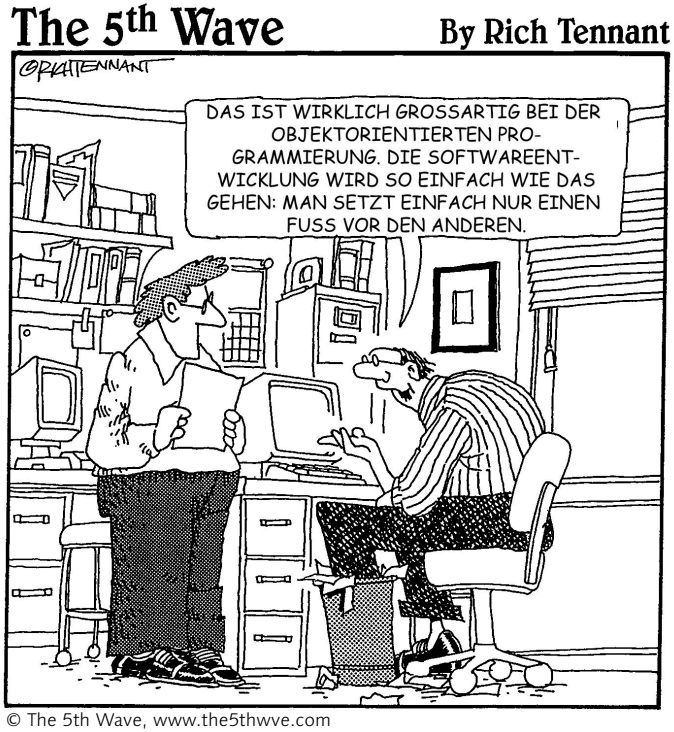
www ...

Tausend Worte oder ein Cartoon.



Beginnen wir mit klassischen W-Fragen zum Content.

Was

Obwohl das erste Kapitel bei Adam und Eva beginnt, sind Inhalt und Darstellung nichts für Anfänger, eher für Umsteiger.[1]

Als Konvertit merkt man schnell, dass Java zwar die Syntax und Semantik der Kontrollstrukturen von *C/C++* geerbt hat, dass damit aber bereits die Vererbung endet.

Englisch lernt man in der Schule. In England oder Amerika angekommen, wird man sofort an der merkwürdigen Art, Sätze zu bilden erkannt, was wiederum zwangsläufig zu Sprachkursen mit dem Tenor »Essential Idioms in English« führt.

1. Die sind wir – ein bestimmtes Alter vorausgesetzt – alle.

- Eine Sprache kann nicht künstlich von ihren Strukturen und Idiomen getrennt werden, die Sache ist von Natur aus symbiotisch.

Spricht man Java wie C++ oder Basic, so mag das zwar der Compiler tolerieren, aber bereits die VM wird bei der Interpretation ins Grübeln geraten.

- Java und essentielle Teile der Plattform werden also anhand der dahinterliegenden Strukturen, Muster und Idiome vorgestellt und kritisch betrachtet.

Wie

Wofür interessiert sich ein Mensch (Leser) in erster, zweiter und dritter Linie?

Die Antwort war bei meinem letzten Verkaufstraining, das leider 25 Jahre zurückliegt, recht unmissverständlich »Für sich selbst«.

- Der Inhalt des Buchs muss sich folglich an den Bedürfnissen und den Fortbildungszielen des Lesers orientieren und nicht am Ego des Autors.

Die Präsentation des Inhalts richtet sich deshalb nach vier Kriterien.

1. Marginalien und Ikonen

Wichtige Sprachregeln, Muster und kritische Anmerkungen werden durch passende Marginalien und Ikonen kenntlich gemacht. Die Buchstaben bedeuten:

Design, ein allgemeines sprachunabhängiges Architektur-Konzept

Pattern, ein sprachunabhängiges Muster zu einem spezifischen Problem

Idiom, ein nur für Java gültiges Muster

Regel, Java-Sprachregel, die – sofern verletzt – zu Compiler- oder VM-Fehlern führt

Tipp bzw. Trap (als Kaktus-Ikone) kennzeichnet einfach nur einen nützlichen Hinweis bzw. eine Falle.

2. UML: Unified Modeling Language

Ideen und Gedanken fasst oder vermittelt man am Besten in Bildern. Selbst Mathematiker »malen«.

Die UML ist ein graphischer Werkzeugkasten, u.a. gut geeignet, Programme in einer normierten, d.h. für andere verständlichen Art zu modellieren.

3. Code-Fragmente

Beispiele sind für die Wissensvermittlung essenziell. Kurze, prägnante Code-Fragmente eignen sich in der Regel besser zum Verständnis als Programmcode, der über viele Seiten geht und leider auch am Leser vorbei.

4. Zertifizierung

Die **Sun-Zertifizierung** [2] – für viele sicherlich ein wichtiges Thema – soll sicherstellen, dass alle Java-Grundlagen synaktisch und semantisch beherrscht werden.

Sie wird von Sun in Form einer Multiple-Choice-Klausur durchgeführt und ist plattform-abhängig, d.h. in ihren Fragen einem steten Wandel unterworfen.

Es ist also im Interesse dieser Leser, betroffene Kapitel mit zertifizierungsrelevanten Fragen abzuschließen. Der Inhalt deckt sich in etwa mit den zur Zeit elf Sektionen, in die der Stoff zur Zertifikation aufgeteilt ist und zu denen Sun Testfragen stellt.[3]

Nein, das Buch ist kein Trainingsbuch, ging nicht, war gegen die Intention!

Es beschränkt sich nicht auf die Vermittlung von Wissen à la »A ist falsch, B und C richtig«. Umgekehrt reicht aber der Inhalt vieler Kapitel locker, um auch **boolsche** Fragen dieser Art beantworten zu können.

Warum

Warum das x-te Buch mit dem bedeutungsschweren Titel **Java 2**? [4]

Es gibt Lehrbücher für Anfänger, es gibt Bücher zu speziellen Themen wie Pattern, Zertifizierung oder Plattform-Bereichen und natürlich die inflationäre Kategorie der **Instant**-Bücher[5].

Mindestens fünf Bücher und dreitausend Seiten lautet eine Alternative. Eine andere besteht vielleicht darin, sich das konzeptionelle Grundwissen zu erarbeiten, um dann erst für die konkreten Aufgaben und Applikationen auf Spezialtitel und das Medium Internet zurückzugreifen.

2. Sun Certified Programmer für die JDK 1.1 oder Java 2-Plattform
3. Wer an der Zertifizierung interessiert ist, sollte sich unbedingt die aktuellen Informationen zur Certification von der Java Developer Connection (java.sun.com) ansehen.
4. An dem Titel bin ich unschuldig.
5. Zum Beispiel „In 22 Tagen zur Herzoperation".

Wer

Wer sollte dieses Buch ignorieren?

- Es ist keine gute Lektüre für die, die möglichst schnell Java lernen müssen, um in drei Wochen ihr erstes professionelles Applet auszuliefern.[6]

Denn ab dem vierten Kapitel werden eher die angesprochen, die nicht (nur) mauern (müssen), sondern das Haus auch entwerfen wollen. Als ergänzende Kursbeilage eignet sich das Buch da schon eher.

Studierende gehören traditionell zur Zielgruppe, wobei drei sogar tatkräftig Korrektur gelesen haben.

Vor allem bin ich Herrn J. Baran zu Dank verpflichtet. Er hat bis zum Schluss mit kritischen Anmerkungen dafür gesorgt, dass das Buch nicht zu früh das Licht der Welt erblickt, was nicht unbedingt im Sinne meiner Lektorin Frau Stevens war, die letztendlich dafür sorgte, dass es überhaupt erschien.

Nun sind Sie als Leser am Zug und ich wünsche Ihnen viel Erfolg!

Hamburg im April 2001

Friedrich Esser

6. Applets werden erst am Ende des letzten Kapitels kurz vorgestellt.

1 Grundlagen

Nach einem Überblick über die »etwas andere« Java-Technologie werden die Details zu den grundlegenden Bausteinen in möglichst einfachen Regeln zusammengefasst.
Konventionen und Stile sind grundlegende Bausteine einer Sprache, also schließt das Kapitel mit Namenskonventionen ab.

Die Zusammenstellung dieser Grundlagen orientiert sich an den Anforderungen der Zertifizierung (siehe Vorwort) und hebt dabei besonders die Aspekte hervor, die Java von C/C++ unterscheiden.

Damit ein weitgehend **goto**-freies Lesen möglich ist, werden verwendete Begriffe wie Interface, Ausnahmen, Array etc. kurz vorgestellt, obwohl sie erst in späteren Kapiteln ausführlich behandelt werden.[1]

Testfragen am Ende des Kapitels

Anhand der abschließenden Fragen kann geprüft werden, ob das Basiswissen für einfache Zertifizierungsfragen ausreicht, und ggf. sollte der zugehörige Abschnitt nachgearbeitet werden.

1.1 Java-Überblick

Java-Technologie: Sprache, Plattform, JVM

Das Wort **Java** steht für eine Technologie, die drei zentrale Komponenten umfasst:

- die eigentliche **Java-Programmiersprache**, die Syntax und Semantik festlegt.
- die **Java-Plattform**, kurz **J2SE** (Java 2 Standard Edition), die auf Basis der Programmiersprache ein umfassendes Klassensystem bereitstellt, um Java-Anwendungen unabhängig vom API[2] des Betriebssystems zu programmieren.
- die **Java Virtual Maschine** (**JVM**), die Java-Code von der realen Maschine unabhängig macht, d.h. den Java-Byte-Code ausführt. Vom Betriebssystem bleibt die JVM natürlich abhängig.

1. Dies ist unweigerlich mit Kompromissen verbunden. Deshalb wird Java-Quereinsteigern empfohlen, parallel eine Einführung mit möglichst vielen Beispielen zu lesen (siehe Literaturhinweise).
2. API: Applications Programming Interface

Die Java-Sprache selbst ist seit der Einführung 1995 bemerkenswert stabil. Aufgrund der Erfahrungen mit **C/C++**, **Smalltalk** und **Oberon** war das Design bereits so ausgereift, dass die Sprache nachfolgend kaum geändert wurde, was man allerdings von den beiden anderen Komponenten (Plattform, JVM) nicht unbedingt behaupten kann.

Die Evolution der Java-Plattform

Obwohl die Sprache die Basis darstellt, liegt die Funktionalität von Java in der Plattform. Sie hat bis zur Version 1.3 gewaltige Änderungen bzw. Erweiterungen erfahren, die zum Teil nur mühsam in das vorherige Klassensystem eingepasst werden konnten. Die Warnung **deprecated**[3] und konkurrierende Funktionalitäten in alten und neuen Klassen belegen dies.

deprecated

Aufgrund der umfangreichen Änderungen wurde ab der Plattformversion 1.2 die offizielle Bezeichnung **Java 2** von Sun eingeführt.

Auch die JVM hat eine ähnlich steile Entwicklung gemacht[4], die hauptsächlich die Effizienz der Programmausführung berührt.

Da die JVM die Verbindung zwischen Betriebssystem, Browser und Java-Plattform darstellt, ist sie der ideale Ansatzpunkt für politische Machtkämpfe.

Die Sun-Initiative »**write once, run anywhere**« hängt entscheidend davon ab, ob die J2SE und die zugehörige JVM in jedem Betriebssystem bzw. Browser installiert sind.

Diversifikation der Java-Plattform

Der Siegeszug von Java hat gleichzeitig eine Schattenseite. Die Java-Technologie muss für PDAs, Handys, Drucker, Smartcards oder Enterprise-Server zwangsläufig adaptiert werden.

Deswegen wurde unter der Parole »**one size doesn't fit it all**«, aber natürlich »**one language is all you need**« bei Sun mit der Diversifikation der Plattformen und der JVM begonnen (siehe auch Abb. 1.1).[5]

Plattform-Abhängigkeit

Obwohl die Sprache an sich davon unberührt bleibt, wird dann jeder Java-Code wieder abhängig, und zwar von der Plattform. Zumindest ein Konkurrent von Sun sieht darin prinzipiell keinen Unterschied mehr zu einer direkten Bindung einer Sprache, z.B. Java, an das API des Betriebssystems.

3. **deprecated**: missbilligt, d.h. ersetzt und verbessert durch Neueres.
4. JIT: Just In Time Compilation, Hotspot Technology etc.
5. J2EE (Enterprise Edition), J2ME (Micro Edition), konfigurierbare virtuelle Maschine, KVM etc.

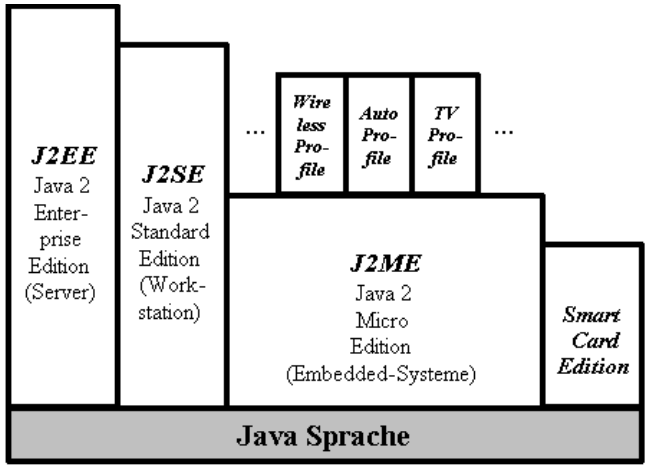


Abbildung 1.1 Java-Plattform-Architektur

1.1.1 Code-Design

Java-Programme sind nicht monolithisch, sondern bestehen aus kleinen autonomen Einheiten, den Klassen (**classes**). Der Compiler (z.B. **javac**) speichert den Byte-Code jeder Klasse in jeweils eigene Dateien. Dies erlaubt es der JVM, Klassen einzeln und nur bei Bedarf (auch über ein Netzwerk) zu laden.

Klassen als ausführbare Einheiten

Um zu starten, muss der JVM nur die Klasse angegeben werden, die sie als erste laden und ausführen soll. Dazu muss diese Startklasse natürlich, abhängig von der Umgebung – Betriebssystem, Browser oder Web-Server – die entsprechende Funktionalität besitzen.

Die Kehrseite dieses Designs besteht darin, dass erst während der Ausführung Konflikte beim Nachladen von Klassen in Verzeichnissen bzw. im Netz entstehen, die bei traditionellen Programmen in der Regel schon vorher beim Linken auftreten.[6]

Klassen gehören immer zu einem **Package**, wobei Java-Packages (über ihre Namen) Hierarchien bilden können. Die J2SE enthält zurzeit ca. 1.500 Klassen, organisiert in 59 Packages.

Um zumindest für eine gewisse Zeit J2SE zu stabilisieren, werden neue Klassen in **Extension Packages** (wie z.B. **Java3D**, **Java Media Framework JMF** etc.) ausgelagert. Dies Pakete gehören damit allerdings auch nicht zu J2SE und müssen bei Bedarf zusätzlich installiert werden.

6. Für die korrekte Zusammenstellung und Distribution von Klassen werden deshalb Deployment-Werkzeuge mit den Entwicklungsumgebungen angeboten.

Interface

Java kennt bereits auf Sprachebene **Interfaces**.

- Ein Interface fasst eine Gruppe zusammengehöriger Methoden unter einem Namen zusammen, der die gemeinsame Aufgabe dieser Methoden charakterisiert.

Interfaces: Rollen, die Objekte spielen können

Ein Interface hat damit den Charakter einer **Rolle**, die ein reales Objekt beherrscht oder nicht. Somit sind Interfaces abstrakt, man kann von ihnen keine Objekte erzeugen.

Eine Klasse kann beliebig viele Interfaces implementieren. Jedes Objekt kann dann z.B. beim Methoden-Aufruf anstelle eines Interface-Parameters stehen, sofern die zugehörige Klasse das Interface implementiert hat.

Damit werden schwerwiegende Defizite der Vererbung unter Beibehaltung der Hauptvorteile wie Substitution behoben.

Methoden und Daten

Keine globalen Funktionen und Daten

Für C/C++-Konvertiten ist die Tatsache ungewohnt, dass es keine globalen Funktionen und Daten außerhalb von Klassen gibt. Der auszuführende Code ist immer in den Methoden (**Methods**) der Klassen enthalten.

Aufruf oder Zugriff mit Hilfe des dot-operators

Da Methoden und Daten an Klassen oder deren Objekte gebunden sind, erfolgt der Aufruf einer Klassen-/Objekt-Methode oder der Zugriff auf Daten von außen über den zugehörigen Klassen- bzw. Objektnamen, separiert durch den Punkt (**dot-operator**)[7] vom Methoden-/Datennamen:

```
Klasse.methode()   bzw.   objekt.methode()

Klasse.feld        bzw.   objekt.feld
```

Die Klammern müssen bei Methoden immer angegeben werden, auch wenn sie keine Parameter haben. Damit ist ein Methoden-Aufruf leicht von einem Datenzugriff zu unterscheiden.

Arrays

Wie bei C/C++ werden Arrays als geordnete Kollektion von Elementen mit gleichem Typ durch eckige Klammern deklariert, wobei auch der Zugriff auf ein Element über den Index in eckigen Klammern erfolgt.

7. Der dot-operator ist an sich kein Operator, sondern ein Separator (siehe 1.4.5).

Damit sind allerdings die Gemeinsamkeiten mit C/C++ auch schon erschöpft. Denn bereits die bevorzugte Art, in Java Arrays zu deklarieren, stellt sich folgendermaßen dar:

Deklaration eines Arrays

- Angabe des Typs der Elemente, gefolgt von einer eckigen Klammer

```
byte[] bArr;   String[] sArr;
```

Da die Angabe der Größe bei der Deklaration nicht erlaubt ist, kann die Array-Variable ein Array beliebiger Länge referenzieren. Diese Art der Deklaration kann auch für die Rückgabe von Arrays aus einer Methode verwendet werden (was C/C++ ohnehin nicht erlaubt).

Anlage eines Arrays mittels new

Mit der o.a. Deklaration ist noch kein Array angelegt. Dies geschieht normalerweise mittels eines **new**-Ausdrucks und kann direkt bei der Deklaration oder auch zu einem späteren Zeitpunkt erfolgen:

```
byte[] bArr= new byte[10];
```

Die Variable `bArr` referenziert nun ein Array mit zehn Elementen vom Typ `byte` (Näheres zu `byte`: siehe 1.3).

Für kleine Arrays gibt es eine sehr praktische Kurzform, die neben der (impliziten) Anlage des Arrays bereits den einzelnen Array-Elementen Werte zuweist:

```
int[] iArr= {1,2,3};
```

Dies ist äquivalent zu:

```
int[] iArr= new int[3];
iArr[0]=1; iArr[1]=2; iArr[2]=3;
```

1.1.2 Java-Applikation

Applications, Applets, Servlets

Java-Code, der direkt (mittels einer JVM) im Betriebssystem ausgeführt wird, nennt man Applikation (**Application)**, innerhalb eines Browsers **Applet** und innerhalb eines Web-Servers **Servlet** oder **Java Server Pages** (JSP).[8]

Die JVM muss bei jeder Startklasse einer **Applikation** natürlich einen ihr bekannten Einstiegspunkt haben. Dieser Einstiegspunkt ist die Methode `main()`. Sie muss immer wie folgt definiert werden:

```
public static void main(String[] args) { ... }
```

8. Sehr gebräuchlich ist die Bezeichnung **Java App** für eine Applikation oder ein Applet.

Auf diese Weise kann jede Klasse zum Testen ihrer Funktionalität mittels `main()` (temporär) ausführbar gemacht werden, selbst wenn sie außerhalb des Packages nicht von anderen Klassen benutzt werden kann.

Beispiel einer ausführbaren Klasse

Mittels des **String-Arrays** `args` können Argumente für die Programmausführung übergeben werden:

```
class StartableClass {
  public static void main(String[] args) {
    // Feld length enthält die Länge des Arrays
    for (int i=0; i < args.length; i++)
      // Methode print() des Feldes out der Klasse System
      System.out.print(args[i]+" ");
  }
}
```

Diese Klasse kann nun – unabhängig von ihrem eigentlichen Zweck – als eigenständiges Programm, genauer als Applikation von der JVM (z.B. **java)** ausgeführt und getestet werden, z.B. mit vier Strings:

```
java StartableClass 2 good 4 u
```

Dies erzeugt auf der Textkonsole das nützliche Echo[9]:

```
too good for you
```

1.2 Programmstruktur

Aufgrund der Konzeption von Java als Internet-Sprache war es zwangsläufig notwendig, Java mit einer konsistenten Konvention für die Programmstruktur auszustatten. Dies zeigt sich u.a. in der J2SE:

- Modularisierung des Codes mit Hilfe des Package-Konzepts
- Ein hierarchisches Namenssystem, das es ermöglicht, Namenskollisionen global zu vermeiden

Die Konvention beinhaltet eine Anleitung zur Abbildung der Package- und Klassennamen in die Verzeichnisstrukturen und Dateien unterschiedlicher Betriebssysteme. Damit müssen natürlich die Namen von Packages **gültige** Verzeichnisnamen bei allen Betriebssystemen sein.

9. Allerdings nur, wenn `StartableClass.class` mittels classpath(-Option) auch gefunden wird (siehe auch 1.2.5) und die JVM diese Art von Humor versteht.

1.2.1 Packages und Namespace

Nach dem Vorbild von **Modula** werden alle Klassen in Java-**Packages** gekapselt. Ein Package besteht aus einer Kollektion von:

Package: Kollektion zusammengehöriger Klassen/ Interfaces

- Klassen
- Interfaces (Schnittstellen)
- Subpackages

Mindestens besteht es aus einem dieser Elemente. Ein Package kann also durchaus nur eine »Hülle« für Subpackages sein, d.h. selbst keine Klassen oder Interfaces enthalten.

Ein Package bildet für die enthaltenen Mitglieder einen so genannten **Namensraum** (**namespace**).

Namespace

Innerhalb des Namensraums müssen die Namen der Mitglieder eindeutig sein, außerhalb können sie immer über den vorangestellten Package-Namen identifiziert werden.

Eindeutigkeit der Namen

Damit wird das Problem der Eindeutigkeit auf die Stufe der Packages verlagert, d.h., für eine Internet-Anwendung muss beispielsweise gewährleistet sein, dass Package-Namen eindeutig sind.

Sun hat sich für die Packages aus dem eigenen Haus die Namen beginnend mit `java`, `javax` oder `sun` reserviert.

Konvention für eindeutige Package-Namen

Der Rest der Welt generiert eindeutige Bezeichnungen anhand der eigenen global eindeutigen Internet-Domain-Namen. Nach Java-Konvention kehrt man sie dazu um und schreibt alles durchgängig klein, also z.B.:

Domain Name	Package-Name startet mit	Spezieller Package-Name
MeinName.de	de.meinname	de.meinname.meinpaket
OurCompany.com	com.ourcompany	com.ourcompany.project1.dbms
IBM.com	com.ibm	com.ibm.sf.samples.addressbook

Tabelle 1.1 Package-Namen basierend auf Domain-Namen

Die letzten beiden Beispiele zeigen hierarchisch angeordnete Packages, wobei ein Subpackage-Name vom übergeordneten Package-Namen durch einen Punkt getrennt wird.

Hierarchie mittels Subpackages

Alle Package-Namen des **SanFrancisco** Business-Projekts von IBM beginnen z.B. mit `com.ibm.sf`, wobei Beispiele in einer Gruppe von Subpackages enthalten sind, die mit `com.ibm.sf.samples` beginnen.

simple vs. fully qualified name

Innerhalb eines Packages müssen die so genannten einfachen Namen (**simple names**) von Subpackages, Klassen und Interfaces eindeutig sein. Es können also keine zwei Mitglieder eines Packages den gleichen Namen tragen.

Beim **vollen Namen** (**fully qualified name**) wird der einfache Name um den vorangestellten vollen Package-Namen ergänzt.

Innerhalb des Subpackages `java.awt` gibt es z.B. ein Subpackage mit einfachem Namen `image` bzw. vollem Namen `java.awt.image`.

Java ist case-sensitive

Damit ist u.a. ausgeschlossen, dass in `java.awt` ein weiteres Mitglied, z.B. eine Klasse mit Namen image existiert, wobei allerdings eine Klasse `Image` erlaubt ist, da Java zwischen Groß- und Kleinbuchstaben unterscheidet.

No-Name/ unnamed/Default-Package

Obwohl »gute Sitte«, muss ein Package-Name nicht unbedingt angegeben werden. Fehlt die Angabe, so gehören alle angegebenen Klassen bzw. Interfaces zum **default** bzw. **unnamed** Package.

Da weder die Anzahl der unnamed-Packages noch ihre Abbildung in ein reales Dateisystem festgelegt ist, sollten nur kleine Beispiele bzw. temporärer Code in einem default Package getestet werden.

Für rein lokale Anwendungen kann die erwähnte Namenskonvention sicherlich ignoriert werden, für kommerzielle Anwendungen ist sie sehr hilfreich.[10]

1.2.2 Java-Code-Struktur

Compilation-Unit

Java-Code – genauer eine Übersetzungseinheit (**Compilation-Unit**) – besteht aus folgenden drei Elementen, die exakt in dieser Reihenfolge aufeinander folgen müssen:

Definieren und Importieren von Packages

- einer Package-Deklaration (optional)
- Import-Anweisungen von anderen Packages (optional)
- Klassen bzw. Interface-Definitionen (mindestens eine)

10. Womit auch **public** deklarierte Klassen bzw. Interfaces ausgeschlossen sind, da diese ja gerade zur Benutzung in anderen Packages vorgesehen sind (ohne Package-Name ist die Verwendung von public-Klassen vom aktuellen Verzeichnis abhängig).

Im folgenden Beispiel wird zuerst ein Package deklariert, gefolgt von zwei Importanweisungen (Erklärung siehe 1.2.3) und einer Klasse:

```
package com.company.samples.test;
import java.awt.*;
import java.math.BigInteger;

// Anschließend Klassen und Interfaces
class Test1 {
  ...
}
```

Durch das vorangestellte Schlüsselwort package wird das Package deklariert. Fehlt die Zeile, handelt es sich um das unnamed Package.

1.2.3 Zugriff auf Klassen und Import

Klassen und Interfaces (nicht Subpackages!) im selben Package können sich gegenseitig über den einfachen Namen referenzieren. Darüber hinaus kann auch auf die fundamentalen Klassen und Interfaces von java.lang immer über den einfachen Namen zugegriffen werden.[11]

public erklärte Klassen

Um auf Klassen und Interfaces in anderen Packages zugreifen zu können, müssen diese dort public erklärt sein:

```
public class Test1 { ... }
```

Für den Zugriff auf public erklärte Klassen anderer Packages muss per Default der volle Name verwendet werden.

Import: Vereinfachter Zugriff auf Klassen anderer Packages

Da dies auf Dauer mühselig ist, kann mittels der Importanweisungen auf alle (oder nur eine) Klasse(n) bzw. Interface(s) des angegebenen Packages auch über den einfachen Namen zugegriffen werden.

Im Beispiel von 1.2.2 werden mit Hilfe des Schlüsselwortes import und des Metasymbols * [12] alle Klassen und Interfaces aus java.awt importiert, gefolgt vom Import genau einer Klasse BigInteger aus java.math.

Führt der Import allerdings zu Namenskollisionen, da in zwei Packages die einfachen Namen gleich sind, muss der Konflikt durch Angabe des vollen Namens beseitigt werden.[13]

11. Dies bedeutet den impliziten Import von java.lang.
12. Bedeutung von *: Sequenz von null oder mehr Zeichen, speziell hier aller Klassennamen.
13. Punkte im Package-Namen stellen Verzeichnistrenner des jeweiligen Betriebssystems dar.

1.2.4 Dateiorganisation und Kompilierung

Die Abbildung des Package-Konzepts in ein Verzeichnis- und Dateisystem ist natürlich vom Betriebssystem abhängig. In Java gibt es hierzu die folgenden Regeln, die aber nicht unmittelbar zum Sprachkern zählen und somit durchaus Ausnahmen zulassen:

Regeln zu Packages, compilation unit, Klassen und Dateisystem

1. Eine Übersetzungseinheit kann beliebig viele Klassen und Interfaces enthalten, wovon jedoch höchstens eine `public` erklärt werden kann.
2. Eine Übersetzungseinheit wird in einer Datei mit der Extension `.java` abgespeichert.
3. Der Name der Datei muss den Namen der Klasse bzw. des Interfaces haben, die `public` erklärt wird. Gibt es keine, kann der Name frei gewählt werden.
4. Nach dem Kompilieren der Übersetzungseinheit wird der Byte-Code jeder Klasse bzw. jedes Interfaces in eine separate Datei mit Namen der Klasse bzw. des Interfaces und der Extension `.class` abgespeichert.
5. Die Package-Hierarchie wird in eine entsprechende Verzeichnishierarchie aufgelöst, d.h., die `.java`- bzw. `.class`-Dateien befinden sich in Unterverzeichnissen, die den Package-Namen abbilden.[14]

Nur die vierte Regel muss von allen Java-Entwicklungsumgebungen eingehalten werden. Die anderen gelten für Entwicklungsumgebungen, die dateibasierend sind und zur Speicherung des Programmcodes keine Datenbank verwenden.[15]

Entsprechend der fünften Regel liegt im Beispiel 1.2.2 die Datei `Test1.class` des Packages `com.company.samples.test` bei Windows im Unterverzeichnis `com\company\samples\test` bzw. bei Unix im Unterverzeichnis `com/company/samples/test`.

Diese Regel besagt nicht, dass die `.java`-Dateien im selben Verzeichnis wie die zugehörigen `.class`-Dateien liegen müssen. Denn die Unterverzeichnisse können für `.java`- und `.class`-Dateien verschiedene Ausgangsverzeichnisse (**Root**-Verzeichnisse) haben. [16]

14. Punkte im Package-Namen stellen Verzeichnistrenner des jeweiligen Betriebssystems dar.
15. Die fünfte Regel ist selbst für dateibasierende Entwicklungsumgebungen recht vage.
16. Bei JBuilder 4 liegen Source- und Class-Dateien in unterschiedlichen Verzeichnissen.

Wäre die Klasse Test1 public erklärt, müsste die Übersetzungseinheit, die den Java-Code enthält, nach der dritten Regel in einer Datei Test1.java abgespeichert sein.

1.2.5 Ausführen einer Applikation

Dem Java-Interpreter, z.B. **java**, braucht nicht angegeben zu werden, wo sich die Klassen der Plattform (J2SE) bzw. der Extentions befinden.

Für alle anderen Klassen, die zur Applikation gehören, sind folgende Regeln zu beachten. Der Interpreter sucht

Regeln zur Ausführung von Applikationen

1. per default innerhalb des aktuellen Verzeichnisses die Klassen des unnamed Packages.
2. per default unterhalb des aktuellen Verzeichnisses in einem Unterverzeichnis, das dem Package-Namen entspricht, nach den Klassen des Packages.
3. unterhalb der Verzeichnisse, die entweder mittels der Umgebungsvariable **CLASSPATH** (des Betriebssystems) oder alternativ der Option –classpath (des Interpreters) gesetzt werden.

Die flexibelste Methode ist wohl, mittels –**classpath** beim Aufruf die Unterverzeichnisse anzugeben. Die Angaben in CLASSPATH werden damit überschrieben bzw. ersetzt.

Aufruf einer Applikation unter Windows

Werden z.B. nur Klassen aus dem Package j2buch.kap1 benötigt, wobei die Klasse j2buch.kap1.Test1 ausgeführt werden soll, dann kann die Klasse Test1 nach der zweiten Regel mittels

```
C:\JB\classes> java j2buch.kap1.Test1
```

aufgerufen werden. Dies setzt voraus, dass sich Test1 im Unterverzeichnis C:\JB\classes\j2buch\kap1 befindet.

Nach der dritten Regel kann Test1 auch mittels

```
java –classpath C:\JB\classes j2buch.kap1.Test1
```

aus jedem beliebigen Verzeichnis aufgerufen werden.

Eine weitere Möglichkeit ist die, alle benötigten Klassen in eine **JAR**-Datei test.jar zu packen, die Test1 als **Main-Class**-Attribut enthält, und diese mittels der Option –jar auszuführen, z.B.:

```
java –jar C:\JB\test.jar
```

Ausnahme (Exception):

Abbruch der normalen Programmausführung durch Exceptions

Fehler, die nicht durch den Compiler abgefangen werden können, d.h. erst bei der Ausführung in der JVM auftreten, werden durch »Auslösen einer Ausnahme« (**throwing an exception)** von der JVM signalisiert.

Diese Ausnahme kann dann entweder im Programm durch einen entsprechenden **Exception-Handler** behandelt, d.h. abgefangen werden, oder die JVM bricht die Programmausführung mit einer detaillierten Fehlermeldung (Art/Ort der Exception) ab.

1.3 Primitive Datentypen

Acht primitive Typen

Acht so genannte **primitive Datentypen** sind Bestandteil der Sprache, werden also vom Compiler direkt erkannt. Hierzu zählen ein logischer (**boolean)** Typ, ein Zeichentyp sowie sechs numerische Typen.

Daneben gibt es noch Referenz-, Klassen- und Array-Typen.

Integraler Typ

Zeichen und ganze Zahlen werden unter dem Begriff **integraler** Typ, **float** und **double** unter **Floating-point**-Typ zusammengefasst.

Die primitiven Datentypen sind in der nachfolgenden Tabelle kurz zusammengestellt:

Type	Wertebereich	Default-Wert	Bit-Größe	Anmerkung
boolean	true, false	false	1	
char	\u0000 .. \uFFFF	\u0000	16	unsigned
byte	-2^7 .. 2^7-1	0	8	
short	-2^{15} .. $2^{15}-1$	0	16	
int	-2^{31} .. $2^{31}-1$	0	32	
long	-2^{63}.. $2^{63}-1$	0	64	
float	Float.MIN_VALUE .. Float.MAX_VALUE, Float.NaN, Float.NEGATIVE_INFINITY, Float.POSITIVE_INFINITY	0.0	32	Darstellbare Werte: ±1.402e-45 .. ±3.402e38
double	Double.MIN_VALUE .. Double.MAX_VALUE, Double.NaN, Double.NEGATIVE_INFINITY, Double.POSITIVE_INFINITY	0.0	64	Darstellbare Werte: ±4.94e-324 .. ±1.79e308

Tabelle 1.2 Primitive Typen

Gegenüber C/C++ gibt es einen eigenen **boolean**-Typ, der eine Umwandlung von/nach Integer nicht zulässt, d.h., `0` oder `1` werden als Ersatz für `false` und `true` nicht akzeptiert (siehe 1.5).

Alle numerischen Typen sind **signed**, d.h. erlauben Vorzeichen und haben eine fest definierte Größe, unabhängig von der Maschine bzw. dem Betriebssystem.

Immer mit Vorzeichen (signed)

1.3.1 Regeln zu Operationen mit Zahlen

Nachfolgend die wichtigsten Regeln zu Zahlen-Operationen:

1. Bei Integer-Arithmetik stellt weder der Compiler noch die JVM sicher, dass der Wertebereich ausreicht. Ist das Ergebnis außerhalb des Wertebereichs, ist es schlichtweg falsch.

   Over- bzw. Underflow bei Integer-Werten

2. Nur eine Division durch Null bzw. Modulo Null erzeugt eine **ArithmeticException** (Ausnahme siehe 1.2.5).

   ArithmeticException

3. Floating-Point-Typen kennen die Werte »nicht definiert«, d.h. `NaN` (Not-a-Number) sowie ±Unendlich, d.h. `NEGATIVE_INFINITY` bzw. `POSITIVE_INFINITY`, die Überschreitungen des Wertebereichs abfangen.

   Wertüberschreitung bei Floating-Point-Typen

4. Eine Operation mit einem »nicht definierten« Wert ergibt kein Ergebnis im normalen Wertebereich.

5. Eine Operation mit `NaN` führt immer zum Ergebnis `NaN`.

Beispiel

Berechnung von	Ergebnis	Anmerkung
-0.0 / 0.0	NaN	Undefiniert, d.h. NaN
1.0 / -0.0	NEGATIVE_INFINITY	es gibt eine negative Null
(0.0 / 0.0) * (1.0 / 0.0)	NaN	NaN bleibt immer bestehen
0.0 == -0.0	true	-0.0 ist gleich 0.0 [17]
-0.0/0.0 == -0.0/0.0	false	NaN ist mit **nichts** gleich[18]

Tabelle 1.3 Ergebnisse spezieller Floating-Point-Berechnungen

17. Mit dem Operator == wird auf Gleichheit hin geprüft.
18. Auch nicht mit sich selbst. Der Test auf `Double.NaN` erfolgt z.B. mittels: `Double.isNaN(0./0.)`.

1.4 Lexikalische Grundlagen

Aufbauend auf einem für Java erlaubten Zeichensatz – dem **Unicode** – besteht Java-Source-Code aus folgenden **atomaren** Elementen:

Bestandteile eines Java-Programms

- Whitespaces (Zwischenräume)
- Comments (Kommentare)
- Identifier (Bezeichner bzw. Namen)
- Separators (Trennzeichen)
- Literals (Literale)
- Keywords (Schlüsselworte)
- Operators (Operatoren)

1.4.1 Unicode

Java verwendet als erlaubten Zeichensatz **Unicode**, womit es sich bereits von C/C++ in seinen Grundlagen unterscheidet.

Eine Sprache für Internet-Applikationen muss neben dem Standardzeichensatz für Englisch auch andere Zeichensätze wie Arabisch, Katakana, Griechisch etc. darstellen können.

Unicode-Tabelle

Da Unicode-Zeichen 16-Bit codiert sind, können prinzipiell 65.536 ($=2^{16}$) Zeichen dargestellt werden. Die Unicode-Tabelle ist in Bereiche für verschiedene Sprachen eingeteilt.

Kompatibel zu ASCII, ISO-Latin-1

Im Tabellenbereich von 0...127 bzw. 0...255 ist Unicode identisch mit dem **ASCII**- bzw. **ISO-Latin-1**-Zeichensatz. Für diese Zeichen kann man also das oberste der beiden Bytes ignorieren.[19]

Da man selten eine Tastatur mit 65.536 Tasten zur Eingabe benutzt, gibt es eine spezielle Escape-Sequenz für Unicode-Zeichen (siehe 1.4.6).

1.4.2 Whitespace

Java ist frei formatierbar (free-form)

Die Anzahl Leerstellen, Tabulatoren oder Zeilenumbrüche (Linefeed bzw. Carriage-Return) zwischen **Token** (Symbolen) können beliebig gewählt werden, sie werden ohnehin bei der lexikalischen Analyse durch den Compiler entsorgt.

19. Siehe hierzu auch 10.6.1

1.4.3 Kommentare

Es gibt drei Arten von Kommentaren, die von C++ adaptiert wurden:

- einzeilige:

```
i=i+1; // oder i++; oder ++i; oder i+=1;
```

- mehrzeilige:

```
/* Arrays können im C/C++ oder im
   Java-Stil deklariert werden */
int iarr[]; byte[] barr;
```

- mehrzeilige Kommentare, die mit /** starten. Hieraus kann mit Hilfe von **javadoc** eine HTML-Dokumentation generiert werden. **javadoc** versteht einfache HTML-Formatierungen sowie mit @ beginnende **Tags**, d.h. Hinweise auf Text mit festgelegter Bedeutung:

```
/** money factory class - a Pattern
@author BG
@version 0.9
<U>code name</U>: <B>W98 Green Banana</B>
*/
```

1.4.4 Identifier

Ein **Identifier** ist ein vom Programmierer wählbarer Name für Variablen, Marken, Methoden und Klassen mit folgenden Restriktionen:

- Das erste Zeichen muss ein (Unicode) Buchstabe, ein Unterstreichungsstrich _ oder ein Währungssymbol $, £ bzw. ¥ sein.
- Ab dem zweiten Zeichen sind noch zusätzlich Ziffern erlaubt.
- Schlüsselwörter (siehe Tabelle 1.5) sind nicht erlaubt.

Nachfolgend einige zulässige (erste Zeile) bzw. unzulässige Identifier:

```
_i__  £4  äß  _123  Õre

2i  ab/1  B-1  ab!  a%b  #a  goto
```

1.4.5 Separator

Java verwendet neun Zeichen als Trennzeichen mit besonderer Bedeutung, wobei der häufigste Separator wohl das Semikolon ist, das Anweisungen abschließt. Die restlichen sind verschiedene Arten von Klammern, Komma und Punkt:

```
; , . ( ) { } [ ]
```

1.4.6 Literale

Literale: konstante Werte

Im Gegensatz zu einem Identifier repräsentiert ein Literal einen konstanten Wert eines bestimmten Typs. Literale können in Ausdrücken, Zuweisungen und als Argumente beim Methoden-Aufruf verwendet werden.

Es gibt Literale zu den primitiven Typen, zur Klasse `String`, das `null`-Literal sowie noch Literale der Klasse `Class`, die im Folgenden kurz vorgestellt werden:

Logische Literale

Die einzigen Literale vom Typ `boolean` sind `true` und `false`.

```
boolean ok = false;
```

Zeichen-Literale

Zeichen-Literale müssen immer in Hochkommata eingeschlossen werden.

Hexadezimale Eingabe von Zeichen

Für Zeichen, die nicht mittels Tastatur eingegeben werden können, wird die Unicode-Tabellenposition in Form von (maximal) vier hexadezimalen Ziffern `0..F` (Basis 16) mit dem Präfix \u eingegeben.[200] Die Hex-Ziffern können groß `A..F` oder klein `a..f` geschrieben werden.

Für die Eingabe von speziellen Zeichen gibt es in Java wie C/C++ die nachfolgenden Escape-Codes:

Escape-Code	Zeichen
`'\''`	Hochkomma (single quote)
`'\"'`	Anführungszeichen (double quote)
`'\\'`	Schrägstrich nach hinten (backslash)
`'\b'`	Schritt zurück (backspace)
`'\t'`	Tabulator (tab)
`'\n'`	Zeilenschaltung (linefeed/newline)
`'\r'`	Wagenrücklauf (carriage return)
`'\f'`	Seitenschaltung (form feed)

Tabelle 1.4 Escape-Codes für Sonderzeichen

20. Alternativ gibt es noch eine oktale Eingabe von \000 bis \377 für die ersten 256 Zeichen.

Mit Hilfe des Escape-Codes lassen sich Unicode-Zeichen eingeben:

```
char c1='\u00c4';
System.out.print(c1=='Ä');     // 💻: true
```

Integrale Literale

Eingabe von ganzen Zahlen

Die als **integrale Literale** bezeichneten ganzen Zahlen vom Typ `int` oder `long` können dezimal, hexadezimal oder sogar oktal eingegeben werden, wobei die hexadezimale Eingabe mit `0x` oder `0X` und die oktale mit `0` beginnen muss.

Alle Literale sind per default vom Typ `int`. Wird an das Literal (das Suffix) `l` oder `L` angehängt, ist es vom Typ `long`.

```
int i= 0x10+10+010;               // 16+10+8
System.out.print(i+" "+0x10L);    // 💻: 34 16
```

Floating-Point-Literale

Eingabe von Dezimalzahlen

Ein **Floating-Point-Literal** vom Typ `float` oder `double` ist eine Zahl mit Dezimalpunkt und/oder einem angehängten Exponent und/oder einem Suffix `f` oder `F` für `float` bzw. `d` oder `D` für `double`. Fehlt das Suffix, ist das Literal per default vom Typ `double`.

Der Exponent beginnt mit `E` oder `e`, gefolgt von einem optionalen Vorzeichen und dem Exponenten (möglicher Wertebereich siehe Tabelle 1.2). Damit sind alle nachfolgenden Zahlen gültige Floating-Point-Literale:

```
0.  .0     +1f    -6e+1     7.1E-1F
```

Dabei ist die letzte Zahl gleich `0.71` und vom Typ `float` .

String-Literale

Strings sind Zeichensequenzen und werden immer in Anführungszeichen (double quotes) gesetzt. Innerhalb des Strings können die einzelnen Unicode-Zeichen (wie bei Zeichen-Literalen) per Escape-Code oder hexadezimal eingegeben werden.

Die Anweisung

```
System.out.print("\"\u00c4\tÖ\tÜ\"\nNeue Zeile");
```

gibt (unter Windows) zwei Zeilen auf der Konsole aus:[21]

21. Eine mittels \n hart kodierte »neue Zeile« ist betriebssystemabhängig. Besser ist `println()`.

```
"Ä    Ö    Ü"
Neue Zeile
```

String-Literale sind Objekte

String-Literale sind keine Werte vom primitiven Typ, sondern sind eine kurze elegante Art, Objekte der Klasse `String` anzulegen.

Somit können auf String-Literale alle Instanz-Methoden der Klasse `String` angewendet werden, wie z.B. die Methode `length()`, die die Länge des Strings zurückgibt:

```
System.out.print("".length()); // 💻: 0
```

Literal null

null: Kein Objekt referenziert

Neben den Variablen vom primitiven Typ gibt es noch **Referenzvariablen**, die auf Objekte von Klassen zeigen. Um anzuzeigen, dass eine Referenzvariable auf **kein** Objekt zeigt, wird das Literal **null** (ein Schlüsselwort) verwendet.

```
String s= null; // s zeigt auf kein String-Objekt
System.out.println(s);           // 💻: null
System.out.println(s.length());  // Exception
```

NullPointer-Exception

Die letzte Code-Zeile führt nicht zu einem Compilerfehler, sondern bei Ablauf des Programms zu einer **NullPointerException**, da von einem nicht existierenden Objekt keine Länge abgefragt werden kann.

Literale zur Klasse Class [22]

Class-Literale enthalten Klassen-Informationen

Die Klasse `Class` enthält Objekte zu allen Datentypen (inkl. sich selbst). Durch Anhängen von `.class` hinter einem beliebigen Typ kann man ein zu diesem Typ zugehöriges Literal vom Typ `Class` schaffen.

```
Class intTyp= int.class;
```

Dies kann u.a. zur Untersuchung von Klassen verwendet werden. Das nachfolgende Code-Fragment gibt alle Methoden der Klasse `String` aus (die Klasse `Method` ist aus Package `java.lang.reflect` zu importieren):

```
Class stringType= String.class;
Method[] methodArr= stringType.getMethods();
for (int i=0; i<methodArr.length; i++)
  System.out.println(methodArr[i]);
```

22. Sie werden hier nur der Vollständigkeit halber erwähnt. Zum Verständnis des Kapitels sind sie nicht notwendig.

1.4.7 Schlüsselwort

Keyword: Identifier mit fester Bedeutung

Schlüsselwörter sind vordefinierte Identifier mit einer speziellen Bedeutung. Sie dürfen deshalb auch nicht als normale Identifier verwendet werden.

Es gibt zurzeit 59 Schlüsselwörter, die anhand ihrer Funktion in Kategorien eingeteilt werden können. Zum Beispiel gibt es für Bedingungen die Schlüsselworte `if`, `else` und `switch` und eine Gruppe von elf Schlüsselwörtern, die zwar reserviert (®) sind, aber nicht benutzt werden.

abstract	const ®	float	int	protected	throw
boolean	continue	for	interface	public	throws
break	default	future ®	long	rest ®	transient
byte	do	generic ®	native	return	true
byvalue ®	double	goto ®	new	short	try
case	else	if	null	static	var ®
cast ®	extends	implements	operator ®	super	void
catch	false	import	outer ®	switch	volatile
char	final	inner ®	package	synchronized	while
class	finally	instanceof	private	this	

Tabelle 1.5 Java-Schlüsselwörter

1.4.8 Operatoren

Operatoren sind spezielle Symbole für Operationen auf Operanden. Je nach Anzahl der Operanden, auf die der Operator angewendet wird, unterscheidet man unäre und binäre Operatoren sowie einen ternären Operator.

Java kennt 37 Operatoren, die anhand ihrer Funktion klassifiziert werden, wie z.B. arithmetische oder logische Operatoren. Operatoren werden in Kapitel 2 behandelt.

1.5 Konvertierung primitiver Typen

Numerische Typen können bei Bedarf ineinander konvertiert werden. Selbst der Typ `char` ist im Prinzip eine nicht negative Zahl zwischen $0..2^{16}-1$ und kann daher in eine ganze Zahl umgewandelt werden.

Keine Konvertierung von/nach boolean

Im Gegensatz zu C/C++ gibt es keine Konvertierung eines primitiven Typs von/nach `boolean`.

Konvertierung: widening vs. narrowing

Es gibt zwei Arten der Konvertierung:

- die **widening Conversion**, die Konvertierung in einen Typ mit einem größeren Wertebereich (siehe Tabelle 1.2)
- die **narrowing Conversion**, die Konvertierung in einen Typ mit einem kleineren Wertebereich

1.5.1 Widening Conversion

Eine **widening Conversion** wird vom Compiler automatisch, d.h. implizit durchgeführt. Nachfolgend wichtige Regeln zur dieser Art der Konvertierung:

Initialisierung bzw. Zuweisung bei byte, short und char

1. Literale vom Typ `char` oder `int` können Variablen vom Typ `byte`, `short` oder `char` zugewiesen werden, sofern sie im erlaubten Wertebereich liegen (wenn nicht, führt dies zu einem Fehler bei der Kompilierung, kurz C-Fehler).
2. Ansonsten kann eine implizite Konvertierung nur in Richtung der Pfeile (siehe Abb. 1.2) erfolgen.
3. Eine automatische Konvertierung findet – sofern notwendig – bei der Übergabe von aktuellen Argumenten an Methoden und bei Zuweisungen statt.

Eventueller Verlust an Genauigkeit

4. Bei Konvertierung einer `int` oder `long` nach Floating-Point kann nicht der Erhalt der Genauigkeit garantiert werden.

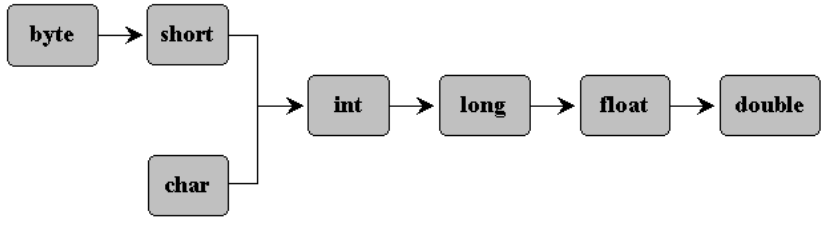


Abbildung 1.2 Richtung der impliziten Typumwandlung

Nach der zweiten Regel kann also der Typ `byte` vom Compiler z.B. in ein `long` konvertiert werden, aber nicht umgekehrt.

Die Typen `byte` und `short` können nicht automatisch in `char` sowie `char` nicht in `byte` und `short` umgewandelt werden.

Beispiele

```
byte b= 'a';  // siehe Ausnahme: wie b= 97;

char c1= 97;  // siehe Ausnahme: wie c= 'a';

char c2= b;   // C-Fehler: kein widening conversion

int i= 1L;    // C-Fehler: kein widening conversion

b= i;         // C-Fehler: kein widening conversion

b= 128;       // C-Fehler: siehe Ausnahme
```

Wird nach der vierten Regel z.B. ein `int` (32 Bit) in eine `float` (32 Bit) umgewandelt, so bedeutet dies evtl. einen Verlust an Genauigkeit, da `float`, bedingt durch den zusätzlichen Exponenten, eine maximale Genauigkeit von nur sieben Dezimalziffern hat:

```
int i=1234567891;
float f= i;
System.out.println(f);   // 💻: 1.23456794E9
```

Numeric Promotion bei Operationen

Operationen mit unterschiedlichen Typen

Nehmen zwei unterschiedliche primitive Typen (siehe Abb. 1.2) an einer Operation teil, werden sie vom Compiler vor der Operation auf einen gemeinsamen Typ konvertiert.

Dies nennt man auch **numeric Promotion** und läuft bildlich nach folgender Regel ab:

Regel für die implizite numerische Umwandlung

- Bei der Konvertierung wird der Typ gewählt, der am weitesten rechts in der Abbildung 1.1 steht, wobei man mit `int` beginnen muss.

Übersetzt in die vier »normalen« Regeln heißt das:

1. Ist einer der Operanden vom Typ `double`, wird der andere in ein `double` konvertiert.
2. Ansonsten: Ist einer der Operanden vom Typ `float`, wird der andere in ein `float` konvertiert.
3. Ansonsten: Ist einer der Operanden vom Typ `long`, wird der andere in ein `long` konvertiert.
4. Ansonsten: Beide Operanden werden in ein `int` konvertiert.

Bei dem nachfolgenden Code-Fragment führen `i*i` bzw. `i*i*1.` zu einem falschen Ergebnis.[23] Die letzte Berechnung ist nach der ersten Numeric-Promotion-Regel allerdings wieder richtig:

Die Reihenfolge der Operationen entscheidet

```
int i= 1000000;
System.out.println(i*i);     // 🖳: -727379968
System.out.println(i*i*1.); // 🖳: -7.27379968E8
System.out.println(1.*i*i); // 🖳: 1.0E12
```

1.5.2 Narrowing Conversion und Casting

Eine den Wertebereich einengende numerische Konvertierung ist nicht implizit, sondern nur explizit möglich.

Dazu stellt man einfach den gewünschten Typ in Klammern vor den zu konvertierenden Wert (**Cast**).

Fehler beim Cast

Cast: Explizites Konvertieren ohne Gewähr

Es ist allein die Aufgabe des Programmierers sicherzustellen, dass die Daten beim **Cast** nicht falsch werden, da z.B. im integralen Bereich die obersten Bits einfach abgeschnitten werden.

```
int i= 0xFFFF;          // i hat den Wert 65535
short s= (short) i;     // s hat den Wert -1
byte b1= 0, b2= 1, b3;
b3= b1+b2;              // C-Fehler [24]
b3= (byte)(b1+b2);      // die 2. Klammer ist notwendig
float f= 0.0;           // C-Fehler, richtig: f= 0.0F
```

Runden beim Cast

Ein Floating-Point-Typ wird in einen integralen Typ durch Abschneiden der Stellen hinter dem Dezimalpunkt umgewandelt:

```
System.out.println((int)10.9999); // 🖳: 10
```

23. Gemäß Regel zur Integer-Arithmetik (siehe 1.3) wird die Überschreitung des Wertebereichs bei `int` und `long` nicht erkannt. Die Operationen werden von links nach rechts ausgeführt (siehe Kapitel 2, Operatoren).
24. Nach der vierten Regel wird mit `byte`, `char` und `short` nicht gerechnet, sondern mit `int`. Das Ergebnis ist also vom Typ `int` und muss explizit per Cast in Typ `byte` oder `char` umgewandelt werden (siehe zu arithmetischen Operationen auch Abschnitt 2.2).

1.6 Initialisierung von Variablen

Java kennt zwei Arten von Variablen:

Member bzw. Felder vs. lokale Variable

- **Member-Variablen**, d.h. Variablen, die als Mitglieder bzw. **Felder** (**fields**) einer Klasse deklariert werden,
- **lokale Variablen**[25] (auch **automatische** Variable genannt), die lokal in einer Methode deklariert sind.

Statische vs. Instanz-Variable

Bei Feldern unterscheidet man zwischen

- **statischen** Variablen, auch Klassen-Variablen genannt (**static** bzw. **Class Variable**), die es unabhängig von den Objekten – den Instanzen von Klassen – nur einmal pro Klasse gibt. Sie erkennt man an dem vorangestellten Schlüsselwort `static`.
- **Instanz-Variablen** (**Instance Variable**), die es für jedes Objekt gibt.

Im Folgenden wird der Begriff **Feld** im Deutschen für Member-Variablen verwendet.

Ablauf und Werte der Initialisierung

Für die Initialisierung gelten folgende Regeln:

1. Felder sowie die Elemente eines Arrays werden immer automatisch initialisiert, logische Variablen mit `false`, primitive mit `0` und Referenz-Variablen mit `null`.
2. Die Initialisierung der statischen Variablen findet beim Laden der Klasse statt, bei Instanz-Variablen diyrekt nach der Erschaffung des Objekts[26] (d.h. vor der Ausführung des Codes im Konstruktor).
3. Lokale Variablen werden nicht automatisch initialisiert.
4. Die Verwendung einer nicht initialisierten lokalen Variable führt zu einer Fehlermeldung des Compilers.

Beispiel

Die nachfolgende Klasse `InitTest` enthält eine Klassen- und eine Instanz-Variable sowie eine Objekt-Methode.

In der Methode `foo()` wird ein `int`-Array und eine nicht initialisierte lokale Variable `i` deklariert, die bei Verwendung zu einem Fehler beim Kompilieren führt.

25. Auch **automatische** Variable genannt
26. Dvor der Ausführung des Codes im Konstruktor

```
class InitTest {
  static double d;               // class field
  String s;                      // instance field
  void foo() {                   // Nonsens Name
    int i;                       // lokale Variable
    int[] iarr= new int[2];      // lokales int-Array
    System.out.println(iarr[0]); // 💻: 0
    System.out.println(d+" "+s); // 💻: 0.0 null
    System.out.println(i);       // C-Fehler: 4. Regel
  }
}
```

1.7 Namenskonventionen

Jeder Programmierer entwickelt über kurz oder lang seinen eigenen Stil beim Codieren. Solange er isoliert arbeitet: kein Problem. Kritisch wird es erst, wenn er im Team arbeitet, sein Code von anderen verstanden und später modifiziert werden muss.

Hier kann Software-Engineering viel von älteren Ingenieurwissenschaften lernen, die Normen und Konventionen zur **Conditio sine qua non** für Ingenieure machen.

Deshalb soll das Kapitel mit einigen einfachen, aber wichtigen Namenskonventionen beendet werden, die das Leben in einer Java-Gemeinschaft sicherlich angenehmer gestalten:

Konventionen bei der Wahl der Namen

Allgemein

- Für die Ausgabe ist Unicode eine »Offenbarung«. Dies bedeutet aber nicht, dass man nun Identifier in Deutsch, Spanisch oder Arabisch schreiben soll. Der ASCII-Zeichensatz ist immer die bessere Wahl, zumal für Klassen, die gleichzeitig auch Dateinamen sind.[27]
- Besteht ein Identifier aus mehr als einem Wort, so beginnt jedes »innere« Wort mit einem Großbuchstaben (Ausnahme: Package).

Package

Der gesamte Name wird in Kleinbuchstaben geschrieben und beginnt – sofern Eindeutigkeit erforderlich ist – mit dem umgekehrten Internet-Domain-Namen (siehe 1.2.1).

27. Wen das nicht überzeugt, der sollte einfach Excel-Tabellen mit (deutschem) VBA-Code für europäische Dependancen erstellen.

Klasse

Eine Klasse beginnt immer mit einem Großbuchstaben. Klassennamen enthalten in der Regel Substantive.[28]

```
MeineErsteKlasse   XMLUtility
```

Interface

Ein Interface unterliegt aufgrund seiner Ähnlichkeit zu einer Klasse der gleichen Konvention. Damit lässt sich ein Interface allerdings aufgrund des Namens nur selten von einer Klasse unterscheiden.[299]

Dies stört nicht unbedingt, sofern man z.B. mit UML-Klassendiagrammen (siehe Kapitel 4, Modellierung und UML) arbeitet.

Alternativ kann man nach MS-MFC[30]-Konvention jeden Namen mit dem Präfix `I` versehen.

```
IPerson   IProduct   IUnknown
```

Konstante

Eine Konstante (eine `static final` deklarierte Variable) vom primitiven Typ wird durchgängig groß geschrieben, wobei Wörter durch einen Unterstreichungsstrich getrennt werden:[31]

```
PI   DEM_EURO_EXCHANGE_RATE
```

Methode, Variable und Parameter

Eine Methode, Variable oder ein Parameter beginnt immer mit einem Kleinbuchstaben.

Eine Methode sollte in der Regel mit einem Verb beginnen.

Der Name eines Parameters oder einer lokalen Variablen kann durchaus nur aus einem Buchstaben bestehen, sofern die Bedeutung klar ist.

```
isEmpty()  setNewName(String s)  firstName
```

28. Zusätzlich sollte das Zeichen $ nicht im Klassennamen verwendet werden, da er zur Separierung von Namen innerer Klassen verwendet wird.
29. Außer wenn der Name Rollencharakter hat, wie z.B. `Cloneable`.
30. MFC: Microsoft Foundation Class (Library).
31. Bei konstanten Referenz-Variablen hängt die Konvention von der Semantik ab: Ist das Objekt, auf das die Referenz zeigt, auch konstant, d.h. immutable, so gilt auch hier die Großschreibregel.

1.7.1 Methodenname

Die Suche nach Methodennamen

Konventionen sind nur ein erster Schritt. Ein weiterer ist die Suche nach einem passenden Namen für eine Methode. [32]

Findet man keinen besseren Methodennamen als **setzeDaten()**, **liefereEierWolleMilchFleisch()** oder **fubar()**, sollte man sein Klassen-Design überdenken. Vielleicht gibt es ja doch bessere und für den Benutzer einsichtigere Methoden.

1.8 Zusammenfassung

Das Wort »Java« steht für eine Technologie, die auf drei Säulen ruht, Sprache, Plattform und JVM. Klassen und Interfaces werden in Packages organisiert, die über mehrere Übersetzungseinheiten verteilt sein können.

Klassen kapseln Daten und Code und können mit Hilfe der Methode **main()** ausführbar gemacht werden. Die JVM lädt Klassen erst bei Bedarf und reagiert auf Laufzeitfehler mit Exceptions.

Unicode, Schlüsselwörter sowie acht primitive Datentypen inklusive ihrer Literale bilden das Fundament der Sprache. Numerische Datentypen können implizit oder explizit ineinander konvertiert werden.

Obwohl es auch zu Strings Literale gibt, zählen sie nicht zu den primitiven Datentypen, sondern sind Instanzen bzw. Objekte der Klasse `String`.

Felder einer Klasse sowie Array-Elemente werden automatisch initialisiert, nicht initialisierte lokale Variablen werden vom Compiler erkannt.

Zu Java gibt es akzeptierte Namenskonventionen, die man durchaus verletzen darf, sofern man **Viren** programmiert.

32. Zu `foo()` bzw. `fubar()` siehe auch www.netmeg.net/jargon/terms/f/foo.html.

1.9 Testfragen

Zu jeder Frage können jeweils ein oder mehrere Antworten bzw. Aussagen richtig sein.

1. Welche Aussagen sind zu der ausführbaren Klasse `Startable` richtig?

A: `Startable` muss `public` deklariert werden.

B: Sie wird in eine Datei `startable.class` kompiliert.

C: `Startable` enthält eine passende Methode `main()`.

D: `Startable` kann in einer Compilation Unit mit Namen `Test.java` enthalten sein.

2. Welche `main()`-Methode kann als Startmethode für eine Applikation verwendet werden?

A:
```
public static void main(String arg[])   { ... }
```

B:
```
public static void main(String[   ] a)  { ... }
```

C:
```
public static void main(string[] args)  { ... }
```

D:
```
public static void Main(String[] args)  { ... }
```

3. Welche Kommentare sind korrekt gesetzt, sodass der Code fehlerfrei kompiliert wird?

A:
```
System.out.println(/**Kommentar**/"Hallo");
```

B:
```
System.out.println(//Kommentar "Hallo");
```

C:
```
System.out.println("Hal"/*//Kommentar*/+"lo");
```

D:
```
System.out.println(\*Kommentar*\);
```

4. Welche Identifier sind gültig?

A: `$byte`

B: `__1`

C: `name-1`

D: `1Name`

E: `operator`

F: `True`

5. Welche Aussagen sind zu folgendem Code-Fragment richtig?

```
char c= '1';
res= 2*c;
System.out.println(res);
```

A: Die Variable `res` kann vom Typ `char` sein.

B: Die Variable `res` muss vom Typ `int` sein.

C: Die Variable `res` kann vom Typ `double` sein.

D: Ist `res` vom Typ `int`, ist die Ausgabe: `2`

E: Die zweite Codezeile ist fehlerhaft, egal von welchem Typ `res` ist.

6. Welche Aussagen sind zur Klasse `InitTest` richtig?

```
class InitTest {
  static boolean ok;
  String s;
  void test() {
    int i;
    //System.out.println(i);
  }
}
```

A: `s` ist eine Instanz-Variable.

B: `ok` hat den Wert `false`.

C: Bei Entfernen der Kommentarsymbole // wird `InitTest` nicht kompiliert.

D: `s` hat den Wert `Null`.

7. Welche Aussagen sind richtig?

A: Der Wertebereich von `byte` ist `-27.. 27`.

B: Die Anweisung `char c= 65000;` wird ohne Fehler kompiliert.

C: Die Anweisung `char c= '/u0100';` wird ohne Fehler kompiliert.

D: Nach der Anweisung `int i= 020;` hat `i` den Wert 16.

E: `0x0101L` ist eine `long`-Literal und hat den Wert 257.

F: `"1"` ist ein Literal vom primitiven Typ.

8. Was sind gültige Literale?

A: `1,2`

B: `-.12e-1F`

C: `"\\a"`

D: `'\U003f'`

E: `True`

9. Welche Aussagen sind zu folgendem Code-Fragment richtig?

```
int i= (int) (1./0.);
System.out.println(i);
```

A: Der Compiler meldet in der ersten Zeile einen Fehler.

B: Bei der Ausführung wird eine Exception aufgrund der Division durch Null generiert.

C: Die Ausgabe ist: `Infinity`

D: Die Ausgabe ist: `2147483647`

10. Welche Aussagen sind zu folgendem Code-Fragment richtig?

```
String s= "1";
int i= (int) s;
System.out.println(i);
```

A: Der Compiler meldet in der zweiten Zeile einen Fehler.

B: Bei der Ausführung wird eine Exception aufgrund des unerlaubten Casts in der zweiten Zeile generiert.

C: Die Ausgabe ist: `1`

5 Vererbung

Zur UML-Spezialisierung des letzten Kapitels wird nun die zugehörige Java-Technologie vorgestellt.
Schwerpunkte bilden die Unterschiede von Overriding zu Overloading bzw. Shadowing, der Einfluss von Modifikatoren auf die Vererbung sowie Objekt-Anlage und -Zerstörung mittels Konstruktoren, Initialisierung und Finalization.
Vererbung, Modifikatoren und Objekt-Erzeugung sind mit wichtigen Regeln verbunden, die anhand von Beispielen, Diagrammen und Idiomen erklärt werden.

Spezialisierung wird in Java mittels **Vererbung** (**Inheritance**) realisiert.

Innerhalb der Klassenhierarchie werden generelle Klassen als Ober- oder **Superklassen** bezeichnet. Wird aus einer Klasse eine spezielle abgeleitet, wird diese direkte Unterklasse bzw. **Subklasse** genannt.

Subklasse vs. echte Subklasse

Da in Java genau genommen jede Klasse Subklasse von sich selbst ist, müssten alle anderen **echte** oder **strikte** Subklassen genannt werden. Im Weiteren wird aber der Einfachheit halber unter einer Super- bzw. Subklasse immer eine echte bzw. strikte verstanden.

5.1 Deklaration von Subklassen

Mit Hilfe des Schlüsselwortes extends realisiert Java Subklassen als **einfache Vererbung** (**single inheritance**), d.h., zu einer Subklasse gehört nur genau eine Superklasse.

Deklaration einer Subklasse

Dagegen ist die Implementierung von beliebig vielen Interfaces mittels des Schlüsselworts implements möglich[1]:

```
[cModifiers] class SubClassName [extends SuperClass]
                                [implements Interface1, ...]
{...}
```

Die Prinzipien der Generalisierung/Spezialisierung (siehe 4.1), insbesondere die Substitutions-Regel und die dynamische Polymorphie, werden durch die Vererbung in Java wie folgt realisiert.

Subklassen erben Instanz-Variable der Superklasse.

1. Vgl. zur Implementierung Kapitel 6, Interfaces und Pattern.

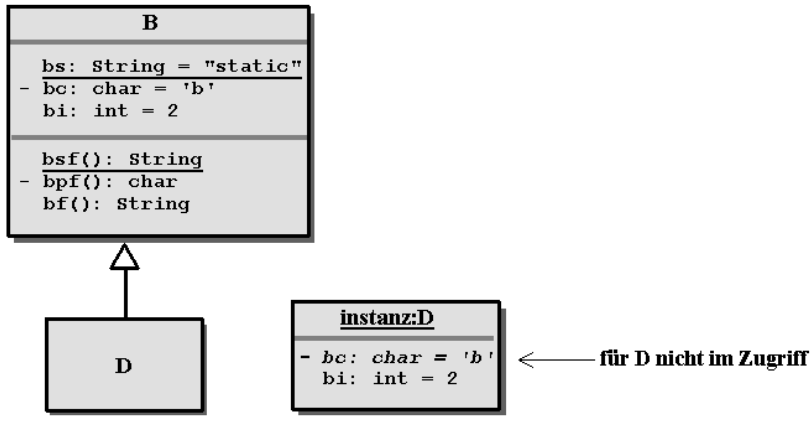


Abbildung 5.1 Die Subklasse D erbt Instanz-Variable von B

Es sei D eine direkte Subklasse von B (siehe Abb. 5.1). Dann gilt folgende Regel:

Zugriff auf Member der Superklasse

Vererbungs-Regel: Die Klasse D, und somit jede Instanz von D, »erbt« alle Instanz-Variablen von B. D kann direkt auf alle statischen oder nicht statischen Variablen und Methoden zugreifen, die in B nicht `private` erklärt sind.

Beispiel (zu Abb. 5.1)

```
class B {
  static String bs= "static";
  private char bc= 'b';   // wird vererbt, aber ohne Zugriff
  int bi=2

  static String bsf() { return bs+" bsf"; }
  private char bpf() { return Character.toUpperCase(bc); }
  String bf() { return String.valueOf(bc) + bpf()+ bi; }
}

class D extends B {
  void test() { // Zugriff auf (nicht) statische Member
    System.out.println(bs+","+bi+","+bsf()+","+bf());
    // System.out.println(bc + "," + bpf());   ← C-Fehler
  }
}
```

Der Aufruf von `test()` über eine Instanz von `D` ergibt dann:

```
new D().test(); // 💻: static,2,static bsf,bB2
```

Obwohl in D keine Member angelegt werden (die Klasse scheint also leer zu sein), erbt sie alle Instanz-Variablen von B und kann auf alle (nicht `private`) Member von B zugreifen, als wären sie in D selbst deklariert.

Die o.a. Regel gilt nicht nur für eine direkte Superklasse, sondern auch für eine Klassen-Hierarchie.

- Die Member-Vererbung ist transitiv.

Vererbung ist transitiv

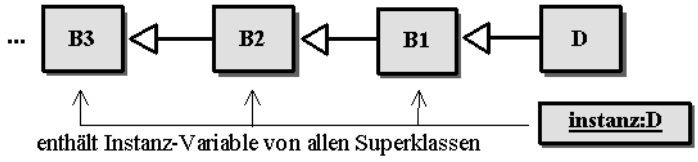


Abbildung 5.2 Vererbungs-Hierarchie

Ist D Subklasse von Superklassen B_i, so enthält D alle Instanz-Variablen der Superklassen B_i (siehe Abb. 5.2).

Die Klasse D kann auch auf alle (nicht `private`) Member der Superklassen B_i zugreifen, wobei dies allerdings uneingeschränkt nur für die direkte Superklasse B_1 gilt (siehe hierzu 5.8.3).

Die Substitutions-Regel aus dem vierten Kapitel führt zur

Widening-Conversion-Regel: Referenzen bzw. Instanzen von D können überall da eingesetzt werden, wo als Typ irgendeine Superklasse B erwartet wird.

Insbesondere kann

Referenzen von Superklassen zeigen auf alle Instanzen in der Klassen-Hierarchie

- eine Referenz vom Typ B auf eine Instanz von D verweisen:

```
B b= new D();
```

- ein Parameter vom Typ B durch ein Argument von D ersetzt werden:

```
method(new D());
// Methode ist deklariert als: void method(B b) {...}
```

Erklärung: Da eine Instanz von D alle Felder von B enthält und alle Methoden von B »versteht«, kann sie jederzeit anstelle einer Instanz von B stehen. Die Umkehrung gilt natürlich nicht, da die Superklasse B von zusätzlichen Feldern und Methoden in D nichts »weiß«.

Member-Vererbung und Widening-Conversion haben weit reichende Folgerungen, die im Weiteren detaillierter besprochen werden.

5.2 Overriding vs. Overloading

Die Neuimplementierung einer geerbten Methode nennt man **Overriding** (**Überschreiben**).

Overriding in Subklassen

- Overriding gibt es nur in Subklassen.
- Overriding verlangt nicht nur dieselbe Signatur (siehe 3.4.3), sondern auch denselben Rückgabetyp wie die Methode der Superklasse.

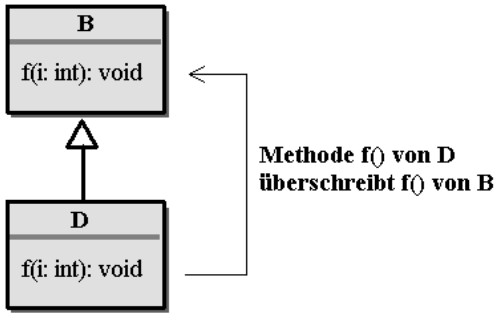


Abbildung 5.3 Overriding einer Methode f()

Overriding stellt immer die passende Methode sicher

Overriding-Regel: Überschreibt die Klasse D eine Instanz-Methode von B, so wird zu einer Instanz von D nur die Methode von D und nicht die von B aufgerufen.

Diese Overriding-Regel gilt uneingeschränkt! Auch über eine Referenz vom Typ B oder mittels Casts nach B ist es nicht möglich, zu einer Instanz von D eine Instanz-Methode von B aufzurufen.

Beispiel (zu Abb. 5.3)

```
class B {          void f(int i) {System.out.println(i);}  }
class D extends B {void f(int i) {System.out.println(i*i);}}

class OverridingTest {
  static void show (B b) { b.f(2); }

  static void test () {
    D d= new D();
    B b= d;       // Instanz von D über b referenzieren
    show (b);     // trotz Argument b  💻: 4
    ((B)d).f(2);  // trotz Cast nach B 💻: 4
  }
}
```

Der Aufruf von `OverridingTest.test();` erzeugt als Ergebnis immer 4.

Erklärung: Die Overriding-Regel ist sinnvoll. Eine Methode in einer Subklasse wird nur dann überschrieben, wenn das Verhalten der Methode der Superklasse nicht mehr gültig ist. Würde also die Overriding-Regel nicht gelten, wäre es mittels Casts oder Referenz möglich, zu Objekten ungültige Methoden aufzurufen.

Overriding ist ein sinnvoller Mechanismus

Overloading

Im Gegensatz zu Overriding bedeutet **Overloading** (Überladen) in einer Klasse die Wiederverwendung des Namens für Methoden mit verschiedener Signatur.

Overloading: Statische Polymorphie

Der Compiler kann bereits beim Übersetzen anhand der unterschiedlichen Signaturen die einzelnen Methoden mit gleichem Namen unterscheiden.

Overriding: Dynamische Polymorphie

- Overriding ist somit unbedingt von Overloading zu unterscheiden.

Die Signatur-Regel in Abschnitt 3.4.3 wird für Klassen-Hierarchien erweitert.

Erweiterte Signatur-Regel: In einer Klassen-Hierarchie können keine zwei Methoden mit derselben Signatur vorkommen, es sei denn, eine Methode in der Subklasse überschreibt eine andere in einer Superklasse.

Signatur in Hierarchien

Beispiel

```
class B {
  String s= "B";
  void f(int i) { System.out.println(i); }
  void g() {}
}
class D extends B {
  // int g() {}                          ← Compiler-Fehler     ①
  void f(byte b) { System.out.println(b); }                     ②
  void f(int i)  { System.out.println(s+" "+i); }
}
public class Test {
  public static void main(String[] args) {
    D d= new D();
    B b= d;                                                     ③
    b.f((byte)1);                                 💻: B 1       ④
    d.f((byte)1);                                 💻: 1
    d.f(1);                                       💻: B 1
  }
}
```

Erklärung:

Zu ①: Nach der Signatur-Regel hat die Methode `int g()` in `D` dieselbe Signatur wie `void g()` in `B`, aber überschreibt sie nicht. Dies erzeugt einen Compiler-Fehler.

Zu ②: Beide Methoden `f()` in `D` sind erlaubt. Die erste hat unterschiedliche Signatur, die zweite überschreibt `f()` in `B`.

Zu ③: Aufgrund von **Widening Conversion** kann eine Instanz von `D` überall da stehen, wo ein Typ `B` erwartet wird.

Zu ④: Eine Referenz `b` vom Typ `B` kennt nur eine Methode `f(int i)`, ruft aber nach der Overriding-Regel die Methode `f(int i)` von `D` auf.

Die Referenz `d` vom Typ `D` kennt zwei verschiedene Methoden mit Namen `f` und ruft deshalb die zum Argument passende auf.

5.3 Superklasse Object

Object, die ultimative Superklasse

Jede Klasse, die man deklariert, hat eine Superklasse, auch wenn keine Superklasse mittels `extends` angegeben wird.

- Alle Klassen haben als Superklasse die Klasse `Object`, die selbst keine Superklasse besitzt.

Damit erbt jede Klasse automatisch alle elf Instanz-Methoden der Klasse `Object`. Hierzu gehören auch die beiden schon besprochenen wichtigen Methoden:

```
public boolean equals(Object obj)
public String toString()
```

Um deren Verhalten an die jeweilige Klasse anzupassen, müssen diese Methoden überschrieben werden.

Verwechslung von Overloading und Overriding

```
class Complex {
  double re,im;
  //...
  public boolean equals(Complex c) {
    return re==c.re && im==c.im;
  }
  //...
}
```

Obwohl die Methode `equals()` gut aussieht, hat sie die Methode aus `Object` nicht überschrieben, sondern nur überladen. Dies kann – je nach Code – zu subtilen semantischen Fehlern führen.

5.4 Vererbung und Modifikatoren

Nachfolgend werden die Modifikatoren in Hinblick auf die Vererbung besprochen, d.h., welchen Einfluss sie auf die Vererbung haben bzw. welche Semantik damit verbunden ist.

5.4.1 abstract

In Java hat der Modifikator `abstract` dieselbe Bedeutung wie in der UML (siehe 4.2.6) und kann auf Klassen und Instanz-Methoden angewendet werden.

abstract: keine Instanzen möglich

Von einer Klasse, die abstrakt erklärt wird, können keine Instanzen angelegt werden. Wird dagegen eine Methode abstrakt erklärt, hat sie keinen Methoden-Rumpf, d.h. keine Implementierung.

abstract: notwendig bei Unvollständigkeit

Eine Klasse muss abstrakt erklärt werden, wenn sie unvollständig ist, genauer gesagt, wenn sie

- eine abstrakte Methode enthält oder erbt, die sie nicht implementiert.
- im Klassenkopf ein Interface enthält, deren Methoden sie aber nicht vollständig implementiert.

Ansonsten kann man eine Klasse abstrakt erklären, auch wenn sie keine abstrakten Methoden enthält.

Dies macht dann Sinn, wenn die Klasse zwar eine Superkasse ist, aber noch so unvollständig ist, dass eigene Instanzen ziemlich nutzlos sind.

5.4.2 final

final: keine Subklassen möglich

Zu einer `final` deklarierten Klasse[2] kann es keine Subklassen mehr geben. Eine `final` deklarierte Methode kann in einer Subklasse nicht mehr überschrieben werden.[3]

Dieser Modifikator wird dafür eingesetzt, sicherzustellen, dass aufgrund einer Vererbung nichts mehr geändert werden kann. Deshalb ist `final` für Klassen bzw. Methoden mit Vorsicht einzusetzen, da es auch Erweiterungen ausschließt. Andererseits dient es zur Geschwindigkeits-Optimierung (siehe 5.8.1).

Die Klassen `Math`, `System` und die Wrapper der primitiven Typen sind häufig benutzte `final` deklarierte Klassen im Package `java.lang`.

2. Interfaces können nicht `final` deklariert werden.
3. Die Bedeutung von `final` für Variablen wurde bereits in 3.4.2 besprochen.

5.4.3 Zugriffs-Modifikatoren (Access-Modifier)

Zugriffs-Modifikatoren: für Package-Ebene und Member

Die drei Modifikatoren `public`, `protected`, `private` sowie der »ohne Namen« regeln den Zugriff auf Klassen, Interfaces und Member.

Es ist zwischen dem Zugriff auf Klassen und Interfaces, die auf Package-Ebene deklariert sind, und dem Zugriff auf ihre Member – Felder und Methoden – zu unterscheiden.

Zugriff auf Klassen/Interfaces (auf Package-Ebene)

Für Klassen und Interfaces auf Package-Ebene ist entweder kein Modifikator oder nur `public` erlaubt.

Auf Package-Ebene: nur public oder default

Ohne Modifikator können Klassen und Interfaces nur in dem Package verwendet werden, in dem sie deklariert sind. Dies nennt man (default) Package-Zugriff oder auch friendly Zugriff.

Wird eine Klasse/Interface `public` deklariert, kann sie in jedem Package verwendet werden.

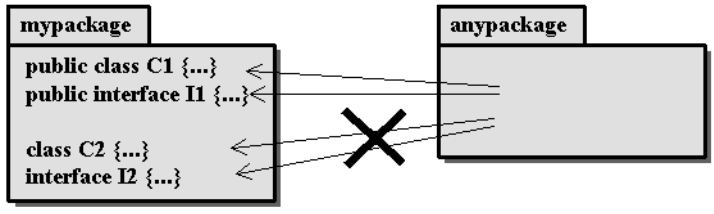


Abbildung 5.4 Inter-Package-Zugriff (UML-Darstellung)

Zugriff auf Interface-Member

Interface-Member: immer public Methoden: abstract Felder: static final

Interface-Methoden sind implizit `public abstract` und können auch nur explizit `public` und/oder `abstract` deklariert werden (was somit ziemlich unnötig ist).

Interface-Felder sind implizit `public static final` und können auch nur explizit `public` und/oder `static` und/oder `final` deklariert werden (was somit wiederum unnötig ist).

Damit bieten alle Member eines Interfaces überall da Zugriff, wo das Interface selbst Zugriff bietet (d.h. im Package oder auch außerhalb).[4]

4. Vgl. Kapitel 6, Interfaces und Pattern.

Zugriff auf Klassen-Member

Klassen-Member: drei Modifikatoren – vier Zugriffsarten

Der Zugriff auf Klassen-Member kann mit Hilfe der Zugriffs-Modifikatoren (**Access-Modifiers)** kontrolliert werden:

- `public`: Die Member einer Klasse bieten überall da Zugriff, wo auch die Klasse Zugriff bietet (im Package oder auch außerhalb).
- `protected`: Die Member einer Klasse bieten im Package Zugriff und auch innerhalb einer Subklasse außerhalb des Packages, sofern die Klasse public erklärt ist.
- kein Modifikator (default/friendly/Package-Zugriff): Die Member einer Klasse bieten für alle Klassen desselben Packages Zugriff.
- `private`: Die Member bieten nur innerhalb der eigenen Klasse Zugriff.

Damit werden also vier verschiedene Stufen der Zugriffsrestriktion auf Member definiert.

Überschreiben der Zugriffs-Modifikatoren

Die Modifikatoren können nach der Stärke ihrer Zugriffsrestriktion geordnet werden (siehe Abb. 5.5).

Ordnung der Zugriffsarten

Diese Ordnung ist dann wichtig, wenn man in einer Subklasse für eine überschriebene Methode die Zugriffsart abweichend von derjenigen in der Superklasse setzen will.

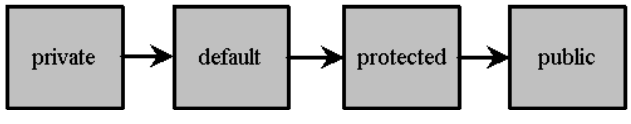


Abbildung 5.5 Erlaubte Richtung der Zugriffsänderung bei Overriding

Änderung der Zugriffsart in Subklassen

Bei Overriding gilt folgende Regel für Zugriffs-Modifikatoren.

Access-Widening: Wird eine Methode überschrieben, kann der Zugriffs-Modifikator der Superklasse in der Subklasse nur mit einem Zugriffs-Modifikator überschrieben werden, wenn dieser gleich oder weniger restriktiv ist (d.h. in Pfeilrichtung bei Abb. 5.5).

Diese Regel ist für **Widening-Conversion** notwendig, d.h. um zu verhindern, dass Methoden durch Überschreiben plötzlich keinen Zugriff mehr bieten.

Package-Grenzen-Problematik bei `protected`:

Werden Subklassen über Package-Grenzen hinweg deklariert, hat der Modifikator `protected` einen kleinen logischen »Twist«.

protected: Vorsicht bei unterschiedlichen Packages

> Der Modifikator `protected` lässt den Zugriff nur auf direkt geerbte Member desselben Objekts zu, nicht jedoch auf `protected` Member in anderen Objekten derselben Superklasse.

Beispiel

Im linken Kasten ist eine Klasse `Base` im Package `p1` erklärt. Im rechten Kasten ist eine Klasse `Derived` im Package `p2` von `Base` abgeleitet. Die Klasse `Derived` kann damit zwar auf das instanzeigene `protected int i` zugreifen, nicht jedoch auf ein `i` einer anderen Instanz.

```
package p1;

public class Base {
  protected int i= 2;
}
```

```
package p2;

class Derived extends p1.Base {
  void foo(p1.Base o) {
    System.out.println(i+ o.i);   ①
  }
}
```

Erklärung: Die Zeile ① erzeugt einen Compiler-Fehler, verursacht durch `o.i`. Der Zugriff auf das eigene geerbte `protected` Feld `i` ist erlaubt, nicht jedoch auf das Feld `i` eines anderen Objekts derselben Klasse.

5.4.4 Unverträglichkeiten von Modifikatoren

Klassen: abstract vs. final

Bei Klassen kann der Modifikator `final` (keine Subklasse erlaubt!) nicht zusammen mit `abstract` (Subklasse notwendig!) verwendet werden.

Member: abstract vs. final, static, private

Bei Methoden können die Modifikatoren `final`, `static` und `private` nicht zusammen mit `abstract` verwendet werden. Abstrakte Methoden verlangen ein Overriding, wogegen `final` und `private` dies gerade ausschließen.

Die Modifikatoren `abstract` und `static` schließen sich dadurch aus, dass statische Methoden nicht im Sinne von Instanz-Methoden überschrieben[5], sondern nur verdeckt werden können (siehe hierzu 5.8.1).

5. Allerdings verwendet man bei statischen Methoden meistens auch den Begriff überschreiben.

5.5 Konstruktoren

Konstruktoren: eine eigene Methoden-Familie

Bei der Anlage einer Instanz wird direkt nach Allokation des Speichers durch die JVM ein **Konstruktor** (**constructor**) aufgerufen, der die Initialisierung des neu erschaffenen Objekts übernimmt.

Konstruktoren bilden syntaktisch eine eigene Gruppe von Methoden und unterliegen damit auch anderen Regeln als die normalen Instanz-Methoden (siehe auch 3.4.3 und 5.5.3).

Es gilt folgende einfache Regel:

Klassen ohne Konstruktoren sind unmöglich

- Es gibt keine Klasse ohne Konstruktor. In jeder Klasse, auch in einer abstrakten, existiert zumindest ein Konstruktor.

5.5.1 Default-Konstruktor

Default-Konstruktor: vom System geschrieben

Der genannten Regel wird dadurch entsprochen, dass in einer Klasse, in der kein Konstruktor explizit deklariert ist, der Compiler automatisch einen so genannten Default-Konstruktor (**default constructor**) anlegt.

Ein Default-Konstruktor hat keine Parameter und hat denselben Zugriffs-Modifikator wie seine Klasse.

Somit können ohne Anlage eines Konstruktors von jeder Klasse `C` Instanzen mit `new C()` angelegt werden.

5.5.2 Deklaration und Initialisierungs-Regeln

Deklaration eines Konstruktors

Legt man explizit Konstruktoren an, müssen sie immer den Namen der Klasse tragen und dürfen keinen Typ als Rückgabe definieren. Damit hat ein Konstruktor die folgende Deklaration:

```
[accessModifier] ClassName (parameterList) [throwsList] {...}
```

Beispiele

No-Arg-Konstruktor: ein Konstruktor ohne Argumente

```
class Point {
  int x, y;
  Point() {}               // analog zum Default-Konstruktor
  Point(int x, int y) { this.x= x; this.y= y; }
}
abstract class Figure {     // abstrakt!
  Point base;
  Figure (int x,int y) {
    base= new Point(x,y);  // legt einen Basispunkt (x,y) an
  }
}
```

Zusammenstellung aller wichtigen Regeln zum Konstruktor

Folgende Konstruktoren-Regeln sind zu beachten:

1. Wird in einer Klasse (explizit) ein Konstruktor mit Parameter deklariert, legt der Compiler keinen Default-Konstruktor mehr an.
2. Ruft ein Konstruktor einen anderen derselben Klasse auf, so erfolgt dies mit der Anweisung

   `this(argumentList);`

   wobei `argumentList` zur `parameterList` des aufgerufenen Konstruktors passen muss.
3. Ruft ein Konstruktor einen anderen der direkt übergeordneten Superklasse auf, so erfolgt dies mit der Anweisung

   `super(argumentList);`

   wobei `argumentList` zur `parameterList` des aufgerufenen Kon-struktors passen muss.
4. Sofern verwendet, muss `this();` oder `super();` die erste Anweisung in einem Konstruktor sein, d.h., sie können nicht beide gleichzeitig verwendet werden.
5. Verwendet man weder `this()` noch `super()`, fügt der Compiler als erste Anweisung `super();` ein, d.h., es wird zuerst der No-Arg-Konstruktor der direkt übergeordneten Superklasse aufgerufen.
6. Für die Felder, die nicht im Konstruktor initialisiert werden, wird vom Compiler automatisch Code generiert. Dieser Code wird nach der Ausführung des Superklassen-Konstruktors und vor dem Code des eigenen Konstruktors ausgeführt. Dabei wird die Reihenfolge der Variablen beachtet.
7. Konstruktoren werden nicht vererbt, jede Subklasse muss ihre eigenen Konstruktoren anlegen.
8. Blank final Instanz-Variablen müssen im Konstruktor oder Instanz-Initialisierer (siehe 5.7.1) einen Wert zugewiesen bekommen.

Reihenfolge der Konstruktoren bei Klassen-Hierarchie

Eine Instanz enthält alle internen Instanzen der Superklassen, die bei einer mehrstufigen Hierarchie wie Schalen einer Zwiebel ineinander liegen.

Vom Compiler wird sichergestellt, dass zuerst immer die Felder der Superklasse initialisiert werden, und zwar (von innen nach außen) in der Reihenfolge der Klassen-Hierarchie. Dies gewährleistet die fünfte Regel (siehe Abb. 5.6).

Beispiel

Abfolge der Konstruktoren-Aufrufe zu den Klassen in Abb. 5.6.

Zwiebelschalen-Modell mit Konstruktoren-Reihenfolge

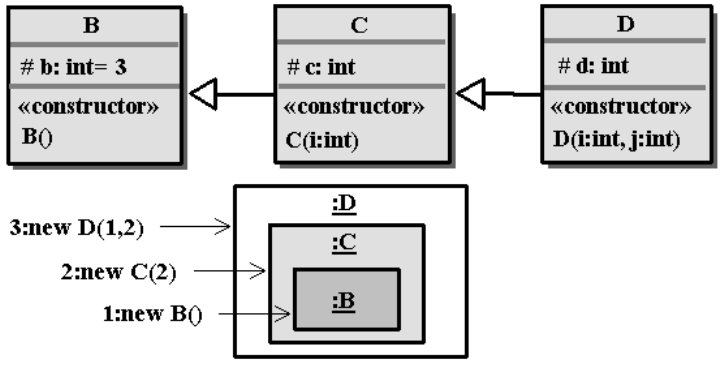


Abbildung 5.6 Reihenfolge der Konstruktoren-Aufrufe

```
class B {
  protected int b= 3;
  B() { System.out.print("B b:"+b); }  // siehe 6. Regel        ①
}
class C extends B {
  protected int c;
  C(int i) {System.out.print("C "); c= b+i; } // siehe 5. Regel ②
}
class D extends C{
  protected int d;
  D(int i,int j) { super(j);          // siehe 3. & 4. Regel
    System.out.print("D "); d= c+i;
    System.out.println("b:"+b+" c:"+c+" d:"+d);
  }
}
public class Test {
  public static void main(String[] args) {
    new D(1,2);                      // 💻: B b:3 C D c:5 d:6
  }
}
```

Erklärung: In ① ist b bereits mit 3 initialisiert, in ② steht aufgrund der fünften Regel ein implizites `super();` als erste Anweisung. Jede Subklasse kann im Konstruktor bereits auf die initialisierten Werte ihrer Superklasse zugreifen.

5.5.3 Signatur-Regel für Konstruktoren

Unterscheidung der Konstruktoren anhand der Signatur

Das Overloading der Konstruktoren unterliegt der einfachen **Signatur-Regel**, d.h., Konstruktoren müssen untereinander anhand der Sequenz ihrer Parametertypen zu unterscheiden sein (siehe auch 5.6.2, Fall 2).

5.5.4 Zusammenarbeit von Konstruktoren

super() erlaubt Zusammenarbeit der Konstruktoren in der Hierarchie

Konstruktoren können über Klassengrenzen hinweg zusammenarbeiten.

Beispiel

```
class Point {
   // wie in 5.5.2
}

class Figure {
  Point base;
  // Figure() {}                                                    ①
  Figure (int x,int y) {
    // Figure enthält einen Punkt, also nicht
    // super(x,y) sondern new Point()
    base= new Point(x,y);
  }
}
```

Vorsicht bei Default-Konstruktoren

```
class Circle extends Figure {
   int radius;            // Compiler-Fehler: 2. & 5. Regel
   // Circle(int x,int y,int r) { super(x,y); radius=r; }   ②
}
```

Erklärung: Nach der zweiten Regel gibt es in `Figure` keinen Default-Konstruktor. In `Circle` gibt es nur einen Default-Konstruktor, der nach der fünften Regel als erste Anweisung implizit ein `super()` enthält. Dies führt zur Compiler-Meldung »constructor Figure() not found«.

Der Fehler kann durch Entfernen der Kommentar-Symbole in Zeile ① und/oder ② behoben werden:

Fügt man nur den Konstruktor in ① ein (und nicht ②), stellt man fest, dass er nicht sehr gut passt:

```
System.out.println(new Circle().base.x);  // Laufzeitfehler!
```

`Figure()` setzt nämlich keinen Basispunkt wie `Figure(int x,int y)`, sondern überlässt die Initialisierung von `base` dem System:

```
System.out.println(new Circle().base);    // 💻: null
```

5.6 Pros und Kons der Initialisierung

5.6.1 Design-Pros

Die Initialisierung bei der Deklaration der Variablen[6] steht im Gegensatz zur Initialisierung erst im Konstruktor. Die Frage ist dann, was zu bevorzugen ist. Die Antwort lautet »Klarheit«:

Direkte Initialisierung vs. Konstruktoren

> Initialisierungs-Prinzip: Sofern nicht im Konstruktor ohnehin eine (Re-) Initialisierung stattfindet, sollten die Instanz-Felder direkt bei der Deklaration initialisiert werden.

Andererseits – werden die Felder in einem Konstruktor wieder auf andere Werte, z.B. die der Argumente gesetzt – verkehrt sich die Klarheit ins Gegenteil:

```
class BadIdea {
  String s="";                        // sinnlos!
  BadIdea(String s) { this.s= s; }
}
```

Innerhalb einer Klasse arbeiten Konstruktoren auf der Basis von **Constructor-Chaining** zusammen:

Constructor-Chaining: Vermeidung von Code-Redundanz

> **Constructor-Chaining**: Um Code-Replikationen zu vermeiden, sollten einfachere Konstruktoren komplexere aufrufen, wobei die einfacheren vor den komplexeren Konstruktoren deklariert werden.

Beispiel

```
class IntMatrix {
  int[][] m;
  IntMatrix(int dim)            { this(dim,dim); }
  IntMatrix(int rows, int cols) { this(rows,cols,0); }
  IntMatrix(int rows, int cols, int val) {
    m= new int[rows][cols];
    if (val!= 0) setVal(val);     // Aufruf einer Methode
  }
  void setVal(int val) {
    for (int i= 0; i<m.length; i++)
      for (int j=0; j<m[i].length; j++) m[i][j]= val;
  }
}
```

6. Dies ist für C++-Programmierer recht ungewöhnlich, da nicht erlaubt.

Die Klasse IntMatrix deklariert drei Konstruktoren, geordnet nach Detaillierung. Es können quadratische und nicht quadratische Matrizen mit bzw. ohne Wert angelegt werden:

```
IntMatrix imat1= new IntMatrix();          // C-Fehler
IntMatrix imat2= new IntMatrix(3);
IntMatrix imat3= new IntMatrix(3,4);
IntMatrix imat4= new IntMatrix(3,4,-1);
```

Der letzte Konstruktor demonstriert, dass in Konstruktoren durchaus andere Instanz-Methoden aufgerufen werden können.

Gemeinsame Initialisierungs-Methoden

Der Aufruf einer Instanz-Methode eignet sich auch vorzüglich, um das Problem zu lösen, dass man nur super() oder this() verwenden darf, aber nicht beides, obwohl dies manchmal sinnvoll wäre.

Das Codierungs-Prinzip ist einfach:

Umgehung der Beschränkung für super() und this()

- Man verwendet super() und ruft anschließend eine **Initialisierungs-Methode auf, in die man** den in den Konstruktoren gemeinsam genutzten Code ausgelagert hat.

Denn eine **Initialisierungs-Methode** muss man nicht unbedingt als erste Anweisung ausführen.

5.6.2 Design-Kons

Das Folgende ist zwar zulässig, aber zu vermeiden, da seine Wirkung nicht klar ist.

Vorsicht: abstrakte Methoden im Konstruktor

Ein Konstruktor ruft eine abstrakte Methode auf, die Felder der Subklasse ändert.

Beispiel

```
abstract class Money {
  Money() { set(1.0); }            // Wirkung unklar!
  abstract void set(double money);
}
class Euro extends Money {
  private double euro /*= 0.0*/ ;        // siehe Erklärung
  void set(double money) { euro= money; }
  double get()           { return euro; }
}
```

Der Fall ist dubios, da primitive Typen wie euro eigentlich automatisch auf 0.0 initialisiert werden. Doch in diesem Fall sind die beiden Deklarationen unterschiedlich:

```
private double euro;          ①
private double euro= 0.0;     ②
```

Die Ausgabe ergibt:

```
System.out.println(new Euro().get());  // ① 💻: 1.0
                                       // ② 💻: 0.0
```

Der Money-Konstruktor wird immer vor dem Euro-Konstruktor aufgerufen. Der Default-Konstrukor von Euro – explizit geschrieben – sieht wie folgt aus:

```
Euro() { super(); }
```

Bei ① wird der Wert also auf 1.0 gesetzt.

Bei ② fügt, gemäß der sechsten Regel in Abschnitt 5.5.2, der Compiler nach super(); allerdings noch die Anweisung euro= 0.0; ein.

Konstruktoren vs. normale Methoden

Falle: void-Konstruktoren

Der Compiler unterscheidet zwischen Konstruktoren und normalen Instanz-Methoden.

- Es können Instanz-Methoden deklariert werden, die den Namen der Klasse haben und sogar dieselbe Signatur wie ein Konstruktor.

Das ist zulässig[7], aber unsinnig, und geschieht meistens dadurch, dass man aus Versehen void vor einen Konstruktor geschrieben hat.

Beispiel

```
class Nonsense {
  Nonsense()  { System.out.println("Konstruktor-Nonsense"); }
  void Nonsense() { System.out.println("Methode-Nonsense"); }
}
```

Instanz-Methode und Konstruktor haben dieselbe Signatur und können nun wie folgt aufgerufen werden:

```
Nonsense u= new Nonsense();    // 💻: Konstruktor-Nonsense
u.Nonsense();                  // 💻: Methode-Nonsense
```

7. Dank der Sprach-Designer von Java

5.7 Initialisierer

Konstruktoren vs. anonyme Klassen

Da Konstruktoren immer den Namen der Klasse tragen müssen, entstand nach Einführung der **anonymen** Klassen[8] in Java 1.1 das Problem der Namensfindung für einen expliziten Default-Konstruktor.

Des Weiteren sind Konstruktoren ungeeignet, um statische Variablen zu initialisieren (siehe 5.7.2).

5.7.1 Instanz-Initialisierer

Die Lösung des ersten Problems war ein Konstruktor ohne Namen für eine anonyme Klasse.

Alternative: Instanz-Initialisierer

Ein **Instanz-Initialisierer (instance initializer)** besteht nur aus geschweiften Klammern, dem Initialisierungsblock:

```
class C {
  {
    // Initialisierungs-Block
  }
}
```

Instanz-Initialisierer: Rolle eines Default-Konstruktors

Ein Instanz-Initialisierer hat keine Parameter, entspricht also dem Default-Konstruktor.

In einer Klasse können sogar mehrere Instanz-Initialisierer vorkommen. Sinnvoll und notwendig sind sie aber nur für anonyme Klassen und auch hier sollte man sich auf einen beschränken.

Es gelten folgende Regeln:

Abfolge bei: Direkt-Initialisierung Instanz-Initialisierer Konstruktoren

1. Bei der Anlage einer Instanz wird der Code im Instanz-Initialisierer vor dem Code eines Konstruktors ausgeführt.
2. Die Variablen, die im Instanz-Initialisierer verwendet werden, müssen vor diesem deklariert sein.
3. Die Initialisierung von Variablen direkt bei der Deklaration und in Instanz-Initialisierern erfolgt in der Reihenfolge der Angaben.

Beispiel

Um die Regeln abzudecken, ist die folgende Klasse `IntVector` konstruiert und verstößt gegen gutes Design.

8. Anonyme Klasse: Innere Klasse ohne Namen, siehe hierzu Kapitel 8, Innere Klassen

```
class IntVector {
  int[] v;
  {
    System.out.println("Initialisierer");
    //standardVal= 1;                          ①
    v= new int[standardDim];  // ok: static
  }
  IntVector () {
    System.out.println("default");
    java.util.Arrays.fill(v,standardVal);
  }
  static int standardDim= 5;  // siehe 5.7.2
  int standardVal= 1;
}
```

Erklärung: Nach der zweiten Regel kann `standardVal` im Initialisierer nicht verwendet werden. Ein Entfernen des Kommentars in ① führt zu einem Compiler-Fehler.

Wird eine Instanz von `IntVector` angelegt, z.B. durch

```
new IntVector();  // 💻: Initialisierer default
```

wird zuerst der Code des Initialisierers ausgeführt, dann `standardVal` auf 1 gesetzt und erst dann der Code im No-Arg-Konstruktor ausgeführt.

5.7.2 Statischer Initialisierer

Statischer Initialisierer: Ausführung beim Laden der Klasse

Statische Variablen existieren unabhängig von Instanzen und werden – bevor irgendwelche Instanzen existieren – bereits beim Laden der Klasse angelegt und initialisiert.[9] Deshalb ist auch die Initialisierung von statischen Feldern in Konstruktoren unsinnig.

Sofern möglich, sollte die Initialisierung der statischen Felder sofort bei der Deklaration erfolgen. Bei einer komplexen Initialisierung ist dies nicht möglich und wird deshalb in einem **statischen Initialisierer** (**static initializer**) durchgeführt, der aus einem Block mit vorangestelltem `static` besteht:

9. Deshalb kann bei der Klasse `IntVector` der Instanz-Initialisierer bzw. ein Konstruktor bereits auf statische Variablen zurückgreifen.

```
class {
   static { // statischer Initialisierungs-Block }
}
```

Ein statischer Initialisierer hat keine Parameter und wird nur einmal ausgeführt, direkt nachdem die Klasse geladen wurde. Eine Klasse kann mehrere statische Initialisierer haben, einer reicht allerdings aus.

Nur statische Felder Abfolge der statischen Initialisierung

Es gelten folgende Regeln:

1. Ein statischer Initialisierer hat – wie alle statischen Methoden – keinen Zugriff auf Instanz-Variablen.
2. Analog zur zweiten Regel in 5.7.1.
3. Analog zur dritten Regel in 5.7.1.
4. Blank final Klassen-Variablen müssen im statischen Initialisierer einen Wert zugewiesen bekommen.

Beispiel

Beim Laden der Klasse SinusTable berechnete Tabelle von Sinuswerten:

```
final class SinusTable {
  final static int MAXDEGREE= 90;
  final static double sin[];                // blank final
  static {
    sin= new double[MAXDEGREE+1];
    for (int i=1; i<=MAXDEGREE; i++)
      sin[i]= Math.sin(i*Math.PI/180.);
  }
  private SinusTable(){}; // keine Instanzen, keine Subklasse
}
```

Erklärung: Die Klasse SinusTable bedient sich genau eines private-Konstruktors.

Damit sind weder Instanzen erlaubt noch kann man eine Subklasse von SinusTable anlegen, da deren Konstruktor den SinusTable-Konstruktor aufrufen muss (was aber nicht geht!).[10]

Die Tabelleneinträge können dann folgendermaßen abgefragt werden:

```
System.out.println(SinusTable.sin[0]);  // 💻: 0.0
System.out.println(SinusTable.sin[30]); // 💻: 0.49999999999999994
```

10. final dient somit nur der Klarheit und ist unnötig.

5.8 Overriding vs. Shadowing

Bei Vererbung verhalten sich Instanz-Methoden im Vergleich zu Feldern und statischen Methoden unterschiedlich.

5.8.1 Statischer vs. virtueller Aufruf von Methoden

Wie bereits besprochen, wird zur Laufzeit von der JVM – unabhängig vom Typ der Referenz – die zum Objekt gehörige Instanz-Methode ermittelt und ausgeführt. Diese Art nennt man

Virtueller Methodenaufruf: JVM ermittelt korrekte Methode

- **virtuellen** Methodenaufruf oder
- **dynamischer** Methodenaufruf oder
- **late** bzw. **dynamic binding**.

Synonyme: dynamisch, late oder dynamic binding

Diese Bezeichnungen stehen dafür, dass der Compiler beim Übersetzen einer Instanz-Methode in der Regel nicht weiß, welches Objekt von einer Variablen nun referenziert wird.

Dies steht im Gegensatz zum **statischen Aufruf** von Methoden bzw. **early binding**, wo bereits bei der Kompilierung feststeht, welche Methode ausgeführt werden soll, die dann auch bereits im Code eingebunden werden kann.

Statischer Methodenaufruf: Compiler ermittelt die korrekte Methode

Statisch aufgerufene Methoden

Der dynamische Methodenaufruf zur Laufzeit kostet Zeit und wird nur dann benutzt, wenn auch ein Overriding vorkommen kann. In einigen Fällen ist Overriding jedoch verboten oder ausgeschlossen:

- Alle `static`, `private` oder `final` deklarierten Methoden bzw. alle Methoden einer `final` deklarierten Klasse benötigen keinen dynamischen Methodenaufruf.

static/final/private-Methoden erlauben statische Aufrufe

Für diese Methoden steht nämlich schon zur Übersetzungszeit fest, dass sie nicht durch Methoden von Subklassen überschrieben werden können.

Interessant sind statische Methoden:

- Statische Methoden einer Superklasse werden in einer Subklasse durch eine identische statische Methode nicht überschrieben, sondern nur **verdeckt** (**shadowed**).[11]

11. Leider spricht man auch bei statischen Methoden häufig von Overriding, obwohl dies nicht korrekt ist. Zu Shadowing siehe auch 5.8.2.

Über den Klassennamen oder per Cast kann man – im Gegensatz zu Instanz-Methoden – alle statischen Methoden aller Klassen der Hierarchie jederzeit aufrufen.

Deshalb sollte man ein einfaches Idiom beachten:

Statische Methode: nur über Klassennamen aufrufen

- Statische Methoden sollte man nicht wie Instanz-Methoden über eine Referenz, sondern immer über den Klassennamen aufrufen.

Verstößt man gegen dieses Idiom, wird – im Gegensatz zu Instanz-Methoden – anhand der Referenz und nicht des Objekts die zugehörige Klasse ermittelt und deren statische Methode aufgerufen.

Dies führt zu schwer durchschaubarem Code, da statische Methoden wie Instanz-Methoden aussehen.

Statische Methoden: Aufruf über Instanzen werden mit Unklarheit bestraft

Beispiel

Der nachfolgende Code zeigt die sehr verwirrende Art, `static` deklarierte Methoden wie Instanz-Methoden aufzurufen:

```
class BC {
  static void sf() {System.out.println("BC.sf()");   }
  void dynf()      {System.out.println("BC.dynf()"); }
}
class DC extends BC {
  static void sf(){System.out.println("DC.sf()");  } // shadowing
  void dynf()    {System.out.println("DC.dynf()");} // overriding
}
public class Test {
  static void test(BC bc) {
    bc.sf();         // besser: BC.sf();       💻: BC.sf()
    ((DC)bc).sf();   // besser: DC.sf();       💻: DC.sf()
    bc.dynf();       // dynamic Lookup!        💻: DC.dynf()
  }
  public static void main(String[] args) {
    DC dc= new DC();
    BC bc= dc;
    dc.sf();         // besser: DC.sf();       💻: DC.sf()
    bc.sf();         // besser: BC.sf();       💻: BC.sf()
    bc.dynf();       // dynamic Lookup!        💻: DC.dynf()
    test(bc);        // siehe oben
  }
}
```

5.8.2 Shadowing von Feldern und super

Wie statische Methoden werden auch Felder von Superklassen in Subklassen nicht überschrieben, sondern nur verdeckt.

Verdeckte Felder in Superklassen: Einsatz von Cast oder super

Sollte man in einer Subklasse einem Feld denselben Namen geben wie einem Feld in der Superklasse, erhält man zwei Felder mit gleichem Namen, auf die man mittels Casts zugreifen kann, sofern dies der Access-Modifier zulässt.

- Innerhalb einer Subklasse kann mit Hilfe von `super.feld` das verdeckte Feld der Superklasse aufgerufen werden, die in der Hierarchie am nächsten liegt.

Beispiel

Einmal wird innerhalb der Subklasse `D`, einmal von außen über `Test` auf verdeckte Felder der Superklassen `B1` und `B2` zugegriffen.

```
class B1 {
  String s= "B1";
  int i=1;
}
class B2 extends B1 {
  String s= "B2";
}
class D extends B2 {
  String s= "D";
  int i=2;
  void f()      {
    System.out.println(super.s);           // 💻: B2
    System.out.println(super.i);           // 💻: 1
    System.out.println(((B2)this).s);      // 💻: B2
    System.out.println(((B1)this).s);      // 💻: B1
  //System.out.println(super.super.s);     Fehler: nicht so!
  }
}
public class Test {
  public static void main(String[] args) {
    D d= new D();
    System.out.println(((B2)d).s+" "+((B1)d).s);  // 💻: B2 B1
    d.f();
  }
}
```

(Superklasse) this hilft

5.8.3 Aufruf überschriebener Methoden mittels super

super: einzige Zugriffsmöglichkeit auf überschriebene Methoden

Innerhalb einer Subklasse kann man nicht nur auf Felder, sondern auch auf überschriebene Methoden der Superklasse zugreifen. Allerdings darf man dazu nur das Schlüsselwort `super` und keinen Cast verwenden.

super vs. super()

Außer dem Namen hat super mit dem Aufruf des Superklassen-Konstruktors `super()` nichts gemeinsam (siehe 5.5.2). Innerhalb einer Instanz-Methode ruft `super.method()` eine überschriebene Methode auf und kann auch überall stehen.

Auch für Methoden gilt – wie bei den Feldern – folgende Einschränkung:

super liefert nur die nächste überschriebene Methode

- Innerhalb einer Subklasse kann mit Hilfe von `super.method()` die überschriebene Methode der Superklasse aufgerufen werden, die in der Hierarchie am nächsten liegt.

Somit ist es mit `super.method()` nicht möglich, eine Methode einer Basisklasse auszuwählen, die in der Klassenhierarchie mehrfach überschriebene Methode wurde.

Die Suche einer Methode `method()` mittels `super.method()` beginnt immer bei der direkten Superklasse und hört bei der ersten passenden Methode einer Superklasse auf.

```
class B1 {
  void f() {System.out.println("B1.f()"); };
  void g() {System.out.println("B1.g()"); };
}
class B2 extends B1 {
  void f() {
    System.out.println("B2.f()");
    super.f();          // muss nicht erste Anweisung sein
  }
}
class D extends B2 {
  void g() {
    super.f();                    // 💻: B2.f()
                                  // 💻: B1.f()
    super.g();                    // 💻: B1.g()
  }
}
```

Die Methode `B1.f()` kann von `D` nicht aufgerufen werden.

5.9 Speicherverwaltung

Das Thema »Speicherverwaltung« hätte aufgrund der recht häufigen Fehlbeurteilung eigentlich ein eigenes Kapitel verdient.

Destruktoren gibt es NICHT!

In Java verläuft zwar die Objekt-Anlage mittels new und Konstruktoren analog zu C++, aber in Java gibt es kein Äquivalent zu Destruktoren bzw. `delete()`, so wie man es in C++ kennen und schätzen gelernt hat.[12]

Garbage-Collection: die Befreiung von der Speicherbereinigung via Destruktor

Der Grund für eine **automatische Speicherplatzfreigabe** – und hierfür steht der Begriff **Garbage Collection** in Java – ist es, den (Anwendungs-)Programmierer von der sehr fehlerträchtigen Aufgabe der manuellen Speicherfreigabe zu entlasten.

Dadurch wird natürlich auch so manches Idiom überflüssig, was man in C++ zur Speicherbereinigung gelernt hat.

Nun lassen sich C++-Konvertiten nicht so leicht überzeugen, dass dieses vormals mühselige und schwierige Geschäft nun reibungslos automatisch geschieht.

Ist finalize() ein Destruktor?

Deshalb wird als Anwort auf die Frage, wie denn z.B. vor der automatischen Objekt-Entsorgung die Ressourcen-Freigabe vorzunehmen ist, die Methode `finalize()` als Destruktor in »Java für C++-Umsteiger« verkauft.[13]

Wider den Mythos Destruktor

Das Hauptanliegen dieses Abschnitts ist es, den **Mythos** der Methode `finalize()` als Destruktor bzw. De-Initialisierer zu zerstören und gangbare Alternativen aufzuzeigen.

5.9.1 Garbage Collection

Garbage Collection (**GC**) ist ein fester Bestandteil der jeweiligen JVM und kann vom Programmierer nicht direkt ausgetauscht bzw. beeinflusst werden.

Erreichbarkeit: Objekte werden noch gebraucht

GC ist nur vom Konzept her festgelegt und basiert auf dem Prinzip der **Erreichbarkeit**:

- Ein Objekt heißt erreichbar (**reachable**), wenn es noch irgendeine (gültige) Referenz im Programm gibt, die auf dieses Objekt verweist.

12. Ein Destruktor ist in C++ eine Instanz-Methode, die bei der Entsorgung nicht mehr benötigter Objekte garantiert aufgerufen wird. Im Gegensatz zu new gibt die Methode `delete()` den Speicher wieder frei.
13. Eventuell in Verbindung mit einem Aufruf von `System.gc()`.

Unerreichbarkeit: Kriterium für Entsorgung

Kann ein Objekt im Programm nicht mehr über eine lokale Variable, ein Feld oder Array-Element angesprochen werden, ist es unerreichbar und kann aus dem Speicher entfernt werden.

Ablauf der automatischen Speicherbereinigung

In der JVM läuft der GC als ein nebenläufiger Prozess (Thread) mit niedriger Priorität, der die folgenden Aufgaben periodisch nacheinander erledigt:

1. Der GC untersucht alle Objekte im Speicher darauf, ob sie noch erreichbar sind.
2. Bevor ein nicht erreichbares Objekt von dem GC aus dem Speicher entfernt wird, ruft der GC die von `Object` geerbte Methode `finalize()` genau einmal auf.
3. Der Speicher, den das Objekt besetzt, wird freigegeben, sofern es noch immer unerreichbar ist.[14]

Speicherbereinigung ist Aufgabe der JVM und damit abhängig von ihr

Die spezifischen Details, vor allem der letzten beiden Punkte, hängen von der jeweiligen JVM ab.

- Das Verhalten des GCs unter einer Maschine ist nicht übertragbar auf die einer anderen oder auf eine zukünftige JVM.

In der Klasse `Object` ist `finalize()` wie folgt deklariert:

```
protected void finalize() throws Throwable {}
```

Zu dieser quasi abstrakten Methode gibt es einen langen SUN-Kommentar, der neben der Beschreibung der Ausführung noch Hinweise auf die Verwendung gibt.

Genau diese Hinweise mögen auch ein Grund sein, anzunehmen, `finalize()` wäre ein Destruktor im Sinn von C++.

5.9.2 GC und Finalization

Nutzen der Speicherbereinigung mittels finalize()

Will man in einer eigenen Klasse den GC nutzen,

- muss man in der Klasse `finalize()` überschreiben.
- müssen Instanzen der Klasse unerreichbar werden.

Beispiel

Nachfolgend wird in der Klasse `TestGC` anhand von `finalize()` getestet, wann der GC zumindest die ersten der beiden in 5.9.1 angegebenen Schritte ausführt.

14. Denn durch `finalize()` könnte die Unerreichbarkeit ja aufgehoben werden.

Ein einfacher Test der Speicherbereinigung mit Hilfe von finalize()

```
class TestGC {
  static int alive; // Anzahl der Instanzen, für die GC
                    // finalize() noch nicht aufgerufen hat
  long[] larr;
  TestGC(int arrSize) {
    larr= new long[arrSize];  alive++;
  }
  protected void finalize() {       // 2.Schritt:
    System.out.print(--alive+" ");  // alive anpassen
  }
}
```

Die nachfolgende Klasse legt elf Instanzen der Klasse Test an, wobei mit ARR_SIZE der Speicherverbrauch pro Instanz variiert werden kann.

```
public class FinalizeTest {
  final static int MAX_UNREACHABLE= 10;
  final static int ARR_SIZE= 10000;
  public static void main(String[] args) {
    System.out.print("Nicht entsorgt: ");
    for (int i=0; i<=MAX_UNREACHABLE; i++) {
      TestGC t= new TestGC(ARR_SIZE);
    }
    System.out.println("Ende: " +TestGC.alive);
  }
}
```

Erklärung: Da nur eine Referenz t zur Verfügung steht, wird nach jedem Iterationsschritt die vorherige Instanz unerreichbar, also zu einem GC-Kandidaten. Nach Beendigung der for-Schleife wird dies auch die letzte Instanz, da t eine lokale Variable ist. Alle Instanzen müssten also vor dem Programmende vom GC entsorgt werden.

Das Testergebnis auf der Konsole ist ernüchternd:

finalize(): ein nicht ermutigendes Ergebnis

```
Für ARR_SIZE<=1000  💻: Aktiv: Ende: 11
Für ARR_SIZE=10000  💻: Aktiv: 6 7 6 5 5 4 3 Ende: 4
Für ARR_SIZE=100000 💻: Aktiv: 1 1 1 1 0 1 1 0 1 Ende: 2
                    💻: 1
```

Die Speicherverwaltung der JVM ist unberechenbar

Fazit: Das Testergebnis ist nicht vorhersehbar und hängt – wie bereits gesagt – von der JVM und dem zur Verfügung stehenden Speicher ab und variiert von Aufruf zu Aufruf mit derselben `ARR_SIZE`.[15]

finalize(): ungeeignet für Ressourcen-Freigabe

Spätestens hier wird klar, dass `finalize()` schlichtweg nicht geeignet ist, wichtige System-Ressourcen wie Netzwerkverbindungen, Dateien, Fenster etc. zuverlässig freizugeben.

- Ob der GC überhaupt läuft, d.h. für unerreichbare Instanzen die Methode `finalize()` aufruft, und in welcher Reihenfolge dies dann noch geschieht, ist unvorhersehbar.

Eine ziemlich verrückte Folgerung daraus ist Code, der so lange für eine bestimmte JVM verändert wird, bis der GC endlich auf einer Testmaschine für alle wichtigen Objekte `finalize()` aufruft.[16]

5.9.3 Alternative zu finalize

Alternativen benötigt man nur dann, wenn man wichtige System-Ressourcen freigeben bzw. schließen muss. Ansonsten sollte man den GC in Ruhe arbeiten lassen.

Ressourcen-Freigabe: release()-Methode und Cleanup-Idiom

- Zur Freigabe von System-Ressourcen wird der Code von `finalize()` in eine `release()`- oder `close()`-Methode ausgelagert werden, die explizit und nicht mehr unvorhersehbar aufgerufen werden kann.
- Um sicherzustellen, dass zu einem Objekt die `release()`-Methode ausgeführt wird, wird das Cleanup-Idiom (siehe 3.3.4) verwendet.[17]

Diese Maßnahmen haben den Vorteil, dass die Freigabe nun unabhängig von der Objekt-Entsorgung ist, also auch bei Bedarf durchgeführt werden kann. Die `release()`-Methode wird zum Cleanup in den `finally`-Block gestellt.

Beispiel (Modifikation von TestGC)

Die Klasse `TestGC` in 5.9.2 wird mit einer `release()`-Methode ausgestattet. Die Anwendung, repräsentiert durch die Klasse `FinalizeTest`, wird mit Hilfe des Cleanup-Idioms angepasst:

15. Sollte die 1 in der letzten Zeile der letzten Ausgabe irritieren: Der GC ist ein asynchroner Thread, der auch noch nach dem `main()`-Thread aktiv sein kann und dadurch die Ausgabe in `finalize()` erzeugt (Näheres dazu in Kapitel 9, Threads).
16. Zum Beispiel durch Einfügen von `sleep()`-Anweisungen, um der GC-Thread Zeit zur Ausführung zu geben.
17. »Unbedingt« bedeutet auch die Ausführung nach Auftreten einer Ausnahme.

Ein Beispiel: Ressourcen-Freigabe mit release(), Cleanup

```
class TestGC {
  private static int alive;
  private long[] larr;
  private boolean isReleased;
  TestGC(int arrSize) {
    larr= new long[arrSize];
    alive++;
  }
  static int getAlive() { return alive; }
  void release() {
    if (!isReleased) {    // verhindert doppelte Ausführung
      // --- Freigabe/Schließen von System-Ressourcen ---
      System.out.print(--alive+" "); isReleased= true;
    }
  }
}
public class FinalizeTest {
  final static int MAX_UNREACHABLE= 10;
  final static int ARR_SIZE= 0;          // nun auch mit 0
  public static void main(String[] args) {
    System.out.print("Released: ");
    for (int i=0; i<=MAX_UNREACHABLE; i++) {
      TestGC t= null;
      try {
        t= new TestGC(ARR_SIZE);
      }
      // Cleanup-Idiom
      finally {
        if (t!= null) t.release();
      }
    }
    System.out.println("Ende: " +TestGC.getAlive());
  }
}
```

Cleanup-Idiom

Die Ausgabe ist nun unabhängig von ARR_SIZE und setzt zuverlässig alle Ressourcen frei, wie die Ausgabe zeigt:

```
💻: Released: 0 0 0 0 0 0 0 0 0 0 0 Ende: 0
```

5.9.4 Einsatz von finalize

Der Nachteil der letzten Lösung ist der explizit notwendige Aufruf einer release()- bzw. close()-Methode in der Anwendung. Jedoch verbietet diese Lösung nicht den zusätzlichen Einsatz von finalize(), die den unbestreitbaren Vorteil der Automatik hat.

Für unerreichbare Objekte, bei denen der Aufruf der release()-Methode einfach »vergessen« wurde, gibt zumindest der GC mit finalize() die Ressourcen frei – sofern er denn in Aktion tritt.[18]

Im Trio: Zusätzlich zu release() und Cleanup noch finalize()

Damit ist finalize() ein zusätzliches Sicherheitsnetz und nutzt dazu die bereits vorhandene release()-Methode, wobei sichergestellt sein muss, dass der Code in der release()-Methode nur einmal aufgerufen wird.

Beispiel (Weitere Modifikation von TestGC)

Die Klasse TestGC aus 5.9.3 wird nun zusätzlich mit einer finalize()-Methode ausgestattet:

```
class TestGC {
  private static int alive;
  private long[] larr;
  private boolean isReleased;
  TestGC(int arrSize) {
    larr= new long[arrSize]; alive++;
  }
  static int getAlive() { return alive; }
```

synchronized: Schutz vor gleichzeitiger Ausführung

```
  synchronized void release() {
    if (!isReleased) {
      System.out.print(--alive+" "); isReleased= true;
    }
  }
  protected void finalize() throws Throwable {
    System.out.print("f ");
    release();
    super.finalize();   // siehe 3. Regel unten
  }
}
```

18. finalize() ist notwendig, aber nicht hinreichend (im mathematischen Sinn).

Für ARR_SIZE=100000 interveniert nun ab und an der GC. Nachfolgend die Ausgabe:

Ein Ergebnis, das erfreut

```
💻: Released: o o f o f o f o f f o o f o f f o o f o f Ende: o
```

Idiom zu finalize

Wird mit `finalize()` gearbeitet, sollten noch folgende vier Punkte beachtet werden, da man ansonsten mit versteckten Fehlern bestraft werden kann:

finalize()-Konventionen unbedingt beachten

1. `finalize()` sollte nicht direkt aufgerufen werden, dies ist einzig Aufgabe des GC.
2. Die letzte Anweisung in der Implementierung von `finalize()` sollte immer `super.finalize()` sein.
3. Die zugehörige Instanz sollte in `finalize()` nicht wieder erreichbar gemacht werden.
4. Die `release()`-Methode, die `finalize()` aufruft, sollte synchronisiert sein.

Erklärung:

ad 1: Kombiniert man eine `public release()`-Methode mit einer `protected finalize()`-Methode, so ist dies wohl gewährleistet und der direkte Aufruf unnötig.

ad 2: Im Gegensatz zu Konstruktoren, bei denen immer der Superklassen-Konstruktor implizit ausgeführt wird, wenn er nicht explizit geschrieben wird, gilt dies nicht für `finalize()`.[19]

Sollten also irgendwelche »versteckten« Ressourcen freigesetzt werden müssen, die von Superklassen geerbt wurden, ist dies nur mit explizitem `finalize()`-Chaining zu realisieren.

ad 3: Wird in der `finalize()`-Methode eine gültige Referenz auf das Objekt gesetzt, ist es plötzlich wieder erreichbar. Dies veranlasst den GC dazu,

- das Objekt nicht aus dem Speicher zu entfernen
- nicht wieder `finalize()` aufzurufen, wenn das Objekt wieder unerreichbar wird und entsorgt wird.

finalize() wird immer nur einmal aufgerufen

19. Dies ist ein weiteres Indiz gegen die Destruktor-Theorie.

Besonders der letzte Punkt muss unbedingt bedacht werden.

Sollte man mittels `finalize()` einen Pool von wiederverwendbaren Objekten aufbauen wollen, so ist dies keine gute Idee. Dies geht einfacher mit Soft-Referenzen.

ad 4: GC ist ein eigener Thread und ruft `finalize()` asynchron auf.

5.10 Kapselung (Encapsulation)

Das Kapitel soll mit einer kurzen Besprechung einer wichtigen Design-Methodik – der Kapselung – beendet werden.

Kapselung: Ausgangspunkt aller OO-Design-Prinzipien

> Encapsulation: Es gibt keine direkten Zugriffe auf alle modifizierbaren Felder einer Klasse.

Dies bedeutet, dass entweder alle Felder der Klasse `final` oder `private` deklariert sind.[20] Zu den Feldern kommen eventuell noch Methoden hinzu, die man nur intern in der Klasse benötigt.

Nachteile der Kapselung

Kapselung: Arbeits-/Zeit-Aufwand durch Getters und Setters

- Für alle modifizierbaren Felder müssen Zugriffs-Methoden (**accessor methods)** geschrieben werden. Nach Java-Konvention haben sie die Namen `getX()`, `isX()` und `setX()`, wobei `X` der Name des Felds ist.[21]
- Da jeder Datenzugriff nur über eine Methode gehen kann, wird zusätzlich die Ausführungsgeschwindigkeit vermindert.

Vorteile der Kapselung

Kapselung: code-unabhängig Schutz vor unerlaubten Werten

- Die Details der internen Implementierung der Klasse werden geschützt, wodurch jederzeit eine Änderung möglich ist, ohne dass alle Anwender der Klasse ihren Code ändern müssen.
- Sind nur bestimmte Wertebereiche erlaubt, können nur `private` deklarierte Felder und Zugriffs-Methoden gegen unerlaubte Werte schützen.

20. `protected` als Zwischenform ist ein Zwitter, der von vielen als versteckter Bruch der Encapsulation angesehen wird.
21. Synonyme Begriffe für accessor methods sind deshalb: **Getters** und **Setters**

Fazit

Jede klasseninterne und -externe Methode, die auf ungeschützte Felder zugreift, muss deren Gültigkeit prüfen, da sie ansonsten inkorrekte Ergebnisse liefern könnte.

Mit oder ohne Kapselung: geprüft werden muss ohnehin!

Dies produziert dann genau wieder den Code bzw. Zeitverlust, den man durch Verzicht auf die Zugriffsfunktionen einsparen wollte.

Beispiel

Die Klasse Teil zeigt exemplarisch den Einsatz von zwei setX()-Methoden.

Kapselung am Beispiel Teil

Eigenschaften sind private

```
class Teil {
  private int nr;      // nicht alle Nr. sind erlaubt
  private String bez;  // keine Restriktion
  private int bArtId;  // sinnvoll: Eigenteil, Fremdteil

  // Konstruktoren und andere Methoden werden weggelassen
  public static boolean isValidNr(int nr) {
// kann recht komplex sein, hier eine Bereichsprüfung
     return 1000000<=nr && nr<=9999999;
  }
```

Setter für die Teile-Nr.

```
  public boolean setNr (int nr) {
// Alternative zu boolean: Exception
    if (isValidNr(nr)) {
      this.nr= nr;
      return true;
    }
    return false;
  }
```

Erweiterbarer Setter für Beschaffungsart

```
  public boolean setBeschaffungsart (String bArt) {
    boolean b= true;
    bArt= bArt.toUpperCase();
    if (bArt.equals("E"))         bArtId= 1;  // Eigenteil
    else if(bArt.equals("F"))     bArtId= 2;  // Fremdteil
    // else if(bArt.equals("EF")) bArtId= 3;                ①
    else b= false;
    return b;
  }
}
```

Ergebnis ohne Kapselung: Die Felder `nr` oder `bArtId` können jederzeit mit unsinnigen Werten besetzt werden. Jede Methode, die dann auf `nr` oder `bArtId` zugreift, muss somit eine Prüfung einbauen, ob die Felder gültige Werte enthalten.

Ergebnis mit Kapselung: Die interne Darstellung `bArtId` der Beschaffungsart ist nach außen transparent und kann jederzeit gewechselt werden. Stellt man fest, dass es auch Teile geben kann, die fremd und eigen gefertigt werden können, ist die Erweiterung der Zugriffs-Methode `setBeschaffungsart()` problemlos (siehe ①).

5.11 Zusammenfassung

Die Umsetzung der Generalisierung in Sub- und Superklassen in Java ist mit vielen sprachlichen Details verbunden.

Neben der Vererbung aller Member ist das Overriding – das Überschreiben von Instanz-Methoden – einer der wichtigsten Mechanismen.

Overriding ist unbedingt von Overloading und Shadowing zu unterscheiden. Statische Methoden können im Gegensatz zu Instanz-Methoden nicht überschrieben werden, obwohl dies umgangssprachlich genauso bezeichnet wird.

Eine Gruppe von Modifikatoren regelt die Art des Aufbaus einer Klassenhierarchie und den Zugriff. Verschiedene Modifikatoren können sich sinnvoll ergänzen oder auch gegenseitig ausschließen.

Eine syntaktisch eigene Gruppe von Konstruktoren regelt die Initialisierung von Objekten. Konstruktor-Chaining ist innerhalb einer Klasse sinnvoll und notwendig in der Klassenhierarchie, um die Initialisierung in der richtigen Reihenfolge zu erzwingen.

Destruktoren wie in C++ sind in Java unbekannt. Java hat eine automatische Speicherverwaltung GC, die den manuellen Aufruf von Destruktoren verhindern soll.

Der GC ruft zwar eine `finalize()`-Methode vor der Entfernung von Objekten aus dem Speicher aus, aber eine Speicherbereinigung von unerreichbaren Objekten kann nicht erzwungen werden.

Um wichtige Ressourcen freizugeben, wurden entsprechende Alternativen diskutiert.

Kapselung, der Schutz der Klassen-Implementierung vor dem direkten Zugriff von außen, ist das letzte von vielen Idiomen und Design-Prinzipien, die in diesem Kapitel vorgestellt wurden.

5.12 Testfragen

Zu jeder Frage können jeweils ein oder mehrere Antworten bzw. Aussagen richtig sein.

1. Welche Aussagen sind zu Overriding und Overloading richtig?

A: Beim Overriding sind die Signaturen der Methoden gleich.

B: Beim Overloading sind die Signaturen der Methoden gleich.

C: Beim Overloading sind die Namen der Methoden gleich.

D: Overriding kann nur in Subklassen vorkommen.

E: Beim Overriding kann der Rückgabe-Typ verschieden sein.

2. Die Klasse D ist Subklasse einer Klasse B. Welche Aussagen sind richtig?

A: Eine Variable vom Typ D kann eine Instanz von B referenzieren.

B: Eine Variable vom Typ B kann eine Instanz von D referenzieren.

C: Eine Instanz von D enthält alle nicht `private` deklarierten Instanz-Variablen von B.

D: Eine Instanz von D kann auf alle nicht `private` deklarierten Instanz-Variablen von B zugreifen.

E: D kann alle Instanz-Methoden von B überschreiben.

F: D kann nur alle `public` erklärten Instanz-Methoden von B überschreiben.

G: D kann eine Instanz-Methode von B ohne Zugriffs-Modifikator (Default-Zugriff) mit einer `protected` erklärten Methode überschreiben.

H: D kann eine `protected` erklärte Instanz-Methode von B mit einer Methode ohne einen Zugriffs-Modifikator überschreiben.

I: Instanzen von D können auf alle `protected` erklärten Felder von Instanzen von B zugreifen.

3. Welche Aussagen sind zu den Modifikatoren richtig?

A: Eine `abstract` erklärte Klasse hat mindestens eine abstrakte Methode.

B: Eine `abstract` erklärte Klasse darf keine `final` erklärten Methoden haben.

C: Eine `final` erklärte Klasse darf keine `abstract` erklärten Methoden haben.

D: Eine Methode kann nicht `private abstract` erklärt werden.

E: Eine Methode kann nicht `static abstract` erklärt werden.

4. Welche Aussagen sind zu Konstruktoren richtig?

A: Konstruktoren können nur Konstruktoren der direkten Superklasse aufrufen.

B: Konstruktoren einer Klasse können sich gegenseitig nur mit `this()` aufrufen (passende Argumente vorausgesetzt).

C: Sofern verwendet, muss `super()` die erste Anweisung in einem Konstruktor sein.

D: Die `super()`-Anweisung muss in einem Konstruktor vor der `this()`-Anweisung stehen.

E: Jeder Konstruktor einer Klasse (außer Object) enthält eine implizite oder explizite `super()`-Anweisung.

F: Die `super()`-Anweisung darf keine Argumente haben.

G: In einem Konstruktor darf keine abstrakte Methode aufgerufen werden.

5. Welche Aussagen sind zu Initialisierern richtig?

A: In einem Instanz-Initialisierer kann auf Variablen, die bei der Deklaration direkt initialisiert werden, immer zugegriffen werden.

B: Ein statischer Initialisierer wird immer vor einem Instanz-Initialisierer ausgeführt.

C: Ein Instanz-Initialisierer wird immer vor jedem Konstruktor ausgeführt.

D: Ein Instanz-Initialisierer kann mit `this()` einen Konstruktor aufrufen.

6. Welche Aussagen sind zu `super` und Shadowing richtig?

A: Mit `super` kann man auf jedes nicht `private` erklärte Feld einer Superklasse von einer Subklasse zugreifen.

B: Mit `Cast` kann man auf jedes nicht `private` erklärte Feld einer Superklasse von einer Subklasse zugreifen.

C: Mit `Cast` kann man auf jede nicht `private` erklärte Methode einer Superklasse von einer Subklasse zugreifen.

D: Mit `super` kann man nur auf die nicht `private` Methoden der direkten Superklasse von einer Subklasse zugreifen.

7. Welche Aussagen sind zur Garbage Collection (GC) und zu `finalize()` richtig?

A: Nicht erreichbare Instanzen können durch den GC entsorgt werden.

B: Nicht erreichbare Instanzen werden vor dem Programmende durch den GC entsorgt.

C: Von einer Instanz, die der GC aus dem Speicher entfernt, wird immer vorher die Methode `finalize()` aufgerufen.

D: `finalize()` muss überschrieben werden, ansonsten ist es wirkungslos.

E: `finalize()` ruft automatisch `finalize()` der Superklasse auf.

8. Folgende Vererbung liegt vor:

```
class A { void f(boolean p) {System.out.print (p+" ");}
          void f(int i)     {System.out.print (i+" ");} }
class B extends A { void f(boolean p){} }
```

Welche Aussagen sind zu folgendem Code-Fragment richtig?

```
B b= new B(); A a= (A)b;
a.f(1==1);                // Zeile 2
b.f(1==1);                // Zeile 3
b.f('1');                 // Zeile 4
```

A: Zeile 2 und Zeile 3 erzeugen beide die Ausgabe: `true`

B: Die Ausgabe in Zeile 2 und Zeile 3 ist gleich.

C: Die Ausgabe in Zeile 4 ist nicht 1.

D: Die Ausgabe in Zeile 4 erzeugt einen Laufzeitfehler.

9. Folgende Vererbung liegt vor:

```
class A { void f(int i) {System.out.print (i);} }

class B extends A{
  void f(int i)  {System.out.print (i+1); }
  void f(byte b) {System.out.print (b==0); }
  void f(char c) {System.out.print (c); }
}
```

Welche Aussagen sind zum nachfolgenden Code-Fragment richtig?

```
B b= new B(); A a= b;
```

A: Die Anweisung `b.f((byte)1);` erzeugt die Ausgabe: `false`

B: Die Anweisung `a.f((byte)1);` erzeugt die Ausgabe: `1`

C: Die Anweisung `a.f('1');` erzeugt die Ausgabe: `1`

D: Die Anweisung `a.f(1);` erzeugt die Ausgabe: `1`

E: Die Anweisung `a.f(1);` erzeugt die Ausgabe: `2`

10. Es liegt folgende Vererbung vor:

```
class A {
  int i;
  A(int i) { this.i=i; }
}
class B extends A {
  int i;
  // B() { }                        // Zeile 7
  // B() { this(0); }               // Zeile 8
  // B() { super(0); }              // Zeile 9
  B(int i) { super(i); this.i=i; }
}
```

Welche Aussagen sind richtig?

A: Die Deklaration `class C extends A{}` erzeugt einen Compiler-Fehler.

B: Die Anweisung `B b= new B();` erzeugt einen Compiler-Fehler.

C: Der Konstruktor `B()` in Zeile 7 würde einen Compiler-Fehler erzeugen.

D: Der Konstruktor `B()` in Zeile 8 würde einen Compiler-Fehler erzeugen.

E: Der Konstruktor `B()` in Zeile 9 würde einen Compiler-Fehler erzeugen.

11. Welche Aussagen sind zu der Klasse X richtig?

```
class X {
   static int si;
   int i;
   int[] iarr={1,2,3};
   static {
     // si= 1;                          // 6
     // i= 1;                           // 7
   }
   {
     // i=1;                            // 10
     // System.out.println(iarr[0]);    // 11
     // d= 0.0;                         // 12
   }

   X() {
     // System.out.println(i);          // 15
     // System.out.println(d);          // 16
     // d= 1.0;                         // 17
   }
   final double d;                       // 19
}
```

A: Die Anweisung in Zeile 6 würde einen Compiler-Fehler erzeugen.

B: Die Anweisung in Zeile 7 würde einen Compiler-Fehler erzeugen.

C: Die Anweisung in Zeile 10 würde einen Compiler-Fehler erzeugen.

D: Die Anweisung in Zeile 11 würde einen Compiler-Fehler erzeugen.

E: Die Anweisung in Zeile 12 würde einen Compiler-Fehler erzeugen.

F: Die Anweisung in Zeile 15 würde einen Compiler-Fehler erzeugen.

G: Die Anweisung in Zeile 16 würde einen Compiler-Fehler erzeugen.

H: Die Anweisung in Zeile 17 würde einen Compiler-Fehler erzeugen.

I: Auch wenn die Anweisung in Zeile 17 eingebunden wird und korrekt ist, erzeugt die Zeile 19 einen Compiler-Fehler.

12. Es liegt folgende Vererbung vor:

```
class A {
  char c='A';
  void f() { System.out.print(c+" "); }
  void g() { System.out.print("Hi "); }
}
class B extends A {
  char c='B';
  void f() { System.out.print(c+" "); }
}
class C extends B {
  // void f() {System.out.println(((A)this).c  );}    // 11
  // void f() {System.out.println(((A)this).f());}    // 12
  // void g() {super.g(); System.out.print(super.c);} // 13
}
```

Welche Aussagen sind richtig?

A: Die Deklaration von `f()` in Zeile 11 würde einen Compiler-Fehler erzeugen.

B: Die Deklaration von `f()` in Zeile 12 würde einen Compiler-Fehler erzeugen.

C: Die Deklaration von `g()` in Zeile 13 würde einen Compiler-Fehler erzeugen.

D: Die Anweisung `new C().g();` erzeugt die Ausgabe: `Hi B`

12 Serialisierung

Serialisieren bzw. Deserialisieren von Objekten ist ein Teilbereich von Java-I/O, geht aber weit über den Rahmen der traditionellen Kommunikation hinaus.
Hinter dem Begriff »Serialisierung« verbergen sich diverse Ebenen der Kommunikation auf Basis von Objekten.
Im Idealfall kann man durch Markieren der Klasse einfach alles der JVM überlassen. Im anderen Fall übernimmt man den gesamten Prozess des Marshalings bzw. Unmarshalings komplett.
In jedem Fall muss man sich über die Konsequenzen seiner Entscheidung in Bezug auf Vererbung, referenzierte Klassen, Klassen-Versionen und Sicherheiten im Klaren sein.

12.1 Serialisierung: Kommunikation auf Basis von Objekten

Einführung

Serialisierung ist der synonym verwendete Begriff für Objekt-Streams, d.h. das Lesen und Schreiben von Objekten. Serialisierung ist nicht auf singuläre Objekte beschränkt, da diese ja wieder andere Objekte referenzieren können.

Übertragen von Objekt-Graphen

Von primitiven Typen einmal abgesehen, müssen beliebige Objekt-Gebilde, präziser Objekt-Graphen, in ihrem aktuellen Zustand serialisiert und in einem Kanal übertragen werden.

- Das Hauptziel der Serialisierung ist ein einfacher standardisierter Default-Mechanismus zur Übertragung von Objekten, der stufenweise erweitert oder auch ersetzt werden kann.

Object-Output(Stream) Object-Input(Stream)

Die Objekt-Kommunikation führt die JVM im Normalfall mit Hilfe von Instanzen der Klassen `ObjectOutputStream` bzw. `ObjectInputStream` aus. Die beiden zentralen Methoden in diesen beiden Klassen sind in den Interfaces `ObjectOutput` bzw. `ObjectInput` deklariert:

write/readObject: Object-Byte-Stream

- `writeObject()`, das ein Objekt in einen Byte-Stream umwandelt.
- `readObject()`, das einen Byte-Stream in ein Objekt umwandelt.

Hierzu benötigen beide Methoden den vollen Zugriff auf das zu übertragende Objekt.

Da beide Klassen indirekt die Interfaces `DataOutput` bzw. `DataInput` implementieren, können sie auch alle primitiven Daten serialisieren.

12.2 Grundlagen der Serialisierung

Allen Protokollen und Mechanismen der Serialisierung liegt eine Prämisse zugrunde:

Objekt-Rekonstruktion aus einer Byte-Sequenz

- Der Empfänger muss anhand des erhaltenen Byte-Streams in der Lage sein, die ursprünglichen Objekte zu rekonstruieren.
- Dies sollte auch dann möglich sein, wenn bei der Deserialisierung nur eine kompatible Klassen-Version vorliegt.

Gerade der letzte Punkt ist für Klassenerweiterungen erforderlich, allerdings auch nicht einfach zu realisieren.

Serialisierungs-Framework

Das Framework besteht aus zwei Teilen (Abb. 12.1).

Framework zur Serialisierung

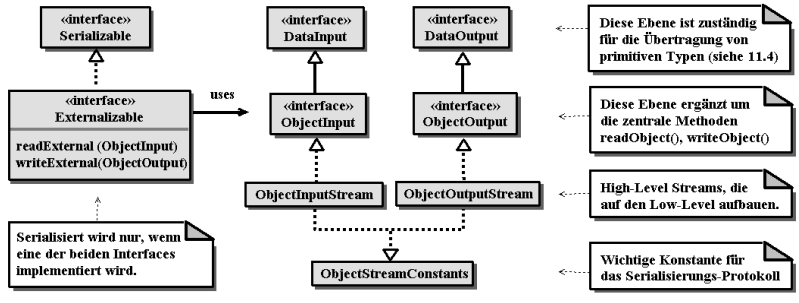


Abbildung 12.1 Interfaces und Klassen zur (De-)Serialisierung

Serialisierung beruht auf Interfaces

Im Unterschied zum allgemeinen I/O- bzw. Streaming-Konzept (vgl. dazu Kapitel 11, Package java.io) beruht die Objekt-Kommunikation auf Interfaces. Somit ist auch eine komplett eigencodierte Serialisierung möglich.

Allgemein gilt:

- Klassen, deren Objekte serialisiert werden sollen, implementieren `Serializable` bzw. `Externalizable`.
- Klassen, die die eigentliche Kommunikation vornehmen, implementieren die Interfaces `ObjectOutput` bzw. `ObjectInput`.

12.2.1 Standard-Serialisierung

Zuerst werden die in der Plattform realisierten Standard-Serialisierungsmechanismen besprochen, da sie zur Realisierung von Erweiterungen oder Ersatz unbedingt notwendig sind.

Das Marshaling bzw. Unmarshaling der Daten wird von Instanzen der Klasse `ObjectOutputStream` bzw. `ObjectInputStream` übernommen:

writeObject() serialisiert

- Um ein Objekt zu serialisieren, wird es der Instanz-Methode `writeObject()` von `ObjectOutputStream` übergeben.

readObject() deserialisiert

- Um ein neues Objekt – eine Art von Klon – auf der anderen Seite des Kanals zu deserialisieren, wird es der Instanz-Methode `readObject()` von `ObjectInputStream` übergeben (siehe Abb. 12.2).

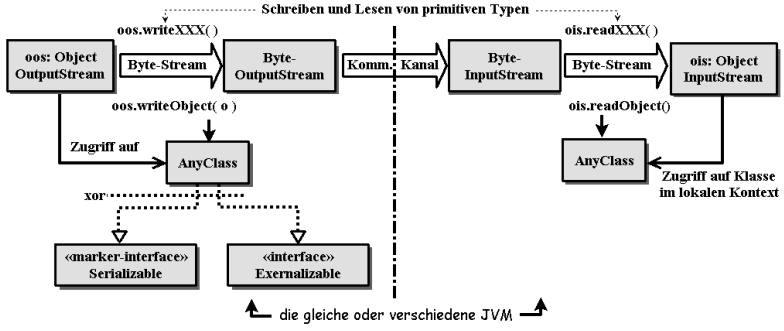


Abbildung 12.2 (De-)Serialisierung von Objekten

Prinzipielles Code-Muster

Zum Serialisieren verwendet man Referenzen der Interfaces `ObjectOutput` bzw. `ObjectInput`:

Standard-Code-Muster für Objekt-Übertragung

```
// Senden eines Objekts serObjectIn der Klasse SerClass
ObjectOutput oos= new ObjectOutputStream(byteOutStream);
oos.writeObject(serObjectIn);
// Empfang eines Objekts serObjectOut der Klasse SerClass
ObjectInput ois= new ObjectInputStream(byteInStream);
serObjectOut= (SerClass) ois.readObject();
```

Der Cast in der letzten Zeile ist notwendig, da `readObject()` ein `Object` zurückliefert.

Grundlegende Serialisierungs-Regeln

Grundsätzliche Serialisierungs-Regeln

1. Es werden nur Objekte serialisiert, keine statischen (Klassen-)Felder.
2. Die Klassen dieser Objekte müssen eines der Interfaces `Serializable` oder `Externalizable` implementieren.
3. Findet die Kommunikation zwischen zwei unterschiedlichen Prozessen statt, müssen beide JVMs Zugriff auf die Klasse des Objekts haben, da kein Byte-Code der Klasse übertragen wird.

Zu 1: Ein Objekt kann nur deserialisiert werden, wenn seine Klasse geladen ist. Dann sind aber bereits die statischen Felder der Klasse initialisiert.

Zu 2: Bei `Serializable` handelt es sich um ein Marker-Interface.

Zu 3: Bei unterschiedlichen Prozessen, z.B. Netzwerkübertragung von Objekten, muss sichergestellt sein, dass die andere JVM ebenfalls Zugriff auf (kompatible) Klassen der übertragenen Objekte hat.

Primitive Typen

Serialisieren von Daten primitiver Typen

Neben Objekten können mit Hilfe der Methoden `write<X>()`[1] und `read<X>()` der Interfaces `DataOutput` und `DataInput` auch primitive Typen übertragen werden (siehe Abb. 12.1 und 12.2).

12.3 Übertragungs-Protokolle

Selbst wenn nur Daten primitiver Typen übertragen werden, beruht die Serialisierung auf einem vereinbarten **Protokoll**.

Serialisierungs-Protokoll mit Header

- Das **Serialisierungs-Protokoll** legt anhand einer Grammatik den Aufbau eines Byte-Streams fest.

Den Anfang eines Byte-Streams bildet ein Header, der aus der `short`-Konstanten `STREAM_MAGIC` (`0xaced`) und der Version `STREAM_VERSION` (zurzeit aktuell `0x0005`) besteht (siehe Beispiel in 12.2.2).

12.3.1 Protokoll zu primitiven Typen

Block-Data-Record für write<X>-Methoden

Die mittels `write<X>()` übertragenen Daten werden in einem Block, dem so genannten **Block-Data-Record**, mit Kennung und Länge übertragen.

Die verwendeten Symbole sind im Interface `ObjectStreamConstants` (Abb. 12.1) definiert.

1. <X> steht für die Bezeichnung des primitiven Typs, z.B. `writeInt()`.

Wie bereits bei der einfachen I/O kann es leider auch hier noch zu Übertragungsfehlern, d.h. Fehlinterpretationen der Daten kommen:

- Das Protokoll für die Serialisierung von Daten primitiver Typen mittels `write<X>()` bzw. `read<X>()` lässt keine zweifelsfreie Rekonstruktion zu.

Fehlinterpretationen nicht ausgeschlossen

Beispiele

Um Interna der Serialisierung besser zu verstehen, benötigt man eine Hex/ASCII-Darstellung der Byte-Streams. Hierzu verwenden wir im Folgenden die statische Methode `toHexAsciiString()` der Utility-Klasse `Sniffer`:

Sniffer: Byte-Arrays in Hex/ASCII-Darstellung

```
class Sniffer {
  private static char hex[]= {'0','1','2','3','4','5','6','7',
                              '8','9','a','b','c','d','e','f'};
  public static String toHexAsciiString (byte[] b,int lfAtPos){
    if (b==null || b.length==0) return "";
    StringBuffer sb= new StringBuffer(4*b.length);
    int i,r;
    for (i= 0; i<b.length;i++) {
      sb.append(hex[(b[i]>>>4)&0xF]).append(hex[b[i]&0xF])
                                    .append(' ');
      if((i+1)%lfAtPos==0) {
        for (int j=i-lfAtPos+1; j<=i; j++) {
          if (32<=b[j] && b[j]<=126) sb.append((char)b[j]);
          else sb.append('.');
        }
        sb.append('\n');
      }
    }
    if (0 < (r= b.length%lfAtPos)) {
      for (i= 0;i<(lfAtPos-r)*3;i++) sb.append(' ');
      for (i= b.length-r;i<b.length; i++) {
        if (32<=b[i] && b[i]<=126) sb.append((char)b[i]);
        else sb.append('.');
      }
    }
    else sb.deleteCharAt(sb.length()-1);
    return sb.toString();
  }
}
```

Die folgende Kommunikation mit ByteArrayOutputStream und ByteArrayInputStream ist rein speicherbasierend, sehr schnell und aus zwei Gründen interessant:

- Sie lässt die Analyse der Struktur des Byte-Streams durch die Sniffer-Klasse sehr einfach zu.
- Sie erlaubt ein einfaches Cloning im Sinne von Deep-Copy (siehe auch 6.10.2).

Byte-Stream eines Block-Data-Records

```
public class Test {
  public static void main(String[] args) {
    byte[] barr= null;
    ByteArrayOutputStream baos= new ByteArrayOutputStream();
    try {
      // schreibt bereits Header, siehe Erklärung unten
      ObjectOutput oout= new ObjectOutputStream(baos);          ①
      oout.writeInt(7);
      oout.writeBoolean(false);
      oout.write((byte)49);
      oout.flush();              // schreibt nach baos
```

Sniffer im Einsatz

```
      // wandelt Byte-Stream in Byte-Array um
      barr= baos.toByteArray();
      // Hex/ASCII-Darstellung mit 16 Zeichen pro Zeile
      System.out.println(Sniffer.toHexAsciiString(barr,16));
    ObjectInput oin= new ObjectInputStream(
                        new ByteArrayInputStream(barr));
      System.out.println(oin.readInt());
      System.out.println(oin.readByte());                       ②
      System.out.println(oin.readBoolean());                    ③
      // System.out.println(oin.readChar());                    ④
    } catch (IOException e) { System.out.println(e); }
  }
}
```

Hex/ASCII: Header, Block-Kennung und -Länge, Daten primitiver Typen

```
ac ed 00 05 77 06 00 00 00 07 00 31            ....w......1
7
0
true
```

Tabelle 12.1 Ausgabe zu Test

Zu ①: Wie am Anfang dieses Abschnitts beschrieben, bilden die ersten vier Bytes den Header. Dieser wird bereits bei der Anlage des `ObjectOutputStream` mit `writeStreamHeader()` herausgeschrieben. Deshalb muss die Anweisung im `try-catch` stehen.

Zum Block-Data-Record: Der Block-Anfang ist mit `77` (`TC_BLOCKDATA`) markiert, `06` ist dann die Block-Länge und anschließend folgen die Daten.

Zu write<X>, read<X>: Typ-Informationen sind im Stream nicht enthalten. Deshalb können die Zeilen ② und ③ z.B. auch durch die Zeile ④ ersetzt werden, ohne dass dieser Fehler erkannt werden kann. Dies würde dann das Zeichen `1` liefern.

Codierung von Zeichen

Zeichen-Codierung in UTF8

- Grundsätzlich werden alle Zeichen vom Typ `char` oder `String` als UTF8 im Stream codiert.

12.3.2 Protokolle zur Objekt-Serialisierung

Für Objekte ist das Protokoll sowie das Stream-Format verständlicherweise wesentlich komplexer.

Mit den Interfaces `Serializable` und `Externalizable` sind zwei unterschiedliche Protokolle verbunden, die festlegen wie Objekte übertragen werden. Zusätzlich zu den bereits o.a. grundsätzlichen Regeln gilt:

Grundlegende Protokoll-Regeln

1. Implementiert eine Klasse keine der beiden Interfaces, so führt der Versuch, ein Objekt dieser Klasse zu serialisieren, zu der Ausnahme `NotSerializableException`.
2. Für die Klassen, die `Externalizable` implementieren, überträgt das Protokoll nur den voll qualifizierten Klassennamen. Alle weiteren Informationen zu den Feldern müssen von der Klasse selbst im Byte-Stream mittels der beiden Methoden
   - `public void writeExternal(ObjectOutput out)`
   - `public void readExternal(ObjectInput in)`

   im Interface `Externalizable` codiert werden.
3. Für Klassen, die nur `Serializable` implementieren, legt das Protokoll automatisch die Reihenfolge und Identifizierung seiner Felder und seiner Superklassen fest, wobei
   - `static` deklarierte Klassen-Variablen nicht serialisiert werden
   - die Klasse die zu übertragenden Instanz-Felder selbst bestimmen kann

Default-Serialisierung: Klassen implementieren nichts außer Serializable

Da Externalizable das Interface Serializable erweitert (Abb. 12.1), ist dass »nur« in der letzten Regel wesentlich. Denn alle Klassen, die Externalizable implementieren, sind auch als Serializable markiert.

Externalizable vs. Serializable

Die zweite und dritte Regel hat eine wichtige Implikation:

Externalizable Klassen: Einbahnstraße für Subklassen

- Für die Subklassen einer Klasse, die Externalizable implementiert, gibt es keine automatische Serialisierung mehr.

Der folgende Code ist syntaktisch korrekt, aber semantisch sinnlos:

```
class E implements Externalizable         { /*...*/ }

class D extends E implements Serializable { /*...*/ }
```

Die Klasse D muss trotzdem die Serialisierung selbst übernehmen.

12.3.3 Protokoll zu Externalizable

Das Interface Externalizable deklariert zwei Methoden, die jeweils als Argument einen Object-Stream übergeben bekommen.

Externalizable: kein Marker-Interface

```
public interface Externalizable extends Serializable {
  void writeExternal(ObjectOutput out) throws IOException;
  void readExternal (ObjectInput in)   throws IOException,
                                              ClassNotFoundException;
}
```

Aufgrund der Object-Streams out bzw. in stehen bei der Implementierung der Methoden somit alle Methoden zum Lesen und Schreiben von primitiven Typen und Objekten zur Verfügung.

Anforderungen an externalizable Klassen

Anforderungen an externalizable Klassen

Klassen, die bei der Objekt-Kommunikation eine vollständige Kontrolle über die Feld-Inhalte benötigen, müssen

- Externalizable mit den beiden Methoden writeExternal() bzw. readExternal() implementieren.
- einen public deklarierten No-Arg-Konstruktor [2] enthalten.

Im Gegensatz zu rein Serializable-Klassen kann eine Externalizable-Klasse von außen beliebig instanziiert werden, was für das Design recht unangenehme Konsequenzen haben kann (siehe auch 12.4.2).

2. Konstruktor ohne Argumente

Protokoll-Ablauf zu Externalizable

Protokoll-Ablauf bei Externalizable

Da die Hauptarbeit der Objekt-Kommunikation bis auf den Klassennamen nicht automatisch erfolgt, ist das Protokoll eigentlich recht einfach.

Bei der Serialisierung wird

1. der voll qualifizierte Klassenname des Objekts ASCII-codiert (nicht in Unicode!) in den Stream geschrieben.[3]
2. die Methode `writeExternal()` der Klasse mit dem entsprechenenden `ObjectOutput`-Stream aufgerufen.

Bei der Deserialisierung wird umgekehrt

1. die Klasse anhand des im Stream enthaltenen Namens identifiziert.
2. eine Instanz der Klasse mit Hilfe des `public` No-Arg-Konstruktors erschaffen.
3. die Methode `readExternal()` der Klasse mit dem entsprechenenden `ObjectInput`-Stream aufgerufen.

Beliebige Manipulation der Feld-Informationen

Mit writeExternal() werden die gewünschten Felder des Objekts in den Stream geschrieben, wobei beliebige Manipulationen, z.B. Verschlüsselung der Daten, möglich sind.

Sollen interne Objekte übertragen werden, muss dies selbst kodiert werden.

Reihenfolge der Felder ist wichtig

Bei `readExternal()` ist dann darauf zu achten, dass die Reihenfolge der Felder exakt eingehalten wird und eventuelle Entschlüsselungen vorgenommen werden.

Externalizable und Felder von Superklassen

Externalizable: Felder von Superklassen nicht implizit enthalten

- Die Felder aller Superklassen – selbst wenn diese als `Serializable` markiert sein sollten – müssen ebenfalls explizit übertragen werden.

Gemäß dem o.a. zweiten Punkt der Deserialisierung, werden zwar automatisch die Felder der Superklassen eines Objekts angelegt, dies sorgt aber nur für die Default-Initialisierung.

Enthält das zu übertragende Objekt davon abweichende Werte, sind diese ohne explizite Übertragung im deserialisierten Objekt nicht vorhanden.

3. Wer also unbedingt darauf besteht, seine Packages und Klassen mit Nicht-ASCII-Zeichen, z.B. deutschen Umlauten, zu benennen, hat spätestens bei der Serialisierung in Netzen viel Freude an seiner Wahl.

Beispiel

Nachfolgend wird eine sehr einfache Klasse E angelegt:

```
package kap12;
import java.io.*;
```

Beispiel: externalizable Klasse

```
class E implements Externalizable {
  byte b= 1;
  public E() {};                    // public notwendig!
  E(byte b)  { this.b= b; }
  public void writeExternal (ObjectOutput out) {
    try { out.writeByte(b);
    } catch (Exception e) {System.out.println(e);}
  }
  public void readExternal (ObjectInput in) {
    try { b= (byte) in.readByte();
    } catch (Exception e) {System.out.println(e);}
  }
}
```

Ein Objekt der Klasse E wird nun wieder (de-)serialisiert und der Byte-Stream mit Sniffer ausgegeben:

```
public class Test {
  public static void main(String[] args) {
    ObjectOutput oout= new ObjectOutputStream(baos);
    oout.writeObject(new E((byte)3)); oout.flush();
    barr= baos.toByteArray();
    System.out.println(Sniffer.toHexAsciiString(barr,16));
    ObjectInput oin= new ObjectInputStream(
                        new ByteArrayInputStream(barr));
    System.out.println(((E)oin.readObject()).b);
  }
}
```

Beispiel: Protokoll-Bytes einer externalizable Klasse

```
ac ed 00 05 73 72 00 07 6b 61 70 31 32 2e 45 c1 ....sr..kap12.E.
b5 79 8a 99 51 65 29 0c 00 00 78 70 77 01 03 78 .y..Qe)...xpw..x
3
```

Tabelle 12.2 Ausgabe zu Test

Erklärung: Nach dem obligatorischen 4-Byte-Header startet das Objekt im Stream mit TC_OBJECT (73).

Nach der Klassenkennung TC_CLASSDESC(72) folgt die Länge (0007) und der Name der Klasse (kap12.E) als ASCII-Zeichen.

Anschließend folgt ein 64-Bit-Hashcode (c1..29), der die Klassen eindeutig identifiziert.

Am Ende stehen dann die Feldwerte des Objekts in einem Block. In diesem Fall ist dies nur ein Byte (03), eingerahmt in TC_BLOCKDATA(77), Länge (01) und TC_ENDBLOCKDATA(78).

Fazit

Auch bei Externalizable ist das Protokoll bis auf die eigentlichen Feld-Informationen vorgegeben. Die Daten der Felder sind aber frei manipulierbar.

Externalizable: eigenes Protokoll zu Feldern

12.3.4 Protokoll zu Serializable

Implementiert eine Klasse Serializable, so kann die Serialisierung völlig transparent von der JVM übernommen werden.

Allerdings stellt das Protokoll gewisse Anforderungen an die Objekte und ihre Klassen.

1. Anforderungen an eine Klasse

Eine Klasse ist **serializable**[4], wenn

Anforderungen an serializable Klassen

- keine ihrer Superklassen Externalizable implementiert und
- sie Zugriff auf den No-Arg-Konstruktor der ersten Superklasse hat, die nicht serialisierbar ist.[5]

2. Anforderungen an ein Objekt

Anforderungen an serializable Objekte

Ein Objekt ist dann serializable, wenn

- seine Klasse serializable ist und
- seine Referenz-Felder entweder null sind oder Objekte von Klassen referenziert, die serialisierbar sind.

4. Im Unterschied zum deutschen Begriff »serialisierbar«, der offen lässt, ob die Klasse Serializable oder Externalizable implementiert.
5. Sollte der Konstruktor nicht im Zugriff stehen, kommt es beim Deserialisieren zu einer InvalidClassException.

Default-Serialisierung

Default-Serialisierung

Unter dem Begriff **Default-Serialisierung** versteht man das Standard-Protokoll einer serializable Klasse, die selbst keine Serialisierungs-Methoden implementiert.

It's magic – read/writeObject(): Zugriff auf private Felder

Das Standard-Protokoll verwendet hierzu `writeObject()` bzw. `readObject()` von `ObjectOutputStream` bzw. `ObjectInputStream`.

- Die Methoden `writeObject()` und `readObject()` haben dazu »magischen« Zugriff auch auf alle `private` deklarierten Felder der Objekte.

Prinzipieller Ablauf der Default-Serialisierung

Ablauf der Default-Serialisierung

Der Serialisierungs-Prozess eines Objekts mit der Methode `writeObject()` läuft im Wesentlichen wie folgt ab[6]:

1. Es werden der Klassen-Deskriptor, d.h. der Klassenname sowie die Namen aller nicht `transient` deklarierten Felder in den Stream geschrieben.
2. Die Werte der Felder werden mit der Methode `defaultWriteObject()` in den Stream geschrieben.

Transitive Hüllen-Operation

3. Dabei werden alle Objekte, die über Felder vom Objekt erreichbar sind, ebenfalls in den Stream geschrieben, wobei für Objekte, die bereits serialisiert sind, nur ein Handle abgelegt wird. Diese Operation wird kurz als **transitive Hülle** (transitive closure) bezeichnet.
4. Ist ein Objekt Instanz einer serializable Subklasse, wird für die serialisierbaren Superklassen ebenfalls `writeObject()` oder es werden ihre speziellen Serialisierungs-Methoden[7] aufgerufen.

ObjectStreamClass zur Feldbeschreibung

ObjectStream-Class-Descriptor

Im Gegensatz zu `Externalizable` werden also zu jedem Objekt auch die Namen der übertragenen Felder im Stream geschrieben.

Zu Klassen- bzw. Feldbeschreibungen wird eine Instanz des Klassen-Deskriptors `ObjectStreamClass` serialisiert.

Standard-Objekte wie `Object` oder `String` werden allerdings nur als Ident in den Stream geschrieben (siehe Konstanten in `ObjectStreamConstants`).

6. Details werden hier weggelassen, da sie für den prinzipiellen Ablauf uninteressant sind. Sie können aber im Object Serialization ReleaseStream Protokoll von Sun nachgelesen werden.
7. Zu speziellen Serialisierungs-Methoden siehe Abschnitte ab Einfache Anpassungen von Serializable in diesem Kapitel.

Beispiel

Beispiel: Klassen-Hierarchie mit Aggregation

Objekte der folgenden Klasse Square sind serializable, unabhängig davon, ob die Superklasse Figure serialisierbar ist oder nicht.

Point muss allerdings in jedem Fall serialisierbar sein (vgl. 12.3.4 »Anforderungen an ein Objekt«).

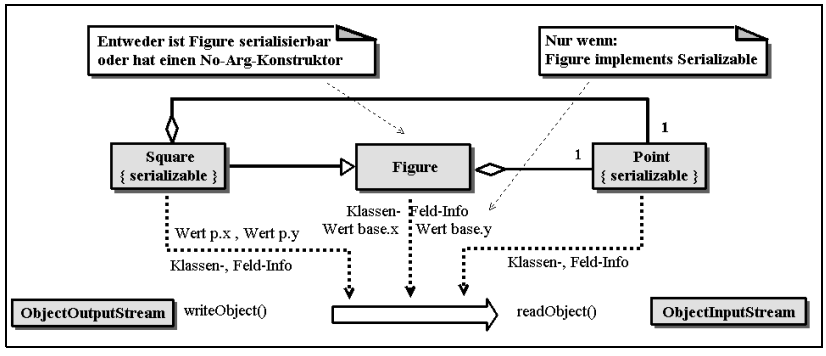


Abbildung 12.3 Klassen-Diagramm zum Beispiel

```
class Point implements Serializable {  int x,y; }
class Figure /* implements Serializable */ {            ①
  protected Point base= new Point();
  public  Figure()            { this(0,0); }
  public  Figure(int x, int y) { base.x=x; base.y=y; }
}
class Square extends Figure implements Serializable {
  private Point p= new Point();
  transient public boolean clockwise= true;
  public Square (int x1, int y1, int x2, int y2) {
    super(x1,y1); p.x= x2; p.y= y2;
  }
  public String toString() {
    return "["+ base.x+","+ base.y+";"+p.x+","+p.y+";"+
           clockwise+"]";
  }
}
```

Zu ①: Klassen- bzw. Feld-Informationen und Werte zu Figure werden in den Stream nur aufgenommen, wenn die Klasse serialisierbar ist, z.B. Serializable implementiert.

Objekt-Default-Deserialisierung

Ablauf der Default-Deserialisierung

Der Deserialisierungs-Prozess der Methode `readObject()` für ein Objekt läuft prinzipiell wie folgt ab:

1. Nach Deserialisierung der `ObjectStreamClass`-Instanz werden die Klassen- bzw. Feldbeschreibungen ausgewertet und die zugehörige Klasse wird im lokalen System geladen.
2. Es wird eine Instanz erzeugt.
   - Für externalizable Objekte wird der `public` No-Arg-Konstruktor aufgerufen und anschließend `readExternal()`.
   - Für serializable Objekte wird – falls notwendig – der No-Arg-Konstruktor der ersten nicht serialisierbaren Superklasse aufgerufen.
3. Die Felder werden ohne Aufruf eines Konstruktors oder Instanz-Initialisierers mit Hilfe der Methode `defaultReadObject()` mit Werten belegt.
   - Felder, für die es keinen Wert im Stream gibt, erhalten den Default-Wert.

Die neu erschaffene Instanz ist somit total unabhängig vom originalen Objekt. Abschließend noch eine Anmerkung:

EOFException bei Vertauschung

- Der Versuch, ein Objekt als primitiven Typ zu deserialisieren, führt zu der Ausnahme `EOFException`.

Beispiel

Die Ausgabe von `Test` hängt also davon ab, ob `Figure` (siehe oben, Zeile ①) das Interface `Serializable` implementiert hat.

```
public class Test {
  public static void main(String[] args) {
    ByteArrayOutputStream baos= new ByteArrayOutputStream();
    ObjectOutput oout= new ObjectOutputStream(baos);
    oout.writeObject(new Square(1,2,3,4)); oout.flush();
    ObjectInput oin= new ObjectInputStream(
                       new ByteArrayInputStream(
                          baos.toByteArray()));
    System.out.println(((Square)oin.readObject()));
    // Figure nicht serialisierbar 💻: [0,0;3,4;false]
    //       Figure serialisierbar 💻: [1,2;3,4;false]
  }
}
```

12.4 Einfache Anpassungen von Serializable

Anpassung der Serialisierungs-Mechanismen

Neben den Varianten Default-Serialisierung bzw. Serialisierung mittels Externalizable gibt es noch weitere – teilweise recht komplexe – Möglichkeiten der Protokoll-Anpassung.

12.4.1 serialPersistentFields: Ersatz für transient

serialPersistent-Fields überschreibt transient

Die Kennzeichnung der nicht serialisierbaren Instanz-Felder als transient ist zwar einfach, aber manchmal nicht flexibel genug.[8]

Dieser Default-Mechanismus lässt sich mit Hilfe des speziellen private static final deklarierten Felds serialPerstistentFields innerhalb der serializable Klasse überschreiben.

serialPersistent-Fields: ein Code-Muster

- Das Feld serialPerstistentFields muss mit einem Array von ObjectStreamField-Instanzen initialisiert werden, die die Namen und Typen der serialisierbaren Instanz-Felder enthalten:

```
class C implements Serializable {
  // ..Felder..
  private static final ObjectStreamField[]
    serialPersistentFields = {
      new ObjectStreamField("fieldname", fieldType.class),
      //...
    };
  //...
}
```

Beispiel

In der Klasse C werden trotz transient die Felder s und i serialisiert:

```
class C implements Serializable {
  transient String s= "abc";      // transient nutzlos
  transient int i= 2;             // dito

  private static final ObjectStreamField[]
      serialPersistentFields = {
        new ObjectStreamField("s", String.class),
        new ObjectStreamField("i", int.class)
      };
}
```

8. Zum Beispiel dann, wenn man ein Feld zu einer Klasse hinzufügen muss, zu der bereits serialisierte Objekte existieren, die auch gelesen werden müssen.

12.4.2 Externalizable: Kapselung unmöglich

Keine Kapselung: gravierender Nachteil von Externalizable

Ein gravierender Nachteil der Implementierung von `Externalizable` liegt darin, dass die Methoden `writeExternal()` bzw. `readExternal()` `public` deklariert werden müssen.

- Die `Externalizable`-Methoden können nicht nur von der JVM zur Serialisierung, sondern für jeden anderen Zweck missbraucht werden.

Aus Sicht der Kapselung sind Anpassungen bzw. Erweiterungen des Default-Mechanismus sicherlich die bessere Wahl.

12.4.3 Klasseninterne Anpassung der Default-Serialisierung

Klassenintern read/write Object() überschreiben Default-Mechanismus

Mit Hilfe der klasseninternen `private` deklarierten Methoden `writeObject()` und `readObject()` kann eine serializable Klasse den Default-Mechanismus anpassen.

Dazu muss die Klasse diese Methoden mit folgender Signatur (mit oder ohne `throws`) implementieren:

```
class C implements Serializable {

defaultWriteObject(): Aufruf des Defaults
  private void writeObject (ObjectOutputStream oout)
               throws IOException {
    //...
    // oout.defaultWriteObject();
    //...
  }

  private void readObject (ObjectInputStream oin)
               throws ClassNotFoundException, IOException {
    //...
    // oin.defaultReadObject();
    //...
  }
}
```

Code-Muster

defaultRead Object(): Aufruf des Defaults

Auch hier wieder: It´s magic

Diese beiden Methoden sind wie ihre gleichnamigen Pendants in den beiden Object-Streams magisch.[9]

- Da `ObjectOutputStream` bzw. `ObjectInputStream` als Argument übergeben wird, können mit Hilfe der Default-Methoden die Werte der »normalen« Felder serialisiert werden.

9. Sie werden in keinem Interface erklärt, von der JVM automatisch aufgerufen und ersetzen ihre gleichnamigen Methoden in den Object-Streams.

Beispiel (in drei Varianten)

Die Klasse C enthält jeweils drei Varianten zu `writeObject()` bzw. `readObject()`. Beide Methoden

1. sind leer, d.h. ohne Anweisung.
2. rufen nur ihre Default-Methoden auf (zusätzlich Zeile ① bzw. ④).
3. rufen ihre Default-Methoden auf und setzen Datum und Zeit des statischen Felds d (zusätzlich Zeile ① ② bzw. ③ ④).

read/writeObject(): drei interne Varianten

```
class C implements Serializable {
  static Date d;
  String s= "hi";
  int i= 7;
  private void writeObject (ObjectOutputStream oout) {
    try {
      // oout.defaultWriteObject();                              ①
      // oout.writeObject(Calendar.getInstance().getTime());  ②
    } catch (Exception e) {System.out.println(e);}
  }
  private void readObject (ObjectInputStream oin) {
    try {
      // oin.defaultReadObject();                                ③
      // d= (Date) oin.readObject();                             ④
    } catch (Exception e) {System.out.println(e);}
  }
  public String toString() { return d+","+s+","+i; }
}
public class Test {
  public static void main(String[] args) {
    byte[] barr= null;
    ByteArrayOutputStream baos= new ByteArrayOutputStream();
    ObjectOutput oout= new ObjectOutputStream(baos);
    oout.writeObject(new C()); oout.flush();
    barr= baos.toByteArray();
    System.out.println(Sniffer.toHexAsciiString(barr,16));
    ObjectInput oin= new ObjectInputStream(
                        new ByteArrayInputStream(
                                    baos.toByteArray()));
    System.out.println(oin.readObject());
  }
}
```

1. Variante

Es werden nur die Klassen- und Feld-Informationen übertragen, da eine Instanz der `ObjectStreamClass` serialisiert wird.

Es fehlen alle Werte. Die Ausgabe ist uninteressant und wird deshalb weggelassen.

2. Variante

Mit mehr Aufwand hat man praktisch die Default-Serialisierung nachgebildet. Auch der Stream-Inhalt ist nahezu identisch mit demjenigen der Default-Serialisierung.

```
ac ed 00 05 73 72 00 07 6b 61 70 31 32 2e 43 2c ....sr..kap12.C,
92 61 0f 51 e2 bc ff 03 00 02 49 00 01 69 4c 00 .a.Q......I..iL.
01 73 74 00 12 4c 6a 61 76 61 2f 6c 61 6e 67 2f .st..Ljava/lang/
53 74 72 69 6e 67 3b 78 70 00 00 00 07 74 00 02 String;xp....t..
68 69 78                                        hix
null,hi,7
```

Tabelle 12.3 Ausgabe zu Test

Es wurde nur das Stream-Flag `SC_SERIALIZABLE` (`0x02`) mit dem Stream-Flag `SC_WRITE_METHOD` (`0x01`) (per AND) zu `0x03` überlagert. Ein zusätzliches `TC_ENDBLOCKDATA` (`0x78`) terminiert die Werte.

3. Variante

Zusatz-Funktionalität: Serialisieren von statischen Feldern

Eine zusätzliche Funktionalität ist eigentlich der Sinn der klasseninternen Implementation der beiden Methoden.

- In diesem Fall wird der Wert eines statischen Felds übertragen, was bei der Default-Serialisierung nicht möglich ist.[10]

```
Thu Dec 07 20:31:34 GMT+01:00 2000,hi,7
```

Tabelle 12.4 Ausgabe zu Test ohne binäre Stream-Darstellung

Die Ausgabe zeigt die Übertragung und erfolgreiche Initialisierung des statischen Felds `d`, das in der zweiten Variante noch `null` war.

10. Die Lösung ist allerdings höchst ineffizient, da in jedem Objekt nun statische Werte übertragen werden (siehe 12.6.1).

12.4.4 Broker-Pattern: Stream-Ersatzobjekte

Serialisierung ist ein zusätzlicher Dienst zu einer Klasse. Es ist nicht unbedingt vorteilhaft, diesen Dienst in der Klasse selbst zu implementieren.

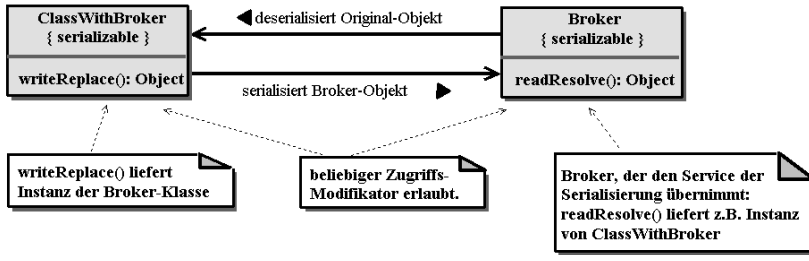
Broker-Pattern: transparentes Service-Objekt

Der nachfolgende Mechanismus basiert auf dem Broker-Pattern.

- Ein **Broker** (Objekt-Makler) ist ein Service-Objekt, das für eine Klasse einen bestimmten Dienst transparent für die Clients abwickelt.

Im konkreten Fall wird der Serialisierungs-Dienst der Klasse nicht von ihr selbst abgewickelt, sondern einem Broker-Objekt überlassen.

- Es wird also kein Objekt der serializable Klasse selbst, sondern ein Broker-Objekt in den Stream geschrieben (Abb. 12.4).



Broker-Muster für die Serialisierung

Abbildung 12.4 Broker übernimmt Serialisierungs-Aufgabe

Vorteile eines Brokers

Die Klasse selbst wird vom Serialisierungs-Prozess entlastet. Die Aufgabe der Übernahme relevanter Objekt-Informationen sowie der Wiederherstellung des Objekts liegt ausschließlich beim Broker-Objekt.

Broker-Vorteil

Ohne die ursprüngliche Klasse ändern zu müssen, können in Broker-Objekten verschiedene Strategien der Übertragung angewendet werden.

Delegation an Broker

Damit der Broker-Mechanismus auch greift, muss die serialisierbare Klasse die folgende Methode (in der Regel transparent als `private`) implementieren:

Broker-Delegation in der Server-Klasse: writeReplace()

```
[public|protected|private] Object writeReplace()
                           throws ObjectStreamException;
```

Wird nun ein Objekt dieser Klasse durch `ObjectOutputStream` serialisiert, ruft dieser die Methode `writeReplace()` auf und schreibt das Resultat als Broker-Objekt in den Byte-Stream.[11]

- Da im Byte-Stream nur das Broker-Objekt enthalten ist, kann die Deserialisierung nur aus der Rekonstruktion des Broker-Objekts bestehen.

Die delegierende Klasse kann zwar auch `Externalizable` implementieren, dies ist aber nicht unbedingt sinnvoll, da sie dann zwei Methoden enthält, die nicht genutzt werden.

Broker-Implementation

Mechanismus in der Broker-Klasse: readResolve()

Implementiert die serialisierbare Broker-Klasse die Methode

```
[public|protected|private] Object readResolve()
                                throws ObjectStreamException;
```

so ruft `ObjectInputStream` beim Deserialisieren die Methode `readResolve()` des Broker-Objekts auf.

Das Resultat dieser Methode ist dann in der Regel das rekonstruierte Objekt des Originals (der delegierenden Klasse).

Beziehung: Klasse vs. Broker

writeReplace() und readResolve() sind unabhängig

Die Methoden `writeReplace()` und `readResolve()` sind unabhängig voneinander, können somit auch einzeln eingesetzt werden.

Die Methode `writeReplace()` kann in einer serialisierbaren Klasse ohne Broker eingesetzt werden, um z.B. nur bestimmte Objekte der Klasse zu deserialisieren.

- Entgegen dem Eindruck der offiziellen Dokumentation kann das Broker-Objekt durchaus mit `readResolve()` ein beliebiges Objekt liefern, nicht unbedingt das der ursprünglichen Klasse.

Kritik

Broker-Lösung basiert nicht auf Interfaces

Der Broker-Mechanismus basiert nicht auf Interfaces, da die beiden Methoden dann nur `public` deklariert sein könnten.

Das aktuelle Interface-Konzept ist also – wieder einmal – nicht flexibel genug, d.h. wird hier durch Reflexion oder (magic) `private`-Zugriffe umgangen.

11. `writeReplace()` kann dazu auch private deklariert sein.

Beispiele

Die triviale Klasse C ist ihr eigener Broker, d.h., es wird nur wieder die Default-Serialisierung nachgebildet:

```
class C implements Serializable {
  private Object writeReplace() throws ObjectStreamException {
    return this;
  }
  private Object readResolve() throws ObjectStreamException {
    return this;
  }
}
```

Zur Klasse Delegator werden mögliche Broker-Varianten vorgestellt:

Delegator-Klasse mit zwei Broker-Varianten

```
class Delegator implements Serializable {
  private int i;
  public Delegator(int i) { this.i= i; }
  public int geti() { return i; }
  private Object writeReplace() throws ObjectStreamException {
    return new Broker(this);
  }
}
```

1. Broker-Variante

Der Broker deserialisiert einfach ein konstantes String-Objekt:

Deserialisierung eines konstanten String-Objekts

```
class Broker implements Serializable {
  public Broker(Delegator d) {}
  private Object readResolve() throws ObjectStreamException {
    return "Delegator?";
  }
}
```

2. Broker-Variante

Der Broker deserialisiert ein Delegator-Objekt im gleichen Zustand:

Deserialisierung eines Delegator-Objekts im gleichen Zustand

```
class Broker implements Serializable {
  private int i;
  public Broker(Delegator d) { i= d.geti()^0xaaaa; }
  private Object readResolve() throws ObjectStreamException {
    return new Delegator(i^0xaaaa);
  }
}
```

12.5 Klassen-Evolution

Klassen-Evolution: Versionswechsel nach Serialisierung

Die Objekt-Kommunikation kann bedingt durch Zeitversatz (Serialisieren in Dateien) oder verschiedene JVMs auf unterschiedliche Klassen-Versionen treffen.

Natürlich trifft dieses Problem gleichermaßen auf serializable und externalizable Klassen zu. Die folgende Diskussion beschränkt sich aber ausschließlich auf Klassen, die `Serializable` implementieren.

SUID

(S)UID: Stream Unique Identifier

Serialisierte Objekte einer anderen Klassen-Version können nicht ohne spezielle Vorkehrungen deserialisiert werden.

Eindeutiger Hashcode der Klassen-Version

- Mit jedem Objekt wird nicht nur die zugehörige Klasse, sondern auch ein eindeutiges Ident – kurz **UID** – vom Typ `long` serialisiert, das nicht nur die Klasse, sondern auch jede Version der Klasse eindeutig identifiziert.

Das UID ist der Hashcode[12], berechnet aus allen relevanten Klassen-Informationen wie Name, Felder, Parameter, Modifier etc. und ändert sich somit mit jeder neuen Version.

Stream-Kompatibilität

- Serializable Klassen mit gleichem Namen, die dieselbe UID haben, werden bei der (De-)Serialisierung als **stream-kompatible** angesehen.[13]

Mit demselben UID wird ausgedrückt, dass die neue Klassen-Version den Client-Kontrakt der alten einhält (siehe Stream-Kompatibilität).

- Für stream-kompatible Versionen einer Klasse `C` kann man das UID manuell durch folgende Anweisung setzen:

```
class C implements Serializable {
  // jede stream-kompatible Version hat dieselbe id
  public static final long serialVersionUID= idL;
  //...
}
```

UID-Anweisung [14]

Der Wert von `id` ist nicht festgelegt, er muss nur identifizierend sein.

12. SHA: Secure Hash Algorithm.
13. Das UID hat keine Bedeutung für externalizable Klassen, da diese ohnehin ihr eigenes Protokoll zur Objekt-Übertragung definieren müssen.
14. `id` steht für den Hashcode, `L` kennzeichnet `long`.

Will man ein UID-konformes Ident vergeben, kann man diesen manuell durch das Utility-Programm `serialver`[15] berechnen lassen oder im Programm mit Hilfe der Klasse `ObjectStreamClass`:

UID-Berechnung

```
class C implements Serializable {

  static public long uid() {
    return ObjectStreamClass.lookup(C.class)
                            .getSerialVersionUID();
  }
  //...
}
```

getSerial-VersionUID()

In der Praxis wird bereits die erste Version einer serializable Klasse, nachdem sie stabil ist, manuell mit einer UID versehen. Denn für jede Klasse, die kein `serialVersionUID` deklariert hat, wird sonst vor dem Streaming-Prozess der Hashcode berechnet.

12.5.1 Stream-Kompatibilität

Der Begriff **stream-kompatibel** soll im Weiteren kurz präzisiert werden. Es gibt hierzu zwei Aspekte, d.h. eine notwendige und eine hinreichende Bedingung.

Verhalten der Default-Serialisierung

Notwendig für eine stream-kompatible Klassen-Evolution ist die Frage, was die Default-Serialisierung beim Deserialisierungs-Prozess an Versionsänderungen toleriert:

Stream-kompatible Klassen

- **Neue Felder**: Sind Felder im Stream, d.h. in der alten Version, nicht enthalten, werden sie mit `0` oder `null` initialisiert (nicht mit ihren Initialisierungswerten!).
- **Fehlende Felder**: Felder im Stream, die es in der neuen Version nicht mehr gibt, werden einfach ignoriert.

Nicht toleriert, d.h. mit einer `InvalidClassException` bestraft, werden dagegen folgende Änderungen:

- **Typ-Wechsel**: Ein Feld mit gleichem Namen wechselt den Typ.
- **Änderung der Serialisierungsart**: Eine Klasse, die im Stream enthalten ist, ändert ihre Serialisierungsart.

15. Siehe JDK

Zusätzlich notwendiger Kontrakt

Eine erfolgreiche Deserialisierung ist zwar notwendig, aber in der Regel zur Einhaltung des Kontrakts bei der Klassen-Evolution nicht hinreichend.

Semantische Fehler durch Regel nicht ausgeschlossen

Vor allem die Default-Werte 0 bzw. null für fehlende Felder, der Wechsel eines Felds von static zu nicht static oder der Klasse innerhalb der Klassen-Hierarchie führen zu unangenehmen semantischen Fehlern.

Beispiel

In einer neuen Version der Klasse Evolve wird ein Feld hinzugefügt bzw. weggelassen:

Klasse: 1. Version

```
class Evolve implements Serializable {
  public static final long serialVersionUID= 1;
  public static final String ss= "V1";
  String s= "hi";                          // fällt weg
  private int i= 1;
}
```

Eine Instanz von Evolve wird in die Datei Version.ser geschrieben:

```
ObjectOutput oout= new ObjectOutputStream(
              new FileOutputStream("C:/temp/Version.ser"));
oout.writeObject(new Evolve());
oout.close();
```

Neue Version der Klasse Evolve:

Klasse-Evolution: ein Feld fehlt, ein Feld hinzugefügt

```
class Evolve implements Serializable
  public static final long serialVersionUID= 1;
  public static final String ss= "V2";
  private int i= 1;
  private float f=1.0f;                    // neues Feld
  public String toString() { return ss+"|"+i+"|"+f; }
}
```

Die alte Instanz wird mit der neuen Version von Evolve deserialisiert:

```
ObjectInput oin= new ObjectInputStream(
                  new FileInputStream("C:/temp/Version.ser"));
System.out.println(((Evolve)oin.readObject())); // 💻: V2|1|0.0
```

Die Ausgabe bestätigt, dass überflüssige Felder im Stream ignoriert werden, und neue Felder wie f ohne Aufruf des Instanz-Initialisierers erzeugt und mit Null initialisiert werden.

12.6 Anpassungen der Object-Streams

Betrachtet man die Signatur von `writeObject()` und `readObject()` der Object-Streams, so akzeptieren diese Argumente vom Typ `Object`.

Da alle einfachen Anpassungen der Serialisierung wie die bisher besprochenen auf dem Default-Mechanismus aufsetzen, wäre eigentlich ein Parameter vom Typ `Serializable` logischer.

Der Grund für diese Entscheidung ist die Flexibilität bei den Subklassen `ObjectOutputStream` bzw. `ObjectInputStream`:

Subklassen von ObjectInput-Stream bzw. ObjectOutput-Stream

- Subklassen der Object-Streams sind dann notwendig, wenn Objekte von Klassen übertragen werden müssen, die nicht serialisierbar sind.

Dies führt zu drei weiteren Serialisierungs-Mechanismen.

12.6.1 annotateClass() und resolveClass() für Klassendaten

Statische Felder werden bereits beim Laden der Klasse angelegt und initialisiert, d.h. vor jeder Objekt-Erzeugung bzw. -Übertragung.

Übertragen von einmaligen Klassen-Informationen

- Allgemein macht eine Übertragung von Klassen-Informationen in jedem Objekt wenig Sinn, wäre redundant und ineffizient.

Der Austausch von Klassen-Informationen sollte nur einmal erfolgen.

Hierzu müssen die Methoden `annotatedClass()` aus `ObjectOutputStream` und `resolveClass()` aus `ObjectInputStream` in den entsprechenden Subklassen dieser Streams überschrieben werden.

Overriding von annotatedClass()

Diese Methode `annotatedClass()` wird ausgeführt, nachdem die Klassen-Beschreibung in den Stream geschrieben wurde, also vor der Übertragung der Objekte.

Mit `resolveClass()` müssen dann umgekehrt die Informationen wieder gelesen werden.

Overriding von resolveClass()

Beispiel

Es soll der Wert des statischen Felds `s` der Klasse `A` übertragen werden.

```
class A implements Serializable {
  static String s;
  char c;
  A(char c) { this.c= c; }
}
```

Hierzu werden Subklassen der Object-Streams angelegt und die o.a. Methoden überschrieben.

Subklasse von ObjectOutputStream mit annotatedClass()

```
class MyObjectOutputStream extends ObjectOutputStream {
  public MyObjectOutputStream(OutputStream out)
                                        throws IOException {
    super(out);
  }
  protected void annotateClass(Class c) throws IOException {
    // überschreibt die leere Methode in ObjectOutputStream
    // schreibt Klassendaten
    if (c==A.class)
      super.writeObject(A.s);
  }
}
class MyObjectInputStream extends ObjectInputStream {
  public MyObjectInputStream(InputStream in) throws
                    IOException, StreamCorruptedException  {
    super(in);
  }
```

Subklasse von ObjectInputStream mit resolveClass()

```
  protected Class resolveClass(ObjectStreamClass v)
              throws IOException, ClassNotFoundException   {
    // resolveClass()in ObjectInputStream holt Klassen-Objekt
    Class c= super.resolveClass(v);
    // hinter der Klassen-Info werden die Daten übertragen
    if (c== A.class)
      A.s= (String) super.readObject();
    return c;
  }
}
```

Vor dem Serialisieren des ersten Objekts der Klasse A werden auch die Klassendaten übertragen. Hierzu zwei kleine Code-Fragemente:

```
ObjectOutput oout= new MyObjectOutputStream(out);
A.s= "Klassendaten";
oout.writeObject(new A('a'));
A.s="";
```

Umgekehrt wird A.s mit dem ersten Deserialisieren wieder gesetzt:

```
ObjectInput oin= new MyObjectInputStream(in);
oin.readObject();
System.out.println(A.s);                    // 💻: Klassendaten
```

12.6.2 replaceObject() und resolveObject()

Steht man vor dem Problem, Objekte serialisieren zu müssen, die weder Serializable noch Externalizable implementiert haben, so kann man diese durch serialisierbare Repräsentanten ersetzen.

Hierzu definiert man Subklassen der Object-Streams, deren Methoden

```
protected Object replaceObject(Object o) throws IOException;
protected Object resolveObject(Object o) throws IOException;
```

man überschreiben muss.

Restriktionen zum Replace/Resolve-Mechanismus

Es sind folgende Restriktionen einzuhalten:

- In den Subklassen von ObjectOutputStream bzw. ObjectInputStream müssen vorher die Methoden enableReplaceObject(true) bzw. enableResolveObject(true) aufgerufen werden.
- Ist ein Sicherheitsmanager installiert, wird die Erlaubnis überprüft[16] und – sofern nicht vorhanden – eine SecurityException ausgelöst.
- Die Methode replaceObject() ersetzt das Originalobjekt durch einen serialisierbaren Repräsentanten.
- Die Methode resolveObject() ersetzt den Repräsentanten wieder durch ein Objekt der ursprünglichen Klasse.

Damit keine heillose Konfusion entstehen kann, gilt zusätzlich:

- Für Objekte der Klassen Class und ObjectStreamClass werden die Methoden replaceObject() bzw. resolveObject() nicht aufgerufen.

Beispiel

Die erste Aufgabe besteht darin, längere Strings komprimiert zu übertragen.

String-Replacement: Komprimierung mittels Zips

Hat ein String unter 100 Zeichen oder kann er nur zu weniger als 50% komprimiert werden, wird er normal übertragen. Ansonsten wird er als komprimierte Instanz von SerString im Stream übertragen.

Die Klasse SerString benutzt hierzu die Klassen Deflater und Inflater des Packages java.util.zip.

Die String-Prüfung und den eventuellen Austausch gegen eine komprimierte SerString-Instanz übernimmt die Factory-Klasse SerString selbst:

16. Die Überprüfung erfolgt mit secManager.checkPermission(SUBSTITUTION_PERMISSION), wobei die Konstante static final SerializablePermission SUBSTITUTION_PERMISSION in ObjectStreamConstants definiert ist.

Eine Replacement-Klasse für String

```
class SerString implements Serializable {
  private byte[] b;
  private int slen;
  // Instanzen können nur durch tryCompress angelegt werden
  private SerString(byte[] cs, int cslen, int len) {
    slen= len;
    b= new byte[cslen]; System.arraycopy(cs,0,b,0,cslen);
  }
  // liefert eine SerString-Instanz oder denselben String
  public static Object tryCompress(String s) {
    if (s.length()>99) {
      byte[] cs;
      int slen, i;
      Deflater d= new Deflater(Deflater.BEST_COMPRESSION);
      try {
        // Byte-Array zum Komprimieren übergeben
        d.setInput(cs= s.getBytes("UTF-8"));
        slen= cs.length;
        cs= new byte[slen/2];  // mindestens 50 % Kompression
        d.finish();            // Ende mitteilen
        i=d.deflate(cs);       // komprimieren
        if (d.finished())      // sofern Array-Größe ausreicht
          return new SerString(cs,i,slen);
      } catch (Exception e) {};
    }
    return s;                  // ansonsten String zurück
  }
 // dekomprimiert wieder zum String
 public String toString() {
    int len;
    byte[] sb= new byte[slen]; // Größe des UTF8-Arrays
    Inflater infl= new Inflater();
    try {
      infl.setInput(b);        // Byte-Array setzen
      len= infl.inflate(sb);   // dekomprimieren
      return new String(sb,0,len,"UTF-8"); // String zurück
    } catch (Exception e) {};
    return null;               // bei Fehler null
  }
}
```

Die zweite Aufgabe besteht darin, mit Hilfe einer Wrapper-Klasse von beliebigen nicht serialisierbaren Objekten zumindest Default-Objekte derselben Klasse deserialisieren zu können.

Wrapper für nicht serialisierbare Objekte

```
class SerWrapper implements Serializable {
  private Class c;
  // jedes nicht serialisierbare Objekt wird
  // durch sein Class-Objekt ersetzt
  public SerWrapper (Object o) { c= o.getClass(); }
  // Konstruktion eines neuen (Default-)Objekts der Klasse
  public Object get() {
    try {
      // setzt No-Arg-Konstruktor voraus!
      return c.newInstance();
    } catch (Exception e) { return null; }
  }
}
```

In der Subklasse von ObjectOutputStream muss die Replace-Operation nun String-Objekte und nicht serialisierbare Objekte abfangen und durch Repräsentanten der beiden o.a. Klassen ersetzen:

Subclassing von ObjectOutput-Stream

```
class MyObjectOutputStream extends ObjectOutputStream {
  public MyObjectOutputStream(OutputStream out)
                    throws IOException, SecurityException {
    // Stream an ObjectOutputStream weiterreichen
    super(out);
    enableReplaceObject(true);          // siehe 1. Regel oben
  }
```

Methode replace-Object()

```
  protected Object replaceObject(Object obj)
                                      throws IOException {
    if (obj instanceof String)
      return SerString.tryCompress((String)obj);
    else if (obj instanceof Serializable)
      return obj;
    else
      return new SerWrapper(obj);
  }
}
```

In der Subklasse von ObjectInputStream führt die Resolve-Operation die zu replace inverse Operation aus:

Subclassing von ObjectInputStream

```
class MyObjectInputStream extends ObjectInputStream {
  public MyObjectInputStream(InputStream in)
                    throws IOException, SecurityException  {
    super(in);
    enableResolveObject(true);
  }
```

Methode resolveObject()

```
  protected Object resolveObject(Object obj) throws IOException {
    if (obj instanceof SerString)
      return ((SerString)obj).toString();
    else if (obj instanceof SerWrapper)
      return ((SerWrapper)obj).get();
    else return obj;
  }
}
```

Zum Test können zwei triviale Klassen verwendet werden.

```
class Any { public String toString() { return "Any"; } }
class SW implements Serializable {
  String s;          // wird auch durch SerString untersucht
  public SW(String s) { this.s= s; }
  public String toString() { return s; }
}
```

Replacement erfolgt rekursiv

Ein Test-Code zur Objekt-Übertragung ist analog zur Default-Serialisierung:

```
public class Test {
 public static void main(String[] args) {
  String s= "dies soll ein langer komprimierbarer String sein...";
  ByteArrayOutputStream baos= new ByteArrayOutputStream();
  try {
    ObjectOutput oout= new MyObjectOutputStream(baos);
    oout.writeObject(new Any()); // wird ersetzt durch SerWrapper
    oout.writeObject(new SW(s)); // String-Feld wird komprimiert
    ObjectInput oin= new MyObjectInputStream(
                  new ByteArrayInputStream(baos.toByteArray()));
      System.out.println(oin.readObject());
      System.out.println(oin.readObject());
    } catch (Exception e) { System.out.println(e); }
  }
}
```

- Interessant ist, dass Replacements auch für Felder rekursiv erfolgen.

12.6.3 writeObjectOverride() und readObjectOverride()

Override-Methoden: komplette Übernahme des Streams/Protokolls

Die letzte hier angesprochene Möglichkeit ist keine Anpassung der Default-Serialisierung, denn es gibt keine vordefinierten Sequenzen. Der Stream ist leer, jedes Byte muss selbst geschrieben werden.

- Die **Override-Methoden** definieren ein komplett neues Protokoll.
- Es gibt nicht wie bei `Externalizable` ein Protokoll, in das man seine Objekt-Informationen einbetten kann.

Diese radikale Übernahme erreicht man durch Überschreiben von[17]

```
protected void writeObjectOverride (Object obj)
                                        throws IOException;
protected Object readObjectOverride() throws
  OptionalDataException,ClassNotFoundException,IOException;
```

Diese beiden Methoden sind in den Object-Streams implementiert, aber ohne Funktion. Für eine eigene Implementierung gelten folgende Restriktionen:

Die Override-Methoden

Restriktionen zu read-/write-ObjectOverride()

- können nur in Subklassen der Object-Streams überschrieben werden, die den No-Arg-Konstruktor der Object-Streams aufrufen.
- können Objekte aller Klassen übertragen, auch solche, die nicht als serialisierbar gekennzeichnet sind.
- müssen das gesamte Übertragungs-Protokoll selbst realisieren.
- haben keinen Zugriff auf `private` deklarierte Felder der übertragenen Objekte.
- werden von einem Sicherheitsmanager – sofern installiert – im No-Arg-Konstruktor geprüft.[18]

Der No-Arg-Konstruktor setzt ein `private` deklariertes Flag, das den Default-Methoden signalisiert, nur ihre zugehörigen Override-Methoden auszuführen.[19]

Da der No-Arg-Konstruktor keinen `Outputstream` bzw. `Inputstream` entgegennimmt, werden keine Felder der Object-Streams gesetzt.

17. Dies ist möglich, denn `writeObject()` und `readObject()` sind final deklariert!
18. Die Prüfung erfolgt mit `sm.checkPermission(SUBCLASS_IMPLEMENTATION_PERMISSION)`
19. Diese Art der Methoden-Delegation ist ein Überschreiben durch die Hintertür mit Sicherheitscheck.

Beispiel

Definition eines eigenen Protokolls

Das Beispiel zeigt nur die prinzipiellen Schritte, da ein eigenes Protokoll verständlicherweise umfangreich und applikationsabhängig ist.

```
class MyObjectOutputStream extends ObjectOutputStream {
  private OutputStream out;

  public MyObjectOutputStream(OutputStream out)
              throws IOException, SecurityException {
    // nur super(); erlaubt, was man sich schenken kann
    // mit out muss man alles selbst in die Hand nehmen
    this.out= out;
  }
```

Methode writeObjectOverride()

```
  protected void writeObjectOverride(Object obj)
                                  throws IOException {
    // ein Trivial-Protokoll als Demo
    String s= obj.getClass().getName();     // Name der Klasse
    out.write(s.length()>>8);               // Länge in 2 Byte
    out.write(s.length());
    out.write(s.getBytes());                // rein ASCII
  }
}
class MyObjectInputStream extends ObjectInputStream {
  private InputStream in;

  public MyObjectInputStream(InputStream in)
               throws IOException, SecurityException {
    this.in= in;
  }
```

Methode readObjectOverride()

```
  protected Object readObjectOverride()
                      throws OptionalDataException,
                ClassNotFoundException, IOException {
    int i= in.read()<<8; i+= in.read();     // Länge lesen
    byte[] b= new byte[i];
    in.read(b);
    // Klasse mit dem gelesenen Namen laden
    Class c= Class.forName(new String(b));
    try {
      return c.newInstance();               // Instanz liefern
    } catch (Exception e) { return null; }; // ansonsten null
  }
}
```

Zum Test wird die bereits im letzten Beispiel verwendete Klasse Any mit Default-Konstruktor benötigt.

```
class Any {
  public String toString() { return "Any"; }
}
```

Der Stream mit eigenem Trivial-Protokoll wird mit Sniffer dargestellt.

Test der Override-Methoden

```
public class Test {
  public static void main(String[] args) {
    byte[] barr= null;
    ByteArrayOutputStream baos= new ByteArrayOutputStream();
    try {
      ObjectOutput oout= new MyObjectOutputStream(baos);
      oout.writeObject(new Any());
      barr= baos.toByteArray();
      System.out.println(Sniffer.toHexAsciiString(barr,16));
      ObjectInput oin= new MyObjectInputStream(
                        new ByteArrayInputStream(
                                      baos.toByteArray()));
      System.out.println(oin.readObject());
    } catch (Exception e) { System.out.println(e); }
  }
}
```

```
00 09 6b 61 70 31 32 2e 41 6e 79                ..kap12.Any
Any
```

Tabelle 12.5 Ausgabe zu Test

Die Realisierung eines eigenen Protokolls ist – wie zu sehen – aufwändig und lohnt nur bei speziellen Anwendungen, wie z.B. für die effiziente Übertragung von großen Arrays einer Klasse.

Eigenes Protokoll? Effizienz, Hardware ...

12.7 Zusammenfassung

Die Idee der Serialisierung ist faszinierend, da sie das Marshaling bzw. Unmarshaling von beliebigen Objekten – in der Regel typsicher – erlaubt.

Mit den Interfaces `Serializable` und `Externalizable` werden die grundlegenden Mechanismen und Protokolle vorgestellt und anhand einer `Sniffer`-Klasse demonstriert.

`Externalizable` ist das Mittel der Wahl, um Felder und Felddaten von Objekten beliebig manipulieren zu können. Abgesehen von allgemeinen Header- und Klassen-Informationen, ist das Stream-Protokoll von `Externalizable` frei bestimmbar.

Neben eigener Traversierung des Objekt-Graphen ist ein gravierender Nachteil von `Externalizable`, dass die beiden Methoden `public` zu implementieren sind.

Die Default-Serialisierung durch Implementation von `Serializable` bietet automatisches Traversieren und Übertragen aller erreichbaren Objekte inklusive der Superklassen-Felder eines Objekts. Es sind jedoch gewisse Regeln zu beachten, die vorgestellt werden.

Beginnend mit sehr einfachen Anpassungen wie `transient` deklarierten Feldern kann die Default-Serialisierung sukzessive durch eigene Funktionalität ergänzt oder ersetzt werden. Hierzu stehen je nach Anforderung sehr verschiedene Mechanismen bereit, die im Einzelnen besprochen und anhand von Beispielen demonstriert werden.

Sehr interessant ist das Broker-Konzept, abgeleitet vom Broker-Pattern, das es erlaubt, die Serialisierung als eigenen unabhängigen Dienst aus der Klasse auszulagern. Diese Separation birgt einige Vorteile.

Da die Default-Serialisierung nur Objekte überträgt, stehen insbesondere für statische Daten zwei zusätzliche Methoden bereit.

Für tief gehende Anpassungen müssen die Object-Streams selbst abgeleitet werden. Die Subklassen können dann so angepasst werden, dass sie sogar Objekte übertragen können, die ansonsten nicht serialisierbar sind.

In letzter Konsequenz kann ein komplett eigenes Protokoll realisiert werden.

Das »Serialisierungs-Gebäude« ist in kurzer Zeit stark erweitert worden, was sich leider im Code der Packages und in teilweise konkurrierenden Konzepten widerspiegelt.

Die Mechanismen laufen teilweise »magic« ab, teilweise beachten sie strikt die normalen Java-Sprachregeln.

12.8 Testfragen[20]

Zu jeder Frage können jeweils eine oder mehrere Antworten bzw. Aussagen richtig sein.

1. Welche Aussagen sind richtig?

A: `Serializable` und `Externalizable` sind Marker-Interfaces.

B: Subklassen einer externalizable Klasse können `Serializable` implementieren, damit ihre Objekte wieder per Default-Serialisierung übertragen werden.

C: Bei einer externalizable Klasse müssen alle Felder, die übertragen werden sollen, explizit codiert werden.

D: Bei einer externalizable (Sub-)Klasse müssen alle Felder von einer Superklasse, die übertragen werden sollen, explizit codiert werden.

E: Bei der Default-Serialisierung werden keine Daten statischer Felder übertragen.

F: `transient` deklarierte Felder können auch von externalizable Klassen nicht übertragen werden.

G: Mit gleichen Werten des `serialVersionUID` werden verschiedene Versionen von Klassen als stream-kompatibel erklärt.

H: Nur wenn der Wert des `serialVersionUID` in verschiedene Klassen-Versionen gleich ist, sind sie stream-kompatibel.

I: Neue Klassen-Versionen sind nur dann stream-kompatibel zur alten, wenn sie alle Instanz-Felder der alten Version auch enthalten.

J: Die Default-Serialierung überträgt mit einer Instanz automatisch nur die Objekte, die von einem Instanz-Feld referenziert werden und serialisierbar sind.

K: Mit den Object-Streams können nur Objekte, aber keine einzelnen Daten von primitiven Typen (außerhalb von Objekten) übertragen werden.

L: Mit den Object-Streams können einzelne Daten von primitiven Typen typsicher übertragen werden, d.h., beim Deserialisieren wird eine Ausnahme ausgelöst, wenn der falsche primitive Typ gewählt wird.

20. Es werden nur Testfragen zum Basiswissen gestellt, die für die Zertifizierung relevant sind.

2. Welche Aussagen sind zu folgenden Klassen richtig?

```
class Int { int i= 1; }

class S implements Serializable {
  private transient Int io;                    ①
  //private Int io;                            ②
  public S(int i) { io= new Int(); io.i= i; }
}
```

A: Objekte der Klasse S können mit Hilfe der Default-Serialisierung übertragen werden.

B: Das Feld `io` eines deserialisierten Objekts von S hat immer den Wert `null`.

C: Beim Deserialisieren wird das Feld `io` eines Objekts von S erzeugt und der Wert von `io.i` auf `1` gesetzt.

D: Wird Zeile ① durch Zeile ② ersetzt, können Objekte der Klasse S nicht mehr mit Hilfe der Default-Serialisierung übertragen werden.

3. Welche Aussagen sind zu folgenden Klassen richtig?

```
abstract class Ex implements Externalizable {
  int i;
}

class ExSub extends Ex implements Serializable {          ①
  String s;
}
```

A: Die Klasse Ex wird fehlerfrei kompiliert.

B: Die Klasse ExSub wird fehlerfrei kompiliert.

C: Die Klasse `ExSub` muss die beiden Methoden `readExternal()` und `writeExternal()` implementieren.

D: Die Angabe `implements Serializable` in Zeile ① ist überflüssig.

4. Welche Aussagen sind zu folgender Applikation richtig?

```
class SInt implements Serializable {
  int i= 1;
}
class S implements Serializable {
  private SInt i1;
  private transient int i2;
  public S(int i) {
    i1= new SInt();
    i1.i= i; this.i2= i;
  }
  public String toString() { return i1.i + ","+i2; }
}
public class Test {
  public static void main(String[] args) {
    try {
      ObjectOutput oout= new ObjectOutputStream(
                          new FileOutputStream("test.dat"));
      oout.writeObject(new S(10));
      oout.close();
      ObjectInput oin= new ObjectInputStream(
                         new FileInputStream("test.dat"));
      System.out.println(oin.readObject());                    ①
    } catch (Exception e) { System.out.println(e); }
  }
}
```

A: Die Ausgabe ist: `10,10`

B: Die Ausgabe ist: `10,0`

C: Die Ausgabe ist: `10,1`

D: Die Ausführung von `Test` führt zu einer Ausnahme in Zeile ①.

5. Welche Aussagen sind zu folgender Applikation richtig?

```
class Base {
  String s1= "Base";
}
class SerD extends Base implements Serializable {
  String s2= "SerD";
  Base b;
  public SerD() { b= new Base(); }                    ①
//public SerD() {}                                    ②
  public String toString() { return s1+s2+b; }
}
public class Testfragen {
  public static void main(String[] args) {
    try {
      ObjectOutput oout= new ObjectOutputStream(
                           new FileOutputStream("test.dat"));
      oout.writeObject(new SerD());
      oout.close();
      ObjectInput oin= new ObjectInputStream(
                         new FileInputStream("test.dat"));
      System.out.println(oin.readObject());
    } catch (Exception e) { System.out.println(e); }
  }
}
```

A: Die Ausführung von `Test` führt zu einer Ausnahme.

B: Die Ausgabe ist: `BaseSerDnull`

C: Wird Zeile ① weggelassen, wird `Test` fehlerfrei ausgeführt.

D: Wird Zeile ① durch Zeile ② ersetzt, ist die Ausgabe: `BaseSerDnull`