünschen und Vorstellungen von Frau Lange auseinanderzusetzen hat. Dazu erarbeitet er sich ein Verständnis, welche geschäftlichen Aktivitäten für einen Online-Shop relevant sind und wie genau die einzelnen Aktivitäten (z. B. Warenbestellung, Zahlungsabwicklung, Mahnwesen, etc.) ablaufen.

Anschließend überlegt er sich das technische Design des Online-Shops. Damit ist allerdings nicht das Aussehen der Nutzeroberfläche gemeint, sondern aus welchen (technischen) Objekten der Online-Shop bestehen soll und wie diese kooperieren sollen. Darüber hinaus muss Herr Koch sich noch entscheiden, welche Technologien eingesetzt werden sollen: Er kann entweder ein bestehendes Shopsystem nehmen und anpassen oder ein neues programmieren. Außerdem ist noch relevant, welche Datenbank infrage kommt und in welcher Programmiersprache er den Online-Shop programmieren möchte.

1.4 Grundprinzip der objektorientierten Systementwicklung

Wie schon genannt ist die Objektorientierung eine Sichtweise auf komplexe Systeme als Zusammenspiel kooperierender Objekte. Grundsätzlich lässt sich jedes System auch ohne Objektorientierung planen und bauen. Allerdings haben die Erfahrungen der letzten 20 Jahre Software Engineering gezeigt, dass es bei konsequenter Anwendung der Objektorientierung weniger Probleme mit der Skalierung, der Stabilität und der Erweiterbarkeit von IT-Systemen gibt. Daher kann Objektorientierung bei der Softwareentwicklung als Prinzip und Mittel verstanden werden, um die Komplexität von Softwaresystemen in den Griff zu bekommen.

Insbesondere die Qualitätsziele

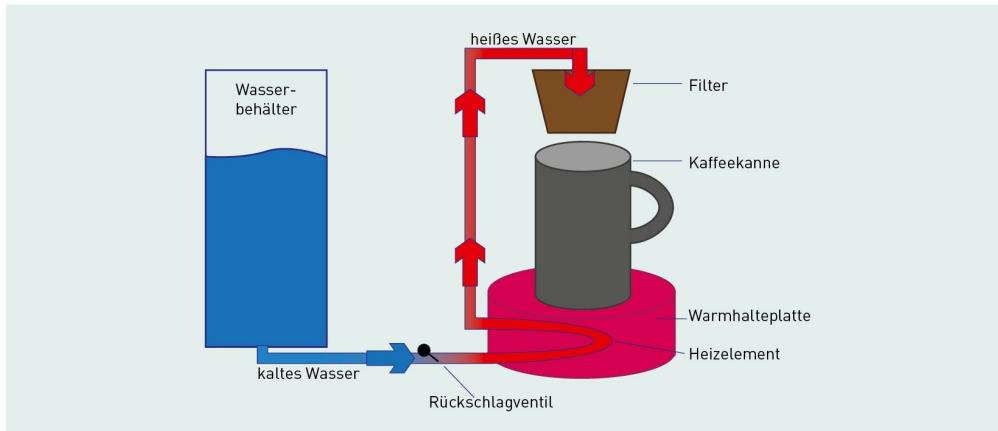
- einfache Erweiterbarkeit,
- bessere Testbarkeit und
- bessere Wartbarkeit

lassen sich durch objektorientierte Konzepte relativ einfach realisieren.

Beispiel

Ohne auf die theoretischen Hintergründe einzugehen, wird im folgenden Beispielszenario einer einfachen Kaffeemaschine gezeigt, wie sich die Objektorientierung auf die Betrachtung technischer Systeme auswirkt.

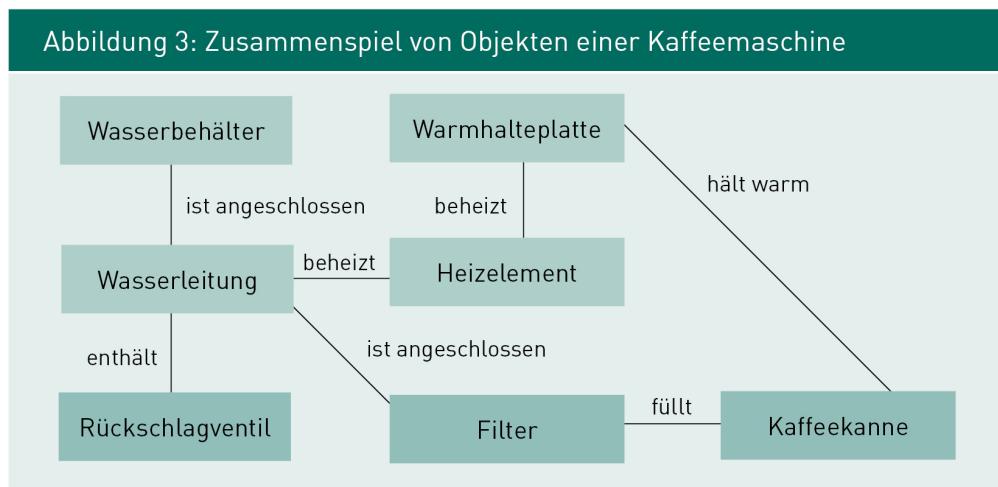
Abbildung 2: Beispielszenario einer einfachen Kaffeemaschine



Quelle: erstellt im Auftrag der IU, 2013 in Anlehnung an Reviewland (2017).

In Abbildung 2 ist der schematische Querschnitt einer Kaffeemaschine dargestellt. Übertragen in die Welt der Objektorientierung stellt sich die Kaffeemaschine als ein „Zusammenspiel kooperierender Objekte“ dar, so wie in Abbildung 3 gezeigt. Das Objekt Wasserbehälter ist an das Objekt Wasserleitung angeschlossen. Die Wasserleitung enthält ein Objekt Rückschlagventil und ist am Objekt Heizelement angeschlossen. Das Objekt Heizelement beheizt die Wasserleitung und das Objekt Warmhalteplatte. Die Warmhalteplatte hält das Objekt Kaffeekanne warm. Die Kaffeekanne wird durch das Objekt Filter gefüllt.

Abbildung 3: Zusammenspiel von Objekten einer Kaffeemaschine

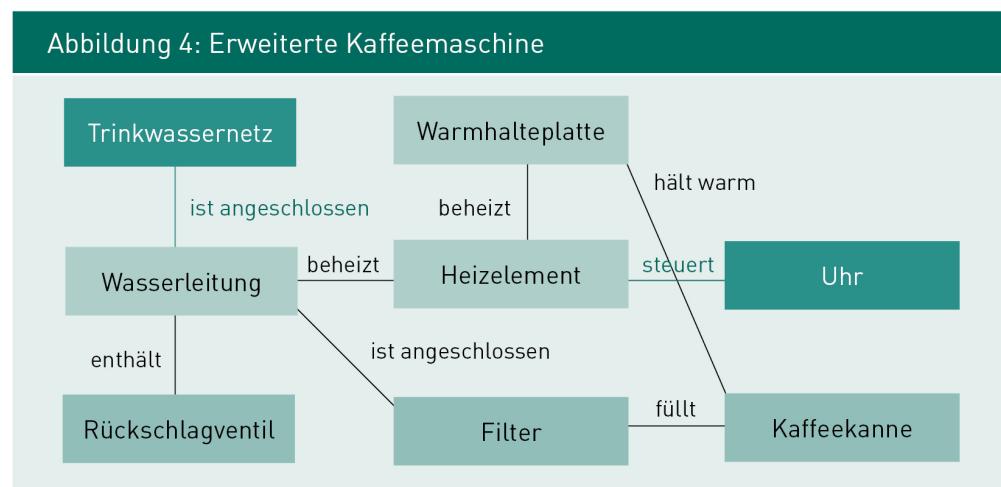


Quelle: erstellt im Auftrag der IU, 2013.

Nach einiger Zeit stellt Frau Lange fest, dass sie die Kaffeemaschine gerne durch eine Uhr steuern möchte und statt dem Wasserbehälter lieber einen direkten Anschluss an das Trinkwassernetz hätte. Abbildung 4 zeigt nun eine im Sinne der Objektorientierung erweiterte Kaffeemaschine: Das Objekt Wasserbehälter wurde durch das Trinkwassernetz ersetzt. Damit kooperiert die Wasserleitung nun nicht mehr mit dem Behälter, sondern

dem Trinkwassernetz (dargestellt durch die Verbindungsleitung zwischen Trinkwassernetz und Wasserleitung). Weiterhin wurde ein Objekt Uhr an das Heizelement angeschlossen. Über diese Verbindung kann nun die Funktion „Heizen“ des Heizelements durch die Uhr gesteuert werden. Alle anderen bestehenden internen Zusammenhänge der Kaffeemaschine bleiben jedoch unverändert bestehen.

Abbildung 4: Erweiterte Kaffeemaschine



Quelle: erstellt im Auftrag der IU, 2013.

Dieses sehr einfache Beispiel veranschaulicht eine wesentliche Eigenschaft von objektorientierten Systemen: die Aufteilung der internen Elemente in Bereiche, die für eine ganz bestimmte Funktion zuständig sind und das Zusammenwirken der einzelnen Objekte. Tabelle 2 zeigt die Zuständigkeiten der Objekte der Kaffeemaschine. Auch wenn die Kaffeemaschine nicht vollständig programmiert werden kann – weil immer noch Wasser, Kaffee und Wärme benötigt werden – werden IT-Systeme nach einem ähnlichen Prinzip designet und gebaut.

Tabelle 2: Objekte einer Kaffeemaschine

Objekte	Zuständigkeiten/Funktion
Wasserbehälter, später ersetzt durch Trinkwassernetz	Wasserversorgung
Wasserleitung	Transportieren von Wasser
Rückschlagventil	Strömungsrichtung des Wassers regeln
Heizelement	Elektrische Energie in Wärme umwandeln
Warmhalteplatte	Kaffeekanne wärmen
Filter	Kaffeepulver zurückhalten und Wasser durchströmen lassen

Objekte	Zuständigkeiten/Funktion
Uhr	Bei Erreichen der eingestellten Uhrzeit Signal auslösen

Quelle: erstellt im Auftrag der IU, 2013.

Durch die klare **Kapselung** von Zuständigkeiten in ganz bestimmte Objekte und die klare Definition von Schnittstellen, können Objekte später ausgetauscht (in unserem Beispiel wurde der Wasserbehälter durch das Trinkwassernetz ausgetauscht) oder erweitert (in unserem Beispiel die Erweiterung der Kaffeemaschine um eine Uhr zur Zeitsteuerung) werden, ohne die Stabilität und Funktionalität des gesamten Systems zu beeinflussen. Das erleichtert wiederum die Maßnahmen zur Qualitätssicherung, da nicht das ganze System komplett neu getestet werden muss, sondern nur die betroffenen Teile des Systems.

Kapselung
Dabei werden Objekte gezielt für abgeschlossene Funktionen und Aufgaben erstellt. Über Schnittstellen können andere Objekte diese konkreten Funktionen und Aufgaben aufrufen.



ZUSAMMENFASSUNG

Die Objektorientierung ist eine Sichtweise auf komplexe Systeme, bei der das System durch das Zusammenspiel kooperierender Objekte beschrieben wird.

Bei der objektorientierten Systementwicklung werden Dinge der „realen“ Welt durch Objekte der „digitalen“ Welt nachgebildet. Dabei werden nur die für den Zweck des Systems relevanten Werte und Funktionen berücksichtigt. Die digital nachgebildeten Objekte speichern Werte in ihren Attributen und können in ihren Methoden Werte berechnen. Objektorientierte Programme bestehen aus kooperierenden „digitalen“ Objekten. Die Kooperation der Objekte erfolgt durch das gegenseitige Aufrufen von Methoden.

Der objektorientierte Entwicklungsprozess besteht aus den Phasen objektorientierte Analyse, objektorientiertes Design und objektorientierte Implementierung. Ziel des Entwicklungsprozesses ist ein System, das den Anforderungen des Auftraggebers in Funktionalität, Qualität und Aufwand entspricht.

Durch die Aufteilung von Systemen in einzelne Objekte mit ganz bestimmten Zuständigkeiten hilft die Objektorientierung bei der Entwicklung, Weiterentwicklung und der Qualitätssicherung von komplexen industriellen Softwaresystemen.

LEKTION 2

EINFÜHRUNG IN DIE OBJEKTORIENTIERTE MODELLIERUNG

LERNZIELE

Nach der Bearbeitung dieser Lektion werden Sie wissen, ...

- wie die Begriffe Objekt und Klasse zusammengehören.
- wie man Klassen identifiziert.
- wie Attribute von Klassen beschrieben werden.
- wie Methoden von Klassen beschrieben werden.
- wie Beziehungen zwischen Klassen beschrieben werden.
- was die Unified Modeling Language (UML) ist.

2. EINFÜHRUNG IN DIE OBJEKTOIENTIERTE MODELLIERUNG

Aus der Praxis

Nachdem Herr Koch nun ganz sicher ist, dass sein Projekt mehrere Wochen dauern wird, hat er sich dazu entschlossen den Online-Shop für Frau Lange zuerst einmal objektorientiert zu analysieren. Dazu möchte er alle benötigten Objekte, deren Attribute und deren Methoden identifizieren und in geeigneter Form dokumentieren. Er hat zwar eine grobe Idee, wie er den Online-Shop und die Artikel darstellen kann, jedoch will er sicher gehen, dass er vor dem Programmieren auch alles verstanden und durchdacht hat. Dafür muss Herr Koch nun wissen, wie man Objekte identifiziert und mit welchen Mitteln man sowohl die Elemente von Objekten (also Attribute und Methoden) als auch die Kooperation zwischen Objekten darstellt.

2.1 Strukturieren von Problemen mit Klassen

In Lektion 1 haben wir das **Objekt** als Grundkonzept der Objektorientierung kennengelernt. Ein Objekt ist eine abgeschlossene, für sich handlungsfähige Einheit, beispielsweise das Auto „E-CL 2343“, die Lampe „Kyist“, der Kunde „Hans Meier“ oder der Vertrag „P-DFD-23“. Bei der objektorientierten Entwicklung werden Objekte aus der physischen (realen) Welt in Objekte der Programmiersprache nachgebildet. So gibt es dann zum „echten“ Kunden „Hans Meier“ ein digitales Gegenstück in Form eines digitalen Objektes „Hans Meier“. Der real existierende Kunde (Herr Hans Meier) wird in Form eines „digitalen Objektes“ der jeweiligen Programmiersprache des IT-Systems nachgebildet. Bei dieser digitalen Nachbildung werden aber nur die wichtigsten Eigenschaften wie Name, Vorname, Geburtsdatum oder Geschlecht übernommen. Eigenschaften wie Haarfarbe, Augenfarbe oder Farbe der Kleidung interessieren in den meisten Fällen nicht, da sie für die Aufgaben des Systems nicht benötigt werden.

Klasse

Das ist eine Struktur zur Bildung von digitalen Objekten. Sie legen fest, woraus konkrete Objekte bestehen. Klassen sind die zentralen Elemente im objektorientierten SW-Entwicklungsprozess.

Eine **Klasse** liefert in der Objektorientierung die Struktur, die zur Bildung eines „digitalen Objektes“ erforderlich ist. Auf Basis einer Klasse können beliebig viele Objekte erzeugt werden. Alle aus einer Klasse erzeugten Objekte haben die gleiche Struktur, also dieselben Attribute und Methoden. Eine Klasse ist insofern etwa vergleichbar mit einer Word-Dokumentvorlage, und ein Objekt ist ein aus dieser Vorlage erzeugtes Word-Dokument. Aus welchen Klassen ein objektorientiertes System besteht und wie die Struktur der benötigten Klassen aussieht, wird in der Phase „objektorientiertes Design“ festgelegt. Ausgangspunkt für das Design sind die ermittelten und spezifizierten Anforderungen.

Bei der objektorientierten Programmierung von IT-Systemen werden (fast) nur Klassen implementiert. Das heißt, der Entwickler legt die Struktur von Klassen fest: also aus welchen Elementen (Attributen und Methoden) die aus den Klassen erzeugten Objekte bestehen. Ein objektorientiertes System erzeugt dann bei Bedarf die entsprechenden Objekte, zum Beispiel aus Datenbankeinträgen oder Benutzereingaben. Durch die Beschreibung der Klassen wird gewährleistet, dass sich alle Objekte, die auf Basis dieser Klassen erzeugt werden, gleich verhalten. Alle Objekte einer Klasse haben dieselben Attribute und verfügen über die gleichen Funktionen und Routinen (die sogenannten Methoden). Damit können alle aus einer Klasse erzeugten Objekte von den Funktionen des Systems in gleicher Weise verarbeitet werden.

Die zentralen Aktivitäten bei der Analyse und später beim Design sind das Identifizieren und Beschreiben von Klassen. Obwohl die Sichtweise „Objektorientierung“ heißt, stehen eigentlich vielmehr die Klassen im Vordergrund der Betrachtung.

Für die Analyse und das Design des Online-Shops muss Herr Koch nun also zunächst geeignete Klassen identifizieren, die später als Vorgabe für die Programmierung des Online-Shops dienen.

2.2 Identifizieren von Klassen

Um bei der Analyse zu bestimmen, was ein System tun soll, und dann darauf aufbauend später beim Design festzulegen, aus welchen Klassen das System bestehen soll, wird in einem ersten Schritt ein **Analysemmodell** des Systems entwickelt: Aus der Aufgabenstellung beziehungsweise den Anforderungen an das System müssen Kandidaten für Klassen identifiziert und notiert werden.

Eine etablierte Vorgehensweise zur Identifikation von Klassen kann wie folgt beschrieben werden:

1. Alle Hauptwörter (Substantive) der Aufgabenstellung werden als Kandidaten für Klassen markiert.
2. Dann wird geprüft, ob die markierten Hauptwörter durch weitere Hauptwörter beschrieben werden können oder ob sie Beziehungen oder Abhängigkeiten zu anderen Hauptwörtern haben. Trifft eine der beiden Aussagen zu, wird das Hauptwort als Klasse modelliert.
3. Wird das Hauptwort verwendet, um ein anderes Hauptwort zu detaillieren, wird es als Attribut einer Klasse modelliert.
4. Alle verbleibenden Wörter, die weder Klasse noch Attribut sind, werden auf Relevanz überprüft und ggf. von der Liste der Kandidaten für Klassen gestrichen.

Analysemmodell

Das ist das Ergebnis der objektorientierten Analyse und dient zur Kommunikation zwischen den Entwicklern und dem Auftraggeber bzw. den Anwendern des Systems.

Beispiel

Die im Online-Shop von Frau Lange verkauften Artikel sollen Medien aller Art sein, insbesondere Bücher, Musikartikel, Filme und Spiele. Jeder Artikel hat einen Hersteller, einen Titel und eine Artikelnummer. Bücher und Spiele haben einen Autor, Filme einen Regisseur und Musikartikel einen Interpreten.

Schritt 1: Nach dem Markieren aller Hauptwörter besteht die Liste aus folgenden Einträgen (dabei werden die Hauptwörter nur in Einzahl notiert):

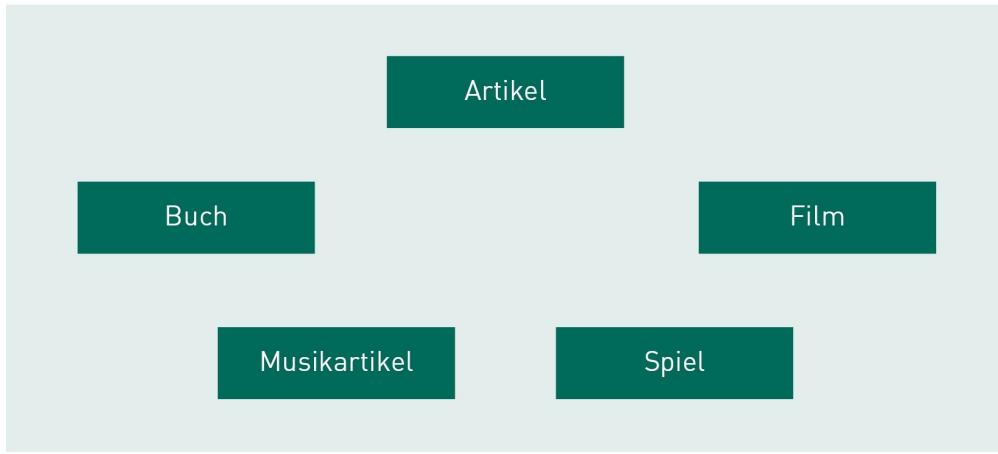
- Online-Shop
- Artikel
- Medien
- Art
- Buch
- Musikartikel
- Film
- Spiel
- Hersteller
- Titel
- Artikelnummer
- Autor
- Regisseur
- Interpret

Schritt 2–4: Überprüfung auf Kandidat für Klasse, Attribut und generell auf Relevanz:

- Online-Shop: nicht relevant, weil es das System als Ganzes bezeichnet
- Artikel: als Klasse relevant
- Medien: nicht relevant, da nur Beschreibung zu Artikel
- Art: nicht relevant, da nur Beschreibung zu Artikel
- Buch: als Klasse relevant, da durch Attribute verfeinert
- Musikartikel: als Klasse relevant, da durch Attribute verfeinert
- Film: als Klasse relevant, da durch Attribute verfeinert
- Spiel: als Klasse relevant, da durch Attribute verfeinert
- Hersteller: als Attribut relevant, da Beschreibung zu Klassen
- Titel: als Attribut relevant, da Beschreibung zu Klassen
- Artikelnummer: als Attribut relevant, da Beschreibung zu Klassen
- Autor: als Attribut relevant, da Beschreibung zu Klassen
- Regisseur: als Attribut relevant, da Beschreibung zu Klassen
- Interpret: als Attribut relevant, da Beschreibung zu Klassen

Abbildung 5 stellt die identifizierten Klassen des Beispielszenarios dar. Klassen werden immer in Form eines Rechtecks dargestellt. Der Name der Klasse wird in das Rechteck geschrieben.

Abbildung 5: Identifizierte Klassen



Quelle: erstellt im Auftrag der IU, 2013.

2.3 Attribute als Eigenschaften von Klassen

Wie bereits beschrieben, bestehen Klassen (und damit auch Objekte) aus Attributen und Methoden. Die **Attribute** (auch: Eigenschaft von Klassen, engl. property) sind statische Elemente von Klassen. In einem Attribut können konkrete Werte zu dem Objekt gespeichert werden. Attribute dienen ausschließlich zum Speichern von Werten, man kann sie als freie Speicherplätze innerhalb eines Objekts verstehen. Zusammengefasst beschreiben die Werte aller Attribute eines Objekts dessen **Zustand**. Objekte, die aus einer Klasse erzeugt wurden und deren Attribute exakt die gleichen Werte haben, sind identisch.

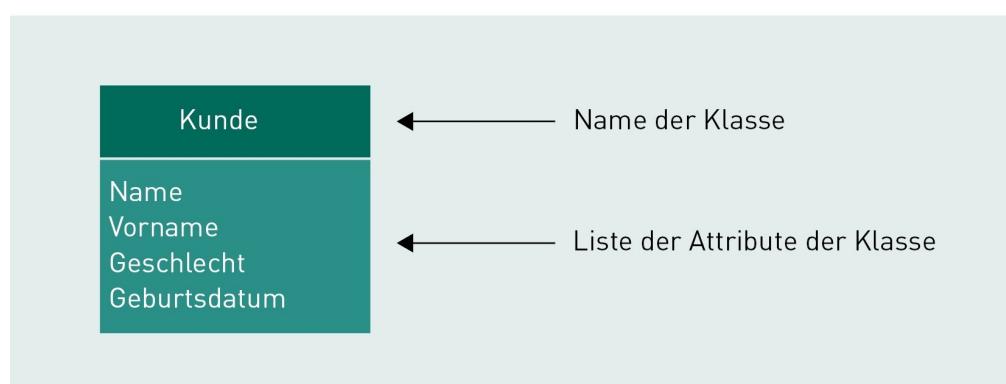
Attribute

Das sind statische Elemente von Klassen und werden zum Speichern von konkreten Werten verwendet.

Beispiel

Für den Online-Shop wurde eine Klasse „Kunde“ identifiziert. Zu jedem Kunden sollen der Name, der Vorname, das Geburtsdatum und das Geschlecht gespeichert werden. Daher werden Name, Vorname, Geburtsdatum und Geschlecht als Attribute der Klasse „Kunde“ modelliert.

Abbildung 6: Attribute der Klasse „Kunde“



Quelle: erstellt im Auftrag der IU, 2013.

Wie in Abbildung 6 dargestellt, werden Attribute unterhalb des Klassennamens in einem eigenen Rechteck notiert. Eine Klasse mit Attributen besteht also aus zwei zusammenhängenden Rechtecken: Im oberen Rechteck steht der Name der Klasse und im unteren die Liste der Attribute einer Klasse.

Bei der Modellierung von Attributen können u. a. die in Tabelle 3 beschriebenen Eigenschaften festgelegt werden. Von diesen Eigenschaften ist in einem Analysemodell mindestens der Name erforderlich. Alle weiteren Eigenschaften können bereits angegeben werden, müssen jedoch nicht. Die fehlenden Eigenschaften werden im Verlauf des SW-Entwicklungsprozesses festgelegt.

Tabelle 3: Eigenschaften von Attributen

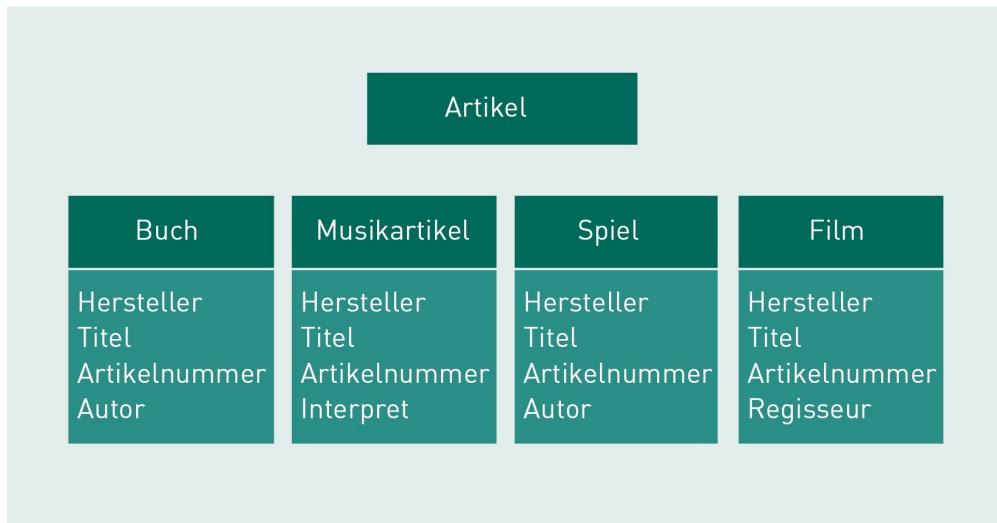
Eigenschaften eines Attributs	Beschreibung	Beispiel
Name	Name des Attributs	Vorname
Datentyp	Legt fest, wie die Werte zu dem Attribut aussehen, also ob es z. B. eine Zahl, eine Zeichenkette oder ein Datum ist	Date (Datum) String (Zeichenkette) Integer (Ganze Zahl)
Konstante (ja/nein)	Legt fest, ob sich der Wert des Attributs ändern darf oder nicht	Gerundete Kreiszahl Pi: 3,1415
Defaultwert	Legt den voreingestellten Wert des Attributes fest	2010-01-01

Quelle: erstellt im Auftrag der IU, 2013.

Beispiel

Nach der Identifikation möglicher Klassen in Abbildung 6 müssen nun die Attribute zu den Klassen identifiziert werden. Gute Kandidaten für Attribute sind Hauptwörter, die zur Beschreibung oder Detaillierung anderer Hauptwörter genutzt werden. In Abbildung 7 sind die im Beispieldaten Online-Shop identifizierten Attribute ergänzt:

Abbildung 7: Identifizierte Attribute zu den Klassen



Quelle: erstellt im Auftrag der IU, 2013.

2.4 Methoden als Funktionen von Klassen

Methoden (auch: Funktionen, Operationen) sind dynamische Elemente von Klassen. Sie enthalten Algorithmen, Anweisungen und Abarbeitungsvorschriften, mit denen Werte erstellt, berechnet, verändert und gelöscht werden können. Die Ergebnisse der Methoden können in Attribute der Klasse oder in neu erzeugte Objekte gespeichert werden. Mit Methoden wird das Verhalten von Objekten beschrieben, also was ein Objekt macht. So können mit Methoden Berechnungen von Werten und das Anwenden von Geschäftsregeln oder Vorschriften zum Erzeugen und Löschen von Objekten programmiert werden.

Bei der Modellierung von Methoden können die in Tabelle 4 beschriebenen Elemente festgelegt werden. In einem Analysemodell wird mindestens der Name benötigt. Detaillierte Informationen zu Rückgabewert, Parameter und der Funktion der Methode können auch im Laufe des Entwicklungsprozesses Stück für Stück ergänzt werden.

Methoden
Das sind dynamische Elemente von Klassen. Sie beschreiben das Verhalten von Klassen. In Methoden werden u. a. Vorschriften zur Berechnung, Änderung und Erzeugung von Attributen und Objekten definiert.

Tabelle 4: Eigenschaften von Methoden

Elemente von Methoden	Beschreibung	Beispiel
Name	Name der Methode	getName()
Parameter	Benötigte Objekte und Werte, die zur Abarbeitung der Methode erforderlich sind	(Date geburtsdatum) (String name, String vorname) (Integer zahl1, Integer zahl2)

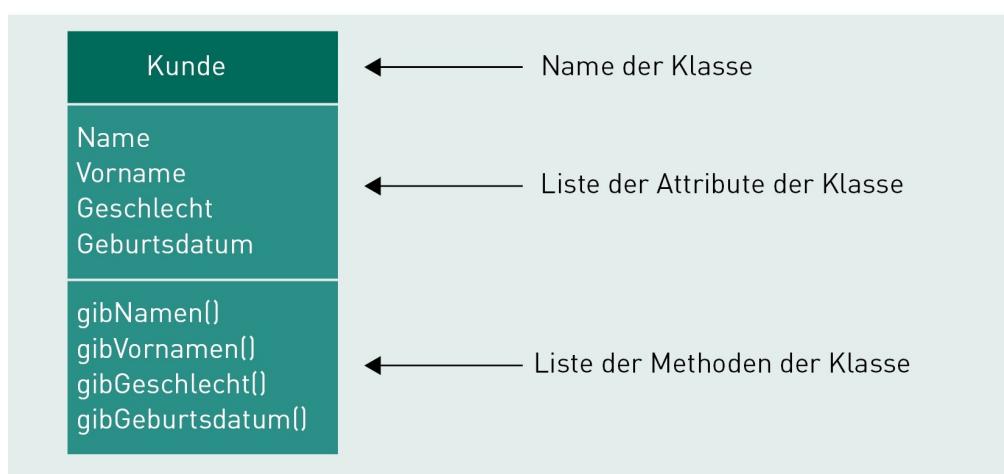
Elemente von Methoden	Beschreibung	Beispiel
Rückgabewert	Angabe des Datentyps des Objektes, in dem das Ergebnis der Methode gespeichert wird	Date (Datum) String (Zeichenkette) Integer (Ganze Zahl)

Quelle: erstellt im Auftrag der IU, 2013.

Beispiel

Zur oben bereits mit Attributen versehenen Klasse „Kunde“ sollen nun die Methoden festgelegt werden. Wichtig ist in jedem Fall, dass die in den Attributen gespeicherten Werte wieder ausgelesen werden können. Wie bereits beschrieben, darf von außen auf Attribute nur über Methoden zugegriffen werden und niemals direkt. Daher muss die Klasse „Kunde“ für jedes ihrer Attribute eine Methode bereitstellen, die den aktuellen Wert des Attributs zurückgibt.

Abbildung 8: Attribute und Methoden der Klasse „Kunde“



Quelle: erstellt im Auftrag der IU, 2013.

Wie in Abbildung 8 dargestellt, werden die Methoden einer Klasse in einem zusätzlichen Rechteck unterhalb der Attribute modelliert. Eine Klasse mit Methoden besteht demnach aus drei zusammenhängenden Rechtecken: Im oberen Rechteck steht der Name der Klasse, im mittleren die Liste der Attribute einer Klasse und im unteren die Liste der Methoden einer Klasse.

2.5 Beziehungen zwischen Klassen

Alle Elemente eines objektorientierten Systems werden in Form von Klassen programmiert. Ein Zusammenspiel verschiedener Klassen ist nur möglich, wenn während der Phasen des Entwicklungsprozesses **Beziehungen** (auch: Assoziationen) zwischen Klassen definiert werden. Durch die Festlegung von Beziehungen zwischen Klassen ist eine Kooperation zwischen Objekten überhaupt erst möglich.

Typische Beziehungen zwischen Klassen sind in Tabelle 5 erläutert:

Beziehungen
Sie beschreiben Abhängigkeiten und Zusammenhänge zwischen Klassen. Durch sie wird die Kooperation zwischen Objekten ermöglicht.

Tabelle 5: Typische Beziehungen zwischen Klassen

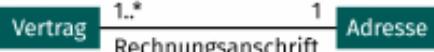
Beziehungstypen	Beschreibung	Beispiel
„hat/kennt“	Dieser Beziehungstyp drückt aus, dass eine Klasse eine andere Klasse „hat“ oder „kennt“.	Ein Versicherungsnehmer hat Kinder. Ein Vertrag hat Versicherungsbedingungen. Ein Kalender hat Monate. Ein Verkäufer kennt seine Kunden.
„besteht aus“	Dieser Beziehungstyp drückt aus, dass eine Klasse ein Bestandteil einer anderen Klasse ist. Er wird genutzt, wenn eine Klasse ein größeres Konstrukt ist, dessen Elemente nicht durch einfache Attribute beschrieben werden können.	Ein Auto besteht aus einem Motor, 4 Rädern, 3 Türen, 1 Getriebe und 2 Sitzen. Ein Haus besteht aus einem Dach, 23 Fenstern, 2 Türen, 6 Räumen und 1 Treppenhaus.
„ist ein“	Dieser Beziehungstyp drückt aus, dass eine Klasse A von der Art her eine Klasse B ist, aber eine spezifischere Bedeutung hat und sich ggf. um bestimmte Attribute und Methoden von Klasse B unterscheidet.	Ein Pkw ist ein Auto. Ein Lkw ist ein Auto. Ein Kunde ist eine Person. Ein Buch ist ein Artikel. Ein Igel ist ein Säugetier.

Quelle: erstellt im Auftrag der IU, 2013.

Um neben Klassen, Attributen und Methoden ebenfalls die Beziehungen grafisch darzustellen, werden Linien beziehungsweise Pfeile zwischen den entsprechenden Klassen modelliert. Die einfachste Art Beziehungen herzustellen, ist die Verbindung von zwei Klassen durch eine durchgezogene Linie. Eine Beziehung kann, wie in Tabelle 6 gezeigt, neben der durchgezogenen Linie um weitere Informationen ergänzt werden.

Tabelle 6: Darstellung von Beziehungen

Darstellung der Beziehung	Bedeutung
 Durchgezogene Linie, ohne Pfeilspitze und ohne Beschriftung	Vertrag und Adresse stehen in einer nicht näher beschriebenen Beziehung zueinander.

Darstellung der Beziehung	Bedeutung
 Durchgezogene Linie, Beschriftung der Linie mit einem Namen für die Beziehung	Vertrag und Adresse sind über eine benannte Assoziation verbunden; die beiden Klassen sind verbunden durch die Beziehung „Rechnungsanschrift“.
 Durchgezogene Linie mit einer Pfeilspitze, ggf. Benennung der Beziehung	Durch die Pfeilspitze wird eine Navigationsrichtung vorgegeben: vom Vertrag zur Adresse. Das bedeutet, dass der Vertrag eine Adresse kennt und man sich vom Vertrag zur Adresse durchhangeln kann. Jedoch nicht von der Adresse zum Vertrag: Ein Vertrag kennt seine Adresse, aber die Adresse weiß nichts von und über die Existenz des Vertrags.
 An den Enden der Beziehung werden Multiplizitäten modelliert und damit Aussagen über die Anzahl der assoziierten Objekte gemacht.	Ein Vertrag hat genau eine Rechnungsanschrift, eine Adresse kann jedoch die Rechnungsanschrift zu mindestens 1 aber maximal beliebig vielen Verträgen sein. (Detaillierte Erklärung dazu siehe unten.)

Quelle: erstellt im Auftrag der IU, 2013.

Multiplizitäten

Das sind Mengenangaben bei Beziehungen zwischen Klassen. Mit Multiplizitäten kann festgelegt werden, wie viele Objekte einer Klassen mit wie vielen Objekten einer anderen Klasse in Beziehung stehen können.

Zu Beziehungen können mit **Multiplizitäten** (auch: Kardinalitäten) Mengenangaben festgelegt werden. Diese Angaben werden jeweils an das Ende und die Spitze einer Beziehung notiert: Links von „..“ steht die Untergrenze und rechts von „..“ die Obergrenze der Mengenangaben. Tabelle 7 gibt einen Überblick über mögliche Mengenangaben:

Tabelle 7: Multiplizitäten in Beziehungen

Notation	Erklärung	Beispiel
0..1	Optionale Assoziation	 Zu einem Auto gehören 0..1 Anhänger. Zu einem Anhänger gehören 0..1 Autos.

Notation	Erklärung	Beispiel
1	Obligatorische Assoziation	Zu einem Auto gehört genau 1 Fahrer.  Ein Fahrer gehört zu genau 1 Auto.
0..*	Optional beliebig	Ein Student kann 0..* Kurse belegen.  Ein Kurs kann von 1..* Studenten belegt sein.
1..*	Beliebig, aber mindestens 1	Ein Tutor kann 1..* Kurse machen.  Ein Kurs wird von genau 1 Tutor durchgeführt.
n..m	Mindestens n und maximal m	Ein Auto hat mindestens 3 und maximal 4 Türen.  Eine Tür gehört zu genau 1 Auto.
	Keine Angabe entspricht 1 (sollte aber vermieden werden, um Missverständnisse zu vermeiden)	Ein Auto hat genau 1 Motor.  Ein Motor gehört zu genau 1 Auto.

Quelle: erstellt im Auftrag der IU, 2013.

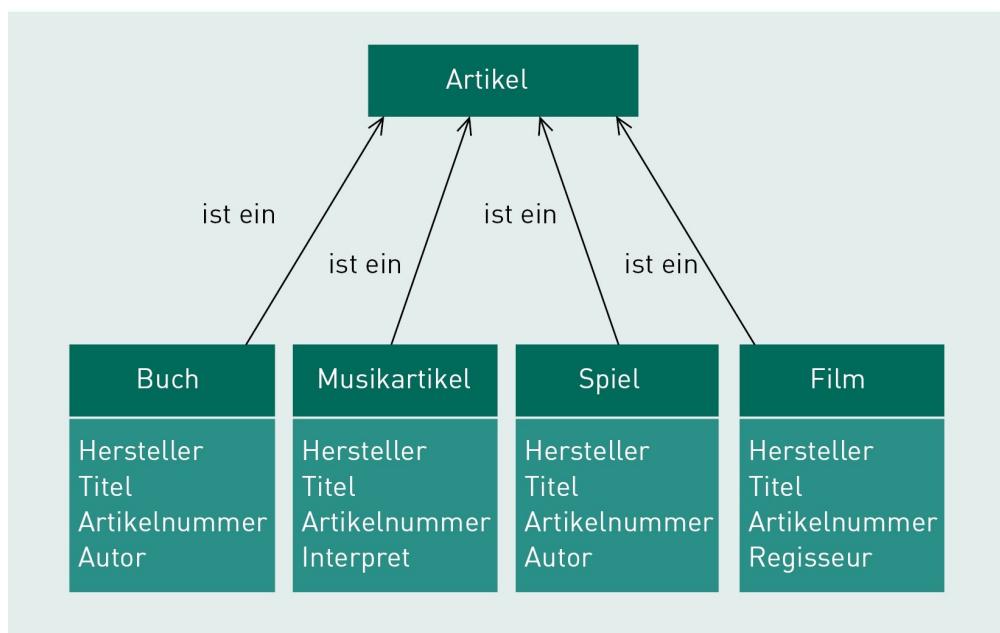
Beispiel

Um mögliche Beziehungen für das Beispielszenario „Online-Shop“ zu identifizieren und zu ergänzen, müssen wir uns erneut den ursprünglichen Text vornehmen:

Die im Online-Shop von Frau Lange verkauften Artikel sollen Medien aller Art sein, also insbesondere Bücher, Musikartikel, Filme und Spiele. Jeder Artikel hat einen Hersteller, einen Titel und eine Artikelnummer. Bücher und Spiele haben einen Autor, Filme einen Regisseur und Musikartikel einen Interpreten.

Abbildung 9 zeigt eine entsprechende Darstellung: Außer den „ist ein“-Beziehungen der Klasse „Artikel“ zu den Klassen „Buch“, „Musikartikel“, „Spiel“ und „Film“ sind vorerst keine weiteren Beziehungen erforderlich.

Abbildung 9: Beispielszenario ergänzt um Beziehungen



Quelle: erstellt im Auftrag der IU, 2013.

2.6 Unified Modeling Language (UML)

Wie bereits in Lernzyklus 1.3 beschrieben, zieht sich die Anwendung der Konzepte (Klasse, Objekt, Methode, Attribut) durch alle Phasen eines objektorientierten Entwicklungsprozesses. Um die Ergebnisse der Phasen OOA und OOD geeignet zu dokumentieren, werden die identifizierten Klassen und deren Beziehungen in einer standardisierten Modellierungssprache dokumentiert.

Unified Modeling Language (UML)

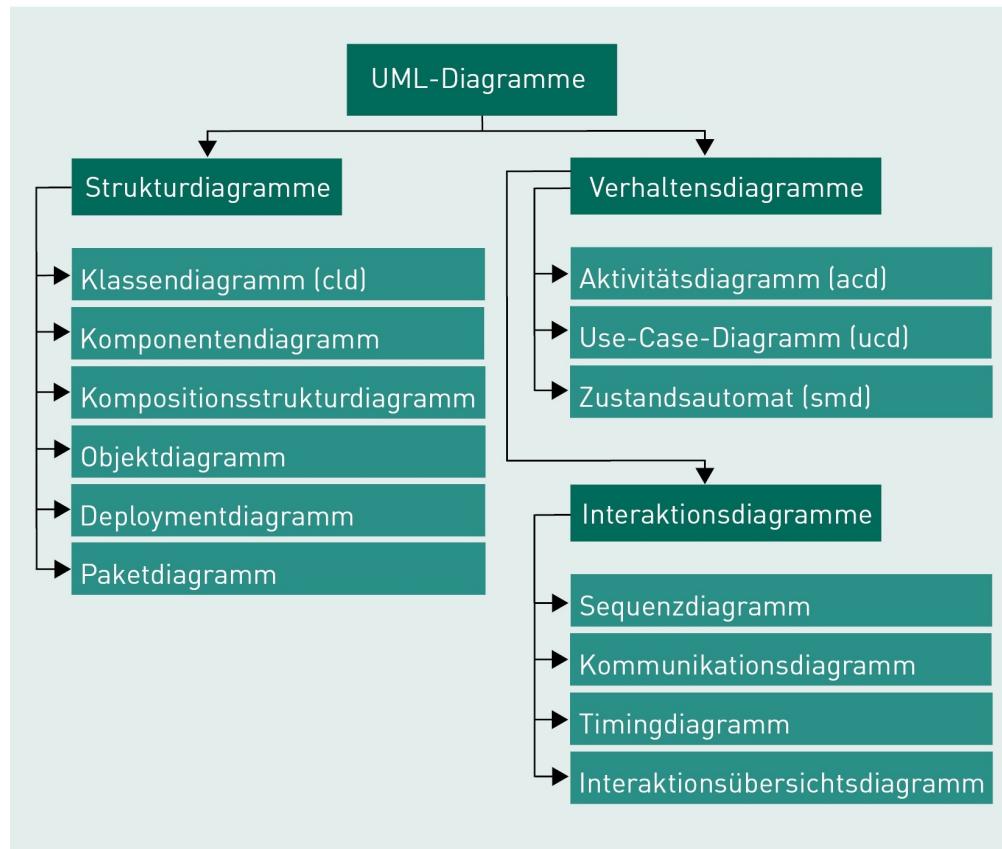
Das ist eine universelle Modellierungssprache und wird als De-Facto-Standard weltweit zur Modellierung von IT-Systemen eingesetzt.

Die **Unified Modeling Language (kurz: UML)** ist eine grafische Modellierungssprache, die etwa zeitgleich mit der Objektorientierung in den frühen 1990er Jahren entwickelt wurde. Die UML umfasst mittlerweile 13 verschiedene Diagrammtypen, mit denen verschiedene Aspekte eines Systems modelliert werden können. Abbildung 10 gibt einen Überblick über die UML-Diagramme. Grundsätzlich lassen sich Diagramme zur Modellierung von Struktur und zur Modellierung von Verhalten unterscheiden. Mit **Strukturdiagrammen** werden Aufbau, Elemente und Zusammensetzung sowie Schnittstellen von Systemen dargestellt.

Vereinfacht gesagt werden Strukturdiagramme eingesetzt, um zu modellieren, *woraus* ein System besteht. Verhaltensdiagramme hingegen kommen zum Einsatz, wenn dargestellt werden soll, *was* in einem System abläuft.

Abbildung 10: Übersicht über UML-Diagrammtypen

Strukturdiagramme
Mit UML-Strukturdiagrammen werden Aufbau, Elemente, Zusammensetzung und Schnittstellen von Systemen modelliert (z. B. Klassendiagramm).



Quelle: erstellt im Auftrag der IU, 2013.

Klassen mit ihren Attributen, Methoden und Beziehungen werden mit dem UML **Klassendiagramm** modelliert. Alle Beispiele in Lektion 2 sind gültige UML Klassendiagramme. Das UML Klassendiagramm ist weltweit eine der am häufigsten genutzten Dokumentationsform bei der objektorientierten Systementwicklung. Alle anderen Strukturdiagramme der UML bauen mehr oder weniger auf den Modellierungskonzepten des Klassendiagramms auf.

Objektdiagramme sind eine Spezialform der Klassendiagramme. Mit ihnen kann man ganz konkrete Ausprägungen von Klassen darstellen. Dazu werden Objekte modelliert, deren Attribute Werte enthalten. Abbildung 11 stellt die Klasse „Kunde“ zwei konkreten Objekten der Klasse „Kunde“ gegenüber. Objekte unterscheiden sich von Klassen dahingehend, dass ...

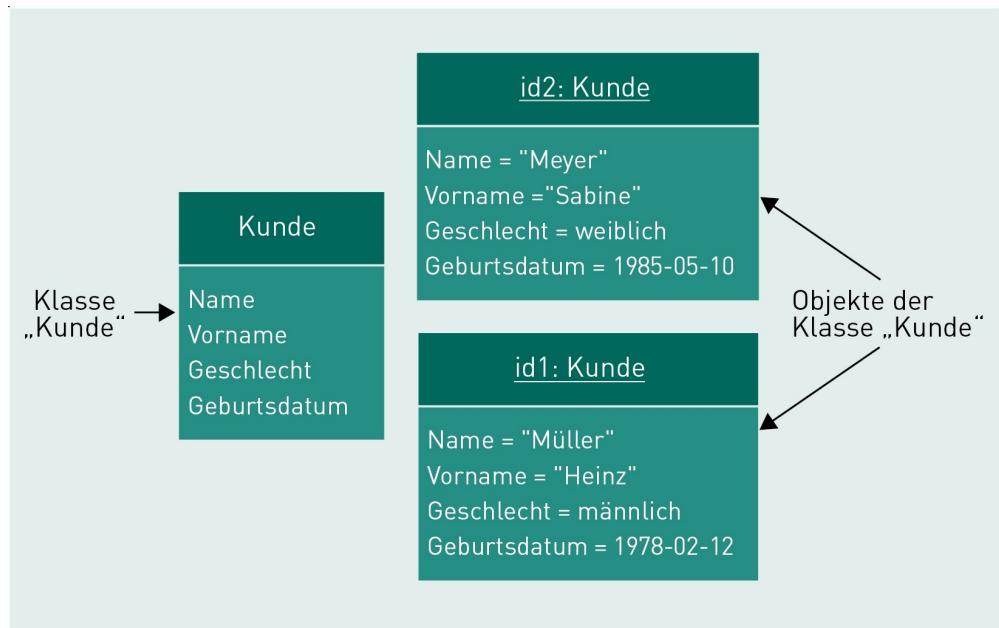
Klassendiagramm
Das ist ein Strukturdiagramm der UML. Es wird zur Modellierung von Klassen inklusive deren Attribute, Methoden und Beziehungen eingesetzt.

Objektdiagramm
Das ist ein Strukturdiagramm der UML. Es wird zur Modellierung von konkreten Objekten inklusive konkreter Attributwerte als Instanzen von Klassen eingesetzt.

- ... jedes Objekt mit einer eindeutigen ID identifizierbar ist (hier „id1“ und „id2“), notiert links vom Klassennamen gefolgt von einem Doppelpunkt „::“.
- ... zu jedem Attribut eines Objekts ein konkreter Wert dargestellt ist.

Alle Objekte einer Klasse haben dabei die gleichen Attribute, nur deren Werte können sich unterscheiden.

Abbildung 11: Objekte und Klassen



Quelle: erstellt im Auftrag der IU, 2013.

Mit den Modellierungskonzepten Klasse, Objekt, Attribut, Methode und Beziehung haben wir bis jetzt die grundlegenden Notationselemente erarbeitet. Alle weiteren Notationen, wie die Vererbung oder Interfaces, bauen auf diesen Grundelementen auf. Wir führen sie später an den Stellen ein, an denen wir sie für die objektorientierte Programmierung benötigen.



ZUSAMMENFASSUNG

Vor der Programmierung wird in den Phasen objektorientierte Analyse (OOA) und objektorientiertes Design (OOD) sowohl das Problem als auch das Design des objektorientierten Systems modelliert. Dafür werden Klassen mit ihren Attributen und Methoden sowie Beziehungen zwischen Klassen mit UML Klassendiagrammen modelliert.

Durch die Analyse der Problemerstellung nach relevanten Substantiven werden mögliche Kandidaten für Klassen identifiziert. Anschließend werden diese Klassen um Attribute und Methoden erweitert, je nachdem wie es die Problemstellung erfordert.

Nach der Identifikation von Attributen können diese durch einen Namen, einen Datentyp, die Information, ob es sich um eine Konstante handelt, und ggf. einen Defaultwert genauer charakterisiert werden. Methoden hingegen werden durch ihren Namen, benötigte Parameter und ihren Rückgabewert beschrieben.

Beziehungen zwischen Klassen können nach ihrem Typ unterschieden werden, neben der „hat/kennt“-Beziehung gibt es auch die „besteht aus“- und die „ist ein“-Beziehung. Einzelne Beziehungen können durch die Beschriftung mit Rollen, Pfeilspitzen und Kardinalitäten verfeinert werden.

Die UML bietet geeignete grafische Modellierungskonzepte, um Analysemodelle und das Design von objektorientierten Systemen standardisiert darzustellen. Insbesondere das UML Klassendiagramm und das UML Objektdiagramm sind für die Modellierung von Strukturen geeignet. Darüber hinaus umfasst die UML verschiedene Verhaltensdiagramme zur Modellierung von Abläufen eines Systems.

LEKTION 3

PROGRAMMIEREN VON KLASSEN IN JAVA

LERNZIELE

Nach der Bearbeitung dieser Lektion werden Sie wissen, ...

- welche Elemente für das Ausführen und das Programmieren von Java-Anwendungen benötigt werden.
- aus welchen Grundelementen eine Java-Klasse besteht und wie man sie in Java programmiert.
- wie Attribute von Klassen in Java programmiert werden.
- wie Methoden von Klassen in Java programmiert werden.
- wie festgelegt wird, was ein Java-Programm beim Starten ausführen soll.

3. PROGRAMMIEREN VON KLASSEN IN JAVA

Aus der Praxis

Nachdem Herr Koch sich ein objektorientiertes Modell zum Aufbau und zur Struktur des Online-Shops erstellt hat, möchte er gerne mit den ersten Schritten in Richtung objektorientierte Programmierung anfangen. Nachdem er nun verstanden hat, dass objektorientierte Systeme aus einer Menge kooperierender Objekte bestehen, will er wissen, wie er Klassen und Objekte mit der Programmiersprache Java erstellen kann und was er dafür vorbereiten muss.

3.1 Einführung in die Programmiersprache Java

Programmierung

Dies ist eine Tätigkeit mit dem Ziel Programmcode zu erstellen, der sich nach dem Kompilieren auf einem Rechner als Programm oder Bestandteil eines Programmes ausführen lässt.

Compiler

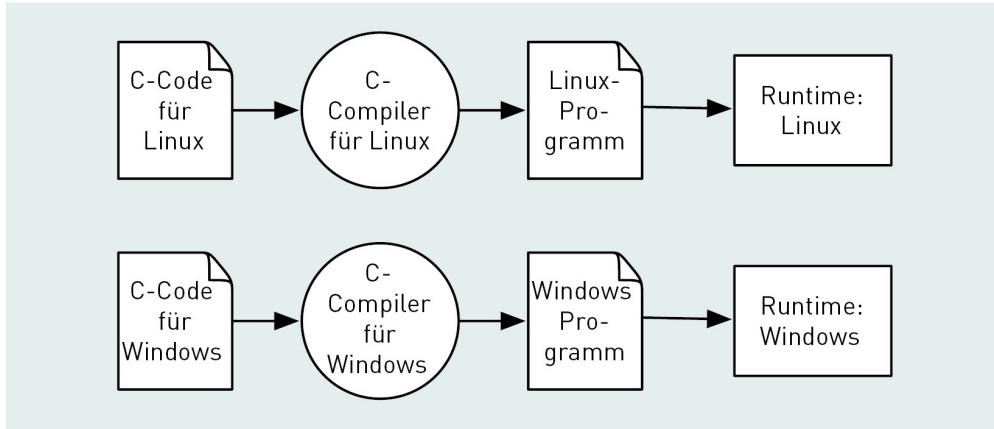
Das ist eine Software, die Programmcode in eine Form übersetzt, die von einem Rechner ausgeführt werden kann.

Ganz grundsätzlich werden bei der **Programmierung** (auch: Implementierung) drei verschiedene Aktivitäten durchgeführt:

1. Programmieren, d. h. Produzieren von Programmcode (auch: Quellcode, Sourcecode, Quelltext, Programmtext),
2. Kompilieren, d. h. automatisches Übersetzen des produzierten Programmcodes in eine Form, die von einem Rechner ausgeführt werden kann, sowie
3. Ausführen des Programms, d. h. Starten und Ausführen des erzeugten und kompilierten Programmcodes, um zu sehen, ob sich das System wie erwartet verhält.

Kann das Schreiben des Programmcodes noch mithilfe eines ganz einfachen Texteditors geschehen, so wird spätestens beim Kompilieren eine Software benötigt, die den vom Entwickler geschriebenen Programmtext in ausführbaren Maschinencode übersetzt (auch: kompiliert). Diese Übersetzungssoftware heißt „Compiler“. Es gibt **Compiler** für jede Programmiersprache. Nachdem der Programmtext durch den Compiler kompiliert wurde, kann er in einer Ausführungsumgebung (auch: Runtime) gestartet und ausgeführt werden. Für viele Programmiersprachen stellt das Betriebssystem eines PCs (z. B. Windows, Linux, MacOS) die Ausführungsumgebung bereit. Da sich die Funktionen und Eigenschaften von Betriebssystemen unterscheiden, müssen Programme für jedes Betriebssystem neu programmiert beziehungsweise kompiliert werden. Abbildung 12 zeigt exemplarisch alle benötigten Elemente für die Entwicklung mit der Programmiersprache C.

Abbildung 12: Entwicklung mit der Programmiersprache C

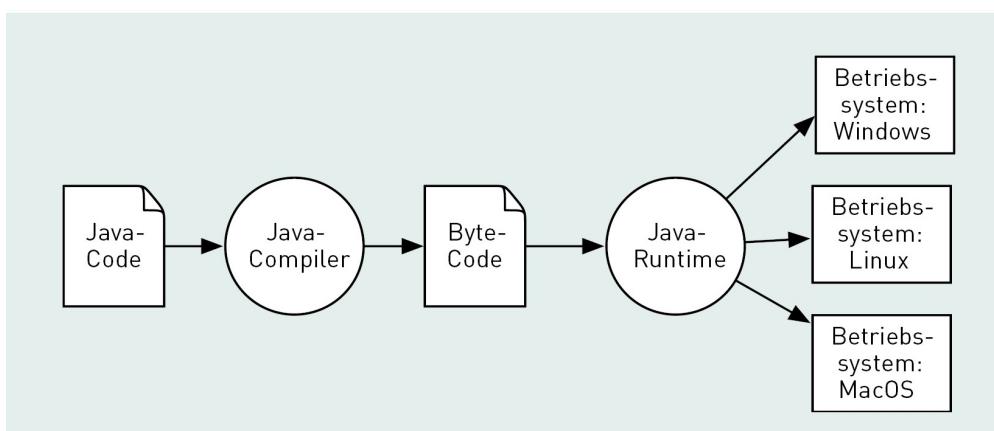


Quelle: erstellt im Auftrag der IU, 2013.

Bei der Programmiersprache Java handelt sich um eine plattform-unabhängige Programmiersprache. Ein in Java geschriebenes Programm muss nicht für jedes Betriebssystem neu angepasst werden. Von anderen Programmiersprachen unterscheidet sich Java dadurch, dass auf den Geräten, auf denen ein Java-Programm ausgeführt werden soll, vorher eine „Java-Laufzeitumgebung“ (auch: Java Runtime Environment, kurz: JRE) installiert werden muss. Der Java-Compiler übersetzt den programmierten Quelltext in einen sogenannten „Bytecode“. Dieser „**Bytecode**“ wird dann von der Java-Laufzeitumgebung geladen, gestartet und ausgeführt. Weil für nahezu jedes Betriebssystem eine entsprechende Java-Laufzeitumgebung verfügbar ist, kann ein Java-Programm sowohl auf einem Linux-, Windows- oder Mac-Rechner ausgeführt werden. Abbildung 13 zeigt eine Übersicht über benötigte Elemente bei der Entwicklung mit Java. Im Unterschied zu Abbildung 12 wird hier eine zusätzliche Runtime benötigt, um die Ausführung auf verschiedenen Plattformen zu ermöglichen.

Bytecode
Das ist eine Form eines Java-Programms nachdem es kompiliert wurde. Er kann von einer Java-Laufzeitumgebung (JRE) als Programm ausgeführt werden.

Abbildung 13: Entwicklung mit der Programmiersprache Java



Quelle: erstellt im Auftrag der IU, 2013.

Java Virtual Machine (JVM)

Dies ist eine betriebssystem-spezifische Software, die Java-Programme in Form von Bytecode auf einem Rechner als Programm ausführen kann.

Um als Nutzer ein Java-Programm ausführen zu können, wird die Java-Runtime (JRE) benötigt. Die Java-Runtime besteht aus der **Java Virtual Machine (kurz: JVM)** und der Java Klassenbibliothek. Die JVM ist die Software, die den Bytecode interpretiert sowie im eingesetzten Betriebssystem startet und ausführt. Die Java-Klassenbibliothek stellt bereits in der Programmiersprache Java vorhandene Funktionen zur Verfügung, sodass häufig eingesetzte Datenstrukturen (z. B. Listen oder Zeichenketten) vom Programmierer zwar wiederverwendet werden können, jedoch nicht jedes Mal neu implementiert werden müssen.

Java Software Development Kit (SDK)

Das ist das Grundwerkzeug des Java-Entwicklers. Es enthält einen Compiler, der Bytecode erzeugt, und eine JVM zum Ausführen von Java-Programmen.



TIPP

Für die Entwicklung werden sogenannte Entwicklungsumgebungen (kurz: IDE) eingesetzt. Für Java gibt es beispielsweise Eclipse, NetBeans, IntelliJ oder BlueJ als kostenfrei verfügbare IDEs – aber auch noch einige andere.

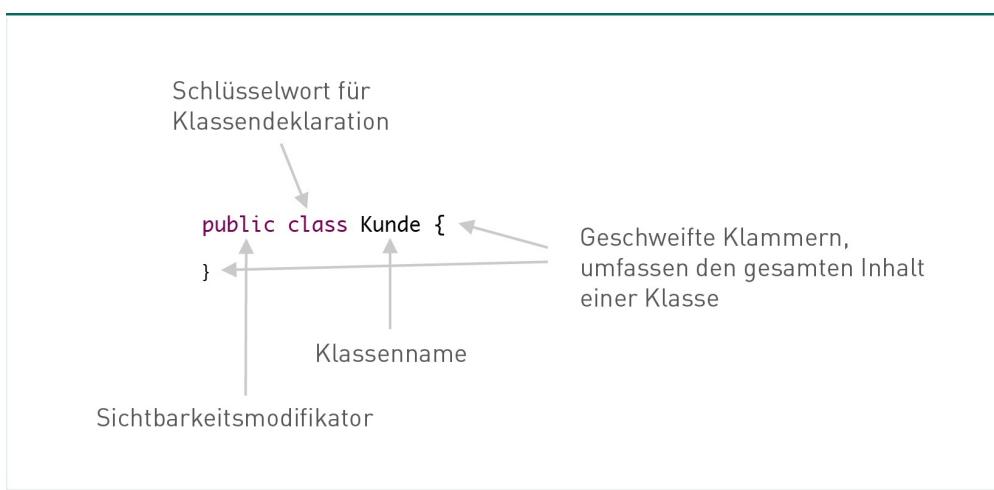
Zum Nachstellen der Beispiele und für praktische Übungen wird Ihnen Ihr Tutor entsprechende Empfehlungen mit Download-Links und Anleitung zur Verfügung stellen.

3.2 Grundelemente einer Klasse in Java

Eine Klasse in der Programmiersprache Java hat einen eindeutigen Namen und kann Attribute und Methoden enthalten. Der Programmcode einer Klasse wird in Java in einer Textdatei mit der Dateiendung `*.java` gespeichert. Für jede Klasse wird eine eigene Datei angelegt. Der Java-Compiler legt für jede Klasse (also für jede `.java`-Datei) eine Klasse mit der Endung `*.class` an, in welcher der kompilierte Bytecode einer Klasse gespeichert wird.

Abbildung 14 zeigt das Grundgerüst der Java-Klasse „Kunde“ aus Abbildung 6, hier allerdings ohne Attribute. Um eine Klasse zu erstellen, werden mindestens die folgenden vier dargestellten Elemente benötigt: der Sichtbarkeitsmodifikator, das Schlüsselwort `class`, der Name der Klasse und ein Paar geschweifte Klammern.

Abbildung 14: Gerüst der Klasse „Kunde“ aus Abbildung 6



Quelle: erstellt im Auftrag der IU, 2013.

Grundsätzlich kann die so dargestellte Klasse „Kunde“ verwendet werden – der Java-Compiler würde keinen Fehler bei der Übersetzung in den Bytecode melden. In der folgenden Tabelle 8 sind die Elemente des Grundgerüsts detailliert beschrieben:

Tabelle 8: Grundelemente einer Klasse in Java

Element einer Klasse	Beschreibung	Beispiel
Sichtbarkeitsmodifikator	Legt die Sichtbarkeit der Klasse für andere Klassen fest (Details siehe Lernzyklus 4.5)	public
Schlüsselwort für die Klassendeklaration	Zeigt dem Java-Compiler an, dass im Folgenden eine Java-Klasse programmiert ist	class
Klassenname	Legt den Namen für die Klasse in Java fest und wird als Dateiname verwendet. Die Klasse „Kunde“ wird in einer Textdatei „Kunde.java“ programmiert	Kunde
Geschweifte Klammern	Markieren den Inhalt einer Klasse (Attribute und Methoden); alles was innerhalb der Klammern steht, gehört zu einer Klasse	{ ... }

Quelle: erstellt im Auftrag der IU, 2013.

Für den Namen der Klasse müssen dabei folgende Vorgaben eingehalten werden:

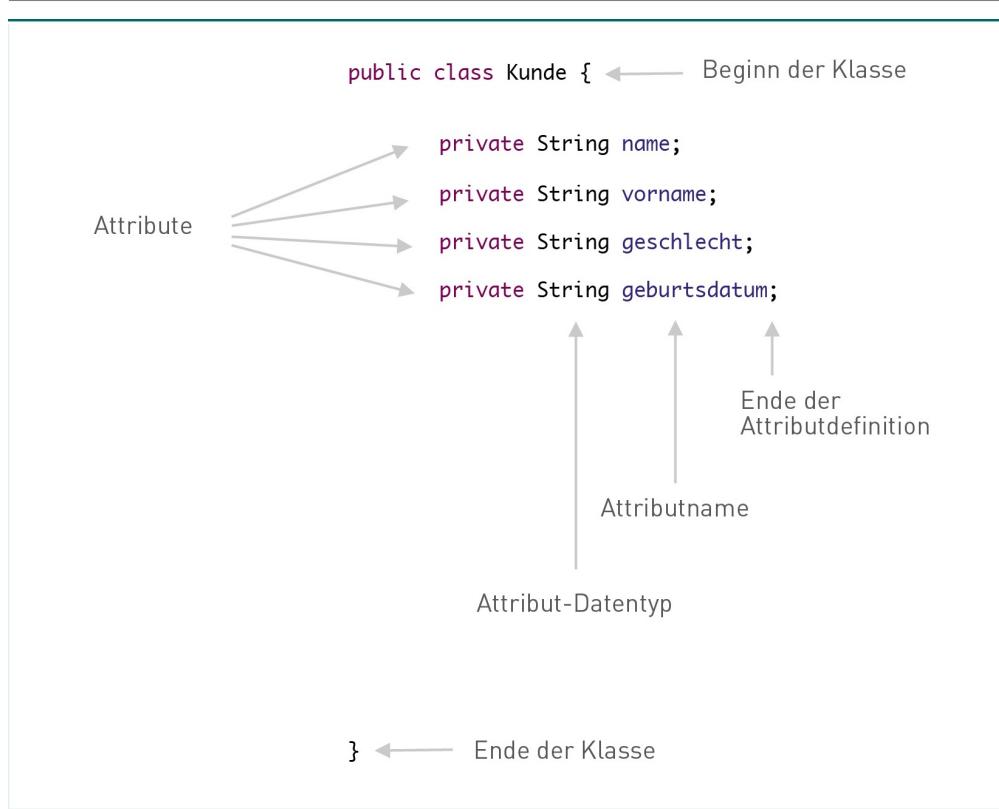
- Beginnt mit Großbuchstaben
- Besteht aus Unicode-Zeichen (mit Einschränkungen, z. B. keine Leerzeichen, keine Umlaute)

- Kann theoretisch beliebig lang sein (begrenzt nur durch die maximale Länge eines Dateinamens)
- Darf kein Schlüsselwort sein (z. B. `class` oder `public`)
- Umfasst der Klassename mehrere Wörter, werden diese ohne Trennzeichen zusammen geschrieben (z. B. `WortWort`, `NameDerKlasse`)

3.3 Attribute in Java

Nach dem Programmieren des ersten Grundgerüsts der Klasse „Kunde“ sollen nun die benötigten Attribute ergänzt werden. In Abbildung 15 ist die um Attribute erweiterte Klasse „Kunde“ aus Abbildung 6 dargestellt. Ein Attribut in Java wird dabei mindestens mit folgenden Elementen beschrieben: dem Sichtbarkeitsmodifikator (Details siehe Lernzyklus 4.5), dem Datentyp und dem Attributnamen. Der in Abbildung 15 verwendete Datentyp `String` wird für Zeichenketten verwendet (Details siehe in Lernzyklus 4.1).

Abbildung 15: Klasse „Kunde“ erweitert um Attribute



Quelle: erstellt im Auftrag der IU, 2013.

Darüber hinaus können bereits bei der Programmierung Werte für Attribute festgelegt werden (auch: **Defaultwerte**). Folgendes Beispiel zeigt zwei Attribute für die im Quellcode jeweils ein Defaultwert festgelegt wurde:

Code

```
private boolean istPremiumKunde = false;
private int anzahlDerEinkaeufe = 0;
```

In der folgenden Tabelle 9 sind die Elemente des Grundgerüsts detailliert beschrieben:

Defaultwert

Das ist ein Wert, der einem Attribut automatisch bei der Erzeugung des Objektes zugewiesen wird.

Tabelle 9: Grundelemente eines Attributs in Java

Element eines Attributes	Beschreibung	Beispiel
Sichtbarkeitsmodifikator	Legt die Sichtbarkeit des Attributs für andere Klassen fest (Details siehe Lernzyklus 4.5)	private
Datentyp des Attributs	Legt den Datentyp des Attributs fest und bestimmt damit Anzahl und Art der Werte, die in dem Attribut gespeichert werden können. Ein Datentyp ist entweder ein primitiver Datentyp (Details siehe Lernzyklus 4.1) oder eine Klasse, ein sogenannter Referenzdatentyp (z. B. String oder Kunde).	String Kunde
Attributname	Legt den Namen für das Attribut der Klasse fest; jeder Name darf innerhalb einer Klasse nur 1x vergeben werden.	name vorname geschlecht
Defaultwert	Legt den initialen Wert des Attributs fest; dieser Wert wird dem Attribut bei der Erzeugung eines Objektes der Klasse zugewiesen.	= 0 = 1 = false
Semikolon	Markiert das Ende der Attributdeklaration.	;

Quelle: erstellt im Auftrag der IU, 2013.

Für den Namen eines Attributs einer Klasse müssen folgende Vorgaben eingehalten werden:

- Beginnt mit einem Kleinbuchstaben
- Besteht aus Unicode-Zeichen (mit Einschränkungen, z. B. keine Leerzeichen, keine Umlaute)
- Kann theoretisch beliebig lang sein
- Darf kein Schlüsselwort sein (z. B. public oder class)
- Groß- und Kleinschreibung wird beachtet (d. h. name ist ein anderes Attribut als nAme)
- Umfasst der Attributname mehrere Wörter, werden diese ohne Trennzeichen zusammen geschrieben (z. B. attributAttribut, nameDesKunden)

3.4 Methoden in Java

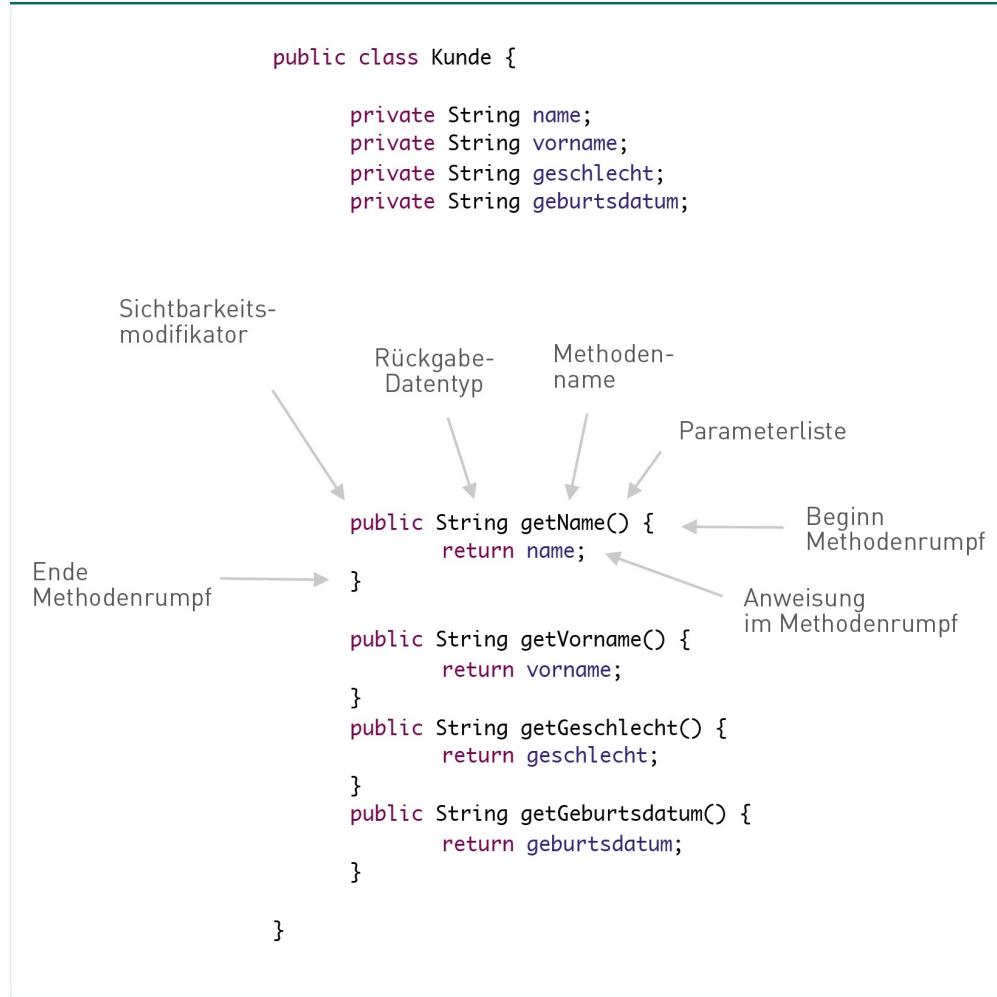
Mit den Kenntnissen aus den Lernzyklen 3.2 und 3.3 sind wir in der Lage, einfache Klassen mit Attributen zu programmieren. Damit können wir bereits Daten in Form von Attributwerten in Objekten speichern. Als nächstes sollen nun Methoden implementiert werden, die als dynamische Elemente von Klassen die Werte von Attributen erstellen, berechnen, verändern und löschen können.

Wie in Lernzyklus 1.2 bereits erwähnt, erfolgt der Zugriff auf Attribute eines Objektes durch andere Objekte nur über Methoden, aber niemals über einen direkten Zugriff auf Attribute einer anderen Klasse (**Prinzip der Datenkapselung**). Daher werden Methoden auch benötigt, um Werte von Attributen auszulesen und sie anderen Klassen zur Verfügung zu stellen.

Prinzip der Datenkapselung
Dabei darf ein Objekt auf Attribute eines anderen Objektes niemals direkt zugreifen. Das Lesen oder Ändern von Attributen sollte nur durch Methoden möglich sein.

In Abbildung 16 ist eine vollständig implementierte Klasse „Kunde“ dargestellt, wobei die Elemente einer Methode anhand der Methode `getName()` erläutert werden. Das passende UML Klassendiagramm zu dieser Implementierung zeigt Abbildung 8.

Abbildung 16: Methoden der Klasse „Kunde“



Quelle: erstellt im Auftrag der IU, 2013.

Die folgende Tabelle 10 erläutert die Elemente, die in Java zur Implementierung einer Methode mindestens benötigt werden:

Tabelle 10: Grundelemente einer Methode in Java

Element einer Methode	Beschreibung	Beispiel
Sichtbarkeits-modifikator	Legt die Sichtbarkeit der Methode für andere Klassen fest (Details siehe Lernzyklus 4.5).	<code>public</code>

Element einer Methode	Beschreibung	Beispiel
Rückgabe-Datentyp der Methode	Angabe des Datentyps des Objektes, in dem das Ergebnis der Methode nach Abarbeitung des Methodenrumpfes ausgegeben wird. Dabei werden ein primitiver Datentyp (Details siehe Lernzyklus 4.1) oder eine Klasse (z. B. String oder Kunde) angegeben. Für den Fall, dass die Methode kein Ergebnis ausgibt, wird void als Rückgabe-Datentyp festgelegt.	String Kunde void
Methodenname	Legt den Namen für die Methode fest; ein Methodenname darf innerhalb einer Klasse nur dann mehrfach vergeben werden, wenn sich Anzahl bzw. Datentyp der Parameter unterscheiden.	getName() getVorname()
Parameterliste	Liste benötigter Objekte und deren Datentypen, die zur Abarbeitung der Methode erforderlich sind; werden keine Parameter angegeben bleibt die Liste leer.	(Date geburtsdatum) (String name, String vorname) (Integer zahl1, Integer zahl2) ()
Methodenrumpf	Enthält die konkreten Anweisungen, was beim Aufruf der Methode in welcher Reihenfolge getan wird. Jede Anweisung wird mit einem Semikolon „;“ beendet, daher ist in Methodenrumpfen das „;“ oft das letzte Zeichen einer Programmzeile. Die Anweisungen innerhalb eines Methodenrumpfes werden der Reihe nach von oben nach unten abgearbeitet. Wird für die Methode ein Rückgabe-Datentyp angegeben, beginnt die letzte Anweisung des Methodenrumpfes mit dem Schlüsselwort return.	ergebnis = zahl1 + zahl2; return name;

Quelle: erstellt im Auftrag der IU, 2013.

Für den Namen der Methode müssen folgende Vorgaben eingehalten werden:

- Beginnt mit einem Kleinbuchstaben
- Besteht aus Unicode-Zeichen (mit Einschränkungen, z. B. keine Leerzeichen, keine Umlaute)
- Kann theoretisch beliebig lang sein
- Darf kein Schlüsselwort sein (z. B. public oder class)
- Groß- und Kleinschreibung wird beachtet (d. h. `methode()` ist eine andere Methode als `mEthode()`)
- Umfasst der Methodenname mehrere Wörter, werden diese ohne Trennzeichen zusammen geschrieben (`nameDerMethode()`, `berechneSumme()`, `gibtNamenZurueck()`)

Signatur

Sie identifiziert eine Methode eindeutig. Sie besteht aus dem Namen der Methode und der Parameterliste.

Ein weiteres Konzept für Methoden in Java ist die **Signatur**. Anhand ihrer Signatur können Methoden eindeutig identifiziert werden. Die Signatur einer Methode besteht aus dem Namen der Methode und der Parameterliste. Der Rückgabe-Datentyp ist nicht Teil der Signatur. Jede Signatur darf innerhalb einer Klasse nur einmal vorkommen, denn nur so kann die JRE zur Laufzeit des Programmes feststellen, welche Methode aufgerufen und abgearbeitet werden soll. Anders ausgedrückt: Der Name einer Methode kann innerhalb einer Klasse mehrfach vergeben werden, solange die Parameterliste sich unterscheidet.

Codebeispiel 1: Hier unterscheiden sich die Namen und damit auch die Signaturen:

Code

```
public void anmelden () {...}  
public void abmelden () {...}
```

Codebeispiel 2: Hier unterscheiden sich die Parameterlisten und damit auch die Signaturen:

Code

```
public int zumWarenkorbHinzufuegen (Artikel artikel) {...}  
public int zumWarenkorbHinzufuegen (Artikel artikel, int anzahl) {...}
```

Codebeispiel 3: Hier unterscheiden sich nur die Rückgabe-Datentypen, jedoch weder die Methodennamen noch die Parameterlisten. Damit sind Signaturen identisch:

Code

```
public boolean bezahlen (boolean pruefung) {...}  
public void bezahlen (boolean pruefung) {...}
```

Um auf Attribute einer Klasse zuzugreifen, können beliebige Methodennamen vergeben werden. Ganz grundsätzlich sollten die Namen jedoch so gewählt werden, dass leicht zu erkennen ist, was die Methode tatsächlich tut. Für das Schreiben und Auslesen von Attributnen werden sogenannte **Getter- und Setter-Methoden** eingesetzt. Eine Getter-Methode („get“ = „holen“) liefert den Wert eines Attributs zurück. Eine Setter-Methode („set“ = „setzen“) ändert den Wert eines Attributes auf den Wert, der als Parameter der Setter-Methode übergeben wird.

Beispiel

Eine vollständig mit Getter- und Setter-Methoden modellierte Klasse „Kunde“ ist in Abbildung 17 dargestellt. Somit ist sowohl der Zugriff auf die gespeicherten Werte der Attribute möglich (über die Getter-Methoden), als auch das Ändern der Werte (über die Setter-Methoden).

Getter- und Setter-

Methoden

Das sind spezielle Methoden, die zum Lesen und Ändern von Attributen eingesetzt werden. Sie dienen zum Umsetzen des Prinzips der Datenkapselung.

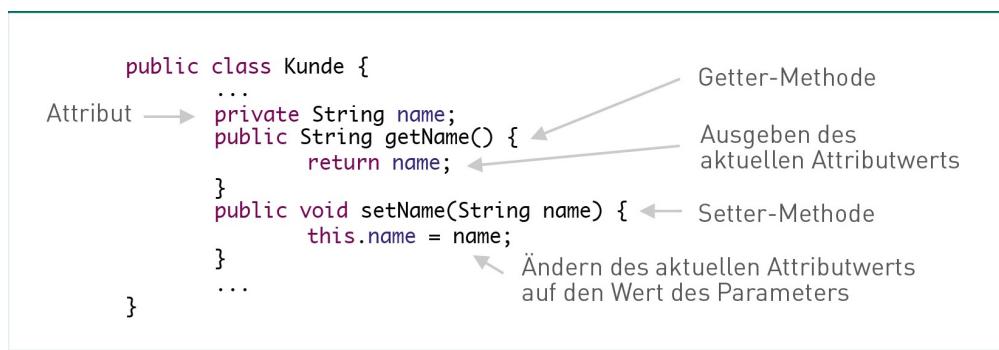
Abbildung 17: Getter- und Setter-Methoden für die Klasse „Kunde“



Quelle: erstellt im Auftrag der IU, 2013.

Eine entsprechende Implementierung dazu am Beispiel des Attributs `name` zeigt Abbildung 18. Für alle übrigen Attribute ist die Implementierung der Getter- und Setter-Methoden ähnlich, nur die Namen der Methoden und Parameter unterscheiden sich. Die grund-sätzliche Struktur bleibt jedoch erhalten. Trotzdem sei an dieser Stelle noch darauf hingewiesen, dass in Abbildung 18 der Parameter der Setter-Methode `setName` mit `name` genauso benannt wurde wie das private Attribut `name` der Klasse – was nicht zwingend so gemacht werden muss. Um aber in einem solchen Fall diese beiden unterscheiden zu können, dient hier das Schlüsselwort `this`, mit dem auf das Objekt selbst und damit auf das Klassenattribut (`this.name`) Bezug genommen werden kann.

Abbildung 18: Implementierte Getter- und Setter-Methode für ein Attribut der Klasse „Kunde“



Quelle: erstellt im Auftrag der IU, 2013.

Um die in einem Methodenrumpf implementierten Anweisungen auszuführen, muss die entsprechende Methode aufgerufen werden. Methoden können aus anderen Methodenrümpfen derselben Klasse heraus oder aus Methodenrümpfen einer anderen Klasse aufgerufen werden. Der Aufruf einer Methode erfolgt durch den Namen.

Codebeispiel 1: Aufruf der Methode `getName()` innerhalb der Klasse „Kunde“

Code

```
String id1= getName();
```

Hier wird die Getter-Methode des Attributs `name` aufgerufen, um den Wert von `name` in einer lokalen Variable `id1` (Details zu Variablen siehe Lernzyklus 4.2) zu speichern.

Codebeispiel 2: Aufruf der Methode `setName()` innerhalb der Klasse „Kunde“

Code

```
setName("Lange");
```

Hier wird die Setter-Methode des Attributs `name` aufgerufen, um den aktuellen Wert von `name` auf den Wert „Lange“ zu ändern.

Codebeispiel 3: Aufruf der Methode `getName()` der Klasse „Kunde“ aus einer anderen Klasse heraus

Code

```
String id2= kunde1.getName();
```

Das Objekt, das die Methode `getName()` in Kunde aufrufen will, hat ein Objekt der Klasse „Kunde“ unter dem Variablenamen `kunde1` gespeichert. Mit dem Aufruf `kunde1.getName()` kann gezielt die Methode `getName()` in dem Objekt `kunde1` aufgerufen werden. Der Wert von `kunde1.getName()` wird in der aufrufenden Klasse in der Variable (Details zu Variablen siehe Lernzyklus 4.2) `id2` gespeichert.

Codebeispiel 4: Aufruf der Methode `setName()` aus einer anderen Klasse heraus

Code

```
kunde1.setName("Lange");
```

Das Objekt, das die Methode `setName()` in Kunde aufrufen will, hat ein Objekt der Klasse „Kunde“ unter dem Variablenamen `kunde1` gespeichert. Mit dem Aufruf `kunde1.setName("Lange")` kann gezielt die Methode `setName(String name)` in dem Objekt `kunde1` aufgerufen werden. Über diesen Methodenaufruf kann aus einem anderen Objekt heraus der Wert „Lange“ als Attribut in das Objekt `kunde1` gespeichert werden.

Überladen von Methoden

Dabei implementiert man mehrere Methoden mit demselben Namen, jedoch unterschiedlichen Parameterlisten innerhalb einer Klasse.

Gibt es in einer Klasse Methoden mit selbem Namen, so spricht man vom **Überladen von Methoden**.

Mit dem Überladen, also dem mehrfachen Zurverfügungstellen einer Methode mit identischem Namen, erhält man als Entwickler die Möglichkeit unterschiedliches Methodenverhalten in Abhängigkeit von den gegebenen Parametern zu definieren.

Beispiel

Die folgende Methode soll verwendet werden, um im Online-Shop von Frau Lange einen Artikel in den Warenkorb zu legen. Dafür gibt es eine Methode `zumWarenkorbHinzufuegen`, die als Parameter den entsprechenden Artikel enthält. Der Artikel ist ein Objekt der Klasse „Artikel“:

Code

```
public void zumWarenkorbHinzufuegen (Artikel artikel) {...}
```

Zusätzlich zu dieser Methode soll es aber auch die Möglichkeit geben, mehrere gleiche Artikel in den Warenkorb zu legen. Dazu wird die Methode `zumWarenkorbHinzufuegen` überladen, indem eine zweite Methode `zumWarenkorbHinzufuegen` implementiert wird, die neben dem Artikel auch die Anzahl der hinzuzufügenden Artikel als Parameter erwartet:

Code

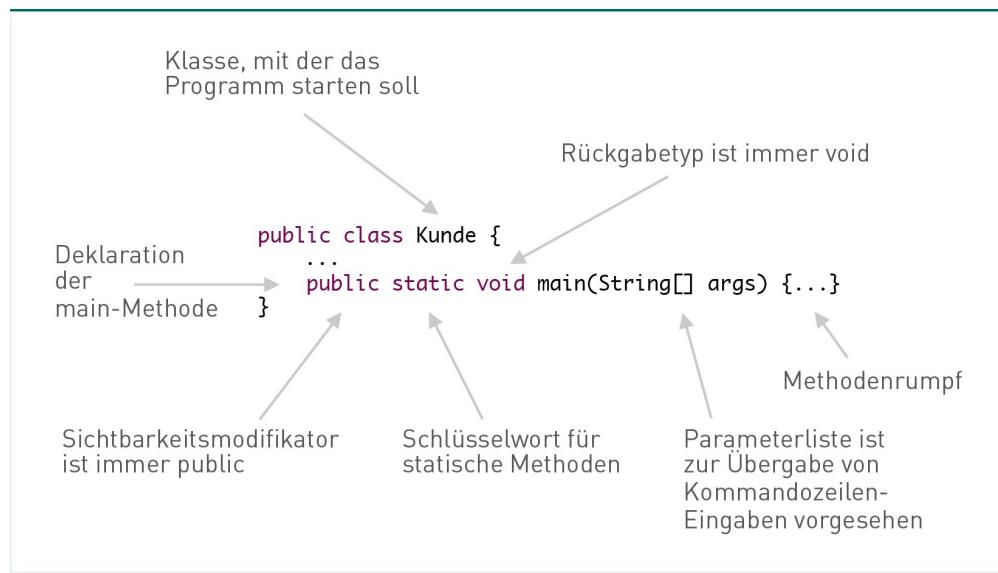
```
public void zumWarenkorbHinzufuegen (Artikel artikel, int anzahl) {...}
```

3.5 main-Methode: Startpunkt eines Java-Programms

Wie bereits beschrieben, bedeutet die konsequente Umsetzung der Objektorientierung das Zerlegen von Problemen in verschiedene Klassen eines UML Klassendiagramms. Ein Java-Programm besteht ebenfalls aus einer Zusammenstellung verschiedener Klassen, die untereinander über den Aufruf von Methoden kooperieren. Zur Laufzeit eines Java-Programms werden aus Klassen Objekte erzeugt, welche wiederum weitere Objekte erzeugen können. Daher kann man sich ein in Ausführung begriffenes Java-Programm als Netz von miteinander verbundenen Objekten vorstellen.

Für jedes Java-Programm gibt es jedoch einen festgelegten Startpunkt, das heißt, eine ganz bestimmte Methode wird immer als Erstes aufgerufen, wenn das Programm startet: die sogenannte **main-Methode**. Die Erzeugung aller von dem Programm benötigter Objekte startet innerhalb dieser Methode. Die main-Methode wird wie in Abbildung 19 dargestellt deklariert.

Abbildung 19: main-Methode als Startpunkt einer Klasse



Quelle: erstellt im Auftrag der IU, 2013.

Folgende Elemente der main-Methode sollten ohne Änderung übernommen werden:

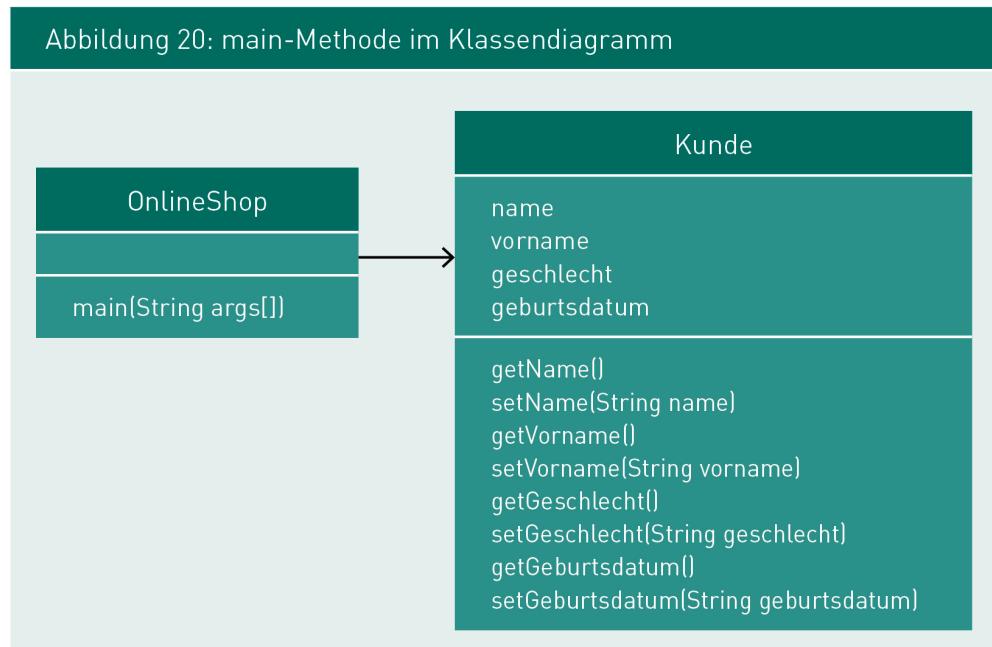
- Sichtbarkeitsmodifikator (public),
- Deklaration der main-Methode als statische Methode (static, Details siehe Lernzyklus 6.3),
- Festlegung, dass es keinen Rückgabetyp gibt (void),
- Name der Methode (main) sowie die
- Parameterliste (String args[]).

Nur der Methodenrumpf wird entsprechend angepasst. In welcher Klasse des Programms die main-Methode implementiert wird, kann der Entwickler frei entscheiden. Um Verwechslungen zu vermeiden, sollte es jedoch nur eine Klasse mit einer main-Methode geben.

Beispiel

Eine kleiner erster Prototyp des Online-Shops ist als Klassendiagramm in Abbildung 20 dargestellt. Eine Klasse „OnlineShop“ hat keine Attribute, aber eine main-Methode. Die Klasse „Kunde“ hat vier Attribute und die jeweiligen Getter- und Setter-Methoden dazu.

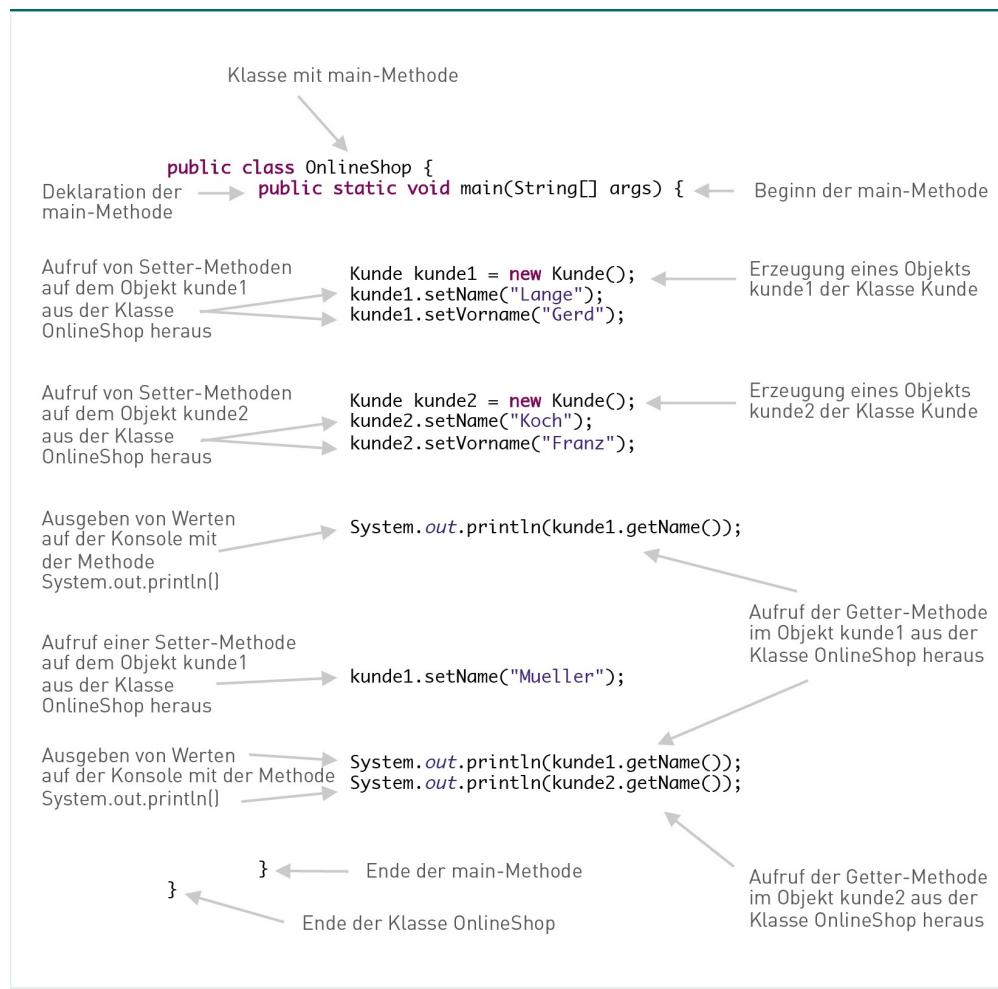
Abbildung 20: main-Methode im Klassendiagramm



Quelle: erstellt im Auftrag der IU, 2013.

Eine passende Beispielimplementierung der Klasse „OnlineShop“ mit der main-Methode liefert Abbildung 21. Im Methodenrumpf der main-Methode und damit beim Ausführen des Programms „OnlineShop“ werden zuerst zwei Objekte vom Typ Kunde angelegt, den lokalen Variablen kunde1 und kunde2 zugewiesen und in beiden Objekten jeweils die Werte für die Attribute name und vorname gespeichert. Dann wird zur Kontrolle auf der Konsole der Wert des Attributs name für das Objekt kunde1 ausgegeben. Anschließend wird das Attribut name von kunde1 auf einen neuen Wert gesetzt und der aktuelle Wert name sowohl für kunde1 und kunde2 auf der Konsole ausgegeben.

Abbildung 21: Beispielimplementierung einer main-Methode



Quelle: erstellt im Auftrag der IU, 2013.

Wie in dem Beispiel erkennbar ist, startet das Programm „Online-Shop“ mit der main-Methode der Klasse „OnlineShop“. Ausgehend von dieser main-Methode werden nun zwei konkrete Kunden angelegt, das heißt Objekte vom Typ Kunde erzeugt und als Variable kunde1 und kunde2 für die weitere Verwendung in der main-Methode von OnlineShop abgelegt.



ZUSAMMENFASSUNG

Die Programmiersprache Java ist eine plattformunabhängige Programmiersprache. Ein Java-Programm kann durch die Verwendung von betriebssystemspezifischen Java-Laufzeitumgebungen ohne Anpassungen auf fast jedem Betriebssystem ausgeführt werden.

Genau wie bei der objektorientierten Modellierung bestehen Java-Klassen neben ihrem eindeutigen Namen aus Attributen und Methoden. Im Vergleich zur Modellierung müssen jedoch bei der Java-Programmierung besondere Konventionen und Regeln eingehalten werden. Zu Attributen muss ein Sichtbarkeitsmodifikator, ein klassenweit eindeutiger Name und ein gültiger Datentyp festgelegt werden.

Bei der Programmierung von Methoden sind folgende Dinge zu beachten: Die Signatur einer Methode (Namen und Parameterliste) muss klassenweit eindeutig sein. Jede Methode muss darüber hinaus einen deklarierten Rückgabe-Datentyp (oder: void), einen Sichtbarkeitsmodifikator und einen implementierten Methodenrumpf haben. Im Methodenrumpf ist programmiert, was die Methode tut, genauer gesagt: die Anweisungen und Abarbeitungsvorschriften.

Der Zugriff auf Attribute durch eine andere Klasse wird in der Regel über sogenannte Getter- und Setter-Methoden ermöglicht. Das ermöglicht die konsequente Einhaltung des Prinzips der Datenkapselung.

Methoden können überladen werden, das heißt, es kann mehrere Methoden geben, die denselben Namen haben und sich nur durch die Parameterliste unterscheiden.

Ein Java-Programm wird immer durch die main-Methode gestartet. Diese Methode ist der Startpunkt für alle Aktivitäten und Abläufe des Programms. Enthält eine Klasse eine nach den Konventionen deklarierte main-Methode, ist damit automatisch bestimmt, wo das Programm mit der Abarbeitung der Anweisungen beginnt.

LEKTION 4

JAVA SPRACHKONSTRUKTE

LERNZIELE

Nach der Bearbeitung dieser Lektion werden Sie wissen, ...

- was für primitive Datentypen es in Java gibt und welchen Wertebereich diese abdecken.
- wie mit Variablen Werte zwischengespeichert werden können.
- welche wichtigen Operatoren es in Java gibt und wie sie verwendet werden.
- welche Kontrollstrukturen es gibt und wie damit kontrolliert Anweisungen wiederholt werden können.
- wie mithilfe von Paketen Sichtbarkeiten von Klassen und deren Elementen gezielt festgelegt werden können.

4. JAVA SPRACHKONSTRUKTE

Aus der Praxis

Herr Koch ist nun in der Lage, die Elemente eines UML Klassenmodells in einen Java-Programmcode zu implementieren. Dies bedeutet, er kann Klassen mit deren Attributen in Java programmieren und er weiß auch, wie man Methoden deklariert. Grundsätzlich ist ihm die main-Methode zwar bekannt, doch die Elemente der Programmiersprache Java, mit denen er die Algorithmen und Geschäftsregeln innerhalb eines Methodenrumpfes umsetzen kann, kennt er noch nicht.

4.1 Primitive Datentypen

Grundsätzlich werden Informationen bei der objektorientierten Programmierung in Attributten von Objekten gespeichert. Wie in Lernzyklus 3.3 bereits erläutert, bestimmt der Datentyp eines Attributes, welche Informationen in einem Attribut abgelegt werden dürfen. Wird als Datentyp eines Attributes der Name einer Klasse angegeben, dürfen nur Objekte dieser Klasse als konkrete Werte diesem Attribut zugewiesen werden.

Primitive Datentypen

So werden Datentypen bezeichnet, deren Werte keine Objekte sind, z. B. einfache Zahlen oder Wahrheitswerte.

Mit den sogenannten **primitiven Datentypen** gibt es in Java allerdings eine Menge von sehr einfachen Datentypen, die nicht durch eine eigene Klasse beschrieben werden.

Primitive Datentypen werden eingesetzt um Wahrheitswerte (`true`, `false`), ganze Zahlen (`1`, `12`, `13131`), Fließkommazahlen (`1.123`, `21234.1232`) und einzelne Zeichen (`t`, `w`, `f`, `d`) zu speichern. Primitive Datentypen sind einfache Standard-Datentypen, die es auch in anderen Programmiersprachen gibt. Eine Übersicht zu primitiven Datentypen in Java wird in Tabelle 11 dargestellt.

Tabelle 11: Primitive Datentypen in Java

Art der gespeicherten Werte	Schlüsselwort in Java	Beschreibung	Beispiele
Wahrheitswerte	<code>boolean</code>	Kann entweder <code>true</code> oder <code>false</code> sein. Weitere Werte sind nicht erlaubt.	<code>true</code> <code>false</code>
Ganze Zahl	<code>byte</code>	8Bit-Wertebereich von -128 bis 127	<code>123, 0, 23, 120</code>
	<code>short</code>	16Bit-Wertebereich von -32.768 bis 32.767	<code>-23000, 0, 13231</code>
	<code>int</code>	32Bit-Wertebereich von -2.147.483.648 bis 2.147.483.647	<code>-12332123, 234, 1102379239</code>

Art der gespeicherten Werte	Schlüsselwort in Java	Beschreibung	Beispiele
	long	64Bit-Wertebereich von -9.223.372.036.854.775.808 bis 9.223.372.036.854.775.808	8347829790645, 13, 0, 34879, 789234789274
Fließkommazahl	float	32Bit-Wertebereich von $1,40239846 \cdot 10^{-45}$ bis $3,40282347 \cdot 10^{38}$	1.87236f (Einfache Zahl mit Komma und angehängtem „f“) -3.938e12f ($-3.938 \cdot 10^{12}$ mit angehängtem „f“)
	double	64Bit-Wertebereich von $4,94065645841246544 \cdot 10^{-324}$ bis $1,79769131486231570 \cdot 10^{308}$	1.87236d (Einfache Zahl mit Komma und angehängtem „d“) -3.938e120d ($-3.938 \cdot 10^{120}$ mit angehängtem „d“)
Schriftzeichen (Character)	char	Ein einzelnes Unicode-Zeichen. Umfasst neben Ziffern, Buchstaben und Symbolen auch Steuerzeichen wie Leerzeichen, Tabulator oder Zeilenumbruch.	'A' (= Buchstabe A), '2' (= Ziffer 2), '\n' (= Steuerzeichen Zeilenumbruch)

Quelle: erstellt im Auftrag der IU, 2013.

Für ganze Zahlen wird in der Regel der Datentyp `int` verwendet, kleinere Datentypen wie `byte` oder `short` werden nur bei der Programmierung von Mikrocontrollern eingesetzt.

Ein weiterer häufig verwendeter Datentyp ist **String**. Er wird zur Speicherung von Zeichenketten verwendet. String ist kein primitiver Datentyp, sondern wird durch eine eigene Java-Klasse beschrieben. Dennoch kann String wie ein primitiver Datentyp Attributen (und Variablen, siehe Lernzyklus 4.2) zugewiesen werden. In Tabelle 12 wird die Verwendung von String beschrieben. Weitere Details zu Strings werden im Kurs „Datenstrukturen und Java-Klassenbibliothek“ behandelt.

String
Das ist ein Datentyp zum Speichern von Zeichenketten und hat in Java Eigenschaften von primitiven Datentypen, obwohl die Werte von String Java-Objekte sind.

Tabelle 12: String als Datentyp für Zeichenketten

Art der gespeicherten Werte	Schlüsselwort in Java	Beschreibung	Beispiele
Zeichenketten	String	Wird zum Speichern von beliebig langen Zeichenketten verwendet. Ist kein primitiver Datentyp. Die Klasse String bietet viele bereits vorhandene Methoden für die Bearbeitung von Zeichenketten.	"online", "Herr Koch", "Was nicht passt wird passend gemacht", ""

Quelle: erstellt im Auftrag der IU, 2013.

4.2 Variablen

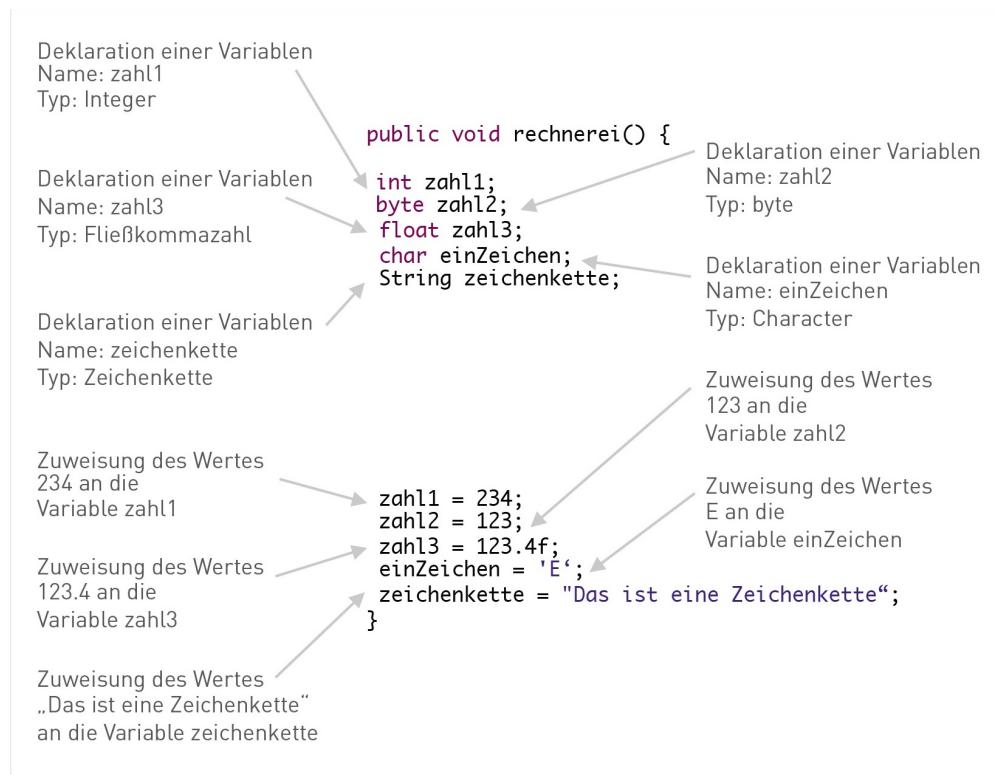
Zwar sind zum Speichern von Werten in Objekten deren Attribute vorgesehen, jedoch müssen während der Abarbeitung von Methodenrumpfen solche Ergebnisse zwischengespeichert werden, die keinen Einfluss auf die in den Attributen gespeicherten Werte haben sollen.

Mit Variablen bietet Java (und alle anderen Programmiersprachen auch) die Möglichkeit, konkrete Werte im Speicher der Anwendung abzulegen. Damit können zum Beispiel Ergebnisse von Berechnungen kurzzeitig gespeichert werden, die als Eingabeparameter für weitere Berechnungen benötigt werden.

Vergleichbar mit einem Attribut muss zur Deklaration einer Variable der Datentyp und der Name einer Variable festgelegt werden. Abgeschlossen wird die Deklaration (sowie im Folgenden auch alle Zuweisungen, Methodenaufrufe und Berechnungen) mit einem Semikolon „;“. Anders als bei Attributen können Variablen jedoch nur innerhalb des Methodenrumpfes verwendet werden. Daher benötigen sie keinen Sichtbarkeitsmodifikator.

Abbildung 22 zeigt im oberen Bereich beispielhaft die Deklaration von fünf Variablen. Nachdem Variablen deklariert worden sind, können sie innerhalb der Methode verwendet werden. Das bedeutet auch, dass ihr erst dann ein konkreter Wert zugewiesen werden kann, nachdem die Variable deklariert wurde.

Abbildung 22: Deklaration und Zuweisung von Variablen



Quelle: erstellt im Auftrag der IU, 2013.

Im unteren Bereich zeigt Abbildung 22 beispielhaft die Zuweisung von Werten an vier Variablen. Die Zuweisung wird durch das Gleichheitszeichen „=“ ausgedrückt. Links von „=“ steht der Name der Variable, rechts von „=“ der Wert, den die Variable erhalten soll. Wie in Abbildung 23 dargestellt, können Deklaration und Zuweisung von Variablen auch innerhalb einer Anweisung erfolgen. So kann der Programmcode aus Abbildung 22 durch den Code in Abbildung 23 vollständig ersetzt werden.

Abbildung 23: Zusammenfassung von Deklaration und Zuweisung

```

public void rechnerei() {
    int zahl1 = 234;
    byte zahl2 = 123;
    float zahl3 = 123.4f;
    char einZeichen = 'E';
    String zeichenkette = "Das ist eine Zeichenkette";
}
  
```

Quelle: erstellt im Auftrag der IU, 2013.

4.3 Operatoren und Ausdrücke

Arithmetische Operatoren

Das sind Operatoren zur Ausführung von mathematischen Funktionen wie Addition, Subtraktion, Division, Multiplikation.

Operatoren ermöglichen das Rechnen und Verändern von in Variablen und Attributen gespeicherten Werten. Die Programmiersprache Java unterstützt dafür eine Menge arithmetischer und logischer Operatoren. Die wichtigsten **arithmetischen Operatoren** sind in Tabelle 13 dargestellt. Generell wird der Datentyp des Ergebnisses eines Operators durch den Operanden, der den größten Wertebereich hat, bestimmt.

Tabelle 13: Wichtige arithmetische Operatoren

Ausgeführte Operation	Operator	Anwendbar auf die Datentypen	Beispiel
Inkrement; erhöht eine Variable um den Wert 1	<code>++</code>	Ganze Zahlen, Fließkommazahlen	<code>int a = 3; a++; float b = 3f; b++;</code>
Arithmetische Addition; der Ergebnistyp der Berechnung entspricht dem des Operanden mit dem größten Wertebereich (z. B. <code>int+int=int</code> , <code>int+long=long</code>)	<code>+</code>	Ganze Zahlen, Fließkommazahlen	<code>int c,d,e; c = 3; d = 5; e = c + d; int x; long y,z; x = 3; y = 4; z = x + y;</code>
Arithmetische Subtraktion	<code>-</code>	Ganze Zahlen, Fließkommazahlen	<code>int e = 3; float f = 4; float g; g = e - f;</code>
Arithmetische Multiplikation	<code>*</code>	Ganze Zahlen, Fließkommazahlen	<code>int c,d; c = 3; d = c * 4;</code>
Arithmetische Division; berechnet Quotienten aus Dividend und Divisor	<code>/</code>	Ganze Zahlen: Sind beide Operanden ganze Zahlen, wird vom Ergebnis der Teil nach dem Komma abgeschnitten. Fließkommazahlen: Ist mindestens ein Operand eine Fließkommazahl, wird das Ergebnis auch eine Fließkommazahl und nicht gerundet.	<code>int h = 4; int i = 3; int j; j = h/i; int o = 4; double p = 3; double q; q = o/p;</code>
Rest (auch: Restwert-Operator); berechnet den Rest der arithmetischen Division	<code>%</code>	Ganze Zahlen, Fließkommazahlen	<code>int k = 11; int l = 5; int m; m = k % l;</code>

Quelle: erstellt im Auftrag der IU, 2013.

Im Unterschied zu arithmetischen Operatoren werten **logische Operatoren** Ausdrücke zu wahr (true) oder falsch (false) aus. Sie werden daher häufig zur Steuerung von Kontrollstrukturen (siehe Lernzyklus 4.4) eingesetzt. Der Ergebnistyp von logischen Operatoren ist immer boolean. Eine Übersicht wichtiger logischer Operatoren ist in Tabelle 14 dargestellt.

Tabelle 14: Wichtige logische Operatoren

Ausgeführte Operation	Operator	Anwendbar auf die Datentypen	Beispiel
Logisches Komplement (Negation); ändert den Wahrheitswert des Operanden	!	boolean	<code>boolean b1; boolean b2 = false; b1 = !b2;</code>
Logisches UND; liefert true, wenn beide Operanden true sind	&&	boolean	<code>boolean b3; boolean b4 = true; boolean b5 = true; b3 = b4 && b5;</code>
Logisches ODER; liefert true, wenn einer der beiden Operanden true ist		boolean	<code>boolean b6; boolean b7 = false; boolean b8 = true; b6 = b7 b8;</code>
Exklusiv-ODER; liefert true, wenn ein Operand true und ein Operand false ist	^	boolean	<code>boolean b9; boolean b10 = false; boolean b11 = true; b9 = b10 ^ b11;</code>

Quelle: erstellt im Auftrag der IU, 2013.

Vergleichsoperatoren vergleichen Ausdrücke miteinander und liefern als Ergebnistyp boolean. Das heißt, sie liefern entweder wahr (true) oder falsch (false) zurück. Ebenso wie logische Operatoren werden Vergleichsoperatoren häufig zur Steuerung von Kontrollstrukturen (siehe Lernzyklus 4.4) eingesetzt. Tabelle 15 gibt einen Überblick über häufig eingesetzte Vergleichsoperatoren.

Logische Operatoren
Das sind Operatoren zur Ausführung von logischen Funktionen wie Negation, logisches UND, logisches ODER.

Vergleichsoperatoren
Das sind Operatoren zur Ausführung von Funktionen zum Vergleich von Werten und Objektreferenzen.

Tabelle 15: Wichtige Vergleichsoperatoren

Ausgeführte Operation	Operator	Anwendbar auf die Datentypen	Beispiel
Gleichheit	==	Primitive Datentypen: liefert true, wenn die Werte der Operanden gleich sind	int z1, z2; boolean e1; z1 = 3; z2 = 3; e1 = z1 == z2;
		Referenzdatentypen: liefert true, wenn in beiden Operanden die Referenz auf das-selbe Objekt enthal-ten ist	Kunde kunde1, kunde2; boolean e2; kunde1 = new Kunde(); kunde2 = kunde1; e2 = kunde1 == kunde2;
Ungleichheit	!=	Primitive Datentypen: liefert true, wenn die Werte der Operanden nicht gleich sind	int z3, z4; boolean e3; z3 = 4; z4 = 3; e3 = z3 != z4;
		Referenzdatentypen: liefert true, wenn in beiden Operanden die unterschiedli-chen Referenzen ent-halten sind	Kunde kunde3, kunde4; boolean e4; kunde3 = new Kunde(); kunde4 = new Kunde(); e4 = kunde3 != kunde4;
Kleiner als liefert true, wenn der Wert des linken Operanden kleiner ist ver-glichen mit dem Wert des rechten Operanden	<	Ganze Zahlen, Fließkommazah-len	int z5, z6; boolean e5; z5 = 4; z6 = 5; e5 = z5 < z6;
Kleiner gleich liefert true, wenn der Wert des linken Operanden kleiner oder gleich ist verglichen mit dem Wert des rechten Operanden	<=	Ganze Zahlen, Fließkommazah-len	int z5, z6; boolean e5; z5 = 4; z6 = 5; e5 = z5 <= z6;
Größer als liefert true, wenn der Wert des linken Operanden größer ist vergli-chen mit dem Wert des rechten Ope-randen	>	Ganze Zahlen, Fließkommazah-len	int z7, z8; boolean e6; z7 = 6; z8 = 5; e6 = z7 > z8;
Größer gleich liefert true, wenn der Wert des linken Operanden größer oder gleich ist verglichen mit dem Wert des rechten Operanden	>=	Ganze Zahlen, Fließkommazah-len	int z7, z8; boolean e6; z7 = 6; z8 = 5; e6 = z7 >= z8;

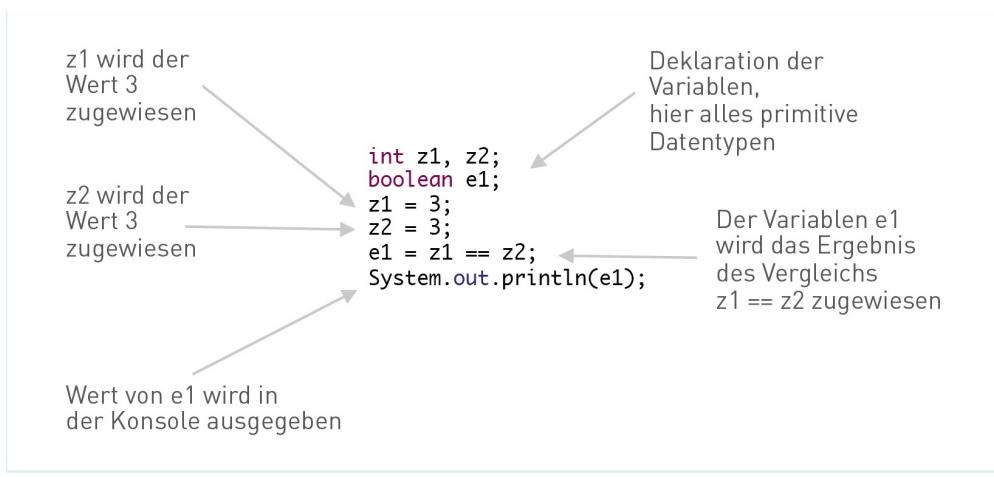
Ausgeführte Operation	Operator	Anwendbar auf die Datentypen	Beispiel
Typvergleich liefert <code>true</code> , wenn der Datentyp des linken Operanden gleich dem gegebenen Datentypen im rechten Operanden ist	<code>instanceof</code>	Referenzdatentypen	<pre>Kunde k1; boolean e7; k1 = new Kunde(); e7 = k1 instanceof Kunde;</pre>

Quelle: erstellt im Auftrag der IU, 2013.

Besondere **Prüfoperatoren** sind die Prüfung auf Gleichheit (`==`) und die Prüfung auf Ungleichheit (`!=`). Denn bei diesen Operatoren bestimmen die Operanden wie verglichen wird. Dabei unterscheidet sich der Vergleich zwischen primitiven Datentypen und Referenzdatentypen grundsätzlich. Sind beide Operanden primitive Datentypen, so werden direkt die Werte der Operanden verglichen. Abbildung 24 zeigt ein Beispiel für den Vergleich primitiver Datentypen, bei dem das Ergebnis (`e1`) auf der Konsole ausgegeben wird. In diesem Fall hat `e1` den Wert `true`.

Prüfoperatoren
Das sind Vergleichsoperatoren zum Ausführen von Funktionen zur Prüfung auf Gleichheit oder Ungleichheit. Dabei legen die Operanden fest, ob Werte oder Referenzen verglichen werden.

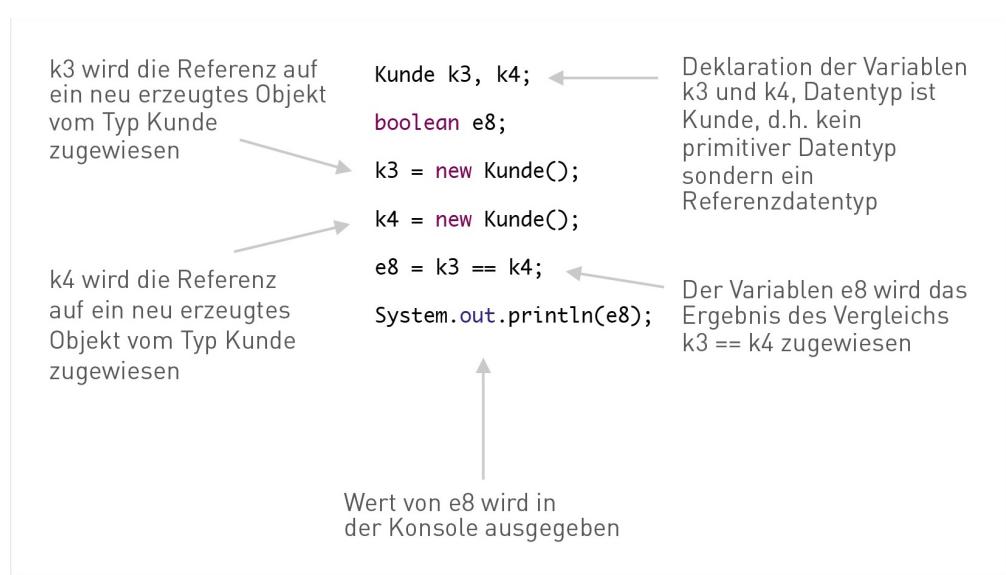
Abbildung 24: Beispiel für Vergleich primitiver Datentypen



Quelle: erstellt im Auftrag der IU, 2013.

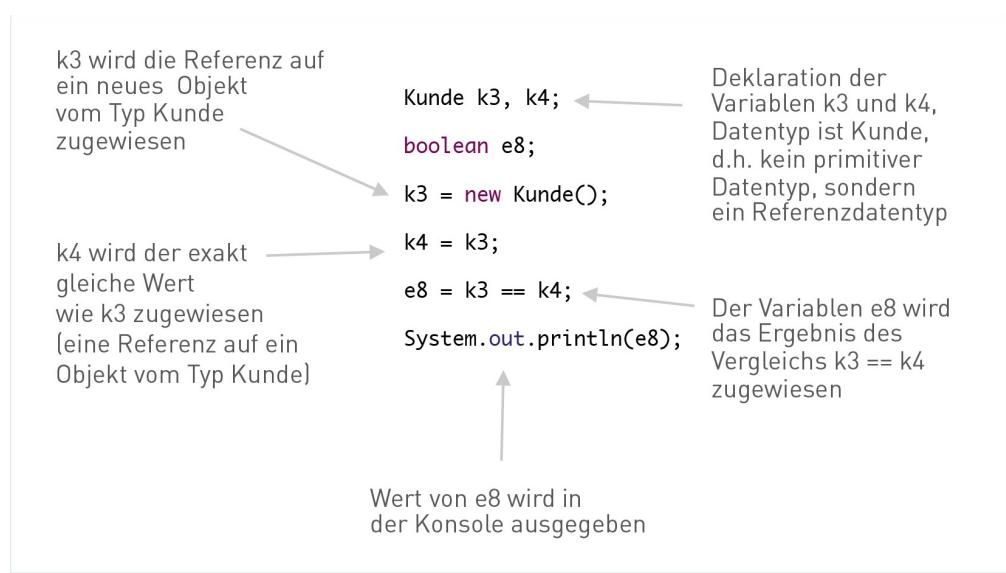
Für den Fall, dass Referenzdatentypen verglichen werden sollen, vergleicht der Operator „`==`“ nicht den tatsächlichen Inhalt der zu vergleichenden Objekte. Es wird vielmehr nur verglichen, ob beide Operatoren dieselben Objekte referenzieren, das heißt, auf das gleiche Objekt „zeigen“. In Abbildung 25 wird `k3` und `k4` jeweils ein neu erzeugtes Objekt zugewiesen. In `k3` ist die Referenz auf das erste Objekt „`Kunde`“ gespeichert, in `k4` die Referenz auf das zweite Objekt „`Kunde`“. Der Vergleich auf „Referenzgleichheit“ von `k3` und `k4` liefert damit den Wert `false`.

Abbildung 25: Beispiel für den Vergleich von Referenzdatentypen



In Abbildung 26 wird das Codebeispiel von Abbildung 25 dahin gehend angepasst, dass der Variablen k4 kein neu erzeugtes Objekt mehr zugewiesen wird. Der Variablen k4 wird nun exakt der gleiche Wert zugewiesen, der in k3 gespeichert ist. Da der Wert von k3 eine Referenz auf ein Objekt vom Typ „Kunde“ ist, hat k4 nach der Anweisung `k4 = k3` genau dieselbe Referenz wie der Wert k3. Damit liefert die Operation `k3 == k3` das Ergebnis `true`.

Abbildung 26: Gleichheit von Referenzdatentypen



Darüber hinaus gibt es noch einen Operator, der häufig verwendet wird, um Zeichenketten miteinander zu verbinden, das heißt, mehrere Zeichenketten zu einer neuen zusammenzuführen. Tabelle 16 zeigt diesen Operator, der die **String-Konkatenation** ausführt.

Tabelle 16: Verketten von Zeichenketten

Ausgeführte Operation	Operator	Anwendbar auf die Datentypen	Beispiel
Verketten von Zeichen (Konkatenation)	+	String	<pre>String s1, s2, s3, s4; s1 = "Hallo"; s2 = " "; s3 = "Welt!"; s4 = s1 + s2 + s3;</pre>
Verketten von mehreren Zeichenketten zu einer neuen Zeichenkette			

Quelle: erstellt im Auftrag der IU, 2013.

String-Konkatenation
Dies ist eine Funktion des Datentyps String, die aus zwei gegebenen Zeichenketten durch einfaches Aneinanderreihen eine neue Zeichenkette erzeugt.

4.4 Kontrollstrukturen

Wie in Lernzyklus 3.4 erläutert, wird die Reihenfolge der Anweisungen innerhalb eines Methodenrumpfes durch die Reihenfolge der implementierten Anweisungen vorgegeben. Beim Ablauf des Programms wird ein Methodenrumpf von oben nach unten durchgearbeitet. Bei der Implementierung von Algorithmen und Geschäftsregeln ist es jedoch häufig notwendig, konkrete Anweisungen nur beim Vorliegen ganz bestimmter Bedingungen durchzuführen. Weiterhin ist es häufig notwendig, dieselben Anweisungen hintereinander für eine Menge von Werten auszuführen, zum Beispiel „für alle Kunden“ oder „für alle Artikel“.

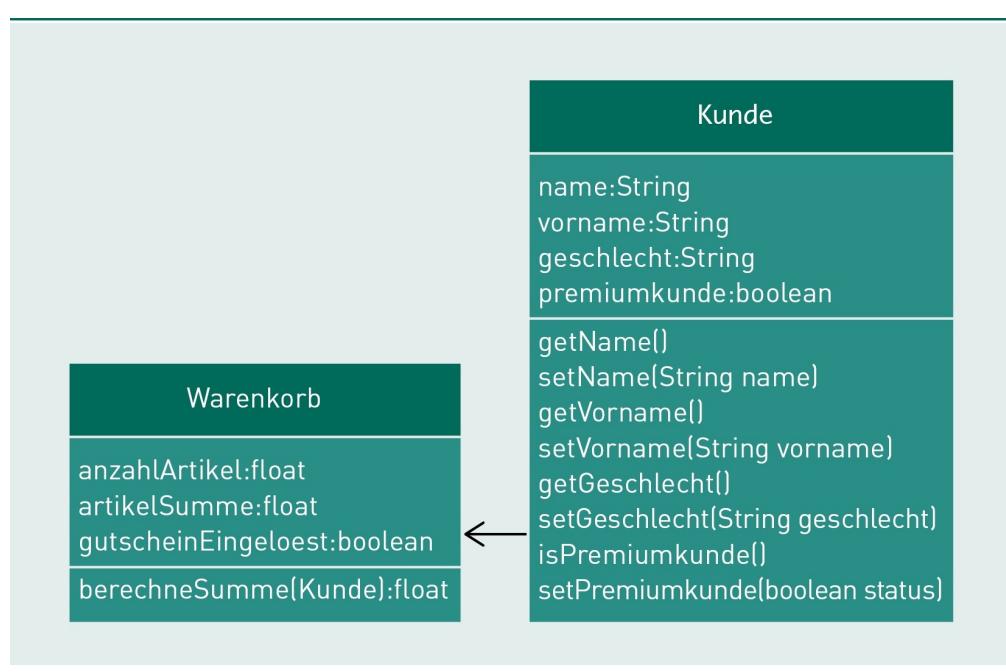
Das kontrollierte wiederholte Ausführen von Anweisungen wird durch sogenannte **Kontrollstrukturen** ermöglicht. Die wichtigsten von Java unterstützten Kontrollstrukturen sind:

- bedingte Verzweigungen (`if-else`) und
- Schleifen (`for`, `while`, `do-while`).

Kontrollstrukturen
Das sind Elemente einer Programmiersprache zur bedingten oder mehrfachen Ausführung von Anweisungen.

Die Klassen aus Abbildung 27 bilden das Grundgerüst für die Beispiele in diesem Lernzyklus. Es handelt sich dabei um die Klasse „Warenkorb“ und die Klasse „Kunde“ mit Attributen sowie teilweise den passenden Getter- und Setter-Methoden dazu. (Hinweis: Die Getter-Methoden für Attribute vom Typ `boolean` beginnen statt mit „get“ mit „is“.) Zunächst interessiert die Implementierung der Methode `berechneSumme(Kunde)` der Klasse „Warenkorb“.

Abbildung 27: Klassendiagramm mit Klassen „Warenkorb“ und „Kunde“



Quelle: erstellt im Auftrag der IU, 2013.

Bedingte Verzweigung

Das ist eine Kontrollstruktur, um die Ausführung von Anweisungen abhängig von Bedingungen zu machen.

Die **bedingte Verzweigung** wird verwendet, um die konkrete Stelle, mit der das Programm fortgesetzt wird, anhand einer Bedingung zu prüfen. Abbildung 28 illustriert einen Anwendungsfall für bedingte Verzweigungen: In Abhängigkeit vom Status des Kunden wird ein Preisnachlass gewährt oder nicht. Die Struktur einer bedingten Verzweigung in Java sieht wie folgt aus:

Code

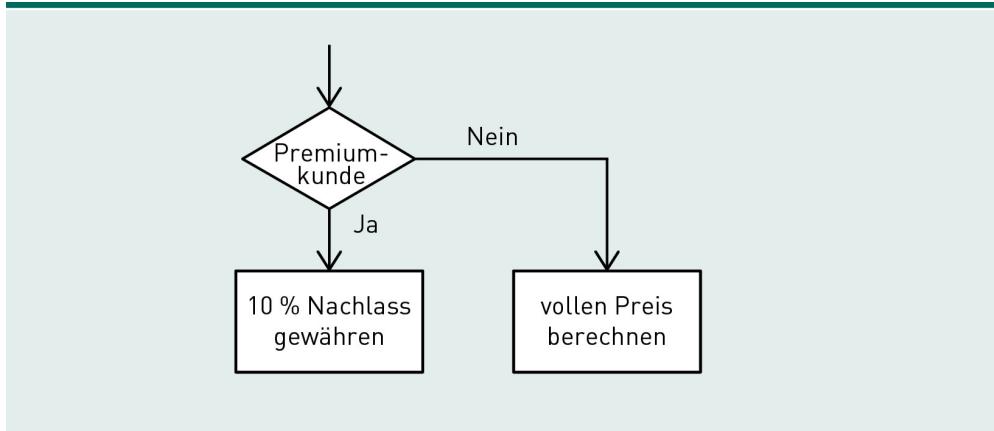
```

if (Bedingung) {
    Anweisung1;
}
else {
    Anweisung2;
}

```

Die Bedingung muss dabei ein Ausdruck sein, der zu `true` oder `false` ausgewertet werden kann (siehe dazu Lernzyklus 4.3). Die `Anweisung1` wird nur dann ausgeführt, wenn die Bedingung `true` ist. In diesem Fall wird `Anweisung2` nicht ausgeführt. `Anweisung2` wird hingegen nur dann ausgeführt, wenn die Bedingung `false` ist und damit `Anweisung1` nicht ausgeführt wurde. Der `else`-Teil in bedingten Verzweigungen ist optional.

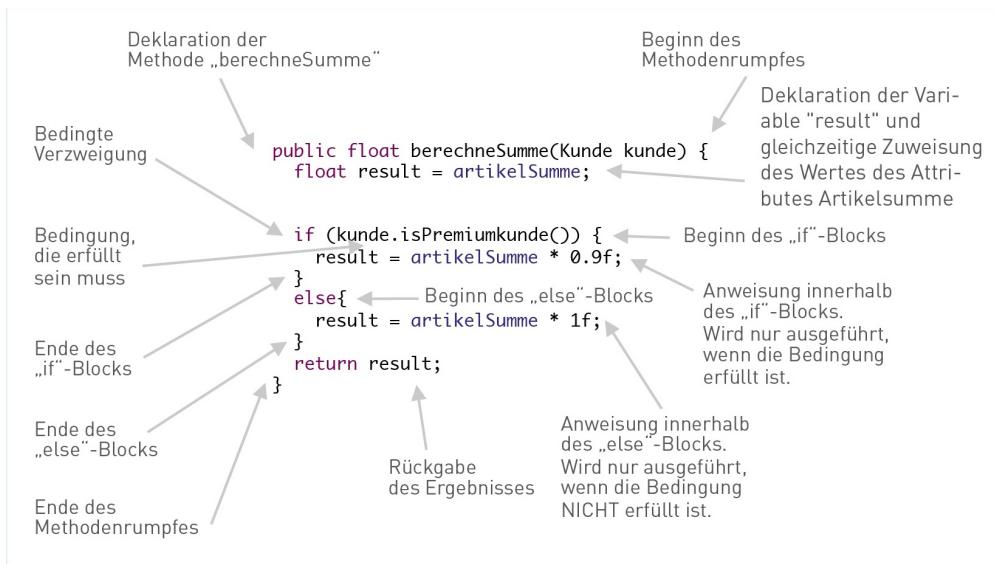
Abbildung 28: Anwendungsfall für bedingte Verzweigungen



Quelle: erstellt im Auftrag der IU, 2013.

Abbildung 29 zeigt die Implementierung einer Methode `berechneSumme(Kunde)` der Klasse „Warenkorb“ die eine einfache bedingte Verzweigung enthält. In dieser Methode wird in Abhängigkeit des Attributs `premiumkunde` die Gesamtsumme eines Warenkorbs berechnet. Ist der Kunde ein Premiumkunde, dann verringert sich die Summe um 10 %. Andernfalls ändert sich die Summe nicht. Bei einer bedingten Verzweigung werden entweder die Anweisungen im „if-Block“ ausgeführt oder die Anweisungen im „else-Block“. Niemals jedoch beide Blöcke. Der „else-Block“ ist dabei optional und kann je nach Problemstellung auch weggelassen werden.

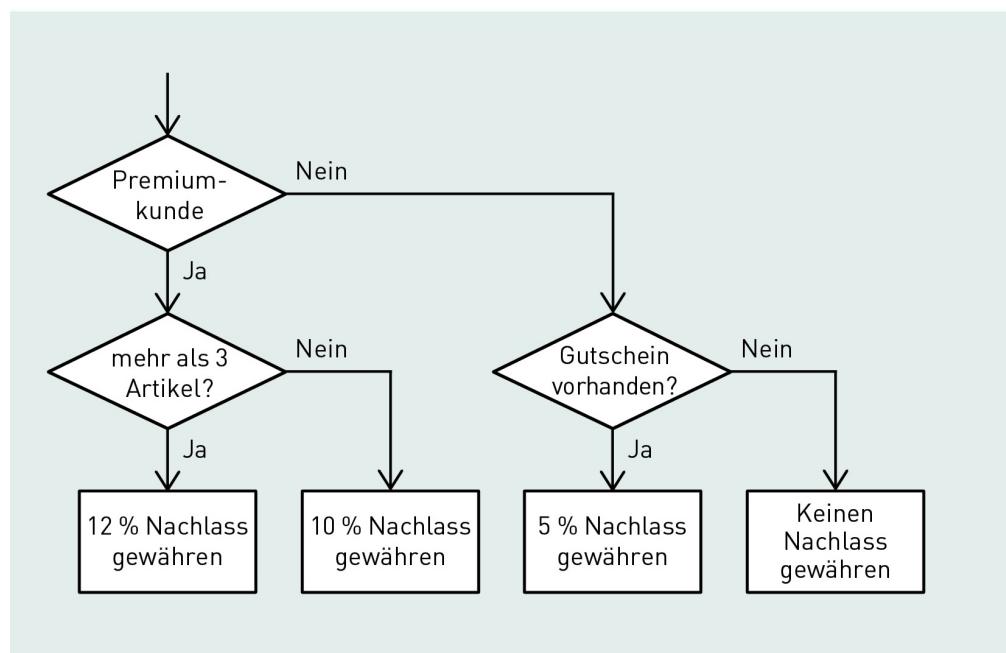
Abbildung 29: Beispiel für einfache bedingte Verzweigungen



Quelle: erstellt im Auftrag der IU, 2013.

Nun soll die Berechnung dahin gehend erweitert werden, dass sowohl die Anzahl der im Warenkorb befindlichen Artikel für Premiumkunden als auch das Einlösen von Gutscheinen für Nicht-Premiumkunden berücksichtigt werden. Abbildung 30 stellt die geänderte Berechnungsvorschrift dar.

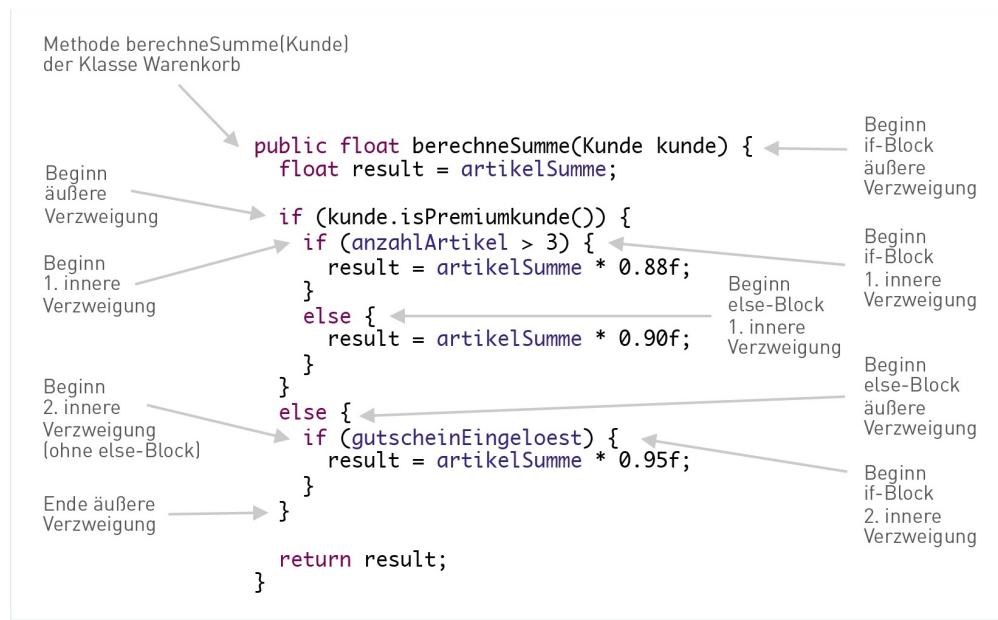
Abbildung 30: Anwendungsfall für verschachtelte Verzweigung



Quelle: erstellt im Auftrag der IU, 2013.

Das entsprechend angepasste Codebeispiel der Methode berechneSumme(Kunde) der Klasse „Warenkorb“ ist in Abbildung 31 dargestellt. Die Anpassung wurde mit verschachtelten if-else-Verzweigungen implementiert, wobei der else-Block in der 2. inneren Verzweigung weggelassen wurde, da für Nicht-Premiumkunden ohne Gutschein kein Nachlass gewährt wird und damit die Summe unverändert bleibt.

Abbildung 31: Beispiel für verschachtelte Verzweigungen



Quelle: erstellt im Auftrag der IU, 2013.

In einer **erweiterten if-else-Verzweigung** wird nicht nur eine Bedingung geprüft, bevor der else-Block abgearbeitet wird, sondern mehrere sich gegenseitig ausschließende Bedingungen. Die Struktur einer erweiterten if-else-Verzweigung in Java sieht wie folgt aus:

Code

```

if (Bedingung1) {
    Anweisung1;
}

else if (Bedingung2) {
    Anweisung2;
}

else {
    Anweisung3;
}

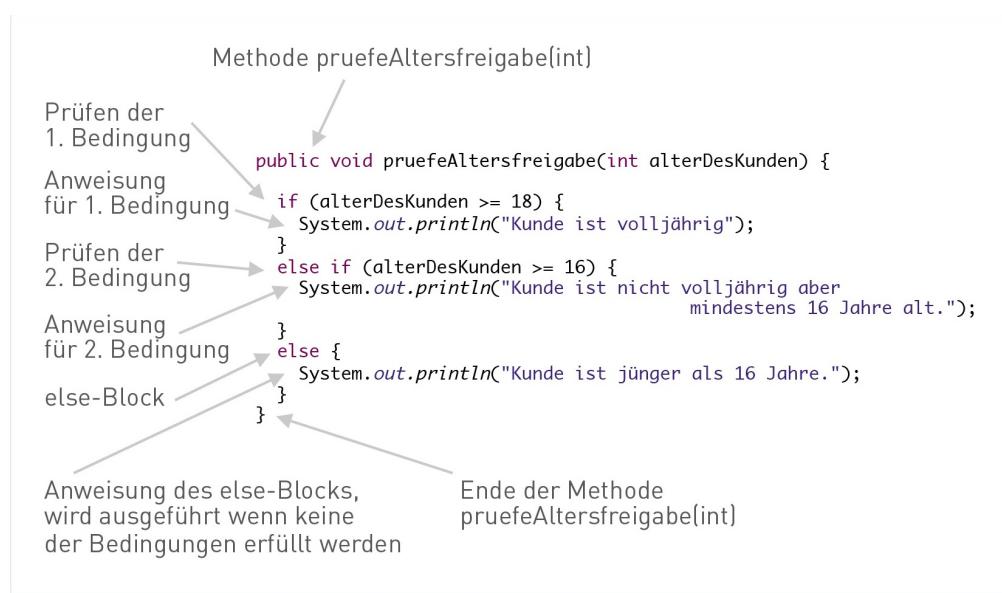
```

Erweiterte if-else-Verzweigung

Dies ist eine Kontrollstruktur, um die Ausführung von Anweisungen abhängig von mehreren, sich gegenseitig ausschließenden Bedingungen zu machen.

Ein entsprechendes Codebeispiel ist in Abbildung 32 abgebildet. Aus Gründen des Jugendschutzes soll die Altersfreigabe überprüft werden. Dazu wird das Alter des Kunden überprüft und der Kunde dadurch in eine Altersklasse eingeteilt sowie eine Meldung auf der Konsole ausgegeben. Auch bei einer erweiterten if-else-Verzweigung wird jeweils nur ein Block abgearbeitet. Obwohl bei einem Alter von 28 Jahren sowohl die 1. als auch die 2. Bedingung erfüllt sind, wird nur die Anweisung für die 1. Bedingung ausgeführt.

Abbildung 32: Beispiel für if-else If-else Verzweigung



Quelle: erstellt im Auftrag der IU, 2013.

Darüber hinaus gibt es in Java auch eine Kontrollstruktur für komplexe Verzweigungen (switch). Diese Struktur wird in diesem Kurs nicht eingesetzt, da mit if-else-Strukturen komplexe Verzweigungen nachgebildet werden können.

Schleifen

Schleifen sind eine Kontrollstruktur, um die mehrfache Ausführung von gleichen Anweisungen zu ermöglichen.

Neben den Verzweigungen sind die sogenannten **Schleifen** eine weitere wichtige Kontrollstruktur, die es übrigens in fast jeder Programmiersprache gibt. Schleifen ermöglichen die mehrfache Ausführung von gleichen Anweisungen hintereinander. Die Anzahl der Schleifendurchläufe wird von einer Einhaltung einer Schleifenbedingung (auch: Laufbedingung, Abbruchbedingung) bestimmt. Grundsätzlich können drei verschiedene Schleifenarten unterschieden werden:

- While-Schleife,
- Do-while-Schleife und
- For-Schleife.

While-Schleife

Das ist eine kopfgesteuerte Schleife. Die Bedingungen wird vor jeder Ausführung der Anweisungen geprüft.

Die **while-Schleife** prüft zuerst die Schleifenbedingung. Wenn die Bedingung erfüllt ist, werden die Anweisungen der Schleife ausgeführt. Andernfalls werden die Anweisungen der Schleife übersprungen. Nach der Ausführung der Anweisungen der Schleife wird erneut die Bedingung geprüft. Wird diese zu false ausgewertet, ist die Schleife beendet. Bei einer Auswertung der Bedingung zu true werden die Anweisungen der Schleife erneut durchlaufen. Die Struktur einer while-Schleife sieht folgendermaßen aus:

Code

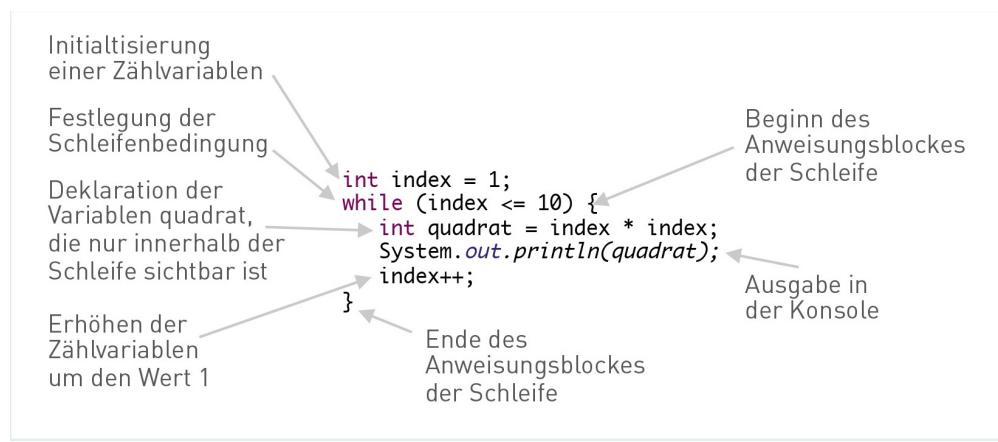
```

while (Bedingung) {
    Anweisungen;
}

```

Die while-Schleife wird auch kopfgesteuerte Schleife genannt, denn bereits vor dem ersten Ablauf wird die Bedingung bereits geprüft. Wird sie dabei zu `false` ausgewertet, wird die komplette Schleife übersprungen, das heißt, keine Anweisung der Schleife ausgeführt. Abbildung 33 zeigt ein Beispiel für eine while-Schleife, die alle Quadratzahlen für die Werte 1 bis 10 berechnet und auf der Konsole ausgibt. Die Zählvariable `index` wird dabei außerhalb der Schleife initialisiert und für die Steuerung der Schleife verwendet: Die Einhaltung der Bedingung wird anhand des aktuellen Wertes von `index` geprüft und `index` während der Abarbeitung der Schleife verändert. Wie oft die Schleife tatsächlich durchlaufen wird, bestimmt das Zusammenspiel des initialen Wertes von `index`, die Schleifenbedingung und die Anweisung, die den Wert von `index` verändert.

Abbildung 33: Beispiel einer while-Schleife



Quelle: erstellt im Auftrag der IU, 2013.

Die **do-while-Schleife** ist eine fußgesteuerte while-Schleife. Die Anweisungen der Schleife werden in jedem Fall mindestens einmal ausgeführt. Erst dann wird die Schleifenbedingung geprüft. Wenn die Bedingung erfüllt ist, werden die Anweisungen der Schleife wiederholt ausgeführt. Andernfalls wird die Schleife beendet. Die Struktur einer do-while-Schleife sieht folgendermaßen aus:

Code

```

do {
    Anweisungen;
} while (Bedingung)

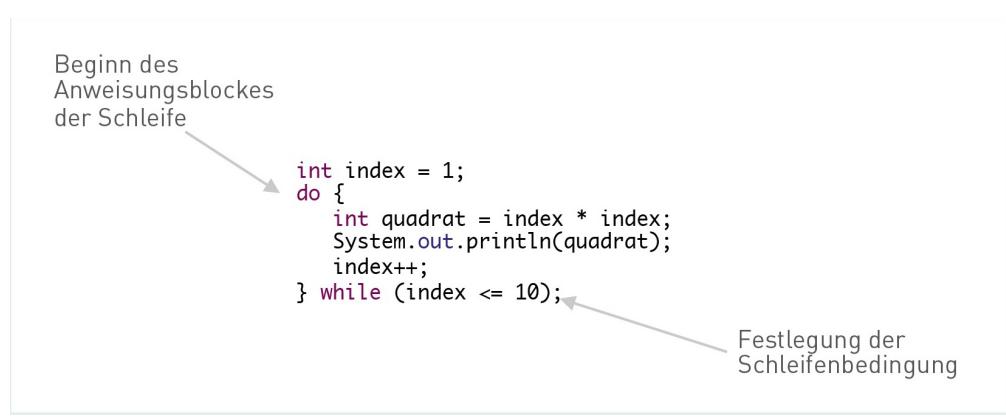
```

Do-while-Schleife

Dies ist eine fußgesteuerte Schleife; Anweisungen werden mindestens einmal durchgeführt. Die Bedingung wird erstmals vor der zweiten Ausführung geprüft.

Abbildung 34 zeigt die Implementierung der Berechnung der Quadratzahlen aus Abbildung 33 unter Verwendung einer do-while-Schleife. Beide Schleifen verhalten sich identisch und können in diesem Fall gegeneinander ausgetauscht werden.

Abbildung 34: Beispiel einer do-while-Schleife



Quelle: erstellt im Auftrag der IU, 2013.

For-Schleife

Die for-Schleife ist eine kopfgesteuerte Schleife, deren Kopf neben der Bedingung auch die Anweisungen zur Initialisierung der Zählvariablen und deren Weiterschaltung enthält.

Im Unterschied zur while-Schleife und zur do-while-Schleife sind bei der **for-Schleife** die Initialisierung, Bedingungsprüfung und das Ändern der Zählvariablen Elemente des Schleifenkopfes. Genauso wie die while-Schleife ist die for-Schleife eine kopfgesteuerte Schleife, das heißt, vor dem erstmaligen Ausführen der Schleife wird geprüft, ob die Bedingung erfüllt ist.

Die Struktur einer for-Schleife kann folgendermaßen dargestellt werden:

Code

```
for (Initialisierung; Bedingung; Schleifenfortschaltung) {
    Anweisungen;
}
```

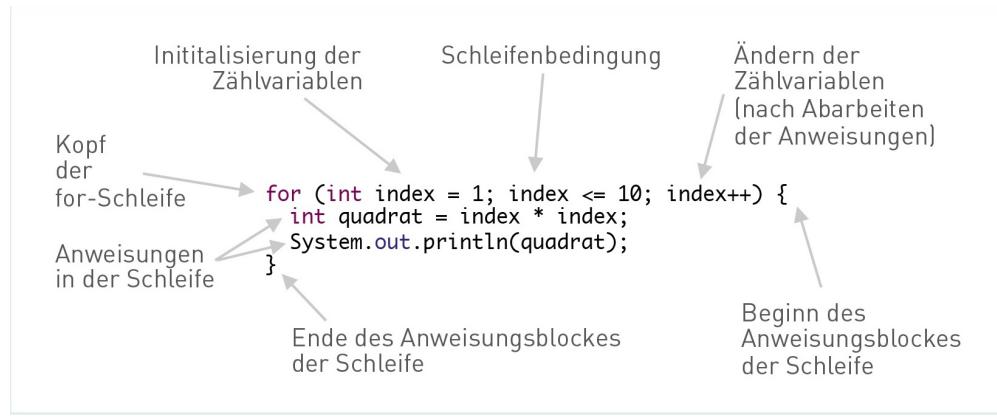
Der Bereich bis zum ersten Semikolon „;“ des Schleifenkopfes ist für die Initialisierung der Zählvariablen reserviert. In den Bereich zwischen dem ersten und dem zweiten Semikolon „;“ steht die Schleifenbedingung. Im Bereich zwischen zweitem Semikolon „;“ und der Klammer „)“ werden Zählvariablen verändert.

Das Ablaufschema einer for-Schleife kann wie folgt beschrieben werden:

1. Ausführen der Anweisung der Initialisierung
2. Prüfung der Bedingung
3. Auswerten der Bedingung
 1. Wenn Bedingung true: Ausführung aller Anweisungen der for-Schleife
 2. Wenn Bedingung false: Abbruch und keine Ausführung der Anweisungen
4. Ausführen der Anweisung der Schleifenfortschaltung
5. Weiter mit Punkt 2 (Prüfung der Bedingung)

Abbildung 35 zeigt eine in Programmcode umgesetzte for-Schleife, welche die Quadratzahlen von 1 bis 10 berechnet und ausgibt. Dabei handelt es sich um die gleiche Berechnung, die oben mit der while-Schleife und der do-while-Schleife durchgeführt wurde. Dabei wird deutlich, dass im Anweisungsblock keine Steueranweisungen der Schleife, wie zum Beispiel das Ändern der Zählvariablen, implementiert sind.

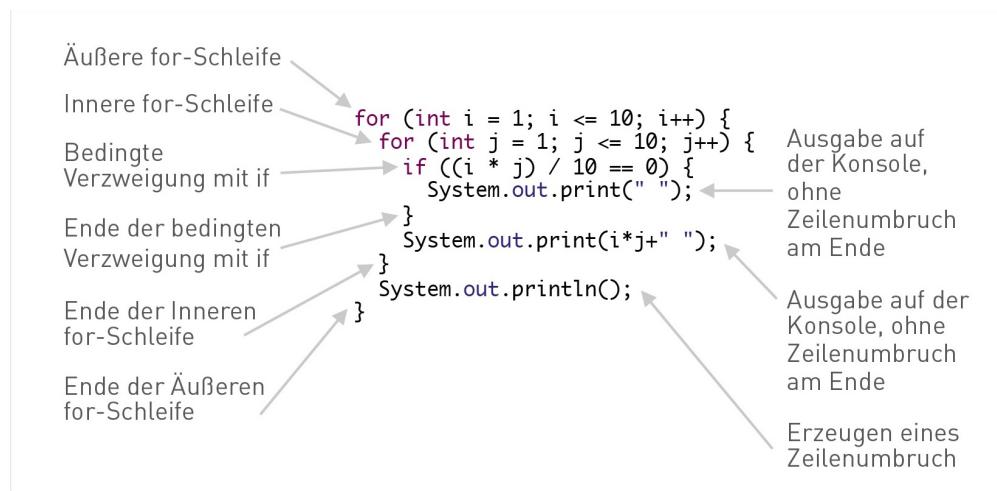
Abbildung 35: Beispiel einer for-Schleife



Quelle: erstellt im Auftrag der IU, 2013.

Kontrollstrukturen können in ihren Anweisungsblöcken selbst wieder Kontrollstrukturen enthalten. Daher können auch Schleifen ineinander geschachtelt werden. In Abbildung 36 ist ein Codebeispiel dargestellt, das zwei verschachtelte for-Schleifen enthält. Alle Variablen, die in äußeren Kontrollstrukturen deklariert wurden, sind auch in inneren Kontrollstrukturen verfügbar. Auf die Variable *i* kann auch innerhalb der zweiten for-Schleife zugegriffen werden. Allerdings können Anweisungen der ersten for-Schleife nicht auf die Variable *j* der inneren for-Schleifen zugreifen.

Abbildung 36: Beispiel für verschachtelte Kontrollstrukturen



Quelle: erstellt im Auftrag der IU, 2013.

4.5 Pakete und Sichtbarkeitsmodifikatoren

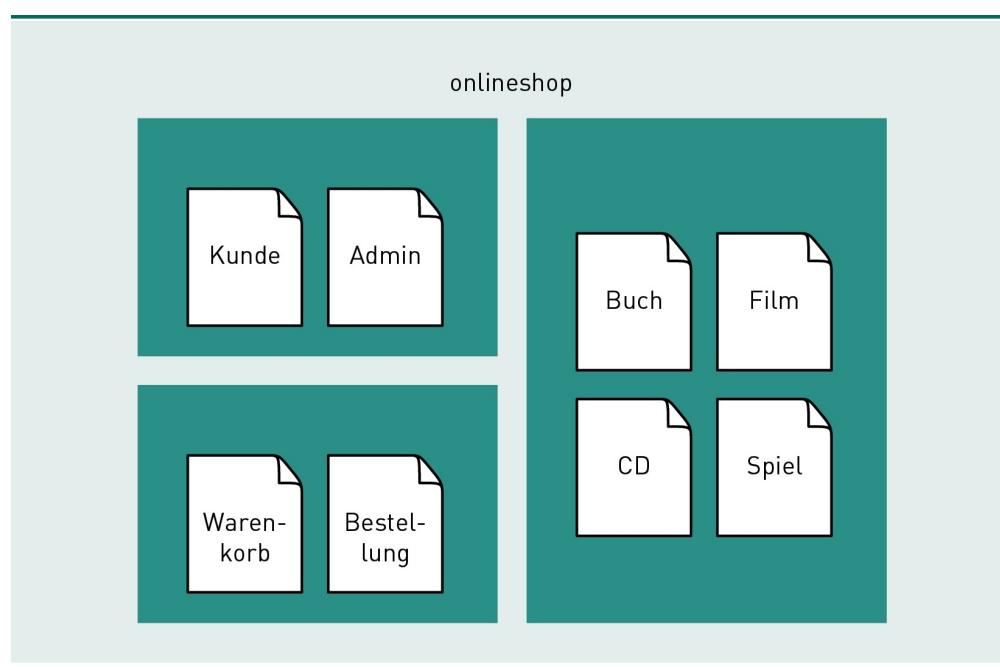
Paket

Das ist ein Element zur Strukturierung der Java-Klassen eines Entwicklungsprojektes. Java-Klassen werden Paketen zugeordnet. Pakete können selber auch Pakete enthalten.

Im Projektverlauf von industriellen Entwicklungsprojekten werden in der Regel mehrere hundert Java-Klassen programmiert. Zur logischen Strukturierung der Klassen innerhalb eines Projektes sowie zur Aufteilung der Aufgaben im Entwicklungsteam wird die Menge aller Klassen in verschiedene **Pakete** unterteilt. Klassen mit ähnlichen Funktionen und Klassen, die stark voneinander abhängen, werden gleichen Paketen zugeordnet. Die Zuordnung zu Paketen hat Auswirkungen auf den Speicherort der Klasse und auf die Zugriffsberechtigungen auf Methoden anderer Klassen.

Pakete lassen sich mit Verzeichnissen des Dateisystems eines Computers vergleichen. Verzeichnisse dienen als Mittel zur Strukturierung von Dateien und speichern selber keine Informationen. Jede Datei wird in genau einem Verzeichnis gespeichert. Verzeichnisse können Unterverzeichnisse enthalten. Dateien können jedoch keine weiteren Dateien oder Verzeichnisse enthalten. Genau so verhält es sich mit Paketen und Klassen anstelle von Verzeichnissen und Dateien. Abbildung 37 zeigt beispielhaft eine Aufteilung von Klassen des Online-Shop-Projektes von Herrn Koch. Das Paket „onlineshop“ enthält drei weitere Pakete „nutzer“, „abwicklung“ und „waren“. In jedem dieser drei Pakete befinden sich Klassen.

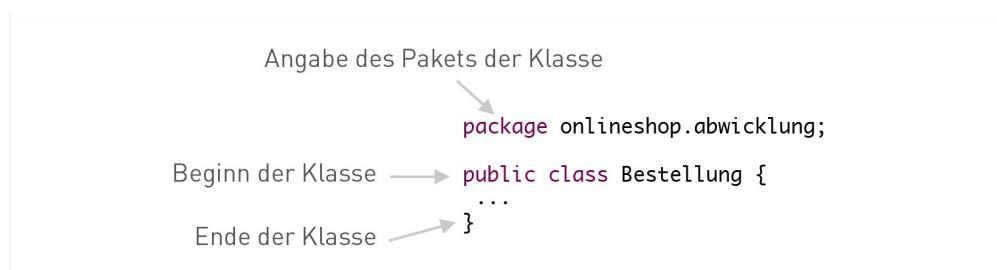
Abbildung 37: Pakete des Online-Shops



Quelle: erstellt im Auftrag der IU, 2013.

Die erste Quelltextzeile in einer Java-Klasse enthält immer die Angabe des Pakets, in dem sich die Klasse befindet. Abbildung 38 zeigt exemplarisch die Klasse Bestellung aus der Paketübersicht in Abbildung 37. Erst nach der Angabe des Pakets folgt die Deklaration der Klasse.

Abbildung 38: Deklaration des Paketes einer Klasse



Quelle: erstellt im Auftrag der IU, 2013.

Wie in Lernzyklus 3.2 bereits erläutert, wird jede Java-Klasse in einer eigenen Datei gespeichert. Der Speicherort der Datei ergibt sich aus dem Paket einer Klasse. Dabei wird für jedes Paket eines Programms ein Verzeichnis angelegt. Die Datei mit der Java-Klasse muss sich in dem Verzeichnis des Pakets befinden. Für die Klasse „Kunde“ im Paket „onlineshop.nutzer“ muss es eine Datei „onlineshop/nutzer/Kunde.java“ geben.

Klassennamen müssen innerhalb eines Pakets eindeutig sein. Der für die Java-Laufzeitumgebung vollständige Name einer Klasse (der sogenannte „qualifizierte Name“) ergibt sich aus dem Paket der Klasse und dem Namen der Klasse. Der Name unter dem die Klasse von außerhalb des Pakets ansprechbar ist, lautet „onlineshop.nutzer.Kunde“. So kann es gleichzeitig auch eine Klasse „Kunde“ in einem Bibliothekssystem geben, die den Namen „bibliothek.nutzer.Kunde“ trägt.

Bei der Einführung der Klassen, Methoden und Attribute in Lektion 3 wurden die Sichtbarkeitsmodifikatoren bereits eingesetzt, aber noch nicht erläutert. Mit Hilfe der **Sichtbarkeitsmodifikatoren** wird die Sichtbarkeit von Klassen, Attributen und Methoden und damit Zugriffsmöglichkeiten von anderen Klassen festgelegt. In Java gibt es, wie in Tabelle 17 beschrieben, vier Sichtbarkeitsmodifikatoren. Diese können auf die Elemente Klasse, Attribut und Methode angewendet werden.

Sichtbarkeitsmodifikatoren
Das sind Elemente von Java, mit denen die Sichtbarkeit von Attributen, Klassen und Methoden gegenüber anderen Klassen und Paketen eingestellt werden kann.

Tabelle 17: Sichtbarkeitsmodifikatoren

Sichtbarkeitsmodifikator	Beschreibung	Anwendbar auf	Beispiel
public	Das Element ist für alle Klassen des Programms sichtbar.	Klassen, Attribute, Methoden	public class Bestellung {} public void berechneSumme() {}
(nichts)	Das Element ist nur im gleichen Paket sichtbar.	Klassen, Attribute, Methoden	class Bestellung {} void berechne Summe() {}

Sichtbarkeits-modifikator	Beschreibung	Anwend-bar auf	Beispiel
protected	Das Element ist nur im gleichen Paket oder abgeleiteten Klassen (Details siehe Lektion 5) sichtbar.	Attribute, Methoden	<code>protected void berechneSumme() {} protected int anzahlArtikel;</code>
private	Das Element ist nur für Elemente der gleichen Klasse sichtbar.	Attribute, Methoden	<code>private void berechneSumme() {} private int anzahlArtikel;</code>

Quelle: erstellt im Auftrag der IU, 2013.

Obwohl Java dem Entwickler die Möglichkeit gibt, die Sichtbarkeiten von Attributen so festzulegen, dass ein direkter Zugriff durch andere Klassen erfolgen kann, sollten Attribute außer in gut begründeten Ausnahmefällen immer mit der Sichtbarkeit „private“ versehen werden.



ZUSAMMENFASSUNG

Mit den primitiven Datentypen können in Java einfache Standard-Datentypen eingesetzt werden, um Wahrheitswerte (`true, false`), ganze Zahlen (`1, 12, 13131`), Fließkommazahlen (`1.123, 21234.1232`) und einzelne Zeichen (`t, w, f, d`) zu speichern. Der Datentyp `String` zur Speicherung von Zeichenketten ist zwar ein sehr häufig gebrauchter, jedoch kein primitiver Datentyp.

Mit Variablen können konkrete Werte im Speicher der Anwendung abgelegt werden, um während der Abarbeitung einer Methode Zwischenergebnisse von Berechnungen kurzzeitig zu speichern.

Arithmetische Operatoren (`+, -, *, /, %, ++`) und logische Operatoren (`!, &&, ||, ^`) ermöglichen das Rechnen und Verändern von in Variablen und Attributen gespeicherten Werten. Vergleichsoperatoren (`==, !=, <, <=, >, >=, instanceof`) vergleichen Ausdrücke miteinander und liefern als Ergebnis `true` oder `false`. Bei den Prüfoperatoren (`==, !=`) bestimmen die Operanden, ob Werte von primitiven Datentypen oder Objektreferenzen von Referenzdatentypen verglichen werden.

Mit bedingten Verzweigungen (`if-else`) kann die Ausführung von Anweisungen abhängig von Bedingungen gemacht werden, wobei bedingte Verzweigungen auch verschachtelt werden können.

Schleifen ermöglichen die mehrfache Ausführung von gleichen Anweisungen hintereinander, wobei die Anzahl der Schleifendurchläufe von einer Einhaltung einer Schleifenbedingung bestimmt wird. Dabei können kopfgesteuerte Schleifen (`while`, `for`) und fußgesteuerte Schleifen (`do-while`) unterschieden werden.

Zur logischen Strukturierung der Klassen innerhalb eines Projektes wird die Menge aller Klassen in verschiedene Pakete unterteilt, wobei jede Klasse einem Paket zugeordnet ist und Pakete weitere Pakete enthalten können. Mit Hilfe der Sichtbarkeitsmodifikatoren (`private`, „`protected`, `public`) wird die Sichtbarkeit von Klassen, Attributen und Methoden und damit Zugriffsmöglichkeiten von anderen Klassen festgelegt.

LEKTION 5

VERERBUNG

LERNZIELE

Nach der Bearbeitung dieser Lektion werden Sie wissen, ...

- zu welchem Zweck Vererbung in objektorientierten Systemen eingesetzt wird.
- wie die Vererbung in UML Klassendiagrammen ausgedrückt wird.
- was die Begriffe Oberklasse und Unterklasse unterscheidet.
- wie die Vererbung zwischen Java-Klassen ausgedrückt wird.
- was es bedeutet, geerbte Attribute und Methoden zu überschreiben.

5. VERERBUNG

Aus der Praxis

Herr Koch hat nun bereits ein paar Klassen implementiert. Bei der Programmierung fällt ihm auf, dass er bei einigen der Klassen gleich lautende Attribute und Methoden notiert hat.

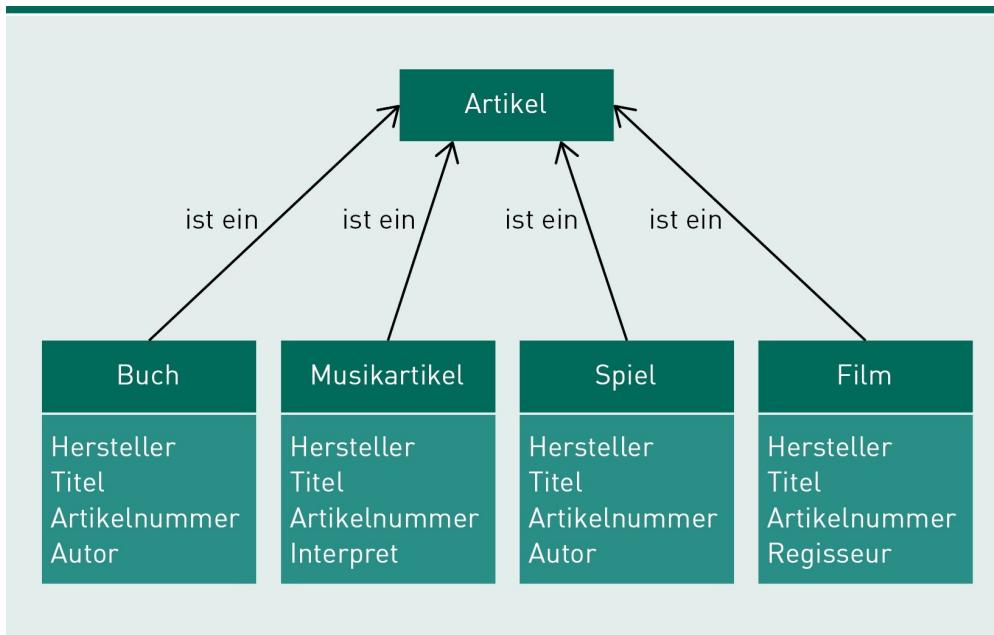
Abbildung 39: Implementierte Klassen

Buch	Musikartikel	Spiel	Film
Hersteller Titel Artikelnummer Autor	Hersteller Titel Artikelnummer Interpret	Hersteller Titel Artikelnummer Autor	Hersteller Titel Artikelnummer Regisseur

Quelle: erstellt im Auftrag der IU, 2013.

Er schaut in seinem Klassenmodell nach, ob an diesen Klassen etwas Besonderes ist. Er stellt fest, dass er bei ihnen jeweils „ist ein/e“-Beziehungen zur Klasse „Artikel“ eingezeichnet hat.

Abbildung 40: Beispielszenario ergänzt um Beziehungen



Quelle: erstellt im Auftrag der IU, 2013.

Herr Koch überlegt nun, wie diese beiden Tatsachen zusammenhängen, und fragt sich, wie er dieses Konzept genauer modellieren und anschließend programmieren kann.

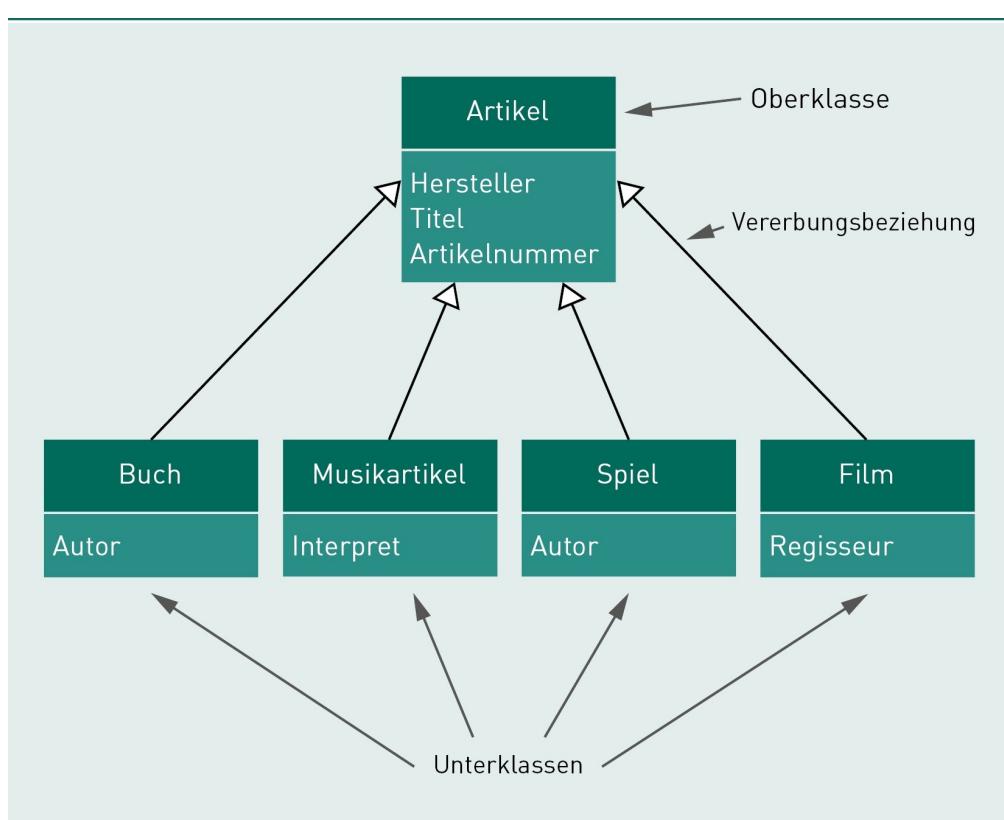
5.1 Modellierung von Vererbung im Klassendiagramm

Vergleicht man die Attribute der Klassen „Buch“, „Musikartikel“, „Spiel“ und „Film“ miteinander, so stellt man fest, dass einige davon in allen Klassen vorkommen und manche nur in einzelnen. Die Attribute „Hersteller“, „Titel“ und „Artikelnummer“ wurden für alle vier Klassen als wichtige Attribute identifiziert: Alle vier Klassen haben diese Attribute gemein. Das Attribut „Autor“ ist hingegen nur in den Klassen „Buch“ und „Spiel“ wichtig, die Attribute „Interpret“ und „Regisseur“ sind sogar nur in jeweils einer Klasse vertreten.

Die Attribute, die alle Klassen gemein haben, können in einer sogenannten Oberklasse zusammengefasst werden. Über die „ist ein/e“-Beziehung werden diese Attribute an die verbundenen Klassen weitergegeben und müssen dort dann nicht noch einmal aufgeführt werden.

In UML Klassendiagrammen wird für die „ist ein/e“-Beziehung eine eigene Notation verwendet. Man zieht zwischen zwei Klassen eine Linie und zeichnet an das Ende eine geschlossene nicht ausgefüllte Pfeilspitze.

Abbildung 41: Gleiche Attribute werden in der Oberklasse zusammengefasst



Quelle: erstellt im Auftrag der IU, 2013.

„ist ein/e“-Beziehung

Diese Beziehung drückt aus, dass eine Klasse eine spezielle Art einer anderen Klasse ist.

Die „**ist ein/e**“-Beziehung ist ein wichtiges Konzept in der Objektorientierung und wird als Vererbungsbeziehung bezeichnet. Man drückt mit dieser Beziehung aus, dass eine Klasse A wie eine andere Klasse B betrachtet werden kann. Im Beispiel von Herrn Koch bedeutet das, dass die Klassen „Buch“, „Musikartikel“, „Spiel“ und „Film“ alle als Artikel betrachtet werden können; Bücher, Musikartikel, Spiele und Filme sind Artikel. Dies ist der Unterschied zur normalen „kennt“-Beziehung zwischen Klassen: Die „ist ein/e“-Beziehung drückt aus, dass eine Klasse eine spezielle Art einer anderen Klasse ist.

Oberklasse

So wird eine Klasse bezeichnet, wenn andere Klassen von ihr ableiten.

Die Klasse „Artikel“ im Beispiel wird als **Oberklasse** bezeichnet, die anderen Klassen als „Unterklassen“. Für den Begriff Oberklasse werden häufig auch die Begriffe „Basisklasse“ oder „Superklasse“ verwendet, für den Begriff Unterklasse die Begriffe „abgeleitete Klasse“ oder „Subklasse“. Man sagt auch, dass eine Unterklasse von einer Oberklasse ableitet.

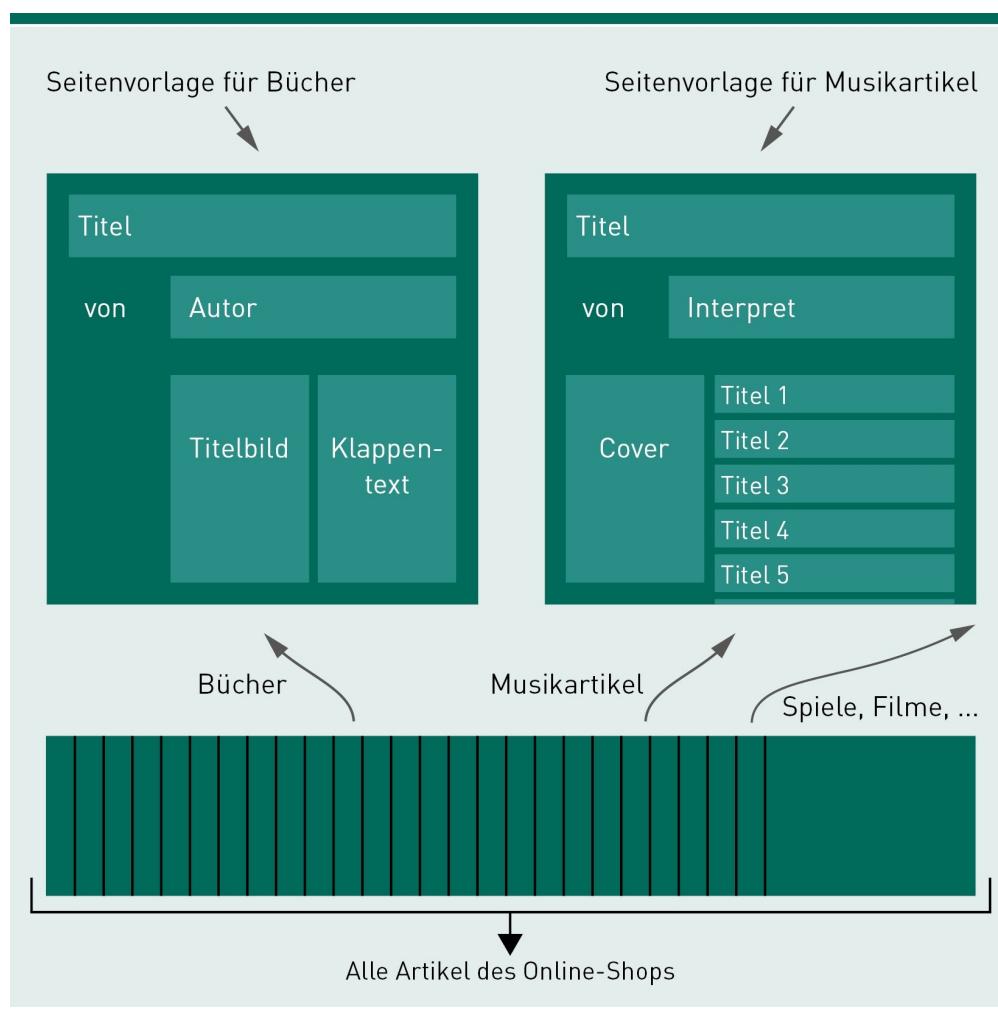
Durch Vererbung werden alle Attribute der Oberklasse an die Unterklassen weitergegeben. Ein „Buch“ besitzt also weiterhin einen „Hersteller“, da das „Buch“ ein „Artikel“ ist und dieser einen „Hersteller“ besitzt. Ein „Spiel“ hat auch weiterhin einen „Titel“, da es dieses Attribut von „Artikel“ erbt.

Die **Unterklassen** sind eine speziellere Art der Oberklasse. Sie besitzen alle Attribute der Oberklasse, definieren darüber hinaus aber noch weitere Attribute, die für die eigene Beschreibung wichtig sind. Für die Beschreibung eines Buches ist der Autor wichtig, für den Verkauf des Buches „als Artikel“ sind allerdings Hersteller, Titel und Artikelnummer ausreichend.

Die Vererbungsbeziehung wird also verwendet, um speziellere Klassen einer anderen Klasse zu modellieren. Im Beispiel wurde die Klasse „Artikel“ modelliert, um ganz generell über Elemente einer Bestellung zu sprechen. Dabei kommt es nicht auf die genauen Eigenschaften des tatsächlichen Artikels an. Allerdings ist es für die Darstellung auf der Webseite dann doch wichtig zu wissen, ob ein bestimmter Artikel nun ein Buch oder eine andere Artikelart ist: Auf der jeweiligen Artikelseite sind die spezifischen Eigenschaften eines Artikels wichtig, zum Beispiel der Autor eines Buches oder der Interpret eines Musikartikels.

Unterklasse
Diese leitet von einer Oberklasse ab und erbt deren Attribute und Methoden. Eine Unterklasse spezialisiert eine Oberklasse, indem sie mit zusätzlichen Attributen und Methoden weitere Funktionalität implementiert.

Abbildung 42: Unterschiedliche Seiten für verschiedene Arten von Artikeln



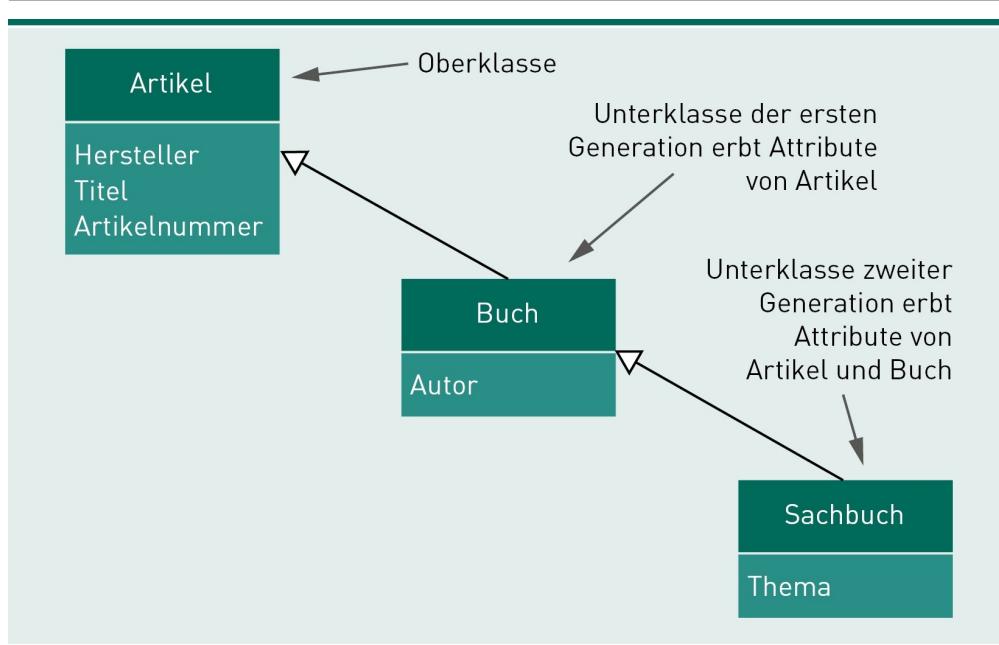
Quelle: erstellt im Auftrag der IU, 2013.

Vererbung
Das ermöglicht das Zusammenfassen von Gemeinsamkeiten und reduziert damit Wiederholungen im Entwurf.

Hier zeigt sich ganz genau, wie die Vererbung bei der Modellierung hilft: Man kann generell ausdrücken, dass jeder Artikel auf seiner eigenen Seite angezeigt werden kann. Wie genau diese Seite jedoch aussieht, unterscheidet sich abhängig von der tatsächlichen Art des Artikels. Für ein Buch werden andere Eigenschaften dargestellt als für ein Spiel. **Vererbung** ermöglicht somit, über Dinge im Allgemeinen und im Speziellen zu sprechen, ohne sich dabei unnötig zu wiederholen.

Die Vererbungsbeziehung ist transitiv, sie vererbt sich gewissermaßen mit. Eine weitere Spezialisierung der Klasse „Buch“ ist auch ein „Artikel“, da „Buch“ schon ein „Artikel“ ist. Im Beispiel in Abbildung 43 erbt „Sachbuch“ sowohl alle Attribute von „Artikel“ als auch von „Buch“.

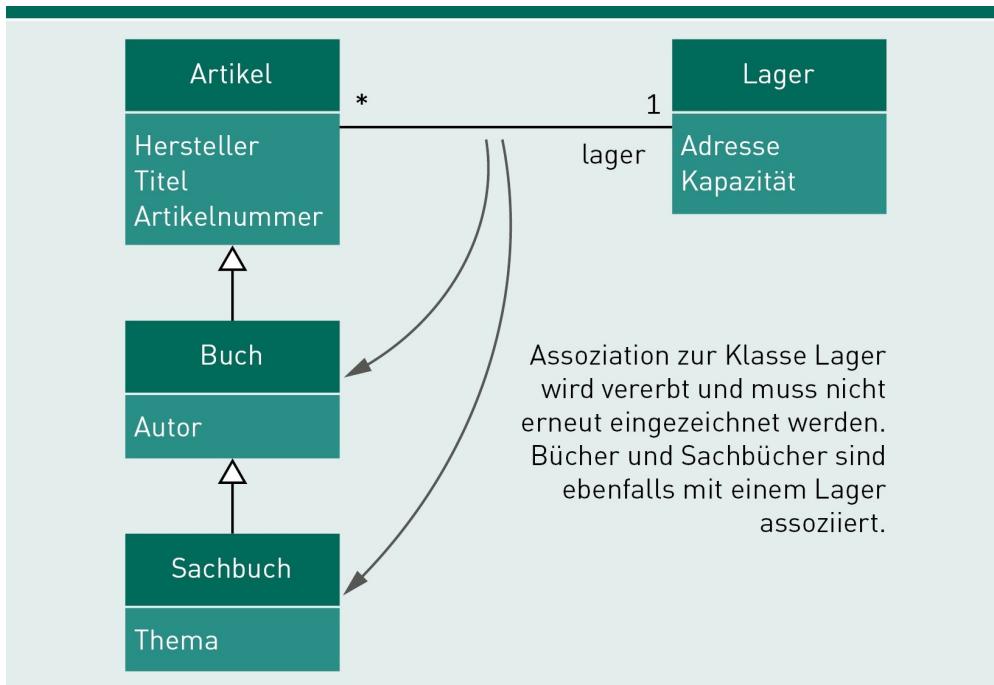
Abbildung 43: Transitive Eigenschaften der Vererbung



Quelle: erstellt im Auftrag der IU, 2013.

Durch Vererbung werden nicht nur die Attribute der Oberklasse an die Unterklassen weitergegeben. Es werden auch die Methoden und Assoziationen der Oberklasse vererbt. Auf diese Weise wird nicht nur die Gestalt der Oberklasse vererbt, sondern auch ihr Verhalten und ihre Verwendung.

Abbildung 44: Vererbung von Assoziationen



Quelle: erstellt im Auftrag der IU, 2013.

Objekte der Unterklassen besitzen dieselben Methoden wie Objekte der Oberklassen und lassen sich deshalb auf dieselbe Art und Weise ansprechen. Über die vererbten Assoziationen wird deutlich, wo sich Objekte der Unterklassen einsetzen lassen und auf welche Weise sie an diesen Stellen verwendet werden.

Mithilfe der Vererbung werden objektorientierte Systeme strukturiert. Durch das Definieren von Oberklassen werden Gemeinsamkeiten in bereits definierten Klassen zusammengefasst. Änderungen an der Oberklasse werden an alle abgeleiteten Klassen weitergegeben. Wird der Oberklasse ein Attribut hinzugefügt, so haben auch alle abgeleiteten Klassen das neue Attribut. Wird ein Attribut aus der Oberklasse entfernt, so ist es auch in den Unterklassen nicht mehr verfügbar. Durch Ableiten von einer bereits definierten Oberklasse wird eine neue Klasse für einen bestimmten Anwendungsfall spezialisiert. Die Attribute und Methoden der Oberklasse werden dabei nicht erneut wiederholt. Auf diese Weise wird Redundanz vermieden und die Übersichtlichkeit des Systementwurfs im Modell verbessert.

Durch das Zusammenfassen von Gemeinsamkeiten und das Ableiten von spezialisierten Klassen kann es im Modell dazu kommen, dass neue Klassen eingeführt werden, die in der Analysephase nicht identifiziert wurden. Dies ist so lange nicht schlimm, wie es der Übersichtlichkeit des Modells dient.

Es ist im Allgemeinen üblich, dass eine Unterkategorie von nur einer einzelnen Oberklasse ableitet. Bei mehreren Oberklassen würde eine Unterkategorie die Vereinigungsmenge der Attribute und Methoden aller Oberklassen erben. Dabei kann es dazu kommen, dass in

zwei unterschiedlichen Oberklassen dieselben Namen vergeben wurden, allerdings mit unterschiedlichen Bedeutungen. Dieser Konflikt führt allerdings häufig zu einer verringerten Verständlichkeit des Modells und zu Fehlern in der Modellierung. Darüber hinaus kann es sein, dass die Programmiersprache, in der das modellierte System umgesetzt wird, die Mehrfachvererbung nicht erlaubt. Java beispielsweise erlaubt sie nicht.

5.2 Programmieren von Vererbung in Java

Erweiterung

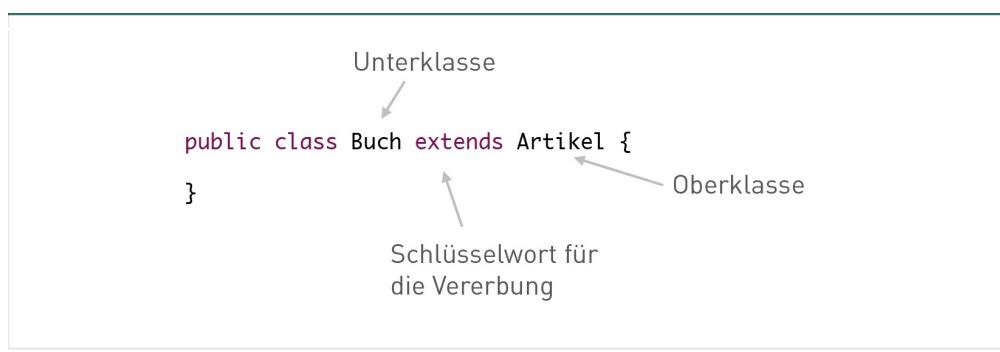
In Java spricht man anstelle von Vererbung oft von Erweiterung. Eine Unterklasse spezialisiert ihre Oberklasse durch neue und angepasste Funktionalität.

In Java realisiert man die Vererbung über ein neues Schlüsselwort bei der Deklaration einer Klasse. Anders als die schon vorgestellten Begriffe „Ableitung“ oder „Spezialisierung“, nennt Java das Konzept **Erweiterung**. Eine Unterklasse „erweitert“ eine Oberklasse, indem sie neue Attribute und Methoden definiert beziehungsweise modifiziert. Das Schlüsselwort lautet deswegen auch `extends`.

Die Vererbungsbeziehung in Java geht von der abgeleiteten Klasse aus, genau wie die Richtung des Assoziationspfeils in UML Klassendiagrammen. Um eine Vererbung in Java auszudrücken, gibt man in der Klassendeklaration der Unterklasse an, von welcher Oberklasse man ableitet.

Die Unterklasse erbt alle Attribute und Methoden der Oberklasse. Eine Ausnahme gilt für die Attribute und Methoden, die in der Oberklasse als `private` deklariert wurden. Diese sind auch in einer Unterklasse nicht verfügbar. Geerbte Attribute und Methoden behalten ansonsten aber ihre Sichtbarkeit: Als `public` deklarierte Attribute und Methoden bleiben also weiterhin `public`.

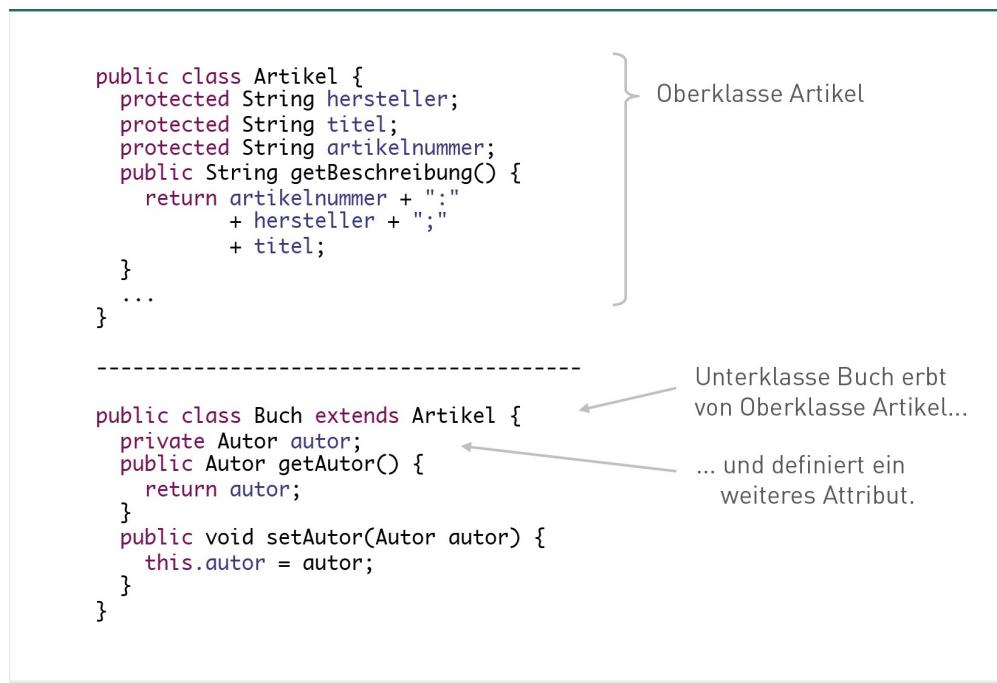
Abbildung 45: Vererbung in Java mittels `extends` in der Klassendeklaration



Quelle: erstellt im Auftrag der IU, 2013.

Abbildung 46 zeigt ein erweitertes Beispiel. Die Klasse „Buch“ wird von der Klasse „Artikel“ abgeleitet. Es wird ein weiteres Attribut namens `autor` definiert und entsprechende Getter- und Setter-Methoden dafür implementiert.

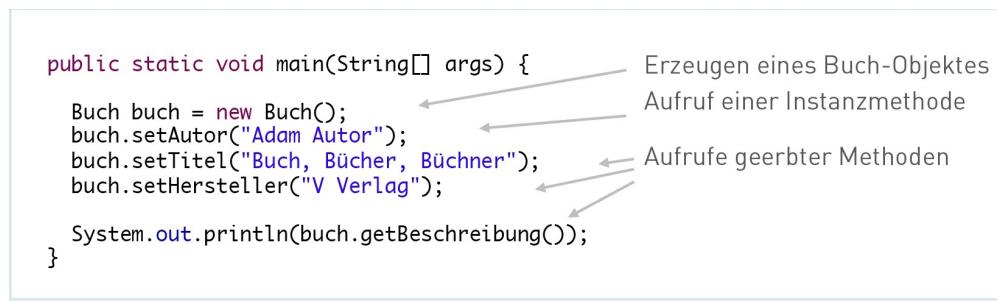
Abbildung 46: Ableitung der Klasse „Buch“ von der Klasse „Artikel“



Quelle: erstellt im Auftrag der IU, 2013.

Nach dem Instanziieren eines Buch-Objektes können die geerbten Attribute und Methoden wie normale Instanzattribute und -methoden verwendet werden. Die geerbten Attribute erhalten ihre Werte aus der Oberklasse. Beim Aufrufen von geerbten Methoden werden die Implementierungen aus der Oberklasse verwendet.

Abbildung 47: Zugriff auf geerbte Methoden



Quelle: erstellt im Auftrag der IU, 2013.

In Abbildung 47 wird ein Objekt `buch` erzeugt und sowohl das neue Attribut als auch die geerbten Attribute über die geerbten Getter- und Setter-Methoden gesetzt. Das Objekt `buch` kann also wie ein Artikel-Objekt verwendet werden, da es dieselben Attribute und Methoden wie ein Artikel hat.

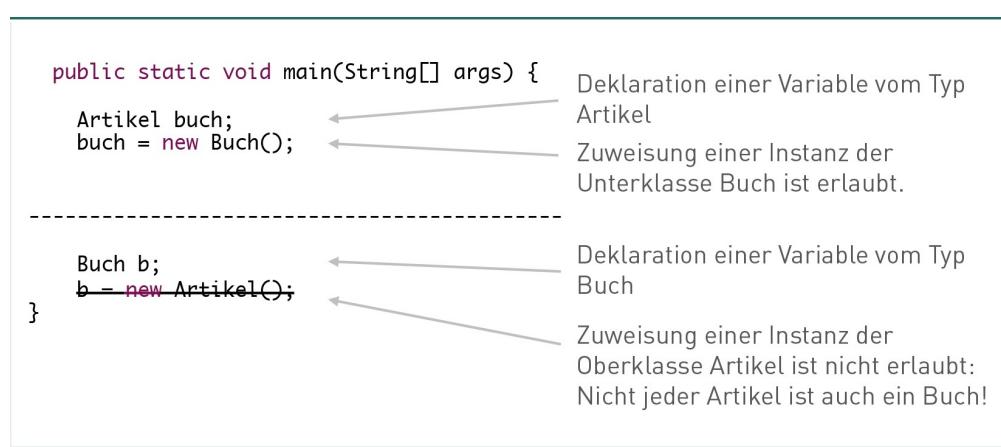
Zuweisungskompatibilität

Diese erlaubt es, einer Variable vom Typ einer Oberklasse auch Unterklassen-Objekte zuzuweisen.

Es ist auch möglich, eine Variable vom Typ Artikel zu deklarieren und ihr ein Buch-Objekt zuzuweisen. Dies wird **Zuweisungskompatibilität** genannt: Jedes Buch ist ein Artikel und kann als solcher behandelt werden.

Andersherum ist diese Kompatibilität nicht gegeben: Nicht jeder Artikel ist ein Buch. Ein erneuter Blick in das Klassendiagramm in Abbildung 41 macht den Unterschied klar. Die Klasse Buch deklariert ein neues Attribut, das nicht in Artikel vorhanden ist. Das instanzierte Artikel-Objekt darf also für den Fall, dass jemand auf das neue Attribut zugreifen will, nicht wie ein Buch behandelt werden.

Abbildung 48: Zuweisungskompatibilität



Quelle: erstellt im Auftrag der IU, 2013.

Es muss bei der Zuweisungskompatibilität noch auf eine andere Sache geachtet werden: Der Typ einer Variable entscheidet, welche Attribute und Methoden aufgerufen werden können. In der folgenden Abbildung 49 wird eine Variable artikel vom Typ Artikel deklariert und ihr eine neue Instanz von Buch zugewiesen. Auf der Variable artikel können danach nur die Methoden der Klasse „Artikel“ aufgerufen werden. Der Aufruf von setAutor wird vom Compiler als Fehler angezeigt, da die Methode in der Klasse „Artikel“ nicht definiert wurde.

Abbildung 49: Verfügbarkeit von Attributen und Methoden abhängig vom Typ der Variable

Abbildung 49: Verfügbarkeit von Attributen und Methoden abhängig vom Typ der Variable

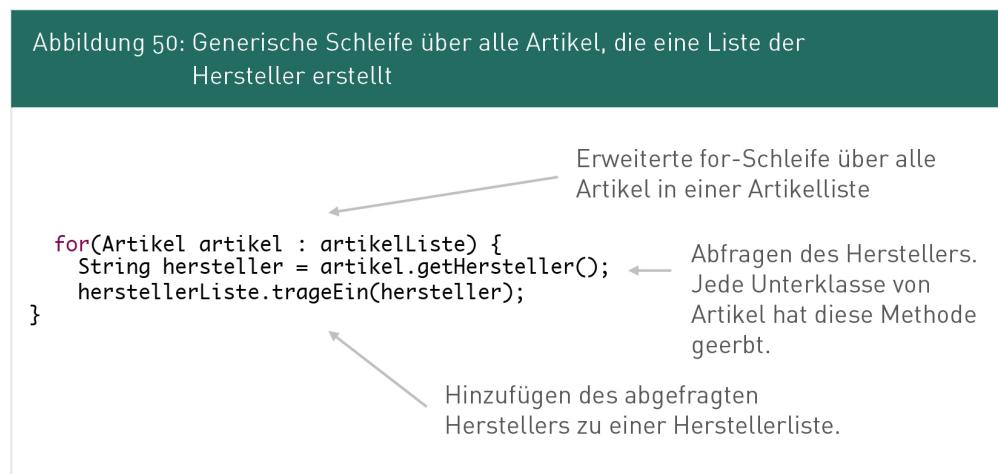
```
public static void main(String[] args) {  
    Artikel artikel; ← Deklaration einer Variable vom Typ Artikel  
    artikel = new Buch(); ← Zuweisung einer Instanz der Unterklasse Buch  
  
    artikel.setTitel("Buch, Bücher, Büchner");  
    artikel.setHersteller("V Verlag"); } ] Aufrufe von Methoden, die in der Klasse Artikel definiert sind  
    System.out.println(artikel.getBeschreibung());  
  
    artikel.setAutor ("Adam Autor"); ← Aufruf der Methode aus der Unterklasse ist aufgrund des Typs der Variable buch nicht erlaubt!  
}
```

Quelle: erstellt im Auftrag der IU, 2013.

Die Zuweisungskompatibilität wird verwendet, um in großen Softwaresystemen eine möglichst lose Kopplung der einzelnen Klassen untereinander zu gewährleisten. Soll beispielsweise eine Liste der Hersteller über die gesamte Artikelmenge erstellt werden, ist es egal, ob es sich bei den einzelnen Artikeln um Bücher, Spiele oder Filme handelt. Die Lösung kann also unabhängig von den konkreten Ausprägungen implementiert werden, indem man sich auf die in der Klasse „Artikel“ deklarierten Funktionalitäten konzentriert. So können im Verlauf der Implementierung noch weitere Unterklassen von Artikel erstellt werden – zum Beispiel für Spielzeug –, ohne dass die Schleife, die die Herstellerliste zusammenstellt, dafür angepasst werden muss.

In Abbildung 50 ist eine mögliche Variante einer Schleife dargestellt, mit der von allen Artikeln aus einer Artikelliste der Hersteller ermittelt und zu einer Herstellerliste hinzugefügt wird. Die Schleife funktioniert unabhängig davon, ob in der Artikelliste nur Bücher, Filme etc. oder eine bunte Mischung aus Objekten der verschiedenen Unterklassen von Artikel sind.

Abbildung 50: Generische Schleife über alle Artikel, die eine Liste der Hersteller erstellt



Quelle: erstellt im Auftrag der IU, 2013.

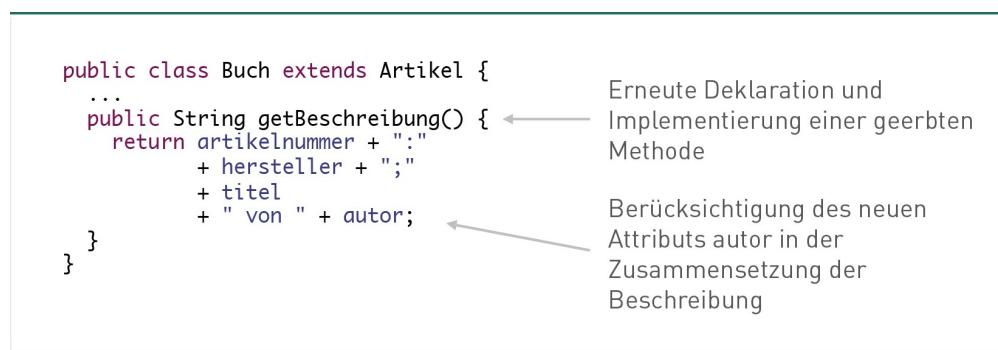
Unterklassen erben zwar die Attribute und Methoden der Oberklasse, sind aber nicht zwingend an die Implementierung gebunden. So gibt zum Beispiel die in der Klasse „Artikel“ definierte Methode `getBeschreibung` für die Unterklassen eine unvollständige Beschreibung aus. Bei Büchern fehlt der Autor, bei Filmen der Regisseur usw. Eine korrekte Implementierung von `getBeschreibung` müsste diese Attribute mit ausgeben. Das kann aber nur in den Unterklassen realisiert werden. Die Klasse „Artikel“ weiß nichts von den zusätzlichen Attributen der Unterklassen, die noch mit ausgegeben werden müssten. Die Unterklassen müssen also die Implementierung der geerbten Methode `getBeschreibung` für ihre eigenen Zwecke anpassen. Dieser Vorgang wird **Überschreiben** genannt.

Überschreiben

Das erneute Implementieren einer geerbten Methode wird Überschreiben genannt.

Methoden können in Unterklassen überschrieben werden, indem man eine Methode mit derselben Signatur erstellt und einen neuen Methodenkörper implementiert.

Abbildung 51: Überschreiben einer geerbten Methode



Quelle: erstellt im Auftrag der IU, 2013.

Erzeugt man eine Instanz der Unterklasse und greift auf geerbte Attribute oder Methoden zu, so werden die überschriebenen Varianten ausgewählt. Die Instanz bestimmt, welche Implementierung ausgeführt wird. In Abbildung 52 wird eine Instanz der Klasse „Buch“ erzeugt und der Variable `buch` zugewiesen. Nach dem Setzen der Attribute wird die überschriebene Methode `getBeschreibung` aufgerufen. Diese gibt eine vollständige Beschreibung aus, mit Berücksichtigung des Autors.

Abbildung 52: Aufruf einer überschriebenen Methode

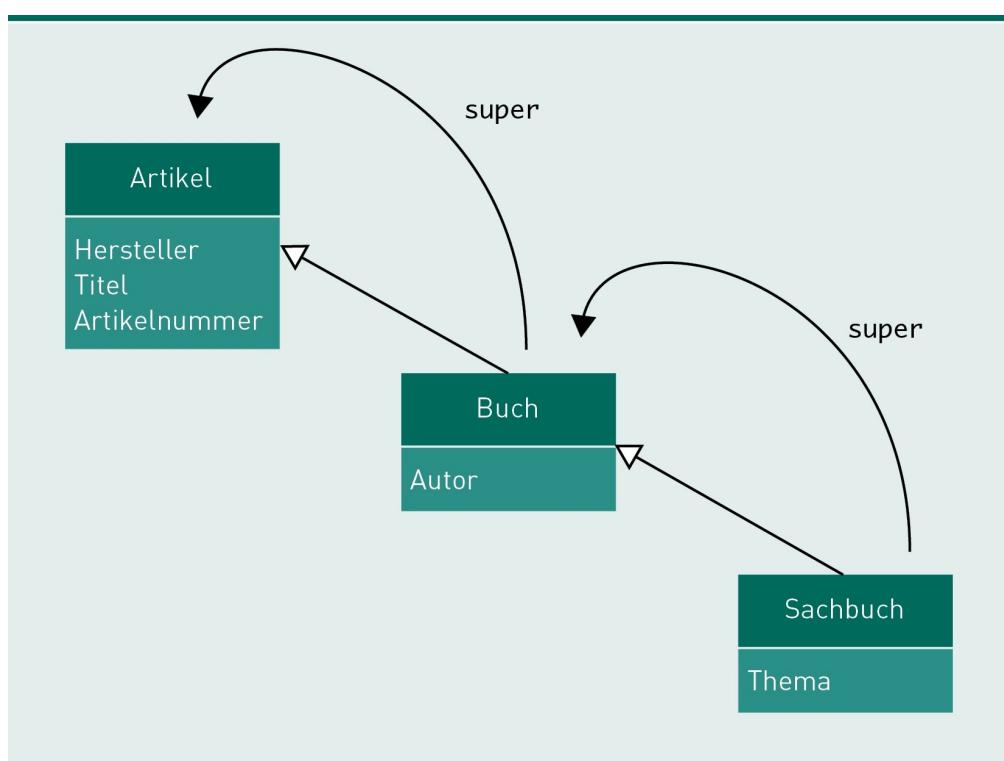
Abbildung 52: Aufruf einer überschriebenen Methode

```
public static void main(String[] args) {  
    Buch buch = new Buch();  
    buch.setArtikelnummer("987-2651-3891-56");  
    buch.setAutor("Adam Autor");  
    buch.setTitel("Buch, Bücher, Büchner");  
    buch.setHersteller("V Verlag");  
    System.out.println(buch.getBeschreibung()); ← Ruft die überschriebene  
} Methode auf und gibt die  
vollständige Beschreibung aus  
  
987-2651-3891-56:V Verlag;Buch, Bücher, Büchner von Adam Autor
```

Quelle: erstellt im Auftrag der IU, 2013.

Beim Überschreiben von Attributen und Methoden wird die ursprüngliche Implementierung aus der Oberklasse von der neuen Implementierung überlagert. Es ist aber möglich, auf die bestehende Implementierung in der Oberklasse zuzugreifen. Diese Möglichkeit bietet das Java-Schlüsselwort `super`. Mithilfe von `super` wird die Oberklasse angesprochen und es kann auf deren Attribute und Methoden zugegriffen werden. Da jede UnterkLASSE nur eine Oberklasse haben kann, ist mit `super` genau definiert, welche Klasse gemeint ist. `super` geht eine Ebene in der Vererbungshierarchie nach oben und wählt die dortige Implementierung aus.

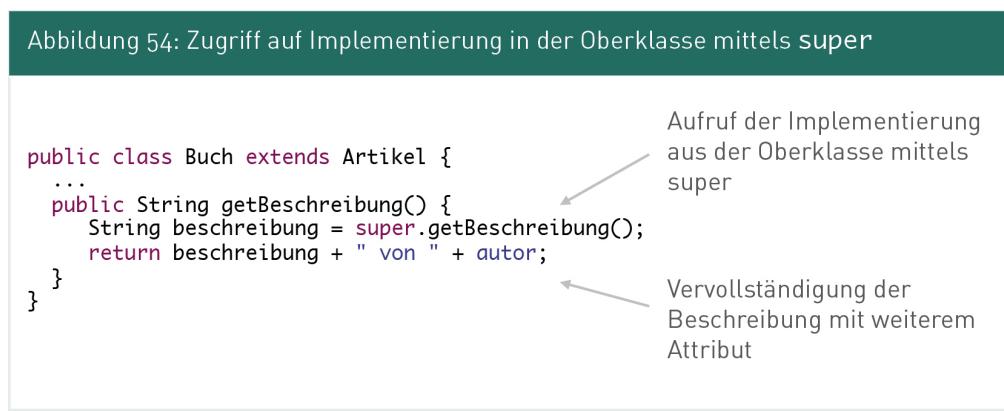
Abbildung 53: super verweist auf die Oberklasse



Quelle: erstellt im Auftrag der IU, 2013.

Mittels **super** können Implementierungen aus der Oberklasse erweitert werden. Zuerst ruft man die bereits vorhandene Implementierung auf und führt dann zusätzliche Anweisungen aus, die für die Funktionalität in der Unterklasse wichtig sind. Die Implementierung der überschriebenen Methode in Abbildung 52 lässt sich zum Beispiel auch wie in Abbildung 54 implementieren. Zuerst ruft man mittels **super** die Implementierung aus der Oberklasse auf und fügt dann dem Ergebnis weitere Attribute an.

Abbildung 54: Zugriff auf Implementierung in der Oberklasse mittels super



Quelle: erstellt im Auftrag der IU, 2013.



ZUSAMMENFASSUNG

Das Konzept der Vererbung erlaubt es, die Funktionalität von bereits vorhandenen Klassen in neuen Klassen zu verwenden und für spezielle Anwendungsfälle anzupassen. Bei der Vererbung leitet eine Unterklasse von einer Oberklasse ab. Die Unterklasse erbt die sichtbaren Attribute und Methoden der Oberklasse.

In UML-Klassendiagrammen wird die Vererbungsbeziehung durch eine Assoziationslinie mit einer leeren Pfeilspitze am Ende in Richtung der Oberklasse ausgedrückt. In den Unterklassen werden nur die neuen Attribute und Methoden notiert. Die geerbten Attribute und Methoden werden nicht wiederholt. Assoziationen, die die Oberklasse besitzt, vererben sich ebenso an die Unterklassen. Die Vererbungsbeziehung ist transitiv und ermöglicht so die Weitergabe von Attributen und Methoden über mehrere Hierarchiestufen.

In Java wird die Vererbung durch das Schlüsselwort `extends` ausgedrückt. In den Instanzen der abgeleiteten Klassen sind dieselben Attribute und Methoden wie in den Instanzen der Oberklasse verfügbar und können genau wie eigene Attribute und Methoden verwendet werden. Aus diesem Grund können Instanzen von abgeleiteten Klassen auch Variablen zugewiesen werden, die vom Typ der Oberklasse deklariert wurden. Dies wird Zuweisungskompatibilität genannt.

Geerbte Attribute und Methoden können in der Unterklasse neu definiert werden. Dies wird Überschreiben genannt. Über das Schlüsselwort `super` greift man auf die überschriebene Version der Attribute und Methoden der Oberklasse zu.

LEKTION 6

WICHTIGE OBJEKTORIENTIERTE KONZEPTE

LERNZIELE

Nach der Bearbeitung dieser Lektion werden Sie wissen, ...

- wie Sie mittels abstrakter Klassen einschränken können, von welchen Klassen Instanzen erzeugt werden dürfen.
- wie Sie mittels abstrakter Methoden vorgeben, welche Methoden Unterklassen zu implementieren haben.
- was der Begriff Polymorphie bedeutet und welche Vorteile er für objektorientierte Systeme bietet.
- wie Sie statische Attribute definieren, deren Werte für alle Instanzen einer Klasse gleich sind.
- wie Sie statische Methoden von Klassen definieren, die aufgerufen werden können, ohne Objektinstanzen erzeugen zu müssen.

6. WICHTIGE OBJEKTORIENTIERTE KONZEPTE

Aus der Praxis

Herr Koch bespricht mit seiner Auftraggeberin Frau Lange den Fortschritt. Er berichtet, wie er die Klassen „Buch“, „Musikartikel“, „Spiel“ und „Film“ als Unterklassen von „Artikel“ implementiert hat. Frau Lange zeigt besonderes Interesse, als ihr Herr Koch erklärt, wie die Unterklassen eine Methode der Oberklasse überschreiben, um eine aussagekräftigere Beschreibung von sich selbst zu erzeugen. Frau Lange kommt auf die Idee, diese Funktionalität für ein besonderes Feature zu erweitern: Sie möchte, dass Kunden einen Artikelwunsch twittern können.

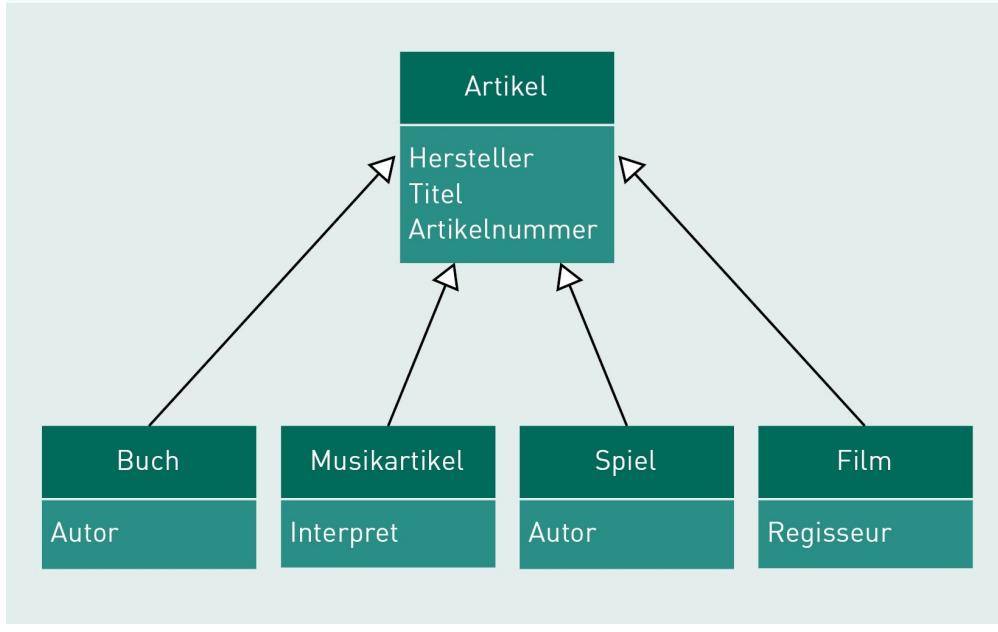
Herr Koch stimmt zu und überlegt, wie er diese Anforderung in seinem System unterbringt. Da er für eine solche Nachricht nur eine begrenzte Anzahl an Zeichen zur Verfügung hat, sollten die jeweiligen Unterklassen eine besonders kompakte Beschreibung generieren. Herr Koch könnte wieder eine entsprechende Methode in der Klasse „Artikel“ implementieren und in den Unterklassen überschreiben. Allerdings muss jede Unterklasse zwingendermaßen die Methode überschreiben, um wirklich nur die wesentlichsten Elemente zusammenzufassen. Herr Koch überlegt, wie er erzwingen kann, dass Unterklassen eine bestimmte Methode überschreiben.

Im Zuge seiner Überlegungen kommt er zu dem Schluss, dass es wegen des neuen Features nicht mehr erlaubt sein sollte, Objekte von der Klasse „Artikel“ zu erzeugen. Jeder Artikel soll sich über die Twitter-Methode aussagekräftig beschreiben. Herrn Koch fällt aber auf, dass so etwas für ein Artikel-Objekt, ohne eine konkrete Zuordnung, welche Art von Artikel es denn ist, keinen Sinn ergibt. Er denkt, dass es doch möglich sein sollte, die Erzeugung von Objekten bestimmter Klassen zu verbieten.

6.1 Abstrakte Klassen

In der Modellierung verwendet Herr Koch die Klasse „Artikel“, um die Klassen „Buch“, „Musikartikel“, „Spiel“ und „Film“ zusammenzufassen. Wie er richtig beobachtet hat, ergibt es wenig Sinn, eine Instanz der Klasse „Artikel“ zu erzeugen. Ein Artikel-Objekt hat zwar einen Titel, einen Hersteller und eine Artikelnummer, aber dadurch ist noch nicht festgelegt, um welche Art von Artikel es sich handelt. Ohne diese Information wird es schwer, den Artikel im System angemessen zu verwenden. So ist beispielsweise nicht definiert, wie die Artikelseite im Onlinesystem für ein Artikel-Objekt aussehen soll. Auch für eine aussagekräftige Beschreibung ist die Art des Artikels ein wesentlicher Bestandteil.

Abbildung 55: Oberklasse „Artikel“ und deren Unterklassen



Quelle: erstellt im Auftrag der IU, 2013.

Die Klasse „Artikel“ dient also im Beispiel vornehmlich der Modellierung und Strukturierung des entstehenden Systems. Es sollen aber keine konkreten Instanzen davon erzeugt werden. Es soll nur möglich sein, die verschiedenen Arten von Artikeln auf eine konsistente Art und Weise zu verwenden.

Die Klasse „Artikel“ wird auch **abstrakte Klasse** genannt. Damit wird explizit ausgedrückt, dass keine Instanzen dieser Klasse erzeugt werden dürfen. Spezifische Details werden in den abgeleiteten Unterklassen implementiert.

In der UML werden abstrakte Klassen mit dem Wort `abstract` in geschweiften Klammern gekennzeichnet. Alternativ kann der Klassename kursiv gesetzt werden.

Abbildung 56: Kennzeichnung der Klasse „Artikel“ als abstrakte Klasse, zwei Varianten



Quelle: erstellt im Auftrag der IU, 2013.

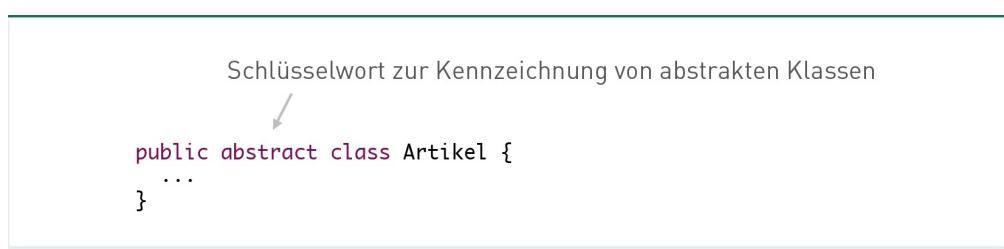
Abstrakte Klasse
Das ist eine Klasse, von der keine Instanzen erzeugt werden können. Sie fasst Gemeinsamkeiten zusammen und gibt eine Schnittstelle vor, die von ihren abgeleiteten Klassen erfüllt werden muss.

Für die Implementierung abstrakter Klassen in Java verwendet man das Schlüsselwort `abstract`. Sobald eine Klasse als `abstract` gekennzeichnet wurde, dürfen keine Instanzen von ihr erzeugt werden. Ansonsten sind sie aber wie normale Klassen aufgebaut:

Sie können Attribute definieren und ihnen Werte zuweisen sowie Methoden definieren und implementieren. Abgeleitete Klassen erben diese Attribute und Methoden wie gehabt.

In Abbildung 57 wird die Klasse „Artikel“ durch Hinzufügen des Schlüsselwortes `abstract` in der Klassendeklaration zu einer abstrakten Klasse gemacht. An den sonstigen Deklarationen muss nichts geändert werden. Versucht man aber nun, Instanzen dieser Klasse zu erzeugen, so meldet der Compiler einen Fehler.

Abbildung 57: Kennzeichnung einer abstrakten Klasse in Java



Quelle: erstellt im Auftrag der IU, 2013.

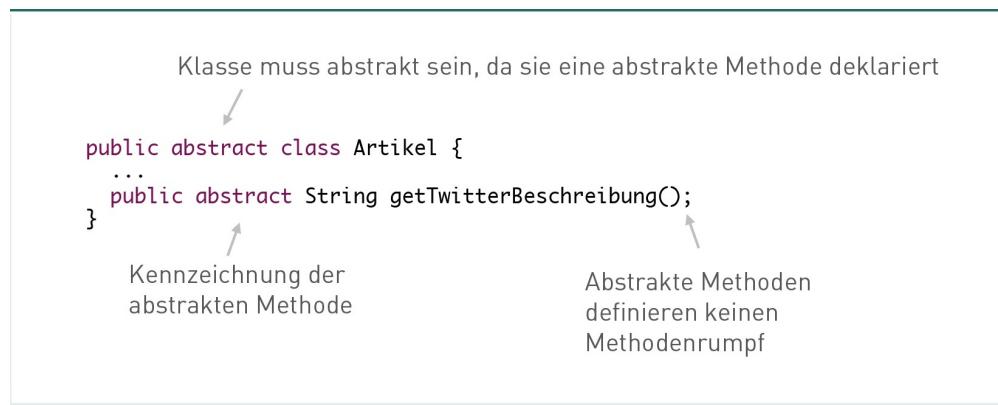
Mit dieser Maßnahme hat Herr Koch erreicht, dass in seinem System keine Artikel-Objekte mehr erzeugt werden dürfen.

Es können aber nicht nur Klassen, sondern auch Methoden einer Klasse als `abstract` gekennzeichnet werden. Eine **abstrakte Methode** hat eine Signatur, aber keine Implementierung. So wie es von abstrakten Klassen keine konkreten Instanzen geben darf, dürfen abstrakte Methoden keine Methodenrumpfe definieren. Das hat zur Folge, dass eine Klasse mit einer oder mehreren abstrakten Methoden nicht vollständig ist. Definiert deshalb eine Klasse mindestens eine abstrakte Methode, so muss sie in der Klassendeklaration ebenfalls als `abstract` gekennzeichnet werden. Die abstrakte Methode muss dann in einer Unterklasse durch Überschreiben implementiert werden.

Abstrakte Methoden müssen zwingend von konkreten Unterklassen implementiert werden. Herr Koch kann dies benutzen, um für die Unterklassen von „Artikel“ die Implementierung der Twitter-Methode vorzuschreiben. In Abbildung 58 wird die Klasse „Artikel“ um die Methode `getTwitterBeschreibung()` erweitert. Diese Methode wird als `abstract` markiert und der Methodenrumpf weggelassen.

Abstrakte Methode
Diese hat eine Signatur, aber keinen Methodenrumpf. Sie gibt vor, welche Funktionalität in Unterklassen implementiert werden muss.

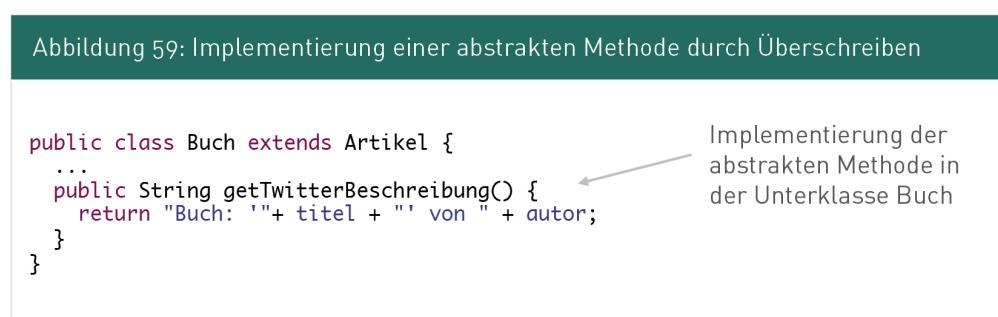
Abbildung 58: Kennzeichnung einer abstrakten Methode



Quelle: erstellt im Auftrag der IU, 2013.

In einer Unterklasse von „Artikel“ muss diese Methode nun implementiert werden. Das funktioniert genauso wie das Überschreiben einer geerbten Methode. In Abbildung 59 wird die abstrakte Methode in der Klasse „Buch“ implementiert.

Abbildung 59: Implementierung einer abstrakten Methode durch Überschreiben



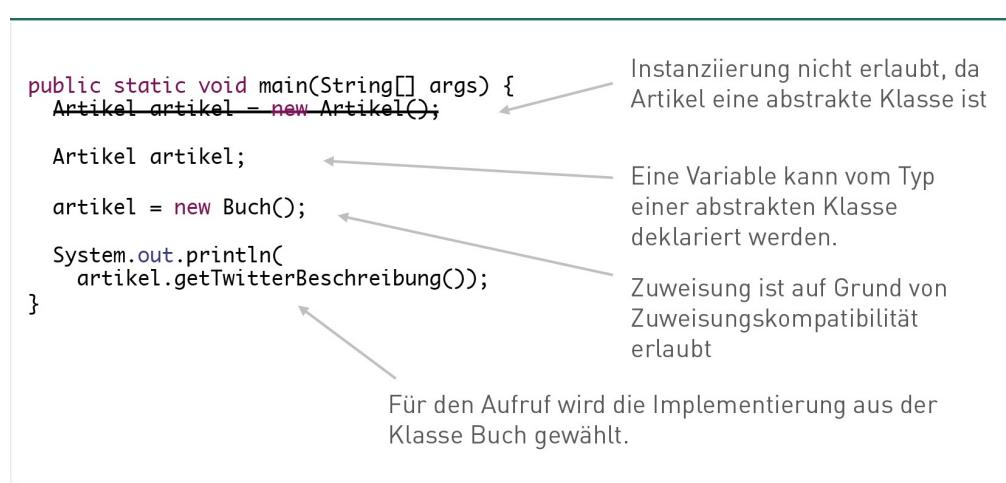
Quelle: erstellt im Auftrag der IU, 2013.

Das Implementieren einer geerbten abstrakten Methode in einer Unterklasse wird vom Compiler erzwungen. Die einzige Möglichkeit, eine abstrakte Methode nicht in einer direkten Unterklasse zu implementieren, ist, sie dort wieder als `abstract` zu kennzeichnen. Das hat allerdings zur Folge, dass diese Unterklasse zu einer abstrakten Klasse wird und so die Verantwortung der Implementierung an weitere Unterklassen delegiert wird.

Es ist möglich, Variablen vom Typ einer abstrakten Klasse zu deklarieren. Da aber keine Objekte von dieser abstrakten Klasse erzeugt werden können, muss der zugewiesene Wert der so deklarierten Variable zwingend ein Objekt einer abgeleiteten Klasse sein.

Im Beispiel in Abbildung 60 wird zuerst versucht, ein Objekt der Klasse „Artikel“ mit `new Artikel()` zu erzeugen. Das wird allerdings vom Java-Compiler nicht erlaubt. Danach wird eine Variable vom Typ „Artikel“ deklariert und ihr eine neue Instanz der Klasse „Buch“ zugewiesen. Das ist erlaubt, da durch die Vererbungsbeziehung zwischen „Buch“ und „Artikel“ die Zuweisungskompatibilität gegeben ist.

Abbildung 60: Deklaration von Variablen des Typs einer abstrakten Klasse



Quelle: erstellt im Auftrag der IU, 2013.

Mithilfe der Konzepte abstrakter Klassen und Methoden hat Herr Koch erreicht, die Erzeugung von konkreten Artikel-Objekten zu verhindern und die Implementierung einer Twitter-Beschreibung für alle Unterklassen verpflichtend zu machen.

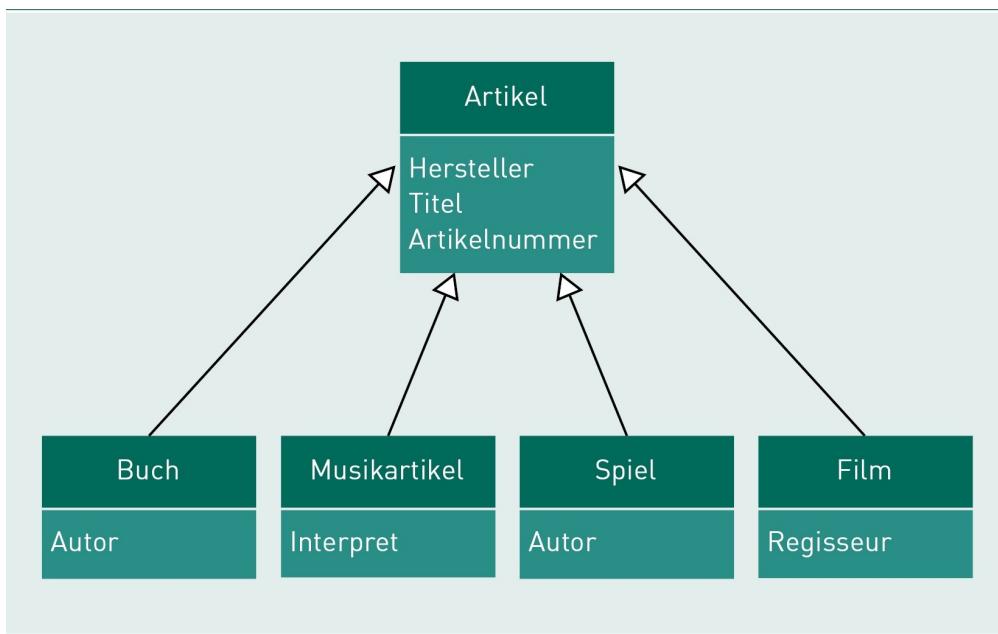
6.2 Polymorphie

Herr Koch freut sich, dass er in seinem System mit Vererbung redundante Definitionen vermeiden und mit abstrakten Methoden die Implementierung von wichtigen Methoden vorschreiben kann. Spezialisierte Funktionalität wie die Erzeugung von Kurzbeschreibungen implementiert er dort, wo sie hingehört, nämlich in den speziellen Unterklassen. Er fragt sich aber, welches objektorientierte Konzept dahinter steckt, dass stets auch die korrekte Implementierung ausgeführt wird, obwohl er seine Variablen manchmal einfach nur vom Typ „Artikel“ deklariert.

Polymorphie
Das bedeutet, dass man einer Variable Objekte unterschiedlicher Klassen zuweisen kann, falls die Klassen in einer Vererbungsbeziehung zueinander stehen. Das ermöglicht es, über ein und dieselbe Variable verschiedene Methodenimplementierungen aufzurufen, je nachdem, welches Objekt der Variable zur Laufzeit zugewiesen ist.

Das objektorientierte Konzept hinter diesem Phänomen nennt sich Polymorphie. **Polymorphie** bedeutet Vielgestalt. Sie bezeichnet in objektorientierten Systemen die Tatsache, dass eine Variable, die vom Typ einer bestimmten Klasse deklariert ist, auch Instanzen der Unterklassen annehmen kann. Eine Variable kann sich also in einer Vielzahl an konkreten Gestalten zeigen. Abbildung 61 zeigt anhand der Vererbungsbeziehungen, dass sich hinter einer Variablen der Klasse „Artikel“ auch Instanzen von „Buch“, „Musikartikel“, „Spiel“ und „Film“ verbergen können.

Abbildung 61: Die unterschiedlichen Arten von Artikeln in der Klassenhierarchie

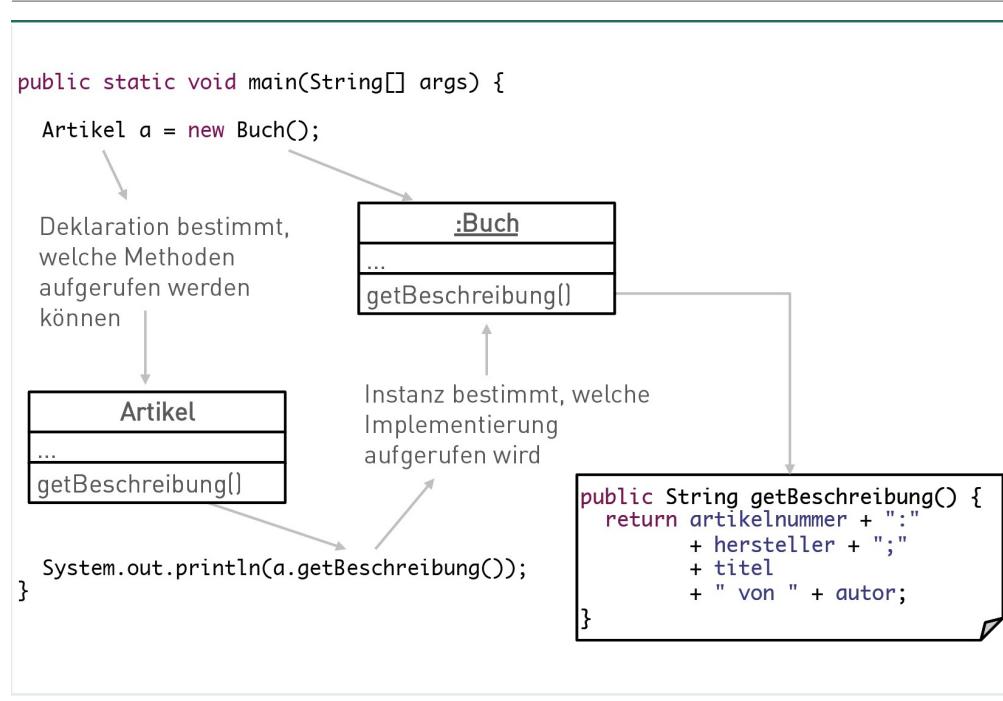


Quelle: erstellt im Auftrag der IU, 2013.

Die einzelnen Unterklassen können Methoden überschreiben, um so das Verhalten zu spezialisieren. Im Beispielsystem wird von Artikeln im Allgemeinen gesprochen und ganz bestimmte Anforderungen – wie beispielsweise die Ausgabe eines Beschreibungstextes – an sie gestellt. Die Umsetzung dieser Anforderung unterscheidet sich aber von Unterkasse zu Unterkasse. Polymorphie erlaubt es, alle Instanzen der Unterklassen als Artikel zu behandeln und trotzdem auf die jeweilige spezielle Implementierung der Methoden zuzugreifen.

So wie der Typ einer Variable darüber entscheidet, welche Methoden aufgerufen werden können, entscheidet die Instanz einer Klasse, welche Implementierung dieser Methoden ausgeführt wird. Abbildung 62 verdeutlicht diesen Vorgang am Beispiel des Aufrufes der Methode `getBeschreibung` bei einer Variable vom Typ „Artikel“.

Abbildung 62: Auswahl der Implementierung zur Laufzeit



Quelle: erstellt im Auftrag der IU, 2013.

Das Konzept der Polymorphie ermöglicht es, Methoden von Objekten aufzurufen, ohne wissen zu müssen, zu welcher konkreten Unterklasse diese Objekte gehören. Es zählt, dass die Objekte die Funktionalität implementieren, die man benötigt.

Möglich ist aber auch, zur Laufzeit zu überprüfen, zu welcher Klasse eine Instanz gehört. Mit dem `instanceof`-Operator kann ein Test auf Klassenzugehörigkeit durchgeführt werden. Dadurch werden allerdings die Vorteile der Polymorphie aufgehoben. Man testet nun auf eine ganz spezifische Klassenzugehörigkeit und kann nicht mehr mit beliebigen Unterklassen-Instanzen arbeiten.

Im folgenden Beispiel wird der `instanceof`-Operator verwendet, um das übergebene Objekt auf die Zugehörigkeit zur Klasse „Buch“ zu testen. Ist dies der Fall, so können spezielle Vorbereitungen für das Verarbeiten eines Buch-Objektes getroffen werden. Es lohnt sich aber, darüber nachzudenken, ob diese Vorbereitungen auch vom Buch-Objekt selber getroffen werden können, um die Abfrage nach der konkreten Klasse zu vermeiden.

Abbildung 63: Verwendung des instanceof-Operator zum Testen auf Klassenzugehörigkeit

```
public static void main(String[] args) {  
    ...  
    if(artikel instanceof Buch) {  
        ...  
    }  
}
```

Test auf Zugehörigkeit zur Klasse Buch

Quelle: erstellt im Auftrag der IU, 2013.

Polymorphie ermöglicht es Herrn Koch, die verschiedenen Arten von Artikeln in seinem System über eine gemeinsame Schnittstelle anzusprechen und gleichzeitig auf deren spezielle Implementierungen zuzugreifen. So bekommt er von jedem Artikel eine passende Beschreibung, wenn er die Methode `getBeschreibung` auruft.

6.3 Statische Attribute und Methoden

Bei der Implementierung seines Systems fällt Herrn Koch auf, dass er bei der Formatierung der Beschreibungstexte immer dasselbe Zeichen für die Trennung der Attributwerte verwendet. Da er befürchtet, das Zeichen später wieder verändern zu müssen, möchte er gerne eine Variable dafür einführen. Diese Variable soll für alle Artikel gelten, weshalb Herr Koch zuerst überlegt, sie als Attribut der Klasse „Artikel“ zu deklarieren. Dabei bemerkt er aber, dass er in diesem Fall erst ein Objekt von „Artikel“ erzeugen müsste, um auf das Attribut zuzugreifen. Er fragt sich, ob es eine bessere Alternative gibt.

Klassenvariablen werden auch statische Attribute genannt, da sie nicht an ein Objekt gebunden sind, sondern bereits ab der Existenz einer Klasse verfügbar sind. Statische Attribute gelten dabei für alle Instanzen einer Klasse. Ihr Wert ist für alle Instanzen der Klasse derselbe. Deklariert werden Klassenvariablen über das Schlüsselwort `static`.

Die folgende Abbildung zeigt, wie Herr Koch die Klassenvariable `beschreibungsTrennzeichen` als statisches Attribut der Klasse „Artikel“ deklariert. Er kann dann im Folgenden über die Klasse auf dieses Attribut zugreifen und muss vorher kein Objekt erzeugen.

Klassenvariablen
Das sind Attribute, die für alle Instanzen einer Klasse gleich sind und über den Klassennamen angesprochen werden.

Abbildung 64: Deklaration und Verwendung eines statischen Attributs

```
public abstract class Artikel {  
    public static String TRENNZEICHEN = ";"  
    [...]  
}
```

Deklaration eines statischen Attributs

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println(Artikel.TRENNZEICHEN);  
    }  
}
```

Zugriff auf das statische Attribut über den Klassennamen

Quelle: erstellt im Auftrag der IU, 2013.

Ändert man den Wert des statischen Attributs, so ändert er sich für alle Objekte der Klasse und der Unterklassen.

Neben Attributen können auch Methoden als `static` deklariert werden. Diese Methoden sind ebenfalls unabhängig von der Existenz eines konkreten Objektes und werden über die Klasse aufgerufen.

Herr Koch verwendet eine statische Methode, um eine Prüfung auf die Gültigkeit einer gegebenen Artikelnummer zu implementieren. Die Prüfung kann somit erfolgen, noch bevor ein konkretes Artikel-Objekt erzeugt wurde. Abbildung 65 zeigt die Deklaration und Verwendung der statischen Methode.

Abbildung 65: Deklaration und Verwendung von statischen Methoden

```
public abstract class Artikel {  
    public static boolean istArtikelnummerGueltig(String artikelnummer) {  
        ...  
    }  
    ...  
}
```

Deklaration einer statischen Methode

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println(  
            Artikel.istArtikelnummerGueltig("abd:1271813186317"));  
    }  
}
```

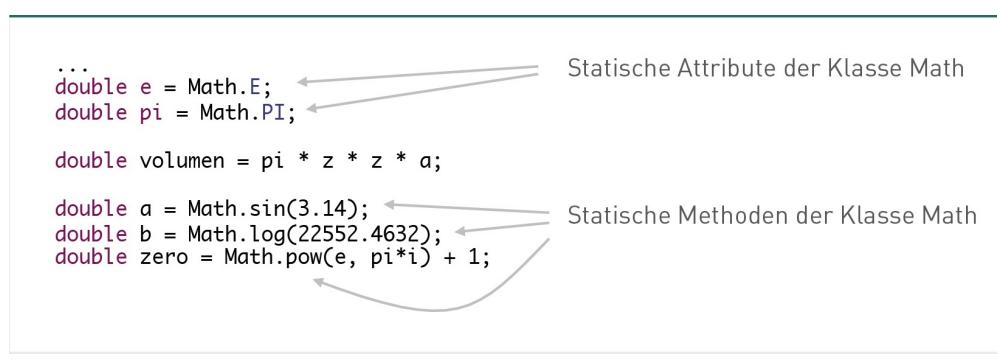
Aufruf der statischen Methode Attribut über den Klassennamen

Quelle: erstellt im Auftrag der IU, 2013.

Da statische Methoden auch ohne die Existenz von Objekten der jeweiligen Klasse aufgerufen werden können, darf im Methodenrumpf nicht auf Instanzmethoden oder -variablen zugegriffen werden.

Statische Attribute findet man in der Java-Klassenbibliothek häufig bei der Definition von Konstanten. So sind zum Beispiel die Kreiszahl pi und Eulersche Zahl e als statische Attribute der Klasse „Math“ (siehe Oracle 2013b) definiert.

Abbildung 66: Statische Attribute und Methoden der Klasse „Math“



Quelle: erstellt im Auftrag der IU, 2013.

Beispiele für statische Methoden findet man ebenfalls in der Klasse „Math“. Die Ergebnisse der Methoden sin, cos, log sind berechenbar, ohne dass auf den internen Zustand eines Objektes zugegriffen werden muss. Vor der Verwendung dieser Methoden muss also keine Instanz von „Math“ erzeugt werden. Die Ausgabe der Methoden ist einzig und allein durch die Werte der Parameter bestimmt.



ZUSAMMENFASSUNG

Abstrakte Klassen und Methoden werden mittels des Schlüsselworts `abstract` deklariert. Abstrakte Klassen sind Klassen, von denen keine Instanzen erzeugt werden dürfen. Sie werden mittels Vererbung in Unterklassen konkretisiert und dienen hauptsächlich der Zusammenfassung von gemeinsamen Attributen und Methoden der Unterklassen. Abstrakte Methoden wiederum sind Methoden, die keinen Methodenrumpf haben. Sie werden verwendet, um zu erzwingen, dass eine Unterklasse diese Methode mit einer konkreten Implementierung überschreibt. Sobald eine Klasse eine abstrakte Methode deklariert, ist die gesamte Klasse abstrakt.

Polymorphie bedeutet Vielgestalt und ermöglicht es, einer Variablen einer bestimmten Klasse auch Instanzen von Unterklassen dieser Klasse zuzuweisen. Der Typ der Variable bestimmt dabei, auf welche Attribute und Methoden zugegriffen werden darf. Erst zur Laufzeit wird aber ermit-

telt, welche konkrete Implementierung verwendet wird. Der Typ der Instanz entscheidet darüber, welcher Attributwert zurückgegeben und welche Methodenimplementierung ausgeführt wird.

Statische Attribute und Methoden werden mittels des Schlüsselworts `static` deklariert. Sie gelten für alle Instanzen einer Klasse. Man greift über den Klassennamen auf sie zu und muss vorher kein konkretes Objekt der Klasse erzeugen. Statische Attribute werden häufig für die Definition von Konstanten verwendet. Statische Methoden können nicht auf Instanzvariablen oder -methoden zugreifen und implementieren häufig Funktionen, deren Ergebnis nur von ihren Eingabeparametern abhängt.

LEKTION 7

KONSTRUKTOREN ZUR ERZEUGUNG VON OBJEKTEN

LERNZIELE

Nach der Bearbeitung dieser Lektion werden Sie wissen, ...

- was ein Standard-Konstruktor ist und wie man mithilfe von Konstruktoren Objekte erzeugen kann.
- wie man mit unterschiedlichen Konstruktoren Objekte mit bedarfsgerechten Vorbelegungen erzeugen kann.
- was bei Konstruktoren im Zusammenhang mit Vererbung zu beachten ist.

7. KONSTRUKTOREN ZUR ERZEUGUNG VON OBJEKten

Aus der Praxis

Beim Erzeugen von Objekten war Herr Koch bislang stets darauf angewiesen, deren Attribute mit Getter- und Setter-Methoden einzeln zu setzen. Dabei unterliefen ihm gelegentlich Fehler, zum Beispiel weil er vergaß Attribute zu setzen. Herr Koch erkundigte sich und las von der Möglichkeit, Konstruktoren zu definieren. Seitdem kann er einen Standard-Konstruktor definieren und damit genau festlegen, wie Objekte der Klasse Kunde erzeugt werden können. Um flexibel auf unterschiedliche Situationen reagieren zu können, kann Herr Koch auch unterschiedliche Konstruktoren definieren, indem er den Standard-Konstruktor überlädt. Er ist außerdem mit der Abarbeitungsreihenfolge der Konstruktoren vertraut, falls die erzeugten Objekte in einer Vererbungshierarchie stehen. Dies wird zum Beispiel relevant, wenn für Premiumkunden des Online-Shops ein reduzierter Mindestbestellwert festgelegt werden soll.

7.1 Der Standard-Konstruktor

Herr Koch weiß mittlerweile, was unter der objektorientierten Programmierung zu verstehen ist und welche Mittel Java bietet, um die damit verbundenen Konzepte einzusetzen. Dabei hat er bereits am Rande herausgefunden, wie Objekte in Java mit dem new-Operator erzeugt werden können, ohne allerdings die Details hinter dieser Anweisung zu kennen (siehe Abbildung 21). Er stellt fest, dass das Erzeugen von Objekten eine zentrale Aufgabe während der Java-Programmierung ist. Deswegen möchte er sich nun genauer damit befassen. Insbesondere will Herr Koch herausfinden, wie man die Erzeugung beeinflussen kann. Hierzu beginnt er bei der `main()`-Methode aus dem bereits bekannten Online-Shop. Sie zeigt den Einsatz des bereits bekannten new-Operators:

Abbildung 67: Einsatz des new-Operators in der main-Methode einer Autoverwaltung

Abbildung 67: Einsatz des new-Operators in der main-Methode einer Autoverwaltung

```
public class OnlineShop {  
    public static void main(String[] args){  
        Kunde kunde1 = new Kunde(); ← Erzeugung der Objekte kunde1  
        kunde1.setName("Lange"); der Klasse Kunde sowie w in der  
        kunde1.setVorname("Gerd"); Klasse Warenkorb durch Aufruf des  
        kunde1.setGeburtsdatum("1967-10-23"); Standard-Konstruktors  
        kunde1.setGeschlecht("m");  
        Warenkorb w = new Warenkorb(); ←  
        kunde1.setWarenkorb(w);  
    }  
}
```

Quelle: erstellt im Auftrag der IU, 2013.

Wie der Name schon verspricht, erfüllt der **new-Operator** den Zweck, ein neues Objekt zu einer gegebenen Klasse (hier: Kunde und Warenkorb) zu erzeugen. Unklar ist bislang, wie genau die Erzeugung eines solchen Objekts definiert ist und was genau passiert, wenn man den new-Operator aufruft. Um zu wissen, mit welchen Werten die Attribute eines Kunden-Objektes belegt werden und wie man diese Definition beeinflussen kann, wird der Standard-Konstruktor eingesetzt. Er wird durch den new-Operator aufgerufen und gibt das erzeugte Objekt als Ergebnis zurück.

Man kann sich den **Standard-Konstruktor** wie eine spezielle Art einer Methode vorstellen, die es nur einmal pro Klasse geben darf. Abgesehen von ein paar Einschränkungen ist die Syntax einer normalen Methode sehr ähnlich (siehe Abbildung 68).

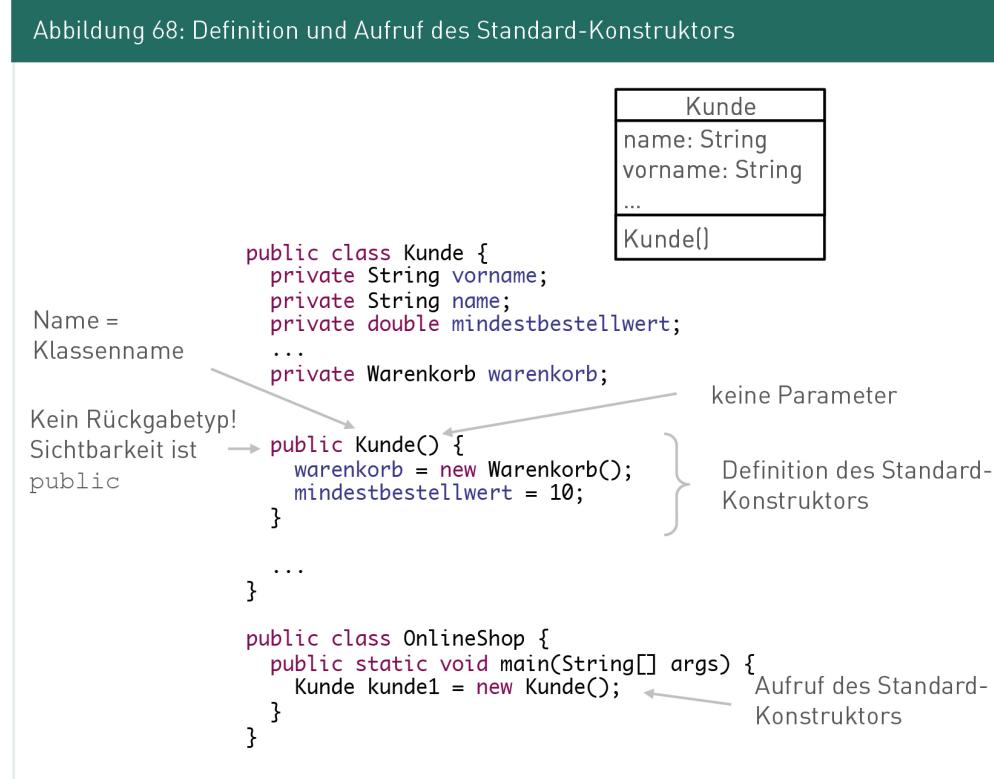
new-Operator

Dieser erzeugt ein neues Objekt zu einer gegebenen Klasse. Zu diesem Zweck wird dabei der Konstruktor aufgerufen.

Standard-Konstruktor

Das ist eine spezielle Methode zur Erzeugung von Objekten einer Klasse.

Abbildung 68: Definition und Aufruf des Standard-Konstruktors



Quelle: erstellt im Auftrag der IU, 2013.

Für die Definition des Standard-Konstruktors gelten die folgenden Regeln: Der Name des Konstruktors muss immer mit der Klasse übereinstimmen (hier: Kunde). Er besitzt keinen Rückgabetyp und die Sichtbarkeit sollte immer public sein, damit andere Klassen ein Objekt dieser Klasse per new-Operator erzeugen können. Im Gegensatz zu den überladenen Konstruktoren, die im nächsten Abschnitt behandelt werden, hat der Standard-Konstruktor keine Parameter. Daher erfüllt der Standard-Konstruktor im Regelfall nur zwei Aufgaben:

1. Anpassen der Defaultwerte für primitive Datentypen
2. Erzeugung von nicht-primitiven Attributen (Objekte)

Bezogen auf das Beispiel bewirkt der Standard-Konstruktor also erstens, dass der Mindestbestellwert nicht standardmäßig auf 0 voreingestellt ist, sondern auf 10 EUR. Zweitens wird für das nicht-primitive Attribut warenkorb ein neues, leeres Objekt erzeugt.

Wenn die beiden oben genannten Aufgaben für eine bestimmte Klasse nicht notwendig sein sollten, so ist auch die Definition eines Standard-Konstruktors nicht erforderlich und man kann sie weglassen. In einem solchen Fall wird implizit ein **leerer Standard-Konstruktor** vom Java-Compiler hinzugefügt. Das heißt, der Konstruktor ist zwar nicht im Quelltext sichtbar, kann aber trotzdem mit dem new-Operator aufgerufen werden und somit ein Objekt erzeugen (siehe Abbildung 67).

Bei der Verwendung von Konstruktoren im Allgemeinen ist zu beachten, dass nicht mehr benötigte Objekte automatisch von der Java-Laufzeitumgebung gelöscht werden. Man spricht in diesem Zusammenhang von „**Garbage Collection**“ oder automatischer Speicherverwaltung. Das heißt, es ist im Gegensatz zu anderen Programmiersprachen wie C/C++ oder Objective C normalerweise nicht nötig, die Objekte durch eine Programmieranweisung explizit zu zerstören. Selbst bei Programmen mit hohem Speicherbedarf funktioniert die in Java integrierte automatische Speicherverwaltung sehr gut. Sollte es trotzdem Bedarf zur Freigabe von Speicher geben, kann der Garbage Collector durch die Anweisung `System.gc()` aufgerufen werden.

Leerer Standard-Konstruktor

Wird kein Konstruktor definiert, fügt der Compiler einen leeren Standard-Konstruktor hinzu.

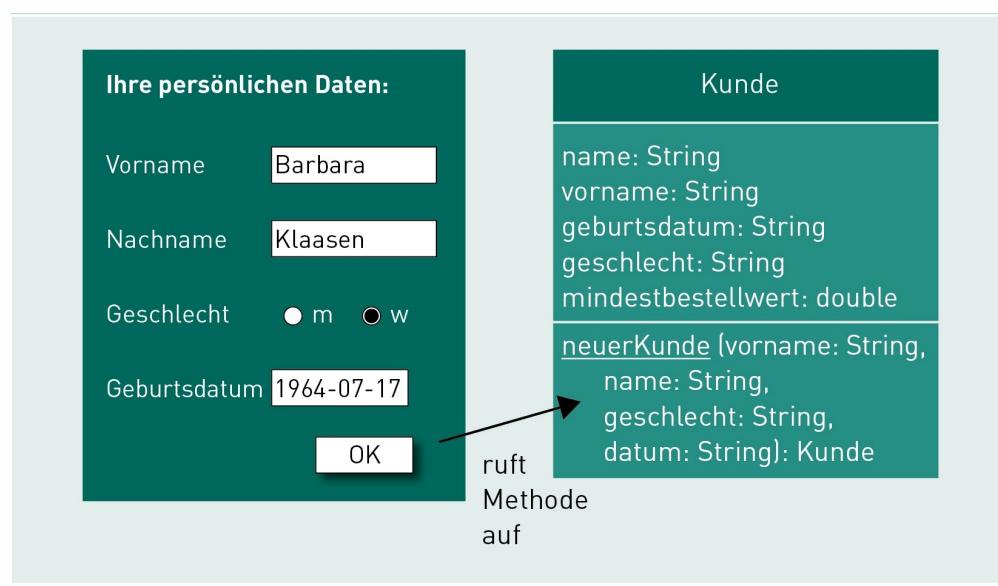
Garbage Collection

Das ist ein Konzept der Programmiersprache Java zur automatischen Speicherverwaltung.

7.2 Überladen von Konstruktoren

In der Praxis ist es üblich, Objekte unmittelbar nach ihrer Erzeugung mit Werten zu beleben. Herr Koch stellt sich zum Beispiel vor, dass ein neuer Besucher des Online-Shops ein neues Kundenkonto erstellen möchte. Die auf dem Online-Formular (siehe Abbildung 69) ausgefüllten Informationen sollen anschließend in einem neuen Kunden-Objekt gespeichert werden. Hierzu benötigt Herr Koch eine neue Methode mit entsprechenden Parametern, die ein Kunden-Objekt zurückliefert (siehe Klassendiagramm).

Abbildung 69: Formular zur Erfassung eines Kundenkontos (links); mögliche Implementierung (rechts)



Quelle: erstellt im Auftrag der IU, 2013.

Mit den bislang bekannten Mitteln könnte das wie folgt programmiert werden:

Abbildung 70: Herkömmliche Belegung der Objektattribute durch Setter- Methoden

```
Die Benutzereingaben werden der Methode als Parameter übergeben  
↓  
public Kunde neuerKunde(String vorname, String name, String geschlecht,  
String datum) {  
    Kunde k = new Kunde();  
    k.setMindestbestellwert(10);  
    k.setVorname(vorname);  
    k.setName(name);  
    k.setGeschlecht(geschlecht);  
    k.setGeburtsdatum(datum);  
    return k;  
}
```

Erzeugung eines Auto-Objekts per Standard-Konstruktor

Die Attribute werden mit den Setter-Methoden gesetzt.

Quelle: erstellt im Auftrag der IU, 2013.

Je mehr Attribute eine Klasse besitzt, desto höher steigt der Programmieraufwand und das Programm wird um viele gleichartige Anweisungen erweitert. Zudem ist es dann sehr schwierig festzustellen, ob tatsächlich alle benötigten Attribute gesetzt wurden oder ob nicht vielleicht ein Attribut vergessen wurde. Für solche Fälle kann der **Standard-Konstruktur überladen** werden (siehe Lernzyklus 3.4). Das heißt, man definiert einen zweiten Konstruktor zusätzlich zum Standard-Konstruktor. Dieser Konstruktor besitzt im Gegensatz zum Standard-Konstruktor eine Parameterliste. Mit der Parameterliste kann nun genau festgelegt werden, mit welchen Attributen das Objekt zur Laufzeit mit Informationen gefüllt werden soll. Die Wertebelegung muss nicht mehr an jeder betroffenen Stelle im Programm manuell mit Setter-Methoden vorgenommen werden, sondern nur einmal im neuen Konstruktor. Abbildung 71 zeigt anhand des bekannten Beispiels, wie der Standard-Konstruktor überladen werden kann.

Überladener Standard-Konstruktor

Ein solcher dient der Initialisierung eines Objekts mit Werten, die erst zur Laufzeit festgelegt sind.

Hierzu definiert er im Gegensatz zum Standard-Konstruktor eine Parameterliste.

Abbildung 71: Beispiel eines überladenen Standard-Konstruktors

```
public class Kunde {  
    private String vorname;  
    private String name;  
    private String geschlecht;  
    private String geburtsdatum;  
    private double mindestbestellwert;  
    private Warenkorb warenkorb;  
  
    public Kunde(){  
        warenkorb = new Warenkorb();  
        mindestbestellwert = 10;  
    }  
  
    public Kunde(String vorname, String name,  
                String geschlecht, String datum) {  
        this.vorname = vorname;  
        this.name = name;  
        this.geschlecht = geschlecht;  
        this.geburtsdatum = datum;  
    }  
    ...  
}
```

Standard-Konstruktor → Auflösung des Namenskonfliktes zwischen Parameter und Attributen mit this

Überladener Standard-Konstruktor mit Parameterliste →

Quelle: erstellt im Auftrag der IU, 2013.

Beim Aufruf des Konstruktors stellt Herr Koch fest, dass sich im Vergleich zum Aufruf des Standard-Konstruktors nicht viel verändert hat. Selbstverständlich wird auch in diesem Fall wieder der new-Operator eingesetzt und der Klassename wird angegeben. Allerdings werden beim Aufruf des überladenen Konstruktors zusätzlich die Parameter in der runden Klammer angegeben (siehe Abbildung 72). Im Vergleich zum Beispiel aus Abbildung 70 ist der Programmcode nun deutlich aufgeräumter. Zudem würde eine Fehlermeldung des Compilers verhindern, dass Herr Koch ein Attribut in der Parameterliste vergisst.

Abbildung 72: Verwendung eines überladenen Konstruktors als Alternative zum Einsatz vieler Setter-Methoden

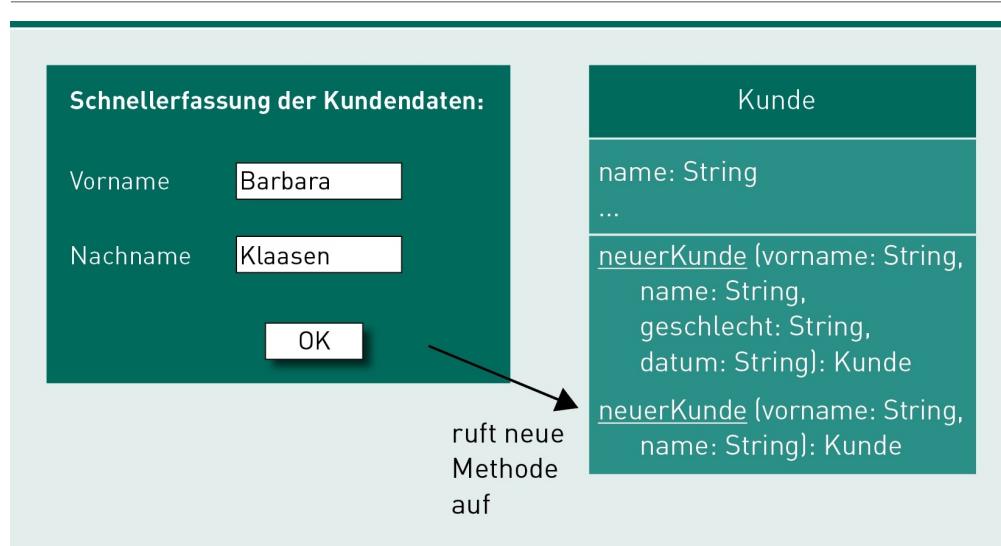
Abbildung 73: Verwendung eines überladenen Konstruktors als Alternative zum Einsatz vieler Setter-Methoden

```
public Kunde neuerKunde(String vorname, String name, String geschlecht,  
                        String datum) {  
    Kunde k = new Kunde(vorname, name, geschlecht, datum); ← Aufruf des überla-  
    return k;                                            denen Konstruktors  
}
```

Quelle: erstellt im Auftrag der IU, 2013.

Ein weiteres Problem tritt auf, wenn Objekte einer Klasse je nach Ausgangssituation mit unterschiedlichen Kombinationen von Attributen initialisiert werden sollen. Beispielsweise im Online-Shop gibt es neben der oben beschriebenen Eingabemaske eine zusätzliche Maske (siehe Abbildung 73).

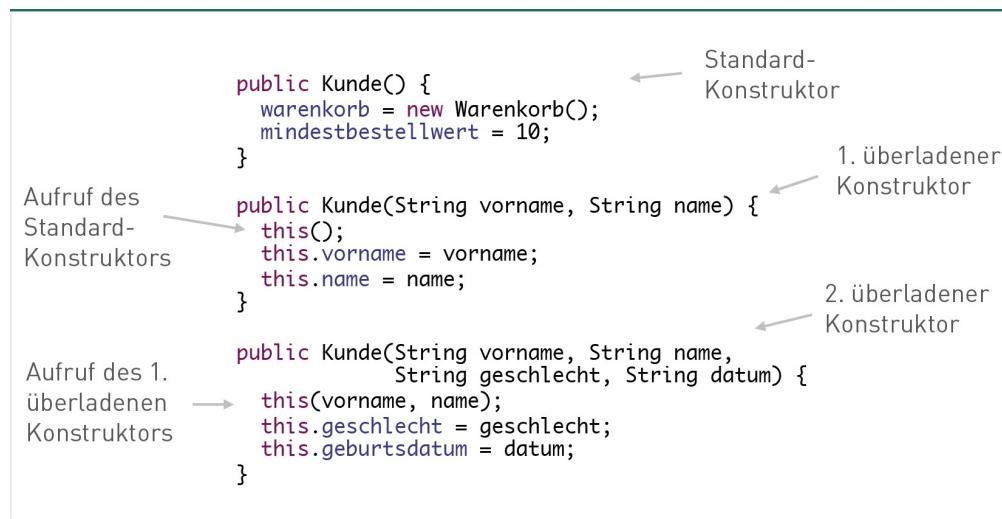
Abbildung 73: Zweite, reduzierte Eingabemaske (links) mögliche Implementierung (rechts)



Quelle: erstellt im Auftrag der IU, 2013.

Im Gegensatz zur ersten Maske dient diese nur der Verwaltung der wichtigsten Attribute (zum Beispiel im Rahmen einer schnelleren, vorläufigen Datenerfassung). In solchen Fällen ist es notwendig, einen passenden Konstruktor für jede Kombination an bereitstehenden Informationen zu definieren. Dank des Überladens ist es in Java möglich, beliebig viele Konstruktoren zu programmieren, sofern sich die Parameterlisten unterscheiden. In der Abbildung 74 zeigen wir, wie das vorangegangene Beispiel um einen weiteren Konstruktor ergänzt werden kann. Dabei ist es durchaus sinnvoll, die Konstruktoren sich gegenseitig aufrufen zu lassen, um doppelten Code zu vermeiden und die Wartbarkeit zu erhöhen. Das Aufrufen von anderen Konstruktoren innerhalb überladener Konstruktoren geschieht mit der Anweisung `this()`. Diese Anweisung muss immer an erster Stelle stehen, andernfalls produziert der Compiler eine Fehlermeldung.

Abbildung 74: Definition mehrerer überladener Konstruktoren



Quelle: erstellt im Auftrag der IU, 2013.

Die Abarbeitungsreihenfolge beim Aufruf einer solchen Verkettung von überladenen Konstruktoren lässt sich an dem abgebildeten Beispiel erläutern: Falls ein Programm den zweiten überladenen Konstruktor aufruft, um ein neues Objekt mit allen Attributen zu erzeugen, so wird zunächst der erste überladene Konstruktor aufgerufen. Ihm werden die erforderlichen Parameter vorname und name übergeben. Der erste Konstruktor ruft wiederum zunächst den Standard-Konstruktor auf, natürlich ohne Parameter. Sobald der Standard-Konstruktor den Mindestbestellwert festgelegt hat und ein neues Objekt für den Warenkorb erzeugt hat, wird das Programm beim ersten Konstruktor fortgesetzt. Dort werden dann Vor- und Nachname gesetzt. Die Informationen dazu stammen aus der Parameterliste des zweiten Konstruktors. Im letzten Schritt springt das Programm wieder zurück in den zweiten Konstruktor und sorgt dafür, dass weitere Attribute wie das Geschlecht und das Geburtsdatum gesetzt werden.

Konstruktoren eignen sich auch zur Erstellung von Kopien gleichartiger Objekte. Wenn zum Beispiel im Online-Shop eine Sicherheitskopie eines Kunden erstellt werden soll, kann man einen „**Copy-Konstruktor**“ verwenden (siehe Abbildung 75), der dann wie folgt aufgerufen wird.

Code

```
Kunde neuesKundenObjekt = new Kunde(bestehendesKundenObjekt);
```

Copy-Konstruktor

Ein solcher dient dem Klonen von Objekten. Man kann ihn so programmieren, dass er eine tiefe Kopie (alle Referenzdatentypen werden ebenfalls geklont) oder eine flache Kopie (nur primitive Datentypen und Strings werden geklont) erzeugt.

Abbildung 75: Copy-Konstruktor, der eine Kopie eines Kunden-Objektes erzeugen kann

Abbildung 76: Copy-Konstruktor, der eine Kopie eines Kunden-Objektes erzeugen kann

```
public Kunde(Kunde original){  
    vorname = original.vorname;  
    name = original.name;  
    geschlecht = original.geschlecht;  
    geburtsdatum = original.geburtsdatum;  
    warenkorb = original.warenkorb;  
    mindestbestellwert = original.mindestbestellwert;  
}
```

Quelle: erstellt im Auftrag der IU, 2013.

Um die Funktion eines Copy-Konstruktors möglichst deutlich zu machen, wurde in diesem Fall auf die Verkettung von Konstruktoren verzichtet. Natürlich könnte man auch den zweiten Konstruktor aus Abbildung 74 verwenden, um redundanten Code zu vermeiden. Man sollte allerdings nicht vergessen, den Mindestbestellwert zu übernehmen. Denn sollte ein Kunde einmal einen individuellen, von 10 EUR verschiedenen Mindestbestellwert haben, muss sich ein fremdes Programm beziehungsweise ein anderer Programmierer darauf verlassen können, dass es sich hier um einen „echten“ Copy-Konstruktor handelt.

In diesem Zusammenhang ist auch die Zuweisung `warenkorb = original.warenkorb` nochmals zu überdenken, denn sie führt dazu, dass anschließend beide Kundenobjekte (das Original und der Klon) *denselben* Warenkorb besitzen, da ihre jeweils in `warenkorb` gespeicherten Referenzen anschließend auf dasselbe Warenkorbobjekt verweisen. In einem solchen Fall spricht man dann auch von einer **flachen Kopie**, denn nur das Objekt selbst, nicht aber auch die von ihm referenzierten Objekte werden geklont. Würde man hingegen in dem Copy-Konstruktur in Abbildung 75 bei der Zuweisung von `warenkorb` wiederum einen entsprechenden Copy-Konstruktor zur Klasse Warenkorb aufrufen, dann hätte man einen geklonten Kunden mit einem ebenfalls geklonten Warenkorb:

Code

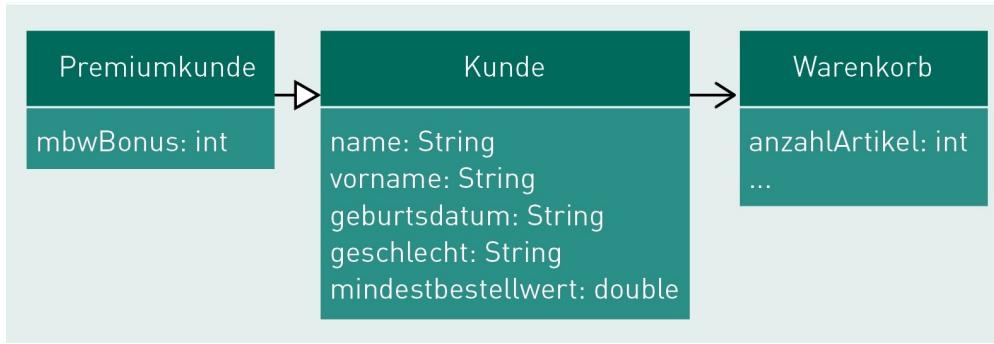
```
warenkorb = newWarenkorb(original.warenkorb)
```

In diesem Fall spricht man dann von einer **tiefen Kopie**, da auch Referenzdatentypen durch das Anlegen neuer Objekte geklont und nicht einfach nur die Referenzen kopiert werden.

Konstruktoren und Vererbung

Sofern Klassen in einer Vererbungshierarchie stehen, kann es in bestimmten Fällen notwendig sein, auf den Konstruktor der Oberklasse zuzugreifen. Angenommen im Online-Shop wird zwischen Premiumkunden und normalen Kunden unterschieden. Im Gegensatz zu allgemeinen Kunden gilt bei Premiumkunden ein reduzierter Mindestbestellwert (siehe Abbildung 76).

Abbildung 76: Erweitertes UML Klassendiagramm des Online-Shop-Beispiels

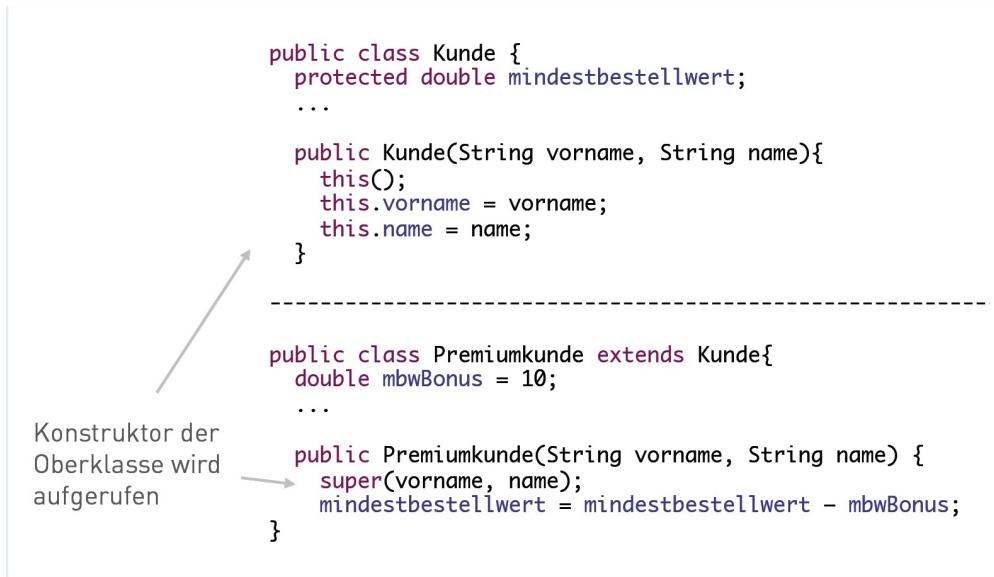


Quelle: erstellt im Auftrag der IU, 2013.

In der Programmiersprache Java ist es möglich, den Konstruktor der Oberklasse aufzurufen. Hierzu wird das in Kapitel 5 eingeführte Schlüsselwort **super** verwendet (siehe Abbildung 77). Genauso wie das Schlüsselwort **this** kann auch **super** mit oder ohne Parameter aufgerufen werden. Im ersten Fall wird nach einem geeigneten überladenen Konstruktor gesucht, im letzteren Fall wird der Standard-Konstruktor der Oberklasse aufgerufen.

super()
Mit diesem Schlüsselwort kann man den Konstruktor der Oberklasse aufrufen.

Abbildung 77: Aufruf eines Konstruktors der Oberklasse mit super()



Quelle: erstellt im Auftrag der IU, 2013.



ZUSAMMENFASSUNG

Konstruktoren ermöglichen es einem Programmierer, die Regeln zur Erzeugung von Objekten festzulegen. Indem der Standard-Konstruktor überschrieben wird, können die voreingestellten Werte für primitive Datentypen angepasst werden. Falls eine Klasse Attribute mit komplexen Datentypen enthält, so können Konstruktoren dafür sorgen, dass immer Instanzen solcher Attribute vorhanden sind.

Für den Fall, dass Objekte einer Klasse je nach Situation auf verschiedene Weise erzeugt werden sollen, können auch mehrere Konstruktoren definiert werden. Anhand der unterschiedlichen Parameterliste kann dann differenziert werden, welcher Konstruktor bei einem Aufruf gemeint ist. Um Mehraufwand durch doppelten Code zu vermeiden, können Konstruktoren verschachtelt werden, das heißt, sie können sich gegenseitig aufrufen und jeweils nur die Menge an zusätzlichen Parametern selbst setzen.

LEKTION 8

AUSNAHMEBEHANDLUNG MIT EXCEPTIONS

LERNZIELE

Nach der Bearbeitung dieser Lektion werden Sie wissen, ...

- in welchen Fällen die Ausnahmebehandlung in Java sinnvoll beziehungsweise notwendig ist.
- welche Mittel Java zur Verfügung stellt, um Ausnahmen (Exceptions) abzufangen, zu werfen oder zu behandeln.
- in welchen Fällen diese Mittel nicht ausreichen.
- wie Sie eigene Exceptions programmieren können, um auch solche Fälle abdecken zu können.

8. AUSNAHMEBEHANDLUNG MIT EXCEPTIONS

Aus der Praxis

Herr Koch hat bislang viel Zeit investiert, um seine Programme robuster gegen Fehler zu machen. Um solche Fehlersituationen an aufrufende Programmkomponenten zurückzumelden, dachte er sich viele verschiedene Fehlercodes aus (z. B. „-1“, falls ein benötigter Buchtitel nicht in der Datenbank des Online-Shops gefunden wurde). Mit der Zeit hat er aber die Übersicht verloren, sodass Fehlercodes manchmal falsch interpretiert wurden oder gar nicht erst als solche erkannt wurden.

Herr Koch hat sich daher mit den in Java eingebauten Instrumenten zur Ausnahmebehandlung (Exception Handling) auseinandergesetzt. Ab sofort nutzt er sie, um Fehlersituationen im Programm einfacher und zuverlässiger abfangen zu können. Er setzt Exceptions gezielt ein, um Ausnahmen an das aufrufende Programm zurückzumelden. Die mühselige Definition und Interpretation von eigenen Fehlercodes ist nun Vergangenheit. Um spezielle Fehlersituationen definieren und mit einer eigenen Fehlermeldung versehen zu können, kann er eigene Exception-Klassen programmieren.

8.1 Typische Szenarien der Ausnahmebehandlung

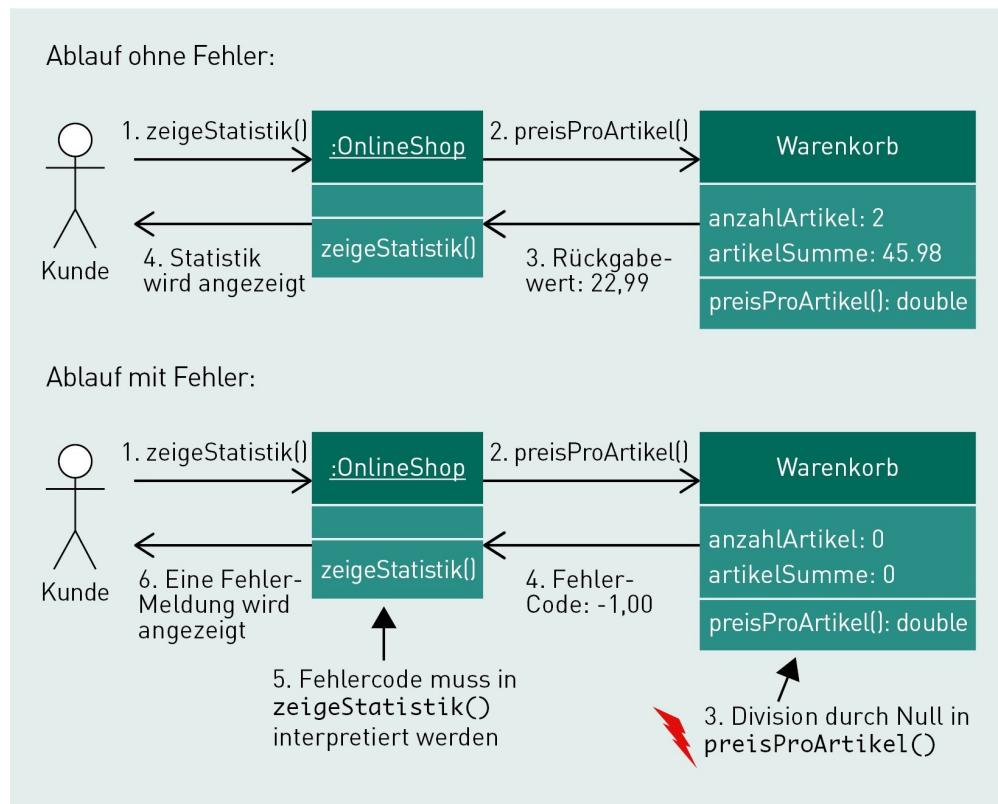
Ausnahme

Das ist ein Zustand, der das Programm an der Fortführung des normalen Ablaufs hindert.

Während der Ausführung eines Programms können unterschiedliche **Ausnahmesituat**ionen auftreten. Zum Beispiel könnte bei einer Berechnung durch Null dividiert werden, eine zu bearbeitende Datei kann nicht gefunden werden oder eine Methode erhält einen ungültigen Parameter. Ganz allgemein kann man solche Situationen als Zustände bezeichnen, die das Programm irgendwie daran hindern, im normalen Ablauf fortzufahren. Solche Situationen sind natürlich unerwünscht und sollen durch Konzepte zur Ausnahmebehandlung vermieden werden. Beispielsweise soll eine falsche Benutzereingabe im Online-Shop nicht dazu führen, dass das gesamte Shop-System komplett ausfällt. Dies hätte nicht nur schlimme wirtschaftliche Folgen, sondern wirkt sich auch negativ auf die Außenwahrnehmung des Unternehmens und auf die Kundenzufriedenheit aus. Stattdessen sollten die Eingaben zum Beispiel ignoriert und dem betroffenen Benutzer eine Nachricht angezeigt werden.

In vielen Programmiersprachen, die kein Konzept zur Ausnahmebehandlung bereithalten, werden solche Ausnahmesituationen meist mit bestimmten Rückgabewerten gekennzeichnet. Die Rückgabewerte entsprechen dann einer vorab festgelegten Definition von möglichen **Fehlersignalen** (z. B. „-1“ für „Division durch null“). Abbildung 78 zeigt einen Anwendungsfall für Ausnahmebehandlung, der im weiteren Verlauf der Lektion als durchgehendes Beispiel verwendet werden soll.

Abbildung 78: Beispiel-Szenario für die Ausnahmebehandlung mit Fehlersignalen



Quelle: erstellt im Auftrag der IU, 2013.

Ein Kunde des Online-Shops möchte im Anschluss an seine Bestellung Statistiken über den Kauf einsehen. Zu diesem Zweck ruft das Online-Shop-System verschiedene Methoden des Warenkorbs auf, darunter auch `preisProArtikel()`. Falls bei diesem Programmablauf kein Problem auftritt, werden die Informationen zuerst an den Online-Shop und dann gesammelt an den Kunden zurückgegeben. Bei dem Ablauf kann es aber auch zu verschiedenen Ausnahmesituationen kommen. Zum Beispiel könnte der Warenkorb durch einen Fehler in der Datenbank leer sein. In solchen Fällen werden bei älteren Programmiersprachen ohne explizite Unterstützung für Ausnahmebehandlung Fehlersignale zurückgegeben. Die Signale müssen erst vom aufrufenden Programm (hier: der Online-Shop) interpretiert und in eine für den Benutzer lesbare Fehlermeldung übersetzt werden (siehe Abbildung 79, unten).

Abbildung 79: Behandlung von Fehlersituationen mit Signalen

```

public class Warenkorb {
    private int anzahlArtikel;
    private float artikelSumme;
    ...
}

public double preisProArtikel(){
    if (anzahlArtikel > 0)
        return artikelSumme / anzahlArtikel;
    else
        return -1.0;
}

```

Rückgabewerte und Fehlersignale werden gleichermaßen als double-Wert ausgegeben. Sie können nur durch richtige Interpretation unterschieden werden.

```

public class OnlineShop {
    ...
    public void zeigeStatistik (Warenkorb w) {
        double ppa = w.preisProArtikel();
        ...
        if (ppa == -1)
            System.out.println ("Ungültiger Preis pro Artikel.");
        else
            System.out.println ("Der Preis pro Artikel lautet:" + ppa);
    }
}

```

Die Anzahl der Artikel darf nicht null bzw. muss positiv sein.

Fehlersignal, das für alle ungültigen Berechnungen gilt. Die Bedeutung des Signals muss dem aufrufenden Programm bekannt sein.

Quelle: erstellt im Auftrag der IU, 2013.

Diese Herangehensweise hat jedoch einen großen Nachteil: Weil Funktionen, wie zum Beispiel `preisProArtikel()`, nur einen Rückgabewert haben, müssen technische Fehler signale und fachliche Rückgabewerte gleichermaßen über diesen Rückgabewert an das aufrufende Programm zurückgegeben werden. Diese Vermischung von technischen Signalen und fachlichen Werten führt zu schlechter Softwarequalität und sollte daher vermieden werden.

Um robustere Programme schreiben zu können, sollte es also einen eigenen Kanal für die Signalisierung von Fehlern geben, der zur Entkopplung von fachlichen Rückgabewerten und Fehlersignalen führt. Eine eigene Datenstruktur für solche Signale böte zudem den Vorteil, differenziertere Fehlermeldungen definieren zu können. Zuletzt wäre es wünschenswert, alle aufrufenden Programme dazu zu zwingen, mögliche Ausnahmen einer aufgerufenen Methode abfangen zu müssen. Denn dadurch könnte sichergestellt werden, dass Lösungsstrategien zu häufigen Fehlerquellen existieren.

8.2 Standard-Exceptions in Java

Die Programmiersprache Java bietet mit der Klasse **Exception** ein objektorientiertes Konzept zur Definition und Verwendung von Ausnahmen. Neben einer umfangreichen Sammlung vordefinierter Standard-Exceptions (Tabelle 18 zeigt einen Auszug; eine vollständige Übersicht findet sich auf der Internetseite von ProgrammingGuide) gibt es auch spezielle Sprachelemente zum Erzeugen („Werfen“) und Auffangen von Exceptions.

Tabelle 18: Vordefinierte Standard-Exceptions in Java (Auszug)

Name	Beschreibung	Beispiel
ArithmException	Division durch Null	<code>int n = 0; return 1/n;</code>
ArrayIndexOutOfBoundsException	Zugriff auf ein Listenelement außerhalb des definierten Bereichs (hierzu später mehr)	<code>int[] zahlen = new int [8]; zahlen [10] = 4;</code>
NullPointerException	Zugriff auf ein nicht instanziiertes Objekt	<code>kunde k = null; k.setName("Meier");</code>

Quelle: eigene Darstellung in Anlehnung an ProgrammingGuide (o. J.).

Bevor in Lernzyklus 8.3 die Definition eigener Exceptions beschrieben wird, soll zunächst mithilfe der neuen Sprachelemente der Einsatz der Standard-Exceptions eingeführt werden. Hierzu wird erneut das Online-Shop-Beispiel aufgegriffen: Angenommen die Klasse Warenkorb stellt eine Methode bereit, die den Durchschnittspreis der gekauften Artikel ermittelt und auf der Konsole ausgibt. Falls der Warenkorb leer ist, wird durch Null geteilt und es kommt zu einer `ArithmException`. Java bietet mehrere Alternativen, wie mit der Exception umgegangen werden soll:

1. Die Exception wird innerhalb der Methode abgefangen.
2. Die Exception wird an das aufrufende Programm weitergeleitet.
3. Die Exception wird abgefangen und dann mit einer spezifischen Meldung an das aufrufende Programm weitergeleitet.

Abbildung 80 zeigt die erste Alternative anhand eines Codebeispiels. Sie soll verdeutlichen, wie die Ausnahme erkannt und behandelt werden kann, ohne dass das aufrufende Programm darüber informiert wird. Der fehleranfällige Programmabschnitt wird von einem **try-Block** umschlossen. Mithilfe des catch-Blocks kann angegeben werden, welche im try-Block auftretenden Ausnahmen abgefangen werden sollen und wie sie behandelt werden sollen. Im Beispiel wird das Ergebnis auf 0 gesetzt, wenn kein Artikel im Warenkorb liegt.

Exception

Diese Klasse bildet den Kern der objektorientierten Ausnahmebehandlung in Java. Ihre Unterklassen können zur Behandlung vieler unterschiedlicher Ausnahmesituationen herangezogen werden.

try/catch-Block

Exceptions werden mit einem try/catch-Block abgefangen. Im try-Block stehen die kritischen Programmanweisungen. Im catch-Block werden die Ausnahmen behandelt.

Abbildung 80: Auffangen von Standard-Exceptions mit try/catch-Block

```

public class Warenkorb {
    private int anzahlArtikel;
    private float artikelSumme;
    ...

    Ausnahmen im
    try-Block
    können
    im catch-Block
    behandelt werden
    →
    public double preisProArtikel(){
        double ergebnis;
        try {
            ergebnis = artikelSumme / anzahlArtikel;
        }
        catch (ArithmetcException ex) {
            ergebnis = 0;
        }
        return ergebnis;
    }
    ←
    Angabe, welche Exception
    abgefangen werden kann
}

```

Quelle: erstellt im Auftrag der IU, 2013.

Das obige Codebeispiel zeigt zwar den Einsatz der neuen Sprachelemente, profitiert aber noch nicht von allen Vorteilen der expliziten Ausnahmebehandlung. Hilfreicher als das bloße Zurückgeben des Wertes 0 ist die Weiterleitung der Exception an das aufrufende Programm. Das macht insbesondere dann Sinn, wenn das aufrufende Programm (hier: der Online-Shop) die Verantwortung zur Behandlung der Exception erhalten soll. Deswegen wird im Folgenden die zweite Alternative behandelt, wie in Java mit Exceptions umgegangen werden kann: das Weiterleiten von Exceptions.

Damit Methoden Exceptions weiterleiten können, muss ihre Signatur um das Schlüsselwort **throws** erweitert werden (siehe Abbildung 81). Es folgt die Auflistung aller Exceptions, die in der Methode auftreten können (hier: `ArithmetcException`). So können aufrufende Programme anhand der Signatur erkennen, welche Ausnahmen von dieser Methode zu erwarten sind. Die Ausnahmebehandlung per try/catch-Block muss in das aufrufende Programm verlagert werden. Dadurch wird die Beispiel-Methode `preisProArtikel()` wieder deutlich übersichtlicher.

throws

Dieses Schlüsselwort zeigt in der Methodensignatur an, welche Exceptions von der Methode geworfen werden können.

Abbildung 81: Zweite Variante: Auffangen einer weitergeleiteten Exception

```
public class Warenkorb {  
    ...  
    public double preisProArtikel() throws ArithmeticException{  
        return artikelSumme / anzahlArtikel;  
    }  
}  
Gibt aufrufendem Programm bekannt,  
welche Exceptions hier auftreten können  
  
-----  
public class OnlineShop {  
    public void zeigeStatistik(Warenkorb w) {  
  
        Das aufrufende → try {  
            Programm muss per double ppa = w.preisProArtikel();  
            try/catch-Block für System.out.println(ppa);  
            die Ausnahmebe- }  
            handlung sorgen ... catch (ArithmeticException ex) {  
                ... System.out.println(ex.getMessage());  
                ...  
... sonst bricht das }  
Programm ab. }  
Es gibt im Ausnahmefall die  
Fehlermeldung aus.  
  
Exception in thread "main" java.lang.ArithmetiException: / by zero  
at ausnahmebehandlung.Warenkorb.preisProArtikel(Warenkorb.java:76)  
at ausnahmebehandlung.OnlineShop.main(OnlineShop.java:13)
```

Quelle: erstellt im Auftrag der IU, 2013.

Als aufrufendes Programm ist es nun die Aufgabe des Online-Shops, die Ausnahmen zu fangen und zu behandeln. Um einen möglichen Programmabbruch wie oben abgebildet zu verhindern, wird der Aufruf der Methode `preisProArtikel()` jetzt in einen `try/catch-Block` eingebettet. Im Ausnahmefall wird über die Methode `getMessage()` des `Exception`-Objekts die Fehlermeldung ermittelt und auf der Konsole ausgegeben. Im Anschluss kann das Programm mit den folgenden Anweisungen fortfahren.



MERKE

Bitte beachten Sie an dieser Stelle, dass nicht alle in Java integrierten Standard-Exceptions behandelt werden müssen. Beispielsweise führt das Weglassen des `try/catch-Blocks` im obigen Beispiel nicht zu einem Compilerfehler, obwohl die Signatur der Methode `preisProArtikel()` darauf hinweist, dass sie eine Exception wirft. Solche Exceptions werden in der Literatur „unchecked“ Exceptions genannt. Man kann sie daran erkennen, dass sie alle Unterklassen von `RuntimeException` sind. Dies trifft zum Beispiel auf alle in Tabelle 18 aufgeführten Exceptions zu. Alle anderen Exceptions, also auch die selbst definierten (mehr dazu im folgenden Kapitel) sind „checked“ Exceptions und müssen abgefangen werden.

throw
Dieses Schlüsselwort ermöglicht das Erzeugen einer Exception.

Das oben aufgeführte Codebeispiel hat einen Nachteil: Die Fehlermeldung ist für den Benutzer des Online-Shops unverständlich, da sie nicht die Ursache (Warenkorb ist leer), sondern nur die Wirkung (Division durch Null) nennt. Viel besser wäre also eine kontextbezogene Fehlermeldung, die auf das tatsächliche Problem hinweist. Daher wird im Folgenden die dritte Alternative vorgestellt: Das Weiterleiten der Exception mit dem Schlüsselwort **throw** und einer spezifischen Fehlermeldung. Hierzu wird in der Methode `preisProArtikel()` die Exception wieder per try/catch-Block gefangen, um dann wieder eine neue `ArithmeticException` zu werfen, die die spezifische Fehlermeldung enthält. Die angepasste Fehlermeldung wird dem Konstruktor der Exception als Parameter übergeben (siehe Abbildung 82).

Abbildung 82: Dritte Variante: Weiterleiten der Exception mit angepasster Fehlermeldung

```
public class Warenkorb {  
    ...  
  
    public double preisProArtikel() throws ArithmeticException {  
        double ergebnis;  
        try {  
            ergebnis = artikelSumme / anzahlArtikel;  
        }  
        catch (ArithmeticException ex) {  
            throw new ArithmeticException("Berechnung unmöglich: " +  
                "Warenkorb ist leer!");  
        }  
        return ergebnis;  
    }  
}
```

Die Ausnahme wird gefangen und erneut geworfen, um eine spezifische Fehlermeldung einzubauen.

Quelle: erstellt im Auftrag der IU, 2013.

Bei der dritten Variante ändert sich beim aufrufenden Programm nichts, da es die deklarierte Exception weiterhin abfangen muss. Lediglich die Fehlermeldung der zu behandelnden Exception hat sich verändert.

Ergänzend sei noch darauf hingewiesen, dass zu einem try-Block auch mehrere catch-Blöcke definiert werden können. Das macht zum Beispiel dann Sinn, wenn ...

- ... im try-Block viele unterschiedliche Exceptions auftreten können und jede dieser Exceptions separat behandelt werden soll oder
- sichergestellt werden soll, dass neben einer bestimmten Standard-Exception auch alle anderen Exceptions abgefangen werden sollen.

Angenommen, die Methode `preisProArtikel()` wäre deutlich komplexer und könnte neben der `ArithmeticException` auch eine `ArrayIndexOutOfBoundsException` auslösen (zum Beispiel weil zur Neuberechnung der aktuellen Artikelsumme auf die Liste der Artikel zugegriffen werden muss). In diesem Fall könnte ein erweiterter try/catch-Block wie in Abbildung 83 aussehen. Wenn eine Ausnahme im try-Block auftritt, wird der Reihe nach überprüft, welcher catch-Block zu der geworfenen Ausnahme passt. Sollte keiner der speziellen catch-Blöcke passend sein, wird die Ausnahme im allgemeinen catch-Block abgefangen, da alle Ausnahmen in Java Unterklassen von `Exception` sind.

Abbildung 83: Erweiterter try/catch-Block

Abbildung 84: Erweiterter try/catch-Block

```
public class Warenkorb {  
    ...  
    public double preisProArtikel()  
        throws ArithmeticException, ArrayIndexOutOfBoundsException {  
    } ...  
}  
  
public class OnlineShop {  
    public void zeigeStatistik(Warenkorb w) {  
        try {  
            double ppa = w.preisProArtikel();  
            System.out.println(ppa);  
        } catch (ArithmeticException ex) {  
        } catch (ArrayIndexOutOfBoundsException ex) {  
        } catch (Exception ex) {  
        }  
    } ...  
}
```

Aussagen zu Abbildung 84:

- Eine weitere Exception kann geworfen werden.
- Nur ausgewählte Exceptions werden gefangen; hier also spezielle Fehler behandeln
- Alle übrigen Exceptions werden abgefangen; hier also alle allgemeinen Fehler behandeln

Quelle: erstellt im Auftrag der IU, 2013.

Die Ausnahmebehandlung mit try/catch-Blöcken bietet viele Vorteile. Allerdings gibt es auch eine bestimmte Situation, in der die bislang bekannten Sprachkonstrukte das Programm verkomplizieren: Wenn nach der Ausführung des try-Blocks in jedem Fall eine bestimmte Programmlogik aufgerufen werden soll (z. B. Aufräumarbeiten), lässt sich das mit den bislang bekannten Mitteln nur sehr umständlich lösen. Denn zusätzlich zum normalen Ablauf ohne Exceptions gibt es gleich drei weitere Möglichkeiten, wie der try-Block verlassen werden kann. In jedem dieser Fälle müsste man dafür sorgen, dass unbedingt notwendige Aufräumarbeiten durchgeführt werden. Um solchen doppelten Code zu vermeiden, gibt es den **finally-Block**. Er wird nach den catch-Blöcken definiert und in jedem Fall ausgeführt – egal ob und welche Exception aufgetreten ist (siehe Abbildung 84).

finally-Block

In diesem werden alle Anweisungen festgehalten, unabhängig davon, ob im try/catch-Block eine Ausnahme aufgetreten ist und behandelt wurde oder nicht. Hier werden meist zwingend notwendige Aufräumarbeiten erledigt.

Abbildung 84: Einsatz des finally-Blocks

```
public static void main(String[] args){  
    ...  
  
    catch (Exception ex) { ← Allgemeine Fehlerbehandlung  
    }  
    finally { ← hier Aufräumarbeiten erledigen, die unabhängig vom  
    }  
}
```

Quelle: erstellt im Auftrag der IU, 2013.

8.3 Definieren eigener Exceptions

Im vorangegangenen Lernzyklus wurde gezeigt, wie man mit den in Java eingebauten Sprachkonstrukten zur Ausnahmebehandlung robustere Programme entwickeln kann. Der Einsatz von try/catch-Blöcken ermöglicht zudem die Trennung von fachlichen Werten und Fehlersignalen. Doch die einzige bislang bekannte Möglichkeit zum Anpassen der verfügbaren Mechanismen war das Werfen einer Standard-Exception mit einer eigenen Fehlermeldung. In diesem Abschnitt soll behandelt werden, wie **eigene Exceptions** programmiert werden können, um noch viel individuellere Ausnahmen erzeugen und abfangen zu können.

Eigene Exceptions
Sie können Programm-spezifische Ausnahmen abbilden. Sie müssen von der Klasse Exception abgeleitet sein.

In dem Beispiel des vorigen Lernzyklus wurde vom Online-Shop die Methode `preisProArtikel()` des Warenkorbs aufgerufen, um dem Kunden Statistiken über den letzten Einkauf anbieten zu können. Die Division durch Null als mögliche Fehlerquelle konnte entschärft werden, indem die dafür vorgesehene Standard-Exception `ArithmeticeException` verwendet wurde. Doch nicht immer gibt es in Java passende Standard-Exceptions. Ein Beispiel hierfür ist der Aufruf der Methode `getMindestbestellwert()` in der Kundenverwaltung des Online-Shops. Die Methode soll eine Ausnahme erzeugen, wenn der Mindestbestellwert negativ ist. Da es sich bei dem Sachverhalt nicht um eine Ausnahme handelt, die von den Standard-Exceptions abgedeckt werden kann, soll nun eine eigene Ausnahme für einen solchen Fall definiert werden: die `MindestbestellwertNegativException` (siehe Abbildung 85).

Abbildung 85: Definition einer eigenen Exception

```
Eigene Ausnahmen müssen – wie Standard-Exceptions auch – von Exception erben. →  
  
public class MindestbestellwertNegativException extends Exception  
{  
    public MindestbestellwertNegativException()  
    {  
        super("Der Mindestbestellwert ist negativ!"); ← Beim Aufruf des Standard-Konstruktors soll eine vordefinierte Meldung ausgegeben werden.  
    }  
  
    public MindestbestellwertNegativException(String message)  
    {  
        super(message); ← Wird dem Konstruktor eine Zeichenkette übergeben, wird diese als Nachricht interpretiert und an den Oberklassen-Konstruktor weitergeleitet.  
    }  
}
```

Quelle: erstellt im Auftrag der IU, 2013.

Jede Ausnahme, egal ob eingebaute Standard-Exception oder eigens definierte Exception, muss von der Oberklasse Exception erben. Mithilfe der Konstruktoren kann festgelegt werden, welche Nachrichten von der Exception erzeugt werden. Wenn zur Erzeugung der Standard-Konstruktor verwendet wird, sollte eine Standard-Fehlermeldung festgelegt werden. Zu diesem Zweck wird der Konstruktor der Oberklasse (Exception) mit der gewünschten Zeichenkette als Parameter aufgerufen. Er interpretiert den übergebenen Parameter als Fehlermeldung und stellt sie über die Methode getMessage() dem aufrufenden Programm zur Verfügung. Wenn man die Ausnahme mit einer erst zur Laufzeit bekannten Fehlermeldung erzeugen möchte, kann man einen weiteren Konstruktor mit Parameter hinzufügen, der die übergebene Fehlermeldung unmittelbar an den Konstruktor der Oberklasse weiterreicht.

Abbildung 86: Werfen einer eigenen Exception

```
public class Kunde {  
    ...  
    public double getMindestbestellwert() throws  
MindestbestellwertNegativException  
    {  
        if (mindestbestellwert >= 0)  
            return mindestbestellwert;  
        else  
            throw new MindestbestellwertNegativException(); ← Ist der Wert negativ, wird die eigene Exception mit der oben definierten Standard-Nachricht erzeugt.  
    }  
}
```

Quelle: erstellt im Auftrag der IU, 2013.

Die Methode getMindestbestellwert() kann jetzt anstelle des Fehlersignals eine MindestbestellwertNegativException werfen (siehe Abbildung 86). In diesem Beispiel wurde die Exception mit der Standard-Fehlermeldung erzeugt. Genau wie bei den

Standard-Exceptions könnte man aber auch hier eine spezielle Fehlermeldung als Parameter übergeben. Im aufrufenden Programm (siehe Abbildung 87), der Methode `getKundendaten()`, muss nun der Rückgabewert nicht mehr nach einem Fehlersignal durchsucht werden. Stattdessen wird der Aufruf von `getMindestbestellwert()` einfach in einen `try/catch`-Block eingebettet. Im Ausnahmefall wird die dazugehörige Fehlermeldung auf der Konsole ausgegeben.

Abbildung 87: Fangen einer eigenen Exception

```
public class OnlineShop {  
    ...  
    public Kunde getKundendaten(int kundenNr) {  
        Kunde k = kundenliste.get(kundenNr);  
        double mbw = 0;  
  
        try {  
            mbw = k.getMindestbestellwert();  
        }  
        catch (MindestbestellwertNegativException ex){  
            System.out.println(ex.getMessage());  
        }  
        ...  
    }  
}
```

Anstelle der Überprüfung des Rückgabewertes auf enthaltene Fehlersignale wird nun die eigene Exception gefangen und behandelt.

Quelle: erstellt im Auftrag der IU, 2013.



ZUSAMMENFASSUNG

Java bietet mit verschiedenen Sprachkonstrukten zur Ausnahmebehandlung ein nützliches Werkzeug zur Programmierung robuster Programme an. Durch einen separaten Kanal für die Signalisierung von Ausnahmen wird gewährleistet, dass Rückgabewerte ausschließlich fachlich korrekte Daten enthalten und frei von Fehlercodes sind. Komplizierte Fallabfragen zur Interpretation dieser Codes und deren Behandlung sind nun überflüssig und können durch einfach strukturierte `try/catch`-Blöcke ersetzt werden.

In Java eingebaute Standard-Exceptions bieten die Möglichkeit zum Auffangen häufig auftretender Fehlersituationen, wie zum Beispiel die Division durch Null oder der Zugriff auf ein nicht initialisiertes Objekt. Falls diese nicht ausreichen, erlaubt Java das Ableiten der Standard-Exceptions durch eigens definierte Exception-Unterklassen mit spezifischen Fehlermeldungen.

LEKTION 9

PROGRAMMIERSCHNITTSTELLEN MIT INTERFACES

LERNZIELE

Nach der Bearbeitung dieser Lektion werden Sie wissen, ...

- was Interfaces sind und in welchen Situationen sie eingesetzt werden.
- welche besonderen Eigenschaften Interfaces im Gegensatz zu Klassen besitzen.
- nach welchen Regeln Interfaces definiert werden können.

9. PROGRAMMIERSCHNITTSTELLEN MIT INTERFACES

Aus der Praxis

Während der Entwicklung des Online-Shops hat Herr Koch der Auftraggeberin in regelmäßigen Abständen Prototypen mit dem aktuellen Entwicklungsstand präsentiert. Dabei hat sich die Auftraggeberin nie wirklich auf eine konkrete Technologie zur Speicherung der Kundendaten festlegen können. Herr Koch musste also die betroffenen Programmkomponenten des Online-Shops immer wieder austauschen. Weil er sich über den dadurch entstehenden Aufwand ärgerte, suchte er nach einer Lösung für sein Problem. Dabei stieß er auf das Konzept der Schnittstellen und beschloss, sich die Sprachkonstrukte von Java zur Definition und Verwendung von Schnittstellen anzueignen.

Seither kann Herr Koch Interfaces in Java nutzen, um austauschbare Lösungen zur Speicherung von Kundendaten im Online-Shop einsetzen zu können. Er kann zudem das Problem der in Java nicht erlaubten Mehrfachvererbung mithilfe von Interfaces lösen und sie dazu einsetzen, nach außen sichtbare Schnittstellen von Softwarekomponenten zu spezifizieren.

9.1 Typische Szenarien für Programmierschnittstellen

Gang of Four

Dies ist eine Gruppe von vier Autoren, die viel zitierte Literatur zum Thema Softwaretechnik verfasst hat.

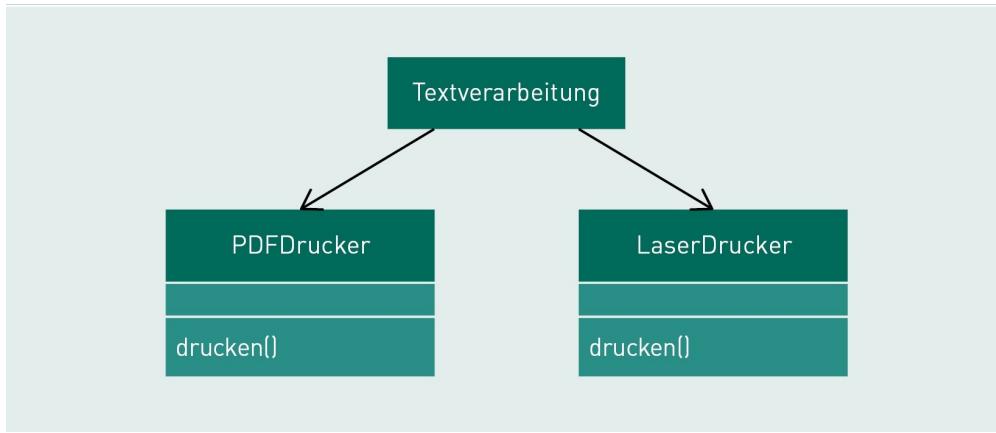
Implementierung

Darunter ist die technische Umsetzung (Programmierung) einer vorgegebenen Funktionalität gemeint. Die Funktionalität kann z. B. durch eine Schnittstelle festgelegt werden.

Bereits seit 1995 proklamiert die sogenannte „**Gang of Four**“ den Einsatz von Interfaces. Es sei ein besserer Programmierstil gegen eine Schnittstelle zu programmieren, als gegen eine Implementierung (vgl. Gamma et al. 1995). Und es gibt tatsächlich zahlreiche Szenarien, in denen der Einsatz dieses Konzeptes sinnvoll erscheint. Denn er sorgt für eine klare Trennung zwischen der Spezifikation und der **Implementierung**. Das heißt, es kann getrennt voneinander festgelegt werden, was bestimmte Klassen oder ganze Pakete können sollen (Spezifikation) und wie diese Funktionalität technisch umgesetzt wird (Implementierung).

Eine solche Trennung bietet zwei Vorteile: Zum einen ermöglicht es eine flexiblere Software-Architektur, da die Implementierung bei Bedarf ausgetauscht werden kann, ohne dass dadurch ein großer Anpassungsaufwand entsteht. Als Beispiel hierfür lässt sich die Druck-Funktion in der Textverarbeitung anführen: Die Schnittstelle für einen Druckertreiber ist klar definiert und bei allen Druckertreiber-Herstellern bekannt. Ein Benutzer kann sich beliebige Treiber installieren und dann bei Bedarf den geeigneten Drucker auswählen – sei es ein PDF-Drucker oder ein echter Drucker (siehe Abbildung 88).

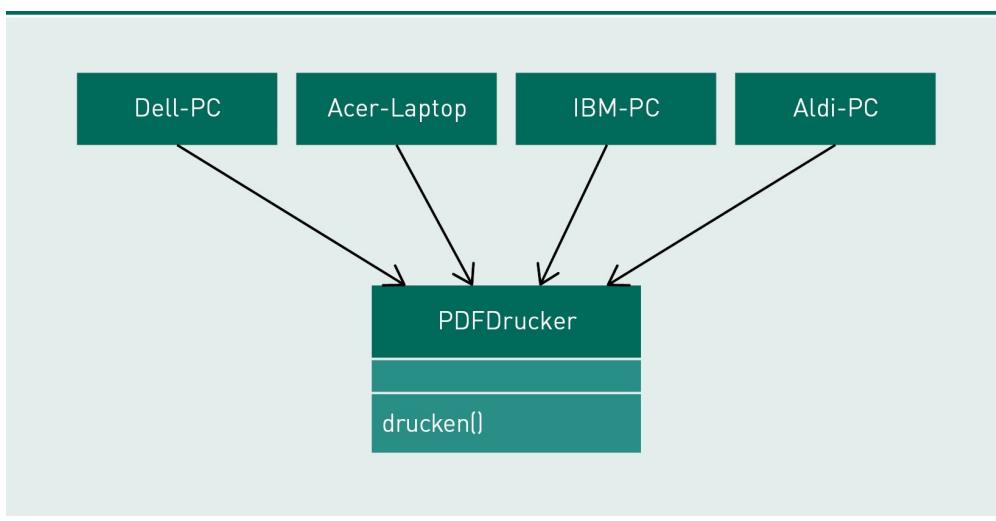
Abbildung 88: Alternative Druckertreiber als Beispiel für flexible Software-Architektur



Quelle: erstellt im Auftrag der IU, 2013.

Der zweite Vorteil der Trennung von Spezifikation und Implementierung liegt darin, dass sie die Wiederverwendung von Klassen oder Paketen erhöht. Eine Klasse, die eine bestimmte Schnittstelle anbietet (implementiert), kann überall dort eingesetzt werden, wo eine solche Funktionalität benötigt wird. Das Programm, in das die Klasse eingebaut wird, muss hierzu nicht angepasst werden. Am Beispiel der Druckertreiber bedeutet das, dass ein Hersteller nicht für jeden PC oder Laptop beziehungsweise deren modellabhängige Ausstattung einen eigenen Druckertreiber programmieren muss. Sofern die Schnittstelle gleich bleibt – und das trifft stets für eine bestimmte Version eines Betriebssystems zu – kann ein Treiber für alle PCs eingesetzt werden, auf denen diese Betriebssystemversion installiert ist (siehe Abbildung 89).

Abbildung 89: Wiederverwendung durch einheitliche Schnittstellen



Quelle: erstellt im Auftrag der IU, 2013.

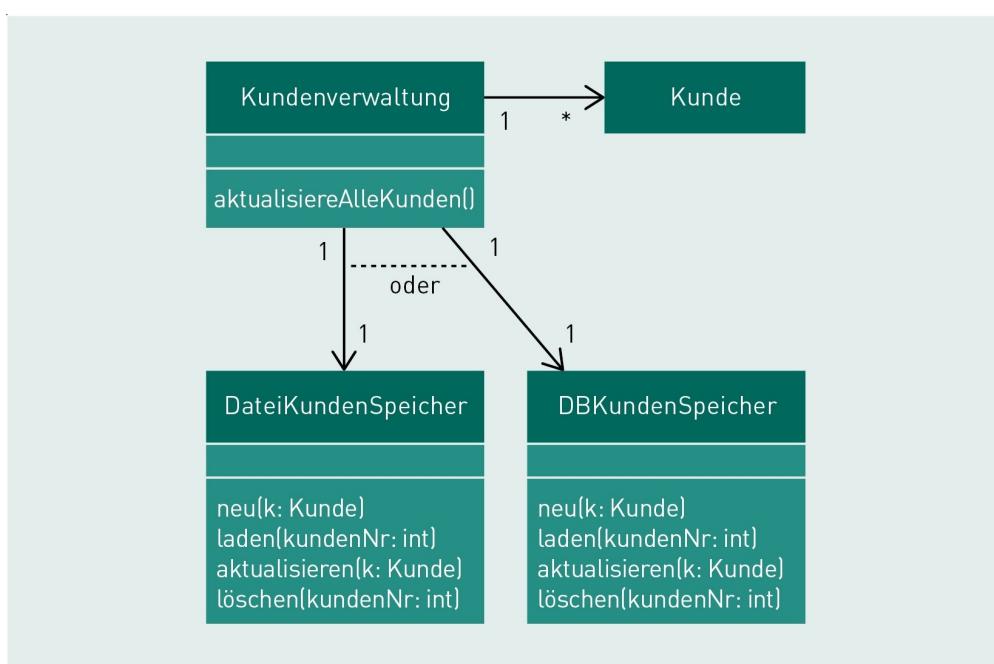


TIPP

Mit der Einführung von Interfaces steht zudem ein weiteres Instrument zur Verfügung, um sich das Konzept der Polymorphie nutzbar zu machen. Im Kapitel 6 haben Sie Polymorphie bereits am Beispiel der Vererbung im Einsatz gesehen. Dort wurde gezeigt, wie die Methode einer Oberklasse von Unterklassen unterschiedlich neu implementiert wurde. Wenn man alle Klassen aus der Vererbungshierarchie in eine gemeinsame Liste speichert und die Methode der Oberklasse aufruft, wird jeweils die passendste Implementierung von der Laufzeitumgebung ausgewählt (dynamisches Binden). Mithilfe von Interfaces kann die Einschränkung aufgehoben werden, dass sich solche Klassen in einer gemeinsamen Vererbungshierarchie befinden müssen. Entscheidend ist nun nur noch, dass sie ein gemeinsames Interface implementieren. Interfaces eignen sich also als noch mächtigeres Instrument für Polymorphie.

Auch am Beispiel des Online-Shops kann man den Zweck von Interfaces zeigen: Angenommen, die Kundenverwaltung benötigt eine flexibel austauschbare Komponente zur dauerhaften Speicherung der Kundendaten. Je nach Bedarf sollen die Daten in einer Datei oder in einer Datenbank gespeichert werden. Mit den bisherigen Mitteln würde man jede Alternative als eigene Klasse programmieren und dann in der Kundenverwaltung verwenden (siehe Abbildung 90). Dabei müsste stets darauf geachtet werden, dass beide Komponenten die gleichen Methoden mit den gleichen Parametern anbieten. Trifft dies bei einer Alternative mal nicht zu, müssten bei jeder Änderung umfangreiche Anpassungen am Programmcode der Kundenverwaltung vorgenommen werden.

Abbildung 90: Beispiel-Szenario für den Einsatz von Interfaces: der austauschbare Kundenspeicher



Quelle: erstellt im Auftrag der IU, 2013.

Im nächsten Lernzyklus wird gezeigt, wie man das Problem in Java mithilfe von Interfaces deutlich eleganter und effizienter lösen kann.

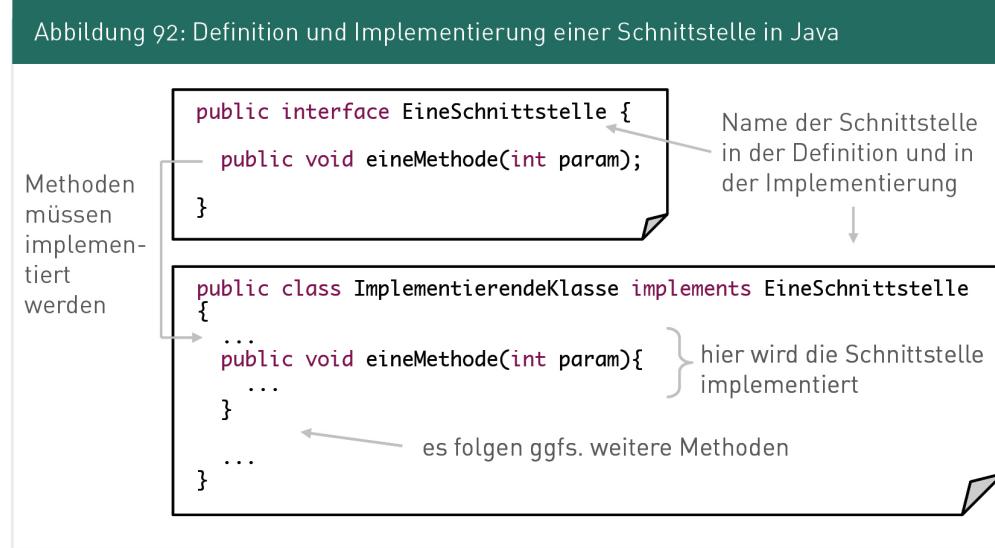
9.2 Interfaces als Programmierschnittstellen in Java

Schnittstellen werden in Java mit dem Schlüsselwort **interface** definiert. Die Syntax erinnert sehr an die einer Klasse, mit dem Unterschied, dass alle Methoden abstrakt sind. Das heißt, die Methoden bestehen lediglich aus der Signatur und haben somit keinen durch geschweifte Klammern eingeschlossenen Rumpf (siehe Abbildung 91). Auf diese Weise können Sie spezifizieren, was eine implementierende Klasse können muss, ohne das „Wie?“ vorwegzunehmen. Bei der Definition von Schnittstellen ist zu beachten, dass sie grundsätzlich zustandslos sein müssen. Es ist daher bis auf Konstanten nicht erlaubt, Attribute zu definieren.

interface

Dieser Begriff definiert eine Menge von Methoden, die von Klassen implementiert werden können. Weil die Methoden erst in den Klassen realisiert werden, findet man in Interfaces nur Methoden ohne einen Rumpf.

Abbildung 91: Definition und Implementierung einer Schnittstelle in Java



Quelle: erstellt im Auftrag der IU, 2013.

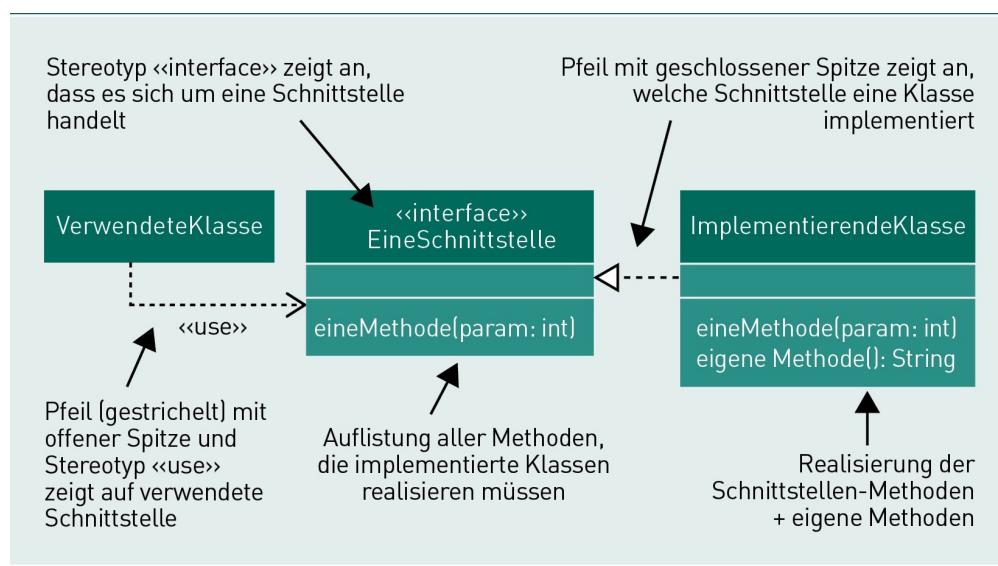
Mithilfe des Schlüsselwortes `implements` kann eine Klasse anzeigen, welche Schnittstelle(n) sie realisiert. Damit verpflichtet sie sich, für jede Methode dieser Schnittstelle(n) eine Implementierung anzubieten.

Auch in der UML ähnelt die Notation von Schnittstellen der von Klassen. Um sie zu unterscheiden, verwendet man das Stereotyp `<<interface>>` (siehe Abbildung 93). Mit einer **use-Assoziation**, dargestellt durch einen gestrichelten Pfeil mit offener Spitze und Stereotyp `<<use>>`, kann gekennzeichnet werden, dass eine Klasse eine Schnittstelle verwendet. Das heißt, die Klasse ruft mindestens eine Methode der Schnittstelle auf.

use-Assoziation

Diese Assoziation kennzeichnet, dass eine Klasse eine Schnittstelle verwendet.

Abbildung 92: Notation von Interfaces in UML

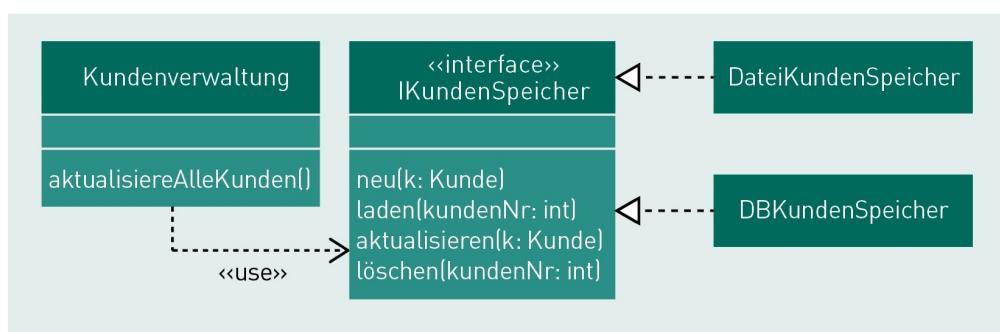


Quelle: erstellt im Auftrag der IU, 2013.

Die implementierende Klasse zeigt, ähnlich zur Vererbungs-Notation, mit einer geschlossenen Pfeilspitze auf die Schnittstelle. Im Gegensatz zur Vererbung wird der Pfeil aber gestrichelt dargestellt.

Mit diesen Möglichkeiten zur Modellierung und Programmierung von Schnittstellen kann das Problem des flexibel austauschbaren Kundenspeichers gelöst werden. Bevor der Java-Code im Detail beleuchtet wird, soll anhand eines Klassendiagramms gezeigt werden, wie die Rollen zwischen den Klassen aufgeteilt sind. Zwischen der Kundenverwaltung und den einzelnen Alternativen zur dauerhaften Speicherung von Kundendaten wird nun ein Interface eingeschoben (z. B. `IKundenSpeicher`). Das Interface definiert eine Reihe von Methoden, die **unabhängig von der Implementierung** für die dauerhafte Speicherung von Kundendaten notwendig sind. Die Kundenverwaltung benutzt dieses Interface in einer ihrer Methoden, spielt also bezogen auf Abbildung 92 die Rolle der verwendenden Klasse. Die Alternativen der Schnittstellen-Implementierung (`DateiKundenSpeicher` und `DBKundenSpeicher`) spielen die Rolle der implementierenden Klassen. Sie bieten je eine Realisierung zu jeder Methode, die in der Schnittstelle `IKundenSpeicher` vereinbart wurde.

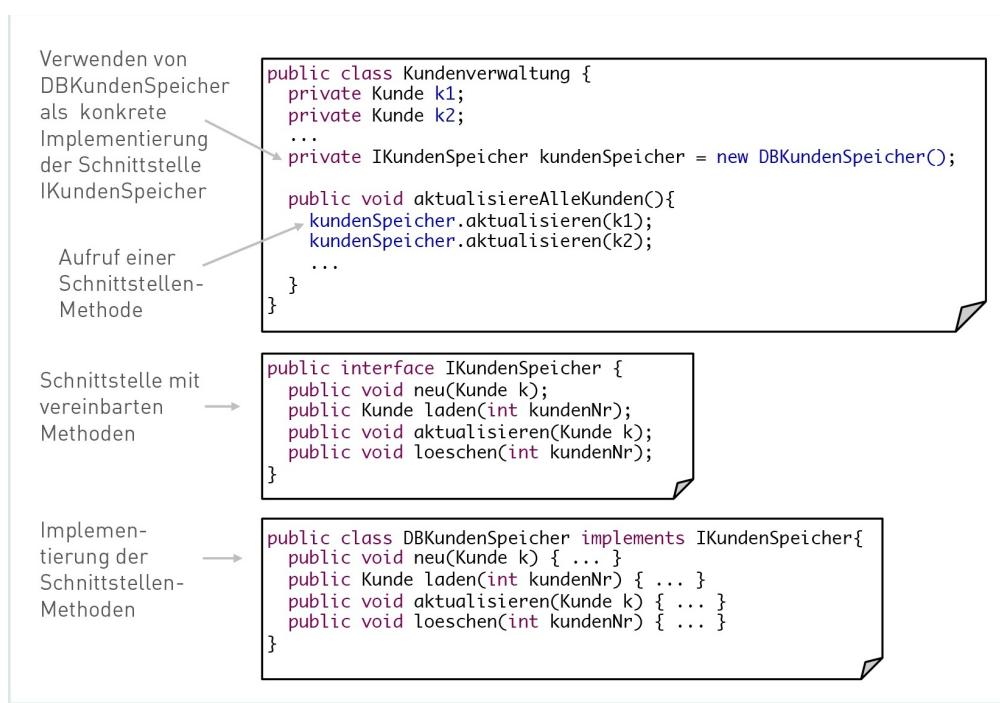
Abbildung 93: Klassendiagramm für den Einsatz von Interfaces im Online-Shop



Quelle: erstellt im Auftrag der IU, 2013.

Einen zum Klassendiagramm passenden Java-Code finden Sie in Abbildung 94. Sie zeigt die Verwendung der Schnittstelle an einer Beispiel-Methode, die Definition des Interfaces sowie eine Beispiel-Implementierung. (Die Anweisungen im Rumpf der implementierten Methoden sind aus Platzgründen abgeschnitten.)

Abbildung 94: Einsatz, Definition und Implementierung eines Interfaces im Online-Shop

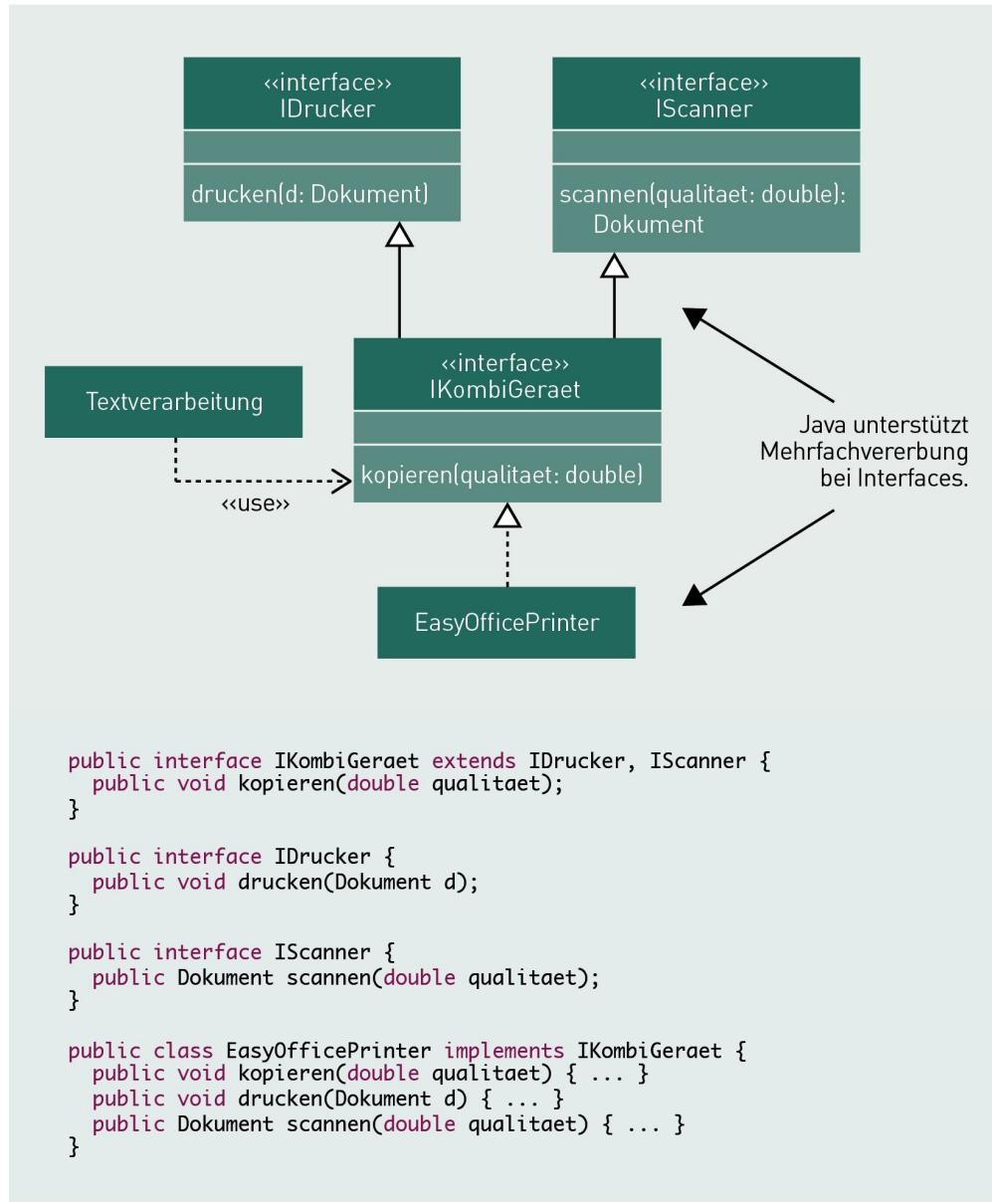


Quelle: erstellt im Auftrag der IU, 2013.

Weil die benötigten Methoden zur Speicherung von Kundendaten nun in einer Schnittstelle definiert sind, können sie ohne großen Aufwand beliebig ausgetauscht werden (Flexibilität). Umgekehrt kann die Implementierung zur Speicherung von Kundendaten prinzipiell an jeder weiteren Programmstelle eingesetzt werden, an der ein solches Interface benötigt wird (Wiederverwendbarkeit).

Die Lektion soll mit weiteren Hinweisen zu Interfaces in Java abschließen. Dazu gehört, dass Interfaces auch von anderen Interfaces erben können. Auf diese Weise können Gemeinsamkeiten von Schnittstellen zusammengefasst und je nach Verwendungszweck ohne großen Wartungsaufwand erweitert werden. Bezogen auf Interfaces unterstützt Java außerdem das Konzept der Mehrfachvererbung. Das heißt, im Gegensatz zu Klassen können Interfaces sogar von mehr als einem Interface erben. Trotzdem wird mit extends das gleiche Schlüsselwort verwendet (siehe Abbildung 95).

Abbildung 95: Vererbung von Schnittstellen am Beispiel eines Drucker-Kombigerätes



Quelle: erstellt im Auftrag der IU, 2013.

Dem aufmerksamen Leser wird zudem nicht entgangen sein, dass die Definition von Interfaces sehr an abstrakte Klassen erinnert (siehe Lektion 6). In der Tat gibt es einige Parallelen: Abstrakte Klassen sind ebenfalls in der Lage, mithilfe abstrakter Methoden Klassenübergreifende Funktionalitäten festzulegen. Im Gegensatz zu abstrakten Klassen spielt bei Interfaces allerdings die Vererbungshierarchie keine Rolle. Das heißt, auch „unverwandte“ Klassen können einem gleichen Interface zugeordnet werden und eine Klasse kann mehr als ein Interface implementieren. Hinzu kommt, dass es bei Interfaces im Gegensatz zu abstrakten Klassen verboten ist, implementierte und abstrakte Methoden zu vermischen. Ebenso wird durch das Verbot Attribute zu definieren verhindert, zustandsbehaftete Interfaces zu programmieren.



ZUSAMMENFASSUNG

Der Einsatz von Schnittstellen sorgt für eine klare Trennung zwischen der Spezifikation und der Implementierung. Es kann also getrennt voneinander festgelegt werden, welche Funktionen bestimmte Klassen anbieten sollen und wie diese Funktionalität technisch umgesetzt wird. Daraus erwachsen zwei Vorteile: Einerseits ermöglicht es eine flexiblere Software-Architektur, da die Implementierung einer Schnittstelle bei Bedarf ausgetauscht werden kann, ohne dass dadurch ein großer Anpassungsaufwand bei den verwendenden Klassen entsteht. Andererseits wird die Wiederverwendung von Klassen oder Paketen erhöht. Denn eine Schnittstellen-Implementierung kann überall dort eingesetzt werden, wo die Funktionalität der Schnittstelle benötigt wird.

ANHANG

LITERATURVERZEICHNIS

- Gamma, E. et al. (1995): *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Verlag, Boston, MA.
- Krüger, G./Stark, T. (2011): *Handbuch der Java-Programmierung*. 7. Auflage, Addison-Wesley Verlag, Boston, MA. (URL: <http://www.javabuch.de> [letzter Zugriff: 23.05.2019]).
- Lahres, B./Raýman, G. (2006): *Praxisbuch Objektorientierung*. Rheinwerk Computing, Bonn. (URL: <http://openbook.rheinwerk-verlag.de/oo/> [letzter Zugriff: 23.05.2019]).
- Oestereich, B. (2012): *Analyse und Design mit der UML 2.5: Objektorientierte Softwareentwicklung*. Oldenbourg Verlag, München.
- Oracle (2013a): *The Java Tutorials*. (URL: <http://docs.oracle.com/javase/tutorial/> [letzter Zugriff: 23.05.2019]).
- Oracle (2013b): *Java Platform Standard Edition 7.0 API Specification*. (URL: <http://docs.oracle.com/javase/7/docs/api/index.html> [letzter Zugriff: 23.05.2019]).
- Oracle (2013c): *Code Conventions for the Java Programming Language*. (URL: <http://www.oracle.com/technetwork/java/codeconventions-150003.pdf> [letzter Zugriff: 23.05.2019]).
- Oracle (2013d): *How to Write Doc Comments for the Javadoc Tool*. (URL: <http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html> [letzter Zugriff: 23.05.2019]).
- ProgrammingGuide (o. J.): *List of Java Exceptions*. (URL: <https://programming-guide.java/list-of-java-exceptions.html> [letzter Zugriff: 23.05.2019]).
- Ratz, D./Scheffler, J./Seese, D. (2011): *Grundkurs Programmieren in Java*. 6. Auflage, Hanser Verlag, München.
- Reviewland (2017): *Kaffeemaschinen Vergleich – Die besten Filterkaffeemaschinen 2017*. (URL: <https://reviewland.website/2017/07/25/kaffeemaschinen-vergleich-die-besten-filterkaffeemaschinen-2017/> [letzter Zugriff: 23.05.2019]).
- Ullenboom, U. (2011): *Java ist auch eine Insel*. 10. Auflage, Rheinwerk Computing, Bonn. (URL: <http://openbook.rheinwerk-verlag.de/javainsell/> [letzter Zugriff: 23.05.2019]).

ABBILDUNGSVERZEICHNIS

Tabelle 1: Übersicht über die geschichtliche Entwicklung von Programmierkonzepten	13
Abbildung 1: Skizze einer Uhr	14
Abbildung 2: Beispielszenario einer einfachen Kaffeemaschine	17
Abbildung 3: Zusammenspiel von Objekten einer Kaffeemaschine	17
Abbildung 4: Erweiterte Kaffeemaschine	18
Tabelle 2: Objekte einer Kaffeemaschine	18
Abbildung 5: Identifizierte Klassen	25
Abbildung 6: Attribute der Klasse „Kunde“	26
Tabelle 3: Eigenschaften von Attributen	26
Abbildung 7: Identifizierte Attribute zu den Klassen	27
Tabelle 4: Eigenschaften von Methoden	27
Abbildung 8: Attribute und Methoden der Klasse „Kunde“	28
Tabelle 5: Typische Beziehungen zwischen Klassen	29
Tabelle 6: Darstellung von Beziehungen	29
Tabelle 7: Multiplizitäten in Beziehungen	30
Abbildung 9: Beispielszenario ergänzt um Beziehungen	32
Abbildung 10: Übersicht über UML-Diagrammtypen	33
Abbildung 11: Objekte und Klassen	34
Abbildung 12: Entwicklung mit der Programmiersprache C	39
Abbildung 13: Entwicklung mit der Programmiersprache Java	39
Abbildung 14: Gerüst der Klasse „Kunde“ aus Abbildung 6	41

Tabelle 8: Grundelemente einer Klasse in Java	41
Abbildung 15: Klasse „Kunde“ erweitert um Attribute	42
Tabelle 9: Grundelemente eines Attributs in Java	43
Abbildung 16: Methoden der Klasse „Kunde“	45
Tabelle 10: Grundelemente einer Methode in Java	45
Abbildung 17: Getter- und Setter-Methoden für die Klasse „Kunde“	48
Abbildung 18: Implementierte Getter- und Setter-Methode für ein Attribut der Klasse „Kunde“	48
Abbildung 19: main-Methode als Startpunkt einer Klasse	51
Abbildung 20: main-Methode im Klassendiagramm	52
Abbildung 21: Beispielimplementierung einer main-Methode	53
Tabelle 11: Primitive Datentypen in Java	56
Tabelle 12: String als Datentyp für Zeichenketten	57
Abbildung 22: Deklaration und Zuweisung von Variablen	59
Abbildung 23: Zusammenfassung von Deklaration und Zuweisung	59
Tabelle 13: Wichtige arithmetische Operatoren	60
Tabelle 14: Wichtige logische Operatoren	61
Tabelle 15: Wichtige Vergleichsoperatoren	62
Abbildung 24: Beispiel für Vergleich primitiver Datentypen	63
Abbildung 25: Beispiel für den Vergleich von Referenzdatentypen	64
Abbildung 26: Gleichheit von Referenzdatentypen	64
Tabelle 16: Verketten von Zeichenketten	65
Abbildung 27: Klassendiagramm mit Klassen „Warenkorb“ und „Kunde“	66
Abbildung 28: Anwendungsfall für bedingte Verzweigungen	67

Abbildung 29: Beispiel für einfache bedingte Verzweigungen	67
Abbildung 30: Anwendungsfall für verschachtelte Verzweigung	68
Abbildung 31: Beispiel für verschachtelte Verzweigungen	69
Abbildung 32: Beispiel für if-else If-else Verzweigung	70
Abbildung 33: Beispiel einer while-Schleife	71
Abbildung 34: Beispiel einer do-while-Schleife	72
Abbildung 35: Beispiel einer for-Schleife	73
Abbildung 36: Beispiel für verschachtelte Kontrollstrukturen	73
Abbildung 37: Pakete des Online-Shops	74
Abbildung 38: Deklaration des Paketes einer Klasse	75
Tabelle 17: Sichtbarkeitsmodifikatoren	75
Abbildung 39: Implementierte Klassen	80
Abbildung 40: Beispielszenario ergänzt um Beziehungen	81
Abbildung 41: Gleiche Attribute werden in der Oberklasse zusammengefasst	82
Abbildung 42: Unterschiedliche Seiten für verschiedene Arten von Artikeln	83
Abbildung 43: Transitive Eigenschaften der Vererbung	84
Abbildung 44: Vererbung von Assoziationen	85
Abbildung 45: Vererbung in Java mittels extends in der Klassendeklaration	86
Abbildung 46: Ableitung der Klasse „Buch“ von der Klasse „Artikel“	87
Abbildung 47: Zugriff auf geerbte Methoden	87
Abbildung 48: Zuweisungskompatibilität	88
Abbildung 49: Verfügbarkeit von Attributen und Methoden abhängig vom Typ der Variable	89
Abbildung 50: Generische Schleife über alle Artikel, die eine Liste der Hersteller erstellt	90

Abbildung 51: Überschreiben einer geerbten Methode	90
Abbildung 52: Aufruf einer überschriebenen Methode	91
Abbildung 53: super verweist auf die Oberklasse	92
Abbildung 54: Zugriff auf Implementierung in der Oberklasse mittels super	92
Abbildung 55: Oberklasse „Artikel“ und deren Unterklassen	97
Abbildung 56: Kennzeichnung der Klasse „Artikel“ als abstrakte Klasse, zwei Varianten ..	97
Abbildung 57: Kennzeichnung einer abstrakten Klasse in Java	98
Abbildung 58: Kennzeichnung einer abstrakten Methode	99
Abbildung 59: Implementierung einer abstrakten Methode durch Überschreiben	99
Abbildung 60: Deklaration von Variablen des Typs einer abstrakten Klasse	100
Abbildung 61: Die unterschiedlichen Arten von Artikeln in der Klassenhierarchie	101
Abbildung 62: Auswahl der Implementierung zur Laufzeit	102
Abbildung 63: Verwendung des instanceof-Operator zum Testen auf Klassenzugehörigkeit	103
Abbildung 64: Deklaration und Verwendung eines statischen Attributs	104
Abbildung 65: Deklaration und Verwendung von statischen Methoden	104
Abbildung 66: Statische Attribute und Methoden der Klasse „Math“	105
Abbildung 67: Einsatz des new-Operators in der main-Methode einer Autoverwaltung ..	109
Abbildung 68: Definition und Aufruf des Standard-Konstruktors	110
Abbildung 69: Formular zur Erfassung eines Kundenkontos (links); mögliche Implementierung (rechts)	111
Abbildung 70: Herkömmliche Belegung der Objektattribute durch Setter- Methoden ..	112
Abbildung 71: Beispiel eines überladenen Standard-Konstruktors	113
Abbildung 72: Verwendung eines überladenen Konstruktors als Alternative zum Einsatz vieler Setter-Methoden	113

Abbildung 73: Zweite, reduzierte Eingabemaske (links) mögliche Implementierung (rechts)	114
Abbildung 74: Definition mehrerer überladener Konstruktoren	115
Abbildung 75: Copy-Konstruktor, der eine Kopie eines Kunden-Objektes erzeugen kann	116
Abbildung 76: Erweitertes UML Klassendiagramm des Online-Shop-Beispiels	117
Abbildung 77: Aufruf eines Konstruktors der Oberklasse mit super()	117
Abbildung 78: Beispiel-Szenario für die Ausnahmebehandlung mit Fehlersignalen	121
Abbildung 79: Behandlung von Fehlersituationen mit Signalen	122
Tabelle 18: Vordefinierte Standard-Exceptions in Java (Auszug)	123
Abbildung 80: Auffangen von Standard-Exceptions mit try/catch-Block	124
Abbildung 81: Zweite Variante: Auffangen einer weitergeleiteten Exception	125
Abbildung 82: Dritte Variante: Weiterleiten der Exception mit angepasster Fehlermeldung	126
Abbildung 83: Erweiterter try/catch-Block	127
Abbildung 84: Einsatz des finally-Blocks	128
Abbildung 85: Definition einer eigenen Exception	129
Abbildung 86: Werfen einer eigenen Exception	129
Abbildung 87: Fangen einer eigenen Exception	130
Abbildung 88: Alternative Druckertreiber als Beispiel für flexible Software-Architektur	133
Abbildung 89: Wiederverwendung durch einheitliche Schnittstellen	133
Abbildung 90: Beispiel-Szenario für den Einsatz von Interfaces: der austauschbare KundenSpeicher	135
Abbildung 91: Definition und Implementierung einer Schnittstelle in Java	136
Abbildung 92: Notation von Interfaces in UML	137

Abbildung 93: Klassendiagramm für den Einsatz von Interfaces im Online-Shop 138

Abbildung 94: Einsatz, Definition und Implementierung eines Interfaces im Online-Shop 138

Abbildung 95: Vererbung von Schnittstellen am Beispiel eines Drucker-Kombigerätes . 139

 **IU Internationale Hochschule GmbH**
IU International University of Applied Sciences
Juri-Gagarin-Ring 152
D-99084 Erfurt

 **Postanschrift**
Albert-Proeller-Straße 15-19
D-86675 Buchdorf

 media@iu.org
www.iu.org

 **Hilfe & Kontakt (FAQ)**
Antworten rund um Dein Studium findest
Du jederzeit auf myCampus.