Programmierung (V 2.3)
Java
SS 2018
B-AI Version

Bernd Kahlbrandt

19. Juni 2018

Vorwort

Wenn ich eine neue Vorlesung halten will oder auch andere, mir neue Dinge erarbeiten will, so hat es sich für mich bewährt aufzuschreiben, was ich gelernt habe. So habe ich im Februar 2009 dieses Dokument begonnen. Es spiegelt meinen ersten ernsthaften Versuch wider, Programmierung systematisch zu lernen. Was einmal daraus werden wird, ist offen. Sicher ist, dass ich viel gelernt habe. Auch das Zusammenspiel von Programmierung (PR, nun PM und PT) und Algorithmen und Datenstrukturen (AD) hat sich für meinen Lernprozess bewährt. Für das Softwareengineering (SE) ist meine Beispielsammlung gewachsen. Eigentlich lehre ich in allen Veranstaltungen (PR, DB, SE, AD) Grundprinzipien der Softwareentwicklung auf verschiedenen Niveaus.

Vor vielen Jahren las ich im ersten Kapitel des Buchs [OK99] meines Kollegen Bernd Owsnicki:

> „Außerdem muß man schon sehr gut programmieren können, damit im Endprodukt der Kern des Algorithmus noch zu erkennen ist."

Warum versuchen wir eigentlich nicht dies den Studierenden nicht beizubringen? In dieser Veranstaltung versuche ich, einen Beitrag dazu zu leisten.

Gemäß der aktuellen Modulbeschreibungen B-AI, B-WI und B-TI PR ($\hat{=}$ PT + PM1) und PM2 werden die folgenden Themen behandelt:

- Elementare Programmiertechniken
  - primitive Typen,
  - Unicode,
  - Arrays,
  - Referenztypen,
  - Sequenz,
  - Selektion,
  - Iteration,
  - Rekursion
- Abstraktionsmechanismen
  - Funktionale Abstraktion
  - Datenabstraktion (ADT)
  - Kontrollabstraktion (z.B. Iteratoren, Streams)
- Objektorientierung (prozedural und funktional)
  - Polymorphie, Overriding,
  - Overloading
  - late binding
- Ausgewählte Elemente objektorientierter Bibliotheken, z. B. :

  - Collections
  - Streams,
  - Channels (aus java.nio)
- Typisierungskonzepte
  - Dynamische vs. statische Typisierung
- Ausgewählte Elemente objektorientierter Bibliotheken, z. B.
  - GUI-Frameworks
  - Generics
- Metasprachliche Konzepte
  - Annotationen,
  - XML
  - Reflection
- Vertiefungen
  - Typ- vs. Implementationshierarchie
  - elementare Entwurfsmuster
  - Modellierungen (anhand UML)
  - nebenläufige bzw. asynchrone Programmierung
  - Deployment
- Correctness
  - Design by Contract (Assertions, Invarianten)
  - Teststrategien

Lernen sollen Sie dieses:

- Beherrschung von handwerklichen Programmierfertigkeiten und elementaren Programmiertechniken.
- Kennenlernen unterschiedlicher Konzepte und Modelle von Programmiersprachen (Programmierparadigmen). Dabei wird zunächst das Paradigma *Objektorientierung* wichtig sein. Andere Paradigmen folgen an passenden Stellen im Verlauf der Veranstaltung.
- Objektorientierte Modellierung und technische Realisierung von Systemen im Kleinen durchführen können.
- Beherrschung von fortgeschrittenen Fertigkeiten und Programmiertechniken.
- Objektorientierte Modellierung und technischen Realisierung von Systemen, die nur teamorientiert erarbeitet werden können, beherrschen.

Selbstverständlich gibt es in den Studiengängen unterschiedliche Schwerpunkte. Insofern werden nicht alle Themen in jedem der Studiengänge behandelt werden.

Die Arbeit an diesem Skript begann im SS 2009 und WS 2009/10 parallel zu den Vorlesungen Programmierung 1 und 2 im Bachelor-Studiengang Technische Informatik am Department Informatik der Fakultät TI an der HAW. Eine erste Version α für Studierende wurde im SS 2010 veröffentlicht. Diese wurde sukzessive über die beiden Semester weiterentwickelt. Zum Sommersemester 2011 erschien eine erste β-Version.

Für das WS 2012 habe ich etwas ergänzt. Im SS 2012 verwende ich das Skript für die Programmiervorlesung im Studiengang Wirtschaftsinformatik. Dazu erscheint es in Etappen. Deshalb habe ich die Reihenfolge der Kapitel so verändert, dass alle Teile am Anfang stehen, die Martin Hübner im ersten Teil der Vorlesung behandelt hat.

Für das WS 2012/13 und das SS 2013 entstand eine in Teilen grundsätzlich überarbeitete Version. So nehme ich in Kap. 2 mit Ihnen eine ganz schlanke „Tiefbohrung“ in die objekt-orientierte Programmierung mit Java vor. Auskommentiert für Rainers Shanghai-Version. Für das Sommersemester 2017 habe ich für die Vorlesung im Studiengang B-AI ein Kapitel aufgenommen, in dem eine Klasse *Rational* nach dem Muster der entsprechenden Ruby-Klasse entwickelt wird, siehe Kap. 3. Im Index verweisen **fett** gesetzte Seitenzahlen auf die Definition des Begriffs und *kursiv* gesetzte Seitenzahlen auf Einträge im Literaturverzeichnis.

Danke

In den Semestern ab SS 2011 konnte ich zum ersten Mal direkte Beiträge von Studierenden zu diesem Skript erwähnen. Ich kann nicht — analog Don Knuth — für die Entdeckung eines Fehlers oder einen konstruktiven Beitrag einem Hex-Euro (2,56) zahlen. Ich bemühe mich aber um eine Würdigung der Beiträge meiner Veranstaltungsteilnehmerinnen und -teilnehmer. Die Tabelle mit dem Vergleich von *compareTo* und *equals* auf S. 58 habe ich als Antwort auf eine Frage von Tugba Karakaya im SS 2011 aufgestellt. Gute Fragen und Beiträge kamen (ohne zeitliche, quantitative oder qualitative Einstufung waren alle hilfreich) von Jeremias Twele, Ihmed Bahrini, Thomas Broja, Clemens Drauschke, Ibrahim Genc, Lars Harmsen, Tugba Karakaya, Vitali Kagadij, Ilyuza Mingazova, Anton Starobinski, Ali Zardosht, . . . Jan-Tristan Rudat hat im WS 2011/12 große Teile Korrektur gelesen.

Im SS 2012 gab es konstruktive Anmerkungen von Charlotte Christophers, Olf Deussen, Michael Erhard und Ramin Mohibi u. a.

Aus dem WS 2012/13 erinnere ich viele Fragen und Beiträge u. a. von Kurt Laabs, Christopher Masch, Henning Krohn, Natalie Gläser.

Im SS 2013 hat Florian Arfert das Kap. 18 korrekturgelesen und mich auf Fehler und Verbesserungsmöglichkeiten hingewiesen.

Zur Auflage WS 2014/15, SS 2015

In dieser Auflage halte ich dieses Skript für soweit gediehen, dass ich es als Version 1.5 ohne Zusatz β veröffentliche. Jetzt ist die aktuelle und hier zu Grunde gelegte Version Java 8. Es ist nicht möglich in der verfügbaren Zeit auch nur die grundlegenden Neuerungen vollständig zu besprechen. Insofern werden z. B. Lambda-Ausdrücke behandelt aber auf eine systematische Einführung in funktionale Programmierung wird verzichtet.

Für die überraschend hinzugekommene Vorlesung Programmieren 3 im Studiengang Electrical Engineering am Shanghai Hamburg College habe ich viele Teile aktualisiert (V 1.6). Während des Semesters in Shanghai entstand dann die Version 1.7 mit vielen weiteren Verbesserungen:

- Ergänzungen im Kapitel 20.
- RMI . . .

Sommersemester 2016

Im Sommersemester 2016 halte ich diese Vorlesung zweimal: In den Kalenderwochen 8–15 als Programmieren 3 im Studiengang EE am SHC in Shanghai und von der 17.–25. KW im Studiengang Angewandte Informatik des Departments Informatik an der HAW. Daher gibt es zwei Versionen, die die Vorkenntnisse der Studierenden und die Anforderungen der Studiengänge berücksichtigen. So gibt es in der einen Version eine Übersicht der Unterschiede und Gemeinsamkeiten von C bzw.

C++ und Java und in der anderen Version Entsprechendes zu Ruby und Java. Die tabellarische Übersicht zu den Unterschieden und Gemeinsamkeiten habe ich erheblich überarbeitet. Es gibt aber immer noch Verbesserungsbedarf!

Beiden Versionen gemeinsam ist aber, das die Teile zur funktionalen Programmierung erstmals ausgearbeitet wurden. Hierbei konnte ich auch auf Beiträge einer Veranstaltung zu den Neuerungen in Java 8 aufbauen, die ich im Wintersemester 2014 durchgeführt habe. Außerdem gibt es eine große Zahl kleiner Verbesserungen, mehr Aufgaben und es gibt wieder ein Glossar. Letzteres ist hier in einer reduzierten Version enthalten, die im Laufe des Semesters ergänzten Versionen habe ich separat veröffentlicht.

Sommersemester 2017

Für diese Auflage habe ich vieles vom Input der TeilnehmerInnen aus dem Sommersemester 2016 aufgenommen, einiges umgestellt und den Teil über funktionale Programmierung erweitert. Außerdem gibt es jetzt je eine Übersicht „Von C, C++ zu Java“ und zu „Von Ruby zu Java“.

Sommersemester 2018

In diesem Semester gibt es zwei Versionen: Eine für Rainer Sawatzki für die Veranstaltung an der USST in Shanghai und eine für B-AI2 PM2. Soweit wie möglich habe ich Java 9 berücksichtigt. Insbesondere die Kap. 11 (Datum und Uhrzeit), Kap. 17 (Streaming API) habe ich deutlich überarbeitet. Außerdem gibt es erste Versionen der Kapitel zu den Neuerungen in Java 9.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einführung

If you want to make an apple pie from scratch
you must first invent the universe.
Carl Sagan [Sag02]

1.1 Übersicht

Ein aktuelles Paradigma der Programmierung ist Objektorientierung. Die Grundidee ist ganz einfach.

> Es gibt *Objekte*, wie Personen, Tische, Stühle Biergläser. Ein *Objekt* gehört zu einer *Klasse*. Eine Klasse definiert die Struktur und das Verhalten ihrer Objekte. Das Verhalten wird durch *Methoden* beschrieben. Jedes System besteht aus Objekten, die zusammenwirken, um die Leistungen des Systems zu erbringen. Um etwas zu bewirken, muss man ein Objekt einer Klasse erzeugen, von dem man dann eine Leistung über den Aufruf einer Methode abrufen kann. Hat man bereits eine passende Klasse, prima, dann erzeugt man sich einfach ein Objekt dieser Klasse. Gibt es keine geeignete Klasse, so schreibt man sich eine und erzeugt ein Objekt dieser neuen Klasse.

Dieses einfache Grundprinzip ist immer da, leider scheint es manchmal hinter technischen Details der Programmierung zu verschwinden. Dies gilt um so mehr, wenn komplexe Werkzeuge eingesetzt werden. Diese müssen und sollen auch erst einmal verstanden werden. Außerdem fällt es Anfängern zunächst oft schwer, sich in den vielen Klassen zurecht zu finden, die zur Verfügung stehen.

Es gibt eine Fülle von Materialien, nach denen Java gelernt bzw. mit denen das Lernen von Java unterstützt werden kann. Ohne Anspruch auf Vollständigkeit seien hier [Mös05, HMHG07, Krü07, Ull14, Ess08, Pan08, RSSW06a, RSSW06b, Job06] bzw. die jeweils neueste Auflage genannt. Ich gebe selten Buchempfehlungen, aber die Bücher von Joshua Bloch [BG05, Blo18] empfehle ich ausdrücklich.

Darüber hinaus gibt es eine Fülle von Online-Quellen.

Ganz besonders weise ich aber auf folgende Quellen hin:

1. Die jeweils aktuelle Java API Dokumentation.
2. Den jeweils aktuellen Java Sourcecode (siehe Abschn. B.3)[1].
3. Die jeweils aktuelle Java Sprachspezifikation: [GJS+17](Java 9), [GJS+14](Java 8).

Und nun noch eine Warnung vor einigen didaktischen Schwierigkeiten, für die soweit ich weiß noch niemand eine perfekte Lösung gefunden hat: Man kann versuchen in das Programmieren so einzuführen, dass jedes neu eingeführte Element als solches verständlich ist und angewendet werden

[1] Life would be so much easier if we could just look at the source code.

kann. Dann dauert es aber sehr lange, bis Sie wirklich ein nützliches Ergebnis sehen. Selbst dann wird aber immer jemand mitdenken und nach Konsequenzen fragen, die eintreten, wenn man den zunächst eingeschränkten Bereich verlässt. Spätestens dann kommt man zu Verweisen auf späteren Stoff und solche „forward references“ sollen vermieden werden. Insofern stellt jede Anordnung des Stoffes einen Kompromiss dar. Ich habe aber versucht bei der Einführung von Objektorientierung und ihrer Anwendung wenig Kompromisse zu machen, vielleicht sogar keine.

Erscheint etwas beim Lesen — sequentiell oder sprunghaft — miss- oder gar unverständlich, so mag der ausführliche Index helfen. Ansonsten gibt es immer noch die Möglichkeit, mir Fragen per E-Mail zu stellen.

1.2 Lernziele

Am Ende dieses Kapitel sollen Sie:

- Die Begriffe Objekt und Klasse erläutern können.
- Den Begriff Objektidentität verstanden haben.
- Die Begriffe Operation, Methode und Schnittstelle erläutern können
- Die Begriffe Assoziation, Delegation, Vererbung erläutern können.
- Die Symbole für Klasse, Assozation und Vererbung (Generalisierung/Spezialisierung) kennen.

1.3 Objektorientierung

Ein Objekt ist irgend etwas, mit dem man etwas machen kann. Dabei kann es sich um einen ganz konkreten Gegenstand der Erfahrungswelt handeln, wie ein Auto, Fahrrad, Stuhl, Tisch, Bierdeckel oder ein Weinglas; etwas weniger Konkretes wie einen Auftrag, eine Buchung, eine Währung, einen Leasingvertrag aber auch um etwas Abstraktes, wie ein Satz im Sinne einer mathematische Aussage, z. B. Satz des Pythagoras, ein Symbol etc. Die Eigenschaften eines Objektes, die für alles Folgende benötigt werden, sind hier zusammengefasst.

Definition 1.3.1 (Objektidentität)
Ein Objekt hat eine von seinen sonstigen Eigenschaften unabhängige Identität. Diese Eigenschaft nennt man *Objektidentität.* ◄

Beispiel 1.3.2 (Auto)
Das Objekt sei mein früherer Firmenwagen mit dem amtlichen Kennzeichen DA-JZ 261, den ich nach Auslaufen des Leasingvertrages erwarb. Gemäß den deutschen Vorschriften musste ich den Wagen dann an meinen Wohnsitz ummelden, wo er das Kennzeichen HH-DR 1134 erhielt. Dadurch hat sich das Auto aber nicht verändert, es ist weiterhin dasselbe Auto, auch wenn es jetzt vielleicht noch mit einem ganz anderen Kennzeichen in Kuweit fährt. Das Prinzip der Objektidentität besagt, dass dieses Objekt trotzdem noch genau wie vorher identifiziert werden kann. ◄

Bemerkung 1.3.3 (Objektidentität)
Die Eigenschaft der Objektidentität entspricht dem, was in der deutschen Sprache mit dem Wort „dasselbe“ bezeichnet wird. Zwei verschiedene Objekte mit gleichen Werten würden „das gleiche“ darstellen, wären aber nicht identisch, also nicht „dasselbe“. In der *Sendung mit der Maus* haben Armin und Christoph das einmal so erläutert: Sie können sehr wohl „das gleiche“ Hemd tragen: Sie haben eben beide ein Hemd an, das den gleichen Schnitt, Farbe, Muster etc. hat. Sie können aber nicht „dasselbe“ Hemd tragen, also nicht gemeinsam ein einzelnes Hemd anziehen. ◄

Bemerkung 1.3.4 (Objektidentität)
Wie Objektidentität überprüft wird, hängt von der jeweiligen Programmiersprache ab: Hier einige Beispiele:

Java Hier leistet dies der Operator ==.

Ruby Hier leistet dies die Methode *equal?*.

Achten Sie bitte auf die Unterschiede in verschiedenen Programmiersprachen!
◄

Objekte haben weitere wichtige Eigenschaften:

Definition 1.3.5 (Attribut)
Ein *Attribut* b eines *Objekt*s A ist ein Objekt, das Bestandteil von A ist. In Java wird ein Attribut als Feld (field) bezeichnet. ◄

Definition 1.3.5 ermöglicht eine hierarchische Konstruktion zunehmend komplexerer Objekte.

Bemerkung 1.3.6 (Bestandteil)
Was die Formulierung „ist Bestandteil“ in Def. 1.3.5 konkret heißt, ist eine technische Einzelheit, die hier noch nicht interessiert. ◄

Eine weitere Eigenschaft von Objekten ist, dass sie ein bestimmtes Verhalten zeigen.

Definition 1.3.7 (Operation und Methode)
Eine *Operation* ist eine Aktivität oder Aktion, die ein Objekt bei Erhalt einer *Nachricht* ausführt. Eine *Methode* ist die Implementierung einer Operation ([BRJ98],[RJB99]). In Java wird beides oft als Methode bezeichnet.

Eine Operation kann keinen, einen oder viele Parameter haben. Mit Parametern können Informationen an eine Operation übergeben werden. ◄

Bemerkung 1.3.8 (Operation und Methode)
Für Operationen gibt es viele Bezeichnungen. Häufig werden Operationen als Methoden bezeichnet. Ich halte mich an die verbreitete Konvention aus Def. 1.3.7. Ich verwende den Begriff *Methode*, wenn die Implementierung einer Operation gemeint ist und der Unterschied wichtig ist. Wird die Sprache durch diese Unterscheidung zu holprig, verwende ich beim Sprechen ganz leger den einen oder den anderen Begriff. Es scheint sich aber inzwischen ein Sprachgebrauch herausgebildet zu haben, der nur noch den Begriff *Methode* verwendet, und ggf. in abstrakte und konkrete Methoden unterscheidet. Ich stelle Zug um Zug auf diese Begriffsbildung um. ◄

Definition 1.3.9 (Nachricht)
Eine *Nachricht* (bzw. das Senden einer *Nachricht* an ein Objekt) ist die Aufforderung an ein Objekt, die entsprechende *Methode* auszuführen, der das Objekt nachkommen muss. ◄

Methoden definieren also, wie Objekte auf Nachrichten reagieren. Das Senden einer Nachricht an ein Objekt ist eine Aufforderung an das Objekt, die entsprechende Methode auszuführen, die das Objekt vertragsgemäß so zu erfüllen hat, wie es für die Methode spezifiziert ist. Nachrichten sind der Verständigungsmechanismus zwischen Objekten. Objekte kooperieren, indem sie über Nachrichten kommunizieren. Objekte haben also sowohl Struktur als auch Verhalten. Beide Aspekte werden einheitlich behandelt. Beides wird soweit sinnvoll und möglich gekapselt.

Definition 1.3.10 (Zustand)
Ein Zustand eines Objekt ist eine Ausprägung von Eigenschaften des Objekts die über einen für den Kontext relevanten Zeitraum erhalten bleibt. ◄

Beispiel 1.3.11 (Zustand)
Ein Zustand gemäß Def. 1.3.10 kann z. B. so definiert sein:

- Ein Attribut *plz* (Postleitzahl) kann für eine Adresse einen Zustand definieren.
- Für ein Girokonto macht es nicht unbedingt Sinn, jeden Kontostand als einen eigenen Zustand anzusehen. Hier können Intervalle sinnvoll sein:
  - Kontostand ≤ 0: „im Haben“
  - Dispolimit $\leq$ Kontostand < 0: „im Soll“, noch im Rahmen des Dispolimits
  - Geduldete Überziehung $\leq$ Kontostand $<$ Dispolimit „im Soll“, noch im Rahmen der geduldeten Überziehung.
- Ein Zustand kann auch dadurch definiert sein, dass eine Methode ausgeführt wird.

◀

Um sich in der Welt zurechtzufinden, abstrahiert der Mensch und fasst gleichartige Objekte zu Klassen zusammen.

Definition 1.3.12 (Klasse)
Eine *Klasse* ist eine Zusammenfassung von gleichartigen Objekten. ◀

Eine Klasse hat innerhalb dieses Kontextes drei wesentliche Eigenschaften, die hier für eine genauere Definition zusammengefasst sind.

Definition 1.3.13 (Präzisierung des Klassenbegriffs)
Der in Def. 1.3.12 eingeführte Begriff der *Klasse* hat drei Aspekte:

1. Eine Klasse ist eine Zusammenfassung von gleichartigen Objekten. In diesem Sinn ist eine Klasse eine Menge aller dieser Objekte.
2. Eine Klasse beschreibt die Eigenschaften aller ihrer Objekte. In diesem Sinne ist eine Klasse ein Metaobjekt, d. h. ein Objekt, das andere Objekte beschreibt.
3. Eine Klasse ermöglicht das Erzeugen von Objekten („Objektfabrik“, Schablone, Template).

◀

Um Klassen und die Zusammenhänge zwischen ihnen übersichtlich darzustellen, werden oft Diagramme verwendet, z. B. sogenannte Klassendiagramme.

Das Symbol für eine Klasse ist ein Rechteck mit drei Abschnitten:

1. Dem Namen der Klasse, ggf. durch weitere Informationen ergänzt.
2. Einem Abschnitt, der die Attribute in normaler Schrifttype zeigt.
3. Einem Abschnitt, der Operationen in normaler Schrifttype zeigt.

Es ist üblich, den Klassennamen oben im Klassensymbol zentriert in fetter Schrift zu setzten. Die Angabe des Namens der Klasse ist obligatorisch. Innerhalb des Klassensymbols können noch weitere Informationen dargestellt werden. Dazu mehr, wenn wir soweit sind, dass wir so etwas brauchen.

Eine Klasse beschreibt also Objekte. Genaugenommen müsste also unterschieden werden:

1. Attribut*wert* für das Objekt, das nach Def. 1.3.5 Bestandteil eines anderen Objekt ist.
2. Attribut für die Beschreibung des Attributs in einer Klasse, also des Attribut*typs*, den alle Objekte der Klasse besitzen.

Diese Unterscheidung macht aber viele Formulierungen unnötig kompliziert und deshalb verzichte ich meistens darauf.

Ein Attribut hat einen *Typ*. Zunächst genügt es völlig, wenn Sie für *Typ* eine *Klasse* einsetzen. Sie werden später lernen, dass es noch andere Arten von Typen gibt. Auch die in der folgenden Definition eingeführte Schnittstelle definiert einen Typ.

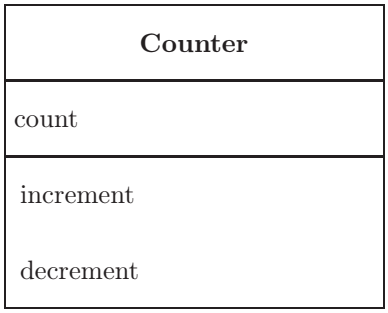


Abb. 1.1: Klassensymbol (einfach)

Definition 1.3.14 (Schnittstelle)
Eine Schnittstelle ist eine Zusammenfassung von Methoden. ◄

Definition 1.3.15 (Signatur)
Die Signatur einer Methode besteht aus ihrem Namen, ihren Parametern und ihrem Rückgabetyp. ◄

Bemerkung 1.3.16 (Rückgabetyp in Signatur)
Sie werden später sehen, dass in der Signatur der Rückgabetyp eine etwas andere Rolle spielt, als der Name und die Parameter. In Java gibt es einen Rückgabetyp *void*. Eine solche Methode gibt nichts zurück. In Ruby gibt jede Methode etwas zurück. ◄

Eine Klasse kann keine, eine oder viele Schnittstellen implementieren. Implementiert eine Klasse eine Schnittstelle, so muss sie für jede nicht implementierte Methode (Operation) der Schnittstelle eine Methode bereitstellen, die die jeweilige Operation implementiert. Eine Schnittstelle (interface) kann in Java aber bereits eine default-Implementierung für Methoden haben.

Eine erste Abstraktionsebene wurde bereits angesprochen, als von einzelnen Objekten abstrahiert wurde und Klassen betrachtet wurden. Dieser Abstraktionsprozess lässt sich unter Umständen weitertreiben. Der passionierte Ornithologe wird sich z. B. freuen, in Deutschland einen Schwarm Seidenschwänze (Bombycilla garrulus) zu sehen. Er wüsste sicher auch, wie diese in die Systematik des Tierreichs einzuordnen sind. Hier genügt es festzuhalten, dass Seidenschwänze Vögel sind, die wiederum Wirbeltiere sind. Es handelt sich hier um eine „Ist-ein“-Hierarchie (englisch: „Is-a“): Jeder Seidenschwanz ist ein Vogel, jeder Vogel ist ein Wirbeltier. Die Spezialisierungen „erben“ alle Eigenschaften der allgemeineren Klassen.

In Java kann eine Klasse für Vögel, hier *Bird* genannt, ganz einfach geschrieben werden:

```
public class Bird {
    private String scientificName;
    private String imageSource;

    public Bird(String scientificName) {
        this(scientificName, "");
    }

     public Bird(String scientificName, String imageSource){
        this.scientificName = scientificName;
        this.setImageSource(imageSource);
    }
    public String getScientificName() {
```

```
        return scientificName;
    }

    public void setScientificName(String scientificName) {
        this.scientificName = scientificName;
    }


    public String getImageSource() {
        return imageSource;
    }

    public void setImageSource(String imageSource) {
        this.imageSource = imageSource;
    }

}
```

Mittels einer einfachen Hilfsmethode können Sie damit sogar schon etwas tun: Der folgende Code zeigt einfach zwei Bilder der entsprechenden Vögel an:

```
public class ShowBird {
    public static void main(String [] args){
        Bird bohemianWaxwing  = new Bird("Bombycilla garrulus",
                                         "./images/seidenschwanz.jpg");
        ShowInFrame.show(bohemianWaxwing.getScientificName(),
         new JLabel(new ImageIcon(bohemianWaxwing.getImageSource())));
        Bird wren = new Bird("Troglodytes troglodytes",
         "./images/Eurasian-Wren-Troglodytes-troglodytes.jpg");
        ShowInFrame.show(wren.getScientificName(),
         new JLabel(new ImageIcon(wren.getImageSource())));
    }
}
```

Dabei verwende ich eine Hilfsklasse „ShowInFrame“. Das können Sie jetzt noch nicht alles verstehen, aber bis zum Ende dieses Semesters werden alle hier verwendeten Dinge noch dreimal vorkommen und unter verschiedenen Gesichtspunkten erläutert werden.

Ein objektorientiertes System besteht aus vielen Objekten, die miteinander kommunizieren, um die Aufgaben des Systems uzu erledigen. Die verschiedenen Objekte müssen sich also kennen.

Definition 1.3.17 (Assoziation)
Eine *binäre Assoziation* ist eine Beziehung zwischen zwei *Klasse*n. Sie gibt an, das es zu einem *Objekt* der einen Klasse ein definierte Anzahl von *Objekt*en der anderen Klasse gibt. Über die *Assoziation* kann von einem *Objekt* effizient auf das *Objekt* oder die *Objekt*e der anderen Klasse zugegriffen werden. Abb. 1.2 zeigt ein einfaches Beispiel.

Abb. 1.2: Binäre Assoziation

Dieses Diagramm bedeutet: Ein Haustier hat genau eine Person als Besitzer. Eine Person kann kein, ein oder viele Haustiere haben. Die Zahlen bzw. $*$ an den Enden der Assoziation bedeuten dabei Folgendes:

1 Zu jedem Objekt auf der anderen Seite gibt es genau ein Objekt.

0..1 Zu jedem Objekt auf der anderen Seite gibt es kein oder ein Objekt.

***** Zu jedem Objekt auf der anderen Seite gibt es kein, ein oder viele Objekte.

1..* Zu jedem Objekt auf der anderen Seite gibt es mindestens ein Objekt.

Assoziationen haben also zwei wichtige Bedeutungen:

1. Sie definieren Regeln: In Abb. 1.2 Zu einem Haustier gibt es genau eine Person als Besitzer.
2. Sie sind Schnellstraßen um effizient von einem Objekt zu einem anderen zu kommen.

◄

Definition 1.3.18 (Generalisierung, Spezialisierung, GenSpec)
Eine Klasse B ist eine Spezialisierung einer Klasse A, wenn jedes Objekt aus B auch ein Objekt aus A ist. A ist dann eine Generalisierung von B. Eine solche Beziehung zwischen zwei Klassen wird kurz als *GenSpec-Beziehung* oder auch Vererbung bezeichnet. ◄

Eine GenSpec-Beziehung wird durch eine Linie mit einem kleinen Dreieck an der allgemeineren Klasse, der Generalisierung, symbolisiert.

Beispiel 1.3.19 (Generalisierung und Spezialisierung)
Abbildung 1.3 zeigt einige GenSpec-Beziehungen, die im Folgenden erläutert werden.

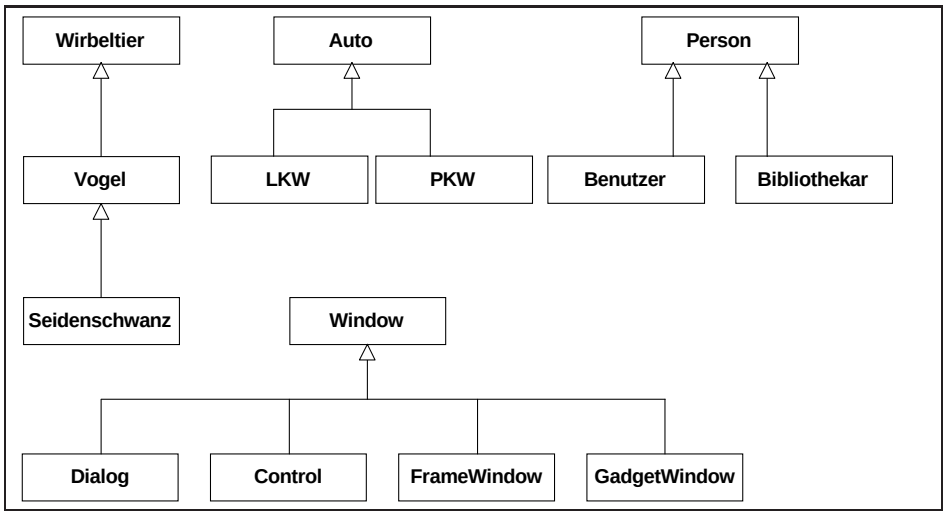


Abb. 1.3: GenSpec-Beziehungen

1. Oben links ist die eingangs erwähnte GenSpec-Hierarchie aus dem Tierreich dargestellt, die keinen Bezug zur Informatik hat. Ob die Spezialisierungsstruktur aus der Zoologie so auch Code übertragen werden soll, hängt von der Aufgabenstellung ab. Die Seidenschwänze bilden eine Familie in der Ordnung der Sperlingsvögel [Gar90]. Es gibt also auch andere Abbildungsmöglichkeiten.
2. Für eine Zulassungsstelle oder einen TÜV ist es sinnvoll, Autos in PKW und LKW zu differenzieren. PKW und LKW haben hinreichend viele Gemeinsamkeiten, die die Zusammenfassung zu einer Klasse rechtfertigen. Sie haben aber auch viele Unterschiede, z. B. verschiedene vorgeschriebene Untersuchungen, die das Bilden spezialisierter Klassen rechtfertigen. Hier sind die beiden spezialisierten Klassen *disjunkt*, da durch Vorschriften eindeutig geregelt ist, wann ein Auto ein LKW und wann ein PKW ist.

3. Eine Klasse Person ist für eine Bibliotheksanwendung viel zu allgemein. Hier wird man Spezialisierungen benötigen, wie Benutzer und Bibliothekar. Ob es gerechtfertigt ist, gemeinsame Eigenschaften dieser Klassen in eine Klasse Person zusammenzufassen, hängt von der konkreten Aufgabenstellung ab. Kriterien, die bei der Entscheidung helfen, werden in den folgenden Kapiteln diskutiert. Diese beiden spezialisierten Klassen sind nicht notwendig *disjunkt*. Ein Bibliothekar wird auch ein Benutzer der Bibliothek sein können, in der er tätig ist. Derartige Fragestellungen werden detailliert in der Vorlesung Software-Engineering behandelt.

4. Als letztes Beispiel zeigt Abb. 1.3 einen Ausschnitt aus der Struktur einer typischen Klassenbibliothek zur Gestaltung von Anwendungen unter MS-Windows oder anderen Betriebssystemen, deren Navigation auf Fenstern beruht, wie KDE für Linux oder OS-X.

◄

Die Differenzierung in Klassen unterschiedlichen Spezialisierungsgrades ermöglicht es, jeweils die Abstraktionsebene zu wählen, die im gegebenen Kontext angemessen ist. Werden nur allgemeine Eigenschaften benötigt, so müssen auch nur die (wenigen) Eigenschaften einer allgemeineren Klasse betrachtet und berücksichtigt werden. Handelt es sich um eine speziellere Aufgabe, so stehen alle Eigenschaften der spezialisierten Klassen zur Verfügung.

Definition 1.3.20 (Polymorphismus)
Polymorphismus bedeutet, dass eine Nachricht unterschiedliche Methoden auslösen kann, je nachdem, zu welcher Klasse das Objekt gehört, an das sie geschickt wird. Bezogen auf Programmiersprachen heißt dies, dass die Methode nicht zur Umwandlungszeit an ein Objekt gebunden werden kann. ◄

Geht eine Nachricht an ein Objekt einer Klasse in einer GenSpec-Hierarchie, so wird die entsprechende Methode der Klasse ausgeführt, die am weitesten „unten“ in der Hierarchie steht.

Innerhalb einer Klasse kann eine Methode *überladen* werden:

Definition 1.3.21 (Überladen)
Eine Operation *op* ist überladen, wenn es mehrere Operationen mit diesem Namen gibt, die sich in ihrer jeweiligen Parameterliste unterscheiden. ◄

Nicht alle Programmiersprachen unterstützen Überladen von Methoden. In Java ist das möglich, in Ruby nicht.

Beispiel 1.3.22 (Überladen)
Eine Klasse Artikel habe eine Operation *getPreis()* ohne Parameter. Gibt es eine feste Preisliste, so mag das genügen. In Deutschland kann man theoretisch um den Preis feilschen wie auf einem Basar. Das geht nicht im Supermarkt, häufig aber etwa beim Kauf teurerer Produkte oder auch bei Banken. Also steht der Preis nicht fest, sondern es kann ein Rabatt vereinbart werden. Dazu könnte eine weitere Operation dienen, die dann *getRabattiertenPreis* heißen könnte und den Rabatt auf den Listenpreis in Prozent erhält. Die Operation *getPreis* kann aber auch überladen werden: *getPreis(rabattSatz)*, so dass es eine ohne und eine mit einem Parameter gibt.

1. getPreis(): Diese liefert den Brutto-Preis, also inklusive gesetzlicher Mehrwertsteuer.

2. getPreis(rabattSatz): Diese liefert den um den vereinbarten Rabattsatz reduzierten Brutto-Preis.

Der Name des Parameters hat mit dem Überladen nichts zu tun. Es geht nur um die Parameter und ihre Typen.

In den Java Klassen *Integer* und *Long* gibt es eine überladene Klassenmethode *toString*:

```
public static String toString(int i)
public static String toString(int i, int radix)
```

bzw.

```
public static String toString(long i)
public static String toString(long i, int radix)
```

Die jeweils erste Methode liefert die String-Darstellung zur Basis (englisch radix) 10 ◄

In einer Vererbungshierarchie kommt noch eine weitere Fähigkeit zum Tragen, die mit Polymorphismus zu tun hat. Eine Methode einer Klasse kann eine Methode aus einer Oberklasse überschreiben:

Definition 1.3.23 (Überschreiben)
Eine Methode *op* einer Klasse A überschreibt eine Methode der Klasse B, wenn A eine Unterklasse von B ist und die Signatur von *op* in A mit der von *op* in B identisch ist. ◄

Beispiel 1.3.24 (Überschreiben)
Jede Java-Klasse erbt von der Klasse *Object* die Methode *toString*. Diese liefert einen für Menschen (zumindest Informatiker) lesbare Darstellung eines Objekts der Klasse als String. Für eine Klasse wie *Person* aus Beispiel 1.3.19 könnten Sie sie so überschreiben: Ich unterstelle dabei, dass die Klasse nur zwei Attribute hat: *nachname* und *vorname*. Dann könnten Sie diese Methode in *Person* so überschreiben:

```
String toString(){
   return vorname + " " + nachname;
}
```

Sie würde also für die Person Beatrice Kiddo aus Kill Bill statt eines beep gerade *Beatrice Kiddo* liefern. ◄

Logisch zusammengehörige Klassen und Schnittstellen gruppiert man zu Teilsystemen, in Java Paket (package) genannt.

Definition 1.3.25 (Paket)
Ein Paket ist eine Menge zusammengehöriger Klassen. In Java liegen die Klassen eines Pakets in einem Verzeichnis. ◄

Ein Paket bildet ein Teilsystem in einem System. Abbildung 1.4 zeigt drei Teilsysteme als Karteikarten mit Reiter (siehe auch [CY90a]):

HIC Human Interaction Component. Dieses Teilsystem enthält die Klassen, die für die Interaktion mit dem Anwender verantwortlich sind.

PDC Problem Domain Component. Dieses Teilsystem enthält die Klassen, die die Anwendungslogik realisieren.

DMC Data Management Component. Dieses Teilsystem enthält die Klassen, die für die Speicherung der Daten verantwortlich sind.

Die «access»-Abhängigkeiten zwischen den Teilsystemen zeigen, dass es sich um eine *Client-Server*-Beziehung handelt: Die HIC verwendet ausschließlich das Teilsystem PDC und die PDC verwendet ausschließlich das Teilsystem DMC. Man nennt eine solche Architektur Dreischichtenarchitektur. Sie können sich das so vorstellen, dass die unterste Schicht eine virtuelle Maschine ist, auf der die zweite „läuft“ und auf dieser wiederum die oberste Schicht. Die gestrichelten Pfeile kennzeichnen die Abhängigkeit: Der Pfeil zeigt auf das Paket das benötigt wird. Die Bezeichnung ist hier «access», hätte aber auch «import»genannt werden können.

Definition 1.3.26 (Kapselung)
Kapselung oder *Geheimnisprinzip* bedeutet, dass die interne Struktur eines Objekts, insbesondere seine Attribute, nicht von außen sichtbar oder gar veränderbar ist. Information über Objekte werden nur kontrolliert über zugängliche Methoden zu Verfügung gestellt. ◄

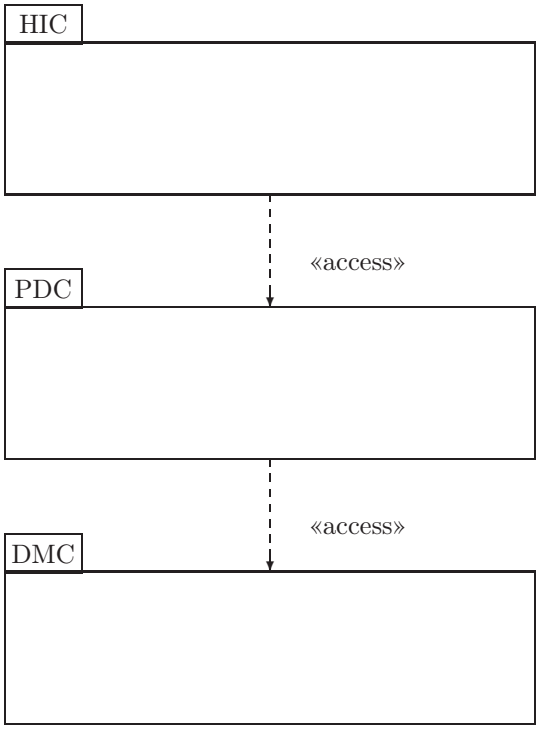


Abb. 1.4: Eine einfache Dreischichtenarchitektur

Die wesentlichen Aspekte der Objektorientierung kann man damit so zusammenfassen: Ein objektorientiertes System ist durch

- Kapselung (Geheimnisprinzip),
- Abstraktion (GenSpec-Beziehungen) und
- Polymorphismus

gekennzeichnet, die als objektorientiertes Dreieck in Abb. 1.5 nach [HS92] dargestellt sind.

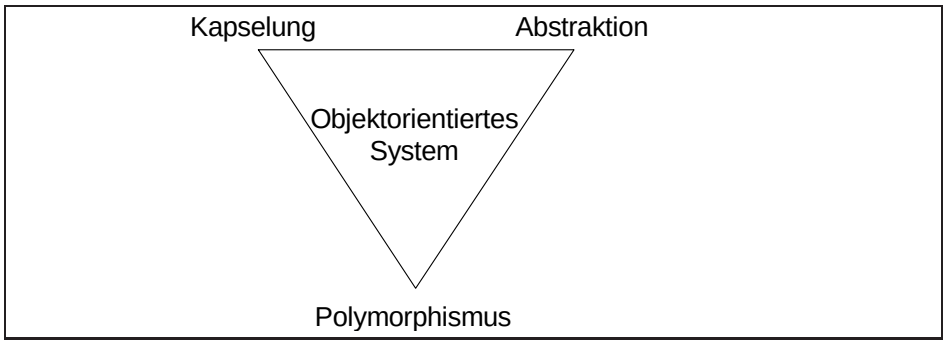


Abb. 1.5: Das objektorientierte Dreieck

Es kann in manchen Situationen sinnvoll sein, Elemente von Klassen zu definieren, die alle Objekte der Klasse gemeinsam haben. Dies gilt sowohl für Attribute als auch für Operationen.

Um diese zu verwenden braucht man also kein Objekt der Klasse. Die Nachricht zum Aufruf einer solchen Klassenoperation geht also an die Klasse und nicht an ein individuelles Objekt. Klassenattribute haben einen Wert, den sich alle Objekte der Klasse „teilen“. Ich definiere deshalb noch:

Definition 1.3.27 (Klassenattribut und -Methode)
Ein Attribut, dass es nur einmal mit einem Wert für alle Objekte einer Klasse gibt, heißt *Klassenattribut*. Eine Methode, die nicht für ein Objekt aufgerufen wird, sondern für die Klasse, heißt *Klassenoperation* bzw. *Klassenmethode*. ◀

Beispiel 1.3.28 (Klassenmethode)
Viele Klassenmethoden finden Sie in den Java-Klassen *Collections* und *Arrays*. ◀

Definition 1.3.29 (Utility-Klasse)
Eine Klasse, die nur Klassenmethoden hat, heißt *Utility-Klasse*. ◀

Bemerkung 1.3.30 (Utility)
Im Amerikanischen ist *utility* ein Unternehmen, dass Dienste wie Wasser, Strom, Gas etc. liefert [Mis98]. Im Deutschen also ein *Versorger* oder Versorgungsunternehmen. ◀

1.4 Programmierung

In den Aufgaben, die ein Programmierer oder Entwickler zu lösen hat, geht es oft um konkrete Objekte aus einem Teil der Realität. Diese müssen genau verstanden werden (Analyse) und es müssen Objekte in einer Programmiersprache entworfen werden (Design), die den Anforderungen genügen. Diese Themen werden ausführlich in den Veranstaltungen zum Software-Engineering behandelt.

Dazu benötigt man Speicherplatz, um Informationen zu halten und die Regeln, nach denen die Informationen verarbeitet werden. Diese finden sich letzendlich dann im Code. Diese Dinge müssen sachgerecht organisiert werden:

- Das System muss übersichtlich sein, damit man es verstehen und ändern oder erweitern kann.
- Das System muss effizient, d. h. in der Regel vor allem schnell sein.
- Das System muss effizient zu entwickeln sein.

Auch zu diesen drei Punkten lernen Sie ab drittem Semester mehr, u. a.in den Veranstaltungen zum Software-Engineering. Aber auch die Programmiersprache spielt für alle drei genannten Punkte eine Rolle.

Sie lernen in den ersten beiden Semestern in dieser Veranstaltung eine objekt-orientierte Programmiersprache — *Java* (Technische Informatik, Wirtschaftsinformatik) bzw. zwei: *Ruby* und Java. Dieses sind aber weder die einzigen Programmiersprachen noch die für jedes Problem besten. Es gibt wahrscheinlich über dreitausend Programmiersprachen. Für viele spezielle Probleme gibt es auch spezielle Programmiersprachen (Domain Specific Languages, DSL). Fasst man den den Begriff weit genug, so gehören auch HTML, XML, TEX, LATEX (hiermit setzte ich z. B. Skripte wie dieses), Mathematica oder MatLab zu den Programmiersprachen. Unter Programmiergesichtspunkten sind für Sie sicher auch Sprachen wie C++, C# (C Sharp) oder JavaScript, PHP, Pearl, Python, Ruby interessant.

Es ist aber weitgehend egal, welche Sprache Sie zu erst lernen. Ziel einer (ersten) universitären Programmierausbildung im Rahmen eines Informatikstudiums muss es immer sein, dass Sie bei Bedarf eine (jede?) weitere Programmiersprache schnell lernen können. Mit „schnell“ meine ich, Sie sollten nach einer Woche in der Lage zu sein produktive Programme zu schreiben oder zumindest ein bestehendes Programm weiter zu entwickeln.

Da die maximale Dauer für eine Klausur vier Stunden beträgt, kann ich das Erreichen dieses Lernziels am Ende des ersten oder zweiten Semesters nicht überprüfen.

Im Vordergrund dieser Veranstaltung stehen Methoden und Stile der Programmierung, die grundsätzlich auf viele Programmiersprachen übertragbar sind und von vielen Informatikern als gut angesehen werden. Einige für diese Veranstaltung sinnvolle Regeln habe ich in Anhang A zusammengestellt.

Im Prinzip brauchen Sie für objekt-orientierte Programmierung nur Folgendes zu wissen:

> Da Sie in dieser Veranstaltung objekt-orientiert programmieren lernen, werden Sie Objekte auf dem Rechner erzeugen. Um objekt-orientiert etwas zu bewirken, brauchen Sie ein Objekt. Haben Sie ein Objekt, so können Sie dessen Operationen aufrufen. Es gibt im Wesentlichen nur diesen Weg mittels objekt-orientierter Programmierung eine Aufgabe zu bewältigen. Das hört sich sehr einfach an und es ist auch wirklich relativ einfach. Haben Sie bereits eine passende Klasse (siehe den Aspekt „Klasse als Objektfabrik“ von Def. 1.3.13), prima, dann erzeugen Sie ein Objekt dieser Klasse. Wenn Sie keine passende Klasse finden, müssen Sie sich eine schreiben.

Ich versuche Ihnen in dieser Vorlesung zu zeigen, wie das geht. In Kap. 2 und Kap. 6 gibt es dazu einfache(?) Beispiele, wie das in Java geht.

1.5 Eclipse

In dieser Veranstaltung verwenden wir Eclipse. Im pub von Johann Abrams finden Sie eine Eclipse-Installation, wie Sie sie im AI-Labor vorfinden. Mindestens in diesem Umfang empfehle ich Ihnen sich auch selbst Eclipse zu installieren.

Wichtige Komponenten um Eclipse zu ergänzen sind:

Java API Dokumentation Diese finden Sie unter download.oracle.com. Sie können unter Runtime Library (rt.jar) aber auch ein lokales Verzeichnis angeben, in dem Sie diese Dokumentation gespeichert haben. Dann sind Sie unabhängig vom Internetzugang.

JUnit Ein nützliches Testframework. Für Sie bietet es zunächst die Möglichkeit, Ihre Klassen auf einfache Art „zum Laufen“ zu bringen und zu testen. Dies ergänzen Sie für ein Projekt, in dem Sie unter den Properties des Projekts die aktuelle JUnit Library dem Java Build Path hinzufügen.

Java Sourcen Unter Runtime Library (rt.jar) können Sie unter *Source Attachement* den Java Sourcecode hinzufügen. Das ist gerade für Anfänger nützlich um Beispiele zur Hand zu haben.

Es gibt noch viele weitere Werkzeuge, von denen Sie später nach Bedarf Gebrauch machen können. Einige Informationen hierzu habe ich in Anhang B zusammengestellt.

1.6 Rechner, Betriebssystem, Compiler und Konsorten

Alles was Sie Programmieren, läuft auf einem Rechner. Wie ein Rechner funktioniert lernen Sie in anderen Veranstaltungen. Aber einige Grundlagen müssen Sie bereits jetzt kennen. Der hier folgende Überblick soll Grundprinzipien erläutern, er erhebt keinen Anspruch auf Wirklichkeitstreue im Detail. Ich beschränke mich auf eine ganz einfache Sicht der Dinge.

Hardware

Aktuelle Rechner bauen auf der von-Neumann-Architektur auf (nach John von Neumann), die in Abb. 1.6 für einen alten Prozessor skizziert ist. Es gibt einen Prozessor (hier oben links der 80486). Dieser ist mit anderen Teilen durch sog. „Busse“ verbunden, über die Daten und Befehle

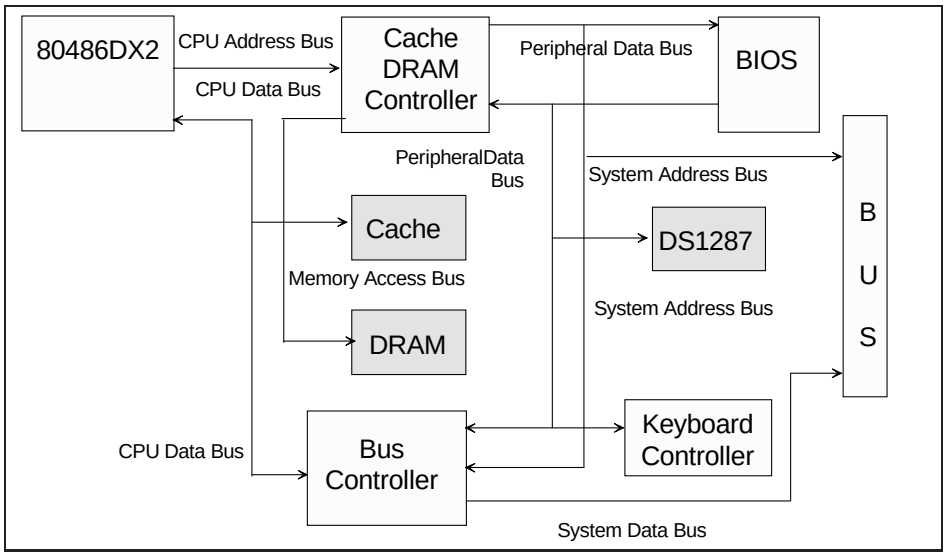


Abb. 1.6: Von-Neumann-Archtektur

übertragen werden. Damit alleine können Sie aber weder als Anwender noch als Programmierer mit einer sog. „höheren“ Programmiersprache viel anfangen. Dazu brauchen Sie ein Betriebssystem.

Betriebssystem

Das Betriebssystem stellt Ihnen Funktionen zum Umgang mit dem Rechner zur Verfügung, die von Programmierern genutzt werden können. Es bietet Schnittstellen zwischen dem Benutzer, den Anwendungsprogrammen und der Hardware und steuert die Ausführung von Programmen. Dieses System können Sie mindestens unter zwei Blickwinkeln („Sichten“) betrachten:

- Es ist eine abstrakte Maschine, die Anwender und Programmierer von der Hardware abschirmt. So laufen Windows oder Android auf verschiedener Hardware, ohne das sich die Nutzung ändert.

- Es ist ein „Betriebsmittelverwalter“, der die Betriebsmittel (Prozessor, Speicher, Dateien, Ein-/Ausgabegeräte) den Anwendungsprogrammen zuteilt. Haben Sie es mit einem einzelnen Prozessor zu tun, so entsteht nur dadurch der Eindruck, dass verschiedene Programme parallel laufen würden.

1.6.1 Compiler/Linker

Viele Programmiersprachen werden als Texte in einer Sprache (C, C++, Java uvam.) geschrieben. Um auf einem Rechner ausgeführt werden zu können, müssen Sie in Maschinen-Code umgewandelt oder übersetzt werden, der auf dem jeweiligen Betriebssystem ausgeführt werden können. Diese Umwandlung leistet ein *Compiler*. Bei einem ganz einfachen Programm könnte es das schon sein. Meistens besteht ein Programm aber aus vielen Einzelteilen. Oft werden diese einzelnen Teile separat übersetzt. Damit sie dann zusammenpassen, müssen sie noch zu einem ausführbaren Programm zusammengebunden werden. Diese Arbeit erledigt ein sogenannter *Linker*. Mit der Frage des „Linkens“ brauchen Sie sich als Anfänger in Java zunächst nicht zu belasten. Ein Linker spielt erst in Java 9 eine (wichtige) Rolle, die Sie zu gegebener Zeit kennenlernen werden.

1.6.2 Virtuelle Maschine

Java und viele andere Sprachen laufen nicht direkt auf einem Betriebssystem, sondern auf einer sog. *virtuellen Maschine*. Java-Programme können auf jedem Rechner ausgeführt werden, auf dem es eine virtuelle Maschine für Java gibt (JVM: Java Virtual Machine). Diese bildet also sozusagen ein Betriebssystem im Betriebssystem. Einzelheiten über die JVM finden Sie in [LYBB14]. Auf der java JVM laufen ca. 300 Programmiersprachen, hier sei nur *Scala* erwähnt.

1.6.3 Java Compiler

Wie der Name schon sagt, ist der Java-Compiler javac der Compiler für Java. Er unterscheidet sich aber von den Compilern für Sprachen wie C, Pascal etc. dadurch, dass er keinen direkt ausführbaren Code erstellt, sondern sog. *Bytecode*. Er übersetzt automatisch alle Teile des Source-Codes in Bytecode. Die JVM übernimmt dann zur Laufzeit, lädt die jeweiligen Teile nach Bedarf in den Speicher der JVM, führt die Betriebssystemfunktionen aus, die durch die Bytecode-Befehle angesprochen werden usw.

1.7 Historische Anmerkungen

Der Beginn der Objektorientierung wird oft auf das Jahr 1980 datiert, in dem die erste Smalltalk-Version erschien. Die Sprache Simula, auf der Smalltalk in gewissem Sinne aufbaute, kann man als objektbasiert bezeichnen. Betrachtet man den Abstraktionsprozess als wesentlich für Objektorientierung, so geht dies natürlich viel weiter zurück. Die wohl erste objektorientierte Programmiersprache, die einen nennenswerten Verbreitungsgrad erreichte, war C++ (gesprochen C plus plus)

Viele Einführungen in Programmiersprachen beginnen mit einem Beispiel namens *HelloWorld* in irgend einer Schreibweise, die mit der Syntax und dem Stil der Sprache verträglich ist. Ich habe darauf bewusst verzichtet, weil dies (nicht nur) für Java völlig untypisch für guten Programmierstil ist. Bestärkt fühle ich mich in dieser Entscheidung auch durch [Wes01].

Die Abhängigkeiten in Abb. 1.4 sind jetzt benutzerdefinierte Stereotypen. Mit UML 2.4 entfiel der bisherige Standardstereotyp «access».

Seit Java 9 gibt es auch in Java einen *Linker*.

1.8 Aufgaben

1. ([02])Was versteht man unter Objekt, was unter Klasse? Was sind die Unterscheidungsmerkmale?
2. ([01]) Was versteht man unter Klasse?
3. ([01]) Was versteht man unter Kapselung?
4. ([01]) Was versteht man unter Polymorphismus?
5. ([02]) Welche Beziehungen bestehen zwischen Polymorphismus und dynamischer Bindung?
6. ([03]) Nennen und erläutern Sie bitte die grundlegenden Konzepte der Objektorientierung!
7. ([02]) Welche charakteristischen Eigenschaften haben Objekte?
8. ([01]) Wodurch wird der Zustand eines Objekts beschrieben?
9. ([05]) Welche Charakteristiken des Anwendungsbereiches lassen sich durch reichhaltige Generalisierungsstrukturen besonders gut beschreiben? Welche Anwendungsbereiche weisen diese Eigenschaften typischerweise auf?

10. ([00])Was versteht man unter Objektidentät?

11. ([10]) Jedes Objekt hat eine Identität und ist unabhängig von seinem jeweiligen Zustand von jedem anderen Objekt unterscheidbar. Für Klassen mit vielen Objekten ist es aber nicht trivial sie zu unterscheiden. Geben Sie für die folgenden Klassen an, wie man ihre Objekte eindeutig charakterisieren könnte:

    11.1. Alle Menschen der Welt zum Zwecke des Postversands.

    11.2. Alle Menschen der Welt für kriminalpolizeiliche Untersuchungen.

    11.3. Alle Kunden mit Schließfächern in einer Bankfiliale.

    11.4. Alle Telefone der Welt um sie anrufen zu können.

    11.5. Alle Kunden einer Telefongesellschaft um die Telefonrechnung erstellen zu können.

    11.6. Alle electronic mail Adressen der Welt.

    11.7. Alle Mitarbeiter einer Firma um ihren Zugang zu Firmen-Ressourcen zu steuern.

12. ([10]) Hier folgen einige Listen von Objekten. Untersuchen Sie, was diese Objekte jeweils gemeinsam haben, und bilden Sie geeignete Klassen.

    12.1. Elektronenmikroskop, Brille, Fernrohr, Laserzielgerät, Fernglas.

    12.2. Fahrrad, Segelboot, PKW, LKW, Flugzeug, Segelflugzeug, Motorrad, Pferd.

    12.3. Nagel, Schraube, Bolzen, Niete.

    12.4. Zelt, Höhle, Hütte, Garage, Scheune, Haus, Wolkenkratzer.

    12.5. Quadratwurzel, Sinus, Cosinus, Exponentialfunktion.

    Stellen Sie die Beziehungen in einem Klassendiagramm dar. Bilden Sie bei Bedarf geeignete, zusätzliche Klassen.

13. ([10])Stellen Sie die folgenden Zusammenhänge als Assoziation, Aggregation bzw. Vererbung dar! Begründen Sie jeweils Ihre Entscheidung:

    13.1. Eine Land hat eine Hauptstadt.

    13.2. Ein essender Philosoph benutzt zwei Stäbchen.

    13.3. Eine Datei ist eine gewöhnliche Datei oder ein Verzeichnis.

    13.4. Eine Datei enthält Sätze.

    13.5. Ein Polygon wird durch eine geordnete Menge von Punkten beschrieben.

    13.6. Ein Objekt einer Zeichnung ist Text, ein geometrisches Objekt oder eine Gruppe.

    13.7. Eine Person benutzt eine Programmiersprache in einem Projekt.

    13.8. Modem und Tastatur sind I/O Einheiten.

    13.9. Klassen können mehrere Attribute haben.

    13.10. Eine Person spielt in einem Jahr in einem bestimmten Team.

    13.11. Eine Strecke verbindet zwei Städte.

    13.12. Ein Studierender hört eine Vorlesung bei einem Professor.

14. ([0]) Was heißt „C++“, wenn Sie es in Java Syntax interpretieren?

Kapitel 2

Ein Einführungsbeispiel

Niedere Mathematik
Ist die Bosheit häufiger
oder die Dummheit geläufiger?
Mir sagte ein Kenner
menschlicher Fehler
folgenden Spruch:
„Das eine ist Zähler,
das andere Nenner,
das Ganze — ein Bruch!“
Erich Kästner, [Käs67]

2.1 Übersicht

In diesem Kapitel nehme ich eine ganz schmale „Tiefbohrung“ in die objekt-orientierte Programmierung mit Java vor. Dabei führe ich einige Grundprinzipien vor, wie Sie an objekt-orientierte Programmieraufgaben herangehen können. Gleichzeitig stelle ich die wichtigsten Hilfsmittel vor, die Ihnen mit Java zur Verfügung stehen. Objekte einer Klasse sollen eine klar definierte Aufgabe haben. Es handelt sich also eher um Spezialisten als um Generalisten. Entsprechend sind die Aufgaben der einzelnen Methoden ebenfalls eng umrissen. Daraus folgt dann auch, dass sie klein sind. Sie können durchaus aus einer einzelnen Zeile bestehen!

Dieses Beispiel illustriert eine Art von Aufgabe, wie Sie Ihnen sehr wahrscheinlich nach Abschluss Ihres Studiums sehr früh gestellt werden wird: Schreiben Sie bitte eine Klasse, die bestimmte, vorgegebene Eigenschaften hat.

Wenn Sie den Eindruck haben, das sei alles viel zu viel: Keine Panik! [Ada81] Alles das, was

Abb. 2.1: Keine Panik!(www.google.com/doodles/douglas-adams-61st-birthday)

in diesem Kapitel im „Schnelldurchgang“ präsentiert wird, erhält später noch eine systematische und ausführliche Erläuterung.

2.2 Lernziele

- Eine Klasse in Eclipse anlegen können.
- Attribute und Methoden in Java kennen.
- Einen JUnit Testfall in Eclipse anlegen können.
- Wissen, dass es die API-Dokumentation zu Java gibt und wo Sie sie finden.
- Wissen, dass der Sourcecode für Java verfügbar ist und wie Sie ihn finden.
- Das Interface *Comparable* kennen.
- Objekte mit *equals* und *compareTo* vergleichen können.

2.3 Motivation

Als erstes Beispiel für das grundlegende Prinzip der Objektorientierung aus Abschn. 1.1 betrachte ich die Aufgabe einen Text auf der Konsole auszugeben:

1. Dazu brauchen Sie ein Objekt einer geeigneten Klasse. Wenn Sie etwas Erfahrung gesammelt haben, werden Sie eine geeignete Klasse kennen oder schnell finden. Ich beschreibe hier Überlegungen, die dafür hilfreich sein können.

   Die Konsole ist ein ganz grundlegendes Element eines Computersystems. Für den Umgang mit derartigen Dingen gibt es ein Java eine Klasse *System*, die Sie im Paket (engl. *package*) *java.lang* finden. Dieses befindet sich im Module *java.base*. Gucken Sie sich die API-Definition dieser Klasse an, so sehen Sie, dass sie ein Klassenattribut *out* der Klasse *PrintStream* hat. Sie brauchen sich also gar kein neues Objekt dieser Klasse erzeugen, Sie holen es sich einfach aus der Klasse *System*: Da dies ein Klassenattribut ist, brauchen Sie dazu gar kein Objekt der Klasse *System*, sondern Sie setzen nach *System* einen „.“ und können dann das gewünschte Attribut angeben, hier *out*.

2. Um auf einfache Weise etwas auf der Konsole auszugeben hat die Klasse *PrintStream* viele Methoden, die alle mit *print* anfangen. Ich nehme hier die Methode *println*, die einen Parameter der Klasse *String* erwartet. Auch diese Klasse befindet sich im Paket *java.lang*, da sie ganz grundlegend ist.

3. Die Java API Dokumentation gibt es im Internet. Googlen Sie einfach nach „Java 8 API“. Sie können sie auch herunterladen und lokal installieren, dann ist sie auch offline verfügbar. In Eclipse brauchen Sie nur *Shift F2* (↑ *F2*) zu drücken und Sie bekommen die API-Dokumentation zu dem entsprechenden Element angezeigt.

4. Damit haben Sie den Methodenaufruf gefunden, den Sie verwenden wollen: Sie schreiben einfach nach *System.out* weiter mit einem „.“ und dann dem Methodennamen mit dem gewünschten „String“. Ein *String* ist eine Zeichenkette. Wenn Sie diese fest vorgeben wollen, so setzten Sie in „Tüttelchen“, also doppelte Anführungsstriche. Die ganze Anweisung wird durch ein Semikolon „;“ abgeschlossen. Auf Hamburgisch z. B. etwa so:

   ```
   System.out.println("Moin, Moin");
   ```

5. Nun müssen Sie dies noch *zum Laufen* bringen. Sie werden später mehr darüber erfahren, wie das geht. Hier nehme ich ein Hilfsmittel, dass Sie oft verwenden können: Ich möchte ausprobieren oder testen, ob ich das richtig verstanden habe und das auch das passiert, was ich möchte. Ein Hilfmittel dafür ist JUnit. Dazu erstelle ich mir in Eclipse einen JUnit Testfall (Testcase): *TestAusgabe* mit nur einer Methode *testPrintlnString*:

```
10 public class TestAusgabe {
20   @Test
30   public void testPrintlnString() {
40      System.out.println("Moin, Moin");
50   }
60 }
```

Ein geschweiftes Klammerpaar {...} definiert in Java einen *Block* und hat nichts mit dem mathematischen Begriff der Menge zu tun.

6. Diese kann ich in Eclipse jetzt als JUnit-Testfall ausführen und auf der Konsole erscheint wie erwartet: „Moin, Moin“. Das ganze Drumherum um den Aufruf der Methode nimmt Ihnen JUnit ab. Sie müssen „nur“ mit Zeile 20 leben: Dort steht *@Test*. Das ist ein Hinweis, an dem JUnit erkennt, das die folgende Methode ausgeführt werden soll, hier also *testPrintlnString*. Was das genau ist, lernen Sie in Kap. 20.

2.4 Klasse Bruch: Grundlagen

Nun sollen Sie lernen, wie eine Java-Klasse entwickelt werden kann. Java hat viele eingebaute Datentypen, wie ganze Zahlen und Dezimalzahlen unterschiedlicher Länge, aber keine Brüche. Eine solche Klasse *Bruch* soll nun entwickelt werden. Dazu sehen wir uns erst einmal an, was für ein Objekt ein Bruch — mathematisch gesprochen eine rationale Zahl $q \in \mathbb{Q}$ – ist und was Sie damit machen können.

1. Ein Bruch q hat einen ganzzahligen Zähler z und einen ganzzahligen Nenner n: $q = z/n$, wobei gelten muss $n \neq 0$.

2. Für ganze Zahlen stellt Java u. a. die Typen *int* und *long* zu Verfügung. Der erstgenannte Typ hat 32 Bit, der andere 64 Bit. Für Einzelheiten verweise ich Sie auf später und Kap. 7. Ich entscheide mich hier für *int*.

Damit sieht die Klasse Bruch zunächst so aus, wie in Abb. 2.2 gezeigt. Sie hat nur zwei Attribute

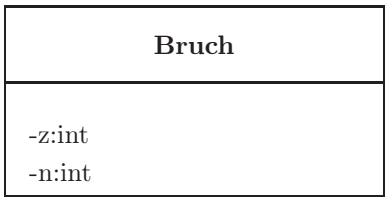


Abb. 2.2: Eine Klasse Bruch, V 1

vom Typ *int*: z (für Zähler) und n (für Nenner).

Die Java-Klasse, die dem Diagramm 2.2 entspricht, ist natürlich ebenfalls noch ganz klein:

```
10 package introexample;
20 /**
30  * Eine Klasse, deren Objekte Brüche repräsentieren.
40  * Zähler und Nenner sind int.
50  * @author Bernd Kahlbrandt
60  *
70  */
80 public class Bruch{
90   /**
```

```
100     * Zähler des Bruchs
110     */
120     private int z;
130     /**
140      * Nenner des Bruchs.
150      */
160     private int n;
170 }
```

Die Klassendeklaration beginnt mit einem *package statement*: Das sagt Ihnen hier, dass sich die Klasse im Paket *introexample* befindet. In den folgenden fünf Zeilen finden Sie einen javadoc-Kommentar: Er beginnt mit „/**“ und endet mit „*/“. In Ihren Klassen erwarte ich zunächst nur einen solchen knappen javadoc-Kommentar. Wie in jedes Dokument gehören auch in Ihren Code der Name bzw. die Namen der Autoren: Für jeden Autor ein *@author*. In Zeile 70 folgt nun das erste wichtige Java-Element: Hier beginnt eine öffentliche (*public*) Klasse (*class*) namens *Bruch*. Nach dem Klassennamen folgt eine öffnende geschweifte Klammer „{“ Die zugehörige schließende geschweifte Klammer „}“ beendet in Zeile 170 die Definition der Klasse *Bruch*. Das Schlüsselwort *public* gibt an, dass alle anderen Java-Klassen diese Klasse verwenden können.

In den Zeilen 120 und 160 werden die beiden Attribute der Klasse definiert: z für Zähler, n für Nenner. Beides sind ganze Zahlen (*int*) und sind im Unterschied zur Klasse *private* deklariert. Die Klasse soll von allen benutzt werden können, die Attribute aber nicht. Sie sollen nur innerhalb der Klasse zur Verfügung stehen. Attribute mit dem Schlüsselwort *private* können nur von Elementen der Klasse (d. h. Methoden von allen Objekten der Klasse) verwendet werden. Im Diagrammen, wie Abb. 2.2, wird ein vorstelltes „-“-Zeichen zur Kennzeichnung von *private* verwendet.

Mit Brüchen können Sie rechnen: Sie brauchen also so etwas wie Addition, Subtraktion, Multiplikation und Division. Dazu brauchen Sie Methoden. Hier zunächst das Beispiel der Addition: Der Kürze halber und trotzdem verständlich nenne ich die Methode *add*. Als Parameter bekommt sie einen Bruch und liefert als Ergebnis die Summe des Bruches, für den sie aufgerufen wird und dem als Parameter übergebenen zurück. Analog kann ich Methoden *sub* und *mult* definieren. Beim Dividieren müssen Sie aber aufpassen: Was ist bei Division durch 0? Ich merke mir das als potenzielles Problem und definiere auch eine entsprechende Methode *div*. Die Klasse Bruch sieht nun so aus, wie im Klassensymbol in Abb. 2.3. Nun zum zugehörigen Java-Code: In Java können Sie

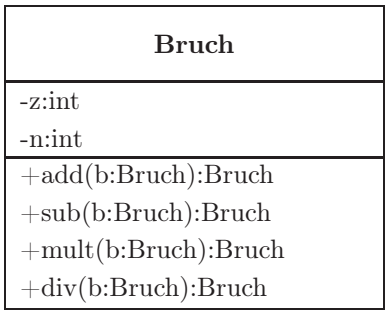


Abb. 2.3: Eine Klasse Bruch V 2

— nach einigen weiteren Erläuterungen — schon den folgenden Code schreiben: Dabei sind die ersten 16 Zeilen genau die aus dem ersten Codeschnipsel. Nur in Zeile 170 steht jetzt nicht mehr die schließende geschweifte Klammer „}“. Diese kommt jetzt erst in Zeile 500.

```
 10 package introexample;
 20 /**
 30  * Eine Klasse, deren Objekte Brüche repräsentieren. Zähler und Nenner sind int;
 40  * @author Bernd Kahlbrandt
 50  *
```

```
 60  */
 70 public class Bruch implements Comparable<Bruch>{
 90   /**
100    * Zähler des Bruchs
110    */
120    private int z;
130    /**
140     * Nenner des Bruchs.
150     */
160    private int n;
170
180    /**
190     * Addiert den übergebenen Bruch zu diesem und liefert das Ergebnis zurück.
200     * @param q ein Bruch
210     * @return Die Summe dieses Bruchs und q.
220     */
230     public Bruch add(Bruch q){
240        return null;
250     }
260     /**
270      * Subtrahiert den übergebenen Bruch von diesem und liefert das
280      * Ergebnis zurück.
290      * @param q ein Bruch
300      * @return Die Differenz dieses Bruchs und q.
310      */
320     public Bruch sub(Bruch q){
330        return null;
340     }
350     /**
360      * Multipliziert den übergebenen Bruch mit diesem und liefert das
370      * Ergebnis zurück.
380      * @param q ein Bruch
390      * @return Das Produkt dieses Bruchs und q.
400      */
410     public Bruch mult(Bruch q){
420        return null;
430     }
440     /**
450      * Dividiert diesen Bruch durch den übergebenen Bruch und liefert das
460      * Ergebnis zurück.
470      * @param q ein Bruch
480      * @return Der Quotient dieses Bruchs und q.
490      */
500      public Bruch div(Bruch q){
510         return null;
520      }
530 }
```

Die Methoden habe ich wie schon die Attribute mit javadoc-Kommentaren erläutert.

Sie sind *public*, denn sie sollen von anderem Java-Code verwendet werden. Nach *public* kommt der Rückgabetyp. In diesem Fall ist das die Klasse *Bruch*, da die Methode einen *Bruch* liefern soll. Dann folgt der Name der Methode. In den runden Klammern „(...)“ nach dem Methodennamen stehen die Parameter. Das ist hier jeweils einer und zwar ein Objekt der Klasse Bruch. Wie nach dem Klassennamen folgt nach dem vollständigen Methodennamen, inklusive der Parameter in

Klammern, eine geschweifte Klammer. Von dieser bis zur entsprechenden schließenden geschweiften Klammer geht der Code der Methode. Es ist hier jeweils eine Zeile die mit *return* beginnt. Auch dies ist ein Schlüsselwort: Es gibt an, dass danach folgt, was die Methode zurückgeben soll. Da Sie ja erst ganz am Anfang der Programmierausbildung gebe ich hier noch *null* zurück. Diese Schlüsselwort bezeichnet das *null*-Objekt für alle Klassen. Es wird Ihnen meistens dann begegnen, wenn Sie vergessen haben, einer Variablen einen Wert zuzuweisen und trotzdem auf die Variable zu greifen. Dann gibt es zur Laufzeit eine Fehlermeldung, genauer eine sog. *exception*.

Ich weise extra auf Folgendes hin: Die vier Methoden werden den Bruch *nicht* verändern, für den sie aufgerufen werden. Es gilt als guter Programmierstil in einer Methode das Objekt nicht zu verändern, wenn die Methode etwas zurückgibt. Umgekehrt soll eine Methode nichts zurückgeben, wenn sie das Objekt verändert.

Im Prinzip könnten wir schon fast anfangen, die Klasse zu benutzen, aber eben noch nicht ganz. Methoden sind deklariert, auch wenn sie noch nichts tun. Ich muss aber auch Brüche erstellen können. Dazu gibt es in Java *Konstruktoren*. Ein *Konstruktor* heißt wie die Klasse. Gibt es keinen, so erstellt Java einen *default Konstruktor*. Der initialisiert *Attribute* vom Typ *int* mit 0. Das ist uns für den Zähler recht, nicht aber für den Nenner, denn der darf ja nicht 0 sein. Also muss ich mir einen eigenen Konstruktor schreiben, der Zähler und Nenner übergeben bekommt:

```
public Bruch(int z, int n){
   this.z = z;
   this.n = n;
}
```

Hier sehen Sie das Schlüsselwort *this*. Es wird hier verwendet, um das jeweilige Objekt anzusprechen (zu referenzieren). „=“ ist der Zuweisungsoperator: Der Wert rechts vom Gleichheitszeichen wird der Variablen links vom Gleichheitszeichen zugewiesen.

Nun generiert mir aber Java keinen Default-Konstruktor mehr. Ein solcher Konstruktor ohne Parameter ist hier aber sinnvoll: Als Default *Bruch* halte ich 0/1 für sinnvoll:

```
public Bruch(){
   this(0,1);
}
```

Das Schlüsselwort *this* wird hier verwendet um einen weiteren Konstruktor aufzurufen. Oft ist es so, dass ein Konstruktor die eigentliche Arbeit macht und die anderen diesen mit verschiedenen Voreinstellungen aufrufen. Hier ruft der Konstruktor ohne Parameter den mit zwei Parametern mit den Voreinstellungen 0 und 1 auf. Dies nennen manche auch *Constructor Chaining*.

Wir können bisher noch gar nicht „in die Objekte hineingucken“, die Attribute der Klasse *Bruch* sind *private*. Aber jede Klasse ist Unterklasse der Klasse *Object* und die hat nach API-Dokumentation eine Methode *toString()*. Probieren Sie die gleich einmal aus, indem Sie in der Methode *testPrintLineString()* der Klasse *TestAusgabe* die folgende Zeilen einfügen:

```
Bruch b = new Bruch(1,2);
System.out.println(b.toString());
```

Als Ausgabe sollten Sie so etwas erhalten wie:

```
introexample.Bruch@187275d
```

Das ist nichts, das ein Mensch so einfach lesen kann. Also überschreibe ich in der Klasse *Bruch* die Methode *toString()* aus *Object*, z. B. so:

```
@Override
public String toString() {
   return "(" + this.z + "/" + this.n + ")";
}
```

Nun erhalte ich als Ausgabe:

(1/2)

@Override ist ein Hinweis an den Compiler. Er besagt, dass diese Methode eine aus der Oberklasse *überschreiben* soll und nicht etwa eine völlig neue Methode ist. Das Schlüsselwort *return* kennen Sie nun schon und ebenso die Strings in doppelten Hochkommata. Neu ist das „+“-Zeichen zwischen *Strings*. Mit diesem werden *Strings* zu einem neuen *String* zusammengefügt. Das funktioniert auch für *int*, hier liefert Java automatisch eine Darstellung der Zahl als *String*. Dies gilt auch bei einigen anderen Typen dieser Art, aber dazu später mehr. Die runden Klammern setze ich um den Bruch, damit ich später auch Rechenoperationen einfach lesen kann.

Bevor ich nun daran gehe, die vier Methoden für die Grundrechenarten zu implementieren, also zwischen die geschweiften Klammern „{...}“ nach dem vollständigen Methodennamen etwas Sinnvolles und nicht „*return null*“ zu schreiben, überlege ich mir, wie ich denn überprüfen könnte, ob ich das richtig gemacht habe. Dabei fange ich mit ganz einfachen Testdaten an und stelle sie in einer Tabelle zusammen.

Methode	Testbruch	Parameter	Erwartetes Ergebnis
add	0	0	0/1
	0	1	1/1
	1/2	1/2	1/1
	2/3	1/3	1/1
sub	0	0	0/1
	0	1	-1/1
	2/3	1/3	1/3
	5/7	7/5	-24/35
mult	0	11/2	0
	1/4	2/1	1/2
	11/7	7/11	1/1
	42/3	3/6	7/1
div	1/1	0	Fehler
	1/2	1/2	1/1
	42/5	1/5	42/1
	25/4	4/5	125/16

Um zu sehen ob das klappt schreibe ich mir eine *JUnit*-Testklasse *BruchTest01*. Die wesentlichen Teile generiert mir schon *Eclipse* (wie dies auch andere Entwicklungsumgebungen tun).

Die Klasse *BruchTest01* finden Sie im pub. Sie weist noch gravierende Schwächen auf und ist deshalb hier nicht abgedruckt.

Nun mache ich mich an die Implementierung der Rechenmethoden.

Wegen

$$\frac{a}{b} + \frac{c}{d} = \frac{a \cdot d + c \cdot b}{b \cdot d}$$

komme ich auf:

```
public Bruch add(Bruch q){
   return new Bruch((this.z*q.n + q.z*this.n),this.n*q.n);
}
```

Bemerkung 2.4.1 (Achtung, das geht besser!)
Bei dieser Implementierung gehe ich mit dem „Holzhammer“ vor. Tatsächlich brauche ich den Nenner nur auf das kleinste gemeinsame Vielfache (kgV), englisch least commen denomimator (lcd) zu erweitern. Damit laufe ich nicht unnötig schnell aus dem Bereich der *int*s hinaus und muss seltener kürzen. ◄

Ich bringe die beiden Brüche auf einen Hauptnenner, addiere die Zähler und erzeuge einen neues *Bruch*-Objekt, das die Methode dann zurückgibt. Das Symbol für die Multiplikation ist „*“, das für die Addition „+“. Ganz analog bei *sub*:

```
public Bruch sub(Bruch q){
   return new Bruch((this.z*q.n - q.z*this.n),this.n*q.n);
}
```

Bei der Multiplikation muss ich nur Zähler mit Zähler und Nenner mit Nenner multiplizieren

```
public Bruch mult(Bruch q){
   return new Bruch(this.z*q.z, this.n*q.n);
}
```

Zur Division multipliziere ich mit dem Kehrwert:

```
public Bruch div(Bruch q){
   return this.mult(new Bruch(q.n,q.z));
}
```

Dies führt allerdings dazu, dass zwei neue Bruchobjekte erzeugt werden, von denen nur eines wirklich benötigt wird.

Die Klasse *BruchTest01* weist noch Schwächen auf: So gibt sie die Brüche nur aus, statt direkt das erwartete Ergebnis gegen das tatsächliche Ergebnis zu testen. Da kann man leicht etwas übersehen. Trotzdem liefert sie nützliche Hinweise:

1. Beim ersten Testfall für *div* gibt es keinen Fehler. Dabei dividieren wir doch durch 0, oder? Aber Achtung: Im Nenner des Bruches, den *div* zurückliefert steht zwar 0, aber es wird ja gar keine Division ausgeführt: wir multiplizieren mit dem Kehrwert. Also gibt es auch keinen Fehler.

2. In fast allen Fällen, wo 1/1 erwartete wurde, steht so etwas wie 2/2, 77/77 o. ä. Auch in anderen Fällen könnte das Ergebnis vereinfacht werden. Hieraus erkennen Sie, dass die Ergebnisse noch gekürzt werden müssen.

3. Es wäre nützlich, wenn Sie an den Wert des Bruchs herankommen könnten.

Diesen Themen müssen wir uns jetzt im nächsten Abschnitt zuwenden.

2.5 Erste Verbesserungen

Um einen Bruch kürzen zu können, müssen Sie den größten gemeinsamen Teiler von Zähler und Nenner finden. Der Algorithmus (das Verfahren) hierfür geht schon auf Euklid zurück.

Um den größten gemeinsamen Teiler $ggT(n, m)$ zweier ganzer Zahlen zu bestimmen, bilde man den Rest der bei Division von m durch n verbleibt: $rest = m \bmod n$. Das ist nur sinnvoll, wenn $m \geq n$ ist. Andernfalls vertausche man zunächst m und n. Ist der Rest gleich 0, so ist n der größte gemeinsame Teiler. Andernfalls machen wir mit $m = n$ und $n = rest$ und $rest = m \bmod n$ weiter, bis wir $rest = 0$ erreicht haben.

Dies geht in Java wie folgt:[1]

```
int ggT(int m, int n) {
   if(m<n){
      int tmp = n;
      n = m;
      m = tmp;
   }
   int rest = m % n;
   while (rest != 0) {
     m = n;
```

[1] Siehe Abschn. 7.8 für weitere Erläuterungen

```
      n = rest;
      rest = m % n;
   }
   return n;
}
```

Da die Methode nur innerhalb der Klasse *Bruch* verwendet werden soll, deklare ich sie als *private*. Mit *if* prüfe ich, ob $m < n$ ist. In den runden Klammern nach dem *if* steht etwas, das im Ergebnis wahr (*true*) oder falsch (*false*) liefert. Ist also $m < n$, so vertausche ich n und m. Dazu benötige ich die Hilfsvariable *tmp* vom Typ *int*. Zu dieser Methode gibt es am Ende dieses Kapitels die Aufgabe 4.

Neu sind auch der *modulo*-Operator „%“ und die *while*-Schleife. Der *modulo*-Operator liefert den Rest nach der Division von m und n. Die *while*-Schleife, d. h. der Block $\{\dots\}$, wird solange durchlaufen, wie die Bedingung in den runden Klammern nach dem Schlüsselwort *while* den Wert *true* liefert.

Damit können wir die Klasse *Bruch* wie folgt weiterentwickeln: Es werden die zum Kürzen benötigten Methoden eingefügt:

```
   private void kuerzen() {
      int ggt = this.ggT(this.z, this.n);
      this.n /= ggt;
       this.z /= ggt;
   }
   private int ggT(int m, int n) {
      if (m < n) {
         int tmp = n;
         n = m;
         m = tmp;
   }
   int rest = m % n;
   while (rest != 0) {
      m = n;
      n = rest;
      rest = m % n;
   }
   return n;
}
```

Hier wird gegen eine Java Code-Konvention verstoßen, siehe Aufgabe 4.

Mit dem Kürzen ändern sich alle Methoden analog zur folgenden neuen Version der Methode *add*:

```
   public Bruch add(Bruch q) {
      Bruch result = new Bruch((this.z * q.n + q.z * this.n), this.n * q.n);
      result.kuerzen();
      return result;
   }
```

Nun liefert die Klasse *BruchTest01* die erwarteten Ergebnisse. Auch der erwartete Fehler im ersten Testfall für *div* tritt jetzt auf: „java.lang.ArithmeticException: / by zero“. Um den Umgang mit so etwas werden wir uns in einem späteren Kapitel kümmern. Außerdem: Wer eine Division durch 0 vornimmt hat selber Schuld! Trotzdem nehme ich das zum Anlass in der Methode *ggT(n,m)* für m gleich 0 den Wert n zurück zu liefern.

Nun verbessere ich noch die Testklasse: Statt dort nur die Ergebnisse auszugeben, will ich diese dort abprüfen. Dazu schreibe ich noch schnell zwei *getter*-Methoden für die beiden Attribute:

```
   public int getZaehler() {
      return this.z;
   }
   public int getNenner() {
      return this.n;
   }
```

Das *return* Statement gibt das danach folgende Objekt zurück und verlässt die Methode. Nun kann ich eine bessere Testklasse *BruchTest* schreiben. Hier ein Auszug:

```
import static org.junit.Assert.assertEquals;
public class BruchTest {
   private Bruch zero = new Bruch();
   private Bruch one =  new Bruch(1,1);
   private Bruch einhalb =  new Bruch(1,2);
   private Bruch eindrittel =  new Bruch(1,3);
   private Bruch zweidrittel =  new Bruch(2,3);
   @Test
   public void testBruch() {
      Bruch q = new Bruch();
      assertEquals(0,q.getZaehler());
      assertEquals(1,q.getNenner());
   }
   ...
   @Test
   public void testAdd() {
      assertEquals(0,zero.add(zero).getZaehler());
      assertEquals(1,zero.add(zero).getNenner());
      assertEquals(1,zero.add(one).getZaehler());
      assertEquals(1,zero.add(one).getNenner());
      assertEquals(1,einhalb.add(einhalb).getZaehler());
      assertEquals(1,einhalb.add(einhalb).getNenner());
      assertEquals(1,zweidrittel.add(eindrittel).getZaehler());
      assertEquals(1,zweidrittel.add(eindrittel).getNenner());
   }
   ...
```

Da *BruchTest* eine Klasse ist, kann sie auch Attribute haben. Hier habe ich mir fünf Brüche definiert, die ich in allen Testfällen verwende. Die Methode *assertEquals* ist eine Klassenmethode, die mit JUnit kommt. Durch die *import*-Anweisung am Anfang brauche ich den Klassennamen (*Assert*) nicht jedes Mal davor zu schreiben. Die Methode *assertEquals* hat hier zwei Parameter vom Typ *int*: Als ersten Parameter das erwartete Ergebnis aus obiger Tabelle und als zweiten Parameter das tatsächliche Ergebnis, dass unsere jeweilige Methode liefert.

Bevor Sie nun daran gehen, die Klasse fertig zu programmieren, stelle ich Ihnen aber vor, wie Sie versuchen können, Ihre Implementierung auf Fehler zu überprüfen.

Dazu überlege ich mir zunächst einige weitere Tests, z. B. für die Konstruktoren.

1. Aufruf des default-Konstruktors, also *Bruch()* ohne Parameter. Ich erwarte, dass ein Bruch (also ein *Bruch*-Objekt mit Zähler 0 und Nenner 1 erstellt wird.

2. Aufruf des anderen Konstruktors mit den Parametern 1 und 2. Ich erwarte, dass der Zähler nun 1 ist und der Nenner 2.

3. Für die anderen Methoden *setWert*, *add*, *sub*, *mult*, *div* sollen Sie sich weitere Testfälle überlegen (siehe Aufgabe 2).

Ich zeige nun, wie solche Fälle in JUnit durchgeführt werden können. Das Ergebnis sehen Sie im Paket *introexample* (bzw. *skript.kap01*.

Wie Sie das in Eclipse machen, lesen Sie bitte in Abschn. B.6 nach und führen es dann selbstständig durch. Bis auf die Testfälle für die Konstruktoren, *setWert*, *getZaehler* und *getNenner* schlagen natürlich noch alle Testfälle fehl.

2.6 Gleichheit

Nun haben wir also Attribute definiert, Methoden zum Rechnen deklariert und Konstruktoren und sogenannte getter-Methoden geschrieben.

In den Testfällen haben wir uns Zähler und Nenner besorgt um das Ergebnis zu überprüfen. In der Klasse *Object* gibt es aber bereits eine Methode *equals*, die auf Gleichheit prüft. Hier der Sourcecode:

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

Da jede Klasse in Java eine Unterklasse von *Object* ist, können zwei Objekte also immer mittels *equals* verglichen werden. Wenn Sie das nicht anders implementieren, werden zwei Objekte dann als identisch angesehen, wenn sie den selben Speicherbereich belegen. Nun ist aber intuitiv $\frac{1}{2} = \frac{2}{4}$, die beiden Brüche würden aber verschiedene Speicherbereiche belegen. Probieren Sie das in den Testfällen aus und vergleichen mit *assertEquals* direkt den erwarteten Bruch mit dem Ergebnis, so erhalten Sie aber immer *false*. Hier der Grund dafür: Eine Variable, die *nicht* von einem primitiven Typ ist, enthält die Variable nicht das Objekt, sondern die Adresse, an der das Objekt in der JVM steht. Diese Adresse nennt man *Referenz*. Also müssen wir die Methode *equals* überschreiben. Die erste Idee dazu wäre die Folgende:

```
@Override
public boolean equals(Bruch b){
     return this.z == b.z && this.n == b.n;
}
```

Hier tauchen schon wieder neue Dinge auf: *@Override* ist eine sogenannte *Annotation* (siehe Kap. 20). Sie dient als Hinweis für den Compiler und hilft Ihnen typische Anfängerfehler zu vermeiden. Sie gibt an, dass die mit dieser Annotation versehene Methode eine aus einem Interface) oder einer Oberklasse (hier *Object*) implementiert oder *überschreibt*. Ein häufiger Anfängerfehler besteht darin, *equals* mit einem Parameter vom Typ der jeweiligen Klasse zu deklarieren, hier also *Bruch*. Dann wird die Methode *equals* aber gar nicht *überschrieben*, sondern *überladen*: Jedes *Bruch*-Objekt hat dann zwei *equals*-Methoden: Die aus *Object* ererbte mit Parameter der Klasse *Object* und die überladene mit Parameter der Klasse *Bruch*. Je nachdem wird die eine oder die andere ausgewählt.

Die primitiven Typen — insbesondere die ganzzahligen — können mit „==“ verglichen werden. Dieser Operator liefert *true* bzw. *false*, je nachdem ob die linke und die rechte Seite gleich sind oder nicht. Der Operator „&&“ ist der logische *und*-Operator. Er liefert genau dann *true*, wenn beide Operanden *true* sind, andernfalls *false*.

Versuchen Sie diesen Code zu compilieren, so bekommen Sie einen Fehler! Aber nur, weil ich die Annotation *@Override* verwendet habe. Sonst würde ich erst zur Laufzeit einen Fehler bekommen. Die Methode *equals* aus *Object* erwartet als Parameter ein *Object*. Also ändere ich die Methode so:

```
@Override
public boolean equals(Object b){
     return this.z == b.z && this.n == b.n;
}
```

Ein *Object* hat aber keine Attribute z und n. Ich bekomme jetzt also einen anderen Fehler. Wir haben also ein Problem zu lösen. Dazu überlege ich mir zunächst dies: Ist ein *Object* kein Bruch, so kann es auf keinen Fall gleich einem *Bruch* sein. Ich muss also abprüfen, ob es sich überhaupt um ein Objekt der Klasse *Bruch* handelt.

Um in den Typ oder die Klasse eines Objekts zu überprüfen gibt es den *instanceof*-Operator. Im Beispiel müssen wir prüfen, ob der übergebene Parameter von *equals* ein Bruch ist:

```
o instanceof Bruch ?
```

Der *instanceof*-Operator überprüft zunächst, ob das Objekt auf der linken Seite (o) initialisiert ist. Ist dies nicht der Fall, ist es also *null*, so liefert der Operator *false*. Andernfalls prüft er, ob das Objekt vom Typ auf der rechten Seite ist; in diesem Fall also, ob es sich um einen Bruch handelt. Ist dies der Fall, so liefert er *true* andernfalls *false*. Handelt es sich um einen Bruch, so können wir das übergebenen Objekt in einen Bruch umwandeln. Das geschieht durch einen sogenannten *Cast*:

```
(Bruch)o
```

macht aus einem Objekt einen Bruch, wenn dies möglich ist. Mit diesem neuen Wissen können Sie nach dem Fragezeichen oben gleich weiter schreiben: Direkt nach dem Fragezeichen, dass was Sie tun wollen, wenn der *instanceof*-Operator *true* liefert, Danach einen Doppelpunkt „:“ und dann das, was Sie tun wollen, wenn der Operator *false* liefert. Damit ist die Methode *equals* fertig:

```
@Override
public boolean equals(Object b){
   return b instanceof Bruch ? this.z == ((Bruch)b).z && this.n == ((Bruch)b).n:false;
}
```

Hier lernen Sie gleich den *ternären Operator* kennen.

2.7 Vergleichbarkeit: Größer und kleiner

Nun gibt es aber zwischen Brüchen aber auch eine Ordnung, man kann je zwei Brüche miteinander vergleichen. Stets ist klar ob der eine größer ist als der andere, umgekehrt oder ob sie gleich sind. Dafür gibt es in Java das Interface *Comparable*. Dies hat eine Methode *compareTo*, die einen Parameter vom zu vergleichenden Typ bekommt und je nach dem eine negative, ein positive ganze Zahl oder im Fall der Gleichheit 0 zurückliefert. Wenn Sie das Interface Comparable implementieren müssen Sie angeben, für welchen Typ bzw. Klasse Sie das tun wollen. Dieser Typparameter wird in spitzen Klammern angegeben, hier also *Comparable<Bruch>*. Gleichheit wird aber auch durch *equals* überpüft. Diese Prüfung muss das gleiche Ergebnis liefern, wie die Gleichheitsprüfung mit *compareTo*.

Also lasse ich die Klasse Bruch auch noch das Interface Comparable implementieren. Zusammengefasst sieht die Klasse nun so aus Der Kreis rechts vom Klassensymbol ist das Symbol für eine Schnittstelle. Die Verbindungslinie zwischen Klassen- und Schnittstellensymbol besagt, dass die Klasse die Schnittstelle implementiert. Dieser Teil des Diagramms bedeutet also: Die Klasse *Bruch* implementiert die Schnittstelle *Comparable*.

Wenn *compareTo* und *equals* ein zusammenpassendes (konsistentes) Ergebnis liefern sollen, muss *equals* genau dann *true* liefern, wenn *compareTo* den Wert 0 liefert. Also müssen Sie nicht nur *compareTo* implementieren, sondern auch *equals* überschreiben. So wie Ihnen eine Entwicklungsumgebung, wie Eclipse automatisch die Methode *compareTo* generiert, können Sie sich dann auch *equals* generieren lassen. Die sehen dann zunächst so aus:

```
public class Bruch implements Comparable<Bruch>{
   ...
   @Override
   public int compareTo(Bruch o) {
```

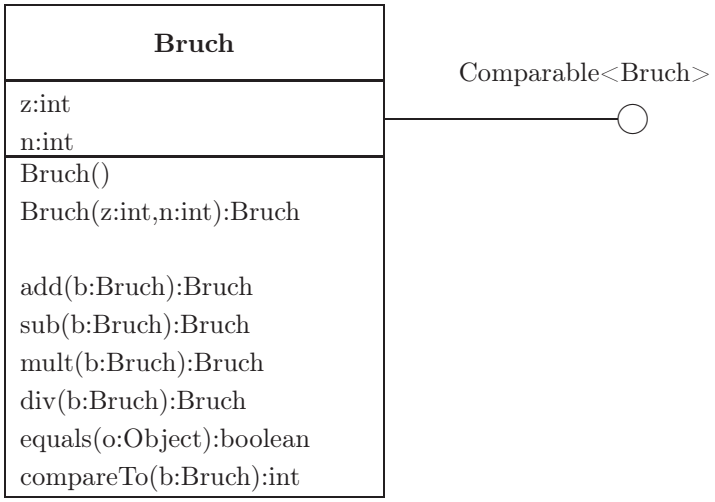


Abb. 2.4: Eine Klasse Bruch

```
// TODO Auto-generated method stub
   return 0;
}
@Override
public boolean equals(Object obj) {
// TODO Auto-generated method stub
   return super.equals(obj);
}
```

Außerdem hat Eclipse natürlich nur sehr rudimentäre (dummy) Methoden generieren können: *compareTo* liefert immer 0, sieht also je zwei Brüche immer als gleich an. *equals* ruft einfach die Methode *equals* der Oberklasse auf. Um sie von der in *Bruch* zu unterscheiden, muss das Schlüsselwort *super* davor geschrieben werden. Um entsprechend zu markieren, dass hier noch etwas getan werden muss, steht dort der Zeilenkommentar

```
// TODO Auto-generated method stub
```

damit Sie nicht vergessen, die Methode richtig auszuprogrammieren.

Nun zu *compareTo* und *equals*. *compareTo* ist die Methode, die hier implementiert werden muss, um die Brüche der Größe nach vergleichen zu können. Die Implementierung von *equals* muss sich also an der von *compareTo* orientieren. Auf den ersten Blick scheint das ganz einfach zu gehen:

```
00   @Override
10   public equals(Object o){
20      if(this.compareTo(o)==0){
30         return true;
40      }else{
50         return false;
60      }
70   }
```

So einfach ist es im Prinzip, aber es geht nicht ganz so. Aber zunächst zur Erläuterung der neuen Syntax-Elemente:

1. *if-then-else* ist eine Verzweigung. In den Klammern nach *if* steht ein Ausdruck, der einen booleschen Wert, also wahr oder falsch, *true* oder *false* liefert. In dem von geschweiften

Klammern begrenzten Block {...} danach stehen die Anweisungen, die ausgeführt werden, wenn der Ausdruck *true* liefert, in dem Block nach dem *else* stehen die Anweisungen, die andernfalls ausgeführt werden sollen. Im Beispiel wird *this.compareTo(o)* mit 0 verglichen. Für numerische Datentypen wie *int* geht das mit dem Vergleichsoperator ==.

2. Zeile 20 führt zu einem Compiler-Fehler: *compareTo* erwartet hier ein *Bruch*-Objekt, bekommt aber ein beliebiges Objekt. Hier müssen Sie also noch mehr lernen.

3. Nun müssen wir noch den Fehler beseitigen, auf den der Compiler hinweist. Um in den Typ oder die Klasse eines Objekts zu überprüfen gibt es den *instanceof*-Operator. Im Beispiel müssen wir prüfen, ob der übergebene Parameter von *equals* ein Bruch ist:

   ```
   o instanceof Bruch ?
   ```

   Der *instanceof*-Operator überprüft zunächst, ob das Objekt auf der linken Seite (*o*) initialisiert ist. Ist dies nicht der Fall, ist es also *null*, so liefert der Operator *false*. Andernfalls prüft er, ob das Objekt vom Typ auf der rechten Seite ist; in diesem Fall also, ob es sich um einen Bruch handelt. Ist dies der Fall, so liefert er *true* andernfalls *false*. Handelt es sich um einen Bruch, so können wir das übergebenen Objekt in einen Bruch umwandeln. Das geschieht durch einen sogenannten *Cast*:

   ```
   (Bruch)o
   ```

   macht aus einem Objekt einen Bruch, wenn dies möglich ist. Mit diesem neuen Wissen können Sie nach dem Fragezeichen oben gleich weiter schreiben: Direkt nach dem Fragezeichen, dass was Sie tun wollen, wenn der *instanceof*-Operator *true* liefert, Danach einen Doppelpunkt „:“ und dann das, was Sie tun wollen, wenn der Operator *false* liefert. Damit ist die Methode *equals* fertig:

   ```
   00   @Override
   10   public equals(Object o){
   20      return   o instanceof Bruch ? this.compareTo((Bruch)o)==0 : false;
   30   }
   ```

   Dies hier ist die typische Implementierung von *equals*, wenn die Klasse *Comparable* implementiert.

   Bemerkung 2.7.1 (Achtung mit Casts!)
   Die hier vorgeführte Situation wird für lange Zeit eine der ganz *wenigen* sein, in der Sie einen Cast benötigen. Wenn Sie bis zum Ende des ersten Semesters in die Versuchung zu kommen, einen Cast zu verwenden: Achtung, mit ganz wenigen Ausnahmen haben Sie wahrscheinlich vorher einen Denkfehler gemacht! ◀

4. Nun müssen wir noch die Methode *compareTo* implementieren, z. B. so:

   ```
   00   @Override
   10   public int compareTo(Bruch o) {
   20      Bruch diff = this.sub(o);
   30      int sign = diff.n*diff.z;
   40      return sign>0?1:sign<0?-1:0;
   50   }
   ```

Führen Sie nun die Klasse *BruchTest* als JUnit-Testfall aus, so sollten Sie einen grünen Balken bekommen: Alle Tests liefern das erwartete Ergebnis.

Nun laufen alle Testfälle erfolgreich durch.

Nun müssten wir uns noch um die Fehlerbehandlung kümmern. Aber das verschiebe ich auf ein späteres Kapitel ebenso wie die Beschreibung der Methode *hashCode()*. Diese ist sehr wichtig. Ihre Beschreibung sprengt aber den Rahmen dieses Kapitels.

2.8 Resumé

Sie haben in diesem Kapitel die Dinge gesehen, die Sie in der Programmierung wieder und wieder verwenden werden. Sie werden Klassen schreiben, deren Struktur Sie durch die Attribute festlegen. Sie werden Methoden schreiben, die das umsetzen, was ein Objekt der Klasse gemäß den Anforderungen tun soll. Sie werden Konstruktoren schreiben, die es Entwicklern ermöglichen funktionsfähige Objekte der Klasse zu erstellen. Das werden Sie so tun, dass die Klasse sich nahtlos mit anderen Java-Klassen verwenden lässt. Dazu gehört es oft, die Methode *toString()* aus *Object* zu überschreiben. Wenn Sie das Interface *Comparable* implementieren überschreiben Sie dessen Methode *compareTo* und müssen deshalb auch die Methode *equals* konsistent mit *compareTo* überschreiben.

Die Attribute sind oft *private*. Viele Methoden sind *public*. Die Methoden waren klein. Die längsten waren die Methoden in der Testklasse *BruchTest*. Das ist durchaus typisch! Aber auch die Testklasse lässt sich noch vereinfachen. Allerdings geht das leichter, wenn Sie noch etwas mehr gelernt haben.

Außerdem haben Sie gesehen, wie ich die Namen der Elemente gebildet habe: Klassennamen in *Upper Camel Case*, Attribut- und Methodennamen in *lower Camel Case* und Paketnamen in *lower case*. Für weitere Konventionen verweise ich auf Kap. A im Anhang.

Einige häufig verwendete Klasssen und Typen kamen vor, z. B. ganze Zahlen *int*, *boolean* und *String*. Auch einige wenige Java Anweisungen haben Sie gesehen: *if*, *while*, Zuweisungsoperator $=$, Vergleichsoperatoren $<$, $<==$, Modulo-Operator, die Grundrechenarten $+, -, *, /$, Zuweisung und Rechnung, hier $* =$ sowie den ternären Operator.

Aber: Diese Klasse hat noch gravierende Schwächen. In Aufgabe 2 haben Sie also wirklich die Chance Fehler zu finden. Das ist schon einmal „eine Menge Holz“. Aber keine Sorge, diese Dinge werden alle auch noch einmal ganz systematisch behandelt werden.

2.9 Historische Anmerkungen

Es gibt ganz verschiedene Ansichten darüber, wie Anfängern und Anfängerinnen Programmierung beigebracht werden soll oder kann. Ich gehe hier für Java den Weg, direkt auf den objektorientierten Ansatz zu bauen.

Ich halte es für sehr schwierig eine Programmiersprache ganz ohne „forward references“, also auf Verweise, die erst später aufgelöst werden, so einzuführen dass Lernende hinreichend schnell Aufgaben lösen können, die Spaß machen und Spaß an der Sache halte ich für ganz wichtig.

Es gibt andere Programmierparadigmen, die Informatiker auch kennen müssen. Aber ich halte es für wichtig, dass erst einmal eines richtig verstanden wird.

2.10 Aufgaben

1. Ändern Sie bitte die Methode *div* von Bruch so ab, dass nur noch *ein* neues Bruch-Objekt erzeugt wird!

2. Die oben angegebenen Testfälle führen auf keine Fehler. Konstruieren Sie bitte weitere Testfälle, die auf Fehler führen!

3. Schreiben Sie bitte eine Klasse *Bruch*, bei der Zähler und Nenner vom Typ *long* sind und die Rechenoperationen den jeweiligen Bruch ändern, statt einen neuen zu erzeugen![2] Ergänzen Sie bitte in dieser Klasse (im Vergleich zur Klasse *Bruch* aus Kap. 2) einen öffentlichen *Copy-Konstruktor*, d. h. einen Konstruktor, der als Parameter einen *Bruch* übergeben bekommt und daraus einen neuen Bruch mit gleichem Zähler und Nenner erstellt. Ein solcher Konstruktor erzeugt also eine Kopie des übergebenen Objekts. Demonstrieren Sie an Hand geeigneter Testfälle, dass Ihre Implementierung funktioniert.

[2] Diese Klasse soll also *mutable* sein, siehe Def. 4.4.3

4. In der Methode *ggt* wird eventuell gegen die Java Code-Konventionen verstoßen. Gegen welche? Verbessern Sie die Rechenmethoden entsprechend (siehe Bem.2.4.1). Schreiben Sie bitte eine regelkonforme Version.

5. Welche Konsequenzen hat es, wenn Sie die Methode *kuerzen* so ändern, dass sie nicht den ursprünglichen Bruch kürzt, sondern den gekürzten Bruch als Rückgabe liefert? Wie beurteilen Sie die beiden Varianten?

6. Das berechnen des ggT ist eigentlich eine Dienstleistung, die in vielen Kontexten benötigt wird. Verschieben Sie bitte diese Methode in eine Utility-Klasse (Arbeitstitel: *IntegerFunctions*) und ändern Sie Ihre Klasse *Bruch* so, dass dort diese Methode verwendet wird.

7. Eine professonielle Klasse *Bruch* muss weitere Eigenschaften haben, für deren Sie Wissen benötigen, dass bis zu diesem Punkt noch nicht präsentiert werden konnte. Siehe hierzu Aufgabe 1 in Abschn. 7.10.

Kapitel 3

Einführungsbeispiel für Rubyisten

Niedere Mathematik
Ist die Bosheit häufiger
oder die Dummheit geläufiger?
Mir sagte ein Kenner
menschlicher Fehler
folgenden Spruch:
„Das eine ist Zähler,
das andere Nenner,
das Ganze — ein Bruch!“
Erich Kästner, [Käs67]

3.1 Übersicht

In diesem Kapitel entwickele ich nach dem Vorbild der Klasse Rational aus der Ruby Core-Library eine analoge Klasse in Java. Dabei zeige ich u. a. wie die in Ruby üblichen Praktiken in Java umgesetzt werden. So werden bereits viele der Punkte aus dem Anhang E illustriert. Notgedrungen wird es einige forward-references geben. Dies liegt vor allem an der statischen Typisierung von Java.

3.2 Lernziele

- Mit Java in Eclipse arbeiten können.
- Gemeinsamkeiten und Unterschiede von Klassendefinitionen in Ruby und Java kennen.
- Etwas über das Interface *Comparable* wissen.
- Das Interface Comparable in Java mit dem Module Comparable in Ruby vergleichen können.
- Von einigen Grundelementen von Java einen ersten Eindruck haben.
- JUnit Testfälle für Java-Klassen schreiben können.

3.3 Einführung

Hier einige der wichtigsten Eigenschaften der Klasse Rational in Ruby:

1. Ein Objekt der Klasse Rational ist ein ausgekürzter Bruch.
2. Rationale Zahlen können addiert, subtrahiert, multpliziert und dividiert werden. Dabei kann der zweite Operand ein beliebiges Numeric-Objekt sein.

3. Rationale Zahlen sind Comparable.
4. Die Klasse Rational hat viele Methoden, wie *ceil*, *floor*, etc. Siehe ruby-doc.org.
5. Weiteres entnehmen Sie bitte der Ruby-Dokumentation.

3.4 Implementierung Klassenrumpf

Da die entsprechende Ruby-Klasse das Module Comparable included, muss die Java Klasse das Interface Comparable implementieren:

```
public class Rational implements Comparable<Rational> {
  ...
}
```

Hier sehen Sie gleich Folgendes:

1. In Java kann eine Klasse eine Sichtbarkeit haben, hier *public*. Eine *public* Klasse kann von allen anderen Klassen genutzt werden.
2. Der Klassenname wird in *UpperCamelCase* gebildet.
3. Nicht hier, aber in der Vorlesung sehen Sie, dass der Name der Klassendatei wie der Klassenname gebildet wird.
4. Das Schlüsselwort *implements* gibt an, das die Klasse dass folgende Interface implementieren muss. Das entspricht in etwa dem include Comparable in Ruby. Mehrere Interfaces können hier angegeben werden, jeweils durch Komma getrennt. Hier eben das Interface *Comparable*. Dieses Interface hat eine besonders wichtige Methode *compareTo(Rational rat)*. Diese Methode entspricht der Methode <=> (spaceship operator), die eine Ruby-Klasse implementieren muss, wenn sie das Module Comparable included.
5. Völlig neu für Sie ist, was auf Comparable in der ersten Zeile folgt: *<Rational>*. Java ist statisch typisiert und Comparable ist ein generisches Interface: Es hat einen Typ-Parameter T: Comparable<T>. Für T muss in diesem Kontext die jeweilige Klasse eingesetzt werden, hier also Rational.
6. Nach der Klassendeklaration mit etwaigen implements (und ggf. anderen Elementen) beginnt der Klassenblock mit einer geschweiften Klammer.

Da die Klasse *Rational* das Interface *Comparable* implementiert, generieren Entwicklungsumgebungen direkt einen Teil des Klassenrumpfs:

```
public class Rational implements Comparable<Rational> {
  ...
   @Override
   public int compareTo(Rational rat) {
   ...
   }
}
```

Was taucht hier alles auf?

1. Vor der Methode *compareTo* steht *@Override*. Das ist eine sogenannte *Annotation*. Diese werden ausführlich in Kap. 20 erklärt. Hier nur so viel: Die Annotation schützt Sie vor groben Anfängerfehlern, die der Compiler mit der Hilfe dieser *Annotation* finden kann.
2. Methoden haben wie in Ruby eine Sichtbarkeit, hier *public*. Diese Methoden können von allen anderen aufgerufen werden, wie in Ruby.

3. Danach folgt der Rückgabetyp. Auch das ist anders als in Ruby und der statischen Typisierung von Java geschuldet. In Java müssen Sie angeben, welchen Rückgabetyp die Methode liefert. Hiere ist es *int*. Das ist ein sogenannter primitiver Typ. Diese ganzen Zahlen sind in Java 32 Bit lange ganze Zahlen. Details hierzu finden Sie in Kap. 7. Das ist ein deutlicher Unterschied zu *Integer* (früher *Fixnum* und *Bignum*) in Ruby.

4. Die Methode heißt *compareTo*. Methodennamen werden in Java in *lowerCamelCase* geschrieben, auch das ein kleiner Unterschied zu Ruby, wo diese in *lower_ snake_ case* gebildet werden.

5. Die Parameter werden in Java immer in runde Klammern eingeschlossen, wie es auch in Ruby kommen wird. Neu für Sie ist die Angabe des Typs des Parameters, in diesem Fall *Rational*. Auch dies muss wegen der statischen Typisierung so sein.

Nun zu den Attributen, in Java fields genannt:

```
private int nenner;
private int zaehler;
```

Erläuterungen hierzu:

1. Attribute haben in Java eine Sichtbarkeit, hier *private*. Dies ist in Ruby der default Wert, in Java ist es ein anderer, also gebe ich *private* an. Da Java statisch typisiert ist, muss ein Typ angegeben werden, hier *int* (s. o.). Die Namen werden in *lowerCamelCase* gebildet, hier nenner und zaehler. Umlaute sind zulässig, aber wie in Ruby rate ich von deren Verwendung ab.

2. Eclipse gibt zu diesem Stand noch die Warnung, dass diese privaten Attribute nicht verwendet werden.

3. Ein Statement endet in java mit einem Semikolon.

Zur Erzeugung von Objekten einer Klasse gibt es in Java wie in Ruby *new*. In Ruby ist das eine Klassenmethode der Klasse, die mit der privaten Instanzmethode *initialize* eng gekoppelt ist. In Java ist die ein *Operator*. Mit diesem Operator wird ein *Konstruktor* aufgerufen, der so heißt wie die Klasse.

Da die Attribute bereits deklariert sind, nutze ich eine Eclipse-Funktion: rechte Maus-Taste → Source → Generate Constructor using fields. Hier das Ergebnis

```
public Rational(int nenner, int zaehler) {
   this.nenner = nenner;
   this.zaehler = zaehler;
}
```

Hierzu folgende Erläuterungen:

1. Der Konstruktor heißt wie die Klasse, hat eine Sichtbarkeit (hier public) und die Parameter haben einen Typ.

2. Objekte einer Klasse werden mittels des Operators *new*, gefolgt von einem Konstruktor mit den korrekten Parametern, erzeugt. Konstruktor und new-Operator entsprechen also der initialize und new Methode in Ruby.

Für jede Java Klasse sollte eine Methode *toString* definiert werden. Diese entspricht genau der Methode *to_ s* in Ruby Klassen. Auch die generiere ich mir ähnlich wie oben:

```
@Override
public String toString() {
   return "Rational [nenner=" + nenner + ", zaehler=" + zaehler + "]";
}
```

Erläuterungen:

1. Die Annotation *@Override* habe ich schon oben kurz erläutert.
2. Wie die Methode *to_s* in Ruby hat *toString* keine Parameter.
3. Die Rückgabe erfolgt in Java nur explizit mittels *return*, gefolgt von dem Rückgabewert oder -Objekt.

Nun zu den Methoden für Gleichheit. Das ist in Java wie in Ruby, aber die Methoden heißen anders.

Objektidentität wird in Java mittels == überprüft. Dies ist in Java ein *Operator*. Hier brauchen und können wir also nichts zu tun, so wie in Ruby *equal?* üblicherweise nicht überschreiben wird. Aber in Java entspricht der Methode == aus Ruby die Methode *equals*. Diese wird üblicherweise überschrieben um Objekte auf klassenspezifische Gleichheit zu überprüfen.

Da die Klasse *Rational* das Interface *Comparable* implementiert, müssen *equals* und *CompareTo* konsistent implementiert werden, ebenso *hashCode* (entspricht *hash* in Ruby). Beides ist also ganz analog zu Ruby. Die Methoden-Rümpfe sehen so aus:

```
public boolean equals(Object obj) {
...
}
public int hashCode() {
...
}
```

Nun zu den anderen Methoden der Klasse Rational aus Ruby. Diese führe ich hier nicht alle auf, sondern ergänze sie nur im Sourcecode. Aber an der Methode *truncate* gibt es noch etwas zu erläutern:

```
public double truncate(){
...
}
public double truncate(int precision){
   ...
}
```

1. Die Methode *truncate* ist überladen: Es gibt sie ohne Parameter und mit einem ganzzahligen Parameter.
2. In Ruby Methoden können Parameter default-Werte haben, in Java nicht. Mit Überladen erreichen Sie aber die gleiche Wirkung: Sie rufen in *truncate()* einfach *truncate(0)* auf.

Ganz wichtig sind natürlich die Rechenoperationen. Ich arbeite hier aber nicht alle aus:

Die Additionsmethode in Ruby erwartet als zweiten Operanden ein *Numeric*. In Java gibt es eine ähnliche Klasse: *Number*. Diese erfordert aber noch einige weitere Methoden, auf die ich gerne verzichten wollte. Deshalb hierzu am Ende des Kapitels Aufgabe 1. Um diese Eigenschaft trotzdem in *Rational* umzusetzten, schreibe ich eine überladene Methode *add*:

```
public Rational add(Rational b){
   ...
}
public Rational add(int b){
   ...
}
```

Bei Zahlen sind einige wichtiger als andere. Ich erwähne hier nur die neutralen Elemente für Addition und Multiplikation. Diese werden in solchen Klassen oft als konstante Klassenattribute definiert:

```
public static final Rational ONE = new Rational(1,1);
public static final Rational ZERO = new Rational(0,1);
```

1. Das Schlüsselwort *static* kennzeichnet das Attribut hier als Klassenattribut.

2. Das Schlüsselwort *final* macht das Klassenattribut unveränderlich. Genauer: Ihm kann nur einmal ein Wert zugewiesen werden.

3. Die Namen von Konstanten werden in *SCREAMING_ SNAKE_ CASE* gebildet.

3.5 Design und Implementierung Testfälle

Bevor ich Ihnen zeige, wie die Methoden implementiert werden, entwickele ich einige Testfälle. Dazu markiere ich die Klasse *Rational* im package explorer links, drücke die rechte Maustaste und wähle unter new JUnit Test Case. Dann muss ich nur noch das Verzeichnis auswählen. Anschließend klicke ich auf next (nicht auf finish). Dort wähle ich dann die Methoden aus, die ich testen will. Das sind in diesem Fall alle außer hashCode. Damit werden mir Dummy-Testmethoden generiert, die alle so aussehen:

```
@Test
public void testAbs() {
   fail("Not yet implemented");
}
```

1. *@Test* ist eine Annotation. Dies ist die sinnvolle Art und Weise für JUnit zu signalisieren, dass eine Methode eine Testmethode ist. Dies entspricht dem Präfix *test_* bei Testmethoden für RUnit.

2. Testmethoden müssen public void sein und dürfen keine Parameter haben.

3. *fail* ist eine Methode aus dem JUnit-Framework, die hier einfach signalisiert, dass die Testmethode noch nicht implementiert ist. Genauer ist es eine Klassenmethode der Klasse Assert.

Hier ein ganz einfaches Beispiel für eine Testmethode:

```
@Test
public void testGetNenner() {
   assertEquals(1, new Rational(1,1).getNenner());
}
```

Und hier hier noch eines:

```
assertEquals(Rational.ZERO,new Rational(0,1));
assertEquals(Rational.ONE,new Rational(1,1));
assertNotSame(Rational.ZERO, Rational.ONE);
```

1. Die Methode *assertEquals* ist eine Klassenmethode der Klasse Assert. Sie ist vielfach überladen, aber dazu später mehr. Hier sind beide Parameter ganze Zahlen. Der erste Parameter ist das nach Spezifikation der getesten Methode (hier getNenner) erwartete Ergebnis. Der zweite Parameter ist das Ergebnis des Methodenaufrufs.

2. Die Methode *assertNotSame* vergleicht den ersten und den zweiten Parameter darauf, ob sie das selbe Objekt referenzieren.

3.6 Ausimplementierung der Klasse

Noch schlagen fast alle Testfälle fehl, weil die Methoden der Klasse Rational bisher nur Dummies waren. Nun machen wir uns Zug um Zug an die Implementierung, bis alle Testfälle bestanden werden.

Ich erläutere hier nur einige der Methoden. Hier ein erster Versuch für die Methode *mult*:

```
public Rational mult(Rational b){
   return new Rational(this.zaehler*b.zaehler, this.nenner*b.nenner);
}
```

1. Das geht so zwar, hat aber eine Schäche, die auch gegen die Anforderung verstößt: Der Bruch ist nicht ausgekürzt. Kürzen ist aber keine Methode, die jeder einzelne Bruch benötigt. Es erscheint daher sinnvoller eine Methode *ggt* (größter gemeinsamer Teiler) in einer Utility-Klasse zu schreiben.

Nun zu *div*:

```
public Rational div(Rational b){
   return new Rational(this.zaehler*b.nenner, this.nenner*b.zaehler);
}
```

1. Wie bei *mult* ist der Bruch nicht ausgekürzt.

2. Der Nenner des neuen Bruches kann 0 sein. Was ist dann? Bei einer ganzzahligen Division durch 0 wird in Java eine *ArithmeticException* geworfen. Diese ist eine Unterklasse von *RuntimeException*. Derartige Exception sind *unchecked*, d. h. der Compiler erzwingt ihre explizite Behandlung nicht. Ich könnte eine solche Operation also einfach „durchgehen“ lassen. Dann kann es aber zu einem späteren Zeitpunkt Probleme geben. Die Ursache ist dann nicht mehr leicht nachzuvollziehen. Insofern entscheide ich mich dafür, in diesem Fall eine *ArithmeticException* zuwerfen.

   ```
   public Rational div(Rational b){
      int newNenner = this.nenner * b.zaehler;
      if(newNenner  == 0){
         throw new ArithmeticException("Divisor darf nicht 0 sein!");
      }
      return new Rational(this.zaehler * b.nenner, newNenner);
   ```

1. Das if-Konstrukt kennen Sie bereits aus Ruby. Danach einen Block zu schreiben und nicht nur einen Befehl gilt als guter Stil und vermeidet Fehler.

2. Das Schlüsselwort zum werfen einer Exception ist *throw*. Mit dem new-Operator gefolgt von einem Konstruktor wird das zu werfende Exception Objekt erzeugt.

3. Da *ArithmeticException* unchecked Exceptions sind, brauchen wir hier nichts weiter zu tun.

3.7 Historische Anmerkungen

3.8 Aufgaben

1. Schreiben Sie bitte eine Klasse für rationale Zahlen, die eine Unterklasse von *Number* ist. Meine Lösung hierfür heißt *RationalNumber*.

2. Schreiben Sie bitte eine Klasse *Rational*, bei der Zähler und Nenner vom Typ *long* sind und die Rechenoperationen den jeweiligen Bruch ändern, statt einen neuen zu erzeugen![1] Ergänzen Sie bitte in dieser Klasse (im Vergleich zur Klasse *Rational* aus Kap. 3) einen öffentlichen *Copy-Konstruktor*, d. h. einen Konstruktor, der als Parameter ein *Rational* übergeben bekommt und daraus einen neuen Bruch mit gleichem Zähler und Nenner erstellt. Ein solcher Konstruktor erzeugt also eine Kopie des übergebenen Objekts. Demonstrieren Sie an Hand geeigneter Testfälle, dass Ihre Implementierung funktioniert.

3. In der Methode *ggt* wird eventuell gegen die Java Code-Konventionen verstoßen. Gegen welche? Verbessern Sie die Rechenmethoden entsprechend. Schreiben Sie bitte eine regelkonforme Version.

[1] Diese Klasse soll also *mutable* sein, siehe Def. 4.4.3

Kapitel 4

Klassen

Knowing the syntax of Java does not make someone a software engineer.
John Knight

4.1 Übersicht

In diesem Kapitel beschreibe ich grundlegende Eigenschaften von Klassen in Java. Etwas präziser versuche ich hier zusammenzufassen, was Sie für den Start in die Programmierung mit Java über Klassen wissen müssen. Tatsächlich werde ich mehr als das Notwendigste erläutern. Dadurch enthält dieses Kapitel mehr Stoff, als Sie zu Anfang benötigen. Aber so muss ich Sie nicht öfter als nötig auf spätere Kapitel verweisen.

Neben Klassen gibt es in Java auch sogenannte primitive Datentypen. Auch diese und ihre Zusammenhänge mit Klassen stelle ich hier vor. Auch der Umgang mit *equals* und *hashCode* wird hier im Kontext von Objektidentität erläutert.

4.2 Lernziele

- Klassen in Java schreiben können.
- Objekte in Java erzeugen können.
- Methoden aufrufen können.
- Das Konzept der Generalisierung und Spezialisierung im Java Kontext beschreiben können.
- Die Java-Konstrukte zur Implementierung von Generalisierung und Spezialisierung einsetzen können.

4.3 Struktur von Klassendateien

Klassen sind die Basis der Programmierung in Java. Sie werden in Dateien mit der Endung *.java* geschrieben. Der Compiler erzeugt daraus eine oder mehrere Dateien mit der Endung *.class*.

Eine Klassendatei enthält Anweisungen und kann Kommentare enthalten. Anweisungen werden vom Compiler verarbeitet, Kommentare nicht. Ich fange mit den einfachsten Dingen an: Bei Java-Kommentaren können Sie fast nichts ganz falsch machen. Es gibt drei Arten von Java-Kommentaren: Ein Kommentar kann durch „/*“ eingeleitet und durch „*/“ beendet werden. Ein solcher Kommentar kann sich über mehrere Zeilen erstrecken. Ein Kommentar, der durch „//“ eingeleitet wird, endet mit dem Ende der Zeile. Für beide werden Sie im Laufe dieser Veranstaltung Beispiele sehen.

Kommentare dienen dazu, den Code näher zu erläutern. Dies ist ein Service, den vielleicht Anfänger besonders schätzen werden. Es sollte aber Ihr Ziel sein, den Code so leicht verständlich zu machen, dass kein Kommentar erforderlich ist.

Eine weitere Art von Kommentaren sind Javadoc-Kommentare. Diese beginnen mit „/**“ und enden mit „*/“. Sie werden wie Java-Kommentare nicht vom Java-Compiler verarbeitet. Im Unterschied zu anderen Kommentaren werden sie aber von einem Programm namens *javadoc* verarbeitet. Dieses Programm erzeugt aus den Javadoc-Kommentaren HTML-Dateien. Diese sind nach einem Standard-Schema untereinander verlinkt. Weitere Verweise können mittels Javadoc- und HTML-Syntax eingefügt werden (siehe Kap. 13). Ich verwende in allen mehr oder weniger fertigen Beispielprogrammen, die ich Ihnen zur Verfügung stelle, diese Art von Kommentaren.

Aus Eclipse heraus führen Sie das Programm *javadoc* z. B. über *export* → *javadoc* aus. Einzelheiten finden Sie im Anhang B.

Bemerkung 4.3.1 (Arten von Kommentaren)
Aufgrund einiger Restriktionen von *Javadoc* mag man die C-artigen Kommentare mit „/* ... */“ oder „//“ weiterhin als sinnvoll ansehen. Sie werden eine kleine Anwendung in Eclipse in Kap. 21 kennenlernen.

Ich versuche in meinem selbstgeschriebenen Java-Code weitestgehend nur *Javadoc* zu verwenden. Ich versuche Kommentare zu vermeiden, die eher der Arbeitsorganisation dienen. Für einen Hinweis, was noch getan oder verbessert werden muss, verwende ich die Möglichkeiten der Entwicklungsumgebung. In Eclipse gibt es dazu z. B. Task-Tags.

Sie sollten sich angewöhnen mindestens einen Javadoc-Kommentar für jede Klasse zu schreiben, der die Aufgabe der Klasse kurz beschreibt und den Autor oder die Autoren nennt. ◄

Üblicherweise beginnt eine Java Klassendatei mit einem Kommentar im C-Stil, der das Copyright etc. enthält. Darauf können Sie im Praktikum meistens verzichten.

Eine Klasse gehört zu einem Paket. Paket heißt in Java wie in UML *package*. Ein *package* kann in einer *package-info*.java-Datei dokumentiert werden, dies ist aber nicht notwendig. Ein Paket entspricht einfach einem Verzeichnis (innerhalb einer Verzeichnishierarchie). Es ist üblich, die Paketnamen an den Domain-Namen der Entwickler zu orientieren, aber in umgekehrten Reihenfolge, also beginnend mit der Top-Level-Domain.

Beispiel 4.3.2 (Paketstruktur)
Für Code, den ich in meiner Funktion als Professor an der HAW schreibe, sollte ich das also z. B. in einem solchen Paket tun: *de.haw-hamburg.informatik.kahlbrandtbernd* o. ä. tun. Code den ich „privat“ schreibe in *de.kahlbrandt.bernd* etc. ◄

Die erste nicht-Kommentar Zeile in einer .java-Datei ist meist ein *package*-Statement. Enthält eine Klassendatei kein *package*-Statement, so liegt die Datei direkt im aktuellen Source-Verzeichnis und nicht in einem Unterverzeichnis. Vermeiden Sie dies bitte!

Hier eine *package-info*.java-Datei, wie ich sie in allen meinen Paketen verwende:

```
/**
 * Dieses Paket enthält alle Code-Beispiele aus dem Kapitel 3: Klassen des Skripts.
 * @author Bernd Kahlbrandt
 *
 */
```

Die Deklaration einer Klasse beginnt mit dem Schlüsselwort *class*. Davor stehen noch weitere Informationen, wie z. B. die Sichtbarkeit (accessibility), etwa *public* (öffentlich).

Definition 4.3.3 (Sichtbarkeit (accessibility))
In Java gibt es vier Stufen der *Sichtbarkeit* von Elementen:

öffentlich (public) Öffentliche Elemente sind von allen anderen Elementen aus sichtbar und benutzbar. Sie werden durch das Schlüsselwort *public* gekennzeichnet. Klassen, Interfaces (damit auch Annotationen), Attribute, Konstruktoren und Methoden können *public* sein.

geschützt (protected) Geschützte Elemente sind von allen Elementen des Pakets und von spezialisierten Elementen aus sichtbar und benutzbar. Attribute, Methoden, Konstruktoren und (innere) Klassen können geschützt (*protected*) sein.

privat (private) Private Elemente sind nur von dem Element (also Objekten der Klasse) aus sichtbar und benutzbar. Attribute, Methoden und Konstruktoren können privat sein, ebenso innere Klassen.

Paket Elemente mit Paket (package) Sichtbarkeit sind von allen Elementen des Pakets aus sichtbar und benutzbar. Elemente haben Paket-Sichtbarkeit, wenn keine andere angegeben ist. Alle Elemente können package Sichtbarkeit haben.

Für Sichtbarkeit werden auch die Begriffe Zugriffsschutz und accessibility verwendet. ◄

Die Wahl der Sichtbarkeit ermöglicht es Ihnen, das Geheimnisprinzip (Kapselung) in der Java-Programmierung umzusetzen, siehe Abb. 1.5, ganz so, wie Sie es gerade benötigen.

Bemerkung 4.3.4 (Sichtbarkeit restriktiv wählen)
Wählen Sie die Sichtbarkeit Ihrer Elemente so *restriktiv* wie möglich. Das bedeutet:

- Attribute sind *private*, wenn es keinen Grund gibt eine andere Sichtbarkeit zu wählen. Nur konstante (*final*) Attribute können ohne Bedenken als *public* deklariert werden.
- Methoden sind oft *public*. Es gibt aber fast immer Hilfsmethoden, die *private* oder *protected* sind.

◄

Bemerkung 4.3.5 (Sichtbarkeit)
Aufgrund vieler Anfängerfehler, die ich beobachtet habe, ist mir nun folgende Metapher für Sichtbarkeit eingefallen: Wenn sie alle Attribute mit *package* Sichtbarkeit deklarieren, so ist dass so, als wenn Sie zu Hause immer nackt herumlaufen würden. Deklarieren Sie diese *protected,* so ist das so, als wenn Sie innerhalb der Familie immer nackt herumlaufen. Deklarieren Sie alle Attribute *public*, so ist das so, als wenn Sie immer nackt herumlaufen. ◄

Klassennamen werden als Substantive im Singular gebildet. In Java werden sSie in *UpperCamelCase* gebildet. Ausnahmen von Klassennamen im Singular werden Sie bald kennenlernen.

Der Name einer Klasse beginnt also mit einem Großbuchstaben. Hier der Rumpf einer noch nutzlosen Klasse:

```
public class Counter{
}
```

Wie man die geschweiften Klammern {} setzt, ist auch Geschmackssache:

- Die öffnende Klammer { als letztes Zeichen in der Zeile des Statements, die schließende Klammer } auf eine eigene Zeile in die Spalte, an der das Statement beginnt, wie in obigem Beispiel. Dies hat den Vorteil, dass man eine Zeile spart.
- Öffnende und schließende Klammer in jeweils eine Zeile in der gleichen Spalte, wie in dem unteren Beispiel. Dies hat den Vorteil, dass der durch die Klammern umschlossene Bereich auf einen Blick zu erkennen ist.

```
public class Counter
{
}
```

Die Java Code-Konventionen von Oracle und Google empfehlen die erstere Variante. Sie finden diese Konventionen im Internet und für alle Fälle auch im pub. Bitte halten Sie sich daran!

Bemerkung 4.3.6 (Klassenname)
Der einfache Name obiger Klasse ist *Counter*. Sie finden Sie in verschiedenen Versionen (*CounterVnn* im Paket *counter*. Der vollständige oder vollqualifizierte Name ist daher *de.hawh.kahlbrandt.counter.Counter*. ◀

Definition 4.3.7 (Block)
Ein Block beginnt mit { und endet mit }. ◀

Ein Klasse definiert also insbesondere einen (benannten) Block.

Direkt vor dem Beginn der Klassendefinition kann die Klasse mit einem Javadoc-Kommentar beschrieben werden, siehe Kap. 13. Ich erwarte mindestens eine Angabe wie diese:

```
/**
 * Eine einfache Klasse zur Einführung in Java.
 *
 * Es wird ein {@link #value Wert} hoch- bzw. heruntergezählt.
 * @author Bernd Kahlbrandt
 *
 */
public class Counter {
   ...
}
```

Nach der Klassendeklaration (die erste Zeile) folgen die Klassenattribute, absteigend nach Sichtbarkeit. Diese werden in Java durch das Schlüsselwort *static* gekennzeichnet. Ein Beispiel sehen Sie auf S. 48 für die Klasse *InstanceCounter*.

Als nächstes folgen die (Instanz-) Attribute. Wie die Klassenattribute werde auch die (Instanz-) Attribute nach Sichtbarkeit geordnet angegeben, von *public* über *protected* und *package* bis *private*.

Als nächstes folgen die *Konstruktoren*, ebenfalls nach Sichtbarkeit geordnet. Konstruktoren dienen der Erzeugung von Objekten von Klassen.

Ein Konstruktor hat den gleichen Namen wie die Klasse. Wird für eine Klasse *Fu kein* Konstruktor definiert, so generiert Java einen *default*-Konstruktor ohne Parameter *public Fu()*.

Nach den Konstruktoren folgen die *Methoden* in einer logischen Reihenfolge, unabhängig von ihrer Sichtbarkeit.

Neue Objekte werden mit dem *new*-Operator erzeugt:

```
Counter counter = new Counter();
```

Wird ein Konstruktor — sei es nun mit oder ohne Parameter definiert — so generiert Java **keinen** *default*-Konstruktor.

Ist ein Konstruktor mit Parametern definiert, so führt ein Aufruf eines Konstruktors ohne Parameter zu einem Fehler, da Java in diesem Fall keinen generiert (siehe Beispiel *basic/Table.java* im Projekt *Fehler*).

Das ist auch gut und richtig so:

1. Da in dieser Situation kein Konstruktor ohne Parameter definiert wurde und auch keiner generiert wurde ist die Fehlermeldung: „The constructor Table() is undefined“ völlig korrekt.

2. Die Fehlermeldung ist auch sinnvoll: Wenn der Programmierer keinen Konstruktor ohne Parameter zur Verfügung stellt, so heißt dies: Ohne Parameter, sozusagen aus dem Nichts, kann kein Objekt erzeugt werden.

Konstruktoren können überladen werden (siehe Def. 1.3.21). Um redundanten Code zu vermeiden, werden Sie in solchen Fällen Konstruktoren gegenseitig aufrufen wollen (*Constructor Chaining*). Das geht mittels des Schlüsselwortes *this* wie in dem folgenden Codeausschnitt:

```
public class Counter {
    private int value;
    private final int resetValue;
    ...
    public Counter(){
        this(0);
    }
    public Counter(int value){
        this.value = value;
        this.resetValue = value;
    }
    ...
}
```

Der default Konstruktor ohne Parameter ruft hier den anderen Konstruktor mit dem Parameterwert 0 auf. Wird Code in verschiedenen Konstruktoren benötigt, so kann man ihn in einen sogenannten Objektinitialisierungsblock auslagern. Dieser Block wird intern an den Anfang jedes Konstruktors kopiert. Ein solcher Initialisierungsblock steht in geschweiften Klammern { ... } am Anfang der Klassendefinition, ggf. nach einem etwaigen statischen Initialisierungsblock. Ein statischer Initialisierungsblock wird durch das Schlüsselwort *static* gekennzeichnet. Ein solcher wird beim Laden der Klasse ausgeführt. Für weitere Einzelheiten verweise ich auf Abschn. 9.3.

Als letztes folgen die Methoden in einer logisch sinnvollen Reihenfolge.

Bemerkung 4.3.8 (Java-Code Empfehlungen)
Die hier gegebenen Empfehlungen für die Reihenfolge von Elementen in einer Java-Klassendatei folgen den Empfehlungen von Oracle und Google. Sie werden aber feststellen, dass auch in den mit Java ausgelieferten Klassen gegen manche dieser Konventionen verstoßen wird. Das hat verschiedene Gründe, z. B. das sich die Standards erst später herausgebildet haben und das es auch gute Gründe gibt, dagegen zu weilen zu verstoßen. ◀

4.4 Attribute, Konstruktoren und Methoden

Klassen enthalten Attribute (Fields, Felder), Methoden und Konstruktoren. Bei Klassen spricht man in vielen Umgebungen von Attributen, in Java ist der Begriff Feld (engl. field) üblich. Ich verwende durchgehend den Begriff Attribut. Manchmal unterscheidet man zwischen den Begriffen *Methode* und *Operation*. Dies geschieht vor allem, wenn man zwischen der Schnittstelle und der Implementierung unterscheiden möchte oder muss (siehe auch Abschn. 4.6). Dann bezeichnet *Operation* den Kopf oder die Signatur der Methode, also den Namen, die Parametertypen und den Rückgabetyp (vgl. Def. 1.3.15 auf Seite 5).

Wie eine Klasse eine Sichtbarkeit hat, haben auch alle Elemente einer Klasse eine Sichtbarkeit. Diese wird in Java über *access modifiers* deklariert.

- Ist keine Sichtbarkeit spezifiziert, so ist die Sichtbarkeit *package*. Alle Elemente des Pakets dürfen auf das Element zugreifen.
- Ist ein Element *public*, so dürfen alle Elemente beliebiger Pakete auf das Element zugreifen.
- Ist ein Element *protected*, so können nur Elemente von Klassen dieses Pakets und von Unterklassen der Klasse auf dieses Element zugreifen.
- Ist ein Element *private*, so können nur Elemente der Klasse darauf zugreifen.

Auch die Sichtbarkeit kann man zur Signatur zählen.

Methoden können *überladen* werden. Das heißt, es kann zwei Methoden geben, die den gleichen Namen haben, sich aber in den Typen oder der Anzahl der Parameter unterscheiden. Sichtbarkeit und Rückgabetyp können nicht zum Überladen herangezogen werden.

Bemerkung 4.4.1 (Überladen)
Sowohl beim Überladen von Konstruktoren als auch beim Überladen von Methoden wird zur Compile-Zeit entschieden, welche Variante verwendet wird. In einem ersten Schritt werden alle zugänglichen und anwendbaren Konstruktoren bzw. Methoden ausgewählt. Im zweiten Schritt wird der bzw. die spezifischste ausgewählt. Weniger spezifisch heißt dabei: Akzeptiert jeden Parameter den auch der bzw. die andere akzeptiert. [BG05], Puzzle 46. ◄

Bei der Definition von Interfaces lässt man die Sichtbarkeit weg, da dort nach Definition nur *public* Methoden vorkommen. Seit Java 9 gibt es auch private Methoden in Interfaces, die auch mit private gekennzeichnet werden.

Java kennt zwei Arten von Datentypen zwischen denen es wichtige Unterschiede gibt: Sogenannte primitive Typen und sog. Referenztypen. Ein wichtiger Unterschied besteht darin, dass ein primitiver Typ keine Objektidentität besitzt, Objekte von Referenztypen aber Objektidentität besitzen. Die primitiven Typen sind *boolean*, *byte*, *short*, *int*, *long*, *float*, *double*, *char*. Alle anderen Typen sind Referenztypen.

Für Programmierer ist es besonders wichtig, die Unterschiede zwischen primitiven Typen und Referenztypen zu kennen.

Für Variable gilt:

Primitiver Typ Variable enthält den *Wert*.

Referenztyp Variable enthält eine „Referenz auf“ das *Objekt*.

Bei primitiven Typen werden Werte übergeben (*call by value)*, bei Referenztypen werden Referenzen auf Objekte übergeben. Haben Sie eine Operation *op(Typ fu)*, so wird in Abhängigkeit davon, ob *Typ* primitiv oder eine Klasse ist unterschiedlich verfahren:

Primitiver Typ (call by value) Wird ein solcher Parameter übergeben, so wird eine Kopie des Werts an die Methode übergeben. Nur diese kann in der Methode verändert werden. Der ursprüngliche Wert kann nicht verändert werden.

Referenztyp (call by reference) Wird ein solcher Parameter übergeben, so wird ebenfalls eine Kopie übergeben, aber nun eine Kopie einer Referenz (Adresse) auf ein Objekt. Elemente dieses Objekts können nun innerhalb der Methode verändert werden und diese Änderungen sind auch außerhalb der Methode wirksam. Die Änderungen geschehen in der Regel durch ändernde Methoden.

Ein einfaches Beispiel hierfür finden Sie in *basic.CallByValueAndByReference*.

Bemerkung 4.4.2 (Begriff call by reference)
Die hier angegebene Beschreibung von *call by reference* entspricht dem Sprachgebrauch in Java (und einigen anderen Programmiersprachen). Im Compilerbau (siehe etwa [ASU86, AS85] wird *call by reference* anders definiert. Es wird eine Referenz (keine Kopie) übergeben. Diese kann in der Methode (oder Funktion, je nach Programmiersprache) verändert werden. In C++ wird können Sie sowohl *call by value* als auch *call by reference* spezifizieren. Hier ein Beispiel aus [Str13], p. 316:

```
void f(int val, int& ref)
{
    ++val;
    ++ref;
}
```

Ein Parameter mit `typ&` wird *by reference* übergaben. Hier inkrementiert `++val` eine lokale Kopie des ersten Arguments während `++ref` das zweite Argument inkrementiert, keine Kopie. Die in Java durchaus übliche Begriffsbildung ist also zumindest für Compilerbauer irreführend: Es ist in deren Sinne immer *call by value*. ◄

Primitive Typen sind keine Klassen und werden von der Programmiersprache vorgegeben, wie *int, float, double, boolean, char*. Viele Referenztypen kommen bereits mit Java, wie etwa *Integer*, *BigInteger*, *Queue* oder *String* u v. a. m., oder werden vom Programmierer geschrieben, wie etwa die Klassen *CounterVn*, $n = 01, \ldots$, siehe *counter.CounterVn.java*. Zu jedem primitiven Typ gibt es eine zugehörige sogenannte Wrapper-Klasse, die sich vom Typ zunächst dadurch unterscheidet, das der erste Buchstabe groß geschrieben ist. So ist die Wrapper-Klasse zum primitiven Typ *float* die Klasse *Float*. Die Objekte der Wrapperklassen kapseln den Wert des entsprechenden primitiven Typs. Sie können sich ja schon einmal Teile des Sourcecodes dieser Klassen, z. B. *Integer* ansehen. Ich werde später darauf zurückkommen.

Die Wrapper-Klassen stellen viele nützliche Methoden zur Verfügung. In *CounterV04View* habe ich bereits eine Klassenmethode der Wrapper-Klasse *Integer* verwendet: *Integer.toString*.

Einzelheiten der numerischen Typen werden in Kap. 7 behandelt.

Attribute können als *final* deklariert werden. Dies hat die Konsequenz, das der Wert dieses Attributs nur einmal zugewiesen und später nicht mehr verändert werden kann. Handelt es sich um einen primitiven Typ, so handelt es sich damit um eine Konstante. Der Name steht dann für den Wert und nicht für eine Variable. Handelt es sich um einen Referenztyp, so kann das referenzierte Objekt geändert werden, dass Attribut verweist aber stets auf das ursprünglich zugewiesene Objekt. Weitere Anwendungen von *final* folgen später in diesem Kapitel.

Attribute einer Klasse dürfen nicht mit lokal innerhalb einer Methode oder eines Blocks definierten Variablen verwechselt werden. Es gibt verschiedene Strategien oder Konventionen, um dies möglichst sicher auszuschließen.

Die eine Möglichkeit ist die Verwendung von *this*. Dieses Schlüsselwort bezeichnet eine Referenz auf das aktuelle Objekt. Verwendet man konsequent *this.attributName*, so kann es nicht passieren, dass man ein Attribut der Klasse mit einer lokalen Variablen verwechselt. In manchen Sprachen, wie z. B. PHP ist man auf *$this->* sogar explizit angewiesen [HV04]. Konsequent *this* zu verwenden hat den weiteren Vorteil, dass man für Parameter den gleichen Namen, wie für Attribute wählen kann und trotzdem klar lesbaren Code erhält.

Eine andere Möglichkeit besteht darin, Attribute mit speziellen Namenskonventionen zu belegen. Aus C++ stammt die Variante, dass Attribute mit einem Unterstrich beginnen, also z. B. *_attributName*. Manche Programmierer verwenden eine solche linkspolnische Notation auch für weitere Zwecke. Sie lassen etwa die Namen von Parametern mit einem *p* beginnen. Das könnte z. B. so aussehen:

```
public void eineOperation(String pattributName}{
    _attributName = pattributName;
}
```

„Linkspolnische Notation“ leitet sich dabei von einer mathematischen Schreibweise ab. Hier werden Abkürzungen wie „_, p“ als Präfix verwendet, um Eigenschaften von Elementen zu kennzeichnen. In mathematischen Formeln wird die Bezeichnung verwendet, wenn der Rechenoperator vor die Operanden geschreiben wird, also z. B. + a b, statt a + b. Entsprechend ist rechtspolnische oder umgekehrte polnische Notation definiert.

Ich bevorzuge die Verwendung von *this* gegenüber einer linkspolnischen Notation. Man kann aber auch die Ansicht vertreten, beides sei überflüssig, da Entwicklungsumgebungen, wie Eclipse Attribute bereits farbig hervorheben, z. B. durch Blau. Dies hilft aber auch nicht in jedem Fall. So werden Namenskonflikte und Verwechselungen damit nicht zuverlässig genug vermieden.

Aber **Achtung**: Auf den Folien spare ich das *this* u. U. ein, um Platz zu sparen! Das sollten Sie zu Ihrer eigenen Sicherheit nicht tun!

Bevor Variablen verwendet werden können, müssen sie deklariert werden. Bei Klassen geschieht dies durch Deklaration in der Klasse. Dies sind die Attribute. In Methoden werden lokale Variable deklariert. Dies sollte so spät wie möglich und am Anfang eines Blocks erfolgen, nämlich dann, wenn sie benötigt werden. Dann müssen sie aber auch initialisiert werden.

Es gibt einen wichtigen Unterschied zwischen Attributen und lokalen Variablen: Attribute werden automatisch mit sinnvollen Werten initialisiert:

byte, short, int, long, float, double	0	numerisch Null
Wrapper:		Wie primitive
boolean	false	logisches falsch
Objekt-Typ	null	typunabhängiger Wert

Der Wert *null* ist dabei ein typunabhängiger Wert, der angibt, dass das Element keinen Wert hat. Es wird als *null-Literal* bezeichnet. Jeder Zugriff auf ein mit *null* initialisiertes Objekt führt zu einer *NullPointerException*. Dies wird wahrscheinlich zu Beginn Ihr häufigster Fehler in der Java-Programmierung sein. Mehr zu Exceptions finden Sie in Kap. 12. Lokale Variablen innerhalb einer Methode werden nicht automatisch initialisiert und müssen deshalb unbedingt explizit initialisiert werden! Manche Umgebungen tun dies zwar automatisch, es ist aber nicht durch die Spezifikation garantiert. In Java ist es üblich, Variablen am Anfang eines Blocks zu deklarieren. Das habe ich bereits am Anfang des Abschnitts 4.3 für Attribute erwähnt. Es gilt aber auch für lokale Variablen.

Außer den bisherigen Attributen, die für jedes Objekt der Klassen einen eigenen Wert haben, gibt es auch Klassenattribute, die einen Wert für die Klasse haben, d. h. es gibt einen Wert, der von allen Objekten der Klasse genutzt wird. Klassenattribute und -methoden (siehe Def. 1.3.27) werden in Java durch das Schlüsselwort *static* gekennzeichnet.

Ein Beispiel gibt die Klasse *InstanceCounter*. In der Source-Datei sind die Elemente dokumentiert.

```
/**
 * Diese Klasse illustriert Klassenattribute und die finalize Methode.
 * @author Bernd Kahlbrandt *
 */
public class InstanceCounter {
    static int instances;
    int count;

    public InstanceCounter() {
        instances++;
    }

    public void increment() {
        count++;
    }

    public void decrement() {
        count--;
    }

    public int show() {
        return count;
    }
    public void finalize(){
        instances--;
    }
}
```

Diese Klasse hält in dem Klassenattribut *instances* fest, wie viele *Counter*-Objekte es gibt, bzw. genauer, wieviele erzeugt wurden. Sie werden später lernen, wie man dies auch wieder sicher herunterzählen kann. Die hier verwendete *finalize*-Methode hat ihre Tücken, dazu später mehr (siehe auch [Blo08], Item 7). Finalizer sollten Sie also vermeiden! Hier deshalb nur soviel: Wenn der *Garbage-Collector* ein *InstanceCounter*-Object beseitigt, wird die *finalize*-Methode aufgerufen, die hier *instances* herunterzählt.

Siehe hierzu auch die Klasse *CounterV05* im Paket *counter*, die in Kap. 6 in Abschn. 6.10 ausführlich beschrieben wird.

Außer Instanzmethoden, die für ein Objekt einer Klasse aufgerufen werden können, gibt es auch Klassenmethoden, die ohne ein Objekt der Klasse aufgerufen werden, vgl. auch Def. 1.3.27. Wie bei Klassenattributen wird dies durch das Schlüsselwort *static* gekennzeichnet. Eine Klassenmethode *fara()* einer Klasse *Fu* wird durch *Fu.fara()* aufgerufen.

Während Instanzmethoden sehr wohl nicht nur auf Instanzattribute sondern auch auf Klassenattribute zugreifen können, ist es verboten, mit Klassenmethoden auf Instanzattribute zuzugreifen, ein Beispiel zeigt *programmierfehler/basic/Queue.java*

Das ist auch sinnvoll: Klassenmethoden und -attribute *gehören* zu der Klasse, Instanzmethoden und -methoden *gehören* zu einem Objekt der Klasse. Klassenmethoden und -attribute können (unabhängig von der Sichtbarkeit) von allen Objekten der Klasse genutzt werden. Umgekehrt ist der Zugriff nicht zweckmäßig, wenn nicht sogar gefährlich und deshalb vom Compiler verboten.

Definition 4.4.3 ((im)mutable)
Eine Klasse heißt *immutable*, wenn die Attribute ihrer Objekte nur bei der Erzeugung der Objekte gesetzt werden können und später nicht mehr verändert werden können. Sind Attribute eines Objekts nach der Erzeugung veränderbar, so nennt man die Klasse *mutable*. ◀

Da Objekt in vielen Fällen ein mehr oder weniger langes „Leben“ haben, mögen Sie sich fragen, warum es überhaupt sinnvoll sein sollte, eine immutable Klasse zu verwenden. Nehmen wir aber einfach einmal an, wir hätten eine immutable Klasse. Dann haben wir davon einige Vorteile und als Programmierer weniger Arbeit:

- Immutable Objekte sind einfach zu erstellen, zu testen und zu verwenden.
- Sie sind automatisch *thread-save*, es gibt also keine Synchronisierungsprobleme, wenn mehrere Nutzer gleichzeitig zugreifen (siehe Kap. 24).
- Die Methode *clone* aus *Object* muss nicht überschrieben werden.
- Die Methode *hashCode* kann *lazy initialization* verwenden. Das heiß: Der Hash-Code muss erst berechnet und gespeichert werden, wenn die Methode *hashCode* aufgerufen wird.
- Klasseninvarianten, d. h. Attribute, die nach der Erzeugung nicht mehr verändert werden können, können bei der Erzeugung etabliert werden und müssen später nie mehr überprüft werden.
- Es wird kein *Copy-Konstruktor* benötigt.

Beispiel 4.4.4 (immutable)

1. *Strings* sind immutable, ebenso *Arrays* der Länge 0 . Dies ist das zunächst wichtigste Beispiel. Sie können daher ein *Array* nicht einfach nachträglich „verlängern“. Verketten Sie zwei *Strings* a und b durch +, wie in folgendem Code-Schnipsel,

   ```
   System.out.println(a+b);
   ```

   so wird für *a+b* ein neues *String*-Objekt erstellt.
2. Alle Wrapper-Klassen für die primitiven Typen sind immutable (Siehe 7.3).
3. Die veraltetete Klasse *Date* ist mutable.
4. Die neuen Datums- und Uhrzeitklassen sind immutable, wie z. B. *LocalDate*.

◀

Bemerkung 4.4.5 (String Details)
String-Konstanten und *String*-Literale, die gleich im Sinne von *equals* sind, sind auch identisch ([GJS+11], 15.28). Um dies zu gewährleisten hat die Klasse *String* einen Pool verwendeter *String*-Konstanten. Siehe hierzu auch die Methode *intern* der Klasse *String*. ◀

Klassen haben Konstruktoren. Selbst wenn Sie keinen geschrieben haben, erzeugt der Compiler einen default Konstruktor, d. h. einen Konstruktor ohne Parameter. Es gibt in Java aber im Unterschied zu etwa C++ keinen Destruktor. Für das Zerstören von Objekten bzw. die Freigabe von Speicherplatz ist der Garbage Collector verantwortlich. Er gibt Speicherplatz von Objekten frei, die nicht mehr referenziert werden. Setzen Sie eine Referenz auf *null*, so gibt es auf jeden Fall eine Referenz auf das Objekt weniger. Gibt es keine Referenz auf das Objekt mehr, so kann der Speicherplatz freigegeben werden.

Die Klasse *System* hat eine Klassenmethode *gc()*, in deren Kurzbeschreibung es heißt:

> Runs the garbage collector.

Sieht man genauer hin, so lautet die Beschreibung:

> Calling the gc method suggests that the Java Virtual Machine expend effort toward recycling unused objects in order to make the memory they currently occupy available for quick reuse. When control returns from the method call, the Java Virtual Machine has made a best effort to reclaim space from all discarded objects.

Diese Formulierung erinnert mich an die Anfangsszene des zweiten Teils von Quentin Tarantinos Kill Bill:

> Bea: Will you be nice?
> Bill: I've never been nice my whole life, but I will do my best to be sweet.
> ...
> Bea: You promised to be sweet.
> Bill: I said „I'll do my best", that's hardly a promise.

System.gc ruft *Runtime.getRuntime().gc()* auf und diese Methode ist *native*, das heißt Plattformspezifisch in etwa C oder C++ implementiert.

4.5 Pakete (Packages)

Pakete (packages) sind das Strukturierungsmittel für Java-Anwendungen. Zu welchem Paket eine Klasse gehört entscheidet sich durch den Ordner, in dem die *.java*-Datei liegt. In der Datei erkennen Sie das Paket an dem Befehl „*package paketname;*".

Geben Sie kein Paket an, also kein *package ...* statement, so liegt die Klasse im default-Paket im aktuellen Verzeichnis. Vermeiden Sie dass bitte. Selbst wenn Sie nur etwas ausprobieren wollen, ist es besser dies innerhalb eines Pakets zu tun. Sie können sich für „Wegwerf-Code" ja ein Paket wie „scratch" oder „dev0" anlegen.

Bemerkung 4.5.1 (Dummies)
Die Bezeichnung *dev0* kommt aus Unix und bezeichnet dort ein Gerät (Device) das alle Schreibzugriffe verwirft. In DOS/VSE hieß (oder heißt) das sysout dummy und in z/OS DEV.NULL. ◀

Beschreibungen für Pakete können Sie in einer *package-info.java* Datei unterbringen, sieh Kap. 13.

Die Namen von Paketen werden üblicherweise in Kleinbuchstaben gebildet. Es gibt einige Ausnahmen von dieser Regel bei gängigen Abkürzungen, z. B. . *org.omg.CORBA*. Sie werden hierarchisch wie Web-Domains gebildet, aber in der umgekehrten Reihenfolge. Zum Beispiel so: *de.hawh. kahlbrandt.basic*.

Ein Paket definiert einen *Namensraum*: Klassennamen müssen innerhalb eines Pakets eindeutig sein. Insgesamt werden Klassennamen durch den vollständig qualifizierten Namen eindeutig. So kann es in verschiedenen Paketen Klassen mit gleichem „einfachen" Namen geben. Durch die Paketnamen werden die vollständig qualifizierten Namen eindeutig, wie in *de.hawh.kahlbrandt.basic.Clazz*.

4.6 Schnittstellen (Interfaces)

Eine Schnittstelle heißt in Java Interface und enthälten vor allem öffentliche Methoden. Für den Aufruf aus default-Methoden können Sie auch private Methode schreiben, die also nicht von „außen“ sichtbar sind und Ihnen bei der Struktierung Ihres Codes helfen. Ein Interface ist also eine Sammlung öffentlicher Methoden. Deshalb wird das Schlüsselwort *public* hier weggelassen. Außerdem können Schnittstellen Klassenattribute enthalten. Diese sind dann automatisch *final*. Es handelt sich de facto also um Konstanten. Viele Methoden in Interfaces sind *abstrakt*. Das heißt, sie haben keine Implementierung (keinen Block mit Code), sind also Operationen. Eine *default*-Methode hat eine Implementierung. Implementierende Klassen müssen also *default*-Methoden nicht überschreiben, wenn die default-Implementierung für sie geeignet ist. Ferner kann ein Interface Klassenmethoden enthalten, allerdings nur, wenn diese *default*-Methoden sind.

Die Methoden von Schnittstellen werden mittels Javadoc dokumentiert. Hier ein einfaches Beispiel aus der Klausur TI2PR2 im WS 2009/10.

```
package wsmmix;

import java.util.Iterator;
import java.util.NoSuchElementException;


/**
 * Dieses Interface deklariert ein schmales Listeninterface für eine sortierte
 * Liste. Deshalb erhalten die {@link #insert} und die {@link #delete} Operation
 * auch nur ein Objekt des Typs E.
 * @author Bernd Kahlbrandt
 *
 */
public interface Liste<E extends Comparable<E>> extends Iterable<E>{
/**
 * Diese Operation fügt das Objekt element an der ersten Stelle ein, an der
 * das vorhergehende Element kleiner oder gleich und das nachfolgende Element
 * größer oder gleich element ist (im Sinne von
 * {@link java.lang.Comparable#compareTo})
 * @param element Das einzufügende Element
 */
void insert(E element);
/**
 * Diese Operation entfernt das erste Element x, bei dem x.compareTo(element)
 * gleich 0 ist. Gibt es kein solches Element, so wird eine
        * java.util.NoSuchElementException geworfen
        *
 * @param element
 */
void delete(E element) throws NoSuchElementException;
/**
 * Diese Operation liefert einen Iterator zurück, der am Anfang der Liste steht.
 */
Iterator<E> iterator();
/**
 * Liefert true, wenn die Liste leer ist, false andernfalls.
 * @return true, wenn die Liste leer ist, false andernfalls.
 */
boolean isEmpty();
```

```
}
```

Das Interface *List* aus dem Paket *java.util* hat sehr viel mehr und auch default Methoden.

Ein Interface definiert — wie auch eine Klasse — einen Typ. In vielen Fällen ist es sinnvoll Variablen mit einem Interface als Typ zu deklarieren. So kann man dann leicht die konkrete implementierende Klasse bei Bedarf ändern. Dies ist insbesondere — aber auch sonst — für die Java-Collection-Klassen empfehlenswert.

4.7 Generalisierung und Spezialisierung

Das Schlüsselwort für die GenSpec-Beziehung — kurz Vererbung — zwischen Klassen ist in Java *extends*. Dabei gibt es einen Unterschied zwischen Klassen und Interfaces. Ein Interface kann beliebig viele andere Interfaces spezialisieren („extenden“). Eine Klasse kann genau eine andere Klasse direkt spezialisieren. Selbst, wenn kein *extends* angeben wird, ist die Klasse aber immer Unterklasse von *Object*. Die GenSpec-Beziehung zwischen Klassen und Interfaces wird über das Schlüsselwort *implements* spezifiziert. Eine Klasse kann beliebig viele Interfaces implementieren.

Da *implements* wie *extends* eine GenSpec-Beziehung spezifiziert ergibt sich eine weitere Eigenschaft von Java: Ein *Interface* definiert einen *Typ*. Jede implementierende Klasse kann also verwendet werden, um Objekte vom Typs des Interface zu erzeugen (abgesehen einmal von abstrakten Klassen).

Innerhalb einer Vererbungshierarchie können Methoden überschrieben werden. Es wird dann jeweils die Methode ausgeführt, die „am nächsten“ an dem Typ des Objekts ist.

Ein einfaches Beispiel liefert die Klasse *CounterV06*. In dieser Methode überschreibe ich die Methode *toString* aus der Klasse *Object*:

```
   @Override
   public String toString(){
      return "Zählerstand: " + this.count;
}
```

„@Override“ gibt dem Compiler einen Hinweis auf die Absicht, das hier eine Methode einer Oberklasse überschrieben wird. Dieser Hinweis, wird von Eclipse und anderen Entwicklungsumgebungen bei konsequenter, zeitsparender Arbeitsweise automatisch eingefügt. Er hilft Anfängern Fehler zu vermeiden und das Konzept wird in Kap. 20 erläutert.

Nun kann ich bei Bedarf Zählerobjekte einfach ausgeben, etwa so

```
CounterV05 counterV05 = new CounterV05(42);
CounterV06 counterV06 = new CounterV06(4711);
System.out.println("Ohne Überschreiben von toString:" + counterV05);
System.out.println("Mit Überschreiben von toString: " + counterV06);
```

mit folgender Ausgabe auf der Konsole:

```
Ohne Überschreiben von toString:counter.CounterV05@addbf1
Mit Überschreiben von toString: Zählerstand: 4711
```

In der Klasse *CounterV05* wird die Methode *toString* nicht überschrieben. Die Methode *println* macht aber aus dem Objekt counterV05 einen String. Dazu wird die Methode *toString* aus der Klasse *Object* verwendet, da *CounterV05* keine eigene *toString*-Methode hat, sondern nur die ererbte. Gemäß der Java API Dokumentation, liefert diese Methode

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

Vor dem @-Zeichen steht der voll qualifizierte Klassenname, nach dem @-Zeichen der hash-Code in hexadezimaler Darstellung. In Der Klasse CounterV06 habe ich die *toString*-Methode überschrieben. Diese wird nun aufgerufen und es wird „Zählerstand: 4711“ ausgegeben.

Eine Klasse, die als *final* spezifiziert wird, kann nicht spezialisiert werden.

Beispiel 4.7.1 (final Klasse)
Einige Klassen, die Sie oft verwenden werden sind *final*: *Integer*, *Double*, ...◀

Eine Methode, die als *final* spezifiziert ist, kann nicht überschrieben werden.

Für Klassenmethoden macht Überschreiben keinen Sinn: Diese werden ja mit Klassenname.Methodenname aufgerufen. Was passiert, wenn Sie versuchen eine Klassenmethode zu überschreiben sehen Sie in den Klassen *Parent* und *Son* im Projekt Programmierfehler, Paket *basic*. Wird die Annotation @Override nicht verwendet, so handelt es sich einfach um eine Klassenmethode in der Unterklasse, wie im Beispiel *public static int getNumberOfChildren()*. Wird die Annotation @Override verwendet, wie in *@Override public static String getName()*, gibt es einen Compiler-Fehler.

Hat eine Klasse eine Oberklasse, so können Sie als ersten Befehl im Konstruktor den Konstruktor der Oberklasse aufrufen. Dazu dient das Schlüsselwort *super*, etwa so:

```
public counterV05(int start){
   super();
   ...
}
```

In vielen Fällen sind mehrere Konstruktoren nützlich. Ein Beispiel haben Sie schon in Kap. 2 gesehen: Die Klasse *Bruch* hat mehrere Konstruktoren. Um keinen Code doppelt schreiben zu müssen, können mittels *this(...* andere Kontruktoren aufgerufen werden. Dies bezeichnet man als *constructor chaining*.

4.8 Die Klasse Object

Da jede Klasse eine Unterklasse von *Object* ist, hat auch jede Klasse die Operationen von *Object*. Von den elf Methoden von *Object* dienen fünf dem Einsatz bei paraller Verarbeitung. Diese werden in Kap. 24 behandelt.

Ich beginne hier mit der Methode *toString*. Diese liefert eine Darstellung des Objekts als Zeichenkette (String). In *Object* ist sie so implementiert:

```
public String toString(){
  return getClass().getName() + '@' + Integer.toHexString(hashCode())
}
```

Das @-Zeichen dient der Trennung und den *Hash-Code* erläutere ich weiter unten.

Die Methode *getClass* liefert den vollqualifizierten Namen der Klasse, d. h. mit der Pakethierarchie, wie in *java.lang.Object*. Weitere wichtige Methoden in diesem Zusammenhang werden im Kapitel 19 behandelt.

Es wird empfohlen, dass *jede* Klasse die Methode *toString* überschreibt. Folgen Sie dieser Empfehlung, so können Sie die Informationen beeinflussen, die Sie in vielen Situationen, z. B. im Fehlerfalle bekommen. Für die tatsächliche Ausgabe von Informationen über ein Objekt in einer praktisch sinnvollen Situation, werden Sie aber in vielen Fällen nicht auf *toString* zurückgreifen. Ich verzichte deshalb in vielen Beispielen darauf *toString* explizit anzugeben.

Es ist eine gute Faustregel, beim Überschreiben von *toString* so zu verfahren, dass von einem Objekt und der String-Darstellung hin- und hergeschaltet werden kann. Um von der String-Darstellung wieder zu dem entsprechenden Objekt zu kommen, können Sie eine entsprechende Fabrikmethode oder einen entsprechenden Konstruktor schreiben. Beispiele hierfür finden Sie in den Wrapper-Klassen, und in *BigInteger* bzw. *BigDecimal*.

Überschreiben von *toString* macht aber die Verwendung einer Klasse durchaus einfacher. Mögen Sie sich auf ein Format festlegen, in dem *toString* ein Objekt darstellt, so können Sie auch eine *Fabrikmethode* anbieten, die aus diesen *String*s ein entprechendes Objekt erstellt. Das Format geben Sie in der API-Dokumentation von *toString* an und verweisen auf die *Fabrikmethode*. So können Programmierer dann zwischen der String-Darstellung und der Object-Darstellung hin- und

herschalten. Eine solche Entscheidung legt Ihnen aber auch ein Korsett für die weitere Entwicklung der Klasse an.

Die Methoden *equals* und *hashCode* spielen eine wichtige Rolle. *equals* vergleicht zwei Objekte. In *Object* ist *equals* so implementiert:

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

Gemäß dieser Implementierung sind zwei Objekte *gleich*, wenn Sie *identisch* sind. Das ist in vielen Fällen sinnvoll. In vielen Fällen aber auch nicht. Dann werden Sie *equals überschreiben*.

Wird *equals* überschrieben, so muss auch *hashCode* damit konsistent überschrieben werden. Die Methode *hashCode* liefert eine ganze Zahl, die für das Objekt charakterisch ist. Sie soll folgenden Kriterien genügen:

- Wird die Methode *hashCode* innerhalb der Laufzeit einer Java-Anwendung mehrfach für das gleiche Objekt aufgerufen, so liefert *hashCode* immer die gleiche Zahl, sofern sich das Objekt zwischendurch nicht verändert hat.
- Sind zwei Objekte gemäß *equals* gleich, so muss *hashCode* für beide Objekte das gleiche Ergebnis liefern.
- Es ist nicht zwingend, das *hashCode* für Objekte, die gemäß *equals* verschieden sind, verschiede Werte liefert. Es wird aber dringend empfohlen.

Eine gute Methode *hashCode* ist insbesondere wichtig, wenn Java-Objekte in eine Hash-Tabelle eingetragen werden, siehe *HashMap, Hashtable*. Sind *equals* und *hashCode* nicht konsistent implementiert, so werden Sie Objekte in einer *HashMap* etc. nicht wiederfinden!

Im Vorgriff auf fortgeschrittenen Stoff (z. B. Algorithmen und Datenstrukturen) gebe ich hier einige Hinweise zur Implementierung von *hashCode*, die ich aus [Blo08], Item 9, übernommen habe.

1. Sie beginnen mit eine ganzen Zahl, vorzugsweise einer Primzahl. Bewährt haben sich insbesondere Zahlen die einen größeren Primzahlzwilling haben, also z. B. die Zahl 17 und speichern diese in einer Variablen *result*.
2. Für jedes Attribut *a* berechnen Sie aus dem Wert bzw. Objekt einen Hashcode und addieren ihn auf *result*:
   - 2.1. Für *boolesche* Attribute: hashcode = a?1:0, also 1 für *true*, 0 für *false*.
   - 2.2. Für Attribute a vom Typ *byte*, *char*, *short*, *int*: hashcode = (int)a.
   - 2.3. Für Attribute a vom Typ *long*: hashcode = `(int) (a ^ (a >>> 32))`. `^` ist das bitweise „entweder oder“, XOR.
   - 2.4. Für Attribute a vom Typ *float*: hashcode = Float.floatToIntBits(a).
   - 2.5. Für Attribute a vom Typ *double*:
     hashcode = Double.doubleToLongBits(a),
     hashcode = `(int)(hashcode ^(hashcode>>> 32))`.
   - 2.6. Für Referenztypen:
     - 2.6.1. Überschreibt die Klasse *equals* so, dass *equals* für Komponenten rekursiv aufgerufen wird, so rufen Sie *hashCode* ebenfalls rekursiv auf.
     - 2.6.2. Ist das Attribut *null*, so geben Sie 0 zurück.
     - 2.6.3. Sind die Verhältnisse komplexer, so berechnen Sie einen Wert von einem der oben genannten primitiven Typen und hashen diese.
   - 2.7. Für Arrays behandeln Sie jedes Element des Arrays entsprechend.
3. Für jedes Attribut bilden Sie result = result*31 + hashcode.

4. Abgeleitete Attribute brauchen nicht berücksichtigt zu werden.

Mittels der Beziehung $31 * i = (i << 5) - i$ können Sie das auch effizient implementieren. Allerdings nehmen nach [Blo08] manche neueren JVMs diese Optimierung automatisch vor.

Bemerkung 4.8.1 (Dokumentation der hashCode-Methode)
Aus zwei Gründen sollten Sie ihre Implementierung der *hashCode*-Methode nicht nach außen offen legen:

1. Wenn Nutzer Ihrer Klasse sich darauf einstellen, dass der Hash-Code immer genauso berechnet wird, können Sie die Methode später nicht verbessern und nicht einmal verändern, wenn Sie die Klasse ändern.

2. Wenn die Implementierung der Methode *hashCode* bekannt ist, kann jemand versuchen gezielt verschiedene Objekte zu erzeugen, die den gleichen Hash-Code Wert haben. Wird mit den Objekten dann in einer *HashMap* o. ä. gearbeitet, so kann das sehr langsam werden.

◀

Damit haben Sie nun etwas über die Methoden

- *getClass*,
- *equals*
- *hashCode* und
- *toString*

gelesen bzw. gehört. Für die Methoden *clone* und *finalize* verweise ich auf das Kap. 9. Deren Tücken sind mit Ihrem bisherigen Wissen noch nicht sinnvoll zu erläutern.

Weitere Methoden zum Umgang mit Objekten beliebiger Art finden Sie in der Utility-Klasse *Objects*.

Will man Objekte einer Klasse gemäß einer Ordnung vergleichen, so benötigt man dafür eine Operation. Eine Operation, die eine Klasse implementieren muss, wird über ein Interface spezifiziert. Für den Vergleich gibt es zwei geeignete Interfaces: *Comparable* (aus *java.lang*) und *Comparator* (aus *java.util*). Beide sind generisch definiert (siehe Kap. 18), d. h. Sie geben bei Ihrer Verwendung an, Objekte welcher Klasse verglichen werden können.

Beide habe nur jeweils eine Operation *compareTo* bzw. *compare*.

Beispiel 4.8.2 (Comparable und Comparator)
Sollen Objekte einer Klasse nach einer festen Regel miteinander verglichen werden können, d. h. ob das eine größer, gleich oder kleiner als das andere ist, so deklarieren sie die Klasse so:

```
public class Fu implements Comparable<Fu>{
   ...

   @Override
   public int compareTo(Fu fu){
      //Vergleiche this mit Fu
      ...
   }
```

Wollen Sie die Objekte der Klasse *Fu* nach einem anderen Kriterium sortieren, so definieren sie sich eine weitere Klasse:

```
public class FuComparator implements Comparator<Fu>{
   @Override
   public int compare(Fu fu1, Fu fu2 ){
     //Vergleiche fu1 mit fu2
      ...
   }
```

Anschliessend können Sie wahlweise eine der beiden *sort*-Methoden aus der Klasse *Collections* aufrufen:

```
  public sortiere(List<Fu> fuList){
  Collections.sort(fuList);//sortieren gemäß compareTo
  Collections.sort(fuList,new FuComparator());
  }
```

◀

Die Methoden *equals* und *compareTo* hängen wie folgt zusammen:

1. Wenn *equals* implementiert wird, so muss sichergestellt sein, dass dadurch eine Äquivalenzrelation auf den von *null* verschiedenen Objekten der Klasse definiert wird.
2. Wenn *compareTo* implementiert wird, muss auch *equals* implementiert werden und beide Implementierungen müssen konsistent sein, d. h. es gilt:

$$x.equals(y) == true \Longleftrightarrow x.compareTo(y) == 0;$$

3. Wenn *hashCode* implementiert wird so muss gelten:

$$x.equals(y) == true \Longrightarrow x.hashCode() == y.hashCode()$$

Die Umkehrung wird nicht gefordert. Es ist aber zu empfehlen, dass für *ungleiche* Objekte, also solche, bei denen *equals false* liefert, auch verschiedene Hashcodes erzeugt werden. Dies verbessert die Performance von Hashtables. Für die Einzelheiten und Schwierigkeiten verweise ich auf die Vorlesung über Algorithmen und Datenstrukturen.

Bei der Implementierung muss man dies berücksichtigen. Ich zeige dies nun an einigen Beispielen.

Beispiel 4.8.3 (compareTo und equals)
Ich betrachte eine Klasse Fu, die *Comparable* implementiert. Diese besitzt dann eine Methode

```
int compareTo(Fu fu)
    ...
```

Um *equals* konsistent mit *compareTo* zu implementieren gibt es grundsätzlich nur eine akzeptable Möglichkeit:

```
@Override
boolean equals(Object o){
return o instanceof Fu?this.compareTo((fu)o)==0:false;
}
```

In manchen Fällen lässt sich mit etwas mehr Code eine gewisse Performanceverbesserung erreichen. Der Kern bleibt aber immer so wie hier. ◀

Hierbei lernen Sie auch eine Anwendung des *instanceof*-Operators kennen: Dieser prüft die Klasse des Objects, dass zu diesem Zeitpunkt von der Variablen referenziert wird. Für Einzelheiten verweise ich auf Kap. 19. Sehen wir uns nun den Zusammenhang von *equals* und *hashCode* an Beispielen an.

Beispiel 4.8.4 (Counter mit Comparable)
Die Klasse *CounterComparable* spezialisiert *Counter* und implementiert *Comparable*.

```
public class CounterComparable extends Counter implements Comparable<Counter> {
    @Override
    public int compareTo(Counter o) {
        return Integer.compare(this.show(),o.show());
    }
    @Override
    public boolean equals(Object o){
        return o instanceof Counter? this.compareTo((Counter)o)==0:false;
    }
}
```

Die Methode *compareTo* muss konsistent mit *equals* implementiert werden, so wie hier geschehen.
◀

In Beispiel 4.8.4 kommt ein weiterer Operator zum Einsatz: der *instanceof*-Operator. Der erste Operand ist ein Objekt, der zweite eine Klasse. Der *instanceof*-Operator liefert *true*, wenn der erste Operand nicht null und von der als zweitem Operand angegebenen Klasse ist, andernfalls *false*. Die Wahrheitstabelle dieses Operators ist dementsprechend:

Objekt	Klasse	Objekt instanceOf Klasse
null	beliebige	false
nicht null Klasse	Klasse oder Oberklasse	true
nicht null Klasse	andere Klasse nicht Oberklasse	false

Ich verweise ausdrücklich auf Kap. 19, Stichwort *reifiable*.

Nun zum zweiten Interface mit einer Vergleichsmethode: *Comparator*. Nach Ihrem bisherigen Kenntnisstand können Sie auf die Idee kommen, dieses ebenfalls in der jeweiligen Klasse zu implementieren:

Beispiel 4.8.5 (Counter mit Comparator)
Die Klasse *CounterComparator* spezialisiert *Counter* und implementiert *Comparator*

```
public class CounterComparator extends Counter implements Comparator<Counter> {
   @Override
   public int compare(Counter c1, Counter c2) {
      return c1.show() - c2.show();
   }
   @Override
   public boolean equals(Object o){
      return o instanceof Counter? this.compare(this,(Counter)o)==0:false;
   }
}
```

◀

Das Beispiel 4.8.5 zeigt aber nicht die typische Verwendung von *Comparator*. Meist werden Sie *Comparator* verwenden um ein solches Objekt einer Methode, wie etwa *sort* in der Klasse *Collections* als Parameter zu übergeben, wie in Bsp. 4.8.2.

Hier nun eine Übersichtstabelle über *compareTo* und *equals*:

equals	*compareTo*
Jede Klasse	Klassen, die Comparable implementieren
Rückgabetyp boolean	Rückgabetyp int
Parametertyp *Object*	Parametertyp Klasse
...	...

Abb. 4.1: Vergleich von *compareTo* und *equals*

4.9 Anwendungsbeispiel: Elementare Verschlüsselung

Angeregt durch ein Beispiel aus [RSSW06a], S. 347–355 betrachte ich folgende Aufgabenstellung.

Eine Nachricht, gegeben als einfacher Text-String, soll verschlüsselt werden, damit er gefahrlos an einen Partner übertragen werden kann, ohne dass dieser vom Inhalt Kenntnis bekommt. Das Verschlüsselungsverfahren soll flexibel gewählt werden können.

Überlegen wir uns zunächst, welche Klassen wir dafür verwenden können oder wollen. Ich liste hier einige mögliche Ideen auf und diskutiere sie dabei gleich.

1. Unter Anfängern sehr beliebt sind Klassenmethoden (*static*). Die können aus einer main-Methode aufgerufen werden und man kann so alles in einer Klasse schreiben. Das ist aber ganz und gar nicht flexibel oder wiederverwendbar. Man erhält riesige, nur einmal verwendbare Klassen. In der hier vorliegenden Form könnte man aber so argumentieren: Das Verschlüsseln von Strings ist eine Funktion, die nichts mit einem bestimmten String zu tun, hat also keine Instanzmethode. Man bilde also eine Utility-Klasse *Codes*, die für jedes Verschlüsselungsverfahren je eine statische *encode*- und *decode*-Methode zum ver- bzw. entschlüsseln besitzt.

   Dieses Vorgehen hat Nachteile: Es liegt in der Verantwortung der Nutzer der Klasse zu einer *encode*-Methode die korrekte *decode* Methode aufzurufen. Die Schnittstelle der Klasse ändert sich mit jedem neue hinzugefügten Verfahren. Diese Nachteile sind bereits so gravierend, dass ich diese Idee verwerfe.

2. Kann man die erste Idee weiterentwickeln: Ich definiere ein Interface *Coder* mit den Operationen *encode* und *decode*:

```
package codes;
public interface Coder {
    String encode(String s);
    String decode(String s);
}
```

Nun können wir Schritt für Schritt Klassen schreiben, die diese Operationen implementieren. Zur Eingewöhnung verwende ich als erstes Beispiel die Geheimsprache der Weißen Rosen aus den Kalle Blomquist Büchern von Astrid Lindgren. Auf Seite 12 in [Lin61] findet sich die Beschreibung: Die Vokale a, e, i ,o, u bleiben erhalten, jeder Konsonant k wird durch kok ersetzt. Da diese Geheimsprache vor allem gesprochen bzw. gesungen wurde, kommt es auf Groß- und Kleinschreibung nicht an. Den Code finden Sie im Paket *codes* in der Klasse *KalleCoder*

4.10 Historische Anmerkungen

Klassen sind ein sehr nützliches Organisationsprinzip. Es gibt aber auch andere Organisationsprinzipien. Ein älteres Prinzip organisiert Code in Programmen und Unterprogrammen. Namen hierfür variieren je nach Programmiersprache. So werden Begriffe wie Prozedur, Funktion, Routine, uvam. verwendet.

Grundsätzlich steckt hinter jedem Programmierparadigma eine Zerlegungsstrategie. In einer objektorientierten Programmiersprache erfolgt die Zerlegung in Klassen. Für mehr hierzu verweise ich auf Bücher und Veranstaltungen zum Softwareengineering.

Die Unterscheidung der Begriffe „Methode“ und „Operation“ verwischt sich nach meinem Eindruck zunehmend. So gibt es seit Java 8 *default-Methoden* in Interfaces. In Java kann ein Interface also inzwischen auch implementierte und nicht nur abstrakte Methoden enthalten. Insofern nehme ich an, dass sich der Begriff „Methode“ durchsetzten wird, differenziert in „konkrete Methoden“ und „abstrakte Methoden“.

4.11 Aufgaben

1. ([0]) Wie werden Variablen vom Typ *long*, *float*, *double*, *char* initialisiert?
2. ([2]) In Beispiel 4.8.3 wurde eine mit *compareTo* konsistente Implementierung von *equals* angegeben. Begründen Sie bitte, warum dies im Wesentlichen die einzige akzeptable ist!
3. ([5]) Hier folgt eine Methode einer Utility-Klasse, die die Fibonacci-Zahlen rekursiv berechnet:

   ```
   public static long fibRec(long n) {
       return (n==0)?1:((n==1)?1:(fibRec(n-1)+ fibRec(n-2)));
   }
   ```

   (Achtung: In dieser Methode erfolgt keine Fehlerbehandlung.) Schreiben Sie bitte eine entsprechende Methode, die zusätzlich die Anzahl Aufrufe zählt, die zur Berechnung der n-ten Fibonacci-Zahl benötigt werden!
4. ([03]) Bringen Sie bitte die folgenden Begriffe in eine logische Reihenfolge! Erläutern Sie Ihr(e) Ordnungskriterien! Attribut, Klasse, Konstruktor, Methode, Paket.
5. ([03]) Wie wird in Java Objektidentität sichergestellt?
6. ([03]) Wie werden String-Literale in Java verwaltet?
7. Aus Aufgabe 11 kennen Sie sieben Fragen, wie Objekte einer Klasse identifiziert werden können. Geben Sie bitte an, wie sie das in Java und ggf. mit bzw. ohne Berücksichtigung von *Comparable* implementieren wollen.

Kapitel 5

Basiskonstrukte

5.1 Übersicht

In diesem Kapitel stelle ich die wichtigsten Java Sprachkonstrukte systematisch vor. Deren Darstellung ergänze ich um Erläuterungen zur Verwendung ausgewählter wichtiger Methoden einiger „Standard"-Klassen. Ich beginne mit einer Darstellung der zulässigen Zeichen in Namen von Klassen, Interfaces, Attributen, Operationen, lokalen Variablen und einer Übersicht der reservierten Worte. Da Ihnen diese Klassen immer wieder „über den Weg laufen werden" stelle ich auch die primitiven Typen und ihre Wrapperklassen vor.

5.2 Lernziele

Die folgenden Dinge sollten Sie nach Lektüre dieses Kapitels beherrschen:

- Die formalen und üblichen Regeln für die Namen von Bezeichnern sicher beherrschen.
- Die Operatoren in Java beherrschen.
- Schleifen-Konstrukte in Java sicher beherrschen.
- Verzweigungen (if, switch, ternärer Operator) beherrschen.
- Einige Ausgabeoperationen kennen.
- Wissen, wie Sie Datum und Uhrzeit bekommen.
- Einige Systemvariablen, wie Pfad, User kennen.

5.3 Deklaration, Zuweisung, Kommentar und Ausdruck

Ausdrücke, die mit einem Semikolon („ ; ") abgeschlossen werden bilden eine Anweisung. Beispiele hierfür aus Kap. 4 sind z. B. die Deklarationen von Attributen oder Paketen. In diesen Deklarationen von Klassen, Attributen oder Paketen haben Sie auch schon Bezeichner gesehen. Der Name eines Pakets, einer Klasse, eines Attributs oder einer Methode ist ein Bezeichner, wie auch die Namen von Parametern. Die Regeln für zulässige Namen sind in der Java Sprachspezifikation [GJS+14] (rekursiv) definiert:

- Ein gültiger *Bezeichner* (*Identifier*) ist definiert als eine *IdentifierChars* Zeichenfolge, aber *kein reserviertes Wort*, *boolesche Literal* (*true* oder *false*) oder das *Null Literal null*.
- *IdentifierChars* ist rekursiv definiert als *JavaLetter* gefolgt von *IdentifierChars JavaLetterOrDigit*.

Die neuen Elemente sind *JavaLetter*: jedes Unicode-Zeichen, dass ein *JavaLetter* ist und *JavaLetterOrDigit*: jedes Unicode-Zeichen, das ein *JavaLetterOrDigit* ist. Der zweite Teil der Definition besagt u. a., dass Java-Bezeichner mit einem Buchstaben und keiner Zahl beginnen.

Ob ein Unicode-Zeichen ein *JavaLetter* etc. ist, bekommen Sie rein technisch durch den Aufruf von *Character.isJavaIdentifierStart(char c)* bzw *Character.isJavaIdentifierPart(char c)* heraus (siehe Abschn. 5.4).

Schreiben Sie eine Klasse, die diese Methoden aufruft und die Werte ausgibt, so sind dort eine ganze Reihe von Zeichen dabei, die auf der Konsole nicht darstellbar sind. Was Sie sehen, hängt in Eclipse von den Einstellungen ab, siehe Ende von Abschn. B.5. Aber viele können Sie erkennen, z. B. :

- Ein JavaLetter ist keine Zahl.
- Ein JavaLetterOrDigit ist kein Rechensymbol, als kein `+,-,*,.,/,%`, etc.
- Ein JavaLetterOrDigit ist kein Vergleichs- bitweiser oder boolescher Operator, also kein `<,>,!,&,|` etc.
- Ebensowenig können [,], {, }, ; oder , etc. als Bestandteile von Java-Bezeichnern verwendet werden.

Die reservierten Worte (Keywords) in Abb. 5.1 finden Sie u. a. in der Java Sprachspezifikation [GJS+14]. Beachten Sie aber bitte, dass die Literale *null* und die booleschen Literale *true* und *false* in dieser Liste nicht erscheinen, da sie keine Worte sondern eben Literale sind. Die reservierten Worte *const* und *goto* werden bisher nicht verwendet.

Sie haben oben schon gesehen, dass weder „/“ noch „*“ in Bezeichnern vorkommen können. Beide Zeichen werden in Java auch verwendet, um Kommentare zu charakterisieren.

`//` Einzeiliger Kommentar. Geht von // bis zum Ende der Zeile.

`/*` Mehrzeiliger Kommentar. Geht von /* bis */. Es könnte sich also auch um einen Teil einer Zeile handeln.

`/**` Javadoc-Kommentar. Geht bis zum nächsten */. Kommentare dieser Art werden von dem Programm Javadoc verarbeitet und nach html umgewandelt.

Alle diese Arten von Kommentaren haben ihre Berechtigung. Die mit // eingeleiteten heißen nach einer Herkunft *C-Stil*-Kommentare. Auch die mehrzeiligen Kommentare sind bereits aus *C++* bekannt.

Bemerkung 5.3.1 (Kommentare in Beispielen)
Ich beschränke mich in den Java-Klassen, die ich schreibe und Ihnen zur Verfügung stelle, weitgehend auf Javadoc-Kommentare. ◄

Die Einzelheiten von Javadoc präsentiere ich in hoffentlich verdaubaren Häppchen an verschiedenen Stellen in der Vorlesung und zusammengefasst in Kap. 13.

Damit haben Sie wesentliche Basis-Elemente von Java bereits kurz gesehen. Was die reservierten Worte und nicht-Java-Zeichen genau bedeuten, wird in den nächsten Abschnitten beschrieben. Abbildung 5.1 zeigt noch zusammengefasst die reservierten Worte in Java: Hinzu kommen noch *false* und *true*. Dies sind aber keine reservierten Worte sondern boolesche Literale, ebenso wie *null* das *null*-Literal ist.

5.4 Primitive und Referenztypen

Wie bereits in Abschnitt 4.4 erwähnt, kennt Java zwei Arten von Datentypen zwischen denen es wichtige Unterschiede gibt: Sogenannte primitive Typen und sog. Referenztypen. Ein wichtiger

abstract	continue	for	new	switch
assert	default	if	package	synchronized
boolean	do	goto	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

–

Abb. 5.1: Die reservierten Worte (keywords) in Java

Unterschied besteht darin, dass ein primitiver Typ keine Objektidentität besitzt, Objekte von Referenztypen aber Objektidentität besitzen.

Jedes Objekt hat eine Methode *hashCode*. Diese Operation muss konsistent mit *equals* implementiert werden. Nach Java API Dokumentation bedeutet dies folgendes:

- Wird *hashCode* mehrfach während der Laufzeit einer Java-Anwendung aufgerufen, so liefert die Methode den gleichen ganzzahligen Wert. Einzige zulässige Aussnahme: Das Objekt wurde zwischenzeitlich so verändert, dass es gemäß *equals* nicht mehr als das gleiche Objekt gilt.
- Wenn zwei Objekte gemäß *equals* gleich sind, so muss *hashCode* für beide den gleichen Wert liefern.
- Es ist *nicht* notwendig, dass *hashCode* für ungleiche Objekte verschiedene Werte liefert.

Als erstes Beispiel für das Zusammenspiel von *equals* und *hashCode* verwende ich die sog. Wrapperklassen. Die Wrapperklassen kapseln primitive Typen und stellen für diese Operationen zur Verfügung. Diese Wrapperklassen sind aus folgenden Gründen notwendig:

- An manchen Stellen benötigt Java einen Referenztyp und kann nicht mit einem primitiven Typ arbeiten. Ein Beispiel ist die Verwendung von primitiven Typen in generischen Typen (siehe Kap. 18) Diese müssen mit der Wrapperklasse und nicht dem primitiven Typ parametrisiert werden.
- Auch für Variablen von primitivem Typ werden (manchmal) Methoden benötigt, die diese nicht haben können. Dann „springt“ eben ein Objekt einer der Wrapperklassen ein.

Eine Übersicht der primitiven Typen mit ihren Wrapperklassen und Wertebereiche finden Sie in Abb. 7.2.

Beispiel 5.4.1 (Hashcode und Wrapperklassen)
Da generische Typen keine Objektidentität haben, wird man erwarten, dass sich dies auch in den Objekten von Wrapperklassen zeigt. Zu erwarten ist, das Objekte von Wrapperklassen, die den gleichen primitiven Wert kapseln gleich sind und den gleichen Hashcode liefern. Die Klasse *HashCode01* illustriert dies für *Integer*, *Float*, *Double Character* und *Boolean*. ◄

Ein Blick in den Sourcecode bestätigt diese Beobachtung.

```
public final class Integer extends Number implements Comparable<Integer> {
    ...
    private final int value;
    ...
```

```
    public int hashCode() {
        return value;
    }
    ...
}
```

Wenn möglich übernimmt Java die Umwandlung von einem primitiven Typ in eine Objekt einer Wrapperklasse (boxing) oder von einem Objekt einer Wrapperklasse in einen primitiven Typ (unboxing) automatisch. Diesen Prozess nennt man auto(un)boxing. Wie nützlich dies ist können wahrscheinlich nur Programmierer nachvollziehen, die Java auch vor Version 5 vom Herbst 2004 verwendet haben. Die Klasse *basic.BoxingExample* zeigt einige der Effekte, die Ihnen das Programmieren erleichtern, den Code einfacher und verständlicher machen.

- Haben Sie eine Variable als primitiven Typ deklariert, so können sie einer Variablen zuweisen, die als Objekt einer Wrapperklasse deklariert ist: Hier findet autoboxing statt.
- Das geht aber auch umgekehrt: Hier findet autounboxing statt.
- Sie können die arithmetischen Operationen, also `+`, `-`, `+=` etc. auf Objekte von Wrapperklassen anwenden oder auch auf Kombinationen von primitiven Typen und Objekten von Wrapperklassen. Je nach dem findet hier also boxing, unboxing etc. je nach Kombination automatisch statt. Der folgende Code funktioniert deshalb und produziert auch keine Compiler-Warnung.

  ```
  Integer iw;
  Integer jw=2;
  int ip = 1;
  iw = ip++;//autoboxing
  iw++; //autounboxing
  iw+=jw;
  iw = jw + ip;
  ip = jw - iw;
  ```

  Sie finden ihn im Paket *basic* als *BoxingExample*.

Das Autoboxing bzw. -unboxing funktioniert heißt aber nicht, dass Sie davon einfach unbedacht Gebrauch machen sollten. Es hat schon seinen Grund, dass es eine Compiler-Option gibt, die bei Boxing bzw. Unboxing Konvertierungen eine Warnung generiert.

Bemerkung 5.4.2 (== und Äquivalenz)
Sie erwarten sicher, dass durch == eine *Äquivalenzrelation* auf int, double, etc. definiert wird. Das ist aber nicht der Fall, siehe Aufgabe 16. ◀

5.5 Befehle, detailliert

In diesem Abschnitt stelle ich die elementaren Programmierkonstrukte vor, dies es in Java gibt.

5.5.1 Schleifen

Es gibt eine Reihe von Schleifen-Konstrukten in Java. Ich beginne mit der *while*-Schleife. Diese hat die Form:

```
while (logischer Ausdruck){
...
}
```

Ein logischer Ausdruck liefert als Ergebnis *true* oder *false*. Ein erstes Beispiel hierfür haben Sie bereits in der Klasse *a00.CounterConsoleView* gesehen. Die „Bedingung“ ist ein Java-Ausdruck, der im Ergebnis wahr (true) oder falsch (false) liefert.

Im einfachsten Fall besteht der logische Ausdruck nur aus einer booleschen Variablen.

```
boolean bedingung=true;
```

Die Schleife lautet dann einfach

```
while (bedingung){
...
}
```

Bemerkung 5.5.1 (Unsinniger Vergleich mit true)
Der folgende Vergleich ist unsinnig:

```
while (bedingung==true){
...
}
```

Die ist eine Art Tautologie: Ist *bedingung true*, so ist der Ausdruck *true*, andernfalls *false*. ◀

Es kann sich aber auch um eine Kombination von booleschen Variablen handeln. Diese können mit den üblichen booleschen Operatoren wie z. B. *und* (&&), *oder* (||) etc. verknüpft werden, siehe Abschn. 5.6. Selbstverständlich können in diesen Ausdrücken auch Operationen verwendet werden. Viele Container-Klassen haben z. B. eine Operation *isEmpty*. So kann ich für eine Queue von Brüchen eine Schleife schreiben, wie die Folgende:

```
Queue<Bruch> myQueue;
 ...
while(!myQueue.isEmpty()){
   ...
}
```

Queue ist dabei ein Interface aus dem Paket *java.util*. In der zweiten Form der Schleife steht die Bedingung nicht am Anfang, sondern am Ende:

```
do{
 ...
}  while(Bedingung)
```

Dies ist also eine Art *until*-Schleife: Führe den Block aus bis *Bedingung* falsch ist. Die *do-while-Schleife* wird auf jeden Fall einmal durchlaufen. Die Bedingung wird erst am Ende eines Durchlaufs ausgewertet. Frühestens am Ende des ersten Durchlaufs wird die Schleife also beendet.

Sehr häufig verwendet wird die *for*-Schleife:

```
for(Initialisierung;Bedingung;Aktualisierung){
   ...
}
```

Das einfachste konkrete Beispiel ist eine Endlosschleife (loop):

```
for(;;){
   ...
}
```

Am häufigsten sehen Sie aber sicher diese Form:

```
  for(int i = 0;i < n;i++){
    // do something
}
```

Wesentlich sind folgende Bestandteile:

Initialisierung Die Variable oder die Variablen müssen initialisiert werden. Das können auch mehrere sein, z. B.

```
for(int i = 0, j = 0;i<j && j < 10;i++,j++){
...
}
```

Mit den initialisierten Werten startet der Schleifendurchlauf, wenn die *Bedingung* wahr ist.

Bedingung Da jede Aussage über die leere Menge ($\emptyset$) wahr ist, läuft die Schleife bei einer leeren Bedingung endlos. Ansonsten wird der Schleifenblock ausgeführt, also alles, was in dem direkt folgenden Paar geschweifter Klammern steht, wenn die Bedingung wahr ist. Ist dies nicht der Fall, so wird das *for*-Konstrukt verlassen. Selbstverständlich kann die Bedingung aus mehreren Operanden bestehen, die durch boolesche Operatoren verknüpft sind.

Aktualisierung Hier werden die Schleifenvariablen aktualisiert. Häufige Varianten sind herauf bzw. herunterzählen mittels postfix-Inkrement bzw. -Dekrement (`i++`, `i--`), siehe Abschn. 5.6.

Um eine weitere Variante des *for*-Konstrukts zu beschreiben, muss ich etwas ausholen: Im Paket *java.util* finden Sie das Interface *Iterator*:

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
}
```

Den Typparameter *E* lernen Sie in Kap. 18 in allen Einzelheiten kennen. Für das Verständnis des *for*-Konstrukts genügt Folgendes: Das Interface Iterator kann mit irgendeinem Typ benutzt werden, also *Iterator<String>*, *Iterator<Bruch>* .... Im Basispaket *java.lang* finden Sie ein weiteres Interface: *Iterable<E>*

```
public interface Iterable<E> {
    Iterator<E> iterator();
}
```

Wenn Sie nun irgendeine Klasse haben, die das Interface *Iterable* implementiert, so können Sie mittels einer *for*-Schleife über deren Elemente laufen: Hier die Klasse *Container*

```
public class Container<Clazz> implements Iterable<Clazz>{
  ...
}
```

und hier ein Beispiel der sogenannten *for each*-Schleife

```
for(Clazz c : Container){
...//tu' was mit c
}
```

In der API-Dokumentation des Interfaces *Iterable* finden Sie bereits viele Klassen, die dieses Interface implementieren.

Die for each-Schleife nutzt ein sehr weit verbreitetes Schema, das *Iterator Muster* (iterator pattern), siehe Abschn. 23.8 in Kap. 23.

Bemerkung 5.5.2 (Wenn möglich for-each!)
Verwenden Sei bitte immer for-each-Schleifen, wenn das möglich ist. Das ist der bevorzugte Stil in Java, einfacher, weniger Fehler-anfällig und oft auch effizienter. ◄

Das Interface *Iterable* hat eine Methode *forEach*. Diese erwartet als Parameter einen *Consumer*. Methoden, die auf den Elementen des Aggregats in der for-each-Schleife ausgeführt werden, können Sie hier als Methodenreferenz oder λ-Ausdruck übergeben.

5.5.2 Kontrollstrukturen

Java kennt folgende Kontrollstrukturen:

if *if-then-else* ermöglicht eine einfache Entscheidung zwischen zwei Möglichkeiten, die sich gegenseitig ausschließen. Hier können beliebige Bedingungen verwendet werden. Ist die Bedingung wahr, so wird der *if*-Zweig durchlaufen, andernfalls der *else*-Zweig.

switch Eine Mehrwegentscheidung. Hier können nur ganzzahlige (also auch Character) Werte (primitiver Typen), *enums* (siehe Kap. 18) oder *Strings* als Parameter für den *switch* verwendet werden. Anschließend könne ausgewählte Werte des Parameters mittels *case* abgeprüft werden. Wird das Konstrukt nicht explizit verlassen (*break*), so werden alle *case*s ab dem ersten „Treffer“ ausgeführt.

Ternärer Operator Der Ternäre Operator hat auf der linken Seite eines „?“ eine boolesche Bedingung und auf der rechten Seite des „?“s zwei, durch „:“ getrennte Anweisungen: Die erste wird ausgeführt, wenn die Bedingung wahr ist, die zweite, wenn sie falsch ist.

```
boolescher Ausdruck?optrue():opfalse();
```

Genaugenommen handelt es sich auch bei einem *try*- und *catch*-Block um eine Kontrollstruktur. Trotzdem behandle ich sie — wie es der Tradition entspricht — im Kapitel 12 im Abschnitt über Exceptions.

Sehen wir uns diese Konstrollstrukturen nun etwas genauer an.

Ich beginne mit dem einfachen *if*.

```
if(Bedingung)
   Anweisung;
```

Bedingung ist dabei ein Ausdruck, der als Ergebnis einen booleschen Wert liefert, also *true* oder *false*. Ich empfehle im Anschluss an das *if* immer einen Block {...} zu verwenden. Dies steht auch so in den Java Code-Konventionen.

In diesem einfachen Fall wird die Anweisung oder werden die Anweisungen ausgeführt, wenn *Bedingung* wahr ist. Andernfalls geht es direkt mit den auf *Anweisung;* bzw. *{...}* folgenden Anweisungen weiter.

Sollen im Fall, dass die Bedingung wahr ist, die Anweisungen in einem Block, andernfalls die in einem anderen Block ausgeführt werden, so lautet das Konstrukt:

```
if(Bedingung){
   //Anweisungen
}else{
  //andere Anweisungen
}
```

Verwenden Sie bitte konsequent Paare von geschweifter Klammern {...}, sonst laufen Sie Gefahr, schwer verständlichen Code zu produzieren: Sie können dann in das *dangling else Problem* laufen. Hier ein Beispiel:

Beispiel 5.5.3 (Dangling else)
Die folgende Methode (Siehe *basic.DanglingElse*) enthält zwei *if*s und ein *else*.

```
/**
 * Welches Ergebnis liefert das Konstrukt für a=1,b=1 und a!=1, b=1 etc.?
 */
public static void ifThenElse(int a, int b){
   System.out.println("Input: a = " + a + " b = " + b);
   if (a == 1)
      if (b == 1)
          a = 42;
   else
     b = 42;
   System.out.println("a = " +a);
   System.out.println("b = " +b);
}
```

Ein nicht ganz aufmerksamer Leser dieses Codes könnte denken, das für $a \neq 1$ der Wert von b auf 42 gesetzt wird. Tatsächlich bezieht sich das *else* aber auf das zweite *if*. Man erhält damit also z. B. folgendes Ergebnis, wenn *ifThenElse* mit den angegebenen Werten aufgerufen wird.

Input		Output	
a	b	a	b
1	1	42	1
2	1	2	1
2	2	2	2

Durch Blöcke { ... } nach *if* bzw. *else* können Sie diesem Risiko wirkungsvoll vorbauen. ◀

Selbstverständlich können *if*-Konstrukte beliebig geschachtelt werden. Das sieht dann z. B. so aus:

```
if(a==b){
   if(b==c){
      if(d==e){
         ...
      }
   }
}
```

Sie können aber auch einfach aufeinander folgen:

```
public void actionPerformed(ActionEvent e) {
                if(e.getActionCommand().equals("about"))
                {
                   (new AboutDialog(c,"Über...")).setVisible(true);
                }
                if(e.getActionCommand().equals("open")){
                   JFileChooser chooser = new JFileChooser();
                   int result = chooser.showOpenDialog(c);
                   if(result == JFileChooser.APPROVE_OPTION)
                    this.c.openFile(chooser.getSelectedFile());
                   //  c.showTree();

}
                if(e.getActionCommand().equals("find")){
                ...
                }
                if(e.getActionCommand().equals("appearance")){
```

```
                c.showOptions();
            }
            if(e.getActionCommand().equals("exit")){
                c.dispose();
            }
        }
```

Haben Sie es mit einer feststehenden Anzahl von Menüpositionen zu tun, so bietet sich oft ein *switch*-Konstrukt an, um die jeweils aufzurufenden Operationen anzusteuern. Dies ist weitgehend unabhängig davon, wie der Benutzer die Auswahl trifft.

Hier ein Beispiel (basic.SwitchMenu):

```
private String [] selections ={"Nichts","Öffnen","Schließen","Suchen"};
private String selection = "Sie haben eine ungültige Auswahl getroffen:";
public static void main(String[] args) {
   SwitchMenu switchMenu = new SwitchMenu();
   int c = switchMenu.getInput();
   switchMenu.switchOnInput(c);
}
private  void switchOnInput(int c) {
   switch(c){
      case 0:
         nichts();
         break;
      case 1:
         oeffnen();
         break;
      case 2:
         schließen();
         break;
      case 3:
         suchen();
         break;
      default:
         this.printSelection(c);
   }
}
```

Das obige Beispiel für aufeinanderfolgende *if*s schreiben Sie mit *switch* so:

```
   public void actionPerformed(ActionEvent e) {
      switch (e.getActionCommand()) {
         case "about": {
            (new AboutDialog(c, "Über...")).setVisible(true);
            break;
         }
         case "open": {
            JFileChooser chooser = new JFileChooser();
               int result = chooser.showOpenDialog(c);
               if (result == JFileChooser.APPROVE_OPTION){
                  this.c.openFile(chooser.getSelectedFile());
                  }
            break;
         }
         case "find": {
```

```
            break;
         }
         case "appearance": {
            c.showOptions();
            break;
         }
         case "exit": {
            c.dispose();
            break;
         }
      }
   }
```

Aufgrund des Kontextes benötigen Sie hier keinen *default*-Ausgang.

Beachten Sie bitte genau die Arbeitsweise des *switch*-Konstrukts: Ist der *switch* Parameter gleich einem Wert in einem *case*-Statement, so wird der Code nach dem Doppelpunkt bis zum nächsten *break*-Statement ausgeführt. ggf. sind das alle folgenden Anweisungen in den folgenden *case*-Statements sowie die nach dem *default*-Ausgang. Das sehen Sie z. B. , wenn Sie in basic.SwitchMenu.java die *break*-Statements auskommentieren.

Als Parameter im *switch* können Variablen folgender Typen verwendet werden:

ganzzahlige numerische primitive Typen: *byte*, *short*, *int*, *long*, *char*.

enums: Hinter diesen stecken letztendlich *int*s.

Strings: String.

Als letztes der Kontrollkonstrukte erläutere ich nun den ternären oder ?-Operator oder conditional operator. Er ermöglich in vielen Fällen eine übersichtliche, kompakte Schreibweise. Ein Beispiel finden Sie schon im bereits verwendeten Beispiel basic.SwitchMenu

```
private int getInput() {
   System.out.println("---Menu---");
   for(int i=0;i<this.selections.length;i++){
      System.out.println(i+" : "+ this.selections[i]);
   }
   Scanner sc = new Scanner(System.in);
   return sc.hasNextInt()? sc.nextInt():0;
}
```

in der vorletzten Zeile: Wenn der Scanner noch eine ganze Zahl zu lesen hat (*sc.hasNextInt()* liefert *true*), so wird diese Zahl gelesen und zurückgegeben, andernfalls wird 0 zurückgegeben. Das Konstrukt

```
Bedingung ? a():b();
```

ist logisch äquivalent zu

```
if(Bedingung){
   a();
}else{
   b();
}
```

Entsprechend den Java Code-Konvention soll eine Methode mit Rückgabe-Typ nicht-*void* möglichst nur ein *return*-Statement haben. Wenn es auch noch der Übersichtlichkeit nützt, ist hier der ternäre Operator das Konstrukt der Wahl. Hier eine typische Anwendung: Implementiert eine Klasse das Interface Comparable, so muss sichergestellt werden, dass *equals* konsistent implementiert wird. Das geht ganz schematisch so:

```
public class Fu implements Comparable<Fu>{
   ...
   public boolean equals(Object o){
      return o instanceof Fu ? this.compareTo((Fu)o)==0 : false;
   }
}
```

equals muss einen Parameter vom Typ *Object* haben, sonst wird die Methode überladen und nicht überschrieben. Durch die Abfrage des Typs von *o* auf *Fu* wird für ein *null* oder nicht-Fu-Objekt *false* zurückgeliefert, für ein Fu-Objekt das Ergebnis von *this.compareTo((Fu)o)==0*.

5.6 Operatoren in Java

Die folgende Tabelle enthält alle Java Operatoren gemäß der Sprachspezifikation. In der Sprachspezifikation werden allerdings „?“ und „:“ als zwei Token gezählt, die gemeinsam den ternären Operator ausmachen. Java hat demnach folgende Operatoren:

Generell	
`=`	Zuweisung
Boolesche Operatoren	
`!`	Negation
`&&`	logisches und
`||`	logisches oder
Relationale Operatoren	
`==`	Vergleich auf gleich
`!=`	ungleich
`<, <=`	Kleiner (gleich)
`>, >=`	Größer (gleich)
Mathematische Operatoren	
`+`	Addition
`++`	pre-/post inkrement
`-`	Subtraktion
`--`	pre-/post dekrement
`*`	Multiplikation
`/`	Division
`%`	Modulo (Rest, Remainder)
`+=`	inkrement um rechte Seite
`-=`	dekrement um rechte Seite
`*=`	Multiplikation mit rechter Seite
`/=`	Division durch rechte Seite
`%=`	Rest bei Division durch rechte Seite
Bitweise Operatoren	
`&`	Bitweises und
`|`	Bitweises inklusives oder
`^`	Bitweises exklusives oder
`~`	Invertierung
`<<`	signed left shift
`>>`	signed right shift
`>>>`	unsigned right shift
`&=`	bitweises $\wedge$ mit rechter Seite

`|=`	bitweises $\vee$ mit rechter Seite
`^=`	bitweises XOR mit rechter Seite
`<<=`	signed leftshift mit Zuweisung
`>>=`	signed right shift mit Zuweisung
`>>>=`	unsigned right shift mit Zuweisung
Ternärer Operator	
`? :`	Bedingung?wenn wahr:wenn falsch
instanceof Operator	
`o instanceof Type`	true, wenn o vom Typ `Type` ist
Cast Operator	
`(Type) o`	Konvertiert o in ein Objekt vom Typ Type, wenn dies möglich ist.
Pfeil eines Lambda-Ausdrucks	
`->`	Bildet Lambda Parameter auf Lambda Body ab.
Method Reference Operator	
`::`	Trennt Objekt bzw. Klasse und Methodennamen

5.6.1 Zuweisungsoperator „=“

Der Zuweisungsoperator weist einer Variablen auf der linken Seite den Wert der rechten Seite zu. Auf der linken Seite („L-value“), kann dabei nicht etwas Beliebiges stehen. So sind dort Konstanten oder Ausdrücke nicht zulässig. Generell ist ein *L-value* etwas, dass auf der linken Seite einer Zuweisung stehen kann ([ASU86]). Auf der anderen, Seite also der rechten, steht das, was zugewiesen werden soll, ein *R-Value*. In der Regel sind *R-values* Werte oder Objekte, während *L-value* Adressen repräsentieren. Nun hat natürlich auch eine Konstante eine Adresse, kann aber aufgrund ihrer Definition nicht auf der linken Seite einer Zuweisung verwendet werden. Auf der rechten Seite sind Konstanten natürlich zulässig.

5.6.2 Boolesche Operatoren

Die Booleschen Operatoren kann man durch ihre Wahrheitstafeln definieren:

a		!a
true		false
false		true
a	b	a&&b
true	true	true
true	false	false
false	true	false
false	false	false
a	b	a||b
true	true	true
true	false	true
false	true	true
false	false	false

Die Negation eines booleschen Ausdrucks ist genau dann wahr, wenn der Ausdruck falsch ist. Das logische *und* von zwei booleschen Ausdrücken ist genau dann wahr, wenn beide Ausdrücke wahr sind. Das logische *oder* zwei booleschen Ausdrücken ist genau dann wahr, wenn mindestens einer der Ausdrücke wahr ist.

Verwechseln Sie aber die booleschen Operatoren && und || nicht mit den bitweisen Operatoren & und |! Bei den booleschen Operatoren && und || verwenden in Java „lazy evaluation“: Ist der erste Operand von && *false*, so ist bereits sicher, dass das Ergebnis *false* ist und der zweite Operand wird nicht mehr ausgewertet. Ganz analog: ist bei || der erste Operand *true*, so ist sicher, dass das Ergebnis *true* ist. Siehe hierzu auch Abschn. 5.6.5 .

5.6.3 Relationale Operatoren

Die aufgeführten relationalen Operatoren haben sehr unterschiedliche Eigenschaften. Bei den Vergleichs-Operatoren == und ! = hängt die Arbeitsweise davon ab, ob es sich um Referenztypen (Objekte) handelt oder um primitive Typen (Werte).

Referenztypen : Hier werden die Referenzen, also die Adressen der Objekte in der JVM verglichen. Wollen Sie also tatsächlich die *Objekte* vergleichen, so bringt Ihnen das gar nichts! Dann müssen Sie die in Abschn. 4.8 beschriebene Methode *equals* verwenden. Für Strings verweise ich außerdem auf Abschn. 9.11.

primitive Typen Bei diesen werden die *Werte* verglichen: Diese liefern bei *boolean*, *byte*, *char*, *int*, *long* das exakte Ergebnis, so wie Sie es aus der Mathematik erwarten. Bei den primitiven Typen *float* und *double* müssen Sie mit Rundungsfehlern rechnen. Einzelheiten dazu finden Sie in Kap. 7.

Beispiele hierzu finden Sie in *basic.RelationalOperators.java*.

Für Referenztypen machen die Operatoren $<, <=, >, >=$ keinen Sinn. Sie sind nur für die primitiven numerischen Datentypen anwendbar.

Für alle diese Operatoren gilt: Tritt als Operand ein Objekt einer Wrapper-Klasse auf, so übernimmt Java automatisch die notwendige Umwandlung. Trotzdem ist empfehlenswert die jeweiligen *compare*-Methoden zu verwenden.

Bemerkung 5.6.1 (Compiler Warnungen)
Sie können dem Compiler mitteilen, wie er sich bei automatischen Konvertierungen von primitiven in Referenztypen und umgekehrt verhalten soll. In Eclipse geht das ganz einfach, siehe Abschn. B.10. „Ignorieren“ ist hier eine gute Wahl, denn da Sie dies bei guter Programmierung im Zweifelsfall nicht vermeiden können, ist das höchstens einmal „nice to know“, wenn Sie genau wissen wollen, wann dies passiert. ◄

Bemerkung 5.6.2 (Rundungsfehler)
Sie studierenden Informatik und nicht numerische Mathematik. Insofern muss ich Sie nicht in der Analyse von Rundungsfehlern schulen. Aber Sie sollten immer im Auge behalten, dass Digital-Rechner nicht mathematisch korrekt rechnen. Insbesondere Assoziativ- und Distributivgesetze gelten nicht immer. Machen Sie z. B. eine Vorbereitung Ihrer Steuererklärung inklusive Umsatzsteuererklärung mittels einer Tabellenkalkulation, so werden Sie sehr schnell zu Rundungsfehlern kommen, die Ihnen das Übertragen in die Steuerklärung etwas erschweren. ◄

5.6.4 Mathematische Operatoren

Die Operatoren „+, -, *, /“ bilden die bekannten Grundrechenarten ab. Sie können für alle primitiven numerischen Datentypen eingesetzt werden. Der „+“-Operator dient darüber hinaus zum Zusammenfügen von *String*s.

Einige Hinweise sind aber auch bei diesen elementaren Operationen notwendig:

1. Die primitiven Datentypen haben begrenzte Wertebereiche. Werden diese verlassen, so kommt es zu Fehlern. Siehe hierzu Kap. 7.

2. Die Division von ganzen Zahlen m und n liefert eine ganze Zahl. Dies ist die größte ganze Zahl q, für die gilt: $|n \cdot q| \leq |m|$. Sie können das aber auch so formulieren:

$$\frac{m}{n} = \lfloor \frac{m}{n} \rfloor.$$

   Das heißt z. B. :

m	n	m/n
1	1	1
32	7	4
-32	7	-4

In der Mathematik eher unüblich ist die Verbindung der Grundrechenarten mit der Zuweisung. Beim Programmieren in *C*, *C++* oder *Java* ist dies aber üblich. Die folgende Tabelle zeigt die Schreibweisen im Vergleich:

Mathematik	Programm
a = a + b	a += b
a = a - b	a −= b
a = a * b	a *= b
a = a / b	a /= b
a = a % b	a %= b

Zwischen der Zuweisung wie $x = x + 1$ und der gleichzeitigen Rechnung und Zuweisung $x+ = 1$ gibt es einen kleinen aber wichtigen Unterschied. Im zweiten Fall erfolgt ggf. eine Typerweiterung oder -einschränkung, siehe [GJS+14], Abschn. 15.26.2. Siehe hierzu auch die Aufgaben 17 und 18 in Abschn. 7.10.

In vielen Fällen werden Variablen um Eins hoch- oder heruntergezählt. Dazu gibt es Inkrement- und Dekrement-Operatoren: + + und - -. Beide gibt es als Prä- und als Post-Variante: Die Prä-Inkrement und -Dekrement-Operatoren werden erst auf den Operanden angewandt und liefern anschließend den (veränderten) Operanden zurück. Die Post-Inkrement und -Dekrement-Operatoren liefern erst den Operanden zurück und verändern ihn dann. Die folgende Tabelle zeigt den Unterschied:

	a=42	
1	System.out.println(- -a);	41
2	System.out.println(++a);	42
3	System.out.println(a- -);	42
4	System.out.println(a);	41
5	System.out.println(a++);	41

Vor der ersten Ausgabe hat die Variable a den Wert 42. Bei der ersten Ausgabe wird a um Eins verringert. Es ist also das gleiche, als wenn Sie geschrieben hätten $a = a-1$. Bereits vor der Ausgabe hat a also den Wert 41 in der nächsten Zeile wird a um Eins erhöht und es wird entsprechend 42 ausgegeben. Anders sieht es in den nächsten beiden Zeilen aus: Der Post-Dekrement-Operator wird erst nach der Ausgabe wirksam. Erst in Zeile 4 wird also der um Eins verringerte Wert 41 ausgegeben. Ebenso in Zeile 5: Dort wird der noch unveränderte Wert ausgegeben und erst anschließend inkrementiert.

Der Modulo-Operator „%“ liefert den Rest, der bei Division von a durch b übrig bleibt. Diese umgangsprachliche Definition ist nicht präzise genug. Bezeichnet man mit $\lfloor x \rfloor$ die größte ganze Zahl, die kleiner oder gleich $x \in \mathbb{R}$ ist, so gilt genauer:

$$a\%b = a - \lfloor \frac{a}{b} \rfloor \cdot b \text{ für } b \neq 0$$

Zur Illustration ein paar Beispiele:

a	b	$\lfloor \frac{a}{b} \rfloor$	a%b
1	1	1	0
32	7	4	4
-32	7	-4	-4

Bemerkung 5.6.3 (Diverse Anmerkungen)
Diese Symbole und Rechenregeln werden Ihnen in anderen Kontexten häufiger begegnen. In [Knu97a] finden Sie zu Dingen wie $\lfloor\ \rfloor, \lceil\ \rceil$ eine Fülle von Material. Der systematische Einsatz

der geeigneten prä- oder post-Inkrement-Operatoren kann Code vereinfachen. Andererseits führt deren unterschiedliche Arbeitsweise bei Anfängern doch ab und an zu Fehlern.

Bei den kombinierten mathematischen Operatoren, wie „+=“ usw. müssen Sie beachten, dass diese ggf. eine erweiternde Konvertierung vornehmen, z. B. von *byte* auf *int*. Also Achtung bei den *shift-Operatoren*, siehe Abschn. 5.6.5 und Kap. 8. ◄

5.6.5 Bitweise Operatoren

Der bitweise Operator *oder*, in Java durch | bezeichnet, vergleicht die Bits Bit für Bit. Seien $a = (a_n \ldots, a_1 a_0)_2$ und $b = (b_n \ldots b_1 b_0)_2$. Die a_i, b_i können also jeweils 0 oder 1 sein. Der Index 2 bedeutet dabei die Darstellung im binären System[1]. Ist der eine Operand kürzer als der andere, so wird er einfach vorne mit Nullen aufgefüllt. Der Operator | liefert also an der Stelle i genau dann den Wert 1, wenn a_i oder b_i oder beide 1 sind, sonst 0.

Entsprechend liefert der Operator & an der Stelle i genau dann 1, wenn a_i und b_i gleich 1 sind, andernfalls 0. Einige auch in Java oft verwendete Anwendungen finden Sie in Kap. 8.

Einige der bitweisen Operatoren verwenden die gleichen Zeichen, wie die ähnlichen booleschen Operatoren. Es gibt aber Unterschiede und mit denen beginne ich diesen Abschnitt. Die booleschen Operatoren arbeiten auf booleschen Operanden, die bitweisen Operatoren arbeiten auf Bit-Strings. Ein boolescher Operand oder ein boolescher Ausdruck sind bzw. ergeben immer entweder wahr (*true*) oder falsch (*false*). Ungeachtet irgendwelcher Fragen der Implementierung in irgendeiner Programmiersprache kann der Wert eines solchen Operanden also durch ein Bit dargestellt werden, meist 0 für *false* und 1 für *true*. Rein logisch ist es also egal, ob sie & oder && bzw. | oder || schreiben.

Damit sich aber fortgeschrittene Programmierer einfacher von Anfängern unterscheiden können (:-)), sind in Java Optimierungen eingebaut: Die Auswertung der booleschen Operatoren wird abgebrochen, wenn das Ergebnis feststeht (lazy evaluation): Ist der Ausdruck a wahr, so wird bei $a||b$ der Ausdruck b gar nicht mehr ausgewertet. Ist der Ausdruck a falsch, so wird bei $a\&\&b$ der zweite Audruck nicht mehr ausgewertet. Die bitweisen Operatoren werten aber immer beide Ausdrücke bis zum „bitteren Ende“ aus.

Handelt es sich bei b um eine boolesche Variable, so passiert nichts weiter. Ist aber b ein Ausdruck, in dem eine Variable verändert wird, so können sich Unterschiede ergeben. Überlegen Sie sich also genau wann Sie was bewirken wollen. Meine Faustregel ist: Will ich boolesche Operatoren, verwende ich diese. Will ich mit Bits arbeiten, verwende ich die bitweisen Operatoren.

Bei den drei *shift-Operatoren* müssen Sie beachten, dass von dem rechten Operanden nur die fünf bzw. sechs niederwertigsten Bits verwendet werden, je nachdem, ob links ein *int* oder ein *long* steht. Die *shift-Distanz* ist also immer im Bereich 0–31 bzw. 0–63.

Mehr über die praktische Anwendung der Techniken zum Umgang mit Bits in Java finden Sie in Kap. 8 und im Zusammenhang mit Algorithmen und Datenstrukturen.

5.6.6 Ternärer Operator

Der ternäre Operator:

```
boolescherAusdruck ? ausdruck1 : ausdruck2;
```

ermöglicht in vielen Fällen eine übersichtliche, knappe Formulierung. Er wird auch als *bedingter Operator*, *Fragezeichen-Doppelpunkt-Operator* etc. bezeichnet. Die Funktionsweise ist einfach: Ergibt die Auswertung des booleschen Ausdrucks vor dem Fragezeichen true, so wird der ausdruck1 zwischen Fragezeichen und Doppelpunkt ausgeführt, ergibt sie false, so der ausdruck2 nach dem Doppelpunkt.

Beispiel 5.6.4 (Ternärer Operator)
Hier ein einfaches Beispiel:

[1] Bis auf das Vorzeichenbit, siehe hierzu Kap. 7.

```
private void print(int n){
   System.out.print("Sie drucken " + n);
   System.out.print(n > 1 ? " Seiten":" Seite" +"\n");
}
```

Hier wird der Plural „Seiten“ ausgegeben, wenn mehr als eine Seite gedruckt wird. ◄

Aufpassen, müssen Sie vor Allem, wenn asudruck1 und ausdruck2 unterschiedliche Typen haben. Die Beschreibung in [GJS+14], §15.25 ist etwas lang. Ich gebe hier nur die wichtigsten drei Punkte an, siehe auch [BG05], Puzzle 8.

1. Haben ausdruck1 und ausdruck2 den gleichen Typ, so ist dies der Typ des ternären Operators. Dies ist die Situation, die Sie anstreben müssen. Alles Andere kann von Übel sein.

2. Hat einer der beiden Ausdrücke ausdruck1 und ausdruck2 den Typ T, T = byte, short, char, und der andere ist ein konstanter Ausdruck vom Typ int, so ist der Typ des ternären Operator T.

3. Andernfalls wird für numerische Typerweiterung für die beiden Ausdrücke vorgenommen und der Typ des ternären Operators ist diese Erweiterung.

Das zitierte Puzzle 8 zeigt dies sehr plastisch.

5.6.7 instanceof

Der *instanceof*-Operator ermöglicht es, zur Laufzeit den Typ eines Objekts zu überprüfen. In folgendem Beispiel wird in Zeile 10 ein Objekt der Klasse *Number* mit 0 initialisiert und in Zeile 20 eines mit $0L$.

```
10   Number ni = 0;
20   Number nl = 0L;
30   boolean bi = n instanceof Integer;
40   boolean bl = n instanceof Long;
```

Welchen Typ hat nun n? Zum einen natürlich den Typ *Number*. Aber 0 ist eine ganze Zahl, hat den Typ *int*. Auf die primitiven Typen kann aber nicht mit dem *instanceof*-Operator überprüft werden. Aber auf die zugehörige Wrapper-Klasse, hier also *Integer* kann überprüft werden. Das liefert hier *true*. Das ist kein Widerspruch, wie ein Blick in die API-Dokumentation zeigt. Die Klasse *Number* hat u. a. die Unterklassen *Integer* und *Long*. Dementsprechend wird in Zeile 20 ein *long* zugewiesen und das Objekt *nl* wird als vom Typ *Long* erkannt.

Etwas präziser formuliert liefert der *instanceof*-Operator für ein Objekt, dass *null* ist, immer *false*. Andernfalls überprüft, er, ob das Objekt vom Typ (Klasse oder Interface) ist, der auf der rechten Seite angegeben wird und liefert entsprechend *true* oder *false*. Ist der linke Operand keine Unterklasse des rechten Operanden, so gibt es einen Compiler-Fehler der Art „Incompatible conditional operand types“.

Für die weiteren Feinheiten des *instanceof*-Operators verweise ich auf Abschn. 19.3.

5.6.8 Cast

Der *Cast*-Operator ermöglicht unter geeigneten Bedingungen die Umwandlung des Typs eines Objekts in einen anderen. Einzelheiten zu den Java-Mechanismen hierfür finden sie in Abschn. 5.14 und im Zusammenhang mit auto(un)boxing.

Zwei Beispiele für die einfache Syntax: Java hat eine Methode, um Zufallszahlen zu erzeugen *Math.random()* liefert eine Zufallszahl aus dem Intervall $[0.0, 1.0)$. Wollen Sie einen Würfel simulieren, so werden Sie eine Klasse *Wuerfel* schreiben.

```
10 public class Wuerfel {
20    public int werfen(){
30        return (int)(Math.random()*6) + 1 ;
40    }
50 }
```

Besser ist aber folgende Version:

```
public class Wuerfel {
    public int werfen(){
        return new Random().nextInt(6) + 1 ;
    }
}
```

Da ein Würfel ganze Augenzahlen hat, müssen Sie in Zeile 30 mit 6 multiplizieren. Damit haben Sie eine Dezimalzahl aus dem Intervall $[0.0, 6)$. Durch einen Cast auf *int* erhalten Sie eine ganze Zahl zwischen 0 und 5 einschließlich. Sie addieren nun 1 darauf und können nun diese ganze Zahl zurückgeben. Die Methode *nextInt* nimmt Ihnen den Cast ab.

Bemerkung 5.6.5 (Bessere Zufallszahlen)
Bessere Zufallszahlen als *Random()*, d. h. solche die näher an zufällige Werte herankommen, liefert *SecureRandom* aus dem Paket *java.security*. ◄

Ein anderes Beispiel kennen Sie bereits aus dem Zusammenspiel von *compareTo* und *equals*. Die Methode *equals(Object obj)* erwartet ein Objekt vom Typ *Object*. Die Methode *compareTo(Clazz element)* erwartet ein Objekt vom Typ *Clazz*. Wenn wir innerhalb von *equals* nun *compareTo* aufrufen wollen, brauchen wir ein Objekt vom Typ *Clazz*. Da hilft nur ein Cast. Der geht aber nur gut, wenn das Objekt zur Laufzeit tatsächlich den Typ *Clazz* hat. Genau das stellen wir durch die vorhergende Überprüfung mit dem *instanceof*-Operator sicher.

5.6.9 .- Operator (Attribut- und Methoden-Zugriff)

Dieses Zeichen habe ich bereits mehrfach verwendet. Es zählt zwar nach [GJS+14] nicht zu den Operatoren, sondern zu den Separatoren, wird aber in manchen Kontexten als Operator angesehen. Er wird wie folgt verwendet:

- Haben Sie ein Objekt *obj*, so können Sie mit dem Punktoperator auf die aus dem Kontext sichtbaren Elemente der Klasse zugreifen. Hat die Klasse des Objekts eine Methode *doIt()* und ein öffentliches Attribut *data*, so können Sie schreiben:

  ```
  ... obj.doIt();
  ... obj.data ...;
  ```

- Hat eine Klassen *Fu* ein Klassenattribut *setting* und eine Klassenmethode *create()*, so können Sie mit dem Punktoperator über die Klasse auf diese Elemente zugreifen:

  ```
  ...  Fu.setting ...;
  ...  Fu.create();
  ```

- Befindet sich die Klasse *Fu* in einem Paket *fibel*, so können Sie in einem anderen Paket auf der gleichen Hierarchie-Ebene mit dem Punktoperator über das Paket auf die Klasse zugreifen. Sie können also deklarieren:

  ```
  fibel.Fu myFu;
  ```

  Einfacher ist es natürlich, das Paket *fibel* zu importieren und einfach *Fu* zu verwenden.

5.6.10 Vorrang von Operatoren

Durch systematisches Setzen con Klammern („(,)“ können Sie klar zum Ausdruck bringen, in welcher Reihenfolge Operatoren in einer Kombination ausgeführt werden sollen. Es gibt aber auch ein Reihen- und Rangfolge bei der ungeklammerten Kombination von Operatoren. Hier einige der wichtigsten Regeln (siehe [GJS+14], Abschn. 15.7).

1. Operanden von Operatoren werden von links nach rechts ausgewertet.
2. Für alle Operatoren außer &&, ||, and ? : gilt: Alle Operanden werden vollständig ausgewertet bevor der Operator angewandt wird.
3. Den geringsten Vorrang hat der Lambda-Operator, gefolgt von den Zuweisungsoperatoren.
4. & hat Vorrang vor |
5. ...

5.7 Initialisierung

Bei primitiven Typen und bei Objekten der Klasse String brauchen Sie einfach nur eine Variable zu deklarieren und ihr einen Wert bzw. bei Strings ein String Literal zuzuweisen.

Bei Referenztypen müssen Sie den *new-Operator* verwenden: Sie deklarieren eine Variable und initialisieren sie sobald wie möglich mit einem Objekt. Dies kann ein existierendes sein oder eines, dass Sie mittels *new* erzeugen. Nach dem dem Schlüsselwort *new* folgt ein Konstruktor mit den entsprechenden Parametern.

Attribute werden automatisch initialisiert:

Typ	Wert
numerisch	0
boolean	false
char	␣
Referenz	null

Die Verwendung von *new* kann in einer Fabrikmethode „versteckt“ sein. Ein gutes Beispiel sind die Datumsklassen in Java. So hat z. B. *LocalDate* keinen öffentlichen Konstruktor, sondern nur (statische) Fabrikmethoden. Für weitere Einzelheiten der Initialisierung verweise ich auf Kap. 9.

5.8 Methodenaufruf

Was passiert beim Aufruf einer Methode in Java? Ganz einfach ist es, wenn die Methode keine Parameter hat. Dann wird der Code der Methode ausgeführt und liefert das spezifizierte Ergebnis. Das heißt:

- Der Code der Methode wird ausgeführt.
- Etwaige Änderungen an dem Objekt der Methode werden vorgenommen.
- Falls der Rücktyp nicht *void* ist, wird der Rückgabewert zurückgeliefert.
- Geht etwas schief, so wird der Fehler in irgendeiner Form behandelt, siehe Kap. 12.

Dies gilt soweit auch, wenn die Methode Parameter hat. Hier müssen Sie aber zwischen der Übergabe primitiver Typen und der von Referenztypen unterscheiden.

Bei der Übergabe von primitiven Typen (genauer: Werten von primitivem Typ), wird an die Methode eine Kopie des Parameterwerts übergeben. Diese Kopie kann in der Methode beliebig verwendet und natürlich auch verändert werden. Es hat aber keinerlei Auswirkungen auf den ursprünglichen Wert, da nur auf einer Kopie gearbeitet wird.

Bei der Übergabe von Referenztypen (genauer: Objekten von Referenztypen) ist es zunächst ganz genauso: Es wird eine Kopie der *Referenz* übergeben. Sie werden diese *Referenz* aber in der Methode kaum verändern wollen. Tun Sie das nämlich, haben Sie keinen Zugriff mehr auf die Originalkopie. Die JVM unterbindet auch zuverlässig, dass Sie die Referenz einfach auf eine andere Adresse in der JVM „umbiegen“. So etwas irrtümlich zu tun ist eine mögliche Fehlerursache in Sprachen wie C++.

Die Kopie der *Referenz* gibt Ihnen Zugriff auf die öffentlichen Elemente des Objekt, also die entsprechenden Attribute und Methoden. Mittels der Kopie der Referenz und dem Zugriff auf die Attribute bzw. Methoden können Sie also das Objekt verändern! Diese Änderungen werden dann auch „nach außen“ sichtbar und wirksam.

Einfache Beispiele finden Sie in *basic.CallByValueAndByReference*

Eine Methode kann eine feste Anzahl von Parametern haben. Java kann aber auch mit Methoden umgehen, die eine variable Anzahl von Parametern haben. Nach einer beliebigen Anzahl von Parametern kann ein weiterer Typ angegeben werden, von dem eine beliebige Anzahl von Parametern folgen kann. Das sieht dann etwa so aus:

```
public static void main(String args...){
  for(String s:args){
     //Werte s aus.
  }
}
```

Sie sehen auf diese Weise auch gleich eine andere Möglichkeit eine main-Methode zu deklarieren, nämlich nicht mit einem Array von Strings, sondern mit einer variablen Anzahl von Strings. De facto ist dies aber nichts Anderes. Die Deklaration mit einem String-Array *String [ ] args* ist die Übliche und vermeidet die unnötige Konvertierung der Parameter in ein Array.

5.9 Rekursion

Viele Probleme können nach dem Prinzip „Teile und Herrsche“, lateinisch „divide et impera“ gelöst werden. Haben Sie ein Problem, dass Sie nicht sofort lösen können, so zerlegen Sie es in kleinere Probleme, bis das Problem so klein ist, dass Sie es lösen können. Anschließend setzen Sie aus den Teillösungen die Gesamtlösung zusammen. Das ist nicht immer ein effizienter Weg, funktioniert aber in vielen Fällen zumindest prinzipiell.

Ein oft verwendetes Beispiel sind die Fibonacci-Zahlen, die wie folgt definiert werden können:

$$\begin{aligned} f_0 &= 1 \\ f_1 &= 1 \\ f_n &= f_{n-1} + f_{n-2} \; \forall \; n \geq 2 \end{aligned}$$

Für $n \geq 2$ können wir f_n nicht unmittelbar angeben. Aber die obige Definition lässt sich ganz einfach in Java umsetzen:

```
public static int fibRec(int n) {
   return (n==0)?1:((n==1)?1:(fibRec(n-1)+ fibRec(n-2)));
}
```

Hier wird die Methode *fibRec* innerhalb ihres Rumpfes aufgerufen, hier sogar zweimal: Einmal für $n-1$ und einmal für $n-2$. Beachten Sie bitte, dass der Fall $n < 0$ nicht abgefangen wird. Wenn eine Methode sich selbst aufruft, spricht man von Rekursion. Beachten Sie bitte außerdem, dass diese Berechnung der Fibonacci-Zahlen auf diese Weise sehr ineffizient ist.

Wichtig ist bei einer Rekursion, dass Sie erkennen, wenn Sie nicht mehr weiter absteigen können, Das geschieht hier durch die Vergleich von n mit 0 bzw. 1. Dann kennen wir ja das Ergebnis unmittelbar aus obiger Definition.

Ein weiteres Beispiel liefert die Berechnung des größten gemeinsamen Teilers zweier ganzer Zahlen, hier für *long*s.

```
public static long ggtRekursiv(long i, long j) {
   return j==0 ? i:ggtRekursiv(j,i%j);
}
```

Beide Operationen finden Sie in *numbers.IntegerFunctions*, die zugehörigen Testfälle in *IntegerFunctionsTest*.

5.10 Arrays

Arrays sind geordnete Collections von Objekten eines Typs. Ein erstes Beispiel kam schon mehrfach in Beispielen vor, ohne näher erläutert zu werden. Eine *main*-Methode hat ein Array von Strings als Parameter:

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

Bemerkung 5.10.1 (main: Alternative Parameterliste)
Mit den heutigen Java-Möglichkeiten kann man *main* auch so schreiben (s. o.):

```
public void main(String... args){...}
```

Die Array-Schreibweise ist aber die übliche und spart die Umwandlung der Parameter in ein Array zur Laufzeit. ◄

Ein Array wird also mit einem Typ (primitiv oder Referenz-) deklariert. Dann folgt ein Paar eckiger Klammern [ ] und dann der Name. Ein Array kann mittels *new* initialisiert werden, wie in

```
Counter [] counterArray = new Counter[10];
```

oder durch eine explizite Initialisierung

```
Counter [] counterArray = {new Counter(),...,new Counter()};
```

Im ersten Fall sind für einen Referenztyp alle Array Einträge *null*. Für einen primitiven Typ haben sie den entsprechenden default-Wert, z. B. 0 für *int*.

Diese Länge ist über das Attribut *length* abrufbar. Der Index läuft von 0 bis $length - 1$.

Strings und Arrays hängen eng zusammen: Haben Sie einen String s, so sind die folgenden beiden Statements logisch äquivalent.

```
s = "abc";
s = new String({'a', 'b', 'c'});
```

Umgekehrt hat die Klasse String eine Methode, um aus einem String ein Array von *char* zu machen:

```
char[]  c  = s.toCharArray()
```

Die Array-Länge ist nach Initialisierung konstant. Wollen Sie ein Array vergrößern, so verwenden Sie die *System.arraycopy*:

```
public static void arraycopy(Object src, int srcPos,
                             Object dest, int destPos, int length)
```

Wollen Sie die Länge z. B. verdoppeln, so geht das so:

```
CounterV01 [] counterArray = new CounterV01[10];
...
CounterV01 [] counterArray2 = new CounterV01[counterArray.length<<1];
System.arraycopy(counterArray,0,counterArray2,0,counterArray.length);
```

Nach [GJS+14], Abschn. 10.7 hat ein Array vom Typ T genau die öffentlichen Elemente, wie die folgende Klasse:

```
class A<T> implements Cloneable, java.io.Serializable {
   public final int length = X ;
   public T[] clone() {
      try {
         return (T[])super.clone(); // unchecked warning
      } catch (CloneNotSupportedException e) {
        throw new InternalError(e.getMessage());
      }
   }
}
```

5.11 Ausgabe

Ich stelle zunächst einfache Ausgaben auf die Konsole vor. Die Basis hierfür ist in der Klasse *java.lang.System* implementiert. Diese Klasse hat drei Klassenattribute vom Typ *Stream* (siehe Kap. 14, in dem die Streams systematisch behandelt werden). *err*, *out* und *in*. Der *Stream in* ist genauer ein *InputStream*, die anderen beiden sind *PrintStream*s.

Da dies Klassenattribute sind, brauchen Sie kein Objekt, um auf sie zuzugreifen, sondern können einfach *System.out* schreiben. Damit haben wir dann wieder über den Punkt-Operator Zugriff auf die Methoden von *PrintStream*. Hier interessieren uns zunächst die Folgenden:

print(T t) Druckt ein Objekt des Typs T, T = boolean, char, char [ ], double, float, int, long, Object, String. Es kann jeweils nur ein Objekt übergeben werden, Zeilenumbrüche müssen mit separaten Statements gemacht werden oder als zusätzliche Zeichen mit angegeben werden.

println(), println(T t) Parameter sind wie bei *print*, aber hier können die Parameter auch ganz fehlen. Diese Methode macht am Ende einer Zeile einen Zeilenumbruch.

printf(String format, Object... args) Formatierte Ausgabe mit den aktuellen Voreinstellungen.

printf(Locale l, String format, Object... args) Formatierte Ausgabe mit den länderspezifischen Voreinstellungen (locale).

printf ist übrigens nichts anderes als *format*.

Die *print* und die *println* Methode sind ganz einfach, ich verweise auf die Java Dokumentation. Genauer gehe ich nun auf die *printf* Methode ein

Die Klasse *Locale* repräsentiert eine geographische, politische oder kulturelle Region. Mit ihrer Hilfe können Unterschiede zwischen Regionen einfach berücksichtigt werden. Einige häufig benötigte Beispiele hierfür sind:

- Dezimalpunkt und Tausenderkomma (angelsächsische Schreibweise) oder Dezimalkomma und Tausenderpunkt (kontinentaleuropäische Schreibweise).
- Uhrzeiten
- Datumsangaben

Ein Objekt der Klasse *Locale* wird so erzeugt:

```
import java.util.Locale;
...
Locale german = new Locale("de");
Locale germany = new Locale("de","DE");
```

Da die Klasse *Locale* einige Konstanten definiert, kann man für einige Sprachen und Länder dies auch sprechender formulieren und vorhandene *Locale*s verwenden:

```
import java.util.Locale;
...
Locale german = Locale.GERMAN;
Locale germany = Locale.GERMANY;
```

Hierbei handelt es sich um Klassenattribute.

Die Darstellung folgt der Java API-Dokumentation ganz nah. Im Wesentlichen habe ich aus dem Englischen übersetzt.

Als Nächstes kommt der Format-String. Dieser hat die folgende Form:

```
%[argument_index$][flags][width][.precision]conversion
```

width gibt einfach an, wie breit die Ausgabe mindestens sein soll. Sind mehrere Variablen auszugeben, so wird unterstellt, dass sich der erste Format-String auf die erste Variable, der zweite auf die zweite usw. bezieht. Ist dies nicht der Fall, so verwendet man den argument_index.

				Flag		
Flag	General	Character	Integral	Floating	Date/	Beschreibung
Flag				Point	Time	Beschreibung
'-'	y	y[a]	y[b]	y	y	linksbündig
'#'	y	-	y	y	-	Format abhängig von conversion
'+'	-	-	y[d]	y	-	immer mit Vorzeichen
' '	-	-	y[d]	y	-	Führender Blank bei positiven Werten
'0'	-	-	y	y	-	Führende Nullen
','	-	-	y[c]	y[e]	-	Locale-spezifische Trennzeichen
'('	-	-	y[d]	y[e]	-	Negative Zahlen in Klammern

[a] Nur für conversions „o“, „x“, „X“
[b] In Abhängigkeit von der Definition von *Formattable*.
[c] Nur für conversion „d“.
[d] Für 'd', 'o', 'x', und 'X' bei BigInteger oder 'd' bei byte, Byte, short, Short, int und Integer, long, und Long.
[e] Nur für 'e', 'E', 'f', 'g', and 'G'.

Die *precision* hängt von der *conversion* ab. Bei Fließkommazahlen gibt dieser Parameter die Anzahl der Nachkommastellen an.

Das erste Zeichen der *conversion* gibt den Typ an. In den meisten Fällen ist das ganz einfach. Dieses Zeichen charakterisiert den Typ, boolean, char, ganzzahlig, Fließkomma. Die folgende Tabelle zeigt die jeweilige Wirkung:

Erstes Zeichen der *conversion*

Conversion	Kategorie	Beschreibung
'b', 'B'	general	Ist das Argument *null*, so ist das Ergebnis *false*. Ist das Argument *boolean* oder *Boolean*, so ist das Ergebnis der Wert von *String.valueOf()*. In allen anderen Fällen ist das Ergebnis *true*.
'h', 'H'	general	Ist das Argument *null*, so ist das Ergebnis *null*. Andernfalls ist das Ergebnis das von *Integer.toHexString(arg.hashCode())*
's', 'S'	general	Ist das Argument *null*, so ist das Ergebnis *null*. Implementiert arg *Formattable*, so wird *arg.formatTo* aufgerufen. Andernfalls ist das Ergebnis das von *arg.toString()*.
'c', 'C'	character	Das Ergebnis ist ein Unicode Zeichen
'd'	integral	Das Ergebnis wird als ganze Zahl im Dezimalsystem formatiert.
'o'	integral	Das Ergebnis wird als ganze Zahl im Oktalsystem formatiert.
'x', 'X'	integral	Das Ergebnis wird als ganze Zahl im Hexadezimalsystem formatiert.
'e', 'E'	floating point	Das Ergebnis wird als Dezimalzahl in wissenschaftlicher Notation formatiert.
'f'	floating point	Das Ergebnis wird als Dezimalzahl formatiert.
'g', 'G'	floating point	Das Ergebnis wird als Dezimalzahl formatiert. Je nach Genauigkeit und dem Wert nach Rundung erfolgt die in Wissenschaftlicher Notation.
'a', 'A'	floating point	Das Ergebnis wird als Hexadezimalzahl mit Signifikant und Exponent formatiert.
't', 'T'	date/time	Präfix für Datum und Uhrzeit conversions. Siehe Date/Time Conversions.
'%'	percent	Das Ergebnis ist ein Literal `'%' ('\u0025')`
'n'	line separator	Das Ergebnis ist der plattformspezifische Zeilentrennstrich

Einige *conversions* gibt es in Groß- und Kleinschreibung. Großschreibung bewirkt dabei eine Kapitalisierung der Ausgabe entsprechend der jeweiligen Locale.

5.12 Datum und Uhrzeit

Ich werde in diesem Abschnitt den Umgang mit Datum und Uhrzeit beschreiben, aber nur sehr knapp:

- Sie brauchen dies häufig.
- Einige Grundprinzipien lassen sich hier gut erläutern.

Die wichtigsten Klassen zum Umgang mit Datum und Uhrzeit finden Sie im Paket *java.time*, etwa *LocalDateTime*. Diese verwenden den ISO-Kalender aus ISO-8601. Weitere Kalender finden Sie im Paket *java.time.chrono*.

Die Klassen im Paket *java.time* und seinen Unterpaketen haben einige Eigenschaften gemeinsam, die ihre Verwendung einfach machen:

1. Haben Sie ein Objekt einer dieser Klassen, so kann es nicht mehr verändert werden; es kann sozusagen „nichts mehr schief gehen“.
2. Es gibt einfache Methoden mit weitgehend selbsterklärenden Namen, hier eine Auswahl:
   - now(): Liefert das aktuelle Datum bzw. die aktuelle Uhrzeit.

- of(...) liefert auf einfache Weise ein gewünschtes Datum oder eine Uhrzeit
- parse
- format
- ...

Brauchen Sie nur ein Datum bzw. eine Uhrzeit, so verwenden Sie *LocalDate* bzw. *LocalTime*. Brauchen Sie beides, so nehmen Sie entsprechend *LocalDateTime*.

Beispiel 5.12.1 (Datum und Uhrzeit)

1. Ohne weitere Maßnahmen bekommen Sie aktuelles Datum und aktuelle Uhrzeit in dieser Form:

```
System.out.println(LocalDateTime.now());
System.out.println(LocalDate.now());
System.out.println(LocalTime.now());
```

   Die Ausgabe ist in diesem Fall etwas in diesem Format:

```
2014-07-24T09:59:55.789
```

2. Wollen Sie ein bestimmtes Datum haben, geht das z. B. so:

```
System.out.println(LocalDate.of(2014,07,24));
System.out.println(LocalTime.of(11,1,01));
System.out.println(LocalDateTime.of(2014,07,24,11,1,01));
```

   Die Ausgabe hat in diesem Fall etwa dieses Format:

```
2014-07-24
11:01:01
```

3. Wollen Sie das nun nicht im amerikanischen Datums- oder Uhrzeitformat haben, so können Sie eine Lösungsstrategie anwenden, die fast immer hilft: Um mit einem Objekt, hier einem Datum (Klasse *LocalDate* etc.) etwas zu machen brauchen Sie ein Objekt, in diesem Fall eines, das die gewünschte Formatierung vornimmt. Mit dieser Lösungsstrategie kommen Sie hier zügig zum Ziel: Die Klassen *LocalDate* etc. haben alle eine Methode *format*, die einen *DataTimeFormatter* als Parameter erwartet. Für Objekte dieser Klasse gibt es praktische *Fabrikmethoden*, z. B. *ofPattern*. Die genauen Möglichkeiten der Formatierung entprechend geeigneter Pattern finden Sie in der API-Dokumentation. Das Ergebnis ist ein *String*, den Sie bei Verwendung der Methode *printf* einfach geeignet ausgeben können. Hier drei Beispiele:

```
System.out.printf("%s\n",LocalDate.of(2014, 07,24).
           format(DateTimeFormatter.ofPattern("dd.MM.YYYY")));
System.out.printf("%s\n",LocalDate.parse("24.12.2014",
                        DateTimeFormatter.ofPattern("dd.MM.yyyy")).
           format(DateTimeFormatter.ofPattern("dd.MM.YYYY")));
System.out.printf("%s\n",LocalDateTime.now().
           format(DateTimeFormatter.ofPattern("dd.MM.YYYY  HH:mm ")));
```

   Das Ergebnis ist hier:

```
24.07.2014
24.12.2014
24.07.2014  17:05
```

Gleichzeitig sehen Sie, wie Sie mittels eines entsprechenden Formatters auch das Eingabeformat eines Datums in der Methode *parse* angepasst bzw. korrekt verarbeitet werden kann. Sie müssen aber genau darauf achten, was Sie tun: So steht hier „m“ für Minute und „M“ für Monat. Den Code finden Sie im Paket *dateandtime* in der Klasse *ClockExample01*

◀

In der API-Dokumentation finden Sie eine vollständige Liste der verfügbaren Formattierungsmöglichkeiten für Datum und Uhrzeit.

Weiteres findet Sie in Kap. 21 in der Beschreibung von Lokalisierung aka I18N (Internationalization hat 20 Buchstaben, also 18 zwischen I und n).

Eiine ausführliche Beschreibung des Date Time APIs finden Sie in Kap. 11.

5.13 Deprecated

Java ist eine Sprache, die sich sehr schnell verbreitet hat. Dabei konnte es nicht ausbleiben, dass Elemente entstanden, die sich nicht auf Dauer bewährten. Da diese aber bereits in vielen Klassen von Programmiern verwendet wurden, konnten sie nicht einfach aus der Sprache wieder entfernt werden. Derartige Elemente werden als *deprecated* gekennzeichnet.

> **dep-re-cate** vt **-cated-cating** [L *deprecatus*, pp. of *deprecari* to avert by prayer], fr. *de* + *precari*...(1628) **1 a** *archaic* : to pray against (as an evil) **a** : to seek to avert **2** to express dissapproval of **3 a** : PLAY DOWN : make little of **3 b** BELITTLE, DISPARAGE

[Mis98]

Sie sollten Elemente die mit *deprecated* gekennzeichnet sind *nicht* mehr verwenden. Wenn Sie sie in der Vergangenheit verwendet haben, so empfehle ich, dass Sie deren Eliminierung planen. Eine immer verwendbare Strategie kenne ich nicht. Je nach Situation haben sich aber folgende Maßnahmen bewährt:

- Bei kleinen Systemen können Sie solche Änderungen an jeder Klasse vornehmen, die geändert wird. Aktuelle JUnit-Testfälle sind hierbei sehr hilfreich.
- Bei größeren Teilsytemen können Sie versuchen, solche Änderungen für den nächsten Entwicklungszyklus eines Teilsystems einzuplanen.

Diese Eigenschaft kann durch den Javadoc tag *@deprecated* (siehe Kap. 13) oder die entsprechende *Annotation @Deprecated*(siehe Kap. 20) zum Ausdruck gebracht werden. Viele Beispiele finden Sie in der Klasse *Date* aus *java.util*, die Teil des ursprünglichen Date Time APIs in Java war: Von den sechs Konstruktoren sind vier deprecated, von den ca. 30 Methoden sind 19 deprecated.

5.14 Typumwandlungen

Java ist eine typisierte Sprache. Jedes Element erhält bereits bei seiner Deklaration einen Typ. So wird gewährleistet, dass Sie nicht Äpfel mit Birnen vergleichen. Sie können nicht einfach ein Objekt eines Typs einer Variablen eines anderen Typs zuweisen. Das geht nur dort, wo die beiden Typen dies zulassen. Dies geht innerhalb einer Vererbungshierarchie. Außerdem nimmt Java einige Konvertierungen vor, sozusagen als Service für den Entwickler.

Nun gibt es aber Situationen, in denen man einen einmal deklarierten Typ auch als einen anderen verwenden will oder muss. In einigen Situationen ist das ganz natürlich:

- Ein Objekt ist Element irgend einer Klasse. Dann kann man dieses Objekt überall verwenden, wo ein Objekt erwartet wird, denn *Object* ist Oberklasse jeder Java Klasse. Ebenso wenn ein Objekt vom Typ eines Interfaces ist.
- Wenn die Klasse in einer Vererbungshierarchie steht, hat also eine direkte Oberklasse hat. Dann kann man das Objekt auch einer Variablen vom Typ einer der Oberklassen zuweisen. Das geht, denn eine Vererbungshierarchie ist eine *is-a*-Hierarchie.
- Sie werden in manchen Situationen flexibel in der Arbeit mit primitiven Typen und ihren Wrapper-Klassen sein. Dies erledigt Java mittels autoboxing bzw. autounboxing.
- Erweiterungen bei numerischen Typen: $byte \rightarrow short \rightarrow int \rightarrow long \rightarrow float \rightarrow double$.

In allen anderen Situationen müssen Sie explizit etwas tun. Dazu gibt es sog. *Casts*. Das Beispiel *Casting* finden Sie im Projekt Programmierfehler im Paket *basic*:

```
13 public static void main(String[] args) {
14   Parent parentMale = new Parent();
15   Parent parentFemale = new Parent();
16   Daughter daughter = new Daughter();
17   Son son = new Son();
18   parentFemale = daughter;
19   parentFemale = son;
20   daughter = (Daughter) parentFemale;
21   son = (Son) parentMale;
     ...
26 }
```

Führen Sie dies einfach so aus, so gibt es zunächst einen Fehler in Zeile 20. Sie können mittels auskommentieren einzelner Zeilen genauer beobachten, was funktioniert und was nicht.

Deshalb noch einmal der Hinweis: Bevor Sie einen Cast verwenden, überprüfen Sie Ihren Entwurf!

5.15 Historische Anmerkungen

Seit dem Jahr 1996 erscheinen in regelmäßigen Abständen Neuerungen der Programmiersprache Java. Alle Versionen bis einschließlich Java 1.6 sind von Oracle als EOL (Oracle, 2015) markiert worden und der Support für diese Produkte wurde, bzw. wird noch in diesem Jahr eingestellt. Tabelle 1 zeigt einen Überblick der bislang veröffentlichten Java-Versionen und deren Einführungsjahr. Im Folgenden stellt der Autor einen kurzen (nicht vollständigen) Überblick über die Funktionalitäten der Java Bibliotheken dar:

Version	Veröffentlicht
JDK 1.0	1996
JDK 1.1	1997
JDK 1.2	1998
J2SE 1.3	2000
J2SE 1.4	2002
Java 5	2004
Java 6	2006
Java 7	2011
Java 8	2014
Java 9	2017
Java 10	2018
Java 11	2018 (geplant)

Version 1 In der Version 1 von 1996 wurden Funktionalitäten, wie elementare Klassen aus den Bibliotheken java.lang, java.io, java.util, java.net, java.awt und java.applet eingeführt. Diese beinhalten grundlegende, für die Datenverarbeitung notwendige Klassen. (vgl. wikimediafoundation, 2015)

Version 1.1 Änderungen der Version 1.1. betrafen im maßgeblichen Sinn eine Erweiterung um innere Klassen, die von nun an genutzt werden konnten. Außerdem wurden die bereits erwähnten Bibliotheken um weitere Funktionalitäten, wie z.B. Internationalisierung, Beans, das seither bekannte Dateiformat JAR und weiteren Neuerungen erweitert.

Version 1.2 Maßgebliche Neuerung der auch als Java 2 bekannten Erweiterung des Sprachpakets war die Einführung eines Just-In-Time-Compilers. Dieser bewirkt die Übersetzung von Teil-Programmen in Maschinencode zur Laufzeit. Eine Angleichung der Geschwindigkeit an bereits vorkompilierte Programme war allerdings nicht zu erzielen. (vgl. wikimediafoundation, 2015)

Version 1.3 Die Version 1.3, welche im Jahr 2000 erschienen ist beinhaltete eine Engine zur Hotspotoptimierung von Codefragmenten, die in häufiger Anzahl ausgeführt werden. Diese Bestandteile konnten von nun an zur Laufzeit in Maschinencode übersetzt werden, was ebenfalls eine Erweiterung der JIT-Kompilierung darstellte. (vgl. wikimediafoundation, 2015)

Version 1.4 Mit Version 1.4 wurden Assertions eingeführt. Diese konnten von nun an genutzt werden, um gewisse Fakten/Daten z.B. als Precondition vorzugeben (assert myStringVariable != null;). Im unzureichenden Fall wird eine entsprechende Exception geworfen. (vgl. wikimediafoundation, 2015)

Version 5 Java 5 beinhaltete einen Funktionsumfang, der bis heute wesentliche Charakteristika der Sprache wiederspiegelt. So wurden z.B. Generics oder auch das Autoboxing/Autounboxing dem Sprachpaket hinzugefügt. Aber auch statischer Klassenimport und z.B. eine einfachere Syntax für Collections wurde mit diesem Update eingeführt. (vgl. wikimediafoundation, 2015)

Version 6 Zu den in Java 6 vorgestellten wichtigen Neuerungen zählen zum großen Teil Performanz Anpassungen. Des Weiteren wurde ebenfalls eine Unterstützung von Skriptsprachen hinzugefügt. (vgl. Oracle, 2015)

Version 7 Mit der Version 7 wurde die Programmiersprache erneut um weitere Funktionalitäten erweitert. So z.B. das Interface Autoclosable, mit dessen Unterstützung Streaminhalte automatisch in einem try/catch-Block geschlossen werden können. Auch ist nun ein Switch mit Strings möglich und eine Unterstützung von dynamischen Programmiersprachen wurde ebenfalls integriert. Auch der in dieser Version eingeführte Java Quick Starter bewirkt beim Start von Java-Programmen Performanz Verbesserungen. (vgl. Oracle, 2015)

Version 8 Die im Frühjahr 2014 veröffentlichte Version Java 8 beinhaltet nun einige Features, die bereits für die Version 7 geplant waren, deren Entwicklung sich allerdings verzögerte und deshalb in die Version 8 integriert wurde. So beinhaltet die Version 8 neben der Einführung von Lambda-Ausdrücken und den neuen *Streams* eine Überarbeitung der Zeitbibliotheken, Closures und z.B. Verbesserungen der Garbage-Collection, sowie weiteren Features. (vgl. Oracle, 2015)

Java 9 2017, Einführung eines Modulsystems und Linkers jlink, interaktive JShell (ähnlich irb in Ruby), verschiedene Verbesserungen bei Interfaces, Streams uvam.

Java ist eine objektorientierte Programmmiersprache. Innerhalb von Methoden wird aber prozeduraler Code verwendet. Java unterstützt aber auch sogenannte strukturierte Programmierung. Dieses Paradigma fordert, dass in einem Programm nur die Konstrukte

- Sequenz: Eine Folge von Anweisungen, die alle ausgeführt werden,

- Verzweigung, also if-then-else bzw. switch,
- Schleife, also for, while, do-while

vorkommen dürfen. Diese können selbstverständlich beliebig geschachtelt werden.

Die Möglichkeit auch *String*-Variablen in *switch*-Konstrukten zu verwenden kam mit Java 7 Mitte 2011.

Mit Java 8 wurde ein neuer Ansatz für den Umgang mit Datum und Uhrzeit eingeführt. Damit sind weitere Teile der Klassen *Date* und *Calendar* obsolet geworden.

Mit Java 8 kamen Elemente der funktionalen Programmierung hinzu. Hierzu mehr in späteren Kapiteln.

5.16 Aufgaben

1. Welche Begründung gibt die Java-Sprachreferenz für die reservierten Worte *const* und *goto*? Erklären Sie den mögliche Nutzen präzise.
2. In Beispiel 5.4.1 wird auch 'a' verwendet. Der Hashcode für 'a' ist 97. Wie kommt dieser Wert zustande?
3. Um zu prüfen, ob eine Zahl ungerade ist, wird in [BG05] wird der folgende Code präsentiert:

```
public static boolean isOdd(int i) {
    return i % 2 == 1;
}
```

   3.1. Funktioniert der Code oder nicht? Begründen Sie bitte Ihre Antwort!

   3.2. Schreiben Sie bitte eine bessere Version!

   Hinweis: [GJS+14], Abschn. 15.7.2.

4. Schreiben Sie bitte die folgende *while*-Schleife in eine äquivalente *do-while*-Schleife um!

```
public int do(int i, int j) {
   while (i != j) {
      if (i > j) {
         i -= j;
      } else {
         j -= i;
      }
   }
   return i;
}
```

5. Schreiben Sie bitte die folgende *do-while*-Schleife in eine äquivalente *while*-Schleife um!

```
public int static do(int i, int j){
   int result = 1;
   do{
      result*=i;
      j--;
   }while(j>0);
 return result;
}
```

6. Schreiben Sie bitte die folgenden *if*-Konstrukte in ein *switch*-Konstrukt um:

```
public static long fakultaet(long n){
   if(n<0){
      return 0;
   }
   else{
      if(n == 0){
         return 1;
      }else{
         return n*fakultaet(n-1);
      }
   }
}
```

7. Was geben die folgenden beiden Zeilen aus? [BG05]

```
System.out.print("H" + "a");
System.out.println('H' + 'a');
```

Also: Achtung liebe Ruby-ProgrammiererInnen!

Kapitel 6

Der Zähler (The Count)

6.1 Übersicht

Ich zeige in diesem Kapitel exemplarisch an einem ganz einfachen Beispiel verschiedene Möglichkeiten des Entwurfs und der Realisierung einer einfachen Anwendung. Die verschiedenen Entwürfe haben Vor- und Nachteile, einige sind akzeptabel, andere einfach nur schlecht und sollen zeigen, wie Sie es besser nicht machen sollten.

Aber sogar an diesem ganz einfachen Beispiel kann ich viele Möglichkeiten (nicht nur) der objektorientierten Programmierung in Java aufzeigen.

Dieses Kapitel ist nicht zur Lektüre in einem Stück am Anfang eines Programmierkurses gedacht. Es dient als *Steinbruch*, aus dem ich Beispiele entnehmen werde. Es wäre schön, wenn dies Kapitel am Ende eines Kurses mit Gewinn und zur Wiederholung gelesen werden könnte. Sie können dieses Kapitel aber auch als einen *Crash-Kurs in Java* lesen und daraus Übungsaufgaben gewinnen. Zur Zeit enthält dieses Kapitel auch eine Reihe von Absichtserklärungen, die noch nicht ausgeführt sind. Diese Teile können zum Üben der entsprechenden Inhalte verwendet werden.

6.2 Lernziele

- Elementare Begriffe der Objektorientierung verstehen.
- Grundlegende Symbole der UML lesen und anwenden können.
- Eine einfache Architektur (Client-Server) kennen.
- Das Observer-Pattern kennen und verstehen.
- Ein erstes Verständnis für Prinzipien der objektorientierten Programmierung gewinnen.

6.3 Einführungsbeispiel - Beschreibung

Es soll ein Zähler entwickelt werden, der bei einem definierten Ereignis einen Wert erhöht bzw. verringert. Bei Bedarf soll der Zähler auf einen Startwert zurückgesetzt werden können. Der jeweils aktuelle Wert soll angezeigt werden.

Man stelle sich hierzu eine Software-Version eines Zählers vor, wie er z. B. in Einkaufszentren zum Zählen von Besuchern verwendet wird. Es kommen für eine Software aber außer einer solchen Variante mit einer entsprechenden Benutzeroberfläche auch andere Schnittstellen in Frage. So könnte es sich um Daten handeln, die von einem Sensor stammen, einem Stromzähler etc. Ebenso könnte man solche Zähler verwenden, um generische Primärschlüssel für Tabellen in relationalen Datenbanken zu erzeugen.

Aus den zuletzt genannten Einsatzmöglichkeiten ergeben sich weitere Überlegungen:

1. Primärschlüssel müssen nicht ganzzahlig sein, man kann auch Werte aus Buchstaben oder Zahlen verwenden.

2. Auch völlig andere Basen von Zahldarstellungen können vorkommen, man denke etwa an Datum und Uhrzeit: Jahr ist ganzzahlig, Monat aus dem Bereich 1 bis 12, Tag aus dem Bereich 1 bis $28, 29, 30, 31$, je nach Monat und Jahr, Stunde aus dem Bereich 0 bis 24, Minute und Sekunde aus dem Bereich 0 bis 60.

3. In den Harry Potter Büchern von Joanne K. Rowlings hat eine Galeone 17 Sickel und ein Sickel 29 Knuts.

Die Entwicklung zeigt zunehmend komplexere Variationen dieses einfachen Beispiels auf. Die Varianten sind einfach mit Vnn durchnummeriert.

1. Ich beginne mit einer einfachen Klasse *CounterV00*. Diese repräsentiert einen Zähler mit eigener Methode *main* zum starten und testen.

2. Im nächsten Schritt entferne ich die *main*-Methode und schreibe mir eine Anwendungsklasse (*CounterV01App*) und eine Klasse zum starten der Anwendung (*CounterV01View*) Hinzu kommt noch ein Klasse *CounterV01Test* zum Testen der Methoden.

3. Im nächsten Schritt kann in der Klasse *CounterV02App* der Anfangswert für den Counter als Kommandozeilenparameter mitgegeben werden.

4. In der Klasse *CounterV03* gibt es keine Methode *main* mehr. Der Counter wird jetzt über eine Klasse *CounterV03start* oder die Klasse *CounterV03View* „betrieben“.

5. Mit *CounterV04* kommt ein einfacher Dialog für die Anzeige zum Einsatz: *CounterV04View*. Dies ist ein erstes Beispiel für GUI-Programmierung in Java.

6. In *CounterV05View* verliert nun auch die View-Klasse ihre *main*-Methode. Stattdessen gibt es nun eine CounterV05Application Klasse, die die Objekte erzeugt, konfiguriert und startet.

7. In CounterV06 wird die *toString*-Methode überschrieben.

8. In der Version V07 der drei Klassen zeige ich, wie das Bisherige auch ohne Vererbung gelöst werden kann.

   Von hier aus gäbe es nun weitere Möglichkeiten weiter voran zu gehen. Statt in verschiedene Äste zu verzweigen mache ich einfach linear weiter:

9. Ich starte aus einer Anwendung mehrere Views, die auf den gleichen Zähler zugreifen.

10. Um die Aktualisierung aller Anzeigen zu erreichen müssen die Klassen noch etwas erweitert werden. (Observer-Pattern)

    Beim parallelen Zugriff kann es zu Problemen kommen, deshalb der nächste Schritt.

11. Diese versehe ich im nächsten Schritt mit einem automatischen Verändern des Counters, ggf. auch zufällig.

12. Im letzten Schritt wird es Probleme geben, zu deren Lösung ich *synchronized* einführe.

13. Dies kann man noch etwas weiter konfigurierbar machen, etwa mit anderen Schrittweiten als ± 1. Das geht mit

14. und ohne Vererbung.

15. Oder man kann mit einer Angabe, um wieviel verändert werden soll arbeiten.

16. Der Zähler kann gespeichert werden(Interface *Serializeable*).

17. Der Stand des Zählers kann auch visualisiert werden. Ganz einfach mit einem Slider.
18. oder mit Balken
19. oder Linien.
20. Der Counter kann mit einer Einheit (Temperatureinheiten, Geldeinheiten in verschiedenen Währungen etc.) konfiguriert werden.
21. Das gibt Anlass ein sehr allgemein einsetzbares Schema zu erläutern (Conversion pattern).
22. Damit kann man dann auch leicht in verschiedene Einheiten umrechnen.
23. Der Counter und der View können in verschiedenen Partitionen laufen (RMI)
24. Der Counter und der View können mittels eines Protokolls wie http kommunizieren.
25. Ein View kann mit einem Counter einer noch unbekannten Klasse konfiguriert werden und sich die geeigneten Methoden zur Laufzeit suchen.

6.4 Analyse

Als Erstes analysiere ich die Aufgabenstellung etwas genauer. Da Programmiersprachen fast alle englische Worte verwenden, mache ich dies in diesem Beispiel ebenfalls. Aufgrund der in Abschn. 6.3 genannten Anforderungen ist die Schnittstelle der Klasse auf den ersten Blick klar: Die Abb. 6.1

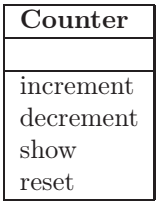


Abb. 6.1: Klasse Counter, Version 1

zeigt aber nur einen Teil dieser Schnittstelle und keine Interna, wie Attribute oder wichtige Details, wie die Rückgabetypen der Operationen. Hier sind die Anforderungen noch völlig unklar. In der einfachsten Version werde ich mich (natürlich nach Rücksprache mit dem Auftraggeber) auf ganze Zahlen beschränken. Wird ein Zähler neu angelegt, so steht er auf 0. Dies führt auf Abb. 6.2 in Abschn. 6.5.

Bemerkung 6.4.1 (Namenskonventionen)
Peter Coad vertritt in [CN93] die Ansicht, man solle bei Klassennamen aufpassen, die auf „-er“ enden.Dies ist bei *Counter* der Fall. Der Name und damit auch die suggerierte Verantwortung der Klasse könne falsch gewählt sein (*-er-er principle*). Für die Klasse *Counter* könnte dies heißen: Es handelt sich nicht nur um den Zähler, sondern auch um dessen Verwaltung. Er wählt dort deshalb den Namen *Count*. Für die deutsche Sprache vermag ich dieses Risiko zumindest bisher an dieser Stelle nicht zu erkennen. Für diesen speziellen Fall hat das deutsche Wort *Zähler* ja sowohl die Bedeutung eines Stromzählers o. ä., der auf Signale reagiert als auch die eines Zählers, der aktiv etwas zählt, wie etwa die Anzahl Passanten in einer Einkaufszone.

Das englische Wort count entspricht hier auch dem deutsche Wort Zahl. In der amerikanischen Sesam-Straße (Saseme Street) gibt es eine Figur Count of Count, der in der deutschen Fassung Graf Zahl heißt. Auch hieran hätte ich in der Namensgebung anknüpfen können. ◀

6.5 Ein erster Entwurf

Aus der Aufgabenstellung ist klar, dass die Klasse Counter (Zähler) in der einfachsten Form etwa so aussehen muss: Hier ist *Counter()* ein Konstruktor, der das Attribut *value* mit dem Wert 0

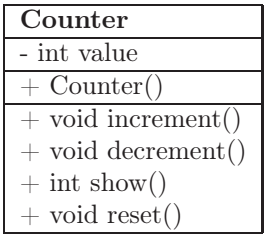


Abb. 6.2: Klasse Counter, Version 1.1

initialisiert. Die Zeichen „-,+“ vor den Elementen charakterisieren die Sichtbarkeit:

\+ public, öffentlich

– private, privat

\# protected, geschützt.

6.6 Eine erste Implementierung

Die Klasse aus Abb. 6.2 kann direkt in Java implementiert werden.

Wer mit *HelloWorld* Java gelernt hat wird hier ein *main*-Methode schreiben. Die entsprechende Variante zeigt die Version V00: *counter/CounterV01.java*. Diese finden Sie wie alle erwähnten Beispiele in meinem pub.

Besser ist es aber, die Funktionalität auf mehrere Klassen aufzuteilen: Das habe ich in *CounterV01* getan. Die *CounterV01*-Klasse hat nur die Counter-Methoden. Gestartet wird die Anwendung über die Klasse *CounterV01App*, die „Oberfläche“ implementiert die Klasse *CounterV01View*. Den Code aus der *main*-Methode aus *CounterV00* habe ich nun in die Methode *execute* von *CounterV01View* verlagert. Ansonsten hat sich nichts verändert. Aber nun kann ich die „Oberfläche“ ausgestalten und muss dazu die anderen Klassen nicht ändern.

6.7 Testen

Besser als eine *main*-Methode zu schreiben, ist es aber einige Testfälle zu schreiben. Das habe ich in *crash/CounterV02Test.java* getan. Die Tests mit *JUnit* während der Entwicklung testen nur ein gewisses Verständnis der Anforderungen. Dies kann in keinem Fall ein systematisches Testen ersetzen. Trotzdem sind (J)Unit-Tests sinnvoll!

Eine erfolgversprechende Idee, Fehler im Code zu finden, besteht darin, maximale oder minimale Werte zu verwenden (Äquivalenzklassen, Grenzwertuntersuchung).

Ich teste deshalb nun auch mit minimalen und maximalen Werten für den Wert, mit dem das Attribut *resetValue* der Klasse *Counter0* initialisiert wird. Dazu dupliziere ich den *JUnitTest*-Fall *Counter0Test* zu *Counter0MinMaxTest*. Dort rufe ich den Konstruktor mit Parameter auf und setzte den *resetValue* auf MAXINT.

Dabei verfolge ich ganz einfache Strategien. Die Implementierung verwendet allerdings fortgeschrittene Java-Techniken. Testfälle:

1. Nach Erzeugen eines *Counter0*-Objekts muss die *show()* Operation den Wert 0 liefern.

2. Wenn hinaufgezählt wird, muss der Wert um 1 erhöht werden. Ich teste nach einem Aufruf von *increment* auf 1, nach 100 weiteren auf 101.

3. Ganz analog teste ich 1 decrement, 100 *decrement*s.

4. *reset* teste ich, indem ich nach 100 Aufrufen der Methode *increments* die Methode *reset* und anschließend die Methode *show* aufrufe.

5. *show* teste ich nicht separat.

6. Nun könnte man natürlich auch noch prüfen, ob nach 1.000.000 weiteren 1.000.101 erreicht ist. Dies ist allerdings nicht sinnvoll. Testfälle sind erfahrungsgemäß besonders wirkungsvoll, wenn sie Grenzbereiche betreffen. Es wäre also nützlich, zu testen, wie sich der Counter verhält, wenn es Maximal- bzw. Minimalwerte erreicht. Dazu müssen Sie den Counter aber gar nicht so weit hochlaufen lassen. Mittels *Reflection* (siehe Kap. 19) können Sie das private Attribut einfach setzen. Ich teste dies in der folgenden Methode:

```
@Test
public void testMinimax() {
   try {
       Field f = this.counter.getClass().getDeclaredField("count");
       f.setAccessible(true);
       f.setInt(counter, Integer.MAX_VALUE);
       } catch (Exception e) {
                e.printStackTrace();
       }
       assertEquals(Integer.MAX_VALUE, counter.show());
       this.counter.decrement();
       assertEquals(Integer.MAX_VALUE-1, counter.show());
       this.counter.increment();
       this.counter.increment();
       assertEquals(Integer.MIN_VALUE, counter.show());
}
```

Ausführlich wird dieses Thema in Kap. 19 behandelt. Hier nur eine knappe Erläuterung:

6.1. *Integer.MAX_ VALUE* und *Integer.MIN_ VALUE* sind Klassenattribute der Wrapper-Klasse *Integer* mit den Werten $2^{31} - 1$ bzw. -2^{31}.

6.2. Die Methode *getClass()* der Klasse *Object* liefert die Klasse eines Objekts.

6.3. Die Methode *getDeclaredField* liefert ein Objekt der Klasse *Field* aus dem Paket *java.lang.reflect*.

6.4. Die Klasse *Field* erbt von der Klasse *AccessibleObject* aus dem gleichen Paket die Methode *setAccessible*, mittels der ich mir Zugriff auf das Attribut verschaffen kann, obwohl es privat ist.

6.5. Mittels der Methode *setInt* kann ich nun den Wert setzen.

6.6. Ich teste dann zunächst, ob der Counter auch in diesem Grenzbereich noch funktioniert und überschreite dann den Grenzbereich. Ich erwarte eine „Wrap-around“, also dass *Integer.MAX_ VALUE* durch Addition von 1 den Wert *Integer.MIN_ VALUE* liefert (siehe dazu Abschn. 7.4.1).

6.7. Die verwendeten Methoden aus dem Paket *java.lang.reflect* werden im Fehlerfall eine *Exception*. Die Klasse könnte nicht vorhanden sein, ein Security-Manager könnte die Verwendung von *setAccessible* unterbinden usw. Deshalb ist ihre Verwendung in einen *try*-Block gekapselt. Im zugehörigen *catch*-Block gebe ich nur die Fehlerursache mittels *printStackTrace()* aus.

Alle Test laufen nun erfolgreich durch.

Bemerkung 6.7.1 (Praktisches Vorgehen)
Ich gehe meistens so vor, dass ich die Methoden zunächst als „Dummies“ schreibe, also ohne den Methodenrumpf auszuimplementieren und ggf. noch einen Konstruktor. Dann kann ich mir die Testmethoden von JUnit generieren lassen. Anschließend schreibe ich dort meinen Testcode. Dann schlagen natürlich noch alle Tests fehl. Anschließend implementiere ich die Klasse aus. So finden Sie sie im angegebenen Verzeichnis. Habe ich alles richtig gemacht, laufen dann alle Tests erfolgreich durch. Eine Garantie für Korrektheit ist das aber selbstverständlich nicht. ◄

6.8 Zwei-Schichten-Modell

Bereits in *CounterV01* habe ich ein Zwei-Schichten-Modell eingeführt. Dieses Modell werde ich auch in der Folge verwenden.

- Eine Anwendungsschicht, die hier nur den Counter (jetzt *CounterV03*) enthält.
- Eine Ausgabe- und später auch Eingabeschicht, die die Methoden von Counter aufruft und Ergebnisse ggf. auf der Konsole ausgibt: *CounterV03View* und *CounterV03App*.

An der *main*-Methode von *CounterV03App* zeige ich hier auch, wie man einem Java-Programm Kommandozeilenparameter mitgibt und diese verarbeitet.

Die *main*-Methode hat die Form:

1. Array von Strings als Parameter

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

2. Eine variable Anzahl von Strings kann auch verwendet werden:

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

Beide Formen sind äquivalent. Die übliche Form ist die erste, d. h. die mit Array. Das Schlüsselwort *static* bedeutet, dass es sich um eine Klassenmethode handelt. Eine *main*-Methode wird aber nicht mit dem Klassennamen aufgerufen. Die Java Virtual Machine (JVM) ruft diese Methode auf, wenn auf der Konsole etwas eingegeben wird wie:

```
java CounterV03App
```

Wollen Sie Parameter mitgeben, so geben Sie diese einfach nach dem Programmnamen an, also z. B.

```
java CounterV03App 42
```

Sie können dies direkt in Eclipse machen. Dazu rufen Sie Run As→ Run Configurations auf. In dem Tab (x)=Arguments können Sie dann die Parameter angeben.

Auf die Parameter können Sie einfach zugreifen. Hier die wesentlichen Zeilen der *main*-Methode aus *CounterV02App*:

```
public static void main(String[] args) {
    new CounterV03View(new CounterV03(
                       args.length==1?Integer.parseInt(args[0]):0))
                       .execute();
}
```

Um den übergebenen Parameter an den Konstruktor zu übergeben, muss ich aus einem String eine ganze Zahl machen. Dazu gibt es eine Klassenmethode *parseInt* der Klasse *Integer*. Ich verwende hier den ternären Operator: Wird genau ein Parameter übergeben, so lese ich diese wie oben beschrieben. Wird kein Parameter übergeben, so wird der Wert für den Default-Konstruktor verwendet, d. h. 0. Diese einfache Implementierung hat mindestens zwei Schwächen:

1. Es wird nicht abgeprüft, ob der übergebene String tatsächlich eine ganze Zahl ist. Gebe ich etwa 42.0 ein, so gibt es eine *NumberFormatException*. Diese stammt aus der Methode *Integer.parseInt*. Diese müsste man eigentlich abfangen und Nutzern ein Chance zur Korrektur geben.

2. Es wird nur geprüft ob genau ein Parameter übergeben wird. Das ist ebenfalls verbesserungswürdig. So könnte bei mehr als einem Parameter ein Hinweis ausgegeben werden.

Einige dieser Schwächen lassen sich leicht beheben: Um mit der Umgebung (Nutzern und Nutzerinnen) zu kommunizieren verwende ich hier zunächst die Konsole. Schreiben und lesen geht in Java über Streams. Um eine Ausgabe auf die Konsole zu machen, brauche ich also ein Objekt einer geeigneten Streamklasse. Dieses finde ich als Klassenattribut der Klasse *System*: *out* ist ein Objekt der Klasse *PrintStream*. Diese Klasse hat viele Methoden. Ich verwende hier eine einfache:

```
System.out.println("Bitte geben Sie den Startwert für den Zähler ein!");
```

Diese gibt den String auf den aktuellen *PrintStream* aus und macht einen Zeilenumbruch. Der Default-PrintStream ist die Konsole.

Wie kommen Sie aber an die Benutzereingabe heran? Das geht wie an einer Supermarktkasse: Sie brauchen einen Scanner. Eine Klasse *Scanner* finden Sie im Paket *java.util*. Einer der Konstruktoren der Klasse Scanner erhält einen *InputStream* als Parameter. Die Konsole wird per Default durch *System.in* repräsentiert.

Ein *Scanner* hat eine Methode *hasNext()*, die *true* liefert, wenn es (noch) etwas zu lesen gibt. Für das Lesen vieler Typen gibt es Operationen, z. B. *nextInt()* etc. Strings werden oft gelesen, deshalb wohl heißt die Methode hierfür *next()*. Um daraus eine ganze Zahl zu bekommen, verwenden wir wie oben *parseInt*.

Der Rest des Codes enthält keine neuen Konstrukte. Was aber sehr wohl noch fehlt, ist eine Fehlerbehandlung. So wird nicht geprüft, ob es sich tatsächlich um eine ganze Zahl handelt, ob es nur ein Parameter ist usw.

6.9 Oberflächen

Mit dem, was hier bisher vorgestellt wurde, können Sie eine ganz einfache Konsoloberfläche für den Counter schreiben. Hier der wesentliche Teil aus *CounterV03View*:

```
while (sc.hasNext()) {
   switch (sc.next().charAt(0)) {
      case '+':
         this.counter.increment();
         break;
      case '-':
         this.counter.decrement();
         break;
      case 'e':
         System.exit(0);
         break;
      default:
         break;
   }
```

```
   System.out.println("Der Zählerstand ist gerade: " + this.counter.show());
   System.out.println("Geben Sie bitte '+' für erhöhen oder " +
                                        "'-' für verringern "+
                                        "'e' für Ende ein!");
}
```

Hier kommen zwei neue Konstrukte vor: eine *while*-Schleife und ein *switch*-Befehl.

Die *while*-Schleife wird durchlaufen, wenn die in Klammern dem *while* folgende Bedingung wahr ist. Andernfalls wird die Ausführung nach dem Block fortgesetzt, der dem *while* folgt. Dies wird hier allerdings nie eintreten. Wie wir gleich sehen werden, wird die Anwendung durch eine entsprechende Eingabe des Benutzers beendet.

Der *switch*-Befehl ist die Java-Variante einer Mehrwegentscheidung: Es wird eine Variable vom numerischem Typ, *enum*(siehe Abschn. 18.14) oder String nach dem Schlüsselwort *switch* angegeben. Hinter jedem der folgenden *case*-Ausdrücke steht ein Zeichen. Die Befehle nach dem ersten *case*-Ausdruck, bei dem der Wert der Variablen nach dem *switch* gleich dem Zeichen ist werden ausgeführt. Mit dem *break*-Befehl wird das *switch*-Konstrukt verlassen. Gibt es kein *break*, so werden auch alle folgenden *case*-Zweige durchlaufen. Der *default*-Zweig wird durchlaufen, wenn keiner der anderen *case*-Zweige durchlaufen wurde.

In diesem Beispiel wird aufgrund der Eingabe im Fall „+“ erhöht, im Fall „-“ verringert und im Fall „e“ das Programm verlassen.

Damit haben wir jetzt die Klassenstruktur in Abb. 6.3 für diese Minianwendung.

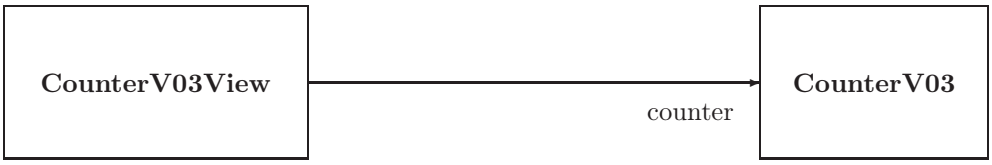


Abb. 6.3: Counter und View-Klasse

Als nächstes zeige, ich wie man mit Java — genauer mit *Swing* — eine einfache graphische Oberfläche „bauen“ kann. Dies geschieht in *CounterV04View*. Da es sich hier um eine ganz einfache Interaktion mit dem Anwender handelt, genügt auch eine einfache Klasse. Ich spezialisiere hier die Klasse *JDialog*. Sie finden Sie wie viele Swing-Klassen im Paket javax.swing. Ich verwende in diesem Beispiel Elemente aus drei Bereichen, die ich hier knapp erläutere.

1. *CounterV04View* ist eine Unterklasse von *JDialog*:

   ```
   public class CounterV04View extends JDialog{ ...}
   ```

   Sie hat zwei Konstruktoren: Einen, der den Counter mit dem Default-Startwert 0 initialisiert und einen, der einen Startwert für den Counter als Parameter erhält. Der Code befindet sich fast ausschließlich im zuletzt genannten Konstruktor.

   Ich rufe mittels *super* zunächst den Konstruktor der Oberklasse *Dialog* aus dem Paket *java.awt* auf. So wird ein zunächst unsichtbarer Dialog ohne Titel erzeugt. Als Erstes passe ich das Aussehen an das jeweilige Betriebssystem an:

   ```
   try {
      UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
   } catch (Exception e) {
         e.printStackTrace();
   }
   ```

So sieht der Dialog unter Windows wie ein typischer Windows, unter Motif wie ein typischer Motif Dialog aus usw. Tue ich dies nicht, so hat er das typische Java Metal-Look-and-Feel.

Zum Schluss spezifiere ich das Verhalten des Fensters beim Schließen, lasse alle Bestandteile (s.u.) anordnen und mache es sichtbar:

```
this.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
this.pack();
this.setVisible(true);
```

2. In dem Dialog brauche ich drei Elemente: Eine Anzeige des Zählerstands und je ein Element zum erhöhen bzw. verringern. Die kann man mit graphischen Editoren machen. Ich erläutere hier ein direktes Vorgehen ohne weitere Hilfsmittel. Für die Anzeige verwende ich ein Label. Ein solches Element kann Text anzeigen, der Text kann aber nicht editiert werden. Das passt hier. Zum erhöhen und verringern des Zählers entscheide ich mich für Buttons. Die beiden JButtons nenne ich *increment* bzw. *decrement*. Um Elemente in einem Fenster und anderen Komponenten zu positionieren bietet Java verschiedene Layouts. Ich entscheide mich für ein *Gridlayout* mit einer Zeile und drei Spalten:

   ```
   this.setLayout(new GridLayout(1,3));
   ```

   Die Elemente werden initialisiert (alle sind Attribute der Klasse *CounterV04View*):

   ```
   this.counterValue = new JLabel(Integer.toString(this.counter.show()),
                                  SwingConstants.RIGHT);
   this.increment = new JButton("+");
   this.decrement = new JButton("-");
   ```

   und dem Dialog hinzugefügt und der Titel gesetzt:

   ```
   this.add(this.counterValue);
   this.add(this.increment);
   this.add(this.decrement);
   this.setTitle("Ein einfacher Zähler");
   ```

3. Mit dem bisherigen sieht man den Anfangsstand des Zählers, aber das drücken der Buttons bewirkt noch nichts. Nur das Schließen des Dialogs funktioniert. Damit ein Button reagieren kann, wenn er gedrückt wird, braucht er einen Listener, in diesem Fall ein Objekt des Typs der durch das Interface *ActionListener* spezifiziert wird. Für diese einfache Anwendung verwende ich hier eine anonyme Klasse, hier am Beispiel *increment*

   ```
   this.increment.addActionListener(new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent e) {
         counter.increment();
         counterValue.setText(Integer.toString(counter.show()));
         repaint();
      }
   });
   ```

   Hierbei passiert Folgendes: Für den JButton *increment* wird die Methode *addActionLister* aufgerufen. Diese erhält ein Objekt des Typs *ActionListener*. Ich kann aber nicht einfach schreiben „*new ActionListener()*“: *ActionListener* ist ein Interface. Deshalb enthält der folgende Block den Code einer Klasse, die die einzige Methode dieses Interfaces implementiert. Der counter wird erhöht, das Label mit dem Zählerstand aktualisiert und die Anzeige mittels Aufruf von *repaint()* aktualisiert. Da die Klasse keinen Namen hat, nennt man sie *anonyme Klasse*. Die zugehörigen .class-Dateien heißen hier CounterV04View$1.class und CounterV04View$2.class.

6.10 Counter mit Klassenattribut

Als Beispiel für ein Klassenattribut, im Java-Jargon *static* genannt, zähle ich jetzt, wieviele Counter-Objekte es gibt. Den Code finden Sie in *CounterV05*. Der Code ändert sich zunächst nur an zwei Stellen: Die erste Nichtkommentar-Zeile im Klassenrumpf lautet nun:

```
private static int counterNumber=0;
```

Beim ersten Laden der Klasse wird das Klassenattribut *counterNumber* mit 0 initialisiert. Im Konstruktor wird dieses Attribut immer um 1 hochgezählt:

```
public CounterV05(int start){
   counterNumber++;
   this.count = start;
}
```

Um das Funktionieren des Konzepts zu prüfen, schreibe ich mir wieder Testfälle, *CounterV05Test*. Vor jedem Testfall lege ich mir ein neues Array von Zählern an:

```
@Before
public void setUp() throws Exception {
   this.counters = new CounterV05[maxCounters];
}
```

Vor dem Ausführen eines Testfalls wird also stets ein Array von *null*-Zählern angelegt. Hier wird also nicht etwa der Default-Konstruktor aufgerufen, sondern die Array-Elemente werden mit *null* initialisiert. Der erste Testfall

```
@Test
public void testCounterV05() {
   assertEquals(0, CounterV05.getCounterNumber());
   this.counters[0] = new CounterV05();
   assertEquals(1, CounterV05.getCounterNumber());
   this.counters[1] = new CounterV05();
   assertEquals(2, CounterV05.getCounterNumber());
}
```

läuft auch erfolgreich durch. Die anderen beiden aber nicht:

```
@Test
public void testCounterV05Int() {
   assertEquals(0, CounterV05.getCounterNumber());
   for(int i=0;i<this.counters.length;i++){
       this.counters[i] = new CounterV05(i);
   }
   assertEquals(10, CounterV05.getCounterNumber());
}
```

läuft aber gleich beim ersten *assert* auf einen Fehler: Erwartet: 0, war aber : 2. Lässt man dieses *assert* weg, so liefert das zweite: Erwartet: 10, war aber : 12. Ganz analoge Ergebnisse liefert der zweite Testfall:

```
@Test
public void testGetCounterNumber() {
   assertEquals(0, CounterV05.getCounterNumber());
   this.counters[0] = new CounterV05();
   this.counters[1] = new CounterV05(42);
   assertEquals(2, CounterV05.getCounterNumber());
}
```

Woran liegt das? Um diese Frage zu beantworten muss ich erklären, wie in Java Klassen geladen und Objekte erzeugt werden. Die Klasse *CountV05* taucht bei der Ausführung von *CountV05Test* zuerst bei der Deklaration des Attributs *counters* auf:

```
CounterV05 [] counters;
```

An diesem Punkt lädt die JVM die zugehörige Klassendatei. Dabei werden die Klassenattribute initialisiert, hier also *CounterV05.counterNumber*. Dieses Klassenatrtibut wird nun bei jedem Anlegen eines Zählerobjekts mittels *new* um 1 hochgezählt. Bis zum Ende des ersten Testfalls funktioniert das auch alles.

Nun kommen wir zum zweiten Testfall. Zwar wird vorher in der Methode *setUp()* ein neues Array von *CounterV05* angelegt. Aber an dem Wert des Klassenattributs *CounterV05.counterNumber* ändert sich nichts, es steht also weiterhin auf dem Wert 2.

Daran wird sich auch nichts ändern, so lange ich nicht etwas dafür tue, dass der Wert auch wieder verringert wird. In C++ gibt es Destruktoren, die zum Zerstören eines Objekts aufgerufen werden können. In so einem Destruktor könnte ich dann das Klassenattribut wieder um 1 reduzieren. In Java gibt es keine Destruktoren. Objekte, die nicht mehr benötigt werden, zerstört der Garbage Collector. Dies geschieht irgendwann, wenn das Objekt nicht mehr referenziert wird. Wann genau wird innerhalb der JVM entschieden. Wird das Objekt vom Garbage Collector „entsorgt“, wird dessen *finalize*-Methode aufgerufen. Diese Methode ist als *protected* bereits in der Klasse Object definiert. Diese kann ich also überschreiben und *counterNumber* wieder reduzieren:

```
@Override
protected void finalize(){
   counterNumber--;
}
```

Erneutes Ausführen der Testfälle liefert aber das gleiche Ergebnis wie zuvor. Die einzige Erklärung, die wir hierfür bisher haben ist diese: Der Garbage Collector hat noch nicht „zugeschlagen“. Es gibt aber die Möglichkeit, diesem Prozess etwas „auf die Sprünge zu helfen“. So kann ich in eine *tearDown* Methode nach jedem Testfall zunächst das Attribut *counters* von *CounterV05Test* auf *null* setzen:

```
@After
public void tearDown() throws Exception {
   this.counters = null;
}
```

Das bringt uns aber zunächst nicht weiter. Also muss ich nach weiteren Möglichkeiten suchen, den Garbage Collector zum arbeiten zu motivieren. Ich werde in der Klasse System fündig: Dort gibt es eine Methode *System.gc* mit folgender Beschreibung:

> Runs the garbage collector.
>
> Calling the gc method suggests that the Java Virtual Machine expend effort toward recycling unused objects in order to make the memory they currently occupy available for quick reuse. When control returns from the method call, the Java Virtual Machine has made a best effort to reclaim space from all discarded objects.

(http:/download.oracle.com/javase/6/docs/api/index.html?java/lang/Object.html. Also probiere ich dies als nächstes aus:

```
@After
public void tearDown() throws Exception {
   this.counters = null;
   System.gc();
}
```

Nun laufen alle Testfälle erfolgreich durch — zumindest in meinen ersten drei Versuchen. Lasse ich allerdings die Zeile mit *this.counters = null;* weg, so schlägt der zweite Testfall fehl.

Die Verwendung von *finalize* ist riskant und kostenträchtig. Ersteres zeigte schon die Konsturktion der Testfälle. [Blo08] berichtet in Item 7 von einem um den Faktor 430 erhöhten Aufwand für Erezugung und Zerstörung eines Objekts, nur durch die Existenz einer *finalize*-Methode.

Eine solche ist aber auch gar nicht notwendig: Schreiben Sie ein Methode *terminate* in der Counter-Klasse. Diese rufen Sie z. B. aus einem *CounterView*-Objekt auf. Ein Gerüst finden Sie in den Klassen *Counter*WithoutFinalize*.

6.11 Mehrere Sichten

Als nächstes baue ich die Klasse zum Starten der Anwendung so aus, dass sie mehrere Dialog-Fenster aufmachen kann und dort auch die Anzahl der Zähler anzeigt. Die will ich so konfigurieren, dass wahlweise ein Zähler von mehreren Dialogen „bearbeitet“ wird oder auch für jeden Zähler ein Dialog verantwortlich ist.

Im ersten Schritt gibt es *CounterV05*, *CounterV05View* Paare. Als Erstes ergänze ich die View-Klasse um ein Feld zur Anzeige der Anzahl Zähler. Dazu ergänze ich zwei weitere JLabel. Eines mit Text und eines mit der Anzahl Zählerobjekte. Das GridLayout wird jetzt mit 2 Zeilen und 3 Spalten definiert.

Aus *CounterV05Test* kennen Sie bereits die Möglichkeit, Objekte in einem Array zu verwalten. Um in dem jetzigen Beispiel flexibel zu sein, nutze ich es gleich um zwei weitere wichtige Dinge einzuführen. Um Objekte eines bestimmten Typs zu verwalten gibt es in Java Container-Klassen. Sie finden viele in den Paketen *java.util* und *java.util.concurrent*. Oft werden Sie das Interface *List* verwenden. Dies ist ein generisches Interface, d. h. es wird mit einem Typparameter verwendet, etwa so:

```
List<CounterV05> counterList = new LinkedList<CounterV05>();
List<CounterV05View> counterViewList=new LinkedList<CounterV05View>();
```

Der jeweilige Typ wird in spitzen Klammern angegeben, wie bei <CounterV05>. Ein so spezifizierter Typ heißt parametrisierter Typ. So deklarierte Listen müssen natürlich mit „richtigen“ Objekten initialisiert werden. Ich habe mich hier für eine verkettete Liste, genauer *LinkedList* aus dem Paket *java.util*. entschieden. Von beiden Typen erstelle ich jeweils eine leere Liste. Nun erzeuge ich mir einige Counter und Views in der *main* Methode von *CounterV05Application*:

```
for(int i=0;i<10;i++){
   counterList.add(new CounterV05(i));
}
counterViewList.add(new CounterV05View(counterList.get(0)));
for(int i=1;i<=3;i++){
   counterViewList.add(new CounterV05View(counterList.get(1)));
}
```

Dabei stelle ich sofort ein Schwäche fest: Es ist nicht zu erkennen, welcher Dialog für welchen Zähler verantwortlich ist. Eine schnelle Lösung besteht darin, dies im Titel des Views mit anzugeben. Dazu muss ich nicht viel tun: JDialog hat ein Methode *setTitle*, mit der ich den Titel festlegen kann. Die *main*-Methode lautet nun:

```
for(int i=0;i<10;i++){
   counterList.add(new CounterV05(i));
   counterViewList.get(0).setTitle("Ansicht von Zähler" + 0);
}
counterViewList.add(new CounterV05View(counterList.get(0)));
for(int i=1;i<=3;i++){
   counterViewList.add(new CounterV05View(counterList.get(1)));
```

```
        counterViewList.get(i).setTitle("Ansicht von Zähler" + 1);
}
```

Nun kann ich „sehen“, dass der erste Dialog zum Zähler 0 gehört und die anderen drei zum Zähler 1. Völlig unbefriedigend ist aber das Verhalten, in dem Fall, dass es mehrere Dialoge gibt, die alle auf einen Zähler zugreifen. Dort scheint der Zähler zu springen, wenn inkrementiert oder dekrementiert wird. Ändert ein Dialog den Zählerstand, so bekommen die anderen dies noch nicht mit. Die Anzeige wird erst aktualisiert, wenn in dem jeweiligen Dialog der „+“ oder „-“ betätigt wird.

Eine Lösung für dieses „Problem“ liefert das sogenannte *Beobachter-Muster*. Es ist auch unter dem Namen *Model-View-Controller*, kurz *MVC* bekannt.

Das Grundprinzip erläutert Abb. 6.4. In Java geht man etwas anders vor, als es dieses Schema

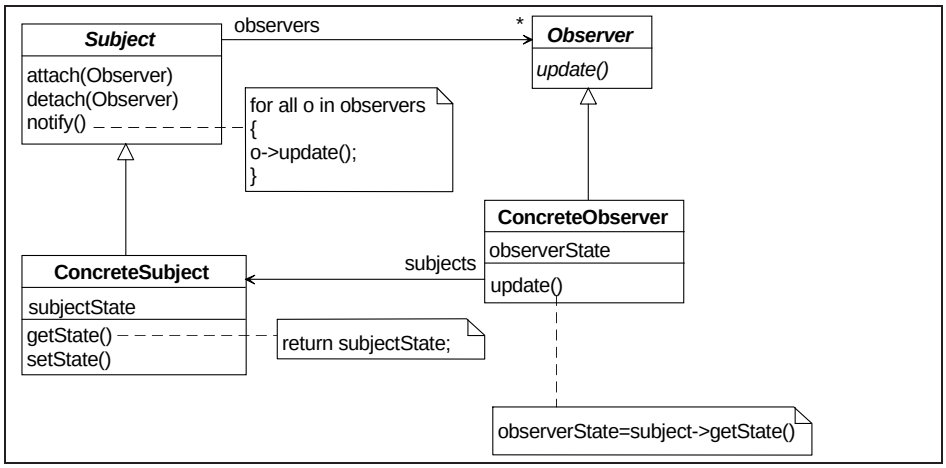


Abb. 6.4: Beobachter-Muster

zeigt: Statt der abstrakten Klassen *Subject* werden und *Observer* werden Interfaces verwendet oder es werden Methoden aus *Object* genutzt. *Object* implementiert sowohl *Subject* als auch *Observer*. Um das zu würdigen, muss ich aber noch etwas weiter ausholen, was ich gerne auf später (PR2) verschiebe.

Hier entscheide ich mich für eine ganz einfache Lösung: Die Klasse *CounterV05* erhält ein weiteres Attribut:

```
private List<CounterV05View> observers;
```

Diese wird im Konstruktor als leere Liste initialisiert:

```
public CounterV05(int start){
   counterNumber++;
   this.observers = new LinkedList<CounterV05View>();
...
```

Nun ergänze ich noch Methoden, mit denen Observer hinzugefügt (*addObserver*, entspricht *attach* in Abb. 6.4), benachrichtigt werden (*notifyObervers*, entspricht *notify* in Abb. 6.4)) und entfernt (removeObserver, entspricht *detach* in Abb. 6.4) werden können:

```
public void addObserver(CounterV05View view){
   this.observers.add(view);
}
private void notifyObservers(){
   for(CounterV05View view:this.observers){
```

```
         view.update();
      }
   }
   public void removeObserver(CounterV05View counterV05View) {
      this.observers.remove(counterV05View);
   }
```

Zum Hinzufügen und Entfernen der Observer (hier sind das Objekt der Klasse *CounterV05View*) verwende ich die entsprechenden Methoden der Liste oder genauer des Interfaces *List*. Die in *notifyObservers* verwendete Methode *update()* muss ich in *CounterV05View* noch ergänzen. Sie soll die Anzeige des Zählerstands aktualisieren:

```
   public void update(){
      this.counterValue.setText(Integer.toString(this.counter.show()));
      this.counterValue.repaint();
}
```

Hier wird also der anzuzeigende Wert mit dem aktuellen Wert des Counters aktualisiert und die Anzeige erneuert. Genaugenommen würde es genügen, nur dieses Label zu aktualisieren und so steht es hier auch im Code.

In der Methode *notifyObservers*l

```
   for(CounterV05View view:this.observers){
       view.update();
   }
```

sehen Sie aber auch ein weiteres Beispiel der *for each*-Schleife. In dieser Variante steht links vom Doppelpunkt ein „Laufvariable“ mit Typ und Name, hier *CounterV05View view*. Rechts vom Doppelpunkt steht ein Container mit Elementen des angegebenen Typs, hier *this.observers*. Der Block $\{\ldots\}$ wird dann für alle $view \in this.observers$ durchlaufen.

Damit die Methode *notifyObservers* etwas tun kann, muss sie natürlcih aufgerufen werden. Dass mache ich hier *increment*:

```
   public void increment() {
      this.count++;
      this.notifyObservers();
   }
```

und ganz analog in *decrement*.

Nun muss ich noch dafür sorgen, dass sich jeder View beim Zähler anmeldet. Dies geschieht in Konstruktor mit Parameter (*CounterV05View(CounterV05 view)*:

```
   public CounterV05View(final CounterV05 counter) {
      ...
      this.counter.addObserver(this);
      ...
   }
```

In der bis jetzt vorgestellten Form wird die Anzeige der Anzahl Zähler noch nicht aktualisiert. Dies finden Sie aber in dem Source-Code im pub..

6.12 Unterschiedliche Typen von Zählern

Eine Richtung in die dieses Beispiel fortgeführt werden kann betrifft andere Typen, wie bereits eingangs angesprochen.

- Erweiterung des Zählbereichs mit *BigInteger*.

- Sekunden, Minuten, Stunden, Tage ...
- Knuts, Sickel, Galeonen.
- Buchstaben a, b, ..., z, aa, ab, ..., mit anderen Worten: Basis 26.
- Ausgabe in anderen Basen.
- Fibonacci Zahlen.

6.13 Speichern

In vielen Systemen finden Sie eine Schichtenarchitektur, wie sie Abb. 1.4 zeigt. Die HIC besteht in diesem Beispiel nur aus der View-Klasse, die ich jetzt *CounterVxxView* nenne, die PDC aus aus der Counter-Klasse, nun *CounterVxx*. Die DMC ist noch nicht vorhanden. Das Counter-Objekt ist weg, wenn die Anwendung endet. Was ist aber, wenn mit dem Zähler-Objekt später weiter gearbeitet werden soll?

Dazu gibt es in Java einen ganz elementaren Mechanismus, der als *Serialisierung* bezeichnet wird. Dieser Mechanismus verwendet das Konzept des *Marker-Interfaces*. Eines dieser Interfaces ist *Serializable* aus dem Paket *java.io*

```
public interface Serializable {
}
```

Dieses Interface hat *keine* Operationen! Implementiert aber eine Klasse dieses Interface, so kann man ihre Objekte mittels eines *ObjectOutputStream*s wegschreiben, z. B. in eine Datei. Dies kann auf verschiedene Arten realisiert werden. Ich entscheide mich hier dafür, die Serialisierung in *CounterVxx* direkt zu realisieren.

Zunächst ergänze ich die Definition von *CounterVxx*:

```
public class CounterVxx implements Serializable{
```

Nun gibt es mehrere Möglichkeiten, wie für das Speichern des Counter-Objekts gesorgt werden kann:

1. Das Counterobjekt ist selbst für die Speicherung verantwortlich. Für diese Variante spricht, dass Nutzer dann nur eine entsprechende Operation, z. B. *save* aufrufen müssen. Dagegen spricht, dass auch Anwender, die diese Funktionalität nicht benötigen, sie „mitschleppen“ müssen.

2. Das View-Objekt könnte die Speicherung übernehmen. Dafür spricht, dass das Counter-Objekt davon unbelastet bleibt. Dagegen spricht, dass eine Schichtenarchitektur wie in Abb. 1.4 immer nur den Zugriff auf die nächst tiefere Ebene zulässt (sie ist geschlossen).

3. Das Counter-Objekt wird beim Beenden der Anwendung gespeichert. Dies kann von der Klasse *CounterVxxApplication* übernommen werden.

4. ...

Serializable hat gravierende Schwächen. Für Einzelheiten verweise ich an dieser Stelle auf [Blo08], Item 74, 75, 3 uvam.

Achtung: Dieser Teil schint redundant zu sein. Nun entwerfe ich eine „Oberfläche“ für den Counter. Ohne weitere Programmierkenntnisse kann ich eine Konsol-Oberfläche bauen, die etwas so aussehen könnte, wie in Abb. 6.5. Für diese Anzeige schreibe ich mir eine Klasse *CounterConsoleView*. Sie enthält eine Schleife mit einer (mittels *printf* formatierte) Ausgabe des Zählerstandes, die Menüausgabe und wartet dann auf eine Eingabe. Bevor ich dahin komme, muss ich aber sicher noch den *resetValue* abfragen und initialisieren. Dazu muss ich drei Dinge wissen:

```
Zählerstand:          1234


erhöhen:              i
verringern:           d
zurücksetzen:         r
beenden:              e


Bitte wählen Sie eine Aktion!
```

Abb. 6.5: Eine einfache Konsol-Oberfläche für den Counter

1. Wie gebe ich etwas auf die Konsole aus?
2. Wie formatiere ich eine Ausgabe?
3. Wie lese ich von der Konsole ein?

Die Antworten auf alle diese drei Fragen könnte heißen: Ich erzeuge mir jeweils ein Objekt einer geeigneten Klasse und rufe eine Operation auf, die das tut, was ich möchte. Hier nun die Antworten, die Ihnen gleich einen Einblick geben, wie Sie in Java arbeiten und wie Sie sich im System zurechfinden.

1. Die Konsole gehört zu Ihrem System. Dementsprechend finden Sie Operationen um damit umzugehen in der Java-Klasse *System* im Paket *java.lang*. *System* ist eine sogenannte *Utility-Klasse*. Solche Klassen sind dadurch gekennzeichnet, dass sie nur Klassenoperationen und Klassenattribute haben. Ein- und Ausgaben erfolgen in Java vielfach über Objekte, die von einer Unterklasse von *InputStream* oder *OutputStream* sind. Eines der drei Klassenattribute von *System* ist das *PrintStream*-Objekt *out*. Auf dieses Attribut können wir direkt zugreifen: mit *System.out* haben wir also ein Objekt der Klasse *PrintStream*. In diesder Klasse finden wir die Operationen *print*, *printf* und *println*. Der *Standard*-Printstream ist die Konsole. Alle *print*- und *println*-Operationen akzeptieren als Parameter einen String oder etwas, dass ich automatisch in einen String umwandeln lässt.

2. Formatiert möchte ich hier nur den Zählerstand ausgeben. Dies ist eine ganze Zahl und mit Tausenderpunkt ist so eine Zahl besser zu lesen. Nun ist es aber so, dass in Europa ein Tausenderpunkt und ein Dezimalkomma verwendet wird. In den USA wird aber ein Tausenderkomma und ein Dezimalpunkt verwendet. Für solche regionalen Unterschiede gibt es eine *Utility*-Klasse *Locale*. Der Einfachheit halber hat diese für oft benötigt Lokalisierungen vordefinierte konstante Klassenattribute, z. B. *GERMAN* für Deutsch. Dieses Objekt *Locale.GERMAN* der Klasse *Locale* ist der erste Parameter für die Operation *printf*. Der zweite Parameter ist ein *Format-String*. Ich brauche hier nur zwei Dinge: Für jeden der folgenden Parameter muss ich angeben, von welchem Typ er ist. Die geschieht z. B. mit %s für String oder %d für ganze Zahlen. Ein „,“ vor dem „d“ bewirkt die Ausgabe von Tausenderpunkt und Dezimalkomma in der jeweils regional üblichen Form. Damit habe ich fast mein ganzes *printf*:

```
System.out.printf(Locale.GERMAN,
                  "%s%,d\n",
                  "Zählerstand: ",this.counter.show());
```

Noch nicht erklärt habe ich das „\n“ am Ende des Format-Strings: Diese bewirkt einen Zeilenumbruch.

3. Wie liest man nun von der Konsole ein? Denken Sie einfach an einen Einkauf im Supermarkt oder an den nächsten Flug: Dort werden die Etiketten bzw. das Gepäck und Personen „gescanned“. Genauso geht das in Java: Sie brauchen ein Objekt der Klasse *Scanner* (aus dem Paket *java.util*). Wie Sie diese benutzen, zeigt der Beispielcode.

```
public class CounterConsoleView {
    private Counter counter;
    private Scanner sc;
    public CounterConsoleView() {
        this.sc = new Scanner(System.in);
        this.counter = new Counter();
    }
    public void start() {
        System.out.println("Starten eines Zählers");
        System.out.println("Geben Sie bitte den Startwert ein:");
            if (sc.hasNextInt()) {
                this.counter = new Counter(sc.nextInt());
            } else {
                this.counter = new Counter();
    }
            run();
        }
...
}
```

Nun brauche ich nur noch ein Klasse, die das Ganze startet, z. B.

```
public class CounterApplication {
public static void main(String[] args) {
CounterConsoleView counterView = new CounterConsoleView();
counterView.run();
}
}
```

Sie finden den vollständigen Code als Lösungsvorschlag im Paket *a01* für das Praktikum des Sommersemesters 2010.

6.14 Erstes Refactoring

In der Vorlesung am 16.03.2010 fiel mir auf, dass bei Schriftgrad 14*pt* die Methode *run* nicht mehr auf eine Bildschirmseite passte. Das habe ich zum Anlass genommen den Code mittels Refactoring zu verändern. Unter *Refactoring* versteht man eine Änderung des Codes, die seine innere Struktur verbessert, aber garantiert nichts an der Funktionalität ändert. Garantierten können Sie dies eigentlich nur, wenn dies automatisch geschieht. Eclipse unterstützt Sie dabei. Einmal gezeigt habe ich bereits das Ändern eines Namens (rename). In Eclipse geht das so: Sie markieren den Bezeichner oder setzen zumindest den Cursor an dessen stelle und betätigen dann die rechte Maustaste. Es öffnet sich dann ein Kontextmenü, aus dem Sie nun *refactor* und anschließend etwa *rename* auswählen.

Ich möchte nun Teile der Methode *run* in kleinere Methode extrahieren. Dazu markiere ich zunächst das gesamte *switch*-Konstrukt. Aschließend betätige ich die rechte Maustaste und wähle unter *refactor* nun die Auswahl *extract method*. Als Namen der Methode wähle ich nun *switchOnSelection*. Anschließend markiere ich noch die Zeilen, in denen die Ausgabe erfolgt und extrahiere

sie in eine Methode *produceOutput*. Die Zeile, in denen die Eingabe gelesen wird, extrahiere ich in eine Methode *getInput*. Im Ergebnis habe ich nun eine übersichtliche Methode *run*:

```
public void run() {
   char selection=' ';
   while (selection!='e') {
      produceOutput();
      selection = getInput();
      switchOnSelection(selection);
}
```

Den vollständigen Code finden Sie im Paket *a00* in der Klasse *CounterConsoleViewV2*.

6.15 Varianten

Man kann natürlich in vielen Systemen zählen.

1. Ganzahlig zu irgend einer Basis. Dazu gehört unser bisher implementierter Dezimalzähler. Genauso aber auch Binär-, Octal- oder Hexadezimalzähler.
2. Es gibt Zahlsysteme die nicht auf Stellen mit einer Basis beruhen:
   2.1. Uhrzeit und Datum
   2.2. Römische Zahlen
   2.3. Fibonacci-Zahlsystem

Den ersten Punkt kann ich leicht integrieren: Ich kann die Operation *show* zu überladen. Dies hat aber Konsequenzen: Ich kann nicht mehr *int* zurückliefern. Das ist aber insofern unkritisch, als die einzige Anwendung, die ich zur Zeit im Sinn habe eine Anzeige ist. Die Anzeige erfolgt als *String*, so lange ich mir dafür nicht eine weitere Klasse schaffe. Um mir Flexibilität zu erhalten, überlade ich deshalb nicht *show*, sondern schreibe eine neue Methode *showInBase*.

Da die Werte des Counters als *int* dargestellt werden, haben wir es mit maximal 32 oder 64 Bit zu tun. In einer Darstellung mit einer ganzzahligen, positiven Basis brauchen wir also maximal 64 Zeichen zur Darstellung. Mittels der in Kap. 7 beschriebenen Verfahrens kann ich also die Integer-Darstellung in eine andere Basis umrechnen (siehe *simple.CounterWithBase*).

6.16 Mehrbenutzerzähler

Bisher haben wir einen Zähler betrachtet, der von einem Programm, einer Person o. ä. verwendet wird. Nun betrachten wir einen Zähler bzw. ein Zählerobjekt, dass von mehreren Programmen angezeigt und inkrementiert werden kann. Ein naives Vorgehen hierzu wurde bereits in Abschn. 6.11 kennengelernt. Nun soll das etwas professionalisiert werden.

Zunächst überlege ich mir ein geeignetes Szenario, um ein *Counter*-Objekt mehrfach zu benutzen. Das geht durch mehrere Fenster oder durch mehrere Threads, die automatisch alle paar Millisekunden hochzählen bzw. inkrementieren. Ich will dabei zeigen, dass dies zu unerwünschten Effekten führen kann. Anschließend zeige ich dann, wie man diese verhindert. Im Minimum werden Threads benötigt, die inkrementieren (etwas jede Sekunde), einen der alle Sekunde dekrementiert und einen der alle 20 Sekunden zurücksetzt.

Beobachter und Threads einführen.

6.17 Weitere Übungsmöglichkeiten

Die rudimentären Klassen dieses Kapitels können zu Übungszwecken weiter ausgearbeitet werden. Hier einige Ideen:

1. Internationalisierung (Externalisierung von Strings).
2. Komfortablere Bedienung der Zähler: Benutzergesteuerte Erzeugung der Zähler und der Views.
3. Konfiguration der Threads mit Steuerung der Schnelligkeit, zwischenzeitlicher Änderung der Schnelligkeit etc.
4. Verteilung von Oberfläche und Zähler über RMI, so dass beide auch auf verschiedene Rechner verteilt werden könnten.
5. Kommunikation über Netzwerk.
6. Manipulatorthread, der unter Verwendung der Reflection Möglichkeiten die Fehlerständer verfälscht.
7. Installation eines Security Managers, um diese Manipulation zu verhindern.

6.18 Historische Anmerkungen

Das Beispiel eines Zählers zur Einführung in die objektorientierte Programmierung habe ich zu erst in dem dritten Band der „Trilogie“ über OO von Peter Coad gesehen [CN93]. Dort wird dieses Beispiel in Smalltalk und C++ diskutiert. Vieles davon ist noch aktuell, aber die verwendete Notation für Analyse- und Designmodelle ist heute nicht mehr üblich. So macht die Lektüre einige Mühe.

6.19 Aufgaben

1. Was passiert bei der Klasse *Counter* aus dem Paket *simple*, wenn beginnend mit *resetValue*$= 2^{31} - 2$ fünf Mal die Methode *increment* aufgerufen wird? Konkret: Welche Wert hat dann *value*? Was passiert, wenn anschließend sieben Mal *decrement* aufgerufen wird?
2. Welche Werte liefert *CounterWithBase* für die Basen $-2, -10, -16$ für $1, 2, 3, \ldots 17$?
3. Ergänzen Sie bitte die Klassen zu *CounterV05* so, dass auch die Anzeige der jeweils aktuellen Anzahl Zähler aktuell ist!

Kapitel 7

Numerische Datentypen und Zahlendarstellungen

7.1 Überblick

Java hat mehrere Datentypen für Zahlen. Diese werden hier mit vielen ihrer Eigenschaften dargestellt. Dazu gehört auch eine Darstellung der zugehörigen *Wrapperklasse*n (*Referenztypen*) und deren Zusammenspiel mit den elementaren oder primitiven Typen. Vor allem aber sollen die internen Zahlendarstellungen und ihre praktischen Konsequenzen vermittelt werden. Dieses Wissen werden Sie nicht immer und überall benötigen. Aber die Grenzen der Genauigkeit von Digitalrechnern müssen Ihnen im Bedarfsfall bewusst sein.

7.2 Lernziele

- Die Zahlentypen in Java kennen.
- Zahlendarstellungen im Rechner, insbesondere in Java kennen.
- Mit Zahlen in verschiedenen Systemen arbeiten können.
- Die numerischen Operationen kennen und anwenden können.
- Die Grenzen numerischer Operationen auf Rechnern kennen.
- Einige bitweise Operationen kennen und anwenden können.

7.3 Primitive Typen und Wrapper-Klassen

Java hat die im Folgenden beschriebenen numerischen primitiven Datentypen:

int Ganze Zahlen der Länge 32 Bit. Dies ist nach [GJS+14] auf allen Plattformen so. Genauer: *int* Werte sind Zahlen zwischen -2.147.483.648 und 2.147.483.647 im 2-er Komplement. Meine Erfahrung zeigt aber, dass es auf manchen 64-Bit Betriebssystemen Implementierungen gibt, bei denen *int* die Länge 64-Bit hat. Als Beispiel sei Windows Vista in der 64-Bit Version und die dafür verfügbare Eclipse-Version genannt.

long Ganze Zahlen der Länge 64 Bit, also an analog zu *int* von -2^{63} bis $2^{63} - 1$.

short Ganze Zahlen der Länge 16 Bit, also von -2^{15} bis $2^{15} - 1$.

byte Ganze Zahlen der Länge 8 Bit, also von -2^7 bis $2^7 - 1$.

char Unicode-Zeichen, die auch als ganze nichtnegative Zahlen der Länge 16 Bit interpretierbar sind und deshalb hier aufgeführt werden.

float Fließkommazahlen mit 32 Bit entsprechend IEEE 754.

double Fließkommazahlen mit 64 Bit entsprechend IEEE 754.

Für die Einzelheiten der internen Darstellung verweise ich auf die Abschnitte 7.4–7.5.

Zu jedem der primitiven numerischen Typen gibt es eine zugehörige sogenannte *Wrapperklasse*. Diese haben den Namen des primitiven Typs, aber nicht in lower case sondern in UpperCamelCase, d. h. der erste Buchstabe wird großgeschrieben und der Name wird ausgeschrieben. Hinzu kommen noch die Klassen *BigDecimal* und *BigInteger*. Allen diesen Klassen bzw. *Wrapperklassen* gemeinsam sind die folgenden Eigenschaften:

- Zu einem Wert eines primitiven Typs gibt es ein entsprechendes Objekt der zugehörigen *Wrapperklasse*.
- Die Operation *hashCode* einer *Wrapperklasse* eines ganzzahligen Typs liefert für das Objekt den Wert des zugehörigen primitiven Typs.
- Die Operation *equals* liefert für zwei Objekte genau dann *true*, wenn die zugehörigen Werte gleich sind.
- Sie haben eine Klassenmethode `int compare(Typ x, Typ y)`.
- Wird ein Wert eines primitiven Typs an einer Stelle verwendet, an der ein Objekt benötigt wird, so erfolgt eine automatische Kapselung in das entsprechende Objekt der zugehörigen *Wrapperklasse* in vielen Fällen automatisch (autoboxing). Wird ein Objekt einer *Wrapperklasse* einer Variablen zugewiesen, die als entprechender primitiver Typ deklariert ist, so erfolgt eine automatische Umwandlung in den entsprechenden Wert (unboxing).

Abbildung 7.1 zeigt einen Teil der Hierarchie der numerischen Klassen in Java. Die Klasse *Cha-*

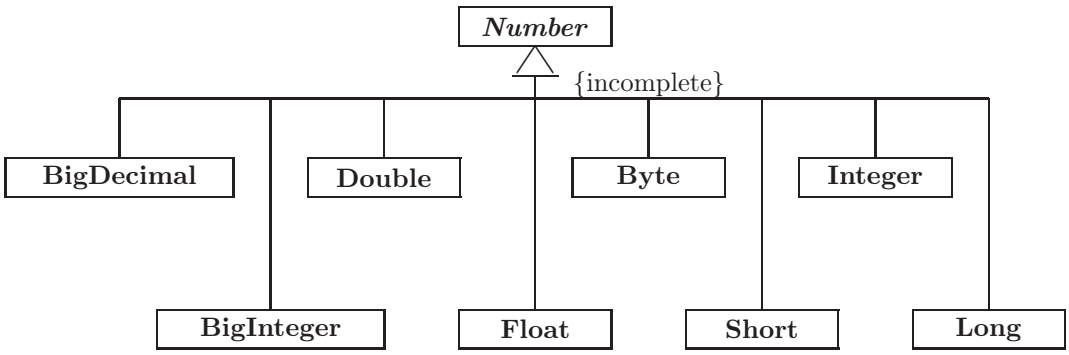


Abb. 7.1: Die numerischen Klassen in Java

racter ist keine Unterklasse von *Number*. Alle Unterklassen von *Number* haben im Unterschied zu *Character* ein Vorzeichen. Siehe hierzu auch den unsigned right shift Operator >>>.

Die Tabelle in Abb. 7.2 zeigt die Typen, Wrapperklassen und Wertebereiche.

Numerische Datentypen ohne Vorzeichen gibt es als primitive Datentypen in Java nicht. Es gibt aber in den numerischen *Wrapperklassen* Methoden für den Umgang mit solchen Datentypen (unsigned API). So gibt es in *Integer* die Klassenmethoden *divideUnsigned*, *compareUnsigned*,

Typ	Wrapper	Bezeichnung	von	bis
int	Integer	ganze Zahl	-2^{31}	$2^{31}-1$
long	Long	lange ganze Zahl	-2^{63}	$2^{63}-1$
short	Short	kurze ganze Zahl	-2^{15}	$2^{15}-1$
byte	Byte	Byte-lang ganze Zahl	-2^{7}	$2^{7}-1$
char	Character	Zeichen (kein Vorzeichen)	0	$2^{16}-1$
float	Float	Fließkommazahl	$-(2-2^{-23})\cdot 2^{127}$	$(2-2^{-23})\cdot 2^{127}$
double	Double	Doppelt genaue Fließkommazahl	$-(2-2^{-52})2^{1023}$	$(2-2^{-52})2^{1023}$

Abb. 7.2: Numerische Datentypen in Java[GJS+14]

parseUnsignedInt und *remainderUnsigned*. Da Addieren, Subtrahieren und Multiplizieren bitweise identisch sind, wenn beide Operanden unsigned (bzw. signed) sind, gibt es für diese keine unsigned Varianten.

7.4 Ganzzahlige Typen - Interna

Generell werden Zahlen oft in einem Stellensystem angeben. Dieses hat jeweils eine Basis b. Eine Zahl z wird dann als

$$z = \sum_{i=-\infty}^{\infty} z_i b^i \tag{7.1}$$

dargestellt.

Beispiel 7.4.1 (π dezimal)
Für die Zahl π ist für $b = 10$ in 7.1 mit $z_i = 0 \forall\ i > 0$, $z_0 = 3$, $z_{-1} = 1$,...

$$\pi = 3.141529\ldots \tag{7.2}$$

Beispiel 7.5.1 zeigt, wie Sie mit Java-Methoden zu der Binärdarstellung kommen. ◄

Gleichung 7.1 kann man nach dem Horner-Schema ([Rot60]) umschreiben zu:

$$\begin{aligned} z &= z_0 + b\cdot(z_1 + b\cdot(z_2 + b\cdot(z_3 + b\cdot(\ldots)))) && (7.3)\\ &\quad + b^{-1}\cdot(z_{-1} + b^{-1}\cdot(z_{-2} + b^{-1}\cdot(z_{-3} + b^{-1}\cdot(\ldots)))) && (7.4)\\ & && (7.5)\end{aligned}$$

Dies reduziert die Anzahl der Rechenoperationen erheblich. Auch die Ermittlung der Darstellung in anderen Basen wird dadurch einfacher. So erhalten Sie schnell die ersten Stellen von π in der binären Darstellung:

$$\pi = 11.001001\ldots \tag{7.6}$$

Rechnen Sie mit ganzen Zahlen, so müssen Sie auf einige Dinge achten:

1. Der Gültigkeitsbereich ist begrenzt, wenn Sie zu große oder zu kleine Rechenergebnisse erhalten, gibt es einen Überlauf (overflow).
2. Es erfolgt zwar eine Umwandlung von z. B. *int* in *long*, aber erst bei Zuweisung eines *int*'s zu eine *long* Variablen, nicht während der Rechnung.

Brauchen Sie große ganze Zahlen, so nehmen Sie *BigInteger*.

Um mit ganzen Zahlen exakt zu rechnen, dienen die Methoden *Math.addExact*, *Math.subtractExact*, ..., die im Falle des Überlaufs eine *ArithmeticException* werfen.

7.4.1 Ganze Zahlen — binär

Für ganze Zahlen stellt die Klasse *Integer* eine Klassenmethode *toBinaryString(int i)* zur Verfügung. Diese liefert die Binärdarstellung einer ganzen Zahl als String. Hieran kann man sehen, dass Java negative Binärzahlen in der 2er-Komplementform speichert: 2er-Komplement ist der „-“-Operator. Der Datentyp *int* hat 32 Bit. Für die positiven Zahlen werden 1 bis $2^{31} - 1$ verwendet, die größte ganze Zahl ist also $2^{31} - 1 = 2.147.483.648 - 1 = 2.147.483.647$, in Binärdarstellung also

0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
3			2									1										
2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0

Aufgrund der obigen Bemerkung zum „-“-Operator sind negative Zahlen also dadurch gekennzeichnet, dass im ersten Bit von links 1 steht, denn bei positiven Zahlen steht dort 0. Daher ist als $(-1)_2 = (11111111111111111111111111111111)_2$ und die kleineste ganze Zahl ist $-2^{32} = -2.147.483.648$.

Den „-“-Operator lässt sich aber auch so beschreiben: Alle Bits links vom niederstwertigen (der „ersten Eins von rechts“) Bits werden invertiert.

Auf zumindest einigen 64-Bit-Systemen, z. B. habe ich es 2009 unter MS-Vista gesehen, ist *int* 64-Bit lang. Auf dem MS-Windows 7 System, das ebenfalls 64 Bit ist, gilt dies mit Eclipse-Juno und Java 1.7.0.04 nicht. Die Java Sprachspezifikation gibt für *int* stets 32 Bit vor.

7.4.2 Ganze Zahlen - Verschiedene Basen

Positive ganze Zahlen lassen sich relativ einfach von einer Basis in eine andere umrechnen:

Haben wir es mit ganzen Zahlen mit ganzzahliger Basis zu tun, so fällt der zweite Teil der Summe im obigen Horner-Schema 7.1 weg:

$$z = z_0 + b \cdot (z_1 + b \cdot (z_2 + b \cdot (z_3 + b \cdot (\ldots)))) \tag{7.7}$$

$$\tag{7.8}$$

Nun „sieht“ man ein Verfahren zur rekursiven Berechnung der Koeffizienten in der Darstellung zur Basis b:

$$\begin{aligned} z_0 &= z \bmod \mathrm{b} \\ q &= \lfloor \frac{z}{b} \rfloor \\ z_1 &= q \bmod \mathrm{b} \\ q &= \lfloor \frac{z}{b} \rfloor \\ &\ldots \\ \text{bis } q &= 0; \end{aligned}$$

Gegeben sei eine Zahl $b_m b_{m-1} \ldots b_1 b_0$ zur Basis b gegeben, die in eine Darstellung zur Basis B umgerechnet werden soll.

Ganz einfach geht das mit Basen von 2 (MIN_RADIX) bis 36 (MAX_RADIX) mittels *Integer.toString*. Diese Methode ist überladen, es gibt *toString(int i, int radix)*, analog für *Long* und *BigInteger*. Hier ist obiger Algorithmus bereits implementiert und die „Ziffern“ größer als 9 werden für Basen größer als 10 durch die 26 Buchstaben a–z ersetzt. Wollen Sie andere Basen verwenden, so müssen Sie den Algorithmus implementieren und sich eine Darstellung für die Ziffern überlegen.

Bei der Verwendung von oktalen oder hexadezimalen numerischen Literalen müssen Sie einen Unterschied zu dezimalen numerischen Literalen beachten. Letztere sind immer positiv, es sei denn Sie schreiben „-“ davor. Oktale und hexadezimale numerische Literale haben keine explizites Vorzeichen. Sie sind negativ, wenn das höchstwertige Bit gesetzt ist. Siehe hierzu [BG05], Puzzle 5.

7.5 Fließkommazahlen — Interna

Jede Fließkommzahl in Java kann in der Form

$$s \cdot m \cdot 2^{e-N+1} \tag{7.9}$$

mit $m = \pm 1$, $0 < n < 2^N$, $e \in [E_{min}, E_{max}]$, $E_{min} = -(2^{K-1} - 2)$, $E_{max} = 2^{K-1} - 1$. dargestellt werden ([GJS+18]):

Parameter	float	float-extended exponent	double	double-extended exponent
N	24	24	53	53
K	8	≥11	11	≥ 15
E_{max}	+127	≥+1023	+1023	≥ +16383
E_{min}	-126	≤-1022	-1022	≤ -16382

Fließkommazahlen werden in Java gemäß IEEE 754 dargestellt. Solche Fließkommazahlen z werden in der Form

$$z = m \cdot b^{e-B},\ b, e, B, \in \mathbb{Z} \tag{7.10}$$

in einem Rechner dargestellt. Dabei heißt m *Mantisse*, der *Exponent* e charakterisiert die Größenordung der Zahl und b ist die *Basis*.

In Java ist für *double* der „Bias“ $B = 1075$, für *Float* ist $B = 150$

Um die Darstellung eindeutig festzulegen wird die Mantisse durch die Forderung

$$\frac{1}{B} \leq m < 1 \tag{7.11}$$

oder $1 \leq m < 2$ normiert. Intern werden diese wie folgt dargestellt:

Typ	Größe	Vorzeichen	Exponent	Mantisse	
float	32	1	8	23	Bit
double	64	1	11	52	Bit

Für den Exponenten steht also jeweils der folgende Bereich zur Verfügung:

Typ	von	bis
float	$-2^7 - 2 = -126$	$2^7 - 1 = 127$
double	$-2^{10} - 2 = -1022$	$2^{10} - 1 = 1023$

Aufgrund der Länge der Mantisse ergeben sich für die kleinste, betragsmäßig kleinste und größte Zahl die folgenden Werte:

Typ	kleinste	betragsmäßig kleinste	größte
float	$(2 - 2^{-23}) \cdot 2^{127}$	2^{-149}	$(2 - 2^{-23}) \cdot 2^{127}$
double	$-(2 - 2^{-52})2^{1023}$	2^{-1074}	$(2 - 2^{-52})2^{1023}$

Java bietet Methoden, um diese Werte zu bestimmen. So gibt es in der Klasse *Double* die Methoden *doubleToLongBits(double value)* und *longBitsToDouble*. Was die tatsächlich tun, zeige ich hier aus Platzgründen an den entprechenden Methoden aus der Klasse *Float*.

Die Grundidee, wie an die interne Darstellung heranzukommen ist, geht über die primitiven Datentypen *long* bzw. *int*. Diese haben die gleiche Länge wie *double* bzw. *Float*. Wir betrachten sie jetzt aber nicht als Zahlen sondern als Strings von Bits. Durch Anwendung der bitweisen Operatoren können wir die einzelnen Teile extrahieren.

Das Vorzeichen bekommt man einfach durch eine Methode folgender Art:

```
public static int vorzeichen(int bits){
   return (bits>>31==0)?1:-1;
}
```

Integer haben 32 Bit. Der *right shift* (>>) um 31 Bit schiebt alle Bits, bis auf das erste von links nach rechts hinaus. Das einzige noch erhaltene Bit ist das Vorzeichenbit. Für nichtnegative Zahlen ist es 0, für negative 1. Da die Methode ± liefern soll, habe ich entschieden 1 oder −1 zurückzugeben.

Danach folgt in der Binärdarstellung der Exponent: Er steht an den Stellen 30–23 (von links): Um den zu bekommen machen Sie zunächst einen Rechtsshift um 23. Bezeichnet x den Wert eines Bits, also 0 oder 1, s das Vorzeichen, so haben wir nun für positive Zahlen

```
s00000000000000000000000xxxxxxxx
```

Für negative Zahlen können links von den x noch Einsen stehen. Durch Bitweises *und* (&) mit

```
00000000000000000000000011111111 ( = 0xff)
```

schneiden wir den Exponenten heraus. Wir bekommen den Exponenten also durch

```
public static int exponent(int bits){
   return (bits>>23)&(0xff);
}
```

Als Letztes extrahieren wir nun die Mantisse. Diese steht an den Stellen 22–0 (von links). Hier brauchen wir also nicht zu shiften. Nach IEEE 754 beginnt die Mantisse immer mit 1. Dieses Bit kann also eingespart werden. Ist der Exponent 0, so elimieren wir also die überschüssigen Bits durch Bitweises *und* (&) mit

```
0x7fffff = 11111111111111111111111,
```

Ist der Exponent nicht 0 so haben wir damit die Mantisse herausgeschnitten und stellen durch oder (|) mit 0x800000 das Vorzeichen wieder her.

```
10000000000000000000000
```

Die Methode lautet also

```
public static int mantisse(int bits){
   return (exponent(bits) == 0) ?
      (bits & 0x7fffff) << 1 :
      (bits & 0x7fffff) | 0x800000;
}
```

Ist der Exponent 0 handelt es sich um eine nicht-normalisierte Fließkommazahl und wir shiften um 1 nach links.

Beispiel 7.5.1 (π)
In *Math* ist die Konstante $PI = 3.141592653589793$ definiert. Durch unsere Methoden zur Bestimmung von Vorzeichen, Exponenten und Mantisse erhalten wir hier:

```
Vorzeichen = 1
Mantisse   = 7074237752028440 (dezimal)
Mantisse   = 11001001000011111101101010100010001000010110100011000 (binär)
Exponent   = 1024
```

Damit ist dann $\pi = 7074237752028440 * 2^{-51}$ $(-51 = 1024 - 1075))$. Binär ist also $(\pi)_2 = 11.00100100001111110110101010001000100001011010001100 0$ in der Genauigkeit von *double*. ◀

Bei endlichen Dezimalbrüchen sehen Sie so genau, wie es zu Rundungsfehlern kommt. So ist z. B.

$$\begin{aligned}(0.3)_{10} &= (0.10011001100110011001100110011001100110011001100110011)_2\\ &\approx (0,29999999999999998889776975374843)_{10}\end{aligned}$$

Dieser Rundungsfehler sind ein Grund, warum Sie bei Vergleichen *Double.compare* bzw. *Float.compare* statt „<“ verwenden müssen. Noch wichtiger ist aber, dass die Vergleichsoperatoren „<“ und „>“ nicht mit *equals* kompatibel sind, während die *compare* Methoden dies sind.

Sie sollten außerdem einige spezielle Werte kennen:

```
POSITIVE_INFINITY = 0x7f800000 =  11111111000000000000000000000000
NEGATIVE_INFINITY = 0xff800000 = 11111111100000000000000000000000
NaN               = 0x7fc00000 =  11111111100000000000000000000000
```

Bemerkung 7.5.2 (NaN)
NaN zeichnet sich dadurch aus, dass es der einzige numerische Wert in Java ist, der nicht gleich sich selbst ist. Es ist nicht möglich eine Zahl auf *NaN* mit *equals* oder gar == zu prüfen. Die einzige Möglichkeit, um eine Zahl n daraufhin zu prüfen, ob sie *NaN* ist, besteht darin $n \mathrel{!}= n$ abzufragen. Das ist genau der Code, den Sie in der Methode *Double.isNaN* finden. ◀

Ferner gibt es dann noch *strictfp*. Dies bedeutet, dass alle Zwischenergebnisse gemäß IEEE 754 dargestellt werden. Andernfalls könnte je nach Plattform auch genauer gerechnet werden. Spezifiziert werden kann dies für Klassen, Interfaces und nicht-abstrakte Methoden.

Für die Fälle, in denen Sie große Zahlen brauchen oder genau rechnen wollen, gibt es die Klassen *BigInteger* und *BigDecimal*. Mit deren Objekten können Sie nicht rechnen, wie mit den primitiven Typen, also den arithmetischen Operatoren. Sie verwenden die Methoden *add*, *subtract*, *multiply* und *divide*. Für kaufmännische Rechnungen wird *BigDecimal* empfohlen, siehe z. B. [Blo08], Item 48.

Ich knüpfe nun an Puzzle 2 aus [BG05] an, um den Umgang von Java mit Fließkommazahlen genauer zu erläutern: Dort geht es um die Ausgabe, die folgende Programmzeile liefert:

```
System.out.println(2.00 - 1.10);
```

In der API-Dokumentation der Klassenmethode *Double.toString(double d)* finden Sie folgende Information (von *NaN* und ∞ sehen ich mal ab): Die Darstellung hat mindestens eine Nachkommastelle. Sie hat so viele Nachkommastellen, wie notwendig ist, um das Argument vom benachbarten *double* Wert zu unterscheiden. Die benachbarten Werte liefern die Methoden *Math.nextDown* und *Math.nextUp*.

Sie sehen hier nochmals, das *float* und *double* für kaufmännische Rechnungen nicht geeignet sind, da hier eine exakte Darstellung von $0,1$ und $0,01$ benötigt wird, wenn die Währung Bruchteile hat, wie Cents in Euro oder Dollar.

Die Lösung ist oft die Verwendung von *BigDecimal*. Achten Sie aber darauf, den Konstruktor *BigDecimal(String)* zu verwenden! Verwenden Sie nie den Konstruktor *BigDecimal(double)*. Letzterer liefert ein *BigDecimal* mit dem exakten Wert des übergebenen *double* Werts. Der ist aber gerade nicht exakt gleich z. B. 0.1! Siehe hierzu auch Kap. 27.

7.6 Spezielle Zahlensysteme

7.6.1 Fibonacci Zahlen

Die Fibonacci-Zahlen sind rekursiv definiert:

$$f_0 = 0 \tag{7.12}$$
$$f_1 = 1 \tag{7.13}$$
$$f_{n+2} = f_{n+1} + f_n \text{ für } n > 0 \tag{7.14}$$
$$\tag{7.15}$$

Insofern sind sie ein klassisches Beispiel um rekursive Programmierung zu illustrieren.

Diese können auf viele Weisen berechnet werden. Ich nenne hier nur vier. Diese alleine zeigen aber viel darüber, was beim Umgang auch mit ganzen Zahlen zu beachten ist.

rekursiv Diese Berechnung verwendet direkt die obige Definition, z. B. im Kontext einer statischen Methode `fibonacci(int n)`:

```
        public static int fibRec(int n){
           if(n==0)
             return 0;
           if(n==1)
             return 1;
           return (fibRec(n-1) + fibRec(n-2));
}
```

Das ist eine sehr ineffiziente Methode zur Berechnung der Fibonacci-Zahlen (siehe *Algorithmen und Datenstrukturen* im dritten Semester).

Array Mit etwas mehr Speicher geht es schneller und man kann größere Zahlen effektiv bestimmen:

```
        public static int [] fibarray(int n){
           int [] f = new int[n+1];
           f[0] = 0;
           f[1] = 1;
           for(int i= 2;i <= n;i++)
                f[i] = f[i-1] + f[i-2];
           return f;
}
}
```

Mit dieser Operation kommt man so bis zu f_{46}.

Iterativ mit drei lokalen Variablen: Hier eine mal so eben hingeschriebene und getestete Version:

```
public static int fibternary(int n){
        if(n == 0){
            return 0;
        }
        if(n==1){
            return 1;
        }
    int f0 = 0;
    int f1 = 1;
    int f2 = f0 + f1;
    int i = 2;
    while(i <= n){
        f2 = f0 + f1;
        f0 = f1;
        f1 = f2;
    }
    return f2;
}
```

Formel Man kann zeigen, dass Folgendes gilt: Sind

$$\Phi = \frac{1+\sqrt{5}}{2} \text{ und } \hat{\Phi} = \frac{1+\sqrt{5}}{2},$$

so gilt:

$$f_n = \frac{1}{\sqrt{5}}(\Phi^n - \hat{\Phi}^n)$$

Macht man sich diese Beziehung zu Nutze, so kann man f_n so berechnen:

```
public static int fibgolden(int n){
   double sqrt5 = sqrt(5);
   double phi = (1 + sqrt5)/2;
   double phihat = (1 - sqrt5)/2;
   return (int) ((pow(phi,n) - pow(phihat,n))/sqrt5);
}
```

Auch hier treten ab f_{47} Rundungsfehler bzw. Überlauf auf.

BigInteger Mit der Klasse *BigInteger* aus dem Paket *java.math* kommmt man weiter.

Man kann zeigen ([Knu73], 1.2.8, ex 34 [Zec72], [Lek52]:

> Jede positive ganze Zahl kann als Summe von Fibonacci-Zahlen geschrieben werden. Schränkt man ein, dass dazu keine zwei direkt auf einander folgende Fibonacci-Zahlen verwendet werden dürfen, so ist diese Darstellung eindeutig.

Damit können wir also jede positive ganze Zahl als eine Folge von 0 und 1 in diesem System schreiben, etwa $(64)_{10} = (100010001)_{fib}$.

Die ist die sog. Zeckendorf-Darstellung (siehe http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/Fibonacci/fibrep.html#fibbase. Diese Darstellung verwendet die wenigsten Fibonacci-Zahlen. Man kann aber auch versuchen eine Darstellung zu finden, die die meisten Fibonacci-Zahlen verwendet.

7.7 Einige numerische Beispiele

Ich schreibe mir eine Utility-Klasse, um einige Grundprinzipien zu illustrieren. Die Klasse nenne ich *Mathe*, um Sie von der Java-Klasse *Math* zu unterscheiden. Ich beginne mit einer rekursiven und einer iterativen Berechnung der Fakultät nichtnegativer ganzer Zahlen.

```
public static int FakRek(int n){
   return n==0?1:n*FakRek(n-1);
}
```

Natürlich habe ich mir gleich Testfälle geschrieben. Aber zunächst muss man sich überlegen, für welche n das funktionieren kann. Dabei hilft die Stirlingsche Formel ([Mes71]):

$$n! \sim \sqrt{2\pi n} \cdot (\frac{n}{e})^n$$

Der größte *int* Wert ist $2^{31} - 1$. Schon mittels eines Taschenrechners sieht man, dass das mit diesen Operationen nur bis $n = 12$ gut gehen kann. Ferner haben wir noch keine Sorge dafür getragen, dass diese Operationen bei Aufruf mit negativen Werten einen geeigneten Fehler oder Wert liefern. Für den Datentyp *long* kommen wir bis $n = 20$.

Natürlich kann man das auch iterativ machen, z. B. so:

```
public static int FakIter(int n){
   int fak=1;
   for(int i=2;i<=n;i++){
       fak*=i;
   }
   return fak;
}
```

7.8 Zahlenalgorithmen und Anwendungen

Es gibt viele Algorithmen für ganze Zahlen und Fließkommazahlen in den Java-Klassen. Spätestens, wenn Sie sich ernsthaft mit der Entwicklung von Algorithmen beschäftigen, werden Sie noch weitere benötigen.

Hier einige einfache Beispiele.

Beispiel 7.8.1 (lowestOneBit)
In den Klassen *Integer* und *Long* finden Sie Methoden namens *lowestOneBit* mit einem Parameter *j* vom Typ *int* bzw. *long*. Diese Methode liefert die entsprechende ganze Zahl, die in ihrer Binärdarstellung genau eine 1 hat, nämlich dort, wo die übergebene ganze Zahl die „rechteste“ 1 in ihrer Binärdarstellung hat. Der Code ist ganz kurz:

```
return j & -j;
```

Versuchen Sie das bitte selber nachzurechnen. Hilfestellung finden Sie ggf. in Kap. 8. ◄

Beispiel 7.8.2 (highestOneBit)
Etwas länger ist der Code für die entsprechenden Methoden *highestOneBit*. Diese Methode liefert die entsprechende ganze Zahl, die in ihrer Binärdarstellung genau eine 1 hat, nämlich dort, wo die übergebene ganze Zahl die „linkeste“ 1 in ihrer Binärdarstellung hat.

Der Code der Methode muss die Länge der Zahlen (32 bzw. 64 Bit) berücksichtigen. Für Integer etwa:

```
i |= (i >>  1);
i |= (i >>  2);
i |= (i >>  4);
i |= (i >>  8);
i |= (i >> 16);
return i - (i >>> 1);
```

Siehe hierzu auch Kap. 8. ◄

Ein konzeptionell einfacher, wenn auch schwierig zu analysierender Algorithmus ist der Euklidische Algorithmus zur Berechnung des größten gemeinsamen Teilers (ggt, Englisch gcd, greatest common divisor) zweier ganzer Zahlen [Knu97a, Knu97b]. Sie finden einige mögliche Implementierungen in der Utility-Klasse *IntegerFunctions* im Paket *numbers*, z. B. diese:

```
public static long ggT(long m, long n) {
   if (m < n) {
      long tmp = n;
      n = m;
      m = tmp;
   }
   return  n == 0 ? m : ggTnotZero(m, n);
}
private static long ggTnotZero(long m, long n) {
   long rest = m % n;
   while (rest != 0) {
      m = n;
      n = rest;
      rest = m % n;
   }
   return n;
}
```

7.9 Historische Anmerkungen

strictfp wurde in Java 1.2 eingeführt. Methoden für unsigned arithmetic in Java 8 im Jahre 2014.

7.10 Aufgaben

1. Schreiben Sie bitte eine Klasse *Rational*, deren Objekte die rationalen Zahlen $\mathbb{Q}$ darstellen! Demonstrieren Sie an Hand geeigneter Testfälle, das Ihre Implementierung funktioniert.

2. Schreiben Sie bitte eine Klasse *Complex*, deren Objekte die komplexen Zahlen $\mathbb{C}$ darstellen! Achten Sie bitte darauf, die Division effizient zu implementieren! Demonstrieren Sie bitte an Hand geeigneter Testfälle, das Ihre Implementierung funktioniert!

3. Schreiben Sie eine Klasse *BigComplex*, deren Objekte beliebig genaue komplexen Zahlen darstellen! Demonstrieren Sie bitte an Hand geeigneter Testfälle, das Ihre Implementierung funktioniert!

4. Die Zeckendorf Darstellung verwendet die minimale Anzahl von Fibonacci-Zahlen. Bestimmen Sie die entsprechende Darstellung, die die jeweils meisten Fibonacci-Zahlen verwendet!

5. $(4)_{10} = (101)_{fib}$. Dies ist die einzige Darstellung von 4 durch Fibonacci-Zahlen. Welche anderen Zahlen haben genau eine Darstellung als Fibonacci-Zahlen?

6. Untersuchen Sie die Anzahl Einsen in der Zeckendorf-Darstellung? Erkennen Sie ein Schema? Können Sie dies in Formeln fassen?

7. Untersuchen Sie die entsprechende Frage für die Anzahl Einsen in der maximalen Darstellung durch Fibonacci-Zahlen!

8. Schreiben Sie bitte eine rekursive Methode, die prüft, ob eine ganze Zahl n durch Drei teilbar ist!

9. Schreiben Sie Code, der alle 10-stelligen Zahlen generiert, die durch $2, 3, 4, 5, 6, 7, 8, 9$ teilbar sind!

10. Wie prüfen Sie, ob eine *double* Variable „Not a Number“, also *NaN* ist?

11. Die russische Bauern oder äthiopische Multiplikation zweier natürlicher Zahlen ist wie folgt definiert: Seien a und b die zu multiplizierenden Zahlen. Dann halbiere man a ganzzahlig bis man bei 1 angekommen ist. b wird jeweils verdoppelt. Ist in einem Schritt der Bruchteil von a gerade, so streiche man das Vielfache von b auf der rechten Seite. Die Summe der verbleibenden Zahlen in der rechten Spalte ist dann gerade $a \cdot b$ [Scr68]. Implementieren Sie dieses Verfahren!

12. Welche Ausgabe liefert die folgende *main*-Methode? Was ist daran irritierend? [BG05].

```
/**
 * What's irritating with the calculation?
 * @author Joshua Bloch
 * @author Neal Gafter
 */
public class Elementary {
    public static void main(String[] args) {
        System.out.println(12345 + 54321);
    }
}
```

13. Ich definiere die *Quersumme* einer ganzen Zahl n als die Summe ihrer Ziffern. Die *totale Quersumme* einer ganzen Zahl n ist wie folgt definiert: Es wird die *Quersumme* gebildet. Ist sie einstellig, so ist es die *totale Quersumme*. Andernfalls wird von dieser wieder die *Quersumme* gebildet, bis das Ergebnis einstellig ist. Beweisen Sie bitte: Die *totale Quersumme* von n ist
$$Quer(n) = \begin{cases} 9 & n\%9 = 0 \wedge n \neq 0 \\ n\%9 & andernfalls \end{cases}$$

14. Welche Ausgabe liefert die folgende Zeile?

```
System.out.println(Long.toHexString(0x100000000L + 0xcafebabe));
```

Erläutern Sie bitte genau, wie das Ergebnis zu Stande kommt! Quelle: [BG05]

15. Welche Ausgabe liefert die folgende Zeile? Warum?

```
System.out.println((int) (char) (byte) -1);
```

[BG05]

16. Der ==-Operator definiert auf den primitiven Typen (*int*, *double* etc.) keine *Äquivalenzrelation*. Welche Bedinungen sind verletzt? Geben Sie bitte jeweils ein Beispiel!

17. Schreiben Sie bitte Deklarationen für die Variablen `i` und `x`, so dass `x += i;` legal ist, nicht aber `x = x + i;`![BG05]

18. Bis Java 6 konnten Sie auch Deklarationen für die Variablen `i` und `x` schreiben, so dass `x = x + i;` legal ist, nicht aber `x += i;` Seit Java 7 sind ggf. beide Konstrukte für das frühere Beispiel legal (`Object x = "Try"; String  i = "again";`).[BG05]

Kapitel 8

Bitweise Operationen

8.1 Übersicht

Heutige Rechner sind und werden voraussichtlich noch lange Digitalrechner sein. Von daher muss ein Informatiker sich mit Bits und Bytes auskennen. Dies ist Manchen ein Gräuel, vielleicht weil sie während der Schulzeit mit Mathematik auf dem Kriegsfuß standen. Andererseits birgt diese Kenntnis auch ein großes Potenzial. Nutzt man diese Architektur aus, so kann man Dinge effizienter tun, als wenn dies nicht geschieht. Die theoretische Basis für dieses Kapitel bilden die Abschnitte 7.1.1 – 7.1.3 von TAOCP ([Knu08a, Knu07, Knu08b]). Einiges finden Sie auch in [War02]. Ich verzichte hier weitgehend auf Theorie und konzentriere mich auf die exemplarische Umsetzung in Java.

Bemerkung 8.1.1 (Nutzen dieses Kapitels)
Dieses Kapitel wird je nach Ihrer Interessenausrichtung von unterschiedlichem Nutzen sein:

- Wer sich überhaupt nicht für Interna im Rechner interessiert, kann daraus vielleicht Nutzen ziehen, wenn es um die Vorbereitung für Prüfungen im Bereich Grundlagen der Informatik geht.
- Wer solche Dinge nur ab und zu benötigt, kann es hoffentlich in späteren Auflagen zum Nachschlagen verwenden.
- Wer solche Dinge braucht und sich dafür interessiert findet Hinweise auf weiterführende Literatur.

◀

8.2 Lernziele

- Bitweise Operationen in Java lesen können.
- Bitweise Operationen in Java verstehen.
- Bitweise Operationen in Java verwenden können.
- Bitmanipulationen und deren Nutzen kennen.

8.3 Grundlagen

Tabelle 1 in [Knu08a] zeigt die 16 booleschen Funktionen zweier Variablen. Hier finden Sie in Abb. 8.1 eine Übersicht über die Bezeichnungen und Namen. Als erstes Beispiel zeige ich die

Notationen	Op Sym.	Op Java	Name(n)
0	$\bot$	false	Widerspruch, immer falsch
$xy, x \wedge y, x\&y$	$\wedge$	&, &&	Bitweises bzw. logisches und
$x \wedge \overline{y}, x \not\supset y, [x > y]$, x-y	$\overline{\supset}$	x&!y	Nichtfolgerung, Differenz, aber nicht
x	$\llcorner$	x	Linksprojektion
$\overline{x} \wedge y$	$\overline{\subset}$	!x&y	Inverse Nichtfolgerung, nicht ... aber
y	R	y	Rechtsprojektion
$x \oplus y, x \not\equiv y, xy$	$\oplus$	$x\hat{}y$	Exklusives oder, xor
	$\vee$	|, ||	bitweises bzw. logisches oder
$\overline{x} \wedge \overline{y}$	$\overline{\vee}$		Weder ... noch
$x \equiv y$	$\equiv$		Äquivalenz; genau dann, wenn; iff
$\overline{y}, \neg y, !y, \sim y$	$\sqsubset$		Rechtskomplement
$x \vee \overline{y}, x \subset y, x \Leftarrow y, [x \geq y], x^y$	$\subset$		Inverse Folgerung, wenn
$\overline{x}$	$\overline{R}$		Linkskomplement
$x \Rightarrow y$	$\supset$		Folgerung;nur wenn; wenn ... dann;
$\overline{x} \vee \overline{y}$	$\overline{\wedge}$		nand; nicht beide ... und; Nicht-und
1	$\top$	true	Tautologie

Abb. 8.1: Die booleschen Funktionen zweier Variablen

Implementierung dieser Tabelle in Java. Wie zeigt man dies am Besten? Wir haben zwei boolesche Variablen a und b und 16 Operationen. Zur Wahl stehen zwei Darstellungen:

1. Eine Matrix mit 18 Zeilen für die Werte der Operatoren und 5 Spalten für die Werte der Operandenpaare und das Ergebnis.

2. eine Matrix mit 4 Zeilen für die Wertepaare der Operanden und 18 Spalten für die Werte und die Ergebnisse der jeweiligen Operatoren.

Das Programm dazu ist ganz einfach

```
  private static void printTruthtable(){
boolean [] arg = {false, true};
System.out.printf("%6s%6s%6s%6s%6s%6s%6s%6s%6s%6s%6s%6s%6s%6s%6s%6s%6s%6s\n",
  "b","c","false","b&&c","b&&!c","b","!b&&c","c","b^c","b||c","!b&&!c","b==c","!c","b||!c",
  "!b","!b||c","!b||!c",true);
for(boolean b:arg){
for(boolean c:arg){
System.out.printf("%6b%6b%6b%6b%6b%6b%6b%6b%6b%6b%6b%6b%6b%6b%6b%6b%6b%6b\n",
  b,c,false,b&&c,b&&!c,b,!b&&c,c,b^c,b||c,!b&&!c,b==c,!c,b||!c,
  !b,!b||c,!b||!c,true);
}
}
```

und das Ergebnis ist

```
     b     c false  b&&c b&&!c     b !b&&c     c   b^c  b||c!b&&!c  b==c    !c b||!c    !b !b||c!b||!c  true
 false false false false false false false false false false  true  true  true  true  true  true  true  true
 false  true false false false  true  true  true false false false false  true  true  true  true  true  true
  true false false false  true  true false false  true  true false false  true  true false false  true  true
  true  true false  true false  true false  true false  true false  true false  true false  true false  true
```

Ganz analog bekommt man auch eine Ausgabe als LaTeX-Tabelle hin:

b	c	false	b&&c	b&&!c	b	!b&&c	c	b^c	b||c	!b&&!c	b==c	!c	b||!c	!b	!b||c	!b||!c	true
false	false	false	false	false	false	false	false	false	false	true	true	true	true	true	true	true	true
false	true	false	false	false	false	true	true	true	true	false	false	false	false	true	true	true	true
true	false	false	false	true	true	false	false	true	true	false	false	true	true	false	false	true	true
true	true	false	true	false	true	false	true	false	true	false	true	false	true	false	true	false	true

Das Ergebnis überzeugt in DIN A4 Hochformatdarstellung noch nicht. Also schreibe ich auch noch eine Methode *printTruthtableTransposed()* für die transponierte Darstellung. Diese liefert die folgende Tabelle. Die Zeilenköpfe sind dabei in den ersten beiden Zeilen die Wahrheitswerte der Variablen und in den folgenden 16 die Operatoren. In den Matrixzellen steht die Auswertung des Operators für die Wahrheitswertpaare in den ersten beiden Zeilen.

```
     b false false  true  true
     c false  true false  true
 false false false false false
  b&&c false false false  true
 b&&!c false false  true false
     b false false  true  true
 !b&&c false  true false false
     c false  true false  true
   b^c false  true  true false
  b||c false  true  true  true
!b&&!c  true false false false
  b==c  true false false  true
    !c  true false  true false
 b||!c  true false  true  true
    !b  true  true false false
 !b||c  true  true false  true
!b||!c  true  true  true false
  true  true  true  true  true
```

oder im LaTeX-Format

b	false	false	true	true
c	false	true	false	true
false	false	false	false	false
b&&c	false	false	false	true
b&&!c	false	false	true	false
b	false	false	true	true
!b&&c	false	true	false	false
c	false	true	false	true
b^c	false	true	true	false
b||c	false	true	true	true
!b&&!c	true	false	false	false
b==c	true	false	false	true
!c	true	false	true	false
b||!c	true	false	true	true
!b	true	true	false	false
!b||c	true	true	false	true
!b||!c	true	true	true	false
true	true	true	true	true

Der Code ist natürlich wenig elegant. Einfacher geht es, wenn wir einfach die Matrix aus dem ersten Beispiel transponieren. So lernen wir gleich ein zweidimensionales Array kennen. Wir schreiben eine Operation wie *printTruthtable()*, geben das Ergebnis aber nicht gleich aus, sondern schreiben es in passende Arrays.

8.4 Anwendungen

Aus Abschn. 7.4.1 wissen Sie bereits, wie Sie aus einer Zahl j die Binärdarstellungen $(j)_2$ und $(-j)_2$ bilden. Zur Erinnerung: $(-j)_2$ erhalten Sie aus $(j)_2$, in dem Sie zunächst das niederwertigste (am

weitesten rechts) stehenden 1 Bit aufsuchen. Dieses und die ggf. rechts davon stehenden 0 Bits bleiben erhalten. Alle links davon stehenden Bits werden invertiert. Mit $\overline{x} = 0$ für $x = 1$ und $\overline{x} = 1$ für $x = 0$ erkennen Sie nun

$$\begin{array}{rcccccc} \texttt{j} & = & x \ldots x & 10 & \ldots & 0 \\ \texttt{-j} & = & \overline{x} \ldots \overline{x} & 10 & \ldots & 0 \\ \texttt{j\&-j} & = & 0 \ldots 0 & 10 & \ldots & 0 \end{array}$$

j&-j liefert also das niederwertigste 1 Bit und genauso finden Sie es auch in der Implementierung von *lowestOneBit* der Klassen *Integer* und *Long*.

Das höchstwertige 1 Bit (highest one bit) eines *int* finden Sie so:

```
i |= (i >>  1);
i |= (i >>  2);
i |= (i >>  4);
i |= (i >>  8);
i |= (i >> 16);
return i - (i >>> 1);
```

Genau dieser Code steht in der Methode *Integer.HighestOneBit*.

Einige Rechenregeln:

$$x\&y = y\&x, \quad x|y = y|x, \quad x \oplus y = y \oplus x \tag{8.1}$$
$$(x\&y)\&z = x\&(y\&z), \quad (x|y)|z = x|(y|z), \quad (x \oplus y) \oplus z = x \oplus (y \oplus z) \tag{8.2}$$
$$(x|y)\&z = (x\&y)|(x\&z), \quad (x\&y)|z = (x|z)\&(y|z) \tag{8.3}$$
$$(x \oplus y)\&z = (x\&z) \oplus (y\&z) \tag{8.4}$$
$$(x\&y)|x = x, \quad (x|y)\&x = x \tag{8.5}$$
$$(x\&y) \oplus (x|y) = x \oplus y \tag{8.6}$$
$$x\&0 = 0, \quad x|0 = x, \quad x \oplus 0 = x \tag{8.7}$$
$$x\&x = x, \quad x|x = x, \quad x \oplus x = 0 \tag{8.8}$$
$$x\&-1 = x, \quad |-1 = -1, \quad x \oplus -1 = \overline{x} \tag{8.9}$$
$$x\&\overline{x} = 0, \quad x|\overline{x} = -1, \quad x \oplus \overline{x} = -1 \tag{8.10}$$
$$\overline{x\&y} = \overline{x}|\overline{y}, \quad \overline{x|y} = \overline{x}\&\overline{y}, \quad \overline{x \oplus y} = \overline{x} \oplus y = x \oplus \overline{y} \tag{8.11}$$
$$-x = \overline{x} + 1, \quad -x = \overline{x - 1}, \quad \overline{x - y} = \overline{x} + y \tag{8.12}$$
$$x \bmod 2^n = x\&(2^n - 1) \tag{8.13}$$
$$\tag{8.14}$$

Den Spezialfall $n = 1$ der letzten Rechenregel kennen Sie bereits aus der Methode *isOdd* aus dem Kap. 5, Aufgabe 3.

Hier nun noch einige weitere nützliche Zusammenhänge:

Ausdruck	Wirkung
i&(i-1)	Schaltet das niederwertigste 1-bit aus
i&(i+1)	Prüft $i = 2^n - 1$
!i&(i+1)	Isoliert das rechteste 0 Bit.
!i&(i-1)	Bestimmt die rechtesten zusammenhängenden 0 Bits
!(i|-i)	
(i&-i)-1	

8.5 Binärbäume

Bäume und insbesondere Binärbäume werden Ihnen in Ihrem Studium und in der Praxis immer wieder begegnen. Sogenannte *vollständige Binärbäume* lassen sich effizient in einem Array speichern. Ein solcher Baum wird in der Informatik von oben nach unten dargestellt. Der Einfachheit

halber lasse ich die Position 0 im Array leer und verwende nur die Plätze von 1 bis $n-1$ (n: Länge des Arrays). Ein Element des Baums hießt *Knoten*. Ein Knoten hat kein, ein oder zwei Nachfolger („Kinder"). Befindet sich der Knoten an Position k des Arrays so sind die Kinder die Knoten an den Stellen $2 \cdot k$ (linkes Kind) und $2 \cdot k+1$ (rechtes Kind). Statt der Multiplikation mit 2 verwendet ein Informatiker aber einen links-Shift um 1: $k << 1$. Der Vorgänger eines Knotens an der Postion k (außer der Wurzel) befindet sich an der Position $k >> 1$. Brauchen Sie also einmal einen Binärbaum, z. B. einen sogenannten Heap und müssen ihn selbst programmieren, erinnern Sie sich bitte an diesen Abschnitt.

Eine etwas andere nützliche Struktur erhalten Sie durch eine andere Bitmanipulation: Hier startet der Aufbau wieder mit dem Knoten an der Position 1 des Arrays.

1. Der linke Nachfolger von $a[j]$ ist $a[j - (j\& - j) >> 1]$ für gerade j.
2. Der rechte Nachfolger von $a[j]$ ist $a[j + (j\& - j) >> 1]$ für gerade j
3. Arrayelemente mit ungeraden Indizes haben keine Nachfolger.
4. Der Vorgänger (parent) von $a[j]$ ist $a[(j - (j\& - j))|((j\& - j) << 1)]$
5. Der Nachbar von $a[j]$ ist $a[j\hat{}((j\& - j) << 1)]$

Zeichnen Sie sich das auf, so sehen Sie das die Struktur von links unten nach rechts und oben wächst.

8.6 Historische Anmerkungen

8.7 Aufgaben

1. Geben Sie bitte Bit für Bit an, wie Sie aus der Binärdarstellung $(j)_2$ einer natürlichen Zahl $j \geq 0$ die Binärdarstellung von $(-j)_2$ erhalten.
2. Erklären Sie bitte die angegebenen Code-Teile, um das niederwertigste 1 Bit einer Zahl (lowest one bit)bzw. das höchstwertigste 1 Bit zu erhalten!
3. Wie finden Sie das höchstwertiges Bit eines *byte*, *short*, *long*?
4. Wie stellen Sie am einfachsten fest, ob eine Zahl gerade ist?
5. Implementieren Sie die booleschen Funktionen zweier Variablen aus Abb. 8.1 als Klassenmethoden einer Utility-Klasse.

Kapitel 9

Mehr über Klassen

9.1 Übersicht

Nicht alles über Klassen wollte ich in das Kapitel 4 zwängen. Schließlich geht es in Java immer um Klassen und Interfaces, wenn man von den primitiven Typen einmal absieht. Klassen sind in Java mit weitem Abstand das wichtigste Element. Es gibt aber auch Elemente von Java, deren Sinn ein Anfänger nicht unbedingt gleich zu Beginn einer Ausbildung leicht verstehen kann. Auch deren sinnvoller Einsatz benötigt weitere Vorkenntnisse. Daher präsentiere ich in diesem Kapitel einige Java-Spezifika für die Realisierung objektorientierter Prinzipen und die konsequente Nutzung von Beziehungen, wie Vererbung und Implementierung von Schnittstellen. Dort fügen sich dann auch anonyme und andere innere Klassen ganz natürlich ein.

9.2 Lernziele

- Wissen, was es bedeutet, wenn in Java eine Klasse eine Schnittstelle (Interface) implementiert (implements).
- Aktiv wissen, was es in Java heißt, wenn eine Schnittstelle eine andere erweitert (extends).
- Schnittstellen implementieren können.
- Wissen, was eine abstrakte Klasse ist.
- Die Unterschiede zwischen abstrakter Klasse und Schnittstelle (Interface) erläutern können.
- Innere Klassen kennen und verwenden können.
- Lokale Klassen kennen und verwenden können.
- Anonyme Klassen kennen und verwenden können.
- λ-Ausdrücke kennen und verwenden können.
- Sachgerecht zwischen dem Einsatz einer anonymen Klasse und einem λ-Ausdruck entscheiden können.

9.3 Initialisierung

Dies ist ein Thema, über das Sie stets stolpern werden, wenn Sie etwas vergessen haben zu initialisieren. Deshalb hier eine weitgehend vollständige Darstellung. Wenn eine Variable in Java nur

deklariert, aber nicht initialisiert wurde, ist sie *null*. In einigen Fällen sorgt Java aber für eine automatische Initialisierung.

Java stellt folgende Initialisierungen für Attribute automatisch sicher, ohne dass Sie dafür etwas tun müssen:

Typ	Automatische Initialisierung
byte	0
short	0
int	0
long	0L
float	0.0F
double	0.0
Referenztyp	null

Für die weitere Darstellung muss ich zwischen Klassen- und Instanzattributen unterscheiden. Klassenattribute können auf zwei Weisen initialisiert werden:

1. Direkt bei der Deklaration. Dies ist dann möglich und sinnvoll, wenn die Initialisierung in einem Befehl erfolgen kann.

2. In einem sogenannten statischen Initialisierungsblock:

   ```
   static {
      ...//Initialisierungscode
   }
   ```

   Dies ist dann sinnvoll bzw. notwendig, wenn die Klassenattribute durch mehr als einen Befehl initialisiert werden müssen.

Da in Konstruktoren auf Klassenattribute zugegriffen werden kann, müssen Klassenattribute bereits sinnvoll initialisiert sein, wenn neue Objekte angelegt werden.

Spätestens hier muss ich anfangen zu beschreiben, wie Java zur Laufzeit funktioniert: Wenn der Name einer Klasse zum ersten Mal im Quellcode vorkommt (außer in import Statements), wird die zugehörige *.class*-Datei geladen. Dabei werden dann auch alle Klassenattribute initialisiert.

Statische Initialisierungsblöcke werden genau einmal in der Reihenfolge ihres Auftretens aufgerufen, wenn die Klasse geladen wird. Für die Details der Initialisierung verweise ich auf die Java-Sprachspezifikation [GJS+14], Abschn. 12.4.

Als besser gilt es, eine statische Initialisierungsmethode zu schreiben. Ist diese *protected*, so kann sie von spezialisierten Klassen genutzt werden.

Für den Code in einem statischen Initialisierungsblock gelten einige Verbote. Die folgende Konstrukte führen zu einem Compiler-Fehler, was auch logisch ist, wenn man die Definition von Klassenattributen und die Reihenfolge der Initialisierung in Java berücksichtigt.

- Ein statischer Initialisierungsblock darf nicht abrupt abbrechen [GJS+14], Abschn. 14.21, d. h.
  - In einem statischen Initialisierungsblock darf keine *exception* geworfen werden, also weder eine *checked* noch eine *RuntimeException* (siehe hierzu Kap. 12).
  - Es darf kein *break* mit oder ohne Label geben.
  - Es darf kein *continue* mit oder ohne Label geben.
  - Es darf kein *return* geben.
- Die Schlüsselworte *super* und *this* sind nicht erlaubt.
- Attribute, die außerhalb des statischen Initialisierungsblock deklariert sind, können in diesem nicht verwendet werden; insbesondere können Instanzattribute nicht verwendet werden. Letzteres ergibt sich bereits daraus, dass ein statischer Initialisierungsblock bereits beim Laden der Klassendatei ausgeführt wird.

Generell gilt: Statische Elemente können nicht auf Instanzelemente zugreifen. Beispiel 9.3.1 zeigt, dass rekursive Initialisierungsversuche ignoriert werden.

Beispiel 9.3.1 (Rekursive Initialisierung)
Das Puzzle 49 aus [BG05]:

```
00 public class Elvis {
05    public static final Elvis INSTANCE = new Elvis();
10    private final int beltSize;
15    private static final int CURRENT_YEAR =
20        Calendar.getInstance().get(Calendar.YEAR);
25    private Elvis() {
30        beltSize = CURRENT_YEAR - 1930;
35    }
40    public int beltSize() {
45        return beltSize;
50    }
55    public static void main(String[] args) {
60        System.out.println("Elvis wears a size " +
65                           INSTANCE.beltSize() + " belt.");
70    }
75 }
```

illustriert einen solchen vergeblichen Initialisierungsversuchs. Die Hypothese ist, Elvis (Presley) lebe noch, habe aber pro Jahr einen Zoll Umfang zugelegt. Die Ausgabe ist aber immer „Elvis wears a size -1930 belt“. Erklären Sie bitte diesen Effekt (Aufgabe 7)! ◀

Alle Einzelheiten der Initialisierung von Klassen und Interfaces finden Sie in [GJS+14], Abschn. 12.4.

Ein völlig sinnloses Beispiel für einen statischen Initialisierungsblock finden Sie in der Klasse *classes.SoNicht.java* im Projekt Fehler.

Instanzattribute können bei der Deklaration initialisiert werden, in einem Initialisierungsblock oder in einem Konstruktor. Die ersten beiden Möglichkeiten bezeichnet bezeichnet man als *initializer*, ebenso die entsprechenden Initialisierungen bei Klassenattributen. Für alle gilt:

> Initializer werden in der Reihenfolge ihres Auftretens im Sourcecode ausgeführt.

Das kann dazu führen, dass ein Attribut dreimal initialisiert wird: Zunächst mit dem *default*-Wert, dann durch einen Initializer bei der Deklaration und dann noch im Konstruktor.

9.4 Schnittstellen

Es heißt, die Informatik sei diejenige Wissenschaft, die davon überzeugt ist, dass jedes Problem durch eine zusätzliche Indirektionsebene lösbar ist[1]. Technisch handelt es sich dabei oft um eine Schnittstelle.

Eine *Schnittstelle* oder *Interface* ist zunächst einmal eine Sammlung öffentlicher Operationen. In Java wird unabhängig von Schnittstelle oder Klasse von „Methode“ gesprochen. Das trägt der Tatsache Rechnung, das ein Interface in Java auch default-Implementierungen für Operationen enthalten kann, sog. *default-Methode*n.

Bemerkung 9.4.1 (Restriktionen für default-Methoden)
Eine default-Methode kann keine nicht-private Methode von *Object* überschreiben, genauer override äquivalent zu einer nicht-privaten Methode von *Object* ist [GJS+17] §9.4.1.2. ◀

[1] Ich habe diesen Spruch 1999 auf der 25 Jahrfeier der Technischen Informatik an der HAW von Volker Claus gehört. In [OW07] wird er Butler Lampson zugeschrieben.

In Java kann eine Schnittstelle auch Klassenattribute haben, die dann automatisch *final* sind. Ein *final* Attribut kann nach der Initialisierung nicht mehr verändert werden, ist also eine Konstante. Darüber hinaus kann ein Interface weitere Interfaces und Klassen enthalten. Annotationstypen sind ebenfalls Interfaces und werden in Kap. 20 behandelt.

Viele wichtige Interfaces enthalten nur öffentliche Operationen. Ein Beispiel für ein Interface, das Sie oft verwenden werden ist *java.util.List*:

```
public interface List<E> extends Collection<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator<E> iterator();
    Object[] toArray();
    <T> T[] toArray(T[] a);
    boolean add(E e);
    boolean remove(Object o);
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean addAll(int index, Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();
    boolean equals(Object o);
    int hashCode();
    E get(int index);
    E set(int index, E element);
    void add(int index, E element);
    E remove(int index);
    int indexOf(Object o);
    int lastIndexOf(Object o);
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);
    List<E> subList(int fromIndex, int toIndex);
}
```

Inzwischen sind noch einige wichtige default-Methoden hinzugekommen, die von *Collection* bzw. *Iterable* geerbt werden: *parallelStream*, *removeIf*, *stream* und *forEach*. Stören Sie sich bitte nicht an dem *Typparameter E*, der hier verwendet wird. Wenn Sie *List* verwenden, setzen Sie hierfür einfach einen *Referenz-Typ* ein. Interfaces sind in Java ein ganz wichtiges Element, um die Implementierung von der Schnittstelle zu trennen. Eine bewährte Faustregel besagt, dass immer für eine Schnittstelle programmiert werden soll.

Für die Implementierung einer Liste gibt es verschiedene Möglichkeiten. Zwei wichtige Implementierungen des Interfaces *List* finden Sie für Java in *java.util.ArrayList* und *java.util.LinkedList*.

In beiden Klassendefinitionen finden Sie das Interface *List*:

```
public class ArrayList<E> extends AbstractList<E>
      implements List<E>, RandomAccess, Cloneable, java.io.Serializable
...
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable
```

Beide Klassen sind Spezialisierungen einer Klasse *AbstractList*. Dies wird in Java durch das Schlüsselwort *extends* zum Ausdruck gebracht. Im Abschnitt 9.6 beschreibe ich Vererbung in Java genauer.

In diesem Abschnitt geht es um *implements*. Eine Klasse kann kein, ein oder viele Interfaces implementieren. Wenn eine Klasse ein Interface implementiert, so heißt das ausgeschrieben: Jede Operation des Interfaces muss durch eine Methode der Klasse implementiert werden. Eine Ausnahme sind *default-Methode*n, die bereits im *Interface* implementiert sind.

Wie eine Klasse definiert auch Interface einen Typ.

Eine Klasse, die eine Methode eines Interfaces implementiert, sollte diese mit der Annotation *@Override* kennzeichnen. Geschieht dies nicht, so ignoriert z. B. Eclipse dieses zwar mit den Standardeinstellungen. Geben Sie allerdings beim Anlegen der Klasse das Interface an, so erzeugt Eclipse diese Annotation. Ich empfehle deshalb, die Optionen so einzustellen, dass der Compiler warnt, falls dies vergessen wurde. Hat das Interface default-Methoden, so verwenden Sie in Eclipse *Source*→Override/Implement Methods, wenn Sie diese überschreiben wollen.

Es ist üblich und nachdrücklich zu empfehlen für die Deklarationen wenn immer möglich ein Interface zu verwenden. Für die Initialisierung braucht man aber immer eine Klasse, es sei denn es handelt sich um ein *funktionales Interface*. Siehe hierzu auch den Abschn. 9.9 über anonyme Klassen und Kapitel 16.

Alle Container-Klassen in *java.util* verwenden das Konzept des *Iterator*s. Ein *Iterator* heißt auch *Cursor*. Ein *Iterator*-Objekt *zeigt* auf jeweils ein Element des Containers. Zunächst zeigt es auf das erste Element. Gibt es weitere, so kann der Iterator mittels der Methode *next* weiterbewegt werden usw. Da *Iterator* ein Interface ist können Sie noch keine Iteratoren direkt implementieren (dabei helfen auch keine *default-Methode*n). Die Container-Klassen aus *java.util* können Sie aber trotzdem verwenden. Sie haben eine Methode *iterator()*, die Ihnen einen Iterator liefert.

Bereits im Beispiel der Klassen *ArrayList* und *AbstractSequentialList* kommen zwei Interfaces vor, die nicht in das bisher präsentierte Schema zu passen scheinen: Die Interfaces *Cloneable* und *Serializable* haben keine Operation. Dies sind sogenannte *Marker-Interfaces*. *Serializable* ermöglicht es auf einfache Weise Objekte zu speichern (siehe Kap. 14). *Cloneable* gibt an, dass es legal, ist eine attributweise Kopie eines Objekts zu machen. Es ist Konvention, dass eine Klasse, die *Cloneable* implementiert, die *protected* Methode *clone* aus *Object* durch eine *public* Methode überschreibt (siehe Abschn. 9.6). Beide Marker-Interfaces werden später genauer erläutert. *Serializable* in Abschn. 14.8, *Cloneable* in Abschn. 9.7.

Hier nun noch schnell ein Beispiel einer wichtigen default-Methode:

Beispiel 9.4.2 (forEach-Methode)
Das Interface *Iterable* hat die default-Methode *forEach*,

```
default void forEach(Consumer<? super T> action) {
    Objects.requireNonNull(action);
    for (T t : this) {
        action.accept(t);
    }
}
```

die ich hier erläutere.

Consumer<T>: Ein Interface mit nur einer abstrakten Methode: *void accept(T t)*. Diese Methode wird in der folgenden *for each*-Schleife auf alle Elemente des *Iterable* angewandt, z. B. die Einträge in einer Liste. Tatsächlich übergeben Sie die Methode *accept*. Siehe hierzu Kap. 16.

Objects.requireNonNull(action): Eine Methode der *Utility-Klasse Objects*, die insbesondere dazu gedacht ist Parameterprüfungen in Methoden und Konstruktoren vorzunehmen.

◀

9.5 Assoziationen

In Java werden Assoziationen durch Referenzen implementiert. Abbildung 9.1 zeigt eine 1 : 1

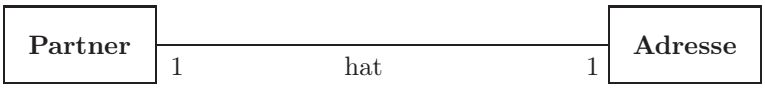


Abb. 9.1: 1:1 Assoziation Partner — Adresse

Assoziation zwischen einer Klasse Partner und einer Klasse Adresse. Das heißt: Zu jedem Objekt der Klasse Partner gibt es genau ein Objekt der Klasse Adresse und zu jedem Objekt der Klasse Adresse gibt es genau einen Partner. Assoziationen enthalten wichtige Informationen:

- Sie bilden Integritätsbedingungen ab. In diesem Fall heißt das, es kann keinen Partner ohne Adresse geben und es kann keine Adresse ohne Partner geben. Abbildung 9.1 sagt dem Programmierer oder der Programmiererin, dass dies durch den Code sichergestellt werden muss.
- Assoziationen sind „Schnellstraßen“. Objektorientierte Systeme bestehen aus Objekten, die zusammenarbeiten, um die Aufgaben des Systems zu erfüllen. Um die mitwirkenden Objekte effizient zu erreichen, braucht man in Java Referenzen.

In diesem Beispiel können Sie leicht in eine catch-22 Situation (Zwickmühle) [Hel61] geraten. Siehe hierzu die Aufgabe 1 in Abschn. 9.13.

Wesentlich häufiger als $1:1$ Assoziationen werden Ihnen $1:*$ Assoziationen, wie in Abb. 9.2 begegnen. Die Richtung von der $*$-Seite hin zur 1-Seite wird in Java wie oben einfach durch eine

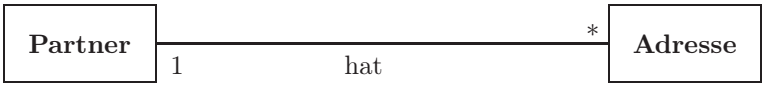


Abb. 9.2: 1:* Assoziation Partner — Adresse

Referenz implementiert. Um die andere Richtung — hier von Partner zu Adresse – zu implementieren, verwenden Sie am einfachsten die Container-Klassen aus dem Paket *java.util*. Das können Sie z. B. wie in *PartnerV02* und *AdresseV02* realisieren. Hier folgt der Code von *PartnerV02*, damit ich Ihnen einige Hinweise geben kann:

```
010 public class PartnerV02 {
020   private String Name;
030   private List<AdresseV02>  adressen;
040   public PartnerV02(String name) {
050      Name = name;
060      this.adressen = new ArrayList<AdresseV02>();
070   }
080   public String getName() {
090      return Name;
100   }
110   public void setName(String name) {
120      Name = name;
130   }
140   public void  addAdresse(AdresseV02 adresse){
```

```
150      if(adresse.getPartner().equals(this)){
160         this.adressen.add(adresse);
170      }else{
180         //Fehlerbehandlung!
190      }
200   }
```

```
210   public Iterator<AdresseV02> getAdressen() {
220      return adressen.iterator();
230   }
240 }
```

Das Attribut *adressen* ist mit dem Interface *List* deklariert. Eine Deklaration mit dem *Interface Collection* ist hier auch denkbar. Im Konstruktor wird es mit einer *ArrayList* initialisiert. Hier müsste noch geprüft werden, dass die Adresse nicht schon in der List enthalten ist. Die sichere Implementierung dieser Richtung der Assoziation ist gewährleistet, wenn ein gloSet verwendet wird. Dieses Interface und seine Implementierungen in *java.util* habe ich aber noch nicht vorgestellt. Im Modell aus Abb. 9.2 muss ein Partner keine Adresse haben, zu einer Adresse muss es aber genau einen Partner geben. Es kann also ein neues Partner-Objekt mit *new* erzeugt werden. Beim Anlegen einer Adresse muss dem Konstruktor ein Partner übergeben werden. Da die Assoziation in beiden Richtungen benutzbar sein soll, muss außerdem die neue Adresse dem Partner hinzugefügt werden. Dies kann mittels *addAdresse* geschehen. Die Sichtbarkeit dieser Methode kann auch auf *protected* beschränkt werden. In der Methode *addAdresse* wird geprüft, ob der Partner in der Adresse auch dieser ist. Informationen zu Fehlerbehandlung finden Sie in Kap. 12.

Um zu einem Partner die Adresse(n) zu bekommen dient die Methode *getAdressen*, die einen *Iterator*, genauer ein Objekt einer Klasse, die das Interface *Iterable* implementiert. So kann die Implementierung der Assoziation in Partner fast beliebig verändert werden. Es muss nur sichergestellt werden, dass *getAdressen* einen Iterator liefern kann. Ein alternativer Rückgabetyp ist einfach *Iterable<AdressV02>*. Über dieses kann dann mit einer for-each-Schelife oder der forEach-Methode iteriert werden.

In Abb. 9.3 sehen Sie eine weitere Variante eines Modells für den Zusammenhang zwischen

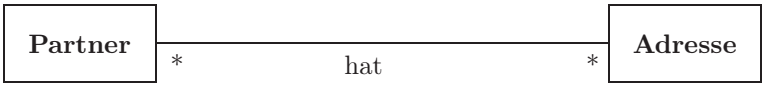


Abb. 9.3: *:* Assoziation Partner — Adresse

Partnern und Adressen. Die Assoziation ist nun $* : *$. Es gibt also zu einem Partner keine, eine oder viele Adressen (s. o.). Nun kann eine Adresse aber zu mehreren Partnern gehören. Denken Sie als Beispiel an ein größeres Mietshaus o. ä. Haben wir dort 10 Partner wohnen, so brauchen wir nur ein Adress-Objekt und bei den Partnern Referenzen auf das Objekt.

Um diese Assoziation zu implementieren brauchen wir wie im vorigen Beispiel Attribute vom Typ einer Container-Klasse, wie etwa List. Nun aber auf beiden Seiten. Die notwendigen Integritätsprüfungen hängen auch davon ab, welche weiteren Bedingungen an die Assoziation gestellt werden. Aber soviel ist sicher: Kein Objekt (Partner, Adresse) kann auf der jeweils anderen Seite der Assoziation mehrfach vorkommen.

9.6 Vererbung

Generalisierungsbeziehungen werden in Java durch das Schlüsselwort *extends* gekennzeichnet. Hier einige Codeschnipsel, mit denen die Klassen eines der Minimodelle aus Beispiel 1.3 spezifiziert werden:

```
public class Person {
   ...
}
public class Benutzer extends Person {
   ...
```

```
}
public class Bibliothekar extends Person {
   ...
}
```

Eine Klasse kann viele Interfaces implementieren, aber nur eine direkte Oberklasse oder Superklasse haben.

Definition 9.6.1 (Abstrakte Klasse)
Eine abstrakte Klasse ist eine Klasse, die keine direkten Objekte hat. Objekte vom Typ einer abstrakten Klasse sind immer Objekte einer Unterklasse. ◄

Um Objekte einer *abstrakten Klasse* zu erzeugen benötigen Sie eine Unterklasse. Haben Sie in Ihrem System konkrete (also nicht-abstrakte) Klassen nur auf der untersten Ebene der Vererbungshierarchie, so ist dies ein Indiz für einen guten Entwurf. Es gibt allerdings auch viele Systeme, in denen dies nicht möglich oder nicht sinnvoll ist.

Eine Klasse ist in Java abstrakt, wenn Klassendeklaration das Schlüsselwort *abstract* steht.

Definition 9.6.2 (Abstrakte Methode)
Eine abstrakte Methode ist eine Methode, die keine Implementierung hat. Sie wird mit dem Schlüsselwort *abstract* gekennzeichent und dur die Deklaration, keinen Rumpf. ◄

Deklarieren Sie eine Methode als *abstract*, so muss auch die Klasse als *abstract* deklariert werden. Vergessen Sie dies, so bekommen Sie einen Compiler-Fehler.

Im Unterschied zu Interfaces können Sie in abstrakten Klassen bereits Methoden und auch Konstruktoren implementieren. Es müssen also nicht alle Methoden abstrakt sein. Es können sogar alle Methoden implementiert sein.

Interfaces spielen in Java eine Rolle, die der von abstrakten Klassen in C++ ähnelt. Abstrakte C++-Klassen sind in vielen Fällen *mixin Klassen*. Solche bringen jeweils einen weiteren Aspekt in eine Klasse hinein, der mit der sonstigen Vererbungshierarchie nichts zu tun hat. So ist es auch bei den Java-Interfaces *Cloneable* oder *Serializable*. Im Unterschied zu Java unterstützt C++ Mehrfachvererbung. Trotzdem finden Sie in den meisten Fällen, in denen dies genutzt wird, nur eine konkrete Oberklasse. Verwenden Sie in Interfaces *default-Methode*n, so haben Sie in Java eine C++ ähnliche Situation.

Bereits in Abschn. 4.8 haben Sie einige Methoden der Klasse *Object* kennengelernt. Diese Klasse hat auch eine Methode *clone*, die *protected* ist. Es ist eine feststehende Konvention, dass Klassen, die das Interface *Cloneable* implementieren, die Methode *clone* aus *Object* durch eine *öffentliche* Methode überschreiben.

Die Methode *clone* aus Object erstellt ein neues Objekt der jeweiligen Klasse und initialisiert dessen Attribute mit genau den Objekten bzw. Werten des Ausgangsobjekt. Den notwendigen Speicherplatz beschaffen Sie sich in Ihrer Methode *clone* durch Aufruf von *super.clone*. Die Attribute werden also nicht „gecloned“, es wird also eine flache Kopie (shallow copy) erstellt. Wollen Sie eine tiefe Kopie erstellen, so müssen Sie mindestens alle *mutable* Attribute ebenfalls clonen.

In vielen Fällen ist es besser einen *Copy Constructor* zu schreiben, als *Cloneable* zu implementieren ([Blo08], Item 11).

Definition 9.6.3 (Copy Constructor)
Ein Copy Constructor ist ein Konstruktor, der als Parameter eine Objekt dieser Klasse bekommt. ◄

9.7 Mehr über Konstruktoren

Schreiben Sie keinen Konstruktor, so hat die Klasse trotzdem einen Konstruktor ohne Parameter, den *default Konstruktor*. Die Attribute können durch diesen nur mit den default Werten initialisiert werden, als je nach Typ 0, 0.0, *false* oder *null*.

Schreiben Sie einen Konstruktor mit einem oder mehreren Parametern,

```
public class NoDefaultConstructor {
   String name;
   public NoDefaultConstructor(String name) {
      this.name = name;
   }
...
}
```

so gibt es *keinen default Konstruktor*! Ein Aufruf

```
NoDefaultConstructor myClazz = new NoDefaultConstructor();
```

liefert dann einen Compilerfehler.

Viele Klassen haben mindestens einen öffentlichen Konstruktor. Oft hat man überladene Konstruktoren. Das Beispiel *CounterV13v01* aus dem Paket *counter* zeigt ein Beispiel. Ein Konstruktor „macht die Arbeit". Das ist hier *CounterV13v01(int start)*. Die anderen rufen diesen mittels *this(...)* mit geeigneten Parametern auf. Etwa so:

```
   public CounterV13v01(CounterV13v01 counter) {
      this(counter.show());
   }
```

Das Schlüsselwort für den Aufruf eines anderen Konstruktors der Klasse ist *this*. Einen Konstruktor der Oberklasse rufen Sie mit *super* und den entsprechenden Parametern auf. Wie überall in der Programmierung gilt es auch hier Redundanz zu vermeiden.

Ein Aufruf von *this(...)* oder *super(...)* muss das erste Statement in einem Konstruktor sein.

Möchten Sie als Autor einer Klasse die Erzeugung von Objekten steuern, so können Sie den Konstruktor (oder die Konstruktoren) *private* deklarieren. Um Objekte der Klasse erzeugen zu lassen, brauchen Sie dann eine Klassenmethode. Ein Beispiel hierfür ist ein Zähler, der als einziger alles zählen soll. Das Beispiel hierzu finden Sie im Paket crash in CounterV14. Hier nur die wesentlichen Codezeilen:

```
public class CounterV14
   ...
   private static CounterV14 counter;
   ...
   public static CounterV14 getCounter(){
      return counter==null?new CounterV14():counter;
   }
   ...
```

Weitere Informationen zu diesem Thema (Singleton Pattern) und die zur Zeit empfohlene Implementierung finden Sie in Abschn. 23.3.

Ein weiteres Beispiel für einen privaten Konstruktor liefern Utility-Klassen. Diese haben nach Def. 1.3.29 nur Klassenmethoden. Deshalb wird nie ein Objekt einer solchen Klasse benötigt. Wollen Sie verhindern, das irrtümlich doch ein Objekt einer solchen Klasse erzeugt wird, so schreiben Sie einfach einen privaten Konstruktor ohne Parameter mit leerem Block. Die Utility-Klassen in *java.util* verwenden diese Technik, siehe z. B. *Collections*. Das obige Beispiel

```
   public CounterV13v01(CounterV13v01 counter)
```

ist auch ein Beispiel für einen „Copy-Konstruktor". Ein solcher Konstruktor erhält ein Objekt der Klasse und erzeugt eine Kopie. In C++ ist ein Copy-Konstruktor Bestandteil der orthodoxen kanonischen Form, die für viele nicht-triviale Klassen dringend empfohlen wird.

In Java ist ein Copy-Konstruktor oft sinnvoll, wenn die Klasse mutable ist. Eine immutable Klasse braucht nicht unbedingt einen Copy-Konstruktor, er kann aber trotzdem sinnvoll sein, siehe z. B. *String*. Nach Abschnitt [Blo08] ist ein *Copy Constructor* eine gute, wenn nicht die bessere, Alternative zur Implementierung von *Cloneable*.

Das liegt an der Definition des Interfaces *Cloneable*: Dieses Marker-Interface enthält keine Methode. Wenn eine Klasse es implementiert signalisiert sie, dass es legitim ist in ihr die Methode *clone* aus *Object* zu überschreiben. Implementiert eine Klasse *Cloneable* nicht, so wird die *clone*-Methode aus *Object* eine *CloneNotSupportedException*, andernfalls liefert *clone* ein bitweise, flache Kopie. *Cloneable* modifiziert also das Verhalten einer Oberklasse. Dass erinnert an ein Problem in C++. Auch hier können Elternklassen Probleme von ihren „Kindern“ erben. Es ist (nur) eine Konvention, das dabei die Sichtbarkeit auf *public* erweitert wird.

Ferner sollten Sie in einer nicht-*final* Klasse, die *Cloneable* implementiert, ein Objekt zurückliefern, dass mittels *super.clone* erzeugt wird. Dies wird aber nicht erzwungen. Die korrekte Klasse des Objekts ist damit nicht sicher gewährleistet. Sind Attribute *final* so kann es sein, dass Sie *clone* gar nicht korrekt implementieren können. In [Blo08], Item 11, werden weitere Probleme mit *clone* plastisch beschrieben. Wenn Sie also *Cloneable* nicht überschreiben müssen, schreiben Sie besser einen *Copy-Constructor*.

9.8 Innere und lokale Klassen

In vielen Beispielen haben Sie genau eine Klasse in einer .java-Datei. Es ist aber so, dass es in einer .java-Datei nur eine öffentliche Klasse geben kann. Genauer darf es nur einen Block einer *öffentlichen* Klasse auf der obersten Ebene geben.

In einer Klassendatei können außer der Klasse mit dem Namen der Datei weitere Klassen definiert werden. Hier ein triviales Beispiel:

```
public class Multiple {
}
class Second {
}
```

Die Klasse *Second* hat hier *package*-Sichtbarkeit. *public* ist nicht zulässig. Die Modifier *abstract* oder *final* wären auch zulässig. Die .class Datei heißt in jedem Fall *Second.class*.

Sie können aber auch innerhalb einer Klassendefinition weitere Klassen definieren. Dies sind dann *geschachtelte Klassen*. Die Klasse, in der eine *innere Klasse* definiert wird, heißt dann *äußere Klasse*. Geschachtelte Klassen können an vielen Stellen in einer Klassendatei (Klasse oder Interface) vorkommen, z. B. auf der obersten Ebene, also der gleichen, wie Attribute, Konstruktoren und Methoden (member class). Sie können auch innerhalb einer Methode, auch innerhalb eines Methodenaufrufs vorkommen.

In manchen Fällen brauchen Sie Objekte einer Klasse im Kontext einer umfassenden äußeren Klasse oft. So ist es in vielen Containerklassen. Eine solche Klasse wird man dann:

- Als innere Klasse deklarieren. Dadurch ist der Namen im umfassenderen Konstrukt (Paket) nicht „verbraucht“.
- Als *private* deklarieren, wenn sie von außen nicht sichtbar sein soll.
- Als *static* deklarieren, wenn dies möglich ist.

Geschachtelte Klassen, die nicht als *static* deklariert sind, heißen innere Klassen. Eine geschachtelte Klasse auf der obersten Ebene (member class) in einem Interface ist automatisch *static*.

Innere Klassen können auf solche Elemente der umschließenden Klasse zugreifen die (effektiv) *final* sind, vor der Klassendefinition deklariert und initialisiert sind. Ein Objekt einer nichtstatischen inneren Klasse hat also intern eine Referenz auf das umschließende Objekt der äußeren Klasse.

Das ist für *static* Klassen anders: Das Keyword bedeutet hier, dass die Klasse keine Referenz auf das Objekt der umschließenden äußeren Klasse hat, so wie statische Methoden keine Referenz auf ein an Objekt der Klasse haben.

es sei denn, sie sind als *static* deklariert.

Statische innere Klassen haben keine Referenz auf ein Objekt der äußeren Klasse und können nicht auf Elemente der umschließenden Klasse zugreifen. Statische innere Klassen sind als kleine Hilfsklassen sinnvoll, die nur im Kontext der umschließenden Klasse eine Bedeutung haben.

Deklarieren Sie eine innere Klasse als *static*, so können Objekte der inneren Klasse *nicht* direkt auf Elemente (Instanz-Attribute oder -Methoden) von Objekten der äußeren Klasse zugreifen. Sie können dies nur tun, wenn Sie eine Referenz auf ein Objekt haben. Das ist also analog zu statischen Methoden.

Ein typisches Beispiel finden Sie in der Klasse *java.util.LinkedList*:

```
private static class Entry<E> {
   E element;
   Entry<E> next;
   Entry<E> previous;

   Entry(E element, Entry<E> next, Entry<E> previous) {
      this.element = element;
      this.next = next;
      this.previous = previous;
   }
}
```

Mehr dazu finden Sie im Kap. 18. Andere Beispiele dienen einfach dazu, die Anzahl .java Dateien zu reduzieren.

Für innere Klassen gelten weitgehend die Regeln für Klassen, wie Sie sie bisher kennengelernt haben. So können Sie auch innere Klassen spezialisieren, sie können Interfaces implementieren usw. Innerhalb der umschließenden Klasse werden sie mit ihrem Namen angesprochen. Lässt ihre Sichtbarkeit dies zu, so können sie von anderen Klassen mit dem vollqualifizierten Namen angesprochen werden. Ist eine innere Klasse *private*, so kann sie nur in in der umschließenden Klasse verwendet werden, in der sie definiert ist. Ist eine innere Klasse protected, so kann sie von anderen Klassen des Pakets und von Unterklassen der umschließenden Klasse verwendet werden.

Auch aus inneren Klassen Klassen erzeugt der Compiler .class-Dateien. Deren Name beginnt mit dem Namen der äußeren Klasse, dann folgt ein $-Zeichen und anschließend der Name der inneren Klasse, ggf. noch mit einer Nummer davor. So werden dann innere Klassen mit gleichen Namen unterschieden, die in unterschiedlichen Blöcken definiert werden.

Innere Klassen können auf der gleichen Ebene wie Attribute und Methoden definiert werden. Dann handelt es sich um Elemente der Klasse und sie können z. B. *public*, *private* oder *static* deklariert werden.

Sie können aber auch innerhalb einer Methode oder bei einem Methodenaufruf in der Parameterliste definiert werden. In diesem Fall sind sie *lokale Klassen*. Diese können keine Modifier haben, wie Sichtbarkeit oder *static*.

In Abschn. 9.9 werden Sie ein Beispiel einer Klassendefinition in einem Methodenaufruf kennenlernen.

9.9 Anonyme Klassen

Anonyme Klassen sind lokale Klassen ohne Namen. Haben Sie eine Schnittstelle, die Sie verwenden wollen, so brauchen Sie, nach allem was Sie bisher wissen eine Klasse, die die Schnittstelle implementiert. Brauchen Sie diese Klasse immer wieder, so ist es auch sinnvoll und richtig, eine solche zu schreiben. Es gibt aber auch Situationen, in denen Sie keine Klasse brauchen, sondern „nur“ ein Objekt vom Typ der Schnittstelle. Für diesen Fall bietet Java das Konstrukt der anonymen Klasse. Die Syntax ist einfach:

```
InterfaceName iF = new InterfaceName() {...};
```

Zwischen dem gescheiften Klammerpaar {...} steht alles, was zum Erzeugen eines Objekts notwendig ist.

Als erstes Beispiel für eine anonyme Klasse verwende ich das Interface *Comparator*. Ich betrachte eine Klasse *Wagon*, die eine Methode *int getTara()* hat, die das Leergewicht liefert. Um eine Liste von Wagons zu sortieren, kann ich dann einfach eine Methode von Collections aufrufen:

```
10   Collections.sort(wagonListe, new Comparator<Wagon>() {
20      @Override
30      public int compare(Wagon w1, Wagon w2) {
40         return Integer.compare(w1.getTara(), w2.getTara());
50      }
60   });
```

Wie das mit einem λ-Audruck gemacht wird sehen Sie in Abschn. 9.10. Ein weiteres typisches Beispiel sind Listener in Swing oder javaFX Sie haben etwa einen *JButton* button und wollen auf das Drücken des Buttons reagieren. Dann brauchen Sie einen *ActionListener*

```
   button.addActionListener(new ActionListener() {

               @Override
               public void actionPerformed(ActionEvent e) {
                  numberOfClicks++;
                  clickDisplay.setText("Anzahl Clicks: " + numberOfClicks);
               }
});
```

Auch Parameter einer Methode können als *final* deklariert werden. Dies bringt aber wenig Zusatznutzen: Parameter von primitivem Typ ohne *final* können auch nicht verändert werden und Parameter von Referenztypen erlauben auch mit *final* den Zugriff auf das Objekt, auf das die Referenz zeigt. Der einzige Nutzen, den die Deklaration als *final* hier bringt, besteht darin, dass irrtümliche Zuweisungen zum Parameter (statt etwa eines Attributs) vermieden werden. In einer Situation ist die Deklaration von Parametern als *final* aber notwendig: Deklarieren Sie in einer Methode eine anonyme innere Klasse, so müssen die Parameter *final* sein, wenn Sie in der inneren Klasse darauf zugreifen wollen. Hier ein einfaches, nur für diese Diskussion konstruiertes Beispiel:

```
public class FinalParameterExample02 {
   public StringMultiple testFinal(final String s, final int times) {
      return new StringMultiple() {
         @Override
         public String getMultiple() {
            StringBuilder temp = new StringBuilder();
            for (int i = 0; i < times; i++) {
               temp.append(s);
            }
            return temp.toString();
         }
      };
   }

   interface StringMultiple {
      String getMultiple();
   }
...
```

Diese anonyme innere Klasse hat Zugriff auf die Parameter der Methode *testFinal* und etwaige Attribute der Klasse *FinalParameterExample02*. Dazu bekommt ein Objekt dieser Klasse eine Kopie der entsprechenden Elemente. Diese entsprechen also den Werten der Elemente zum Zeitpunkt

der Erzeugung des Objekts der anonymen Klasse. Die innere anonyme Klasse kann also nur auf die richtigen Werte vertrauen, wenn diese nicht verändert werden können. Deshalb sind hier die *final*-Deklarationen notwendig. Dies war bis Java 7 so. Ab Java 8 wird dies nicht mehr benötigt. Das Schlüsselwort *final* bei Parametern wird nicht zur Signatur gerechnet. Daher wird es auch nicht beim Überschreiben berücksichtigt. Dies erfolgt also unabhängig davon, ob Sie es verwenden müssen oder nicht. Ferner gilt für anonyme Klassen Folgendes (vgl. [GJS+14], Abschn. 15.9.5):

- Eine anonyme Klasse ist nicht abstrakt.
- Eine anonyme Klasse ist eine innere Klasse.
- Eine anonyme Klasse ist nicht *static*.
- Eine anonyme Klasse hat keinen explizit deklarierten Konstruktor.
- Die Parameter des (default)-Konstruktors ergeben sich aus den Deklarationen des Kontexts, in der Reihenfolge der Deklaration.
- Die Signatur eines anonymen (default)-Konstruktors kann private oder geschützte Deklarationen referenzieren.

Auch lokale Variablen können als *final* deklariert werden, mit den gleichen Wikrungen wie bei Attributen. Ihnen kann nur einmal ein Wert zugewiesen werden:

- Werden sie bei der Deklaration initialisiert, so kann der Wert später nicht verändert werden.
- Werden sie ohne Initialisierung deklariert, so können ihnen später höchstens einmal ein Wert zugewiesen werden.

9.10 Anonyme Methoden

Eine Alternative zum Einsatz von anonymen Klassen in dieser Situation ist die Verwendung eines Lambda-Ausdrucks. Diese kann man auch als anonyme Methode bezeichnen. Dann wird statt einer anonymen Klasse eine anonyme Methode verwendet. In den beiden obigen Beispielen geht das etwa so:

```
Collections.sort(wagonListe, (Wagon w1, Wagon w2)-> {
      return Integer.compare(w1.getTara(), w2.getTara());
});
```

bzw.

```
button.addActionListener((ActionEvent e)-> {
               numberOfClicks++;
               clickDisplay.setText("Anzahl Clicks: " + numberOfClicks);
            });
```

Einzelheiten werden Sie in Kapitel 16 kennenlernen.

Auch Methoden können anonym sein. Ein Beispiel dafür finden Sie in Bsp. 9.4.2 mit der Methode *forEach*. Hier ein ganz einfaches Beispíel für eine Liste irgendwelcher Objekte, etwa Strings. Folgender Code gibt einfach die Strings der Liste aus:

```
List<String> lst = new ArrayList();
...//Füllen mit Inhalt
lst.forEach((s)->System.out.println(s));
```

`(s)->System.out.println(s)` hat keinen Namen, ist also anonym, und implementiert die Methode *accept*, die im Interface *Consumer* deklariert ist. Der Parameter s in den runden Klammern wird auf System.out.println(s) abgebildet. Sie können das aber auch mit einer benannten Methode machen, da es die Methode *println*: bereits gibt:

```
lst.forEach(System.out::println);
```

Hier handelt es sich um eine sog. Methodenreferenz.

Eine systematische Darstellung finden Sie in Kap. 16.

9.11 Strings

Die Klasse *String* verwenden Sie sehr häufig.

Eine wichtige Eigenschaft der Klasse *String* ist, dass Sie *immutable* ist. Ein String kapselt eine Zeichenfolge von *char*. Die Methode *length()* liefert die Anzahl Zeichen. Aber denken Sie daran, dass immer ab 0 gezählt wird: So liefert die Methode *charAt(int i)* liefert für $i = 0$ das erste Zeichen des Strings, für $i = length() - 1$ das letzte Zeichen.

Dies ist die einzige nicht-numerische Klasse in Java, mit der der Operator „+“ verwendet werden kann. Er fügt hier zwei Strings zu einem neuen String zusammen (sie werden „konkateniert“). Das ist wörtlich gemeint: Die Klasse *String* ist immutable. Es wird bei jeder Konkatenierung ein neues *String Objekt* erzeugt. Das kann teuer werden! In den Anfangszeiten von Java erkannten sich Java Gurus daran, dass Sie wussten, dass man Strings nicht so konkateniert. Heute sollte das jeder Programmierer wissen.

Wollen Sie String effizient zusammenfügen, so verwenden Sie die Klasse *StringBuilder* oder *StringBuffer*. Das Schema ist einfach: Sie definieren etwa eine lokale Variable vom Typ *StringBuilder*. Anschließend fügen Sie die Strings in der richtigen Reihenfolge mittels der Methode *append* an. Als letzten Schritt machen Sie mittels der Methode *toString* aus dem *StringBuilder* einen String. *StringBuilder* verwendet keine Synchronisierung und ist damit für parallelen Zugriff nicht geeignet. Nur wenn dieser vorkommen kann, müssen Sie auf *StringBuffer* ausweichen. In vielen Fällen kommt paralleler Zugriff in dieser Situation nicht vor. Dann ist immer *StringBuilder* die Klasse der Wahl.

Bemerkung 9.11.1 (String-Concatenation)
Seit Java 8 ist es Compiler-Herstellern freigestellt bei der Concatenation von Strings mittel „+“ einen *StringBuffer* oder ähnliche Techniken zu verwenden, um keine zusätzlichen String-Objekte zu eerzeugen. Die hier beschriebenen Techniken könnten also in naher Zukunft obsolet sein. ◀

Beispiel 9.11.2 (StringBuilder)
Die Methode *toString* der Klasse *Modifier* liefert ein Beispiel für das korrekte Konkatenieren von Strings:

```
public static String toString(int mod) {
    StringBuilder sb = new StringBuilder();
    int len;

    if ((mod & PUBLIC) != 0)        sb.append("public ");
    if ((mod & PROTECTED) != 0)     sb.append("protected ");
    if ((mod & PRIVATE) != 0)       sb.append("private ");

    /* Canonical order */
    if ((mod & ABSTRACT) != 0)      sb.append("abstract ");
    if ((mod & STATIC) != 0)        sb.append("static ");
    if ((mod & FINAL) != 0)         sb.append("final ");
    if ((mod & TRANSIENT) != 0)     sb.append("transient ");
    if ((mod & VOLATILE) != 0)      sb.append("volatile ");
    if ((mod & SYNCHRONIZED) != 0)  sb.append("synchronized ");
    if ((mod & NATIVE) != 0)        sb.append("native ");
    if ((mod & STRICT) != 0)        sb.append("strictfp ");
    if ((mod & INTERFACE) != 0)     sb.append("interface ");
```

```
        if ((len = sb.length()) > 0)    /* trim trailing space */
            return sb.toString().substring(0, len-1);
        return "";
    }
```

Selbst, wenn Sie nur drei Strings zu konkatenieren zu konkatenieren haben, sollten Sie so verfahren!
◀

Die Klasse *StringJoiner* aus dem Paket *java.util* dient zum „Zusammenbau“ von Strings, die durch ein ausgewähltes Zeichen (Delimiter) getrennt sind. Wahlweise kann ein Präfix oder ein Suffix angegeben werden. Im Konstruktor werden Trennzeichen, Prä- und Suffix angegeben:

```
StringJoiner sj = new StringJoiner(",","(", ")");
```

Die Strings werden also durch Kommata getrennt und die ganze Folge in Klammern eingeschlossen, wie bei der mathematischen Schreibweise einer Folge oder eines Vektors.

Beispiel 9.11.3 (StringJoiner)
Der folgende Code fügt einfach die Buchstaben a–z zu einem String zusammen:

```
public class StringJoinerExample01 {
    public static void main(String[] args) {
        StringJoiner sj = new StringJoiner("");
        for (char c = 'a'; c <= 'z'; c++) {
            sj.add(Character.valueOf(c).toString());
        }
        System.out.println(sj.toString());
    }
}
```

◀

String Literale werden in der JVM „interned“: Sie verweisen alle auf das selbe *String-Objekt*. Sie können also auch mit == verglichen werden. Das gilt aber nur für *String*s, die Werte von konstanten Ausdrücken sind, siehe [GJS+14], §15.28. Durch „+“ konkatenierte *String* werden zur Laufzeit neu erstellt, sind also unterschiedlich. Die Auswirkungen zeigt ein Beispiel von Lars Harmsen aus dem SS 2013, der über dieses Verhalten gestolpert war:

```
    public static void main(String [] args){
        Scanner in = new Scanner(System.in);
        String foo = in.next();
        if(foo == "bla"){
            System.out.println("foo ist \"bla\"");
        }else{
            System.out.println("es wurde nicht \"bla\" eingegeben");

        }
        in.close();
    }
```

Hier ist der *String foo* kein konstanter Ausdruck, sondern eine Variable, die durch eine Eingabe — hier über die Konsole — initialisiert wird. Von daher sind der *String foo* und das Literal `bla` gleich gemäß *equals*, aber nicht referenzgleich (class.StringEx01–StringEx02). Durch Aufruf der Methode *intern* können Sie Strings „internalisieren“, siehe Abschn. 23.9.

Intern werden Strings in der Klasse *String* als *byte*-Array verwaltet. In einer weiteren *byte*-Variablen

9.12 Historische Anmerkungen

Es gibt viele syntaktische Ähnlichkeiten zwischen Java und C++. Um so wichtiger ist es, auf die Unterschiede zu achten. C++ unterstützt Mehrfachvererbung, Java nicht. Java besitzt das Konzept des Interfaces, C++ nicht. Insofern unterscheiden sich die Programmierstile. In vielen Fällen verwendet ein C++ Programmierer eine abstrakte Klasse, wenn ein Java-Programmierer ein Interface und ggf. eine anonyme Klasse verwendet. Mit Java 8 kamen *default-Methode*n, die die Unterschiede zu C++ weiter reduzieren. Trotzdem: Selbst wenn die Syntax identisch aussieht, müssen Sie mit unterschiedlichen Ergebnissen rechnen.

Die Klasse *StringJoiner* wurde in Java 8 neu hinzugefügt. Ebenso die überladene Methode *String::join*

Die am Ende von Abschn. 9.11 beschriebene interne Speicherung von Strings wurde in Java 9 (freigegeben am 21.09.2017) eingeführt. In vorhergehenden Releases war es ein *char*-Array.

9.13 Aufgaben

1. In der Abb. 9.1 ist eine 1:1 Assoziation dargestellt, die auf den ersten Blick trivial aussehen mag, aber gar nicht so einfach zu implementieren ist. Hierzu finden Sie im Paket *crm* die Klassen *PartnerV01. AdresseV01* und *PartnerAdresseV01.* Hierzu nun diese Denksportaufgabe:

   Wie können Sie dafür sorgen, dass Objekte dieser Klassen erzeugt werden können? Es kann sein, dass Sie dabei Kompromisse machen müssen. Mit den bisher präsentierten Themen und Regeln ist dies aber lösbar. Ich nenne hier (sozusagen als Hinweise):

   - Klassen
   - Methoden
   - Attribute
   - Möglichst restriktive Sichtbarkeit
   - Java Code-Konventionen (siehe auch Anhang A des Skripts)

   Demonstrieren Sie an Hand aussagefähiger Beispiele (*JUnit*), dass Ihre Implementierung funktioniert.

2. Was bedeutet das Schlüsselwort „static“ bei einer inneren Klasse?

3. Implementieren Sie bitte die Assoziation aus Abb. 9.2! Demonstrieren Sie bitte an Hand geeigneter Testfälle, dass Ihre Implementierung funktioniert!

4. Implementieren Sie bitte die in Abb. 9.3 dargestellte *:* Assoziation! Demonstrieren Sie bitte an Hand geeigneter Testfälle, dass Ihre Implementierung funktioniert!

5. Schreiben Sie bitte eine Klasse ISBN, die eine ISBN (International Standard Book Number) kapselt! Demonstrieren Sie bitte mit geeigneten Testfällen, dass Ihre Implementierung funktioniert, insbesondere nur ISBN mit gültiger Prüfziffer möglich sind.

6. Bringen Sie bitte die folgenden Begriffe in eine logische Reihenfolge (nicht die alphabetische)! Erläutern Sie Ihr(e) Ordnungskriterien!

   anonyme Klasse, Annotation, Attribut, innere Klasse, Interface, Klasse, Konstruktor, Methode, Modifier, Paket.

7. Erklären Sie bitte das Verhalten der Klasse aus Beispiel 9.3.1! Wie erreichen Sie das gewünschte Ergebnis? Hinweise: [GJS+17], Abschn. 4.12, 12.4.

8. Nennen und erläutern Sie bitte knapp alle Elemente die *Interface*s in Java haben können!

9. In welchen Kontexten kann eine Klasse *private* bzw. *protected* sein? Begründen Sie Ihre Antwort mit logischen Argumenten!

10. Gegeben sei eine 1:*-Assoziation zwischen Kunde und Auftrag wie in folgendem Klassendia-

gramm:

Der folgende Code soll diese implementieren:

```
010 public class Kunde {
020   string name;
030   string adresse;
040
050   Auftrag []auftrag;
060   Kunde(string name,
070         string adresse){
080      auftrag=new auftrag[42];
090   }
100   void addAuftrag (auftrag a)
110   auftrag [anzahlauftraege ++]
120   storniere Auftrag (auftrag a);
}
```

```
010 public class Auftrag<kunde> {
020   Date datum;
030   kunde[] kunde;
040   {
050      assert (kunde != null);
060      this.kunde = this.kunde;
070      Kunde.addAuftrag(this);
080   }
090 }
```

Identifizieren Sie bitte alle Fehler und sonstigen Unzulänglichkeiten in diesem Code. Sie müssen den Code nicht verbessern bzw. korrigieren, sondern nur angeben, was an diesem Code zu Compiler-Fehlern oder -Warnungen führt oder nicht den Codekonventionen entspricht, also die Kritikwürdigkeit der Punkte begründen. Javadoc und andere Kommentarkonventionen sollen Sie *nicht* berücksichtigen. Sie können mehr als 5 Punkte bekommen, aber bei ca. 10 Punkten werde ich aufhören zu zählen.

11. In den folgenden Codeschnipseln werden Lambda-Ausdrücke verwendet.

```
method1(x, y, d -> Math.cos(d));
someList.forEach(entry -> System.out.println(entry));
method2(a, b, c, (d1,d2) -> Math.pow(d1,d2));
someStream.reduce(0, (i1,i2) -> Integer.sum(i1, i2));
method3(foo, bar, (a,b,c) -> Utils.doSomethingWith(a,b,c));
method4(() -> Math.random());
```

11.1. Ersetzen Sie bitte die λ-Ausdrücke durch äquivalente Methodenreferenzen

11.2. Geben Sie bitte einen korrekten Typ für den jeweils letzten Parameter der Methoden *method1–method4* an!

λ-Ausdruck	Methodenreferenz
d -> Math.cos(d)	
(entry -> System.out.println(entry)	
(d1,d2) -> Math.pow(d1,d2)	
(i1,i2) -> Integer.sum(i1, i2)	
(a,b,c) -> Utils.doSomethingWith(a,b,c)	
() -> Math.random()	

Kapitel 10

Modules

10.1 Übersicht

Modularisierung ist ein Bestandteil vieler, wenn nicht sogar aller Programmiersprachen. In Java sind von Beginn an Klassen die erste Ebene. Diese werden zu Paketen gruppiert. Als weitere Aggregationsebene dienen Module, wie in Abb. 10.1 illustriert. Mittels des *Java Platform Module System* wird auch Java selbst modulatisiert. So können kleinere Laufzeitsysteme verwendet werden, wenn die Anwendung nicht den gesamten Java-Umfang benötigt. Das einzige Module, das eine Java-Anwendung unbedingt benötigt, ist das Module *java.base*.

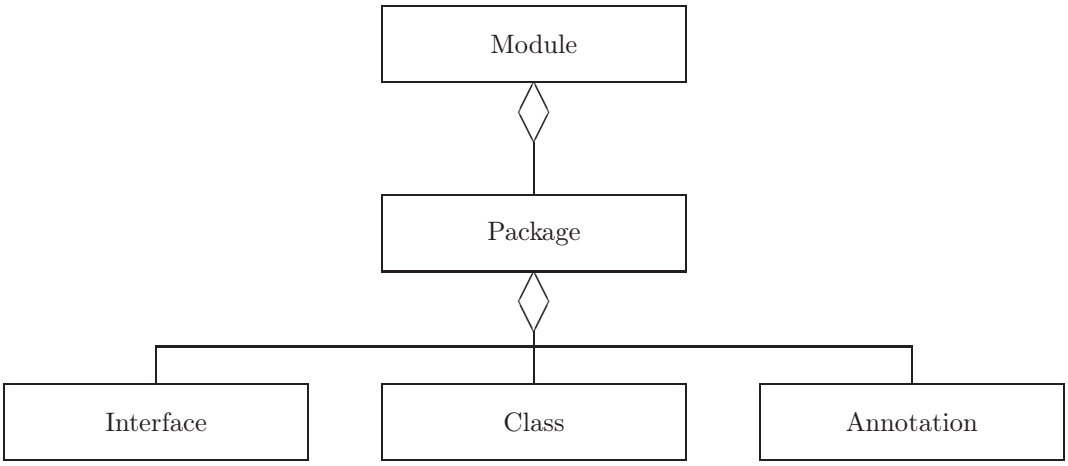


Abb. 10.1: Teil des Java-Metamodells

Ein *Module* ist eine eindeutig benannte — nach dem Schema, wie auch Pakete — wieder verwendbare Gruppe von Paketen (packages), zusammen mit Ressourcen, wie Bildern oder XML-Dateien. Ein *Module* hat einen *Module-Descriptor*, in dem spezifiziert wird:

- Der Name des *Modules*.
- Die Abhängigkeiten des *Modules*, d. h. von welche anderen *Modules* dieses *Module* benötigt (dependencies) (requires).
- Die Pakete, die das *Module* für andere *Module* zur Verfügung stellt. Alle anderen sind implizit nicht für andere *Module* verfügbar (exports).
- Die Services, die es anbietet (provides).
- Die Services, die es benötigt (uses).

- Für welche anderen Module es reflexiven Zugriff (Kap. 19) gestattet (opens).

Die Einführung von Modulen ermöglicht es zusammen mit dem Java Compiler, Linker und dem neuen Versionierungskonzept auch Elemente aus Java zu entfernen. Bisher wurden Elemente als deprecated gekennzeichnet, wenn sie durch bessere Ersetzt wurden. Da der Java Compiler für eine ausgewählte Version compilieren kann, können derartige Elemente in Zukunft auch ganz entfernt werden.

10.2 Lernziele

- Erklären können, wozu weitere Modularisierung dient.
- Module-Deklarationen mit Abhängigkeiten mittels *requires* und Spezifikation der Packages mittels *exports*, die für andere Modules verfügbar sind, erstellen können.
- Reflexiv mit Modulen umgehen können.
- Services verwenden können, um Komponenten lose koppeln zu können.
- Services in Modules verwenden können und Services aus einem Module verfügbar machen können.
- Mittels *jdeps* die Abhängigkeiten eines Modules ermitteln können.
- Nicht-modularisierten Code nach Java 9 migrieren können.
- Mittels jlink kleinere runtimes erstellen können.

10.3 Grundlagen

An ordinary compilation unit consists of three parts, each of which is optional:

- A package declaration (§7.4), giving the fully qualified name (§6.7) of the package to which the compilation unit belongs. A compilation unit that has no package declaration is part of an unnamed package (§7.4.2).
- import declarations (§7.5) that allow types from other packages and static members of types to be referred to using their simple names.
- Top level type declarations (§7.6) of class and interface types.

A modular compilation unit consists of a module declaration (§7.7), optionally preceded by import declarations. The import declarations allow types from packages in this module and other modules, as well as static members of types, to be referred to using their simple names within the module declaration.

Every compilation unit implicitly imports every public type name declared in the predefined package java.lang, as if the declaration import java.lang.*; appeared at the beginning of each compilation unit immediately after any package declaration. As a result, the names of all those types are available as simple names in every compilation unit.

Deklariert wird ein *Module* in einer *module-info.java*-Datei wie folgt:

```
ModuleDeclaration:
    {Annotation} [open] module Identifier {.Identifier}
    {{ModuleDirective} }

ModuleDirective:
    requires {RequiresModifier} ModuleName ;
```

```
 exports PackageName [to ModuleName {, ModuleName}] ;
 opens PackageName [to ModuleName {, ModuleName}] ;
 uses TypeName ;
 provides TypeName with TypeName {, TypeName} ;
 RequiresModifier:
(one of)
transitive static
```

Ein Module ist *offen*, wenn es den *open* Modifier hat. Dies bedeutet: Ein offenes Module gewährt zur Compile-Zeit nur zu den Zugriff auf Typen aus den Paketen des Modules, die explizit exportiert wurden. Zur Laufzeit gewährt es Zugriff auf alle Typen in allen seinen Paketen, so, als wären sie exportiert.

Ein *Module* ohne den *open* Modifier heißt *normal*. Ein *normal*es Modul gewährt sowohl zur Compile- als auch zur Laufzeit nur den Zugriff auf Typen aus explizit exportierten Paketen.

Bemerkung 10.3.1 (Offen-Geschlossen-Prinzip)
Im Software-Engineering gibt es das Offen-Geschlossen-Prinzip für Module. Dieser Begriff hat eine erhebliche Spannbreite. Unabhängig davon besagt dieses Prinzip, dass das Innere eine Moduls gekapselt ist (geschlossen) und ohne

◀

A module is observable if at least one of the following is true:

- A modular compilation unit containing the declaration of the module is observable (§7.3).
- An ordinary compilation unit associated with the module is observable.

The host system determines which compilation units are observable, except for the compilation units in the predefined package java and its subpackages lang and io, which are all always observable. Each observable compilation unit may be associated with a module, as follows:

- The host system may determine that an observable ordinary compilation unit is associated with a module chosen by the host system, except for the ordinary compilation units in the predefined package java and its subpackages lang and io, which are all associated with the java.base module.
- The host system must determine that an observable modular compilation unit is associated with the module declared by the modular compilation unit. The observability of a compilation unit influences the observability of its package (§7.4.3), while the association of an observable compilation unit with a module influences the observability of that module (§7.7.6).

A package is observable if and only if at least one of the following is true:

- An ordinary compilation unit containing a declaration of the package is observable (§7.3).
- A subpackage of the package is observable. The packages java, java.lang, and java.io are always observable. One can conclude this from the rule above and from the rules of observable compilation units, as follows. The predefined package java.lang declares the class Object, so the compilation unit for Object is always observable (§7.3). Hence, the java.lang package is observable, and the java package also. Furthermore, since Object is observable, the array type Object[] implicitly exists. Its superinterface java.io.Serializable (§10.1) also exists, hence the java.io package is observable.

module-info.java Datei für Module-Deklaration.

Development tools for the Java programming language are encouraged to highlight requires transitive directives and unqualified exports directives, as these form the primary API of a module.

Module *java.base* ist implizit *required*, muss also nicht unter der *required* Modules angegeben werden.

required hat die Modifier transitive und static:
export und open und *to*
uses, java.util.ServiceLoader
provides
A module is observable if at least one of the following is true:

- A modular compilation unit containing the declaration of the module is observable (§7.3).
- An ordinary compilation unit associated with the module is observable.

10.4 Der Java Linker

10.5 Der Java Compiler

10.6 Portability

Java SE Standard Module gewähren keine Lesbarkeit für nicht-Standard Module. Deshalb ist Code für alle Java SE-Implementierungen portierbar, wenn er nur von Standard-Modulen abhängig ist. Sie können aber sehr wohl von solchen abhängig sein. Diese Abhängigkeit ist aber nicht transitiv.

10.7 Modules und Reflection (RTTI)

10.8 Historische Anmerkungen

Module wurden in der Version Java 9 eingeführt, die am 21.09.2017 freigegeben wurde. Seitdem gibt es die entsprechende Darstellungsebene auch in Javadoc. Sie können in der API Dokumentation zwischen Modulen und Paketen wählen.

10.9 Aufgaben

1. In welche Module ist Java in der Version 9 gegliedert?

Kapitel 11

Datum und Uhrzeit

Time flies like an arrow. Fruit flies like a banana.
Origin: Unkown. I got it from Townes Van Zandt.

11.1 Übersicht

Java verfügt über eine einfach zu verwendende Sammlung von Schnittstellen und Klassen für den Umgang mit Datum und Uhrzeit[1]. Sie finden diese im Paket *java.time* und seinen Unterpaketen. Alle Klassen und Schnittstellen folgen wenigen, einfachen Grundprinzipien:

- Die Klassen sind immutable, thread-safe und wertbasiert (value based).
- Trennung von Datum und Kalender.
- Klassen für Datum, Uhrzeit und Zeitpunkt.
- Klassen für Zeiträume, wie *Duration* oder *Period*, mit Methoden, um mit Zeiträumen umzugehen.
- Es gibt komfortable Fabrikmethoden. Die Konstruktoren sind privat.
- Im Paket *java.time.chrono* gibt es verschiedene *Chronologien*: *IsoChronology*, *HijrahChronology*... Diese repräsentieren weitere Kalendersysteme.

11.2 Lernziele

- Die Klassen für Datum und Uhrzeit sicher verwenden können.

11.3 Grundlagen

Alle Java-Klassen für den Umgang mit Java hat Klassen, um mit Zeitpunkten (Instant), Datumsangaben (LocalDate, u. a.) mit und ohne Zeitzone, Uhrzeiten (LocalTime, ZonedDateTime), und Dauern (Duration, Period) umzugehen, folgen einigen einfachen Grundprinzipien. Diese Klassen sind immutable, thread-save und wertbasiert (value based).

Sie haben keine öffentlichen Konstruktoren, sondern Objekte werden über Fabrikmethoden erzeugt, z. B. *of* uvam.

[1] Dieser Abschnitts enthält Teile einer Hausarbeit von Max Zender aus dem WS 2014/15.

Objekte der Klasse *Instant* beschreiben einen Punkt auf der Zeitachse, der intern durch die Anzahl der vergangenen Sekunden seit dem 01.01.1970, 00:00 Uhr und die seit der Sekunde vergangenen Nanosekunden repräsentiert wird. Positive Werte liegen danach, negative davor. Es handelt sich um ein „Maschinendatum“ und kann auch als Zeitstempel (timestamp) verwendet werden.

Für die Rechnung mit Instants gibt es einfach zu verwendende Methoden:

```
Instant inst = Instant.now();
Instant instPlus2Days = inst.plus(Duration.ofDays(2));
Instant instPlus3Days = inst.plus(3, ChronoUnit.DAYS);
Instant instPlus5Days = ChronoUnit.DAYS.addTo(inst, 5);
```

Ein Instant ist aber kein Datum sondern eben nur ein Punkt auf der Zeitachse. Ein Instant hat nicht die Einheiten, die Menschen beim Umgang mit Zeiten gewohnt sind. Umrechnungen in andere Einheiten sind aber einfach:

```
LocalDateTime ldt = LocalDateTime.ofInstant(inst, ZoneId.systemDefault());
```

Das oben verwendete Enum ChronoUnit werde ich im Anschluss an die Datumsklassen erläutern.

Auch die Berechnung von Zeiträumen wurde in Java 8 wesentlich erleichtert. Hierzu wurde die Klasse *Duration* eingeführt, die einen Zeitraum repräsentiert und Vergleiche mit anderen Zeiträumen und arithmetische Operationen auf diesen erlaubt.

```
long calcDiffInDays(Instant time1, Instant time2) {
   return Duration.between(time1, time2).toDays();
}
```

Weitere Methoden sind z. B. toHours, toMinutes, toSeconds ...

Bemerkung 11.3.1 (Darstellung von Duration)
Intern wird eine *Duration* durch die Anzahl Sekunden und die Anzahl *Nano*sekunden innerhalb der laufenden Sekunde dargestellt. Durations sind gerichtet. Mit positivem Vorzeichen der Sekunden Richtung Zukunft, mit negativem Vorzeichen der Sekunden in Richtung Vergangenheit. Eine Duration von -1 *Nano*sekunde wird durch -1 Sekunde und 999.999.999 *Nano*sekunden dargestellt. ◀

Während ein Objekt der Klasse *Instant* für einen Zeitpunkt steht, repräsentiert ein Objekt der Klasse *LocalDate* ein Datum ohne Zeitzone und Tageszeitinformationen. Objekte der Klasse *LocalDate* repräsentieren ein für Menschen lesbares Datum, wie z. B. einen Geburtstag.

Eine Instanz von *LocalDate* speichert keine Tageszeitinformationen. Der folgende Code zeigt, dass das API dem von Instant sehr ähnlich ist, nun aber mit Methoden, die für Menschen verständlicher Einheiten liefern bzw. damit arbeiten.

```
LocalDate date = LocalDate.of(2014, 12, 1);
int lengthOfMonth = date.lengthOfMonth(); // => 31
```

Auch *LocalDate* hat keinen öffentlichenKonstruktor, aber viele Fabrikmethoden, u. a. die überladenen Methoden *now* und *of*.

Die Klassen *LocalDate* und *LocalTime* repräsentieren Datum und Uhrzeit ohne Berücksichtigung einer Zeitzone, wie z. B. ein Geburtsdatum. Sie repräsentieren im Unterschied zu *Instant* keinen Zeitpunkt auf der Zeitachse.

Sie sind für alle Zwecke ausreichend, in denen Benutzer nicht über Zeitzonen hinweg kommunizieren müssen. Benötigen Sie eine Zeitzone, so verwenden Sie *ZonedDateTime* etc.

Beispiel 11.3.2 (LocalDate)
Brauchen Sie im Rahmen einer Desktop-Anwendung einen Kalender, so ist *LocalDate* die passende Klasse. Eine typische Einsatzsituation ist die Nutzung bei der Eingabe gültiger Datumsangaben. Soll ein Kalender in verschiedenen Zeitzonen verwendet werden, so ist *ZonedDate* die geeignete Klasse, um *LocalDate*s dem Benutzer in der Oberfläche anzuzeigen. ◀

Für die Umrechnung von *LocalDate*s in *ZonedDate*s gibt es z. B. die Methoden *atZone*, *atStartOfDay*, *ZonedDateTime.of*.

Wie die im vorigen Abschnitt beschriebene *Duration*-Klasse für *Instant*s, gibt es für *LocalDate* eine ähnliche Klasse: Die Klasse *Period*. Sie erlaubt die Berechnung von Zeiträumen zwischen zwei *LocalDate*s, den Vergleich von Zeiträumen und arithmetische Operationen auf diesen.

Der folgende Code zeigt die Berechnung einer Period aus *LocalDate*s und die Angabe in Tagen.

```
long calcDiffInDays(LocalDate date1, LocalDate date2) {
   return Period.between(time1, time2).getDays();
}
```

Bemerkung 11.3.3 (Darstellung von Period)
Intern werden *Period*s durch Jahre, Monate und Tage dargestellt. Einen Unterschied gibt es bei der Zeitumstellung (Sommer-, Winterzeit). Wird am Tag vor der Umstellung zu einem *ZonedDateTime* eine Duration von 24 Stunden addiert, so werden tatsächlich 24 Stunden hinzugefügt. Wird eine Period von einem Tag addiert, so wird die Zeitumstellung berücksichtigt und ggf. damit 23 bzw. 25 Stunden addiert. ◀

Die Klassen *LocalTime* und *LocalDateTime* repräsentieren Tageszeit- und die letztere zusätzlich Datumsinformationen. Die Klasse *ZonedDateTime* berücksichtigt zusätzlich verschiedene Zeitzonen. Letztere speichert zusätzlich den Offset zur Greenwich Mean Time (GMT). In der Verwendung unterscheidet sich diese Klasse kaum von *LocalDateTime*.

Das Interface *TemporalAdjusters* unterstützt die Berechnung von Zeitpunkten relativ zu einem anderen Zeitpunkt. Ein typisches Beispiel ist die Bestimmung des Datums für den ersten Freitag im Monat, oder für den letzten Tag eines Monats. Zum Teil war diese Funktionalität bereits in *Calendar* enthalten, allerdings war eines der Motive für die Entwicklung von *TemporalAdjusters* die Externalisierung dieser Logik, um eigene Implementierungen von „Adjustern“ (nach dem Strategy-Pattern) zuzulassen Der folgende Code ist funktional äquivalent zu dem in im für `lengthOfMonth` (s. o.) beschriebenen Beispiel, illustriert aber die aktuelle Lösung des Problems mit Verwendung von *TemporalAdjusters*.

```
LocalDate.of(2014, 12, 1).with(TemporalAdjusters.lastDayOfMonth());
```

```
// Aktuelles Datum
LocalDate date = LocalDate.now();
// Auf den nächsten Monat setzen
LocalDate nextMonth = date.plusMonths(1);
// Auf ersten Freitag im Monat setzen
nextMonth.with(TemporalAdjusters.firstInMonth(DayOfWeek.FRIDAY));
```

Auch diese Klassen sind *immutable*, *value based* und *thread-safe*. Sie repräsentieren Datum bzw. Uhrzeit im Kontext eines „lokalen Betrachters“, daher die Klassennamen. Die Klassen haben Fabrikmethoden mit klaren Namen, die ihre Bedeutung sofort erkennen lassen: Für *LocalDate*

from(TemporalAccessor temporal) Hier werden *TemporalAccessor* verwenden, wie *DayOfWeek (MONDAY, TUESDAY, ...)*, *Month (JANUARY, FEBRUARY, ...)*

now() Liefert das aktuelle Tagesdatum des jeweiligen Rechners (daher *lokales Datum*)

now(Clock clock) Liefert das Datum der jeweiligen Uhr (Clock).

now(ZoneId zone) Liefert das aktuelle Datum für die angegebene Zeitzone.

of(int year, int month, int dayOfMonth)

of(int year, Month month, int dayOfMonth)

ofEpochDay(long epochDay)

ofYearDay(int year, int dayOfYear)

parse(CharSequence text)

parse(CharSequence text, DateTimeFormatter formatter)

bzw. *LocalTime*

from(TemporalAccessor temporal)

now()

now(Clock clock)

now(ZoneId zone)

of(int hour, int minute)

of(int hour, int minute, int second)

of(int hour, int minute, int second, int nanoOfSecond)

ofNanoOfDay(long nanoOfDay)

ofSecondOfDay(long secondOfDay)

parse(CharSequence text)

parse(CharSequence text, DateTimeFormatter formatter)

Die Klasse *LocalDateTime* kombiniert Datum und Uhrzeit.

Beispiel 11.3.4 (Datum und Uhrzeit)
Hier einige kleine Beispiele zur Verwendung der Fabrikmethoden:

```
LocalDate.parse("24.12.2014", DateTimeFormatter.ofPattern("dd.MM.yyyy"));
LocalDateTime.of(2014,07,24,11,1,01);
LocalTime.parse("9:45:30");
```

◀

Für die Bestandteile von Datum oder Uhrzeit gibt es die entsprechenden getter, wie *getMonth*, *getYear*, *getHour getMinute* u. a.

11.4 Einheiten

Die Zeiteinheiten sind im Enum *ChronoUnit* im Paket *java.time.temporal* definiert. Dieses Enum definiert Dinge wie Stunden (*HOURS*), Tage (*DAYS*) etc.

Um die Länge eines Zeitraumes zu bestimmen gibt es mindestens zwei Möglichkeiten. Das *enum ChronoUnit* hat eine Methode *between*. Diese bekommt zwei Objekte der jeweiligen Klassen *LocalDate*, *LocalTime* oder *LocalDateTime*.

Die Klassen *LocalDate LocalTime* und *LocalDateTime* haben eine Methode *until*. Diese liefert von einem Wert ausgehend die Anzahl Einheiten (*ChronoUnit*) bis zum angegebenen Wert.

Diese Dinge und andere Vorstellungen von Tag etc. müssen Sie auseinanderhalten, wie das folgende Beispiel zeigen wird. So gibt es die Klassen *Duration* und *Period*. Die bereits oben erwähnte Methode *until* ist für *LocalDate* überladen. Ohne den Parameter für die *ChronoUnit* liefert diese Methode eine *Period*.

Beispiel 11.4.1 (Einheiten)
Bereits oben haben Sie die Klassenmethode *between* der Klasse Duration kennengelernt:

```
Duration.between(time1, time2).toDays();
```

Außerdem gibt es eine analoge Methode im Enum *ChronoUnit*.

```
ChronoUnit.DAYS.between(LocalDate.now(), LocalDate.of(2015, 12, 24));
```

Analog erhalte Sie die Anzahl Wochen, Jahre etc. Die Klasse *Period* hat aber auch eine Methode *getDays()*. Diese liefert die Anzahl Tage in dieser Periode von x-Jahren, y-Monaten und so und soviel Tagen. ◀

Die Datums- und Uhrzeitklassen spielen mit Interfaces wie *TemporalUnit*, *TemporalField* und implementierenden Klassen wie *ChronoUnit*, *ChronoField*. Nicht alle diese Klassen unterstützen alle Einheiten. So unterstützt die Methode *int get(TemporalField field* der Klasse *ZoneOffset* nur das *TemporalField OFFSET_SECONDS*, alle anderen *ChronoField*-Objekte führen auf eine *UnsupportedTemporalTypeEception*. Die abstrakte Klasse *Clock* bietet Ihnen über einige Klassenmethoden Zugriffsmöglichkeiten auf die aktuellen Objekte von *Instant*, *LocalDate* und *LocalTime* unter Verwendung einer Zeitzone (*TimeZone*). Die Implementierung der Klasse *Clock* verwendet die Methode *System.currentTimeMillis()*. Diese bietet keine Genauigkeitsgarantie. Wird eine hohe Genauigkeit benötigt, so müssen Sie eine Unterklasse schreiben, die einen NTP-Server verwendet. Es gilt als guter Stil, ein Objekt der Klasse *Clock* an jede Methode zu übergeben, die den aktuellen Zeitpunkt benötigt. Am häufigsten werden Sie vielleicht die Klassenmethoden von *Clock* verwenden. *Clock* greift auf die gleichen Ressourcen zu, wie *System.currentTimeMillis*, berücksichtigt aber keine Schaltsekunden. Implementierende Klassen sollten nach API-Dokumentation *Serializable* sein.

11.5 Vergleiche mit Date und Calendar

Da es viel Code gibt, der die alten Datumsklassen verwendet, zeige ich diesen hier auch. Das tue ich aus zwei Gründen:

1. Sie sollen sehen, dass die neuen Klassen viel einfacher zu verwenden sind.

2. Finden Sie alten Code im Internet oder haben Sie alten Code zu warten, so helfen Ihnen die Beispiele vielleicht bei der „Übersetzung“.

Die Klasse *Instant* besitzt eine ähnliche Funktionalität wie die seit JDK 1.0 existierende Klasse *java.util.Date*. Wollte man mit letzterer allerdings zeitliche Berechnungen durchführen, wie beispielsweise die Addition oder Subtraktion von Tagen, war dies nur mithilfe der Klasse *java.util.Calendar* möglich, wie hier zu sehen ist.

```
Date timestamp = new Date();
Calendar cal = Calendar.getInstance();
cal.setTime(timestamp);
cal.add(Calendar.DAY_OF_MONTH, 5);
Date timestampPlusFiveDays = cal.getTime();
```

Ich weise darauf hin, das der in der ersten Zeile verwendete Default-Konstruktor einer der beiden Konstruktoren von *Date* ist, die (noch) nicht *deprecated* sind.

Sie haben in Abschn. 11.3 gesehen, wieviel einfacher das mit der Klasse *Instant* ist Möchte man beispielsweise die zeitliche Differenz in Tagen zwischen zwei Zeitpunkten feststellen, so lässt sich dies in Java 7 mit folgender Methode lösen:

```
long calcDiffInDays(Calendar time1, Calendar time2) {
   // Zeitunterschied in Millisekunden
   long diffInMs = time2.getTimeInMillis()-time1.getTimeInMillis();
   // Dauer eines Tages in Millisekunden
   long dayDurationInMs = 86400000;
   return diffInMs / dayInMs;
}
```

In Java 8 wird dies durch die o. g. Klasse *Duration* wesentlich erleichtert:

Ein Äquivalent dazu war bis Java 7 nicht vorhanden: Behelfsmäßig wurde ein *GregorianCalendar* mit der Uhrzeit „00:00“ verwendet, wie im folgenden Codeausschnitt zu sehen ist.

```
// Uhrzeit wird standardmäßig gesetzt auf 00:00 Uhr
GregorianCalendar cal = new GregorianCalendar(2014, 12, 1);
int lengthOfMonth = cal.getMaximum(Calendar.DAY_OF_MONTH); // => 31
```

Die Berechnung der Tage, die zwischen zwei Daten liegen ist für Java 7 in diesem Fall äquivalent zu der Berechnung in obigem Code für Java 7.

Bis Java 7 sah die Bestimmung des Datums für den letzten Tag im aktuellen Monat aus, wie im folgende Codeausschnitt beschrieben.

Erheblich vereinfacht durch *TemporalAdjusters* wird auch die Bestimmung der Daten von in der Vergangenheit oder Zukunft liegenden Wochentagen. Hier ein Beispiel für die Verwendung von *TemporalAdjusters* zur Bestimmung des ersten Freitags im nächsten Monat mit Java 7.

```
// Aktuelles Datum
Calendar cal = Calendar.getInstance();
// Auf den nächsten Monat setzen
calendar.add(Calendar.MONTH, 1);
// Wochentag auf Freitag setzen
calendar.set(Calendar.DAY_OF_WEEK, Calendar.FRIDAY);
// Setze Anzahl der Vorkommen des gesetzten Tages auf 1
calendar.set(Calendar.DAY_OF_WEEK_IN_MONTH, 1);
```

Diese Lösung erscheint umständlich und unnötig verbos. Eine wesentlich elegantere Lösung mithilfe von *TemporalAdjusters* ist in Listing 11.3 beschrieben.

11.6 Historische Anmerkungen

Das in Java 8 eingeführte API für Datum und Uhrzeit (datetime API) ist der dritte Anlauf zu einem solchen API. Der ursprüngliche Ansatz mit der Klasse *Date* hatte einige Schwächen:

- *Date* und *SimpleDateFormatter* sind nicht Thread-safe.
- Jahre beginnen in 1900, Monate mit 1, Tage mit 0.
- Die *toString*-Methode von *Date* gibt eine Zeitzone (timezone) an.

In Java 1.1 kam die Klasse *Calendar* im Paket *java.util* hinzu. Vor Java 8 standen für den Umgang mit Zeit und Datum nur die Klassen *java.util.Date* und *java.util.Calendar* zur Verfügung. Diese brachten aber einige Probleme mit sich. So sind Objekte beider Klassen immer *mutable*, was die Verwendung dieser Objekte in Multithread-Anwendungen verhinderte. Die Klasse *Date* liefert in der *toString*-Methode eine Zeitzone. Eine Zeitzone wird in der API-Dokumentation aber nicht erwähnt. Außerdem war das Rechnen mit Daten (Addition/Subtraktion von Tagen/Wochen/Jahren) recht umständlich und nicht intuitiv . Die neuen Date-Time APIs im Seit Java 8 gibt es im Paket *java.time* und seinen Unterpaketen eine einfache Lösung zum Umgang mit Datums- und Zeitangaben: Alle Klassen aus dem *java.time* Package sind nicht veränderbar (*immutable*), damit *thread-safe* und sie sind *wertbasiert*. Viele ihrer Eigenschaften werde ich in diesem Kapitel erläutern.

11.7 Aufgaben

1. Falls Sie imWintersemester (also WS 2016/17) Ihr Studium begonnen haben, so begann Ihr Studium planmäßig am 01.09.2016. Die Regelstudienzeit beträgt für Vollzeitstudierende 6 Semester. Um dieses Daten herum folgen einige Aufgaben, die Sie bitte mittels geeignetem Java-Code beantworten. Geben Sie die Ergebnisse bitte auch nach europäischen Standard formatiert auf der Konsole aus!

   1.1. Wieviele Tage wird Ihr Studium gedauert haben, wenn Sie zum Ende der Regelstudienzeit abgschlossen haben und zum Ende Ihres sechsten Semesters exmatrikuliert werden?

   1.2. Fallen Sie bei einer Prüfung durch, so müssen Sie spätestens nach einem Jahr einen zweiten Versuch unternehmen. Für viele Prüfungen haben Sie insgesamt drei Versuche. Der Einfachheit nehmen wir an, die Prüfung sei jeweils am letzten Semestertag (31.08. bzw. bzw. 29. Februar). Bis wann müssen Sie eine Prüfung aus dem 3. Semester also bestanden haben?

   1.3. Um wieviele Minuten verlängert sich Ihr Studium gegenüber der Regelstudienzeit, wenn Sie in die Situation aus 2. „hineinstolpern“ und die drei Versuche ausschöpfen müssen?

2. Ihr Prof möchte für die am 04.07.2017 stattfindende Klausur am 12.07.2017 um 10:00 Klausureinsicht anbieten.

   2.1. Schreiben Sie bitte *a03* Code, mit dem Werte wie die folgenden ermitteln werden können!

   - Wieviele Minuten hat er dann für die Durchsicht, wenn er am Dienstag, 28.06.2016 um 6:00 beginnt?
   - Am 04.06.2016 waren 54 Studierende angemeldet. Wenn 50 Studierende teilnehmen, wieviele Minuten hat er dann pro Klausur? Wieviele, wenn er 8 Stunden pro Tag Pause macht?
   - Wieviel Zeit hat er unter diesen Annahmen, wenn im gleichen Zeitraum auch noch 30 weitere Klausuren zu bewerten sind?

   2.2. Schreiben Sie bitte aussagefähige Testfälle nicht nur, aber auch unter Verwendung der angegeben Beispieldaten!

3. Entwicklen Sie bitte eine *Chronology* für das Mad Datum [Knu57, Knu11]!

Kapitel 12

Fehlerbehandlung

12.1 Übersicht

Zu den wichtigsten Qualitätseigenschaften von Software gehört die *Zuverlässigkeit*. Diese hat mindenstens drei Aspekte: Ein Softwaresystem ist zuverlässig, wenn sie folgende Merkmale besitzt:

Korrektheit Ein Softwaresystem ist korrekt, wenn sein Verhalten mit der Spezifikation übereinstimmt.

Robustheit Ein Softwaresystem ist robust, wenn es für alle Eingaben eine definierte Reaktion hat.

Verfügbarkeit Ein Softwaresystem ist um so verfügbarer, je höher die Wahrscheinlichkeit ist, es zu einem gegebenen Zeitpunkt in funktionsfähigem Zustand vorzufinden.

Manche Fehler können spezifiziert werden. So kann bei der Eingabe einer ungültigen Kundennummer etwa eine entsprechende Fehlermeldung generiert werden. Nicht jeder Fehler ist aber prognostizierbar. Trotzdem muss die Anwendung mit allen auftretenden Fehlern umgehen.

In diesem Kapitel versuche ich auf einfache Weise zu zeigen, wie Sie in Java mit Fehlern sinnvoll umgehen können.

12.2 Lernziele

- Fehlerarten beschreiben können.
- Compilerwarnungen bewerten können.
- Compilerfehler verstehen und die Ursachen beheben können.
- Exceptions sinnvoll einsetzen können.
- Vorbedingungen, Nachbedingungen und Zusicherungen kennen und einsetzen können.
- Das assert statement sinnvoll einsetzen können.

12.3 Klassifikation von Fehlern

Fehler können nach vielen Kriterien klassifiziert werden. Gleich zu Beginn der Programmierausbildung haben Sie sicher Compiler-Fehler und -Warnungen kennengelernt. Compiler-Fehler sind unkritisch in dem Sinne, dass sie nicht zu Laufzeitproblemen führen. Freuen Sie sich also über

einen Compiler-Fehler. Er gibt Ihnen die Chance ein Laufzeitproblem zu vermeiden. Compiler-Warnungen können zu Laufzeitproblemen führen, müssen dies aber nicht. Ein ganz einfaches Beispiel haben Sie z. B. beim Compiler-Fehler, dass eine Variable möglicherweise nicht initialisiert worden sei. Sie können den Grad solcher Meldungen einstellen. Sind Sie sich sicher, dass die Variable initialisiert ist, so könnten Sie die den Fehlertyp auf Warnung „zurückstufen“. Ich rate dringend davon ab. Sind Sie nicht absolut sicher, dass die Warnung zu keinerlei Problemen führen kann, so beheben Sie sie. „Absolut sicher“ heißt in diesem Kontext: Sie sind bereit eine hohe Summe darauf zu wetten, dass Ihr Programm in anderen als Ihren Händen keinen dadurch verursachten Laufzeitfehler produziert.

Eine völlig andere Klassifikation verwendet den Bereich des Fehlers:

1. Eingabefehler des Anwenders (Bedienungsfehler) oder Datenfehler eines kooperierenden Systems.
2. Von der Spezifikation abweichende Verarbeitung („Programmierfehler“).
3. Systemfehler, wie „Datei nicht gefunden“, „Datei“ oder „Platte voll“ etc.
4. ...

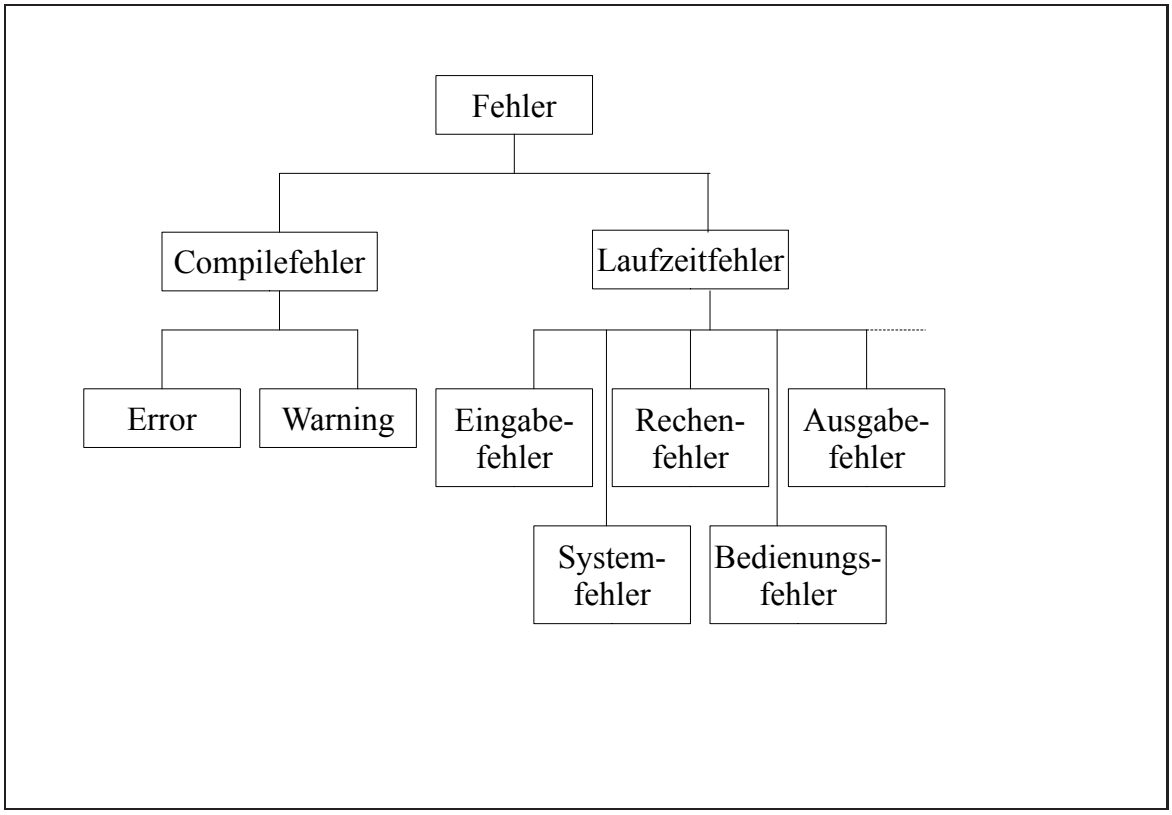


Abb. 12.1: Fehlerklassifikation

12.4 Compiler-Fehler und -Warnungen

Compiler-Fehler und -Warnungen mögen Sie als Entwickler — besonders als Anfänger — einfach nerven. Sie sollten aber die Hilfe des Compilers schnell zu schätzen lernen: Alle Fehler die der Compiler findet, können Sie vor dem tatsächlichen Einsatz Ihrer Software beheben.

Hier einige Beispiele für Compiler-Warnungen und wie Sie sich bei diesen verhalten sollten, ich meine sogar *müssen*.

Beispiel 12.4.1 (Compiler-Warnungen)

Element nicht benutzt: Attribut, Methode, Parameter oder Variable wird nie benutzt. Hier gibt es zwei Möglichkeiten:

1. Sie stehen am Anfang der Entwicklung und haben z. B. eine neue Klasse angelegt, die ein Interface implementiert. Bevor Sie die Klasse ausimplementieren, schreiben Sie sich aber schon einmal sinnvolle Testfälle. Dann ist dies Warnung tolerierbar, bis Sie meinen, mit der Klassenimplementierung fertig zu sein.
2. Meinen Sie, mit der Entwicklung einer Klasse (vorläufig) fertig zu sein, so gibt Ihnen diese Warnung Hinweise auf Dinge, die Sie übersehen haben, vergessen haben, ursprünglich anders gesehen haben etc. Dann müssen Sie Ihre Implementierung entsprechend überarbeiten.

Deprecated Warnungen, die auf die Verwendung von deprecated (veralteten, durch bessere ersetzte) Elemente, dürfen in Ihrem jetzt neu geschriebenen Code nicht vorkommen. Arbeiten Sie an einem größeren System mit, so lässt sich die Verwendung solcher Elemente nicht immer sofort vermeiden. Dann sollte ihre Behebung geplant werden, z. B. bei der nächsten Version.

List is a raw type. References to generic type List<E> should be parameterized: Diese Warnung weist auf eine Altlast hin, die schon 2004 in Java 1.5, dem sogenannten Tiger-Release endlich beseitigt wurde. Bis dahin waren die Java Collections nicht generisch. Sie konnten deshalb nur Elemente vom Typ *Object*, der Wurzel der Java-Klassenhierarchie enthalten. Entnahm man ein Element, so musst es in den „richtigen“ Typ „gecasted“ werden. Das ist aus vielen Gründen ärgerlich und gefährlich. Ausschließlich aus Gründen der Abwärtskompatibilität werden derartige Klassen noch ohne Typparameter zugelassen. Das ist nun schon viele Jahre her und es gibt keinen Grund mehr, solchen Code zu schreiben.

Generell: Warnungen weisen auf mögliche Probleme für die Weiterentwicklung oder zur Laufzeit hin. Missachten Sie sie nur, wenn Sie wirklich ganz genau wissen was Sie tun! ◀

12.5 Laufzeitfehler

In Abb. 12.2 zeigt eine Metapher für eine IT-Systemarchitektur. Der geplante Kern ist dabei der Teil, bei dem Sie genau wissen, wie Sie auf jede Situation zu reagieren haben: Auf Parameter-Werte, Rückgabewerte usw. Sie können immer eine angemessene Entscheidung treffen. Eine *Rückfrage* beim Anwender oder anderen Systemen (generell Akteuren) ist nicht notwendig.

Die Administrationsschicht gewährleistet, dass den geplanten Kern nur Dinge erreichen, mit denen er auch umgehen kann. Die Interaktionsschicht ist für die Interaktion mit Benutzern oder anderen Systemen verantwortlich. Die spontane Hülle umfasst alles, was Anwender oder andere Systeme noch außerhalb dieses Systems erledigen müssen, um mit ihm arbeiten zu können.

Tritt zur Laufzeit eine Situation ein, in welcher der spezifizierte Ablauf nicht vollständig durchgeführt werden kann, so muss im Code dafür Vorsorge getroffen werden. Dazu gibt es zunächst einmal die Möglichkeit, dass eine Methode, innerhalb derer eine solche Situation auftritt, selbst versucht, diese zu meistern. Gelingt dies nicht, so kann sie das System beenden. Das würde aber dazu führen, dass solche Dinge über nahezu alle Methoden des Systems verteilt würden. Dies wäre nicht zu handhaben, benutzerunfreundlich, Fehler schwer zu lokalisieren und das System wäre kaum noch wartbar. Besser wäre es schon, wenn die Methode einen Fehlercode an die aufrufende Methode zurückgibt. Diese könnte sich dann um Abhilfe bemühen und ggf. weiter so verfahren.

Letzeres steht aber oft in Konflikt mit der in Abschn. A.5 genannten Regel, dass verändernde Methoden den Rückgabetyp *void* haben sollen. Trotzdem ist dies oft eine akzeptable Lösung. Eine

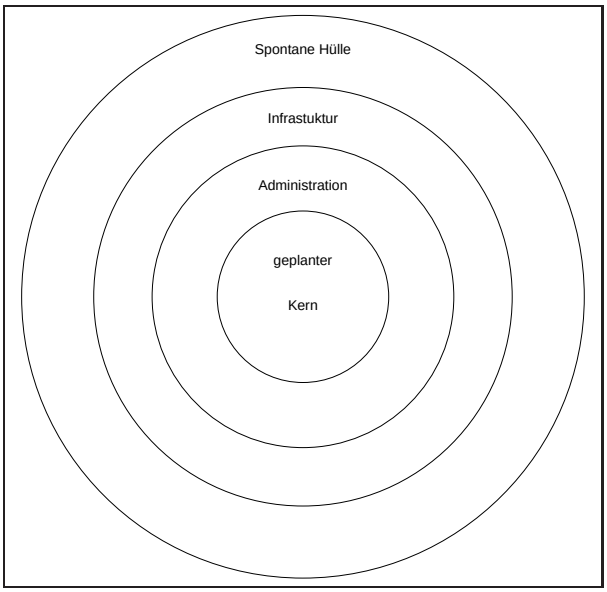


Abb. 12.2: Zwiebelmodell eines IT Systems

Reihe von Java-Collectionen Klassen lassen dem Anwender die Wahl, ob diese Strategie zum Einsatz kommen soll oder eine andere (s. u.), z. B. *java.util.Deque* oder *java.util.Queue*. Daran sehen Sie bereits, dass diese Strategie sehr verbreitet ist. Diese Strategie gerät dann an ihre Grenzen, wenn der Rückgabewert entweder ein gültiges Ergebnis oder ein Fehlerindikator (Fehlercode) sein kann.

Beispiel 12.5.1 (Problematischer Returncode)
Wir betrachten eine Methode, die eine Funktion $f : \mathbb{R} \longrightarrow \mathbb{R}$, etwa ein Polynom, definiert durch seine Koeffizienten, übergeben bekommt und die kleinste Nullstelle berechnet und zurückliefert. Wie soll eine solche Methode mitteilen, dass es keine Nullstelle gibt? Jeder *double* Wert scheidet aus. Eine Anekdote behauptet aber, das „-1.0“ für diesen Fall schon gewählt wurde. ◄

Dieses Problem war ein wichtiger Grund für die Einführung von Exceptions in C++. Es tritt dort nämlich beim *new*-Operator auf. Dieser liefert die Adresse (ähnlich Referenz in Java) zurück, falls das Objekt angelegt werden konnte und sonst 0. Nach jedem *new* müssten Sie also auf 0 prüfen. Das verunstaltet den Code auf Dauer doch sehr. Eine Lösung hierfür sind *Exceptions*. Dieses Konzept bietet auch Java. Ich beschreibe es in Abschn. 12.6.

12.6 Exceptions

Das objektorientierte Grundprinzip (siehe Kap. 1, Seite 1) wird auch für einen wichtigen Mechanismus verwendet, um mit Fehlersituationen umzugehen. Tritt in Java eine Ausnahmesituation auf, so kann darauf mittels des Erzeugens eines Objekts reagiert werden. Dies Objekt wird dann von einem parallel zur ursprünglichen Anwendung (main) laufenden Vorgang (Thread) bearbeitet. Wie bereits in Abschn. 12.5 geschrieben, gibt es auch hier zwei Möglichkeiten.

Zunächst am einfachsten ist es sozusagen „Schwarzer Peter“ zu spielen.

Bemerkung 12.6.1 (Schwarzer Peter)
Schwarzer Peter ist ein Kartenspiel, bei dem es 15 bzw. 18 Kartenpaare und eine weitere Karte, den „Schwarzen Peter“ gibt. Die Karten werden gemischt und gleichmäßig an die Spieler verteilt.Erhält ein Spieler dabei ein Paar, so legt er es sofort ab. Nun zieht z. B. der Spieler links vom Geber eine

Karte des Gebers. Hat er dadurch ein Paar, so legt er es ab. Dann zieht dessen linker Nachbar usw. Verloren hat der Spieler, der zum Schluss nur noch den „Schwarzen Peter“ hat. ◀

Tritt eine Ausnahmesituation auf, so werfe ich eine *Exception*. Die Klasse *Exception* hat einen Konstruktor, der eine Nachricht als Parameter bekommt, etwa so:

```
...
throw new Exception("Hier ist etwas schiefgegangen");
```

Das Java Schlüsselwort zum Werfen einer Exception ist *throw*. Wirft eine Methode eine Exception, so kann sie dies dem aufrufenden Code mitteilen. Handelt es sich nicht um eine *RuntimeException*, s. u., so muss sie das sogar tun. Dies geschieht durch Angabe in der Signatur. Das Schlüsselwort dafür ist *throws*. Etwas weiter vervollständigt muss das obige Beispiel also lauten;

```
public void noop() throws Exception{
  ...
  throw new Exception("Hier ist etwas schiefgegangen");
  ...
}
```

Nun ist es nicht immer angemessen, in der Programmierung „Schwarzer Peter“ zu spielen. Die Alternative ist es, den Code-Teil auszuführen zu „versuchen“. Dazu gibt es in Java das Konstrukt des *try-catch*. Dies beginnt mit dem Schlüsselwort *try*:

```
try{
   //Code in dem eine Exception geworfen werden kann.
   ...
}catch(Exception e){
  //Umgang mit Exception e
}
```

Dem *try* folgt also der Block, in dem mit einer Exception gerechnet werden muss. Oft ist dies der Aufruf einer Methode aus einer Klassenbibliothek, die durch eine *throws*-Deklaration mögliche Exceptions signalisiert. Zu diesem Konstrukt gehört immer ein *try*- und mindestens ein *catch*-Block.

Die Wurzel der Hierarchie von Ausnahmen in Java bildet die Klasse *Throwable*. Die direkten Unterklassen von *Throwable* sind *Error* und *Exception*. *Error*s zeigen schwere Fehler an, die Anwendungen kaum versuchen sollten zu behandeln. Sie stammen meistens aus der JVM. Hierauf kann im Code kaum sinnvoll reagiert werden.

Beispiel 12.6.2 (Error und Exception)
Es gibt eine Klasse *NoClassDefFoundError*. Wie oben ausgeführt macht es keinen Sinn auf diesen *Error* in einem *catch*-Block zu behandeln ([BG05], Puzzle 44):

```
public class Strange1 {
   public static void main(String[] args) {
      try {
         Missing m = new Missing();
      } catch (java.lang.NoClassDefFoundError ex) {
           System.out.println("Got it!");
      }
   }
}
```

Wenn Sie aber mit derartigen möglichen Problemen zu tun haben, sollten Sie hier *Reflection* verwenden (siehe Kap. 19), etwa so:

```
public class Strange {
   public static void main(String[] args) throws Exception {
      try {
         Object m = Class.forName("Missing").newInstance();
      } catch (ClassNotFoundException ex) {
         System.err.println("Got it!");
      }
   }
}
```

◀

Ob für Exceptions im Code Vorsorge getroffen werden muss, hängt von der Art der Exception ab. *RuntimeException*s müssen weder abgefangen noch in einer *throws*-Klausel deklariert werden. Alle anderen Exceptions — die *checked exceptions* — müssen Sie berücksichtigen!

Lassen Sie sich bitte durch den obigen Code, den ich hier zur Abschreckung wiederhole

```
catch(Exception e){
    //Umgang mit Exception e
  }
```

nicht dazu verleiten, einen leeren *catch*-Block zu schreiben. Das folgende Beispiel verdanke ich Mario Hanna aus dem SS 2011:

Beispiel 12.6.3 (Verlorene Exception)
In einer Klasse stand Folgendes, soweit es dieses Beispiel angeht:

```
public class Console {
010   Scanner sc;
     ...
087   public char charReader(){
094      try{
096         character = this.sc.next().charAt(0);
            ...
100      }catch(Exception e){
101
102      }
```

Das ohne Begründung package sichtbare Attribut *sc* wurde nirgends initialisiert. Soweit so schlecht. In Zeile 096 gibt es also eine *NullPointerException* und es wird in den *catch*-Block gesprungen! Dort passiert nix! Das war also ein Satz mit „X“! Dem Autor fällt also gar nicht auf, dass die Initialisierung vergessen wurde! Es kann so auch gar nicht auffallen. Also: Wenn Sie schon eine Exception abfangen, dann fangen Sie auch etwas damit an. Mindesten die Stacktrace sollten Sie im *catch*-Block mittels *e.printStackTrace()* ausgeben! Sonst bekommen Sie unter Umständen gar nicht mit, dass etwas schiefgegangen ist. ◀

Einen Hinweis auf eine weitere Möglichkeit für verlorene Exceptions habe ich aus [Blo08] bekommen:

Wenn in einer *finalize*-Methode ein Exception geworfen wird (also nicht in einem catch-Block abgefangen wird), so wird sie ignoriert und die finalize-Methode wird abgebrochen ([GJS+14], Abschn. 12.6).

Das ist nicht mehr (besonders) wichtig, da die Methode *finalize* seit Java 9 deprecated ist. Da es aber viel prä-Java 9 Code gibt, ist dieser weiterhin anfällig für *finalizer-Attacken*, siehe [Blo18].

Ein weiteres Element zum Umgang mit Exceptions ist ein *finally*-Block

```
public class FinallyWorks {
  static int count = 0;
```

```
  public static void main(String[] args) {
    while(true) {
      try {
        if(count++ == 0)
          throw new ThreeException();
        System.out.println("No exception");
      } catch(ThreeException e) {
        System.err.println("ThreeException");
      } finally {
        System.err.println("In finally clause");
        if(count == 2) break;
      }
    }
  }
}
```

Der *finally*-Block wird immer ausgeführt, unabhängig davon, ob der *try*-Block bis zum Ende ohne Exception ausgeführt wird oder in einen *catch*-Block gesprungen wird.

Java kommt mit einer umfangreichen Sammlung von Exceptions. Abbildung 12.3 zeigt einen Ausschnitt. Die Exception *RuntimeException* und ihre Unterklassen sind die *unchecked exceptions*.

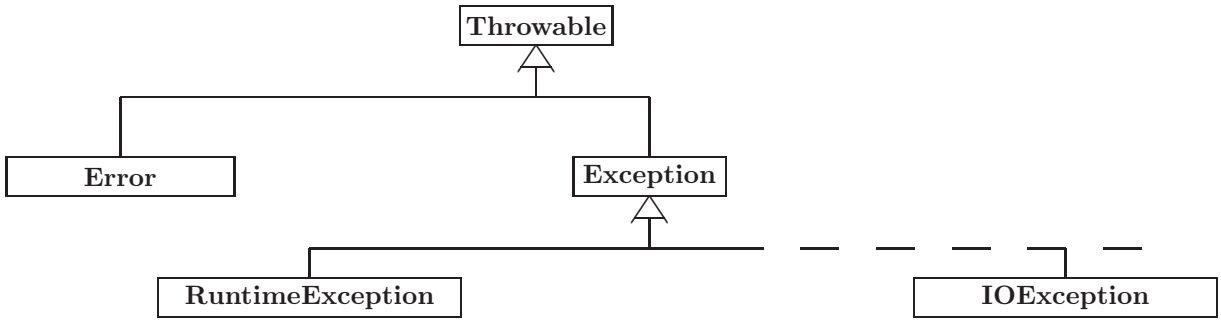


Abb. 12.3: Die Wurzel der Exception Hierarchie

Es ist kein Compiler-Fehler eine *RuntimeException* nicht zu behandeln. Alle anderen Exceptions sind *checked exceptions* und müssen behandelt werden. Das heißt, Sie werden *try-catch-finally*-Blöcke schreiben müssen.

Hier ein Beispiel für einige der Überlegungen, die Sie dabei anstellen müssen. Für einige Einzelheiten verweise ich auf Kap. 14.

Beispiel 12.6.4 (IO-Exceptions)
Beim Zugriff auf eine Datei können verschiedene Dinge „schief gehen“. So kann die Datei nicht existieren oder zwar existieren, aber es kann einen Fehler beim Zugriff geben. Trennt man dies nach dem Motto teile und herrsche auf, so erhält man Code wie den folgenden (io.FileReadExample01):

```
List<String> l = new ArrayList<String>();
BufferedReader br=null;
try {
    br = new BufferedReader(
         new InputStreamReader(
```

```
            new FileInputStream("./src/io/neu.txt")));
       } catch (FileNotFoundException e) {
            e.printStackTrace();
       }
   String line = null;
   if (br != null) {
      try {
          while ((line = br.readLine()) != null) {
               l.add(line);
          }
      } catch (IOException e) {
               e.printStackTrace();
      }
      try {
         br.close();
      } catch (IOException e) {
           e.printStackTrace();
   }
}
```

Dieser Code hat verschiedene Schwächen:

1. Die Initialisierung des *BufferedReader*s unterdrückt nur eine Warnung des Compilers, vermeidet aber keine *NullPointerException*.
2. Die Abfrage *if (br != null)* ist aus dem gleichen Grund unschön.
3. Es gibt zwei *try*-Blöcke mit zugehörigem *catch*-Block für das Fangen einer *IOException*.
4. Trotzdem ist nicht sichergestellt, dass der *BufferedReader* geschlossen wird, z. B. in einem *finally*-Block.

In dieser Version kann auf die Variable *br* in *catch*- oder *finally*-Block zugegriffen werden.

Die folgende Version (io.FileReadExample02) ist schon etwas kompakter, aber immer noch nicht befriedigend.

```
List<String> l = new ArrayList<String>();
try {
   BufferedReader br = new BufferedReader(
                       new InputStreamReader(
                       new FileInputStream("./src/io/neu.txt")));
   String line;
   while ((line = br.readLine()) != null) {
          l.add(line);
        }
   br.close();
   } catch (IOException e) {
       e.printStackTrace();
   }
```

Hier gefallen mir zwei Dinge noch nicht:

1. Es ist nicht sichergestellt, dass der *BufferedReader* geschlossen wird, denn im *catch*-Block ist er ebensowenig bekannt, wie in einem etwaigen *finally*-Block. Dies ist ein wichtiger Unterschied zur ersten Version.
2. Es wird nicht zwischen *IOException* und der mehr spezifischen *FileNotFoundException* unterschieden.

Nun implementieren alle zum Erstellen des *BufferedReader* benötigten Klassen das Interface *AutoCloseable*. Ich verwende deshalb in io.FileReadExample03 ein *try-with-resources*:

```
List<String> l = new ArrayList<String>();
try(BufferedReader br = new BufferedReader(
                        new InputStreamReader(
                        new FileInputStream("./src/io/neu.txt")))) {

   String line;
   while ((line = br.readLine()) != null) {
      l.add(line);
   }
} catch (FileNotFoundException e) {
      e.printStackTrace();
} catch (IOException e) {
      e.printStackTrace();
}
```

Hiermit sind jetzt alle bisherigen Kritikpunkte beseitigt:

1. Die Ressource, hier der *BufferedReader* br, wird geschlossen, egal ob es zu einer *FileNotFoundException* oder *IOException* kommt oder nicht, da die Klasse *AutoCloseable* implementiert.
2. Eine Unterscheidung zwischen *FileNotFoundException* und *IOException* erfolgt. Letztere könnte beim automatischen Schließen geworfen werden.

◀

Beispiel 12.6.4 zeigt Ihnen, dass *try-with-resources* oft das Konstrukt der Wahl ist.

Benötigen Sie mehrere *catch*-Blöcke, wie in io.FileReadExample03 und stehen die Exceptions nicht in einer Vererbungsbeziehung, so können Sie die *catch*-Blöcke zusammenfassen, z. B. :

```
catch(IOException|SQLException e){
   ...
}
```

Der Compiler prüft sowohl bei mehreren *catch*-Blöcken als auch bei einem multi-*catch*-Block, ob alle *Exception*s bearbeitet werden können.

Überlegen Sie sich genau, wann Sie *checked Exceptions* und wann Sie *RuntimeExceptions* verwenden: *RuntimeExceptions* zwingen Nutzer der Methode nicht, den Aufruf in einem *try-catch-Block* zu kapseln. Handelt es sich z. B. um eine *Collection*, die leer sein kann, so können Sie eine Methode *isEmpty* anbieten und die entsprechende *Exception* als *RuntimeException* implementieren oder eine vorhandene *RuntimeException* verwenden. Nutzer können dann *isEmpty* verwenden, wenn sie eine Exception vermeiden wollen udn brauchen keinen *try-catch-Block*. In anderen Fällen werden Sie sich vielleicht entscheiden eine *checked exception* zu verwenden, um Nutzer explizit zur Behandlung zu zwingen.

12.7 Fehlererkennung zur Laufzeit

Wissen Sie genau, welche Fehler auftreten können, so können Sie versuchen diese entsprechend zu behandeln. Als letztes Auffangbecken kann ein *outer-try-Block* dienen, den Sie auf der obersten Ebene, in einer Java-Anwendung also z. B. in der startenden *main*-Methode schreiben. Etwa so:

```
public class OuterTryBlock {
   public static void main(String[] args) {
      try {
```

```
         //start der eigentlichen Arbeit
      } catch (Exception e) {
          e.printStackTrace();
        }
   }
}
```

Eine Alternative zu einem solchen *outer-try-Block* ist die Verwendung eines *DefaultException-Handler*s. Dazu gibt es eine Klassenmethode in der Klasse *Thread*. Hier ein einfaches Beispiel:

```
public class DefaultExceptionHandler {
     public static void main(String[] args) {
        Thread.setDefaultUncaughtExceptionHandler(new UncaughtExceptionHandler() {
            @Override
            public void uncaughtException(Thread t, Throwable e) {
                System.err.println("Thread " + t + " has thrown " + e);
                e.printStackTrace();
                System.out.println("Das Programm musste
                                    wegen eines Fehlers beendet werden!");
            }
        });
        throw new IndexOutOfBoundsException();
    }
}
```

Es ist Ihre Aufgabe bei der Programmierung möglichst alle denkbaren Fehler zu erkennen und je nach Art von vorneherein zu verhindern, zu beheben oder die Auswirkungen so gut es geht zu begrenzen. In welchen Teilen Ihrer Anwendung Sie welche Art von Fehlern behandeln, hängt von der Architektur ab. Abbildung 12.2 kann als erste Richtschnur dienen.

12.8 Vorbedingungen

Vorbedingungen werden in der Informatik in verschiedenen Kontexten eingesetzt:

1. Anwendungsfälle: Im Softwareengineering und speziell im Requirementsengineering werden Anwendungsfälle zur Spezifikation der funktionalen Anforderungen an Software eingesetzt. Eine Vorbedingung spezifiziert hier, welche Bedingungen Akteure sicherstellen müssen, bevor sie diesen Anwendungsfall starten können.

2. Operationen: In Interfaces können Vorbedingungen spezifiziert werden, die Bedingungen spezifizieren unter denen diese Operation aufgerufen werden kann.

3. Methoden: Methoden einer Klasse (die etwa Operationen eines Interfaces implementieren) können Vorbedingungen haben. Diese geben dann an, unter welchen Bedingungen eine Methode aufgerufen werden kann.

Vorbedingungen sind Teil eines Vertrages, den der Entwickler des entsprechenden Artefakts mit anderen Artefakten schließt: Wird die Vorbedingung eingehalten, so wird die korrekte Funktion des Artefakts garantiert.

Sie können sich durchaus auf den Standpunkt stellen, dass Sie das Einhalten einer Vorbedingung nicht überprüfen müssen. In der Praxis werden Sie Vorbedingungen aber mit hoher Wahrscheinlichkeit überprüfen. Formulieren Sie in der Vorbedingung, dass Sie im Falle ihrer Verletzung eine Exception werfen oder einen Returncode zurückgeben, so müssen Sie das dann auch tun.

12.9 Nachbedingungen

Nachbedingungen sind das Gegenstück zu Vorbedingungen und kommen in den gleichen Kontexten zum Einsatz. Eine Nachbedingung ist eine Verpflichtung, die Entwickler eines Artefakts anderen gegenüber eingehen: Bei Verwendung unter Einhaltung der Vorbedingung wird garantiert, das am Ende der Ausführung die Nachbedingung erfüllt ist.

Vorbedingungen können nicht immer sicher eingehalten werden. So kann ein falscher Dateiname angegeben werden. Selbst wenn dies vor dem Aufruf einer Methode noch sichergestellt ist, könnte die Datei immer noch zwischenzeitlich gelöscht werden. Insofern können die entsprechenden *throws*-Klauseln bei Methoden mit zu möglichen Nachbedingungen gerechnet werden.

12.10 Zusicherungen

Zusicherungen gibt es in Java in der Form von Assertions. Assertions ermöglichen es Code in Java Programmen einzubauen, der nur bei einer bestimmten Einstellung ausgeführt wird. So könne etwa zu debug-Zwecken zusätzlich Ausgaben erfolgen, wenn eine definierte Bedingung eintritt. Das Schlüsselwort hierfür ist *assert*. Die Syntax ist

```
assert Ausdruck1

assert Ausdruck1 : Ausdruck2
```

Ausdruck1 muss *boolean* oder *Boolean* sein. *Ausdruck2* darf nicht *leer* sein. Ist *Ausdruck1* falsch, so wird in der zweiten Form *Ausdruck2* ausgeführt und ein Error, genauer ein *AssertionError* geworfen, andernfalls passiert nichts.

Ob Assertions ausgeführt werden, wird beim Laden der Klasse entschieden. Dies passiert durch den JVM-Parameter *-ea* (enable assertions)

So können Sie beim Auftreten eines Fehlers zusätzliche Informationen zur Unterstützung der Fehlerlokalisierung erhalten, in dem Sie die *assert*s aktivieren.

Als Faustregel sollten Sie Folgendes beachten: Nicht öffentliche Methoden und Konstruktopren sollten ihre Parameter mittels *assert* auf Verletzung von Vorbedingungen überprüfen.

12.11 Fehlerbehandlungsstrategien

Bei der Behandlung von Fehlern unterscheide ich danach, ob die Erkennung durch den Programmierer möglich oder unmöglich ist und ob sie erfolgt oder nicht. Ein Fehler der VM, nicht mehr verfügbarer Speicher o. ä. kann vom Programmierer nicht erkannt werden. Da die dadurch ausgelösten Exceptions *unchecked* sind, kann nur durch einen *outer try Block* oder einen *Exception Handler* eine gewisse Vorsorge getroffen werden.

Exceptions sind eine software-technisch gute Lösung, verursachen aber auch Aufwand. Es muss also oft entschieden werden, ob ein Fehler durch eine Exception oder einen Returncode gemeldet wird. Wichtig ist es, hierfür eine einfache Strategie festzulegen, die von allen Entwicklern eingehalten wird. Die Klassen der Java-Klassenbibliothek bieten oft beides an. Verwenden Sie allerdings Methoden, die Exceptions werfen, so haben Sie in diesem Bereich der Anwendung keine Wahl. Am Besten ist es dann, dies in einer Schicht zu kapseln, wenn in anderen Bereichen mit Returncodes gearbeitet werden soll.

12.12 Fehlermeldungen

Fehlermeldungen können als Strings im Code stehen. Das hat den Vorteil eines „Quasikommentars“. Sie können direkt lesen, welcher Fehler an dieser Stelle möglich erscheint. Es hat aber entscheidende Nachteile:

1. Die Fehlermeldungen werden über den gesamten Code verteilt.
2. Die Konsistenz der Fehlermeldungen ist schwierig zu gewährleisten.
3. Eine Übersetzung ist praktisch nicht möglich.

Etwas besser ist es schon, wenn Sie die Strings der Fehlermeldungen externalisieren. Dazu müssen Sie grundsätzlich genau das tun, was auch für die anderen Optionen notwendig ist: Sie brauchen Schlüssel, um die Fehlermeldungen zu identifizieren. So werden die Fehlermeldungen in eine Datei ausgelagert. Java bietet dazu einfache Möglichkeiten (siehe Abschn. 21.5), und Eclipse automatisiert das sogar.

Handelt es sich um eine Datenbankanwendung, so werden Sie die Fehlermeldungen in einer Datenbanktabelle speichern und bei Bedarf daraus lesen.

12.13 Historische Anmerkungen

Assertions gibt es in Java seit 1.4. try-with-resources wurde in Java 7 zusammen mit dem Interface *AutoCloseable* eingeführt.

12.14 Aufgaben

1. Ein vollständiges *try*-Konstrukt enthält außer dem *try*-Block einen oder mehrerer *catch*-Blöcke und ggf. einen *finally*-Block. Wann wird welcher Teil ausgeführt?
2. Welche Exceptions müssen Sie abfangen, welche nicht?
3. Worauf müssen Sie achten, wenn Sie über Fehler mittels eines Returncodes informieren?
4. Warum kann es empfehlenswert sein, eigene Exceptions zu schreiben? Sie können doch auch einfach eine Exception mit Ihrer Nachricht werfen!
5. Schreiben Sie bitte eine *main*-Methode, die einen Dateinamen einer Textdatei als Parameter übergeben bekommt, diese Datei einliest und Zeile für Zeile auf die Konsole ausgibt! Es wird eine vollständige, sinnnvolle Fehlerbehandlung erwartet!
6. Gibt es Situationen, in den es sinnvoll ist, eine *NullPointerException* zu werfen? Begründen Sie bitte Ihre Antwort!

Kapitel 13

Javadoc

Dokumentation ist der Lebertran der Softwareentwicklung. Manager wissen, dass sie wirken muss, denn Entwickler hassen sie so sehr. Gerald M. Weinberg

13.1 Übersicht

Dieses Kapitel beschreibt den vielfach bewährten Einsatz von Javadoc um Klassen und Schnittstellen und andere Java-Elemente zu dokumentieren. Selbstverständlich ist dies nur ein Teil der Dokumentation. Im Software-Engineering werden bereits zu frühen Zeitpunkten Schnittstellen und Klassenstrukturen dokumentiert. Heute verwendet man dafür z. B. UML oder SysML, je nach Anwendungsgebiet. Für Java-Code ist Javadoc üblich. Liegen die Namen der Elemente früh genug fest, so sollte dies bereits in Analyse oder Design-Sichten einsetzbar und für Java-Code wiederverwendbar sein.

Im Kern geht es hier darum den Code um Informationen zu ergänzen, die der Java-Compiler ignoriert und die von dem Programm javadoc in einen Satz von HTML-Seiten umgewandelt werden. Innerhalb dieser kann man dann einfach navigieren, um die Informationen zu erreichen, die zur Nutzung der Klasse erforderlich sind. Es gehört zum guten Java-Stil diese Form von Dokumentation zu schreiben. Ob auf Englisch oder in einer anderen Sprache hängt von der Zielgruppe ab.

Auch in anderen Programmiersprachen gibt es entsprechende Tools. So können Kommentare im Javadoc-Stil auch mittels Doxygen für C oder C++ geschrieben werden. Doxygen unterstützt mehrere Ausgabeformate. HTML und LaTeX sind wohl die Wichtigsten. In *Ruby* gibt es entsprechend rdoc (Ruby Doc). Mittels *Yard* können Sie ganz ähnlich wie mit Javadoc dokumentieren.

13.2 Lernziele

- Javadoc Kommentare schreiben können.
- Javadoc sinnvoll verwenden können.
- Skizzieren können, wie man Javadoc erweitern kann.
- Einige nützliche Elemente von HTML kennen und für Javadoc-Kommentare einsetzen können.

13.3 Einführung

Javadoc Kommentare beginnen mit /** und enden mit */. Sie werden — wie andere Kommentare auch — nicht vom Compiler verarbeitet, sondern eben von Javadoc. Dieses Programm erstellt aus

dieser Art Kommentar HTML-Seiten. Werden sie gut und professionell geschrieben, so kann das Ergebnis lesbar und nützlich sein.

Da Javadoc HTML-Code erzeugt, kann HTML nach Bedarf im Kommentar verwendet werden. Ich empfehle mit dieser Möglichkeit sparsam umzugehen (es sei denn, Sie haben Web-Design erfolgreich gelernt). Die m. E. wichtigsten Einsatzmöglichkeiten von HTML in diesem Kontext erläutere ich an den entsprechenden Stellen.

Werden keine Javadoc-Kommentare angegeben, so generiert Javadoc einen Basissatz von Seiten, der den Namen der Klasse, die Stellung in der Vererbungshierarchie und die implementierten Schnittstellen enthält. Es gibt für die Elemente, also Attribute (Fields), Konstruktoren, Methoden, ..., eine Zusammenfassung (Summary) und eine Detailübersicht. Alle diese sind so natürlich fast nichtsagend: Sie zeigen nur die Signaturen.

Um diese HTML-Seiten zu erzeugen, verwenden Sie in Eclipse die Exportfunktion für das entsprechende Paket. Dort wählen Sie Javadoc aus. Standardmäßig wird in ein doc-Verzeichnis auf der Ebene des src-Verzeichnisses generiert. Beim Generieren können Sie die Sichtbarkeit angeben, ab der generiert werden soll:

- *public*: Nur für *public* Elemente wird generiert.
- *protected*: Für *protected* und für *public* Elemente wird generiert.
- *package*: Für Elemente mit den Sichtbarkeiten *package*, *protected* und *public* wird generiert.
- *private*: Für alle Elemente wird generiert.

Die letztgenannte Möglichkeit verwende ich für die Generierung der HTML-Dokumentation zu den Programmbeispielen, die ich Ihnen zur Verfügung stelle bzw. empfehle Ihnen diese Option, wenn Sie die API-Doku selbst aus dem Source-Code generieren.

Beim Generieren von Javadoc mittels Eclipse können Sie angeben, auf welche Projekte und Archive eine Verlinkung der Dokumentation erfolgen soll.

Ein Javadoc Kommentar wird nur dann dem beschriebenen Element korrekt zugeordnet, wenn er direkt davor steht.

Die aus den Javadoc-Kommentaren generierte Dokumentation besteht aus verlinkten HTML-Seiten. Abbildung 13.1 zeigt das Schema. Im größeren, rechten Teil sehen Sie die Dokumentation

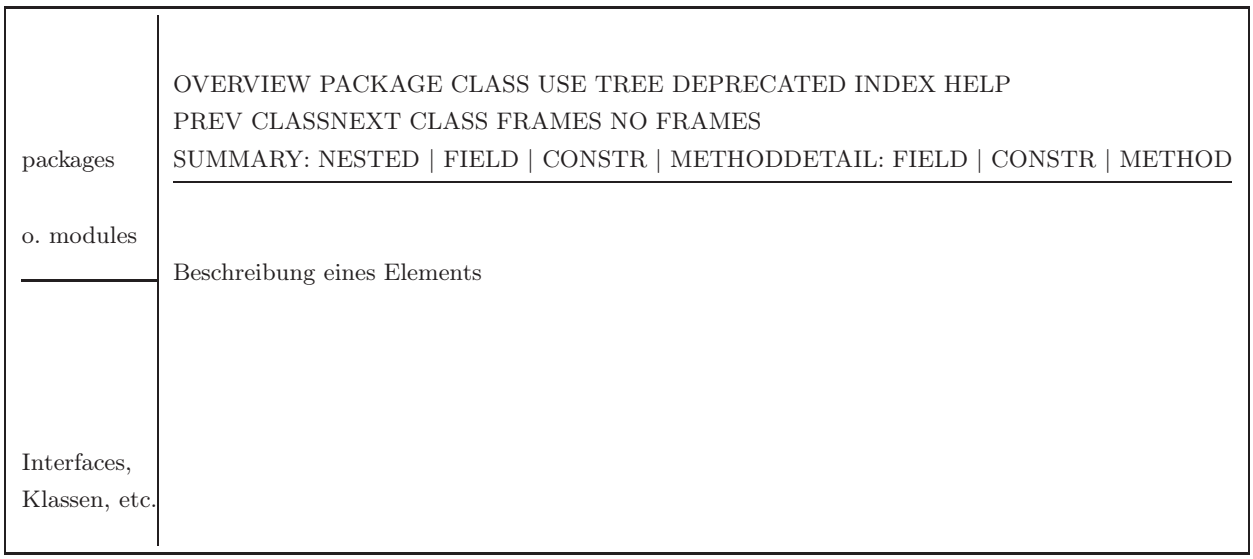


Abb. 13.1: Java API Dokumentation — Schema

eines Elements, z. B. eines Interfaces oder einer Klasse. Links ehen Sie im oberen Teil eine Liste von Paketen oder Modulen und im unteren Teil die Interfaces, Klassen etc. im ausgewählten Paket.

Die Pakete erscheinen im oberen Teil aphabetisch. Die Elemente eines Pakets in der Reihenfolge *Interfaces*, *Klassen*, *Enums*, *Exceptions*, *Errors*, *Annotation Types*, jeweils alphabetisch sortiert.

Bemerkung 13.3.1 (Javadoc in Java 9)
In Java 9 können Sie oben links zwischen einer Liste von packages und von modules wählen. ◀

Im rechten Teil sehen Sie zu jedem Element den voll qualifizierten Klassennamen sowie die Spezifikation, z. B.

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

Es folgt dann eine mehr oder weniger lange Beschreibung des Elements.

Zum Schluss der Beschreibung steht, seit welcher Java-Version es dieses Element gibt und ggf. Hinweise auf interessante ähnliche Elemente o. ä.

Die weitere Darstellung eines Elements hängt von der Art des Elements ab:

Interface Hier sehen Sie u. a. die bekannten implementierenden Klassen. Die weiteren Informationen sind weitgehend mit denen bei einer Klasse identisch.

Klasse Nach den oben genannten Elementen folgen Übersichten der enthaltenen Elemente (Summary):

- Field Summary: Zusammenfassung der Attribute und Klassenattribute.
- Constructor Summary: Zusammenfassung der Konstruktoren.
- Method Summary, mit je einem Tab für *All Methods*, *Static Methods (Klassenmethoden)*, *Instance Methods*, *Concrete Methods*, *deprecated Methods*. Für *abstrakte Methoden* gibt es ggf. einen weiteren Tab, ebenso bei *Interfaces* für *default Methoden*. Durch Anklicken des jeweiligen Elementnamens kommen Sie zur detaillierten Beschreibung.

  Im Anschluss der in der jeweiligen Klasse definierten Methoden folgen Verweise, auf die Methoden aus Oberklassen.

Sie können in dieser Ansicht mit oder ohne *Frames* arbeiten, je nach dem, was Sie schneller zum gesuchten Ergebnis führt.

13.4 HTML in Javadoc

Da Javadoc HTML generiert, können Sie in Javadoc-Kommentaren HTML dort verwenden, wo es Ihnen angemessen erscheint. Javadoc kann auch die Syntax des HTML Codes prüfen, siehe Abschn. 13.7.

Ich empfehle aus zwei Gründen, HTML-tags in Javadoc sparsam einzusetzen:

1. Wer kein guter Web-Designer (oder Schriftsetzer) ist, neigt leicht dazu ein Dokument mit Auszeichnungen zu überfrachten.

2. Nicht alle Tags werden von allen Browsern gleich „gut“ verarbeitet.

Ich beschränke mich daher auf folgende Empfehlungen:

1. HTML-eigene Zeichen; Die folgende Tabelle enthält Zeichen, die in HTML verwendet werden. Wollen oder müssen Sie diese in Javadoc einsetzten, so müssen Sie sie wie in der folgenden Tabelle beschrieben angeben:

Zeichen	Beschreibung	Name in HTML	Unicode in HTML
"	Anführungszeichen oben	&quot;	&#34;
&	Ampersand-Zeichen, kaufmännisches Und	&amp;	&#38;
<	öffnende spitze Klammer	&lt;	&#60;
>	schließende spitze Klammer	&gt;	&#62;

Verwenden Sie dort bitte die Elemente aus der zweiten Spalte!

Diese *Ersatzzeichen* brauchen Sie nicht für html-tags. Sie müssen Sie verwenden, wenn Sie zb. „a < b“ schreiben wollen.

2. Gliederung größerer Texte in Absätze. Ein neuer Absatz beginnt mit `<p>`

3. Man kann zur Illustration Codeteile in Javadoc aufnehmen. Diese werden dann durch `<pre>` ... `</pre>` (preformatted) begrenzt. So erzeugt man eine Schriftart, in dem jeder Buchstabe die gleiche Breite hat. Eine bekannte Schrift dieser Art ist `Courier`. Nach de.selfhtml.org/html/text/logisch.htm#elemente gilt es als guter Stil, in dieser Situation zusätzlich eine logische Auszeichnung vorzunehmen. In diesem Fall ist das `<code>`, so dass sich insgesamt `<pre><code>...</code></pre>` ergibt. Die Verwendung von `<code>` wird auch von Oracle empfohlen. Dies gilt für Java Schlüsselworte, Paket-, Klassen-, Schnittstellen-, Attribut- und Parameternamen ebenso wie für Code-Beispiele. Sie hierzu auch {@code}.

4. In manchen Fällen sind Tabellen nützlich.

5. Tags wie `<img>` etc. können bei Bedarf für die Einbindung von Abbildungen verwendet werden. Dies ist sinnvoll, wenn komplexe Dinge durch Visualisierung besser verständlich werden.

Der erste Satz eines Javadoc-Kommentars wird in Zusammenfassungen (Summary) nach dem Namen des Elements angezeigt. Der erste Satz ist zu Ende, wenn ein Satzendepunkt kommt, also ein „.“ gefolgt von einem blank, tab, Zeilenende oder Tag kommt. Handelt es sich nicht um einen Satzendepunkt, sondern z. B. um einen Abkürzungpunkt, so können HTML-Sonderzeichen, etwa „. “ (non breaking space) anstelle von .␣ verwendet werden. Die bessere Möglichkeit ist aber die Verwendung von `{@literal}` in solchen Situationen.

Beispiel 13.4.1 (Non breaking Space)
Was passiert, wenn Sie es anders machen, konnten Sie an der Klasse *FileSystemView* beim Stand aus Java 7 besichtigen: In der Method Summary der API-Dokumentation sahen Sie in vielen Methoden am Ende „(i.e.“ oder „(e.g.“. So wird diese Abkürzungspunkt in den Methoden *getFiles*, *isComputerNode*, *isDrive* und anderen als Satzendepunkt interpretiert, da ein Blank und keine non braking space „ “ folgt. In Java 8 war dies am 06.09.2014 nur noch bei der Methode *getFiles* der Fall. ◄

13.5 Javadoc-Dateien

Die Source-Dateien, aus denen Javadoc HTML generiert, sind folgende:

Class Source Dateien Das sind einfach die Klassendateien (.java). Aus diesen Dateien erzeugt javadoc eine gleichnamige Datei, nur nun mit der Endung .html statt .java.

Package Comment Dateien Die sind die *package-info.java* Dateien im Sourceordner des Pakets. Der javadoc-Inhalt dieses Pakets wird an den Anfang des package-summary.html kopiert.

Module Comment Dateien Die sind die *module-info.java* Dateien im Sourceordner des modules.

Overview Comment Dateien Diese beschreiben mehrere Pakete. Sie heißen üblicherweise *overview.html* und werden an die Spitze des entsprechenden Source (Teil-) Baums geschrieben.

Diverse Weitere Dateien die unverändert übernommen werden, wie html-, Bild- etc. Dateien. Diese schreibt man in ein Verzeichnis *doc-files* in einem Verzeichnis, dass Source Dateien des Pakets enthält.

Bemerkung 13.5.1 (package.html)
Seit Java 1.2 konnte eine Datei *package.html* in dem Source-Verzeichnis des Pakets angelegt werden. Seit Java 1.5 wird der Name *package-info.java* empfohlen. ◄

Javadoc verarbeitet nur solche Dateien, deren Name nach Entfernen der Endung .java ein gültiger Klassenname, Paketname etc. ist.

13.6 Javadoc-Befehle

Javadoc-Befehle steuern die Verbindung zwischen den verschiedenen Teilen einer HTML-Datei und zu anderen HTML-Dateien. Sie beginnen mit einem „@“. Man unterscheidet dabei Block Tags und Inline Tags.

Block Tags Diese haben die Form `@tag` und können nur in der *tag section* im Anschluss an die Hauptbeschreibung verwendet werden. Das „@“-Zeichen eines Block Tags muss das erste Zeichen einer Zeile, abgesehen von „*“, „/**“ und whitespace sein.

Inline tags Diese können überall in der Hauptbeschreibung oder den Kommentaren für Block Tags verwendet werden. Sie haben die Form `{@tag}`

Mit dem Stand von Java 10 gibt es die folgenden Tags:

Tag	Beschreibung	Seit JDK/SDK
@author	Verwendet mit -author option, mehrfach möglich	1.0
{@code}	Äquivalent zu `<code>{@literal}</code>` Bei mehreren Zeilen innerhalb <pre> ... </pre> verwenden.	1.5
@deprecated	Alle Elemente. Nicht mehr verwenden, auch wenn das Element noch funktionieren könnte	1.0
{@docRoot}	Relativer Pfad zur Wurzel der Dokumentation	1.3
@exception	Synonym für @throws	1.0
{@index}	Definiert einen suchbaren Indexeintrag	9
{@implSpec}		8
{@inheritDoc}	Kopiert die Dokumentation des nächsthöheren Elements in der Vererbungshierarchie	1.4
{@link}	Verweis auf ein anderes Element package.class#member label	1.2
{@linkplain}	Wie @link, aber in einfachem Font	1.4
{@literal}	Text ohne html oder javadoc tags zu interpretieren	1.5
@param	Erforderlich für jeden Parameter	1.0
@return	Erforderlich für jede Methode mit Rückgabetyp nicht *void*	1.0
@see	Verweis auf eine Referenz, s. u.	1.0
@serial	Für Attribute wie serialVersionUID und andere, die bei Serialisierung besonders behandelt werden. Javadoc erstellt für so markierte Elemente eine Seite „Serialized Form“.	1.2
@serialData	Typen und Reihenfolge in serialisierter Form	1.2
@serialField	Für jedes ObjectStreamField	1.2
@since	Version der Software, seit der das Element existiert.	1.1
@throws	Exception die aus einem Block geworfen werden können. Synonym für @exception	1.2
{@value}	Zeigt den Wert einer Konstanten	1.4
@version	Version des Elements	1.0

Der Text zu `@param`, `@return`, `@throw` wird üblicherweise nicht mit einem Punkt abgeschlossen.

Gute Beispiele für die Verwendung dieser Tags finden Sie u. a. in den Collection Klassen in java.util. Ich erläutere hier nur eine Auswahl.

Das *@author* Tag kann einmal oder mehrmals in einem Javadoc Block verwendet werden:

1. Gibt es mehrere *@author* Tags, so fügt Javadoc **Author** nach jedem Namen ein Komma und ein Leerzeichen ein.

2. Folgen einem *@author* Tag mehrere Namen, so wird der gesamte Text in das erzeugte Dokument kopiert.

Das *@deprecated* Tag sollte so verwendet werden:

```
/**
 * @deprecated  As of JDK 1.1, replaced by
 *              {@link #setBounds(int,int,int,int)}
 */
```

Der Tag *@link* hat die folgende Formen: `@link package.class#member label`. Diese Form verweist auf die Javadoc-Datei zu einer Klasse bzw. einem Element der Klasse (Konstruktor, Methode, Attribut)

Der Tag *@see* hat eine der folgenden Formen:

- `@see "string"`. Diese Form verweist auf eine Referenz, die nicht als URL zur Verfügung steht. Etwa so

```
@see "The Java Programming Language"
```

- @see label. Diese Form verweist auf eine Referenz, für die eine URL ggf. mit Angabe einer section zur Verfügung. Das Label erscheint im Text, der Link liegt *darunter*.

- @see package.class#member label. Diese Form verweist auf die Javadoc-Datei zu einer Klasse bzw. einem Element der Klasse (Konstruktor, Methode, Attribut).

Die folgenden Tags sind in Planung und werden vielleicht in der Zukunft genutzt (Nach http://www.oracle.com/technetwork/java/javase/documentation/proposed-tags-142378.html, besucht am 09.09.2014):

Tag	Beschreibung
@category	For logically grouping classes, methods, fields (possibly shortened to @cat). Possible syntax: @category name For one interpretation of this tag, see: Feature Request #4085608 in Developer Connection.
@example	For an example source code file (*.java), either in-place or a link to an example. Syntax probably: @example text For more discussion, see: Feature Request #4075480 in Developer Connection.
@tutorial	For a link to a tutorial. Syntax undecided, with several choices. For more discussion, see: Feature Request #4125834 in Developer Connection.
@index	For text to appear in the alphabetic Index. Possible syntax: @index entry[:sub-entry]* For more discussion, see: Feature Request #4034228 in Developer Connection. Eine (erste) Version dieses tags wurde in Java 9 realisiert.
@exclude	For API to be excluded from generation by Javadoc. Programmer would mark a class, interface, constructor, method or field with @exclude. Presence of tag would cause API to be excluded from the generated documentation. Text following tag could explain reason for exclusion, but would be ignored by Javadoc. (Formerly proposed as @hide, but the term „hide“ is more appropriate for run-time dynamic show/hide capability.) For more discussion, see: Feature Request #4058216 in Developer Connection.
@todo	Indicates that work needs to be done on this part of the code. This tag takes a text argument that is a description of the needed work.
@internal	For holding comments that are internal to the company developing the code. This might contain comments about implementation details which might not be relevant to end users who are going to be using the class files. This would enable companies to have an internal version of the generated documentation that would contain more information to help other developers who may be new to the project. This information could be filtered out with a switch on the Javadoc command line when generating the documentation to distribute to clients. Syntax: @internal text For more discussion, see: Feature Request #4102647 in Developer Connection.
@obsolete	To be used when deprecated API is actually removed (if ever). Syntax to be determined.
@threadsafety	Indicates whether a class or method is thread safe or not. Example: @threadsafety This class and its methods are thread safe

Diese Liste stammt wohl ursprünglich aus der Zeit von Java 1.2. Sie können mit der Option -tag auch selbst tags definieren, die dann von javadoc entsprechend verarbeitet werden. Dabei sollten Sie bei der Verwendung der vorgeschlagenen Tags vorsichtig sein.

13.7 Praktische Hinweise

Das Programm javadoc finden Sie nicht im *jre*, sondern im *jdk*. Es ist schließlich ein Werkzeug für Entwickler. Javadoc kann natürlich nur die Informationen finden, auf dies das Programm explizit hingewiesen wird. Um die Verlinkung zu den Java Standardklassen zu ermöglichen bietet Ihnen Eclipse bei Export→Javadoc die Möglichkeit, die entsprechenden Bibliotheken auszuwählen. Sie müssen dazu den Speicherort der jeweiligen API-Dokumentation angeben. Beachten Sie aber, dass Javadoc keine API-Dokumentation in zip-Daten unterstützt. Wollen Sie selbst API-Dokumentation erstellen, die auf die Java-Standard-Klassen verweist, so müssen Sie die entsprechende zip.-Datei entpacken.

Javadoc hat ebenso wie der Java-Compiler viele Optionen. Ich stelle hier einige vor, von denen ich meine, das Ihre Kenntnis besonders hilft. Javadoc hat drei Gruppen von Optionen

Standardoptionen: Stehen für alle *Doclets* zur Verfügung.

Erweiterte Optionen: Beginnen mit *X*.

Doclet Optionen: Auch Doclets können weitere Optionen bieten.

Nicht besonders wichtig, aber praktisch ist die Option „-quiet“: Javadoc läuft mit dieser Option schneller: Dann bekommen Sie nur die Ausgaben der Warnungen und Fehler. Mittels javadoc.exe -help erhalten Sie eine Übersicht der Optionen:

Ich empfehle als Zeichensatz für Java-Projekte UTF-8. Entsprechend sollten Sie javadoc mit den Optionen:

```
-encoding UTF-8 -charset UTF-8 -docencoding UTF-8
```

ausführen, damit auch Sonderzeichen, wie Umlaute ß etc. korrekt dargestellt werden.

Seit Java 8 gibt es für Javadoc die Option -Xdoclint. Sie sorgt dafür, das Javadoc auch html prüft. Diese Option ist per default eingeschaltet. Um Sie auszuschalten verwenden Sie -Xdoclint:none Wenn Sie neu mit Java beginnen, enmpfehle ich, diese Option zu verwenden, so wie es der default ist. Um die Prüfungen genauer zu steuern haben Sie die Möglichkeiten, die Javadoc sonst auch bietet: Für welche Elemente (private, protected, package, public) etc., z. B.

```
-Xdoclint:all/private
-Xdoclint:none
```

Auch Mehrfachnennungen sind möglich:

```
javadoc -Xdoclint:html -Xdoclint:syntax -Xdoclint:accessibility filename
javadoc -Xdoclint:html,syntax,accessibility filename
```

Die vollständigen Informationen bekommen Sie so

```
>javadoc -X
  -Xmaxerrs <number>                Set the maximum number of errors to print
  -Xmaxwarns <number>               Set the maximum number of warnings to print
Provided by standard doclet:
  -Xdocrootparent <url>             Replaces all appearances of @docRoot followed
                                    by /.. in doc comments with <url>
  -Xdoclint                         Enable recommended checks for problems in jav
adoc comments
  -Xdoclint:(all|none|[-]<group>)
        Enable or disable specific checks for problems in javadoc comments,
        where <group> is one of accessibility, html, missing, reference, or syntax.
These options are non-standard and subject to change without notice.
```

13.8 Doclets

Ein *Doclet* ist ein Template, dass das Format der von Javadoc erzeugten HTML-Datei spezifiziert. Es ist in etwa mit einem Stylesheet (.css) zu vergleichen.

Das Standard-Doclet und die von ihm verwendeten Klassen finden sie im SDK unter src/com/sun/javadoc. (zumindest in Java 8.0.20) Unter Doclet.com finden Sie einige weitere Doclets.

Auch *doxygen* verarbeitet .java-Dateien mit javadoc-Kommentaren und erzeugt HTML und LaTeXAusgaben.

13.9 Historische Anmerkungen

Seit Java 1.4 sind die führenden Sterne am Anfang jeder Zeile zwischen Javadoc Start und Javadoc Ende optional.

Mit Java 8 kamen Prüfungen auf Korrektheit der html-Anweisungen hinzu. Diese stehen auch für den Compiler zur Verfügung.

Zeitlich als Vorläufer und von der Leistungsfähigkeit höher kann man Don Knuth' literate programming System CWEB ansehen. CWEB liefert im Kern ein ausführbares Programm, wenn es durch einen C-Compiler umgewandelt wird und ein dvi-Dokument, wenn es mit TeX gesetzt wird. Javadoc Kommentare liefern entsprechend eine HTML-Dokumentation.

13.10 Aufgaben

1. Für welche Zielgruppen verwenden Sie die auf Seite 172 genannten Sichtbarkeiten, um für die jeweiligen Elemente HTML-Dokumentation aus Javadoc-Kommentaren zu generieren?
2. Wie verweisen Sie mit javadoc auf ein gedrucktes Buch?
3. Wie verweisen Sie mit javadoc auf eine Internetressource?
4. Wie überprüfen Sie Ihre HTML-Syntax in Javadoc-Kommentaren auf Korrektheit?

<

Kapitel 14

Ein- und Ausgabe

14.1 Übersicht

Eine ganze Reihe von einfachen Möglichkeiten für Ein- bzw. Ausgabe habe ich schon in früheren Kapiteln verwendet. Ich erwähne aus Kap. 4 z. B. die Klasse *PrintStream* des Klassenattributs *out* der Klasse *System*. Zum Einlesen können sie die einfachen Dialoge aus *JOptionPane*, wie *showInputDialog* verwenden und für Eingaben von der Konsole einen *Scanner*. Nun ist es an der Zeit diese systematisch darzustellen. Wichtig ist für Sie, dass viele Ein- und Ausgaben über das Konzept des Streams realisiert werden. Auf diesem einfachen Konzept basieren viele der in diesem Kapitel behandelten Dinge. In Abschn. 14.3 stelle ich einiges über bereits behandelte Themen zusammen, die später genauer untersucht werden.

Streams können Sie sich als Ströme oder Flüsse vorstellen. Wie Flüsse von der Quelle zur Mündung fließen, stellen Streams eine Verbindung zwischen Ihrem Programm und Quellen oder Senken von Daten dar. Den Inhalt können Sie sich wie Wasser vorstellen und die wirklich interessierenden Inhalte als Fische, die Sie aus dem Fluss fischen wollen bzw. dort wieder aussetzen.

14.2 Lernziele

- Mit Objekten der Klasse *File* umgehen können.
- *Streams*, *Reader*, *Writer* und *Scanner* verwenden können.
- Das Marker-Interface *Serializable* implementieren können und wissen, wann Sie das besser bleiben lassen.

14.3 Ein- und Ausgaben von und auf die Konsole

Ausgaben auf die Konsole kann man ganz einfach mit *System.out.println()* bewirken. Dabei ist *out* ein Klassenattribut der Klasse *PrintStream*. Die Klasse *System* hat noch einen weiteren *PrintStream* als Klassenattribut: *err*. Auf diesen werden z. B. die Ausgaben von *printStackTrace()* geschrieben. Sie erscheinen auf der Konsole in Eclipse z. B. in rot.

Eingaben über die Konsole kann man mittels der Methoden von *InputStream* einlesen: Hierzu gibt es ein Klassenattribut *in* der Klasse *System*. Hierzu gibt es z. B. die elementare Methode *read()*, mit der ein Byte eingelesen werden kann und die Methode *read(byte[] b)*, mit der höchstens so viele Bytes gelesen werden, wie das übergebene Array lang ist.

Wollen Sie aus einer Datei lesen, z. B. um die Eingaben nicht immer wieder machen zu müssen, so können Sie Ihr Java-Programm mit „< Dateiname“ aufrufen. Wollen Sie etwas statt über *System.out* in eine Datei ausgeben, so geht das entsprechend mit „> Dateiname“. Siehe hierzu in Abschn. 14.6 über das Umleiten von Streams.

Aber warum so kompliziert, wenn es auch einfach geht: Beim Einlesen nicht nur von Tastatureingaben hilft die Klasse *Scanner* aus dem Paket *java.util*. Die Klasse *Scanner* verarbeitet zeichenorientierten Input und hat verschiedene Konstruktoren, z. B. für einen *String*, *File*, oder einen *InputStream*. Den *InputStream System.in* werden Sie oft benötigen, ebenso wie den oft benutzten output-Stream *System.out*. Haben wir einen *Scanner*, so können wir den zu scannenden Input systematisch verarbeiten. Aus der API-Dokumentation sehen Sie, das ein *Scanner* ein *Iterator* ist. Genauer: *Scanner* implementiert das Interface *Iterator<String>*. Folgende Methoden der Klasse *Scanner* erwähne ich hier exemplarisch:

hasNext() Diese überladene Operation mit dem Rückgabetyp *boolean* liefert *true* zurück, wenn es weitere Elemente in dem Stream gibt, ggf. auch mit gewissen Eigenschaften.

hasNext(String pattern) Liefert *true*, wenn das nächste Token dem im String spezifizierten Muster entspricht.

hasNextInt Liefert *true*, wenn es noch eine zu scannende ganze Zahl gibt.

hasNext... entsprechend für *BigInteger*, *Float*, *Double* ect.

next Diese Operation liefert das nächste Token als String.

nextInt Diese Operation liefert das nächste Token als ganze Zahl.

next... entsprechend für *BigInteger*, *Double* , *Float* etc.

nextLine Überspringt den zu lesenden Input, liefert den übersprungen Inhalt zurück (ohne etwaige Zeilenendezeichen) und positioniert den Scanner am Anfang der nächsten Zeile.

Die main-Methode von *io.ConsoleIO01.java* zeigt einige Beispiele. Achten Sie darauf, dass das Dezimalkomma den lokalen Gegebenheiten entsprechend erwartet wird. In meiner Eclipse-Installation und der im AIL ist dies ein Komma, im angelsächsischen Bereich meist ein Punkt.

14.4 Dateien (Files)

Den Einstieg in die Arbeit mit Dateien bildet die Klasse *File* aus dem Paket *java.io*. Die Klasse *File* stellt Methoden zur Verfügung, mit denen man sich grundlegende Informationen über das Dateisystem beschaffen kann. Die einzelnen Methoden und Klassenmethoden können Sie sich selbst in der Java-Dokumentation ansehen. Ich erläutere nur einige, die ich in Beispielen verwende. Hier nun also einige Punkte, die ich für den Anfang als nützlich ansehe. Viele weitere Einzelheiten finden Sie in der Java API-Dokumentation.

Die Konstruktoren erhalten als Argument(e) im wesentliche einen relativen oder absoluten Dateinamen oder einen relativen Dateinamen und einen Pfadnamen, entweder String oder als File.

Um sich einen ersten Überblick über das Dateisystem zu verschaffen, gibt es die Klassenmethode *listRoots*:

```
for(File f:File.listRoots()) {
   System.out.println(f);
}
```

Diese liefert eine Ausgabe wie diese:

```
C:\
D:\
E:\
```

Ich verweise aber auf die Klasse *FileSystemView* aus dem Paket *javax.swing.filechooser*. Diese Klasse liefert die im jeweiligen Betriebssystem übliche Darstellung. So hat sie eine Methode *getSystemDisplayName*:

```
FileSystemView fsv = FileSystemView.getFileSystemView();
for(File f:File.listRoots()) {
   System.out.println(fsv.getSystemDisplayName(f));
}
```

Die Klassenmethode *getFileSystemView* liefert ein Objekt der abstrakten Klasse *FileSystemView*. Sie ist also eine Art *Fabrikmethode* (Mehr dazu bei Entwurfsmustern später in dieser Vorlesung und im Verlauf des Studiums). Dies liefert in der gleichen Situation:

```
System (C:)
Volume (D:)
Wechseldatenträger (E:)
```

FileSystemView liefert Ihnen mittels der *Fabrikmethode* jeweils ein Objekt der für das jeweilige Betriebssystem passenden Unterklasse. Das jeweilige Wurzelverzeichnis erhalten Sie mit der Methode *getRoots*. Diese liefert, abhängig vom Betriebssystem, die jeweiligen Root-Partitionen, in Windows etwa *Desktop*. Die Methode *getSystemDisplayName(File f)*, liefert den Namen einer Datei, eines Verzeichnisses etc. so, wie er in der Datei-Ansicht des jeweiligen Betriebssystems erscheint, in Windows also wie im Windows Explorer. Der folgende Code

```
for (File f : fsv.getRoots()) {
   System.out.println(f);
   System.out.println(fsv.getSystemDisplayName(f));
}
```

liefert also unter einem Windows-System so etwas wie

```
C:\Users\Fu\Desktop
Desktop
```

Im Paket *io* im Projekt Programmierbeispiele finden Sie rudimentär einige Einsatzmöglichkeiten der Methoden der Klasse *File*.

Mittels der Klasse *Scanner* kann man auch einfache Operationen mit Dateien vornehmen. Hier ein einfaches Beispiel (*io.FileIO01*):

```
public static void main(String[] args) throws FileNotFoundException {
   Scanner sc= new Scanner(new File("./src/io/ConsoleIO01.java"));
   while(sc.hasNext()){
      System.out.println(sc.next());
   }
}
```

Diese kleine *main-Methode* gibt einfach die Datei *ConsoleIO01.java* auf der Konsole aus.

Mit den Methoden der Klasse *File* können Sie einfach durch das Dateisystem navigieren. Mit der Methode *isDirectory* stellen Sie z. B. fest, dass die Datei ein Verzeichnis ist (oder eben nicht) und können dann rekursiv (siehe Abschn. 5.9) in das Verzeichnis absteigen. Ein Beispiel hierfür zeigt die Klasse *FileExample* im Paket *io*. Weitere Methoden liefern die Eigenschaften der Datei, wie *canExecute* usw. Die Klasse *FileSystemView* bietet weitere Methoden, um die systemspezifischen Eigenschaften einer Datei entsprechend dem gewählten Look & Feel zu bekommen.Weitere hilfreiche Methoden zum Umgang mit Dateien finden Sie in der Utility-Klasse *java.nio.file.Files*.

14.5 Eingabe von Dateinamen und Auswahl von Dateien

Das Einlesen von Eingaben, z. B. von Dateinamen, kann mittels vorhandener Java-Klassen und -Methoden auch anders und auch eleganter gestaltet werden. So bietet die Klasse *JOptionPane* aus dem Paket *javax.swing* u. a. die in folgender Tabelle aufgeführten vier Klassenmethoden.

showConfirmDialog	Fragt nach Bestätigung oder Ablehnung, wie ja/nein/abbrechen.
showInputDialog	Verlangt eine Eingabe.
showMessageDialog	Gibt eine Information für den Benutzer aus.
showOptionDialog	Eine Zusammenfassung der drei vorstehenden Dialoge.

Mit Code wie

```
String fileName = JOptionPane.showInputDialog("Geben Sie bitte einen
                                                vollständigen Dateinamen an!");
```

können Sie einfach einen *String* von der Tastatur einlesen. Der vom Benutzer eingegebene String wird von der Methode als Rückgabe geliefert.

Diese Art der Auswahl eines Dateinamens ist aber immer noch sehr mühselig und fehleranfällig. Einfacher geht es mit einem *JFileChooser* aus dem Paket *javax.swing*. Diese Klasse hat u. a. die Methoden *showOpenDialog* und *showSaveDialog*. Hier etwas Code dazu:

```
 20   JFileChooser jfc = new JFileChooser();
 30   jfc.setFileSelectionMode(JFileChooser.FILES_AND_DIRECTORIES);
 40   if(jfc.showOpenDialog(null)==JFileChooser.APPROVE_OPTION) {
 50      JOptionPane.showMessageDialog(null,
 60                                    "Die Datei :" +
 70                                    jfc.getSelectedFile() +
 80                                    " wurde ausgewählt");
 90   } else {
100              JOptionPane.showMessageDialog(null, "Ausgewahl abgebrochen.");
110   }
```

Der Dialog zur Auswahl einer Datei sieht etwa für Windows-Benutzer gewöhnungsbedürftig aus; er hat nicht das übliche Aussehen eines solchen Windows-Dialoges. Das Standard-Aussehen („look and feel“) in Swing ist *Metal*. Dieses „Look & Feel“ bekommen Sie so (Fehlerbehandlung weggelassen):

```
 10   UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
```

vor obigem Code. Nun zur Erläuterung dieser Methoden:

1. *UIManager* ist eine Utility-Klasse für den Umgang mit Look&Feel (Zeile 10). Die Klassenmethode *getSystemLookAndFeelClassName* liefert den Namen der Klasse des default-Look&Feels des jeweiligen Systems. Die Klassenmethode *setLookAndFeel* setzt das Look&Feel dann entsprechend. Tun Sie so etwas nicht, so verwendet Java das Metal Look&Feel.

2. In Zeile 20 wird einfach ein neuer *JFileChooser* erzeugt.

3. In Zeile 30 wird bestimmt, dass sowohl Dateien als auch Verzeichnisse ausgewählt werden können. Für die oft benötigten Auswahlmöglichkeiten gibt es Konstanten, die den jeweilige FileSelectionMode repräsentieren.

4. In Zeile 40 wird abgefragt, ob die Auswahl bestätigt wurde. Ist das der Fall, so wird

5. in Zeile 50ff ein Message-Dialog angezeigt, andernfalls

6. in Zeile 100 ein entsprechender anderer Message-Dialog angezeigt.

14.6 Streams

Von *Streams* war nun oft genug die Rede, nun ist es an der Zeit dieses Thema genauer zu behandeln. Für *Streams* gibt es in Java zwei Konzepte:

1. Datenquellen und -Senken, die in diesem Kapitel behandelt werden. Sie finden die Klassen z. B. in den Paketen *java.io* und *java.nio*.

2. Hüllen um Daten, von *Collections* bis zu Daten, die durch eine Funktion generiert werden. Die entsprechenden Interfaces finden Sie im Paket *java.util.stream*. Ich behandele sie in Kap. 16.

Die Streams (Datenströme) aus *java.io* und *java.nio* sind ein Mechanismus in Java um Daten aus einer Quelle zu heben oder in einer Senke verschwinden zu lassen. Die Wurzeln dieser Stream-Hierarchie bilden die abstrakten Klassen *InputStream* und *OutputStream*. Beiden gemeinsam ist, dass sie mit Bytes arbeiten. Abbildungen 14.1 und 14.2 zeigen einen (zumindestens am 30.04.2011

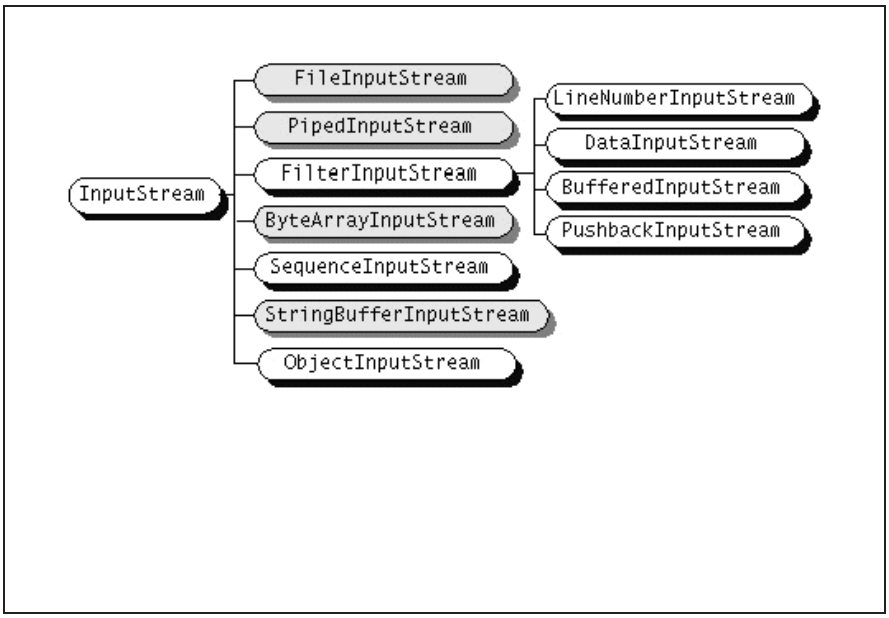


Abb. 14.1: Java Input Streams

aktuellen) Ausschnitt aus der Java Stream-Hierarchie. Sie finden hier Streams zum Bearbeiten verschiedener Arten von Eingaben. Die Klasse *ObjectInputStream* werden Sie in Abschn.14.8 kennenlernen.

Bereits in Kap. 5 haben Sie ein Objekt der Klasse *InputStream* kennengelernt: Das Klassenattribut *in* der Klasse *System* ist ein Objekt dieser Klasse. Ändern Sie das nicht, so ist *in* mit der Tastatur verbunden. Sie können dies aber ändern. So können Sie Ihre Tastatureingaben in einer Datei *ConsoleIO.input* speichern, z. B. diese:

```
42
Hallo
3,1415
3,14159265358979323846264338327 95
```

Dann können Sie z. B. die Klasse *io.ConsolIO01* ausführen und beim Aufruf *<ConsoleIO.input* nach dem Namen der Klasse angaben. Dann liest der Scanner die Eingaben aus der Datei und Sie erhalten ohne weitere Interaktion die Ausgabe:

```
Geben Sie bitte eine ganze Zahl ein!
42
Geben Sie bitte einen String ein!
Hallo
```

```
Geben Sie bitte eine Float Zahl ein (Dezimalkomma!)
3.1415
Geben Sie bitte eine Double Zahl ein (Dezimalkomma!
3.141592653589793
```

Die Klasse *System* ermöglicht es aber auch, den *InputStream* im Programm zu ändern. Dazu gibt es die Klassenmethode *setIn*. Übergeben Sie als Parameter

```
fis = new FileInputStream(new File("./src/io/ConsoleIO.input"));
```

so erhalten das gleiche Ergebnis wie oben. Für zeichenorientierte Eingaben ermöglicht die Klasse *Scanner* dann eine einfache Verarbeitung. Die Klasse Scanner nimmt Ihnen für zeichenbasierten Input viel Arbeit ab! Sie können einen *Scanner* auch für einen *BufferedInputStream* erzeugen. Da letzterer einen Konstruktor für einen *InputStream* hat, können Sie mittels eines *Scanner*s auch aus einer Textdatei lesen.

Wollen Sie Eingaben als Byte-Ströme verarbeiten, so müssen Sie anders vorgehen. Das können Sie direkt tun. Die Streamklassen bieten rudimentäre Methoden dafür. So können Sie für *FileInputstream*s mit *available* die ungefähre Anzahl noch verfügbarer Bytes abfragen und mit einer der drei überladenen *read*-Methoden die Bytes einlesen.

Etwas komfortabler geht es mit den *Reader*-Objekten. Hier können Sie das Zeichensystem und einige weitere Dinge berücksichtigen. So hat etwa *InputStreamReader* einen Konstruktor, der ein *CharacterSet* als Parameter bekommt und eine Methode *getEncoding*, die Namen des Zeichensatzes liefert. Ein einfaches Beispiel für die Verwendung eines *InputStreamReader*s zeigt das Beispiel *io.FileReadExample03*.

Außer der Klasse Scanner gibt es weitere Klassen zum Einlesen von InputStreams. Die abstrakte Klasse *Reader* bildet die Wurzel eine Hierarchie von spezialisierten Readern. Zu den direkten Unterklassen gehört die Klasse *BufferedReader*. Ein *BufferedReader* wird auf Basis eines *Readers* erstellt. Die Default-Größe des Buffers ist zur Zeit 8192. Es gibt aber auch einen Konstruktor, der eine Buffergrößer als Parameter erhält.

Streams kommen zunächst einmal als Streams von Bytes. Eine der Brücken zwischen Bytes und einer Darstellung als Character bildet die Klasse *InputStreamReader*. *InputStreamReader* werden häufig mit *BufferedReader* verwendet.

Ganz entsprechend gibt es Writer für die Ausgaben.

Sowohl für die Ein- als auch für die Ausgabe wird hier ein Prinzip verwendet, dass als *Decorator Pattern* populär geworden ist. Siehe hierzu [GHJV95] und das Kapitel über Entwurfsmuster, Abschn 23.11.

Nun zu den Output-Streams. Eine der ersten Java Klassen, die Sie gesehen haben, war *PrintStream*. Sie ist eine Unterklasse der Unterklasse *FilterOutputStream* von *OutputStream*, die Sie als Klasse des Klassenattributs *out* von *System* kennengelernt haben. Ein anderes Objekt dieser Art ist das Klassenattribut *err* der Klasse *System*. Auch diese können Sie wie die Eingabe-Ströme mit Hilfe des Betriebssystems umleiten. Nun heißt es „>“ statt „<“. Oder Sie verwenden die Methode *setOut* bzw. *setErr*.

Wollen Sie die Fehlermeldungen aus den Ausgaben für Anwender heraushalten, so können Sie den Standard-Fehler-Strom *System.err* in eine Datei umleiten. Hierzu verwenden Sie analog zu oben die Klassenmethode *setErr*.

14.7 Channels

Für Input- und Output-Streams gibt es Channels, die die Programmierung vereinfachen und effizient implementiert sind. Sie können sich *Channels* als „Kanäle“ vorstellen, durch die *Streams* (Ströme, Flüsse) einfacher und schneller fließen können.

Ich zeige ihren Einsatz hier an einigen Möglichkeiten, wie Sie in Java eine Datei kopieren können (CopyCat01–CopyCat04). In allen Fällen unterstelle ich, dass ich die Dateinamen von Quelle und Ziel als Strings *inputName* bzw. *outputName* vorliegen habe. Mit den Mitteln der Streams geht dies z. B. so:

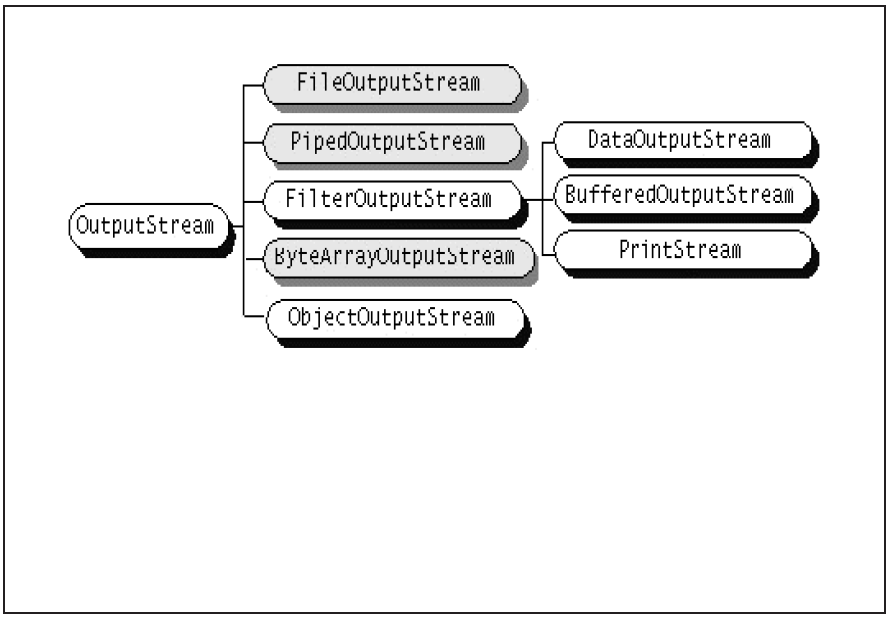


Abb. 14.2: Java Output Streams

```
   try (BufferedInputStream bis = new BufferedInputStream(
                                  new FileInputStream(
                                  new File(inputName)),BUFFERSIZE);
        BufferedOutputStream bos = new BufferedOutputStream(
                                   new FileOutputStream(
                                   new File(outputName)),BUFFERSIZE)){
        while(bis.available()>0) {
              bos.write(bis.read());
        }
   } catch (IOException e) {
         e.printStackTrace();
}
```

Bitte gewöhnen Sie sich an, für *AutoCloseable* Ressourcen konsequent try-with-resources zu verwenden. Die Konstante BUFFERSIZE sei im jeweiligen Gültigkeitsbereich definiert. Unter Verwendung von *Channels* können Sie das etwas vereinfachen:

```
   try (FileChannel in = new FileInputStream(inputName).getChannel();
        FileChannel out= new FileOutputStream(
                         new File(outputName)).getChannel()){
        in.transferTo(0, in.size(), out);
   } catch (IOException e) {
         e.printStackTrace();
   }
```

Ab Java 7 geht es auch wie folgt, wenn ich *java.nio.file.StandardCopyOption.** statisch importiere.

```
   try {
      Files.copy(Paths.get(inputName), Paths.get(outputName),
                 COPY_ATTRIBUTES, REPLACE_EXISTING);
   } catch (IOException e) {
      e.printStackTrace();
}
```

Eine weitere Möglichkeit ist:

```
   try {
      Runtime.getRuntime().exec("copy" + inputName + " " + outputName);
   } catch (IOException e) {
e.printStackTrace();
}
```

Aber so machen Sie das besser nicht. Das gibt in einer Klausur nicht einmal einen Mutpunkt. Damit kennen Sie nun mehrere Möglichkeiten Dateien einzulesen und auszugeben.

14.8 Serialisierung

Als Serialisierung in Java 1.1 eingeführt wurde, waren die Gefahren bekannt:

> When serialization was added to Java in 1997, it was known to be somewhat risky. The approach had been tried in a research language (Modula-3) but never in a production language. While the promise of distributed objects with little effort on the part of the programmer was appealing, the price was invisible constructors and blurred lines between API and implementation, with the potential for problems with correctness, performance, security and maintenance. Proponents believed the benefits outweighted the risk, but history has shown otherwise. [Blo18]

> Java deserialization is a clear and present danger as it is widely used both directly by applications and indirectly by Java subsystems such as RMI (Remote Method Invocation), JMX (Java Management Extension), and JMS (Java Messaging System). Deserialization of untrusted streams can result in remote code execution (RCE), denial-of-service (DoS, and a range of other exploits. Application can be vulnerable to these attacks even if they did nothing wrong.[Sea17]

Mit *Serialisierung* wird in Java ein einfaches Verfahren zum „Wegschreiben“ (serialisieren) und wieder Einlesen von Objekten (deserialisieren) bezeichnet. Klassen, die dies ermöglichen sollen, müssen das Interface *Serializable* implementieren. Dies ist ein sogenanntes Marker-Interface: Es kennzeichnet nur die Serialisierbarkeit und enthält keine Methoden.

Wann immer eine Klasse serialisierbar ist, so erwartet der Compiler ein Klassenattribut *serialVersionUID*. Dies hat den Typ *long*. Sein Wert soll die Version der Klasse charakterisieren, mit der ein Objekt geschrieben wird. Dadurch kann beim Wiedereinlesen überprüft werden, ob das Objekt von der gleichen Version der Klasse ist. Ist dies nicht der Fall, so wird eine *InvalidClassException* geworfen. Andere Gründe für diese Exception können unbekannte Datentypen in der Klasse sein, oder dass es keinen öffentlichen default-Konstruktor gibt.

Definieren Sie keine *serialversionUID*, so wird zur Laufzeit eine mittels *serialver.exe* erzeugt. Dadurch geht die Kompatibilität mit zukünftigen Versionen bei praktisch jeder Änderung verloren.

Der Code für so etwas ist ganz einfach, hier einige Code-Teile aus *io.SaveCounter*, um die Erläuterungen zu illustrieren:

```
CounterSerialized ctVorher = new CounterSerialized();
ctVorher.increment();
try (ObjectOutputStream os =
     new ObjectOutputStream(new FileOutputStream("test.ser"))){
     System.out.println("Counter vor Serialisierung  :"+ctVorher);
     os.writeObject(ctVorher);
} catch (IOException e) {
   e.printStackTrace();
}
```

Zum Serialisieren brauchen Sie einen *ObjectOutputStream*. Diesem übergeben Sie im Konstruktor z. B. einen *FileOutputStream*. Geschrieben wird das Objekt mit der Methode *writeObject*.

Alle Attributwerte des Objekts werden mit serialisiert bzw. deserialisiert, wenn sie davon nicht gezielt ausgenommen werden. Nur Attribute, die mit dem Schlüsselwort *transient* gekennzeichnet sind, werden von der Serialisierung ausgenommen.

Soweit die einfachen Grundlagen. Nun gibt es hier noch einige weitere Dinge zu beachten.

1. Sind alle Attribute einer Klasse serialisierbar, so kann jedes Objekt der Klasse mittels der Methode *writeObject* serialisiert werden. Alle primitiven Typen sind serialisierbar, da die Wrapper-Klassen serialisierbar sind, ebenso *String*.

2. Ein Attribut, das als *transient* gekennzeichnet sind, wird nicht mit serialisiert.

3. Schwieriger kann es werden, wenn eine serialisierbare Klasse Attribute von Typen deklariert, die nicht serialisierbar sind. Sind diese Attribute als *transient* gekennzeichnet, so werden sie eben nicht mit serialisiert. Beim Deserialisieren müssen Sie dann vom default-Konstruktor initialisiert werden. Daraus ergibt sich Folgendes: eine serialisierbare Klasse muss in diesem Fall einen zugänglichen Konstruktor ohne Parameter haben.

4. Eine Unterklasse einer serialisierbaren Klasse ist serialisierbar.

5. Die Umkehrung gilt nicht: Eine Unterklasse einer nicht-serialisierbaren Klasse kann serialisierbar gemacht werden, wenn die Oberklasse(n) sichtbare Default-Konstruktoren haben. Gegebenenfalls müssen die Methoden

   ```
   private void writeObject(java.io.ObjectOutputStream out)
        throws IOException
    private void readObject(java.io.ObjectInputStream in)
        throws IOException, ClassNotFoundException;
    private void readObjectNoData()
        throws ObjectStreamException;
   ```

   mit exakt diesen Signaturen implementiert werden.

In *writeObject* müssen Sie zuerst die Methode *defaultWriteObject* aufrufen.

Die andere Richtung, die sogenannte *Deserialisierung*, verläuft ganz analog. Nun brauchen Sie einen *ObjectInputStream*, der mit einem *FileInputStream* (s. u.) verbunden wird. Eingelesen wird das zuvor serialisierte Objekt mittels der Methode *readObject*. Der Cast auf die jeweilige Klasse muss hier sein, da der Rückgabetyp von *readObject* eben *Object* ist. Auch dieser Code ist ganz einfach:

```
try (ObjectInputStream is = new ObjectInputStream(new FileInputStream("test.ser"))){
     CounterSerialized ctNachher = (CounterSerialized) is.readObject();
     System.out.println("Counter nach Serialisierung :"+ctNachher);
} catch (ClassNotFoundException e) {
   e.printStackTrace();
} catch (IOException e) {
   e.printStackTrace();
}
```

Schreiben Sie Ihre eigene *readObject*-Methode, so rufen Sie zunächst die Methode *defaultReadObject* auf. Anschließend lesen Sie den *ObjectInputStream* so ein, wie Sie ihn als *ObjectOutputStream* geschrieben haben. Achten Sie darauf, die Elemente in genau der Reihenfolge wieder zu lesen, wie Sie sie geschrieben haben, sonst riskieren Sie eine *OptionalDataException*.

Bemerkung 14.8.1 (Serialisierungsfilter)
Seit Java 9 gibt es Deserialisierungsfilter, mit denen Sie den ObjectInputStream überprüfen können. Dazu mehr in einer späteren Auflage. ◄

Sind die Regeln bei der Serialisierung beachtet worden, so sollte hier alles funktionieren.

Auch Arrays sind nach dem Ende von Abschn. 5.10 serialisierbar.

Eine Klasse serialisierbar zu machen hat Konsequenzen: Die serialisierte Form (als Byte-Stream) der Klasse wird Bestandteil des APIs der Klasse und Sie müssen es ggf. „auf ewig“ unterstützen. Es ist also wichtig, sich zu überlegen, welche Persistenz-Mechanismen Sie einsetzen wollen. So sollten (nicht-statische) innere Klassen *Serializable* nicht implementieren. Diese Klassen haben vom Compiler erzeugte *synthetische Attribute*. Wie diese Attribute zur umschließenden Klasse gehören ist nicht spezifiziert. Daher ist die default-Serialisierung für (nicht-statische) innere Klassen nicht wohldefiniert. Leider fehlt ein etabliertes deutsches Wort für das Gegenteil. Der englische Begriff ist „ill defined“.

*Enum*s sind serialisierbar, aber nicht deserialisierbar, siehe hierzu Abschn. 23.3.

Serialisierung kann sehr gefährlich sein, siehe [Blo18], Kap. 12, [Svo16]. Hier ein Beispiel von Wouter Coekart (Siehe https://gist.github.com/coekie/a27cc406fc9f3dc7a70d):

```
/**
 * billion-laughs-style DoS for java serialization.
 * {@linkplain https://gist.github.com/coekie/a27cc406fc9f3dc7a70d},
 * last visited at 04/30/2019.
 * @author Wouter Coekarts
 */
public class SerialDOS {
  public static void main(String[] args) throws Exception {
    deserialize(payload());
  }

  static Object deserialize(byte[] bytes) throws Exception {
    return new ObjectInputStream(new ByteArrayInputStream(bytes)).readObject();
  }

  static byte[] payload() throws IOException {
    Set<Object> root = new HashSet<>();
    Set<Object> s1 = root;
    Set<Object> s2 = new HashSet<>();
    for (int i = 0; i < 100; i++) {
      Set<Object> t1 = new HashSet<>();
      Set<Object> t2 = new HashSet<>();
      t1.add("foo"); // make it not equal to t2
      s1.add(t1);
      s1.add(t2);
      s2.add(t1);
      s2.add(t2);
      s1 = t1;
      s2 = t2;
    }
    return serialize(root);
  }

  static byte[] serialize(Object o) throws IOException {
    try (ByteArrayOutputStream ba = new ByteArrayOutputStream();
         ObjectOutputStream oos = new ObjectOutputStream(ba)) {
      oos.writeObject(o);
      oos.close();
      return ba.toByteArray();
    }
```

```
  }
}
```

Zum Umgang mit Serialisierung und Deserialisierung gelten folgenden Empfehlungen ([Blo18]):

1. Es gibt keinen Grund Java Serialisierung in neuen Anwendungen zu benutzen. Es gibt bessere (einfachere, performantere, gut bekannte) Alternativen wie die cross-platform structured data representations. Hier sind JSON, Protocol Buffers.

2. Sind Sie gezwungen Serialisierung zu verwenden, so deserialisieren Sie bitte keine nicht-vertrauenswürdigen Daten (http://www.oracle.com/technetwork/java/seccodeguide-139067.html#8: „**Note: Deserialization of untrusted data is inherently dangerous and should be avoided.**“

3. Wenn Sie Daten deserialisieren müssen, der Integrität Sie nicht gewährleisten können, verwenden Sie *java.io.ObjectInputFilter*. Dies ist ein *funktionales Interface*.

4. Wenn Prüfungen bei der Erzeugung von Objekten im Konstruktor erfolgen, die zu einer Exception führen können, so müssen Sie dafür sorgen, dass diese auch bei Deserialisierung vorgenommen werden. Sie müssen also *readObject* schreiben. In dieser Methode rufen Sie zuerst *defaultReadObject* aufrufen.

5. Wenn die logische Struktur der Klasse von der physischen abweicht, sollten Sie eine angepasste serialisierte Form schreiben (custom serialized form). Das Standardbeispiel hierfür ist Liste, die als eine verkettetete Liste implementiert ist. Die logische Struktur ist hier eine Folge mit Positionen. Die physische Struktur ist eine Verkettung mit einer geschachtelten statischen Klasse (*Entry*) mit je einem Listen-Element und Referenzen auf Vorgänger und Nachfolger. In Abschn. sec:genclass auf S. 238 finden Sie ein Beispiel einer solchen Klasse. Deklarieren Sie *Entry* als Serializable, so wird diese Struktur bei Serialisierung offengelegt. Besser ist es einfach die Größe der Liste und die Elemente in ihrer Reihenfolge zu serialisieren (*writeObject*.

6. Eine andere Möglichkeit ist ein Serialization Proxy: Sie schreiben eine statische geschachtelte Klasse namens z. B. *SerializationProxy* mit den Eigenschaften Ihrer Klasse, die Sie serialisieren lassen wollen. Außerdem schreiben Sie die Methoden *writeReplace* und *readObject*. Erstere liefert ein Objekt der Klasse *SerializationProxy* zurück. Letztere verhindert „nur“ Deserialisierungsattacken:

   ```
   private void readObject(java.io.ObjectInputStream stream)
       throws java.io.InvalidObjectException {
       throw new java.io.InvalidObjectException("Proxy required");
   }
   ```

   Ein gutes Beispiel für diese Variante finden sie in der Klasse *EnumSet*.

14.9 Historische Anmerkungen

In Java 7 gibt es das neue Interface AutoCloseable. Dies ermöglicht es, den Code für Streams und Ähnliches sehr viel stromlinienförmiger zu gestalten (siehe die Diskussion in Kap. 12). Darüberhinaus wird eine gefährliche Lücke geschlosssen: Tritt in einem finally-Block eine gleiche Exception auf, wie im try-Block so geht überlagert diese die ursprüngliche im try-Block. Die ursprüngliche Exception geht also de facto verloren.

Die Metapher mit den Fischen aus Abschn. 14.1 verdanke ich Michael Eckard.

14.10 Aufgaben

1. Was versteht man unter *Serialisierung* bzw. *Deserialisierung*?
2. Wann wird eine *InvalidClassException* geworfen?
3. Wie kann es kommen, dass ein Objekt einer Klasse beim Versuch der Einlesens (der Deserialisierung) unbekannte Datentypen enthält?
4. Welchen Nutzen bringt das Interface AutoCloseable?
5. Schreiben Sie bitte Code, der eine Datei kopiert! Unterscheiden Sie die Exceptions und reagieren Sie benutzerfreundlich.

Kapitel 15

Parametrisierte Klassen und Interfaces

15.1 Übersicht

In Java bekommen Sie mit der Klassenbibliothek viele Klassen und Interfaces mit einem Typ-Parameter. Diese Klassen oder Interfaces können Sie verwenden, wenn Sie in Deklarationen für den Typ-Parameter den von Ihnen benötigten Referenztyp einsetzen. Seit langem bietet Java Container-Klassen und -Interfaces in *java.util* und anderen Paketen. Diese leisten genau das, was der Name erwarten lässt: Sie bieten verschiedenartige Möglichkeiten Objekte in einem Behälter (Container) für weitere Verarbeitungen zu organisieren. Schon in *java.lang* finden Sie die Interfaces *Comparable* und *Iterable* mit Typ-Parameter. In diesem Paket finden Sie auch die grundlegende generische Klasse *Class*. Viele weitere wichtige Eigenschaften von Java sind ebenfalls in generischen Interfaces und Klassen definiert, wie in den Paketen *java.util.stream* oder *java.util.function*.

Damit das typsicher ist, sind diese Klassen und Interfaces *generische* (siehe Kap. 18). Sie haben einen oder mehrere Typparameter, die in spitzen Klammern angegeben werden, etwa *List<E>*, *ArrayList<E>*, *Map<K,V>* etc.

Ganz analog gibt es generische Methoden, die mit einem Typparameter deklariert sind, wie z. B. *<T> sort(List<T> list)* u. a. in den Utility-Klassen *Arrays* und *Collections*.

Um diese generischen Elemente — Klassen oder Interfaces — zu verwenden, setzen Sie an der Stelle des Typparameters den von Ihnen benötigten Typ ein, also die jeweilige Klasse oder das jeweilige Interface. Eine mit einem solchen „konkreten“ Typ parametrisiertes generisches Element nennt man parametrisierte Klasse bzw. parametrisierte Methode.

Ich beschreibe in diesem Kapitel einige wichtige Klassen, Schnittstellen und Methoden dieser Art. Ich folge hier dem Prinzip „Erst konsumieren, dann produzieren“. Ich zeigen Ihnen also, wie Sie diese Elemente verwenden, die einen Typ-Parameter benötigen. Wie Sie selber solche Klassen schreiben und worauf dabei zu achten ist, werden Sie in Kap. 18 lernen.

15.2 Lernziele

- Für die jeweilige Aufgabe eine geeignete parametrisierte Klasse odre ein parametrisiertes Interface auswählen können.
- Parametrisierte Klassen verwenden können.
- Generische Methoden verwenden können.
- Assoziationen mittels Container-Klassen implementieren können.

15.3 Parametrisierte Methoden

In den Utility-Klassen *Collections*, *Arrays* und *Objects* finden Sie eine Reihe nützlicher Methoden. Ich nenne hier als erste Beispiele mit ihrer vollständigen Signatur

```
static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)
```

und

```
static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)
```

aus der Klasse *Collections*.

Die erstgenannte sucht aus einer sortierten Liste von vergleichbaren Objekten das erste Objekt mit dem angegebenen Schlüssel (*key*) und liefert dessen Position in der Liste zurück. Die andere leistet das gleiche für eine beliebige Liste, die mittels eines *Comparator*s sortiert ist.

Aus der Utility-Klasse *Arrays* nenne ich

```
public static <T extends Comparable<? super T>> void parallelSort(T[] a)
```

und

```
static <T> void parallelSort(T[] a, Comparator<? super T> cmp)
```

Die Laufzeit dieser Methoden skaliert weitgehend linear mit der Anzahl Prozessoren, die Ihnen in Ihrem Rechner zur Verfügung stehen. Sie stehen Ihnen auch in vielen anderen Situationen zur Verfügung (siehe hierzu das Ende dieses Abschnitts, den Abschnitt 15.4 und Kap. 17.

Aus der Utility-Klasse *Objects* erwähne ich

```
static <T> T requireNonNull(T obj, Supplier<String> messageSupplier)
```

Die Deklarationen dieser Methoden mögen auf den ersten Blick lang und kompliziert erscheinen. Die Ihnen noch nicht bekannten Elemente sind das „?“ (Wildcard) sowie in diesem Kontext die Schlüsselworte *extends* und *super*. Auch der Typparameter <T> ist noch neu für Sie.

Hier nun die Erläuterungen:

- Für den Typparameter setzen Sie den Typ ein, den Sie benötigen.
- Die Formulierung *<? super T>* beim Interface *Comparable* bedeutet Folgendes: „?“ ist eine Wildcard. An ihre Stelle kann jeder Typ treten, der allgemeiner als (der für) T (eingesetzte) ist. Allgemeiner sind dies Oberklassen und die implementierten Interfaces. Diese Formulierung ist notwendig, damit die Methode auch mit Objekten von Klassen umgehen kann, bei denen das Interface *Comparable* bereits in einer Oberklasse implementiert ist.

Beispiel 15.3.1 (Comparable<? super T>)
In Abb. 15.1 implementiert die Klasse *Waggon* das Interface *Comparable*. Aufgrund der Definition der Methode *binarySearch* funktioniert sie auch für *FrachtWaggon* oder *PersonenWaggon*. ◀

Die Wildcard „?“ taucht aber auch gleich am Anfang der Parameterliste von *binarySearch* auf: *? extends Comparable...*. Durch diese Deklaration kann die Methode mit beliebigen Objekten umgehen, die *Comparable<...>* implementieren, also etwa Objekten weiterer Unterklassen von *Waggon* oder *PersonenWaggon*.

Die Methode *binarySearch* ist überladen. Es gibt auch eine Variante mit einem Comparator. Diesen werden Sie oft mittels einer anonymen Klasse oder eines λ-Ausdrucks implementieren.

Die analoge Methode in der Klasse *Arrays* ist noch weiter überladen: Außer den generischen Varianten für *Comparable* Objekte und andere mittels *Comparator* gibt es Varianten für alle primitiven Datentypen und Ausschnitte des Arrays, z. B. diese:

```
static int  binarySearch(byte[] a, byte key)
static int  binarySearch(char[] a, char key)
static int  binarySearch(double[] a, int fromIndex, int toIndex, double key)
```

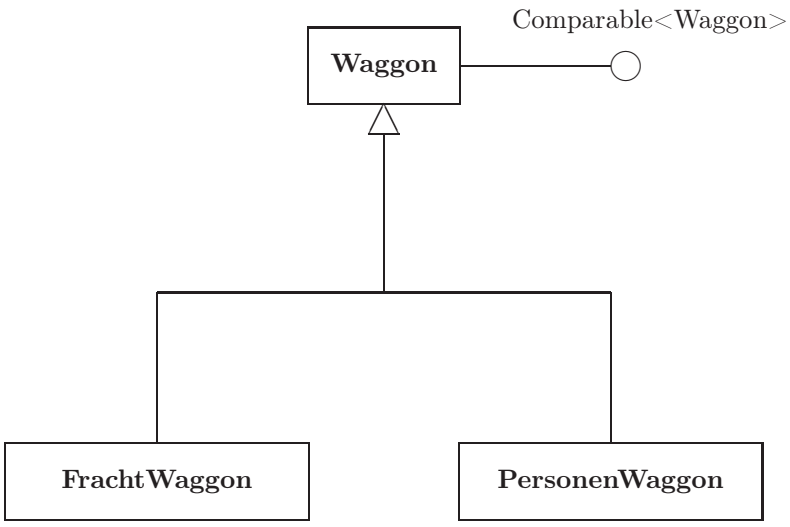


Abb. 15.1: Waggon-Hierarchie

Die Methode *binarySearch* erwartet eine sortierte Liste oder ein sortiertes Array. Sortieren können Sie viele Container mittels der Methode *sort* aus der *Klasse* Collections. Es gibt zwar keine Methode *sort*, die eine beliebige *Collection* sortiert. Aber das Interface *Collection* bietet die Methode *toArray*.

Die Methoden *sort* und *parallelSort* in der Klasse *Arrays* und *Collections* folgen genau dem bei *binarySearch* beschriebenen Muster. Es gibt überladene Varianten für *Comnparable* und *Comparator* und in *Arrays* auch für die primitiven Datentypen. Für Arrays gibt es in Utility-Klasse *Arrays parallelSort*-Methoden. Diese verwenden das Fork-Join Framework.

Spätestens im Laufe Ihrer weiteren Beschäftigung mit Programmierung werden Sie feststellen, dass auch andere Methoden dieser Utility-Klassen sehr nützlich sein können.

15.4 Parametrisierte Klassen

Als erstes generisches Element wird Ihnen wahrscheinlich das Interface *List<E>* aufgefallen sein. Dieses Interface stellt die Basis für alle Listen in Java dar. Eine Liste ist eine Datenstruktur mit den in der folgenden Definition genannten Eigenschaften.

Definition 15.4.1 (Liste)
Eine Liste ist eine Collection, die durch die folgenden Operationen in Abb. 15.2 beschrieben wird. Die einzelnen Operationen bedeuten dabei folgendes:

insert Einfügen eines Elements.

lookup Erhält ein Objekt als Parameter und liefert einen „Zeiger“ auf das (erste) Element der Liste, das das gleich diesem Objekt ist bzw. den oder den Index des Objekts oder `null`, wenn das Objekt nicht in der Liste enthalten ist.

nth erhält eine natürliche Zahl und liefert eine „Referenz“ auf das n-te Element der Liste oder `null`, wenn die Liste weniger als n Elemente enthält.

delete Erhält ein Objekt als Parameter und löscht die Elemente der Liste, die das Objekt enthalten.

◀

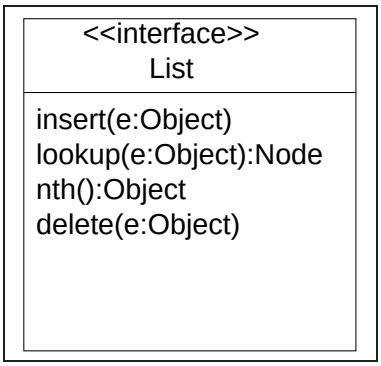


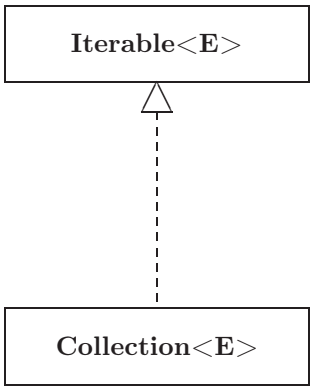
Abb. 15.2: Verkettete Liste - Schnittstelle

Je nach konkreter Aufgabe der jeweiligen Liste können weitere Methoden hinzukommen. Auch die Signatur kann je nach Bedarf anders gestaltet werden. So kann über einen Fehler über eine Exception oder über einen Rückgabewert informiert werden.

Im Interface *List* entsprechen den oben genannten die Methoden im folgenden Codeausschnitt:

```
public interface List<E> extends Collection<E>{
   ...
   boolean add(E e);
   void add(int index, E element);
   int indexOf(Object o);
   E get(int index);
   E remove(int index);
   boolean remove(Object o);
   ...
}
```

Das Konzept der Java-Container erläutere ich beginnend mit dem Interface *Collection<E>*.



Dieses Interface erweitert das Interface *Iterable<E>*. Letzteres hat drei Methoden:

1. *iterator()*, die einen *Iterator<E>* liefert. Dieses Konzept wird also in allen Java-Containern verwendet. Das Grundprinzip besteht darin, die Fähigkeit zum Durchlaufen eines Containers aus dem Container herauszulösen. Diese steckt im *Iterator*. Diese Interface deklariert zwei Methoden und eine weitere optionale:

```
boolean hasNext();
E next();
void remove();  //Optional
```

Die folgenden beiden Methoden sind bereits als default-Methoden in *Iterable* implementiert.

2. *forEach(Consumer<? super T> action)*: Dieser Methode wird eine Aktion übergeben, die auf jedes Element des *Iterable*s angewandt wird.

3. *spliterator()*: Das Interface *Spliterator* beschreibt Objekte, die eine Quelle traversieren und zerlegen können, z. B. für anschließende parallele Verarbeitung.

Jeder *Collection* können Sie als Datenquelle eines *Stream*s für Ihren Code verwenden. Dies leisten die default-Methoden (hier die Signaturen):

1. default Stream<E> stream()

2. default Stream<E> parallelStream()

Das Interface *Collection* hat einige Subinterfaces, z. B. *List*, *Deque*, *Queue* und *Set*.

Außerdem gibt es einige nützliche abstrakte implementierende Klassen, die Sie bei Bedarf als Basis für eigene Implementierungen nutzen können, z. B. *AbstractList*, *AbstractSet* uvam.

Details zu den Implementierungsmöglichkeiten für diese Interfaces lernen Sie in Veranstaltungen zum Thema *Algorithmen und Datenstrukturen*. Hier skizziere ich nur die Grundprinzipien für zwei Implementierungen des Interface *List*.

Eine wichtige Möglichkeit zur Implementierung einer Liste ist ein *Array*. Die Listenelemente werden in einem Array gespeichert. Das erste Element kommt an die Stelle 0, das zweite an die Stelle 1 usw. Ist das Array voll, so wird es vergrößert. Wird ein Element gelöscht, so werden die rechts davon stehenden Elemente um eine Position nach links verschoben. Wird eines zwischen anderen eingefügt, so werden die rechts davon stehenden entsprechend um eine Position nach rechts verschoben. Diese Art der Implementierung ermöglicht einen effizienten Zugriff über den Index. Dieses Konzept implementiert in Java z. B. die Klasse *ArrayList*.

Eine andere Möglichkeit ist die Implementierung als doppelt verkettete Liste. Dazu wird eine innere Klasse *Node* definiert, die etwa so aussieht:

```
private static class Node<E> {
    E item;
    Node<E> next;
    Node<E> prev;

    Node(Node<E> prev, E element, Node<E> next) {
        this.item = element;
        this.next = next;
        this.prev = prev;
    }
}
```

Ferner gibt es in der Liste z. B. Attribute vom Typ *Node<E> first* und *last*, die das erste bzw. letzte Element der Liste enthalten. Um das Element an einer bestimmten Position zu finden, muss vom Anfang bzw. Ende der Liste aus entsprechend die Kette der Referenzen verfolgt und gezählt werden. Dies ist langsamer, als bei einer Implementierung mittels eines Arrays, dafür können andere Operationen effizienter implementiert werden. So entfällt beim Einfügen das Verschieben nach rechts.

Dieses Konzept implementiert in Java z. B. die Klasse *LinkedList*.

Für viele Anwendungen brauchen Sie Container mit speziellem Verhalten. Die wahrscheinlich wichtigsten Beispiele sind

Stack Ein Last-in first-out (LIFO) Speicher. Die üblichen Operationen sind *push*, um ein Element auf den *Stack* zu legen, *peek*, um das oberste Element zu erhalten und *pop*, um das oberste Element zu entfernen.

Queue Ein First-in first-out (FIFO) Speicher. Die üblichen Operationen sind *enqueue*, um ein Element am Ende der *Queue* einzufügen, *peek*, um das oberste Element zu erhalten und *dequeue*, um das erste Element zu entfernen.

Die genaue Ausgestaltung der Operationen hängt von der jeweiligen Verwendung der Container ab: Soll ein Element immer entnommen werden, wenn es verarbeitet wird, so werden Sie auf *peek* verzichten können und *pop* bzw. *dequeue* werden das jeweilige Element zurückliefern. Muss häufiger auf das letzte bzw. erste Element zugegriffen werden, so wird man dafür *peek* implementieren und *pop* und *dequeue* mit dem Rückgabetyp *void* definieren. Letzteres entspricht einem viefach empfohlenen Stil.

Das Interface *Deque* — double ended queue — definiert eine Struktur, die als *Stack* oder als *Queue* betrieben werden kann. Eine bewährte Implementierung ist *LinkedList*.

Weitere generische Klassen werden Sie bei der Programmierung mit *Swing* kennenlernen, z. B. *JList* und nach Bedarf entsprechend parametrisieren.

15.5 Assoziationen

Eine wichtige Anwendung der Container-Klassen ist die Implementierung binärer Assoziationen. Hier nochmals das überstrapazierte Beispiel Kunde—Auftrag: Die Richtung von Auftrag zu Kunde wird einfach durch ein Attribut vom Typ *Kunde* implementiert. Für die andere Richtung von Kunde zu Auftrag benötigen Sie eine geeignete Collection. Haben Sie keinerlei weitere Informationen bzw. Anforderungen, so bietet sich die Deklaration eines *Set*s an, denn eine Menge (engl. Set) enthält keine Duplikate. Wählen Sie stattdessen ein *List*e, so müssen Sie sich um diese Dinge selber kümmern. Das können Sie individuell machen oder sich eine Listenimplementierung von *Set* schreiben. Analoges gilt, wenn Sie diese Richtung mittels eines Arrays implementieren wollen. Außerdem haben Arrays eine feste Größe, so dass Sie sich auch um das Wachstum kümmern müssen.

Abb. 15.3: 1:* Assoziation Kunde — Auftrag

Mit der Definition der Assoziation in Java über die beiden genannten Attribute sind Sie aber noch nicht fertig: Beim Erzeugen von Objekten müssen Sie die Assoziation pflegen. Das heißt in diesem Beispiel:

1. Zu einem *Auftrag* muss es genau einen Kunden geben, d. h. mindestens einen und auch nicht mehr als einen. Daraus ergibt sich, dass die Klasse *Auftrag* einen *Konstruktor* benötigt, der neben anderen Parametern auf jeden Fall eine Referenz auf ein Kundenobjekt übergeben bekommt.

   1.1. In diesem Konstruktor muss die Referenz auf Kunde auf die übergebene Referenz gesetzt werden. So wird die Richtung der Assoziation von Auftrag zu Kunde gepflegt.

   1.2. Aber auch die andere Richtung muss gepflegt werden: Dazu muss es eine Methode *addAuftrag* in der Klasse Kunde geben.

2. Kunden müssen keinen Auftrag haben, also können Kundenobjekte einfach mit den notwendigen Attributen erzeugt werden, die hier nicht spezifiziert wurden.

   Was Sie an weiteren Operationen benötigen, hängt von den Anforderungen ab:

3. Wenn ausgehend von einem Kunden neue Aufträge angelegt werden sollen, so benötigen Sie eine Methode *addAuftrag* in der Klasse Kunde. Diese muss im Wesentlichen die Parameter bekommen, die der oben genannte Konstruktor von *Auftrag* bekommt. Innerhalb dieser Methode müssen dann zwei Dinge auf jeden Fall passieren:

   3.1. Der *Konstruktor* von Auftrag muss mit *this* als Referenz auf das Kundenobjekt aufgerufen werden.

   3.2. Die Richtung der Assoziation von *Kunde* zu *Auftrag* muss gepflegt werden. Hier kann es möglicherweise zu Konflikten kommen: Beim Aufruf des Kosntruktors von Auftrag wird diese Richtung bereits gepflegt. Es muss verhindert werden, dass dies zweimal passiert. Dies ist schon fast eine Zwickmühle (catch-22): Können Sie sicherstellen, dass der Ablauf immer bei einem Kunden beginnt und dann direkt ein neuer Auftrag erzeugt wird, so gibt es kein Problem.

   Ist die Methode *addAuftrag* von Kunde public, so könnte aber jemand auf die Idee kommen, irgend einen gültigen Auftrag einem Kunden hinzuzufügen. Ohne Überprüfung und entsprechende Aktion könnte dabei Vieles schief gehen:

   3.2.1. Es könnte ein Auftrag hinzugefügt werden, der schon zu einem anderen Kunden gehört. Wird dass in der Methode *addAuftrag* ignoriert, etwa so:

```
public void addAuftrag(Auftrag auftrag){
   this.auftrage.add(auftrag);
   auftrag.setKunde(this);
}
```

   so steht *auftrag* nach wie vor beim ursprünglichen Kunden in der Collection der Aufträge (auftraege). Die Assoziation wäre also nicht mehr korrekt nachvollzogen. Gibt es keine öffentliche Methode *setKunde* in Auftrag so ist das Problem bereits erheblich entschärft.

   3.2.2. Es könnte ein Auftrag, der bereits zu dem Kunden gehört, noch einmal hinzufügt werden. Dies wird z. B. abgefangen, wenn die Collection mittels eines *Set* implementiert wurde.

   Ist ein *Kunde* u. a. dadurch gekennzeichnet, dass er mindestens einen Auftrag hat („1..*“ am Auftragsende der Assoziation), so muss noch mehr bedacht werden. In diesem Fall muss es zu einem Kunden auch mindestens einen Auftrag geben. Naiv implementiert können Sie hier in eine Endlosschleife geraten. Generell gilt: Können Sie die Nutzung der Methoden nicht gegen fehlerträchtige Verwendung aufrufen, so müssen Sie alle möglichen Fehlersituation in den Methoden behandeln und angemessen reagieren.

In vielen Fällen haben Assoziationsenden mit einer Multiplizität „*“ etc. weitere Eigenschaften. So werden im obigen Beispiel aus Abb. 15.3 die Aufträge eines Kunden oft nach Erteilungs- oder Eingangsdatum geordnet sein. Dann ist ein Set nicht geeignet, sondern Sie werden eine geordnete Collection wählen, wie z. B. ein *SortedSet*.

15.6 Technische Einzelheiten

Aufgrund eines Problems, das in einem Praktikum auftrat, gebe ich hier ganz wenige Informationen zum Thema „Heap Pollution“.

Definition 15.6.1 (Heap pollution)
Heap pollution liegt vor, wenn ein Objekt eines parametrisierten Typs auf ein Objekt verweist, dass nicht von diesem parametrisierten Typ ist ([GJS+14], §4.12.2). ◄

Heap Pollution kann nur vorkommen, wenn:

1. Ein Programm Operationen mit einem *raw type* vornimmt, die zu einer Compiler Warnung führen.

2. Ein Programm für eine Array-Variable eines nicht-reifiable Elementtyps einen Alias mit einer Array-Variablen einführt, deren Elementtyp ein Supertyp oder raw ist.

Ein Beispiel liefert die Methode *m* der Klasse *HeapPollution* im Projekt Fehler, Paket *container*.
Also noch einmal: Deklarieren Sie keine Variablen irgend einer Art als raw type!

15.7 Historische Anmerkungen

Siehe Abschn. 18.15.

15.8 Aufgaben

1. Gegeben seien die folgenden Minimodelle in Abb. 15.4–15.7: Implementieren Sie bitte alle

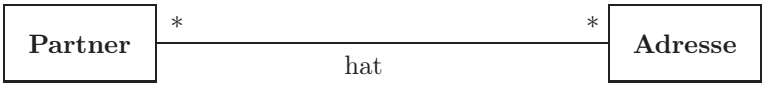


Abb. 15.4: *:* Assoziation Partner — Adresse

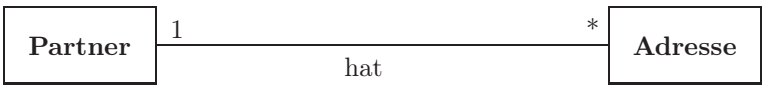


Abb. 15.5: 1:* Assoziation Partner — Adresse

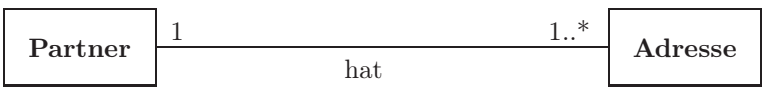


Abb. 15.6: 1:* Assoziation Partner — Adresse

angegebenen Minimodelle sicher in Java. Begründen Sie bitte Ihre Entwurfsentscheidungen!

2. In der folgenden Methode soll aus einer *List<Zahlung> zahlungsliste* ein Eintrag herausgesucht und gelöscht werden. Gibt es keinen passenden Eintrag, so soll eine *IndexOutOfBounds-Exception* geworfen werden. Finden Sie alle Fehler bzw. Unzulänglichkeiten und machen Sie Vorschläge zu deren Beseitigung.

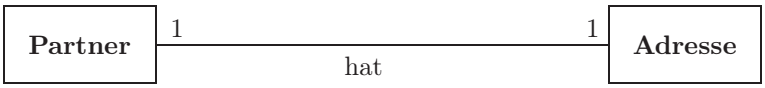


Abb. 15.7: 1:* Assoziation Partner — Adresse

```
   private List<Zahlung> zahlungsliste = new LinkedList<Zahlung>;
   ...
   public void remove(LocalDate zahlungsdatum)
               throws IndexOutOfBoundsException {
      for (int a = 0; a < zahlungsliste.size();) {
         if (zahlungsliste.get(a).getZahlungsdatum()==zahlungsdatum)
            zahlungsliste.remove(a);
         return;
      }
      throw new IndexOutOfBoundsException();
}
```

3. In vielen *Collection*s braucht man eine Methode *isEmpty*, die *true* zurück gibt, wenn die *Collection* leer ist und anderfalls *false*. Was ist an folgendem Implementierungsversuch aus einer Klausur falsch?

```
public boolean isEmpty() {
   return this==null ? true : false;
}
```

Kapitel 16

Noch mehr zu Klassen und Schnittstellen: λ-Ausdrücke

16.1 Übersicht

Auch in dieser Auflage enthält dieses Kapitel Ergänzungen, die den Rahmen der Kap. 4, 9 und 15 des Skripts zum Programmieren sprengen würden. Ich hoffe diese Inhalte später besser auf geeignete Kapitel des Skripts verteilen zu können.

Die Basis für λ-Ausdrücke in Java bilden funktionale Interfaces. Von diesen werden viele nützliche direkt mit Java geliefert, so dass Sie ein solches Interface oft gar nicht selbst werden schreiben müssen.

16.2 Lernziele

- Lambda-Ausdrücke in Java beherrschen.
- Sachgerecht über die Verwendung von Lambda-Ausdrücken und alternativen Lösungsansätzen entscheiden können.

16.3 Funktionale Interfaces

Definition 16.3.1 (Funktionales Interface)
Ein *funktionales Interface (functional interface)* ist ein *interface*, dass außer den Methoden aus *Object* nur eine abstrakte Methode hat. ◄

Dieses ist die entscheidende Eigenschaft! Wird ein funktionales Interface als Typ spezifiziert, so kann ein λ-Ausdruck als entsprechender aktueller Parameter verwendet werden oder einer mit diesem Typ deklarierten Variablen zugewiesen werden.

Bemerkung 16.3.2 (@FunktionalInterface)
Ein *funktionales Interface* kann mit der Annotation *@FunktionalInterface* gekennzeichnet werden. Das ist für die Verwendung nicht wichtig, verhindert aber, das eine weitere abstrakte Methode hinzugefügt wird. ◄

Beispiel 16.3.3 (Funktionales Interface)
Hier eine Reihe einfacher Beispiele:

1. Das Interface *Comparable<T>* aus *java.lang* ist ein funktionales Interface: Es enthält nur die abstrakte Methode *compareTo*.

2. Das Interface *Comparator<T>* aus *java.util* ist ein funktionales Interface. Es enthält nur die abstrakte Methode *compare*. Alle anderen Methoden, die nicht aus *Object* stammen, sind *default*-Methoden.

◀

Definition 16.3.4 (Untersignatur)
Die *Signatur* einer Methode m_1 ist *Untersignatur (subsignature)* der *Signatur* einer Methode m_2, wenn entweder

- m_2 die gleiche *Signatur* wie m_1 hat oder
- Die *Signatur* von m_1 ist gleich der *Signatur* von m_2 nach Typauslöschung (erasure) (vgl. Abschn. 18.4).

◀

Beispiel 16.3.5 (Untersignatur)
Die Signatur einer Methode, die mit einem raw type deklariert ist, ist Untersignatur der entsprechenden generisch deklarierten Methode. ◀

Definition 16.3.6 (Override equivalent Methods)
Zwei *Methoden* m_1 und m_2 heißen genau dann *override equivalent*, wenn die *Signatur* der einen *Untersignatur* der anderen ist. ◀

Definition 16.3.7 (Rückgabetyp-substituierbar)
Eine Methodendeklaration m_1 mit Rückgabetyp R_1 ist mit einer Methodendeklaration m_2 mit Rückgabetyp R_2 genau dann *Rückgabetyp-substituierbar*, wenn eines der drei folgenden Dinge gilt:

1. Wenn R_1 *void* ist, so ist auch R_2 *void*.
2. Wenn R_1 ein *primitiver Typ* ist, so ist R_2 identisch mit R_1.
3. Wenn R_1 eine *Referenztyp* ist, so gilt eines der folgenden Dinge:
   3.1. Werden die Typparameter von R_1 an die von m_2 angepasst, so ist dies ein Untertyp von R_2.
   3.2. R_1 kann durch eine *unchecked conversion* in einen Untertyp von R_2 konvertiert werden.
   3.3. m_1 und m_2 haben nicht die gleiche *Signatur* und es ist $R_1 = |R_2|$ (R_2 nach Typauslöschung)

◀

Definition 16.3.8 (Funktionales Interface)
Etwas allgemeiner ist ein *functional interface* so definiert: Sei I ein Interface und M die Menge der abstrakten Methoden aus I, die nicht Methoden von *Object* sind. Dann ist I ein funktionales Interface, wenn es eine Methode $m \in M$ gibt, für die gilt:

1. Die Signatur von m ist Untersignatur der Signatur von m $\forall m \in M$.
2. m ist für jede Methode $m \in M$ Rückgabetyp substituierbar.

◀

Paket *java.util.function* enthält eine Reihe nützlicher funktionaler Interfaces, die als Zieltyp (target type) für *Lambda-Ausdrücke* verwendet werden können.

Bemerkung 16.3.9 (Funktionales Interface)
Eine *funktionales Interface* kann also mehrere Methoden haben. Diese müssen aber *override equivalent* sein. ◀

Beispiel 16.3.10 (Funktionales Interface)
Das Das Interface *Comparator<T>*

```
interface Comparator<T> {
   boolean equals(Object obj);
   int compare(T o1, T o2);
   ...
}
```

ist ein funktionales Interface: Die Methode *equals* stammt aus *Object*, *compare* ist neu und *abstrakt*. Die anderen Methoden (neue in Java 8) sind *Default-Methoden* oder *Klassenmethoden*. Das Interface *Comparable<T>*

```
public interface Comparable<T>{
    int compareTo(T o);
}
```

ist ein funktionales Interface: Die einzige Operation *compareTo* ist abstrakt. ◄

Definition 16.3.11 (Default-Methode)
Eine *Default-Methode* eines *Interfaces* ist eine Methode, die den *Modifier default* hat und einen Block mit einer Implementierung. Sie stellt eine Implementierung für Klassen zur Verfügung, die das Interface implementieren, ohne die Methode zu überschreiben. ◄

Beispiel 16.3.12 (Default-Methode)
Das Interface *Iterable* hat die *Default-Methode forEach*, die so implementiert ist:

```
    default void forEach(Consumer<? super T> action) {
        Objects.requireNonNull(action);
        for (T t : this) {
            action.accept(t);
        }
    }
```

◄

16.4 Lambda-Ausdrücke

Lambda-Ausdrücke können auch als *anonyme Methoden* bezeichnet werden. Die Definition macht dies deutlich:

Definition 16.4.1 (Lambda-Ausdruck)
Ein Lambda-Ausdruck ist definiert als:

```
LambdaExpression:
   LambdaParameters -> LambdaBody
```

LambdaParameters ist einfach eine Liste von Parameternamen, wie beim Aufruf einer Methode. *LambdaBody* ist ein Codeblock {...}. Die Auswertung eines *Lambda-Ausdrucks* liefert eine Instanz eines *funktionalen Interfaces*. Der *Lambda-Block* wird erst ausgeführt, wenn die Methode des funktionalen Interfaces ausgeführt wird. ◄

In *LambdaParameters* kann in vielen Fällen auf die Angabe der Typen verzichtet werden. Das liegt daran, dass ein funktionales Interface nur eine abstrakte Methode hat, die dem Compiler ja bekannt ist.

Einige Details fehlen noch, aber Sie können jetzt schon erkennen, warum es immer heißt, nun könnten in Java Funktionen (genauer Methoden) als Parameter übergeben werden. Haben Sie ein funktionales Interface:

```
@FunctionalInterface
Blah{
    String foo(String s);
}
```

so können Sie solchen Code schreiben

```
void someMethod(Blah b){
   ...
   b.foo(...);
   ...
}

String result = someMethod(s->s.toUpperCase()+"!")
```

Innerhalb der Methode *someMethod* müssen Sie den Namen der Methode *foo* kennen. Die gilt auch, wenn Sie eine annonyme Klasse verwenden. Bei Verwendung eines λ-Ausdrucks verwenden Sie den Namen nicht.

Der Methode *someMethod* wird ein Objekt vom Typ des Interfaces *Blah* übergeben. Dessen einzige abstrakte Methode *foo(String s)* hat durch den Lambda-Audruck den Rumpf

```
return s.toUpperCase()+"!";
```

Diese Implementierung der abstrakten Methode wird in *someMethod* überall dort verwendet, wo *foo* verwendet wird. *Lambda-Ausdrücke* können nur in folgenden Situationen verwendet werden

1. Auf der rechten Seite einer Zuweisung ([GJS+14], §5.2).
2. Beim Aufruf einer Methode ([GJS+14], §5.3).
3. In einem Cast ([GJS+14], §5.5).

Jede andere Verwendung führt zu einem Compiler-Fehler.

Beispiel 16.4.2 (Lambda-Ausdrücke)
Für jede dieser Verwendungen nun je ein Beispiel:

Zuweisung Ich verwende das funktionale Interface *BinaryOperator* aus *java.util.function*:

```
public interface BinaryOperator<T> extends  BiFunction<T,T,T>{
   T apply(T left, T right);
}

   BinaryOperator<Integer> sum = (x, y) -> x + y;
```

Aufruf

```
   List<String> strLst = new ArrayList<>();
   strLst.sort((s1,s2)->s1.compareToIgnoreCase(s2));
```

cast Kann der Typ eines *lambda-Audrucks* nicht vom Compiler festgestellt werden, muss ggf. ein Cast erfolgen, so wie hier am Beispiel des *funktionalen Interfaces Runnable*: Der folgende Ausdruck führt auf einen Compiler-Fehler:

```
   Object t = ()-> System.out.println("Tschüß!!");
```

Da *Object* kein funktionales Interface ist, führt dies auf den Fehler: „Target type of this expression must be a functional interface“. Funktionieren tut es mit cast:

```
Object o = (Runnable) () -> { System.out.println("moin"); };
```

oder mit der engeren Deklaration:

```
Runnable r = ()-> System.out.println("Moin!!");
```

◀

Für *λ-Ausdrücke* gelten einige Regeln, die sich zum Teil aus Compiler-Gesichtspunkten oder praktischen Erwägungen ergeben:

- *Lambda-Parameter* dürfen nicht nur aus einem einzelnen Unterstrich _ bestehen. Zukünftige Versionen werden diesen Namen eventuell als Schlüsselwort definieren oder eine besondere Bedeutung zuweisen [GJS+14], §15.27.1.
- *this* und *super* in einem Lambda Body haben dort die gleiche Bedeutung wie im jeweiligen Kontext. Analog hat der Lambda-Block Zugriff auf die Elemente, die aus dem Kontext heraus verwendet werden können.
- Jede lokale Variable, formaler Parameter oder Exception-Parameter, die in einem *λ-Audruck* verwendet, aber nicht deklariert werden, muss als *final* deklariert sein oder effektiv *final* ([GJS+14], §4.12.4) sein.
- Jede lokale Variable die in einem *LambdaBody* verwendet aber nicht deklariert wird, muss vor dem *LambdaBody* definitiv zugewiesen werden ([GJS+14] §16).

Bemerkung 16.4.3 (Selbstreferenz im Lambda-Block)
Es ist unüblich, dass ein Lambda-Block sich selbst referenziert, sei es rekursiv oder um seine anderen Methoden aufzurufen. Weitaus häufiger ist es, dass aus einem Lambda-Block auf Elemente der umschließenden Klasse zugegriffen werden muss. Muss ein Lambda-Ausdruck sich selbst referenzieren (z. B. via *this*), so sollte stattdessen eine *Methodenreferenz* oder eine *anonyme Klasse* verwendet werden. [GJS+14] ◀

Definition 16.4.4 (void kompatibel)
Ein *Lambda-Body* ist *void kompatibel*, wenn jedes *return*-Statement in dem Block die Form `return;` hat. Das heißt: Die Methode des funktionalen Ziel-Interfaces hat den Rückgabetyp *void*. ◀

Definition 16.4.5 (Normal beenden)
Leger formuliert beendet ein Statement normal, wenn die Ausführung nicht vor dem Semikolon beendet wird. Beispiel hierfür sind *return*, *break* und der Aufruf der Methode *System::exit*:

In Abschn. 14.21 der Java Language Specification [GJS+14] beschreibt detailiert, wann ein Statement *normal beenden* kann. ◀

Definition 16.4.6 (Wert-kompatibel (value compatible))
Ein *Lambda-Body* ist *Wert-kompatibel (value compatible)*, wenn er nicht normal beenden kann (siehe Def. 16.4.5, [GJS+14], §14.21) und jedes return Statement die Form `return Ausdruck` hat. Leger formuliert: Der Lambda-Body liefert etwas (nicht void). ◀

Bemerkung 16.4.7 (void und value kompatibel)
Ein *Lambda Block* muss *void* oder *value kompatibel* sein, andernfalls gibt es einen Compiler-Fehler. ◀

Zur Laufzeit wird ein Lambda-Ausdruck ähnlich behandelt, wie das Erzeugen eines Objekts einer Klasse (instance creation expression): Die normale Ausführung liefert eine Referenz auf ein Objekt. Das heißt aber nicht, dass dann auch der Lambda-Body ausgeführt wird.

Lambda-Ausdrücke und funktionale Schnittstellen (functional interfaces) gehören eng zusammen:

Jeder Ausdruck in Java liefert entweder kein Ergebnis oder eines, das zur Compile-Zeit bestimmt werden kann. Dabei kann der Ziel-Typ einen impliziten Cast erfordern.

16.5 Methodenreferenzen

Methodenreferenzen ermöglichen es, vorhandene Methoden dort einzusetzen, wo Sie sonst einen Lambda-Ausdruck schreiben müssten. Die Konsequenzen sind:

1. Sie können Methodenreferenzen nur dort einsetzen, wo Sie auch Lambda-Ausdrücke verwenden können.

2. Sie brauchen jeweils ein geeignetes funktionales Interface.

Die allgemeine Syntax für Methodenreferenzen ist:

```
MethodReference:
   ExpressionName :: [TypeArguments] Identifier
   ReferenceType :: [TypeArguments] Identifier
   Primary :: [TypeArguments] Identifier
   super :: [TypeArguments] Identifier
   TypeName . super :: [TypeArguments] Identifier
   ClassType :: [TypeArguments] new
   ArrayType :: new
```

Ein einfaches Beispiel verwendet das Interface *Function* aus dem Paket *java.util.function*.

```
@FunctionalInterface
public interface Function<T,R>{
   R apply(T t);
}
```

Um dieses Interface einzusetzen, brauchen Sie Code, der mit der (einzigen) Methode *apply* arbeitet, z. B. :

```
      public <T,R> void callMethodByReference(Function<T, R> f,T t) {
        System.out.println(f.apply(t));//Zeigt nur die Funktionalität
    }
```

Diese können Sie dann so aufrufen:

```
  callMethodByReference(Math::cos,0.0);
```

Entscheidend ist nur das oben ausgeführte: Sie schreiben an die Stelle der Parameter des Aufrufes zwar eine Methodenreferenz und in diesem Fall den Parameter der Funktion. Für einige der primitiven Typen gibt es nicht-generische Varianten, so etwa *DoubleFunction* oder *DoubleToIntFunction*.

Auch der *new Operator* kann als Methodenreferenz verwendet werden (s. o., Fall ClassType). Da dieser ein Objekt einer Klasse liefert, brauchen Sie ein funktionales Interface mit einer Methode, die ein Objekt einer Klasse liefert. Auch ein solches finden Sie in *java.util.function*

```
@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

Diese Methode liefert bei Bedarf z. B. eine *ArrayList*, wenn die folgende Methode:

```
    private static <T> List<T> buildList(Supplier<List<T>> supplier) {
        return supplier.get();
    }
```

mit

```
List<String> list = buildList(ArrayList::new);
```

Im Beispiel mit *cos* wurde eine Klassenmethode verwendet. Instanzmethoden können genauso verwendet werden: Sollen z. B. Elemente vorverarbeitet werden, bevor sie einer Liste hinzufügt werden, so können Sie es machen, wie in folgendem Beispiel:

Beispiel 16.5.1 (Methodenreferenz)
Wir haben eine Liste von Strings, die aber nur Strings enthalten soll, bei denen bestimmte Veränderngen vor dem Speichern in der Liste vorgenommen werden sollen. Hierzu kann eine add-Methode wie folgende dienen:

```
public void add(String s, Function<String,String> func){
      add(func.apply(s));
}
```

Soll der String z. B. vor dem Speichern in Großbuchstaben konvertiert werden, so geht dass dann so:

```
List<String> lstList = ...
lstList.add("Hugo",String::toUpperCase());
```

◀

16.6 λ-Ausdrücke als Alternative

In einigen typischen Situationen, in denen bisher gerne anonyme Klassen verwendet wurden, kommen jetzt auch Lambda-Ausdrücke in Frage. Hier zwei Beispiele:

```
   Collections.sort(wagonListe, new Comparator<Wagon>() {
      @Override
      public int compare(Wagon w1, Wagon w2) {
         return Integer.compare(w1.getTara(), w2.getTara());
      }
   });

   button.addActionListener(new ActionListener() {

               @Override
               public void actionPerformed(ActionEvent e) {
                  numberOfClicks++;
                  clickDisplay.setText("Anzahl Clicks: " + numberOfClicks);
               }
});
```

Eine Alternative zum Einsatz von anonymen Klassen ist die Verwendung eines Lambda-Ausdrucks. Dann wird statt einer anonymen Klasse eine anonyme Methode verwendet. In den beiden obigen Beispielen geht das etwa so:

```
   Collections.sort(wagonListe, (Wagon w1, Wagon w2)-> {
         return Integer.compare(w1.getTara(), w2.getTara());
   });
```

bzw.

```
button.addActionListener((ActionEvent e)-> {
                numberOfClicks++;
                clickDisplay.setText("Anzahl Clicks: " + numberOfClicks);
            });
```

Das Schema und die Unterschiede kann man also so beschreiben

1. Wird ein Objekt von einem Interface-Typ benötigt, so schreiben Sie eine anonyme Klasse. Im Ergebnis wird eine weitere .class-Datei erzeugt und die dort enthaltenen Methoden werden wie üblich aufgerufen.

2. Ist der Interface-Typ *funktional*, so können Sie stattdessen an der Stelle des Objekts der anonymen Klasse einen Lambda-Ausdruck übergeben. An der entsprechenden Stelle im Byte-Code steht dann direkt der Byte-Code der Methode. Bei einem *ActionListener* ist das also *actionPerformed*.

Methoden, die bereits einen Namen haben, können Sie mittels Methodenreferenzen an der Stelle von Lambda-Ausdrücken verwenden. Die Syntax hierfür ist in Kurzform *TypeName::MethodName*. In Langform ([GJS+14]):

```
MethodReference:
   ExpressionName :: [TypeArguments] Identifier
   ReferenceType :: [TypeArguments] Identifier
   Primary :: [TypeArguments] Identifier
   super :: [TypeArguments] Identifier
   TypeName . super :: [TypeArguments] Identifier
   ClassType :: [TypeArguments] new
   ArrayType :: new
```

Viele Beispiele für deren Verwendung finden Sie im Zusammenhang mit *Streams*.

16.7 Datum und Uhrzeit

Mit Java 8 kommen neue Pakete für den Umgang mit Datum und Zeit, insbesondere *java.time* und seine Unterpakete.

Die wichtigsten Klassen zum Umgang mit Datum und Uhrzeit finden Sie im Paket *java.time*, etwa *LocalDateTime* und verwenden den ISO-Kalender aus ISO-8601. Weitere Kalender finden Sie im Paket *java.time.chrono*.

Die Klassen im Paket *java.time* und seinen Unterpaketen haben einige Eigenschaften gemeinsam, die ihre Verwendung einfach machen:

1. Haben Sie ein Objekt einer dieser Klassen, so kann es nicht mehr verändert werden; es kann sozusagen „nichts mehr schief gehen“. Alle diese Klassen sind *immutable*, ihre Objekte werden durch komfortable Fabrikmethoden erzeugt.

2. Es gibt einfache Methoden mit weitgehend selbsterklärenden Namen, hier eine Auswahl:

   - now(): Liefert das aktuelle Datum bzw. die aktuelle Uhrzeit.
   - of(...) liefert auf einfache Weise ein gewünschtes Datum oder eine Uhrzeit
   - parse
   - format
   - ...

Brauchen Sie nur ein Datum bzw. eine Uhrzeit, so verwenden Sie *LocalDate* bzw. *LocalTime*. Brauchen Sie beides, so nehmen Sie entsprechend *LocalDateTime*.

Beispiel 16.7.1 (Datum und Uhrzeit)

1. Ohne weitere Maßnahmen bekommen Sie aktuelles Datum und aktuelle Uhrzeit in dieser Form:

```
System.out.println(LocalDateTime.now());
System.out.println(LocalDate.now());
System.out.println(LocalTime.now());
```

   Die Ausgabe ist in diesem Fall etwas in diesem Format:

```
2014-07-24T09:59:55.789
```

2. Wollen Sie ein bestimmtes Datum haben, geht das z. B. so:

```
System.out.println(LocalDate.of(2014,07,24));
System.out.println(LocalTime.of(11,1,01));
System.out.println(LocalDateTime.of(2014,07,24,11,1,01));
```

   Die Ausgabe hat in diesem Fall etwa dieses Format:

```
2014-07-24
11:01:01
```

3. Wollen Sie das nun nicht im amerikanischen Datums- oder Uhrzeitformat haben, so können Sie eine Lösungsstrategie anwenden, die fast immer hilft: Um mit einem Objekt, hier einem Datum (Klasse *LocalDate* etc.) etwas zu machen brauchen Sie ein Objekt, in diesem Fall eines, das die gewünschte Formatierung vornimmt. Mit dieser Lösungsstrategie kommen Sie hier zügig zum Ziel: Die Klassen *LocalDate* etc. haben alle eine Methode *format*, die einen *DataTimeFormatter* als Parameter erwartet. Für Objekte dieser Klasse gibt es praktische *Fabrikmethoden*, z. B. *ofPattern*. Die genauen Möglichkeiten der Formatierung entprechend geeigneter Pattern finden Sie in der API-Dokumentation. Das Ergebnis ist ein *String*, den Sie bei Verwendung der Methode *printf* einfach geeignet ausgeben können. Hier drei Beispiele:

```
System.out.printf("%s\n",LocalDate.of(2014, 07,24).
           format(DateTimeFormatter.ofPattern("dd.MM.YYYY")));
System.out.printf("%s\n",LocalDate.parse("24.12.2014",
                          DateTimeFormatter.ofPattern("dd.MM.yyyy")).
           format(DateTimeFormatter.ofPattern("dd.MM.YYYY")));
System.out.printf("%s\n",LocalDateTime.now().
           format(DateTimeFormatter.ofPattern("dd.MM.YYYY  HH:mm ")));
```

   Das Ergebnis ist hier:

```
24.07.2014
24.12.2014
24.07.2014  17:05
```

   Gleichzeitig sehen Sie, wie Sie mittels eines entsprechenden Formatters auch das Eingabeformat eines Datums in der Methode *parse* angepasst bzw. korrekt verarbeitet werden kann. Sie müssen aber genau darauf achten, was Sie tun: So steht hier „m“ für Minute und „M“ für Monat.

◀

In der API-Dokumentation finden Sie eine vollständige Liste der verfügbaren Formatierungsmöglichkeiten für Datum und Uhrzeit. Die Klasse *DateTimeFormatter* liefert Ihnen viele Formatierungen „frei Haus“.

16.8 Strings

Die Klasse *StringJoiner* aus dem Paket *java.util* dient zum „Zusammenbau“ von Strings, die durch ein ausgewähltes Zeichen (Delimiter) getrennt sind. Wahlweise kann ein Präfix oder ein Suffix angegeben werden. Im Konstruktor werden Trennzeichen, Prä- und Suffix angegeben:

```
StringJoiner sj = new StringJoiner(",","(", ")");
```

Die Strings werden also durch Kommata getrennt und die ganze Folge in Klammern eingeschlossen, wie bei der mathematischen Schreibweise einer Folge oder eines Vektors.

Beispiel 16.8.1 (StringJoiner)
Der folgende Code fügt einfach die Buchstaben a–z zu einem String zusammen:

```
public class StringJoinerExample01 {
    public static void main(String[] args) {
        StringJoiner sj = new StringJoiner("");
        for (char c = 'a'; c <= 'z'; c++) {
            sj.add(Character.valueOf(c).toString());
        }
        System.out.println(sj.toString());
    }
}
```

◀

16.9 Historische Anmerkungen

Mit Java 8 wurden viele Neuerungen eingefügt. Die Prominentesten sind wohl Lambda- (λ-) Ausdrücke und *Streams*.

16.10 Aufgaben

Kapitel 17

Streaming API

17.1 Übersicht

Streams gibt es in Java in zwei Paketen, sozusagen „Geschmacksrichtungen“:

- Die aus anderen Sprachen bekannten *InputStream*s und *Outputstream*s und ihre Verwandten. Diese finden Sie im Paket *java.io*.
- Datenhüllen um Datenquellen, z. B. Collections. Die entsprechenden Klassen und vor allem Interfaces finden Sie im Paket *java.util.stream*

Erstere bilden Datenströme von bzw. zu einem Programm ab und werden in Kap. 14 detailliert beschrieben.

Letztere Streams kapseln Datenquellen und bieten effiziente Methoden um Elemente sequentiell oder parallel zu verarbeiten. Um diese geht es in diesem Kapitel. Dieses *Streaming API* stellt im Wesentlichen eine Variante des *Iterator Patterns* zur Verfügung: *Interne Iterator*en (siehe Abschn. 23.8. In dieser Form wird eine Funktion auf alle Elemente des Aggregats angewendet. Das weist schon auf eine Beziehung zu λ-Ausdrücken hin.

Dies folgt dem Vorgehen vorhergehender Java-Versionen, viel Funktionalität über Interfaces zur Verfügung zu stellen. Die Utility-Klasse *StreamSupport* bietet Fabrikmethoden zur Erzeugung von Streams.

Hier einige wichtige Eigenschaften von Streams:

- Lazy Evaluation: Wenn klar ist, das nichts getan werden muss, dann wird auch nichts getan.
- Short-ciruit: Manche Stream-Methoden sind effizient, weil sie schnell fertig sein können.
- Es gibt für *Streams intermediate* und *terminal* Methoden.
- Unbegrenzte Streams sind möglich: Es muss keine maximale Größe bei Beginn angegeben werden.
- Verarbeitung kann seriell oder parallel erfolgen. Es muss (im Prinzip, es grüßt Radio Eriwan keine Vorsorge für *thread safety* getroffen werden, wenn der Kontext dies zu lässt.

17.2 Lernziele

- Die Methoden des *Streaming API*s beherrschen.
- *Stream*s sicher verwenden können.
- *Stream*s nach Bedarf seriell oder parallelisiert verarbeiten können.
- Den Unterschied zwischen *intermediate* und *terminal* Methoden kennen und diese sicher einsetzen können.

17.3 Grundprinzipien

Die in desem Kapitel behandelten Streams bilden Hüllen um Datenquellen.[1] Beispiele sind beliebige Collections, Arrays, Generatorfunktionen oder I/O-Kanäle. Ein auf eine dieser Weisen erzeugtes Stream-Objekt hält selber keine Daten, sondern ist nur eine Hülle um die jeweilige bestehende Datenquelle. Dabei können Streams die Datenquelle nicht verändern (nur Elemente filtern) und gewähren keinen indexierten oder wahlfreien Zugriff auf diese. Dadurch soll erreicht werden, dass Streams möglichst „leichtgewichtig“ und performant sind.

Mit diese Eigenschaften stehen sie im Kontrast zu Collections, die ihre eigenen Daten halten und verwalten.

Auf einem Stream lässt sich eine beliebige Anzahl aneinander gereihter intermediate Methoden anwenden, die selber jeweils wieder einen Stream erzeugen. Diese intermediate Methoden nutzen in der Regel die in Java 8 hinzugekommenen Lambda-Ausdrücke. Gängige Beispiele hierfür sind *map* oder *filter*. Ein Stream wird mit einer einzelnen terminal Methode abgeschlossen. Diese terminal Methode darf im Gegensatz zu den intermediate Methoden Seiteneffekte haben und einen Wert zurückgeben. Durch die Benutzung von Streams mit Quelle, intermediate Methoden und abschließender terminal Methode wird eine Stream-Pipeline erstellt, durch die die Elemente des Streams gereicht werden.

Aus den bekannten Collections in Java lässt sich mittels der default Methode *stream* aus dem Interface *Collection* ein sequentieller *Stream* erzeugen. Die Klasse Stream ermöglicht es mittels der terminal Methode *collect* und dem Interface *Collector* aus einem Stream eine *Collection* zu erzeugen.

17.4 Das Interface Stream

Die Wurzel der Stream-Hierarchie bildet das Interface *BaseStream*. Diese Klasse fasst einige Eigenschaften aller *Stream*s zusammen. Die Klasse *StreamSupport* bietet eine Reihe von Klassenmethoden zur Erzeugung von Streams von *int*, *long*, *double* (*xxxStream* oder einem *Stream*<*T*> von *(*<T>) (stream). Alle *Stream*s sind *AutoCloseable*.

Die wichtigen Methoden für *Stream*s stehen im Interface *Stream*. Die hier definierten Methoden haben jeweils eine oder mehrere der folgende Eigenschaften:

intermediate Transformieren einen *Stream* in einen weiteren *Stream*. Sie produzieren einen neuen Stream und werden erst ausgeführt, wenn eine terminal Methode aufgerufen wird. Intermediate Methoden sind also immer *lazy*.

terminal Nach einer terminal Methode ist ein *Stream* verarbeitet (konsumiert) und kann nicht mehr verwendet werden.

lazy Werden erst ausgeführt, wenn eine terminal Methode aufgerufen wird.

short-circuit Können aus einem unbegrenzten *Stream*(infinite) einen begrenzten *Stream*(finite) erzeugen.

stateful Der Zustand eines Element des Streams wird nicht festgehalten, wenn das nächste geliefert wird.

stateless Der Zustand eines Element des Streams wird festgehalten, wenn das nächste geliefert wird.

Beispiel 17.4.1 (Intermediate Methoden)
◀

Beispiel 17.4.2 (Terminal Methoden)
◀

[1] Teile dieses Kapitels stammen aus einer Hausarbeit von Steffen Giersch aus dem Wintersemester 2014/15

Beispiel 17.4.3 (Short-circuit Methoden)
◄

17.5 Erzeugen und Benutzen von Streams

Beispiel 17.5.1 (Erzeugung und Benutzung eines Streams)
Der folgende Code filtert die geraden Zahlen aus einem Stream von ganzen Zahlen heraus.

```
List<Integer> foo = new ArrayList<Integer>(
              Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10));
foo.stream().filter(i -> (i&1) == 0)
            .forEach(System.out::println);
```

In diesem Beispiel wird in Zeile 1 zunächst eine Liste aus Integern mit dem Namen foo erzeugt. Ab Zeile 2 wird dann die Methode stream aus java.util.Collection benutzt um einen Stream aus Integern zu initialisieren. Auf diesen Stream wird nun die Methode *filter* mit einem Lambda-Ausdruck aufgerufen, der für gerade Zahlen *true* und für ungerade Zahlen *false* ausgibt. Die Methode *filter* erwartet ein *Predicate* als Parameter. Sie ist eine intermediate-Methode, weil sie einen neuen Stream aus allen, nicht herausgefilterten, Elementen erzeugt. Auf den gefilterten Stream wird dann die Methode *forEach* mit einem einfachen Parameter *System.out::println*, einer Methodenreferenz aufgerufen, um die übrig gebliebenen Elemente auf der Konsole auszugeben. Die Methode *forEach* ist eine terminal Methode, allerdings macht sie in diesem Fall keinen Gebrauch von Seiteneffekten. ◄

Streams können auch aus Generatormethoden erzeugt werden. Hierzu lässt sich die statische default Methode *generate* aus dem Stream-Interface nutzen. Diese Generatormethode erzeugt einen unendlichen, sequentiellen, ungeordneten Stream. Diesen können Sie dann verarbeiten.

Beispiel 17.5.2 (Generator-Methode)
Hier eine ganz einfache Generatormethode:

```
0   public static int i = 0;
1   private static Integer generator() {
2      return i++;
3   }
    ...
4   Stream.generate(Generierung::generator)
```

In diesem Beispiel wird ein Stream mittels einer Generator-Methode erzeugt. Die Generator-Methode (Zeile 1–3) basiert auf einer Zählvariablen (Zeile 0), die bei jedem Aufruf inkrementiert wird. In Zeile 4 wird die Methode generate aus Stream aufgerufen. Immer wenn der Stream nun ein Element benötigt ruft er die Generator-Methode auf.

Mittels der weiteren Methode *iterate(T seed, UnaryOperator<T> f)* zum Erzeugen unbegrenzter Streams können Sie Beispiel 17.5.1 auch so schreiben:

```
Stream.iterate(1, i -> i + 1).limit(10).
       filter(i -> (i & 1) == 0).
       forEach(System.out::println);
```

Denken Sie bitte daran, auch eine short-circuit-Methode aufzurufen! ◄

Um Streams performanter zu machen werden viele Methoden (z.B. map oder filter) erst aufgerufen, wenn von ihnen ein Element gefordert wird - sie setzen also lazy-evaluation um. Dazu kommen short-circuit Methoden, die oftmals nur wenige Elemente aus den vorherigen Streams benötigen um zu terminieren. Wenn nun eine short-circuit Methode nach einer Methode mit lazy-evaluation aufgerufen wird, so wird die Methode mit lazy-evaluation nur so viele Elemente zunächst verarbeiten und dann neu erzeugen, wie die short-circuit Methode benötigt.

Beispiel 17.5.3 (Short-Circuit-Methoden)
Der folgende Code

```
Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10).
   map(i -> {
      System.out.println("Map has been called!");
      return i;
   }).
   limit(3).
   forEach(System.out::println);
```

liefert diese Ausgabe:

```
Programm Output:
Map has been called!
1
Map has been called!
2
Map has been called!
3
```

In diesem Beispiel wird ein Stream aus den Zahlen 1 bis 10 erzeugt und die Methode *map* aufgerufen. Auf dem erzeugten Stream wird die Methode *limit* mit einer Beschränkung von 3 aufgerufen. Zuletzt werden die übrig gebliebenen per System.out::println ausgegeben. Es entsteht der Eindruck, als würde map für alle 10 Elemente aufgerufen, von denen durch limit aber nur 3 benutzt werden. Wie allerdings am Output zu erkennen ist, wird map nur genau 3 mal aufgerufen. ◀

Um Streams noch performanter zu machen, können diese auf sehr einfache Weise parallelisiert werden. Hierzu reicht es die intermediate-Methode *parallel* aufzurufen. Wenn diese Methode aufgerufen wurde laufen alle folgenden intermediate-Methoden und die abschließende terminal-Methode nebenläufig. Hierdurch sinkt die Laufzeit Verarbeitung der Streams fast linear mit der Anzahl der verfügbaren Prozessoren.

Beispiel 17.5.4 (Parallele Streams)
Der folgende Code

```
Stream<Integer> serialStream = Stream.of(1, 2, 3, 4);
Stream<Integer> parallelStream = Stream.of(1, 2, 3, 4).parallel();

long serialStartTime = System.currentTimeMillis();
serialStream.forEach(s -> StreamSamples2.doSlowOp());
long serialEndTime = System.currentTimeMillis();

long parallelStartTime = System.currentTimeMillis();
parallelStream.forEach(s -> StreamSamples2.doSlowOp());
long parallelEndTime = System.currentTimeMillis();

System.out.println("Serial time: \t" + (serialEndTime - serialStartTime)
                              + " milliseconds");
System.out.println("Parallel time: \t" + (parallelEndTime - parallelStartTime)
                                + " milliseconds" + " on "
                                + Runtime.getRuntime().availableProcessors()
                                + " Processors");
```

liefert z. B. diese Ausgabe:

```
Serial time:  4098 milliseconds
Parallel time:  2006 milliseconds on 2 Processors
```

In Zeile 1 wird ein serieller Stream mit den Zahlen 1–3 initialisiert. In Zeile 2 wird sein paralleles Gegenstück erzeugt. Der einzige Schritt, der notwendig ist um diesen Stream zu parallelisieren, ist es, die Methode *parallel* aufzurufen. In den Zeilen 4 und 7 wird sowohl auf den parallelen als auch auf den seriellen Stream die Methode *doSlowOp* aufgerufen. Die einzige Aufgabe dieser Methode ist es, 1 Sekunde zu warten. In den Zeilen 3, 5, 6 und 8 werden Zeitstempel genommen, um die Laufzeit der jeweiligen Stream-Aufrufe zu messen. Wie am Output zu sehen ist, braucht der parallele Stream nur ein Drittel der Laufzeit des seriellen Streams, ein deutliches Indiz dafür, dass die Elemente parallel verarbeitet werden. ◀

17.6 Stream-Methoden

Das Stream-Interface besitzt eine Vielzahl an Methoden, um mit diversen Streams zu arbeiten. Diese Methoden lassen sich in zwei Kategorien einteilen: intermediate Methoden und terminal Methoden. Des Weiteren gibt es short-circuit Methoden welche sowohl intermediate als auch terminal Methoden sein können. Zusammen lässt sich aus diesen Methoden eine Stream-Pipeline bauen, die mit einer Quelle beginnt, beliebig viele intermediate Methoden enthält und mit einer terminal Methode endet. Intermediate Methoden konsumieren die Elemente des aufrufenden Streams.

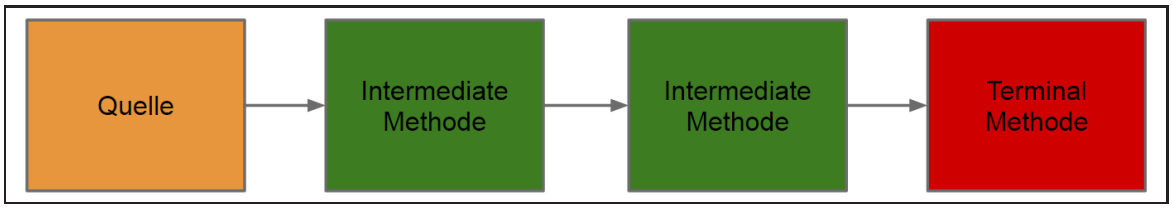


Abb. 17.1: Stream-Methoden

Zusätzlich verarbeiten, verändern oder filtern sie diese und erzeugen aus den neuen Elementen einen neuen Stream. Von diesen Methoden können beliebig viele hintereinander gereiht werden. Intermediate-Methoden können in zwei weitere Kategorien eingeteilt werden: stateless und stateful Methoden. Stateless Methoden wie *filter* und *map* setzen das Prinzip der lazy-evaluation um. Dieses Prinzip wird soweit umgesetzt, dass kein einziges Element aus der Quelle von der erzeugten „Stream-Pipeline“ verarbeitet wird, bis die terminal Methode am Ende der Pipeline aufgerufen wird. Stateful Methoden wie *distinct* und *sorted* setzen das Prinzip der lazy-evaluation nur teilweise oder gar nicht um. Dies kann bei, beispielsweise sorted, so weit gehen, dass vom vorherigen Stream jedes Element verarbeitet sein muss, damit sorted anfangen kann Elemente an die nächste Methode weiter zu geben. Des Weiteren speichern stateful Methoden im Gegensatz zu stateless Methoden Informationen von vorher verarbeiteten Elementen zwischen, um bei folgenden Elementen darauf zurückgreifen zu können.

Das ist auch ein Unterschied zwischen den beiden Methoden iterate und generate der Klasse Stream: Letztere erwartet einen Supplier. Liefern Sie diesen über einen λ-Ausdruck, so ist er stateless. Schreiben Sie eine Klasse und sei es eine annonyme Klasse, so können Sie diesen Supplier statefull gestalten.

Im Folgenden werden einige wichtige Intermediate-Methoden vorgestellt.

Die Methode *map* ist eine der grundlegenden Methoden für die Verarbeitung von Streams. Mit ihr wird auf alle Elemente eines Streams eine Funktion ausgeführt. Der Rückgabewert dieser Methode ist die veränderte Datensammlung. Bekannt ist diese Methode beispielsweise aus funktionalen Programmiersprachen wie Haskell oder dem *map-reduce*-Prinzip.

Definition 17.6.1 (MapReduce)
MapReduce ist ein Programmiermodell mit zugehöriger Implementierung und der Zielrichtung große Datemengen zu verarbeiten. Anwender spezifieren zwei Funktionen (Methoden):

map Verarbeitet key/value-Paare und erzeugt eine Menge von key/value-Paaren als Zwischenergebnis.

reduce Verarbeitet alle key/value-Paare zu einem key-Wert zu einem Ergebnis.

[DG04] ◀

Im Stream Interface Stream<T> ist die Methode mit folgender Signatur definiert:

```
<R> Stream<R> map(Function<? super T,? extends R> mapper)
```

Als Argument wird also eine Funktion übergeben, die die Elemente vom selben Typ <? super T> konsumiert, aus denen der Stream besteht. Produzieren tut diese Methode Elemente vom Typ <? extends R>. Siehe hierhzu auch 18.5.1 auf 231. Die Methode map produziert einen Stream<R>, auf dem wiederum eine beliebige intermediate oder terminal Methode aufgerufen werden kann. Die Methode map gehört zu den stateless Methoden.

Die Methode *filter* gehört zu den grundlegenden Methoden zum Verarbeiten von Datensammlungen. Sie lässt sich zum Filtern von Elementen mit bestimmten Eigenschaften nutzen. Der Rückgabewert dieser Methode ist ein Stream aus Elementen, die der Bedingung genügen. Das Prinzip ist beispielsweise aus SQL mit der WHERE Klausel bekannt.

Im Interface Stream<T> ist die Methode mit folgender Signatur definiert:

```
Stream<T> filter(Predicate<? super T> predicate)
```

Als Argument wird ein Prädikat mit dem Eingabetyp <? super T> gefordert. Die Prädikatsfunktion muss einen *boolean* zurück geben. Die Methode *filter* produziert wiederum einen Stream<T>, also einen Stream vom selben Typ wie der Stream, von dem *filter* aufgerufen wurde. Auch die Methode *filter* gehört zu den *stateless* Methoden.

Mit der Methode *sorted* lässt sich eine Datenquelle sortieren. Dies kann mit der unverarbeiteten Datenquelle, zwischen zwei Bearbeitungsschritten oder zuletzt vor der Ausgabe der verarbeiteten Datenquelle passieren. In der Regel kann eine Vergleichsfunktion angegeben werden, mit der die einzelnen Elemente untereinander verglichen werden. Generell ist es empfehlenswert, dass die Laufzeit der Vergleichsfunktion in $\mathcal{O}(1)$ liegt, da das Sortieren ansonsten sehr viel Laufzeit in Anspruch nehmen kann. Bekannt ist diese Funktion unter anderem aus SQL mit der Klausel ORDER BY.

Im Stream Interface Stream<T> ist die Methode mit den folgenden Signaturen definiert:

```
Stream<T> sorted()
Stream<T> sorted(Comparator<? super T> comparator)
```

sorted ohne Argumente verlangt, dass T das Interface *Comparable* implementiert, ansonsten wird bei Aufruf der terminal Methode am Ende des Streams eine *ClassCastException* geworfen. *sorted* mit einem *Comparator*, also einer Vergleichsfunktion als Argument fordert dies nicht, dafür muss die Vergleichsfunktion selber definiert oder zumindest übergeben werden. Die Vergleichsfunktion muss Elemente vom Typ <? super T> verarbeiten und einen Rückgabewert vom Typ *int* haben.

Eine *terminal* Methode ist das Endstück in der Stream-Pipeline. Wenn eine terminal Methode aufgerufen wurde und terminiert ist, ist der Stream verbraucht und kann nicht mehr benutzt werden. Wie auch intermediate Methoden unterstützen einige terminal Methoden lazy evaluation, wodurch unendliche Streams in vielen Fällen terminieren können. Terminal Methoden konsumieren in der Regel die Elemente des aufrufenden Streams. Nur die terminal Methoden *iterator* und *spliterator* aus dem tun dies nicht, sondern erlauben dem Nutzer das eigenständige Traversieren über die Elemente aus dem aufrufenden Stream.

Die Methode *iterator* Liefert einfach einen Iterator für die Elemente des Streams. *spliterator* anlaog einen *Spliterator*. Ein *Spliterator* ähnelt einem *Iterator*, ermöglicht aber das Aufteilen der Elemente und die Elemente einzeln mittels *tryAdvance* oder alle ab dem aktuellen Element *forEachRemaining* zu verarbeiten.

Terminal Methoden sind verantwortlich dafür, Anfragen an die Stream-Pipeline zu stellen, sodass von der Quelle aus Elemente bis zur terminal Methode durchgereicht werden. Im Gegensatz

zu intermediate Methoden können einige terminal Methoden wie forEach und peek Seiteneffekte haben. Diese sind allerdings wie alle Seiteneffekte mit Vorsicht zu benutzen und können besonders bei parallelen Streams ungewünschte Effekte haben, wenn die aufgerufenen Methoden oder Variablen der Elemente nicht threadsave sind.

Terminal Methoden liefern keinen weiteren zu verarbeitenden Stream, sondern können einzelne Elemente oder Informationen über den Stream als Rückgabewert haben. Beispiele hierfür sind *anyMatch*, *findFirst* oder *match*.

Im Folgenden werden einige wichtige terminal Methoden vorgestellt und Benchmarks für diese gezeigt. Anschließend werden sie mit Java7-Alternativen verglichen.

Das Traversieren über alle Elemente einer Datenquelle um diese Elemente zu manipulieren gehört zu den grundlegenden Methoden der imperativen oder prozeduralen Programmierung. Dies wird in Java u. a. mit der for- oder for-each-Schleife realisisiert. Das entsprechende Konstrukt in der funktionalen Programmierung realisiert die Methode *forEach*, manchmal auch nur „for“ genannt. Auf diese Weise können Aktionen, die für jedes Element einer Datenquelle ausgeführt werden sollen, in wenigen Zeilen und unabhängig von der Menge der Elemente durchgeführt werden. Des Weiteren lassen sich mit dieser Hilfe beliebige Suchen leicht umsetzen. Im Unterschied zur for- oder for-each-Schleife ist die forEach-Methode ggfs. leicht zu parallelisieren.

Im Stream Interface Stream<T> ist die Methode mit der folgenden Signatur definiert:

```
void forEach(Consumer<? super T> action)
```

forEach hat als Argument eine *Consumer*-Methode, welche Elemente vom Typ <? super T> verarbeitet. Als Rückgabewert hat *forEach void*. Das entspricht der Konvention, da die übergebene Methode etwas verändern könnte. Die *Consumer*-Mehode, die als Argument an *forEach* übergeben wurde, kann also Seiteneffekte haben. Bei der Verarbeitung von parallelen Streams ist *forEach* explizit nicht deterministisch. Dies würde eine Einhaltung der Reihenfolge der Elemente fordern, was den Vorteil der Parallelisierung zunichte machen würde. Das heißt, dass für jedes Element die *Consumer*-Methode zu jedem möglichen Augenblick und in jedem möglichen Thread ausgeführt werden kann. Falls die Reihenfolge der Elemente entscheidend ist, kann auf die Methode *forEachOrdered* zugegriffen werden, welche diese beibehält.

Die Methode *reduce* wird dafür verwendet, alle Elemente in einer Datenquelle zu einem einzelnen Element zu reduzieren. Dafür benötigt *reduce* eine Datenquelle und eine Akkumulationsfunktion, um die Elemente der Datenquelle miteinander zu akkumulieren. So können zum Beispiel viele verschiedene Zählungen zu einer Zählung reduziert werden. Bekannt ist diese Methode beispielsweise aus funktionalen Programmiersprachen wie Haskell unter dem Namen *fold* oder aus dem *map-reduce*-Prinzip, siehe Def. 17.6.1.

Im Stream Interface *Stream<T>* ist die Methode mit den folgenden Signaturen definiert:

```
T reduce(T identity, BinaryOperator<T> accumulator)
Optional<T> reduce(BinaryOperator<T> accumulator)
```

Die erste Signatur erwartet ein Identitätselement vom Typ T, welches als Basiselement für die Akkumulationsfunktion genutzt wird. Die Akkumulationsfunktion erwartet zwei Elemente vom Typ T. Als Rückgabewert hat diese Signatur ein Element vom Typ T. Die zweite Signatur erwartet kein Identitätselement und hat ansonsten, wie auch die erste Signatur, eine Akkumulationsfunktion für Elemente vom Typ T als Argument. Als Rückgabewert hat die zweite Signatur im Gegensatz zur ersten ein *Optional<T>*. Dies ist darauf zurück zu führen, dass die erste Signatur durch ihr Identitätselement immer einen Rückgabewert hat, dies aber bei der zweiten Signatur nicht garantiert ist, weil der Stream auch leer sein kann. Keine der Versionen von reduce garantiert eine Abarbeitungsreihenfolge, sodass die Akkumulationsfunktion das Assoziativgesetz einhalten muss. Im Stream<T>-Interface existiert eine speziellere Version von *reduce* namens *collect*. Diese ist dafür gedacht, mit Methoden aus der Klasse *Collectors* aus Stream-Elementen neue Datenstrukturen wie Collections zu generieren. Des Weiteren existiert die folgende Signatur:

```
<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator,
                         BinaryOperator<U> combiner)
```

17.7 Beispiel: Numerische Integration

Siehe hierzu die entsprechenden Folien.

17.8 Historische Anmerkungen

Streams kamen 2014 mit Java 8. In Java 9 kamen vier Methoden zum Interface *Stream* hinzu: *takeWhile*, *ofNullable*, *dropWhile* und *iterate* (überladen, weiterer Parameter vom Typ *Predicate*).

17.9 Aufgaben

1. Ein sogenanntes compostolanisches *Heiliges Jahr* ist, wenn der *25.07.* auf einen Sonntag fällt. Das erste Heilige Jahr wurde im Jahr 1126 begangen. Definieren Sie bitte einen unbegrenzten Stream, der die Heiligen Jahre liefert! Ermitteln Sie bitte aus diesem Stream die Heiligen Jahre im 21. Jahrhundert! Geben sie bitte die ersten 200 entsprechend der aktuellen *Locale*(siehe Kap. 21) formatiert aus!

2. Der 13.05.2016 war ein Freitag. Schreiben Sie bitte einen unbegrenzten Stream, der die folgenden Freitage liefert, die auf den 13. eines Monats fallen!

3.

4.

Kapitel 18

Generics

18.1 Übersicht

Als Programmierer möchte man mit möglichst wenig Code möglichst viel erreichen. So würde man gerne Code schreiben, der für möglichst viele Klassen wiederverwendet werden kann. Ein wichtiges Beispiel sind Container-Klassen. Hier werden bewährte Datenstrukturen implementiert, die für diverse Zwecke effizient eingesetzt werden können. Kann man die Klasse der Objekte in einem Container nicht in irgendeiner Weise einschränken, so muss man damit rechnen, dass sich darin irgendwelche Objekte der Klasse *Object* befinden. Wie Sie in Bem. 18.3.5 sehen werden, könnten Sie dann in einer solchen Klasse für die Elemente nur Methoden der Klasse *Object* verwenden. Dies würde den Programmierer und den Nutzer der Container-Klasse einschränken: Der Programmierer könnte nur die Operationen von *Object* verwenden. Nutzer müssten immer in die Klasse des Objekts casten, die erwartet wird und könnten nicht sicher sein, dass dies klappt.

Generische Klassen geben dem Programmierer die Möglichkeit, in diesen Situationen typsicher zu programmieren. Davon profitieren die Nutzer der Klasse direkt und können sich darüber hinaus noch unnötige Casts sparen.

Der Programmierer oder die Programmiererin kann so möglichst viel der Dokumentation in die Spezifikation des Codes verschieben. Durch die Angabe der möglichen Typparameter macht der Entwickler einer generischen Klasse deutlich, was Anwender mit dieser Klasse tun können und was sie von ihr erwarten können. Programmierer, die die generische Klasse parametrisieren, bringen durch den gewählten Typparameter ihre Absicht bei der Nutzung klar zum Ausdruck. Auch viele Fehler werden früher erkannt: Die Fehlererkennung wird für viele Fälle von der Laufzeit in die Compile-Zeit vorverlagert.

In diesem Kapitel werden generische Elemente in Java systematisch eingeführt. Wichtige Beispiele liefern die Container-Klassen aus dem Paket *java.util*.

Da Sie generisch definiert sind, führe ich auch *enums* in diesem Kapitel ein. Sie als *Aufzählungstypen* zu bezeichnen, wird ihrer Leistungsfähigkeit nicht gerecht. Mit Ihnen können Sie viel mehr machen und ich versuche hier ein bisschen davon zu zeigen.

18.2 Lernziele

- Den Umgang mit den generischen Klassen in Java beherrschen.
- Generische Klassen und Methoden schreiben können.
- Parametrisierte Klassen und generische Methoden professionell verwenden können.
- Die Konsequenzen von type erasure kennen.
- Das factory pattern sachgerecht einsetzen können.

- Aufzählungstypen (enums) kennen und einsetzen können.
- Das singleton pattern sachgerecht implementieren können.

18.3 Grundlagen: Einfache Typparameter

Dieser Abschnitt beschreibt die grundlegenden Syntaxelemente generischer Konstrukte in Java. Ein generisches Element besitzt einen oder mehrere Parameter, die in spitzen Klammern angeben werden: <E>. Hinzu kommen noch weitere Syntaxelemente, die später eingeführt werden.

Definition 18.3.1 (Generische Klasse, einfache Typvariable)
Eine generische Klasse mit einfacher Typ-Variablen oder einfachem Typparameter hat eine Form wie die folgende Klasse *Cache*:

```
public class Cache<T> {
        private final T value;
        public Cache(T v) {
            this.value = v;
        }
        public T getValue() {
            return this.value;
        }
}
```

Der Typparameter kann im Rumpf der Klasse an jeder Stelle verwendet werden, wo sonst ein Referenz-Typ (Klasse oder Interface) stehen könnte. Im Beispiel ist dies der Typ des Parameters im Konstruktor und der Rückgabetyp der Operation *get*. Zu den bestehenden Einschränkungen später mehr. *Cache* hat hier keine *set*-Operation für das Attribut *value*. Dieses kann also nur einmalig angelegt und dann nicht mehr verändert werden. Die Klasse ist also immutable. Um dies auch explizit zum Ausdruck zu bringen, ist das Attribut *value* als *final* spezifiziert. ◀

Bemerkung 18.3.2 (Mutable und defensive copying)
Beispiel 18.3.1 hat potenzielle Schwächen, die gefährlich werden können.

1. Die Deklaration des Attributs als *final* hilft hier nicht, um die Klasse wirksam *immutable* zu machen. Ist die Aufgabe von *Cache*, ein Objekt zu kapseln, auf das dann oft zugegriffen werden soll, so kann das Ziel, die Klasse *immutable* zu machen leicht unterlaufen werden: die Methode *getValue* liefert eine Referenz auf das Attribut *value* zurück. Dies kann also von jeder nutzenden Klasse verändert werden.
2. Soll die Klasse wirklich und wirkungsvoll immutable sein, so müssen weitere Maßnahmen ergriffen werden. Mit der Implementierung aus Beispiel 18.3.1 geht das nicht: *new T* ist nicht möglich, also ist die naheliegende Idee:

   ```
   this.value = new T();
   this.value = v;
   ```

   im Konstruktor oder

   ```
   T result = new T();
   result = this.value;
   return result;
   ```

   syntaktisch falsch (s. u.).

3. Aus der vorstehenden Überlegung folgt: *Cache* kann nur dann sicher funktionieren, wenn *defensive copying* möglich ist. Das geht z. B. so:

```
public class Cache01<T> {
    private final T value;
    private Factory<T> factory;
    public Cache01(T v, Factory<T> f) {
        this.value = v;
        this.factory = f;
    }
    public T getValue() {
        return this.factory.newInstance(value);
    }
}
public interface Factory<T> {
    T newInstance();
    T newInstance(T t);
}
```

oder besser so:

```
public class Cache02<T> {
    private final T value;
    public Cache02(T v) {
        this.value = v;
    }
    public Cache021(Cache01<T> c){
        this(c.value);
    }
    public T getValue() {
        return this.value;
    }
}
```

◀

Verwendet werden generische Klassen als parametrisierte Klassen wie folgt. Für den Typparameter wird ein Referenz-Typ eingesetzt. Hier ein Beispiel für die Verwendung der Klasse *Cache* aus der Def. 18.3.1:

```
Cache<String> stringCache = new Cache<String>("abc");
String s = stringCache.get();
Cache<Integer> intCache = new Cache<Integer>(1);
int i = intCache.get();
Cache<Integer> integerCache = new Cache<Integer>(new Integer(1));
Integer i = integerCache.get();
```

Beim Konstruktor kann der Typparameter weggelassen werden:

```
Cache<String> stringCache = new Cache<>("abc");
Cache<Integer> intCache = new Cache<>(1);
Cache<Integer> integerCache = new Cache<>(new Integer(1));
```

Die leeren spitzen Klammern werden manchmal als *diamond operator* bezeichnet. Der Compiler kann aus der Deklaration den Typparameter herleiten. Das spart nicht immer viel Schreibarbeit, aber es gibt zwei Situationen, in denen diese kleine Änderung in Java 7 Vorteile bringt:

- Wenn der Typparameter lang ist, wie etwa in:

  ```
  List<List<Kunde>> listenListe = new ArrayList<List<Kunde>>();
  ```

  In einer solchen Situation wird der Code übersichtlicher.

- Wird eine Variable mit einem parametrisierten Typ deklariert, kann oder soll aber noch nicht initialisiert werden, so verringert diese reduzierte Redundanz die Fehlermöglichkeiten bei späteren Änderungen.

Aufgrund von Autoboxing können parametrisierte Klassen, die mit einer Wrapper-Klasse deklariert wurden, einfach mit dem primitiven Typ verwendet werden. Sie deklarieren also etwa

```
List<Integer> intList = new LinkedList<>();
```

und schreiben einfach

```
intList.add(42);
int i = intList.get(0);
```

Bemerkung 18.3.3 (Primitive Typen)
Der Typparameter in einer generischen Klassendefinition muss bei Deklaration einer parametrisierten Klasse ein Referenztyp sein. Wollen Sie eine parametrisierte Klasse mit einem primitiven Typ verwenden, so müssen Sie sie mit der zugehörigen Wrapperklasse deklarieren. Bei der Verwendung können Sie dann die primitiven Werte direkt verwenden. Autoboxing sorgt dafür, das dann alles funktioniert, wenn Sie etwas setzen, einfügen etc. Autounboxing erledigt das in der anderen Richtung. ◄

Beispiel 18.3.4 (Generische Klasse)
Gerne als Beispiel für eine einfache generische Klasse wird eine dieser Art genommen:

```
public class Pair<K, V> {
  private K key;
  private V value;
  public Pair(K k, V v) {
    this.key = k;
    this.value = v;
  }
  ...
}
```

Eine konkrete Anwendung hierfür ist die innere Klasse *Entry*, wie sie z. B. in der java.util Klasse *HashMap* verwendet wird. Siehe hierzu auch Abschn. 18.13. Bei privaten inneren Klassen sind auch die Überlegungen aus Bem. 18.3.2 irrelevant. ◄

Eine generische Klasse ohne Typparameter heißt *raw type* (roher Typ). Zwecks Kompatibilität kann ein parametrisierter Typ auch mit dem raw type deklariert und verwendet werden. Der Compiler gibt in diesem Fall aber eine Warnung. Ich rate dringend von der Verwendung von raw types ab! Sie verzichten andernfalls leichtsinnig auf eine Möglichkeit Ihre Intention bei der Programmierung explizit zum Ausdruck zu bringen und viele mögliche Fehler zur Laufzeit frühzeitig auszuschließen.

Es gibt nur eine Rechtfertigung für den Einsatz eines *raw type*, siehe Abschn. 19.3.

Zu einer generischen Klasse gehört genau eine *.class*-Datei. Diese wird für alle parametrisierten Klassen zu dieser generischen Klasse verwendet, ebenso bei Interfaces. Daher kennt die JVM nicht die konkrete Klasse, mit der die parametrisierte Klasse deklariert wurde. In der JVM wird diese Typinformation ausgelöscht (type erasure). In der *.class*-Datei der generischen Klasse steht *Object*, wenn es keine Einschränkung für den Typparameter gibt (s. u.).

Bemerkung 18.3.5 (Einfacher Typparameter)
Ist eine generische Klasse mit einfachen Typparametern T, ..., ohne weitere Einschränkungen definiert, so können in der Klasse als Operationen von T ausschließlich die Methoden der Klasse *Object* verwendet werden. ◄

Definition 18.3.6 (Parametrisierte Klasse)
Eine Klasse, bei der für den Typparameter einer generischen Klasse ein existierender Typ eingesetzt wird, heißt *parametrisierte Klasse*. ◄

Beispiel 18.3.7 (Parametrisierte Klasse)
Es ist guter Stil Elemente mit einem Interface zu deklarieren. Insofern wird die Variable *liste* in diesem Beispiel als *Liste* deklariert. Initialisiert werden muss sie natürlich mit einer konkreten Klasse.

```
Liste<Liste<String>> liste = new LinkedList01<>();
```

◄

Sowohl generische als auch parametrisierte Klassen können spezialisiert werden. Es geht also sowohl

```
public class B<T> extends A<T>{...}
public class SpecialCache<T> extends Cache<T>{...}
```

als auch

```
public class C extends A<Kunde>{...}
public class TicTacToe extends AbstractRegularGame<Pair<Byte, Byte>> {...}
```

Ganz analog können generische Interfaces geschrieben werden.

Zwei parametrisierte Typen $C < A >$ und $C < B >$ mit $A \neq B$ sind nie kompatibel. Auch dann nicht, wenn A Superklasse von B ist. Allerdings ist ein parametrisierter Typ immer kompatibel zum raw type. Aber natürlich ist $C < Object >$ etwas ganz anderes als der raw Typ C. Siehe hierzu *Programmierfehler.generics.Example02*

Die Zuweisung eines Objekts einer spezialisieren parametrisierten Klasse zu einem Objekt einer Oberklasse ist nur zulässig, wenn die Typen exakt übereinstimmen:

```
BaseClass<Number> bc = new SubClass<Number>();  // ok
BaseClass<Number> bc = new SubClass<Integer>();  // Fehler, obwohl Superklasse
```

Diese Tatsache, dass die Typparameter in diesem Kontext exakt übereinstimmen müssen, bezeichnet man als *Invarianz*.

Ebensowenig können Sie eine generische Klasse vom Typparameter spezialisieren:

```
class DoNotExtend<T> extends T {... }
```

Dies kann aus vielen Gründen nicht zugelassen werden. Hier nur einige:

1. T ist ein Typparameter, kein bekannter Typ. Der Versuch, dies durch ein *import T* zu korrigieren, ist sinnlos. Trotzdem schlägt Eclipse dies vor oder tat dies zumindest früher.

2. T hat hier keine Einschränkung. Es könnte also auch eine *final* Klasse sein. Diese dürfte aber nicht spezialisiert werden.

3. Da T keiner Einschränkung unterliegt, kennt die .class-Datei nur den Typ *Object*. Das ist aber keine zusätzliche Spezifikation.

18.4 Einschränkungen für Typparameter

Der Typparameter kann beschränkt werden. Dazu werden eine oder mehrere Einschränkungen (bounds) verwendet:

```
public class Cache<T extends Ausweis>

public class Cache<T extends Ausweis & Serializable>
```

Im ersten Fall können für T nur die Klasse *Ausweis* oder deren Unterklassen verwendet werden. Im zweiten Fall muss T darüber hinaus das Interface *Serializable* implementieren.

Das Schlüsselwort ist immer *extends*, unabhängig davon, ob eine Klasse oder ein Interface folgt. Es kann maximal eine Klasse angegeben werden und diese muss in der Liste als erste Einschränkung erscheinen. Nach dem „&“ können nur weitere Interfaces angegeben werden.

Bei der Implementierung einer generischen Klasse, deren Typparameter durch *extends* nach oben beschränkt ist, stehen nicht nur die Methoden von *Object* zur Verfügung, sondern auch die Methoden der Bounds. Beispiele hierfür zeigen *generics.UseOfBound01* und *generics.UseOfBound02*.

Für die Typauslöschung (*type erasure*) gelten gemäß der Java Sprachspezifikation [GJSB05] und dem aktuellen Stand der Spezifikation für Java 7 vom 18.03.2011 folgende Regeln. Dabei sei $|T|$ der ausgelöschte Typ von T:

Typ	Erasure
$G < T_1, \ldots, T_n >$	$\|G\|$
$T.C$	$\|T\|.C$
$T[\,]$	$\|T\|[\,]$
T extends $T_1 \& \ldots \& T_n$	$\|T_1\|$
T	T

Gibt es Einschränkungen, so steht in der .class-Datei die erste. Mit einem Editor können Sie sich leicht selbst davon überzeugen, wenn er eine hexadezimale Anzeige ermöglicht.

Für die Hierarchie der aus einem generischen Typ (Klasse oder Interface) hervorgehenden parametrisierten Typen gelten einige Regeln, die Sie sich zum größten Teil aus dem bisher Beschriebenen herleiten können.

Seien A und B Klassen oder Interfaces und $G<T>$ ein generischer Typ. Dann sind $G<A>$ und $G<B>$ nicht kompatibel, auch nicht, wenn etwa A eine Unterklasse von B ist. Dies wird durch *Programmierfehler.generics.Example02* illustriert (A entspricht Fuehrerschein, B entspricht Ausweis):

```
16   List<Ausweis> ausweisList = new ArrayList<>();
17   List<Fuehrerschein> fuehrerscheinList = new ArrayList<>();
18
19   ausweisList = fuehrerscheinList;
20   fuehrerscheinList = ausweisList;
```

Da *Fuehrerschein* eine Unterklasse von *Ausweis* ist, kann die Zuweisung in Zeile 20 nicht funktionieren. Aber sowohl die Zuweisung in Zeile 19 als auch die in Zeile 20 können grundsätzlich nicht funktionieren. Zum einen verstößt dies gegen das ober erwähnte Prinzip der Invarianz. Nun ist umgangsprachlich eine Liste von Führerscheinen sicherlich eine Liste von Ausweisen und man könnte man sich natürlich vorstellen, dass dies auch in Java umgesetzt werden könnte. Wie aber z. B. das Beispiel in Abb. 2.3 aus [Ess08] zeigt, könnte man dann Mehrfachvererbung erreichen und das geht in Java eben nicht, wie Abb. 18.1 zeigt.

Eine parametrisierter Typ $G<A>$ ist kompatibel mit dem raw type G. Ein Beispiel hierfür zeigt *Programmierfehler.generics.Example03*. Natürlich gibt es eine Warnung bei der Deklaration der Variablen mit dem raw type und die ist auch gerechtfertigt.

$G<T>$ ist äquivalent zu $G<T$ *extends Object*$>$. Eine bound *Object* wird aber genausowenig angegeben wie eine Oberklasse *Object*.

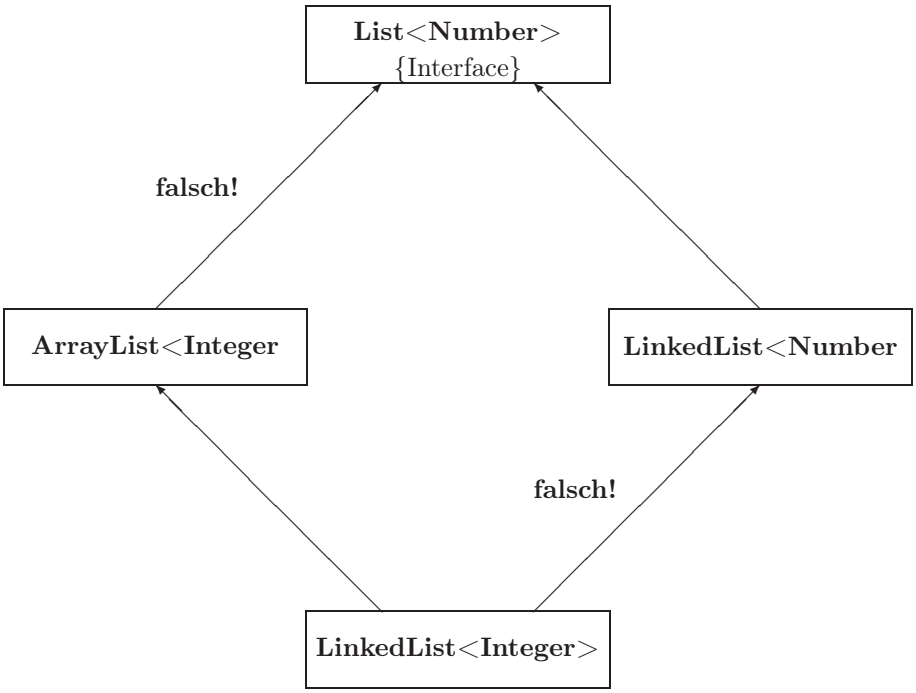


Abb. 18.1: Diamonds are not programmers best friends

18.5 Wildcards

Eine besondere Rolle spielen Wildcards „?“ im Kontext von parametrisierten Typen. Bisher wurden parametrisierte Typen erläutert, bei denen für den Typparameter einer generischen Klasse ein konkreter Referenztyp eingesetzt wird. Das ist manchmal unflexibel. Eine höhere Flexibilität kann durch Wildcards erreicht werden. Eine Reihe der Restriktionen beim Einsatz generischer Elemente wird durch Wildcards abgemildert. Eine Wildcard wird durch eine Fragezeichen „?“ symbolisiert und steht an Stelle eines einfachen aktuellen Typs. In einem parametrisierten Typ kann als Typparameter also ein Referenztyp oder eine Wildcard verwendet werden. Außerdem können Wildcards in generischen Methoden und Klassen verwendet werden, aber in genau diesem Kontext.

Die folgenden Beispiele finden Sie in *generics.wildcard.WildCardUse* im Projekt *Programmierbeispiele* bzw. *Programmierfehler*.

Im einfachsten Beispiel (Invarianz) ist

```
C<A> ca = new C<A>();
C<B> cb = new C<B>();
C<?> c0 = ca;
C<?> c1 = cb;
```

Ist eine parametrisierte Variable mit einer mit einer Wildcard deklariert (wie in *C<?> c0 = ca;*), so kann ihr jede mit einem konkreten Typ parametrisierte Variable zugewiesen werden.

Auch der zweiten Ebene des Typparameters kann eine Wildcard verwendet werden:

```
List<C<?>> list = new ArrayList<C<?>>();
list.add(ca);
list.add(cb);
```

In diesem Fall kann der Liste *list* jedes Objekt einer Klasse *C<K>*, *K konkreter Typ* hinzugefügt werden.

Auf der ersten Ebene ist dies auch bei *Collections* möglich, wie hier:

```
List<?> l0 = new ArrayList<>();
```

Dies hat aber Konsequenzen: Dies bewirkt einen Schreibschutz: In diese Liste können nur Objekte geschrieben werden, die von jedem Typ sind und dass ist nur das *null*-Literal. Die Zeile

```
l0.add(null);
```

wird anstandslos compiliert. Lässt die Implementierung *null*-Objekte zu, wie hier *ArrayList*, so funktioniert das auch zur Laufzeit. Akzeptiert die Implementierung keine *null*-Objekte, so gibt es aber zur Laufzeit eine *NullPointerException*. Dies ist z. B. bei der Klasse *ConcurrentLinkedDeque* aus dem Paket *java.util.concurrent* der Fall.

Das hinzufügen eines nicht-*null*-Objekts liefert einen Compiler-Fehler, wie im Beispiel:

```
C<A> ca = new C<A>();
l0.add(ca);
```

Eine Wildcard ist aber etwas anderes als ein *Object*: Die zweite der folgenden Zeilen produziert im Unterschied zur entsprechenden Zeile im Beispiel weiter oben einen Compiler-Fehler:

```
C<A> ca = new C<A>();
C<Object> o = ca;
```

„Type Mismatch: can not convert from C<A> to C<Object>“, was wegen des Prinzips der Invarianz auch so sein muss. Aber es zeigt auch, wie Wildcards die Flexibilität erhöhen.

Einige Dinge gehen aber auch mit Wildcards nicht. Die Klasse C ist dabei ganz einfach:

```
public class C<T> {
   private T t;
   public void set(T t){
      this.t = t;
   }
   public T get(){
      return this.t;
   }
}
```

Von den folgenden drei Zeilen aus *WildCardUse* im Projekt *Programmierfehler* ist nur die erste (22) syntaktisch korrekt:

```
22  C<?> c = ca;
23  c.set(new A());
24  A a = c.get();
```

Zeile 23 ist illegal, weil der parametrisierte Typ „?“ ist, also irgendein Typ. Von dem kann im Unterschied zu *A* nicht garantiert werden, dass er die Methode *set(A a)* hat. Die Zeile 24 ist illegal, weil für c nur bekannt ist, das der Typparameter irgend ein *Object* ist. Es kann aber nicht zugesichert werden, dass es vom Typ *A* ist. Mit der Deklaration *Object* geht das aber:

```
Object o = c.get();
```

Eine weitere häufige Anwendung ist die Beschränkung einer Wildcard „nach oben.“ Das erste Beispiel bringt keine neuen Erkenntnisse:

```
C<A> ca = new C<A>();
C<? extends A> cSuperA;
cSuperA = ca;
```

Interessant ist erst das nächste Beispiel:

```
C<B> cb = new C<B>();
C<? extends A> cSuperA;
cSuperA = cb;
```

Die letzte Zeile liefert nun einen Compiler-Fehler: *cSuperA* können nur Objekte mit der Klasse *A* oder ihrer Unterklassen als Typparameter zugewiesen werden.

Wie Sie bereits weiter oben gesehen haben ist die Deklaration *C<?>* äquivalent zu *C<? extends Object*.

Auch die folgenden Zeilen sind alle korrekt (*SubClass* ist eine Unterklasse von *SuperClass*) (siehe *generics.wildcard.ExtendsBoundedWildcards*)

```
15  C<SuperClass>   ca = new C<>();
16  C<SubClass> cb = new C<>();
17  ca.set(new SubClass());
18  ca.set(new SuperClass());
19  cb.set(new SubClass());
20
21  C<? extends SubClass> c0 = cb;
22  C<? extends SuperClass> c1 = c0;
23  C<?> c2 = c1;
24  SubClass b = c0.get();
25  SuperClass a = c1.get();
26  Object o = c2.get();
```

Die Zeilen $15 - 19$ legen nur die Basis. Wildcards kommen im zweiten Block dazu.

In den Zeilen $21 - 22$ ist in Zeile 22 die Bound weiter als die in Zeile 21. Also funktioniert die Zuweisung, da *SubClass* eine Unterklasse von *SuperClass* ist. In Zeile 23 ist die implizite Bound *Object*, also funktioniert auch diese Zuweisung. Die Zeilen $24 - 25$ funktionieren, da durch die Bounds in den vorstehenden Zeilen sichergestellt ist, dass der aktuelle Typ mindestens *SubClass* bzw. *SuperClass* ist. Zeile 26 kennen wir schon, das geht immer.

Diese Form der Einschränkung nennt man *kovariant*.

Es geht aber auch „andersherum“: Dazu betrachten wir eine weitere Klasse, die Unterklasse *SubSubClass* von *SubClass* und eine generische Klasse *D<T>*. Dann funktioniert Folgendes:

```
28   D<SuperClass> da = new D<>();
29   D<SubClass> db = new D<>();
30   D<SubSubClass> dc = new D<>();
31
32   D<? extends SubClass> d0;
33   d0 = db;
34   d0 = dc;
```

Die Zuweisungen in den Zeilen $33 - 34$ funktionieren, da in beiden Fällen sichergestellt ist, das der Typ von *d*0 ein mit *SubClass* parametrisierter Typ ist.

Folgendes geht aber nicht:

```
d0 = da;
```

Hier ist ja *SuperClass* keine Unterklasse von *SubClass*.

Dreht man die Richtung aber um und verwendet *super* statt *extends*, so drehen sich die Verhältnisse um (siehe *generics.wildcard.SuperBoundedWildcards*).

```
D<? super B> d1;
d1 = da;
d1 = db;
```

Nun geht natürlich

```
D<? super SubClass> d1;
d1 = dc;
```

nicht, denn *SubSubClass* ist keine Oberklasse von *SubClass*. Ebenso falsch ist

```
C<? super SuperClass extends SuperClass> c;
```

Man könnte vermuten, dass die Wildcard dadurch auf *SuperClass* eingeschränkt würde. Das ist aber einfacher ohne Wildcard zu erreichen. Diese Form der Einschränkung mit *super* nennt man *kontravariant*.

Mit Ko-, Kontra- bzw. Bivarianz bezeichnet man die Richtung der Übertragung von Eigenschaften in einer Hierarchie. Bei *kovariantem* Verhalten überträgt sich die Eigenschaft von einem Element nach unten in der Hierarchie, bei *kontravariantem* in der Hierarchie nach oben und bei *bivariantem* in beide Richtungen.

Diese drei Möglichkeiten gibt es auch bei Wildcards:

kovariant `GenType<? extends ActualType>`. Jeder Subtyp von `ActualType` kann an Stelle von „?“ verwendet werden.

kontravariant `GenType<? super ActualType>`. Jeder Supertyp von `ActualType` kann an Stelle von „?“ verwendet werden.

bivariant `GenType<?>`. Jeder Typ kann verwendet werden.

Haben wir Klassen `class B extends A` und `class C extends B`, so geht etwa Folgendes:

```
Cache<A> ca = new Cache<A>;
Cache<B> cb = new Cache<B>;
Cache<C> cc = new Cache<C>;

Cache<? extends B> c0;

c0 = cb;
c0 = cc;
```

nicht aber

```
c0 = ca;
```

Anders herum mit super: Dann geht

```
Cache<A> ca = new Cache<A>;
Cache<B> cb = new Cache<B>;
Cache<C> cc = new Cache<C>;

Cache<? super B> c0;

c0 = ca;
c0 = cb;
```

nicht aber

```
c0  = cc;
```

Aus [Ess08] Hinweis 2.23–24:

- Eine ungebundene Wildcard ersetzt Typ-Variable für variierende unbekannte Typen in parametrisierten Ausdrücken.
- Der Wildcard-Ausdruck `SomeType<?>` ist der Supertyp aller parametrisierten Typen der Form `SomeType<T>`, wobei T für einen beliebigen Typ steht.

- Ein Objekt, das durch eine ungebundene Wildcard repräsentiert wird, kann nur als Typ *Object* gelesen und mit *null* geschrieben werden.
- Der Wildcard-Ausdruck `SomeType<? extends BaseType>` ist der Supertyp für alle Typen `SomeType<SubtypeOfBaseType>`
- Eine Instanz, die durch `? extends BaseType` repräsentiert wird, kann nur als Typ BaseType gelesen und mit *null* geschrieben werden.

Einige Dinge gehen aber auch nicht.
So kann das folgende Programmsegment nicht umgewandelt werden:

```
List<String>sLst = new ArrayList<String>();
dLst instanceof List<Double> ? ... : ...;
```

Wegen type erasure kann das nicht funktionieren, sondern höchstens

```
dLst instanceof List ? ... : ...;
```

Statische Elemente können nicht generisch sein. Folgendes geht also nicht:

```
class NoStaticGenVar<T> {
   static T t;  // NO
   static void foo(T t) {} // NO  }
```

Das folgt ganz logisch aus der Definition von Klassenelementen und dem Grundprinzip der type erasure: Ein Klassenattribut gibt es nur einmal pro Klasse. Wegen type erasure kann dies nur einen Typ, nämlich *Object* oder die erste Einschränkung haben.

In einigen Fällen brauchen Sie Wildcards auch in generischen Klassen. Das ist etwa der Fall, wenn der Typparameter *Comparable* implementieren muss. Die Klassendefinition beginnt dann so:

```
public class OrderedList<T extends Comparable<? super T>>{
```

Hier muss es „Comparable<**? super T**>“ heißen: Nur so funktioniert der Code zur Laufzeit, wenn nicht der konkrete Typ, der für den Typparameter eingesetzt wird, *Comparable* implementiert, sondern dies bereits in einer Oberklasse geschehen ist.

Bemerkung 18.5.1 (PECS)
Für die Verwendung von Wildcards mit *extends* und *super* gilt die Faustregel **PECS**

Producer extends, Consumer super

[Blo08] ◄

18.6 Generische Methoden

Nicht immer braucht man eine generische Klasse. In vielen Fällen möchte man auf einfache Art und Weise eine Operation für viele oder alle Typen implementieren können.

Definition 18.6.1 (Generische Methode)
Eine generische Methode ist eine Methode einer (nicht notwendig generischen) Klasse, die einen Typparameter als Rückgabetyp oder Parameter hat. ◄

Wenn etwas ganz allgemein gilt, so kann man das für Objekte implementieren und hat die Operation damit für Objekte aller Klassen zu Verfügung. Eine Operation ist aber nicht immer einem Objekt zuzuordnen: Welches der möglicherweise vielen Objekte einer Klasse sollte etwa für das Sortieren von Objekten dieser Klasse verantwortlich sein? Hier brauchen wir eine statische Methode in einer geeigneten Klasse. In Java ist dies etwa die Klasse *Collections* mit den sort-Methoden in Beispiel 18.6.2

Beispiel 18.6.2 (Generische Methode)
Die erste Methode akzeptiert eine Liste von Objekten, die das Interface *Comparable* implementieren.

```
static <T extends Comparable<? super T>> void sort(List<T> list)
```

Die zweite Methode akzeptiert eine Liste und ein *Comparator*-Objekt.

```
static <T> void sort(List<T> list, Comparator<? super T> c)
```

◀

In Beispiel 18.6.2 sehen wir bereits zwei weitere Syntax-Elemente:

1. Der Typparameter kann beschränkt sein: z. B. *T extends SuperKlasse*. Dies ist hier im ersten Beispiel der Fall.
2. Auch bei Wildcards kann *extends* verwendet werden. Ein einfaches Beispiel aus der Utility-Klasse *Collections* zeigte die erste obige Methode. Es geht hier aber auch in der anderen Richtung: *extends* definiert eine obere Schranke für den Typparameter in einer Klassen-Hierarchie. Das Schlüsselwort *super* definiert eine untere Schranke für den Typparameter in einer Klassen-Hierarchie, wie bereits dieses Beispiel zeigt.
3. Es können *Wildcards* verwendet werden. So heißt es in Beispiel 2 *Comparator<? super T>*. Es kann also ein Comparator für irgendeine Klasse verwendet werden, die Oberklasse von T ist.

Die Verwendung von *super* im letzten Punkt der vorstehenden Aufzählung ist durchaus typisch: Bei Verwendung dieser Klassenmethode *sort* muss oder kann auch ein Comparator für eine Oberklasse des verwendeten Typparameters übergeben werden. Ohne die nach unten beschränkte Wildcard würde das wegen des Grundprinzips der Invarianz parametrisierter Typen nicht funktionieren.

Auch hier sehen Sie wieder das in Bem. 18.5.1 genannte Prinzip.

Wie immer ist auch bei generischen Methoden in generischen Klassen Sorgfalt geboten, wie das folgende Beispiel aus *Programmierfehler.generics* zeigt:

Beispiel 18.6.3 (Generische Methode in generischer Klasse)
Die folgende Klasse *GenClassGenMethod*

```
public class GenClassGenMethod<T> {
   private T t;
   ...
   <T> T doSomething(T arg){
            return t;
  }
}
```

ist generisch und hat die generische Methode *doSomething*. Aber der Typparameter *T* der Methode verdeckt den Typparameter der Klasse. Das führt zum Compilerfehler „Type mismatch: cannot convert from T to T“. In solchen Fällen muss der Typparameter der Methode anders benannt werden. ◀

Bei generischen Methoden steht der Typparameter vor dem Rückgabetyp. Etwaige Bounds müssen also auch dort angegeben werden. Ein Beispiel liefern Methoden aus *java.util.Collections* wie

```
public static <T extends Comparable<? super T>> void sort(List<T> list) { ...
```

Sie können hier *nicht* deklarieren:

```
public static <T> void sort(List<T extends Comparable<? super T>T> list) { ...
```

18.7 Verwendung parametrisierter Elemente

Den ersten Kontakt mit generischen Klassen und Methoden haben Sie, wen Sie die Java Collection-Klassen, z. B. aus *java.util* verwenden. Auf die „Feinheiten“ brauchten Sie dabei zunächst noch nicht zu achten. Nun ist es ab an der Zeit dies genauer zu erläutern.

1. Manche Dinge können nicht für beliebige Typen verwendet werden. So kann etwa die Methode *Collections.sort(List<T> list)* nur für solche Typen T verwendet werden, die das Interface *Comparable<T>* erweitern bzw. implementieren.

2. Lesen Sie die Java API Dokumentation sorgfältig!

Das folgende Beispiel verdanke ich Lars Harmsen aus dem SS 2011.

Beispiel 18.7.1 (Collections.copy)
Die Utility-Klasse Collections hat eine Methode copy mit folgender Signatur

```
public static <T> void copy(List<? super T> dest, List<? extends T> src)
```

In der API Dokumentation steht dazu Folgendes:

> Copies all of the elements from one list into another. After the operation, the index of each copied element in the destination list will be identical to its index in the source list. The destination list must be at least as long as the source list. If it is longer, the remaining elements in the destination list are unaffected.

Hat man nun eine Liste so könnte man auf folgende Idee kommen:

```
List <IWagon> waggons = new ArrayList <IWagon> (this.waggons.size());
Collections.copy(waggons, this.waggons);
```

Das geht aber ggf. gnadenlos schief, d. h. führt zu einer *IndexOutOfBoundsException*. Das ist aber mit einem Blick in die API Dokumentation leicht zu erklären. Dort steht klar und deutlich „The destination list must be at least as long as the source list.“. Die Methode *size()* einer Liste liefert aber die Anzahl Elemente der Liste nicht die Länge oder genauer die Kapazität. In diesem Fall ist die Kapazität die aktuelle Länge des Array in der Klasse ArrayList! Wurde die Liste wie in diesem Fall als ArrayList mit der default Kapazität von 10 angelegt und enthält sie weniger als 10 Elemente, so wird die Destination List mit dieser kleineren Größe angelegt. Folge: Die genannte Exception! Die Lösung ist hier ganz einfach: Nehmen Sie einen anderen Konstruktor von ArrayList:

```
List <IWagon> waggons = new ArrayList <IWagon> (this.waggons);
```

◀

18.8 Verwendung generischer Elemente

Generische Klassen können weitgehend wie nicht-parametrisierte eingesetzt werden. Zum Verständnis der Regeln und ggf. Compiler-Warnungen oder -Fehler müssen Sie aber wissen, wie Generics in Java funktionieren.

Ein ganz wichtiger Punkt dabei ist die Kompatibilität mit früheren (prä 1.5) Java-Versionen. Vor Java 1.5 gab es keine Typparameter: Die entsprechenden Klassen waren so geschrieben, dass sie nicht Objekte eines spezifischen Typs T verwenden, sondern Objekte vom Typ (der Klasse) *Object*. Um Code dieser Art auch ab Java 1.5 noch zu unterstützen findet etwas statt, was als *type-erasure* (Typ-Auslöschung) bezeichnet wird. Der Compiler ersetzt an jeder Stelle, wo T steht, T durch *Object*. Dies ist der sogenannte raw-Type einer generischen Klasse *Fu<T>*.

Dies hat Konsequenzen für die Entwickler und die Nutzer generischer Klassen:

1. Da der Typ T ausgelöscht wird und zur Laufzeit nur Klassen mit einer konkreten Klasse, wie *Kunde* vorkommen, kann man in einer generischen Klasse nicht schreiben:

```
T meinT = new T();
```

Schließlich kennt die .class-Datei nur den Typ *Object* bzw. die erste bound.

2. Genausowenig können generische Arrays erzeugt werden.

Legen Sie etwa eine neue Liste von Kunden mit

```
List<Kunde> kundenListe = new LinkedList<Kunde>();
```

an, so wird an jeder Stelle, wo in der Klasse *LinkedList*$<T>$ T steht, die Klasse *Kunde* eingesetzt. Ein solches Attribut könnte etwa eine Richtung einer ?:*-Assoziation implementieren.

Generische Klassen können alle Elemente wie andere Java-Klassen haben. In vielen Containern in *java.util* finden Sie z. B. innere Klassen.

Müssen Sie in einer generischen Klasse Elemente vom Typ des Typparameters erzeugen, so gibt es dafür folgende Wege, die alle ihre Vor- und Nachteile haben.

1. Sie deklarieren das Element mit dem Typ *Object* bzw. der engsten Bound, wie in

   ```
   private Object data;
   ```

   Methoden, die das Attribut zurückliefern müssen, dann auf T casten. Aber nach Bemerkung 18.8.2 ist das keine wirkliche Lösung, wie die unterdrückte Warnung zeigt (Siehe *Programmierbeispiele.generics.GenericElement01*). Dieser Weg wird z. B. in *java.util.ArrayList* gewählt.

   Diese Form der Deklaration ist aber sicher: Zwar dient der Cast auf T vor allem der Beruhigung des Compilers. Er ist aber auch sicher, da durch die Deklaration der Einfüge-Operation sichergestellt ist, dass nur Objekte vom Typ T eingefügt werden können.

2. Sie deklarieren das Element mit dem Typparameter, wie in

   ```
   private T data;
   ```

   Um das Attribut z. B. im Konstruktor neu anzulegen, erzeugen Sie ein neues *Object* und casten dann auf den Typparameter. Wie die unterdrückte Warnung zeigt, ist das auch keine wirkliche Lösung, wie Beispiel 18.8.1 zeigt. (Siehe *Programmierbeispiele.generics.GenericElement02*). Tatsächlich erfolgt ja kein Cast auf den Typparameter sondern auf *Object* bzw. die engste bound.

3. Sie schreiben nur einen Konstruktor, dem das Element übergeben wird. Nutzer können dieses Objekt erzeugen, da sie den konkreten Typ festlegen. Das Problem wird also auf den Nutzer verlagert (Siehe *Programmierbeispiele.generics.GenericElement03*).

4. Sie deklarieren das Attribut mit dem Typparameter (T) und verwenden das Factory pattern (Fabrikmuster):

   ```
   public GenericElement04(Factory<T> factory) {
      this.data = factory.create();
   }
   ```

   Das Interface *Factory* hat hier nur eine Operation,

   ```
   interface Factory<T>{
      T create();
   }
   ```

die Nutzer z. B. in einer anonymen Klasse implementieren. Auch bei dieser Lösung wird das Problem auf die Nutzer der generischen Klasse verlagert (Siehe *Programmierbeispiele.generics.GenericElement04*).

Ganz analog können Sie natürlich auch eine *ArrayFactory* deklarieren, wenn Sie ein Array vom Typparameter benötigen.

Beispiel 18.8.1 (Warning)
Der folgende Code von Sebastian Kurt und Stephanie Böhning aus dem SS 2010 zeigt, dass die Warnung *unchecked cast* nicht einfach ignoriert werden darf:

```
public class QueueRing<E extends Comparable<E>> implements IQueue<E> {
private E[] ringBuffer;
...
 public QueueRing(int size) {
    this.ringBuffer = (E[]) new Object[size];
    this.head = 0;
    this.tail = 0;
    this.count = 0;
}
...
```

Rufen Sie nun diesen Konstruktor auf:

```
IQueue<String> queueString = new QueueRing<String>();
```

so erhalten Sie:

```
Exception in thread "main" java.lang.ClassCastException:
[Ljava.lang.Object; cannot be cast to [Ljava.lang.Comparable;
```

Daran erkennen Sie auch, dass nicht auf `E`, sondern auf die engste Bound (Comparable) gecasted wird, siehe Bemerkung 18.8.2.

Das Problem liegt hier allerdings an einer anderen Stelle. Die Deklaration der Klasse müsste lauten:

```
public class QueueRing<E extends Comparable<? super E>> implements IQueue<E> {
```

◀

Schon wieder: 18.5.1.

Bemerkung 18.8.2 (Cast nach Typ-Variable)
Nach [Ess08] Hinweis 2.21 führt ein Cast auf den Typparameter T nicht auf T, sondern auf *Object* oder die erste Einschränkung (bound) . ◀

Ebenso, wenn Sie ein Array von *T*s benötigen.

```
(T[])(new Object[n])
```

Dann verwenden Sie ein Factory mit einer *create*-Methode, die die Größe des Arrays als Parameter erhält. In *Bug 5098163* wird gefordert, letzteres zu ändern und das Erzeugen von Arrays von t zu ermöglichen. Genauer ging es dort um *Reification* generischer Elemente. Diese Frage wird als Enhancement Request mit niedriger Priorität behandelt.

In aller Regel werden Sie parametrisierte Typen mit dem jeweiligen konkreten Typparameter verwenden und nicht mit dem *raw type*. Eine wichtige Abweichung hiervon betrifft *instanceof*. Der *instanceof* Operator erlaubt auf der rechten Seite keinen parametrisierten Typ, sondern nur *raw* Types. Sie können also ganz im Java prä 1.5 Style abfragen:

```
List<Kunde>kundenliste = new LinkedList<Kunde>();
kundenliste instanceof List;
```

nicht aber

```
List<Kunde>kundenliste = new LinkedList<Kunde>();
kundenliste instanceof List<Kunde>;
```

So wissen Sie also immerhin, dass es sich um eine Liste handelt, können aber den Typ der Objekte nicht sicherstellen (s. a. Abschn. 19.3). Dies muss dann anschließend noch geschehen.

18.9 Die Java Collection-Klassen

Java stellt eine Reihe generischer Collections zur Verfügung. Die Basisschnittstelle für alle Collections ist das Interface *Collection*.

boolean add(E e) Eine optionale Operation, die garantiert, dass nach ihrer Ausführung die *Collection* das spezifizierte Element enthält.

boolean addAll(Collection<? extends E> c) Eine optionale Operation, die alle Elemente der spezifierten *Collection* dieser *Collection* hinzufügt. Diese Operation ist z. B. bei *Collection*s sinnvoll, die sich nicht automatisch bei Bedarf vergrößern. Bei diesen muss der Benutzer dies ja leisten und da kommt eine solche Operation gerade recht.

void clear() Eine optionale Operation, die alle Elemente der *Collection* entfernt.

boolean contains(Object o) Liefert *true*, wenn die *Collection* das spezifizierte Element enthält.

boolean containsAll(Collection<?> c) Liefert *true*, wenn die *Collection* alle Elemente der spezifizierten *Collection* enthält.

boolean equals(Object o) Vergleicht das übergebene Objekt darauf, ob es gleich dieser *Collection* ist.

int hashCode() Liefert den Hashcode für diese *Collection*.

boolean isEmpty() Liefert *true*, wenn die *Collection* keine Elemente enthält.

Iterator<E> iterator() Liefert einen *Iterator* über die Elemente der *Collection*.

boolean remove(Object o) Eine optionale Operation, die eine einzelne Instanz des spezifizierten Elements entfernt, so eine solche existiert.

boolean removeAll(Collection<?> c) Entfernt alle Elemente aus der *Collection*, die auch Elemente der übergebenen *Collection* sind.

boolean retainAll(Collection<?> c) Eine optionale Operation, die die Elemente aus der *Collection* entfernt, die in der spezifizierten *Collection* enthalten sind.

int size() Liefert die Anzahl der Elemente in dieser *Collection*.

Object[toArray()] Liefert ein Array, dass alle Elemente der *Collection* enthält.

<**T**> **T[** toArray(T[] a)] Liefert ein Array, dass alle Elemente der *Collection* enthält. Der Typ ist der Typ der parametrisierten *Collection*.

18.10 Set

Das Interface *Set<T>* aus *java.util* beschreibt eine Menge. Insbesondere gibt es in einem *Set* keine doppelten Objekte. Allerdings sind viele der dort spezifizierten Methoden *optional*.

An diesem Beispiel weise ich aber auf einige Dinge hin, auf die Sie unbedingt achten müssen. Tun Sie das nicht, so laufen Sie Gefahr schwer zu entdeckende Laufzeitfehler zu verursachen.

Schreiben Sie eine Klasse, die das Interface *Set* implementiert, so müssen Sie damit rechnen, dass der Typparameter *mutable* ist. Da *Set* ein Sub-Interface von *Collection* ist, hat auch Ihre Klasse eine Methode *toArray*. Liefern Sie hier die Elemente des *Set*s als Array-Elemente zurück, so können Nutzer die Elemente verändern. Sie haben dann keine Kontrolle darüber, dass Elemente so verändert werden, das im Ergebnis mehrere gemäß *equals* gleiche Elemente im *Set* enthalten sind. Die Spezifikation der *toArray*-Methoden schreibt daher vor, dass dies sicher ausgeschlossen werden muss.

Analoge Überlegungen gelten für die anderen Methoden, die Objekte zurück liefern. In diesen Fällen sollten Sie also Kopien zurückgeben und nicht die Elemente selber (siehe hierzu auch Item 50 (Make defensive copies when needed) in [Blo18]). Umgekehrt müssen Sie beim Einfügen eine Kopie einfügen: Denn sonst könnte das Element ja ebenfalls verändert werden. Ob das in einer Anwendung erforderlich ist, müssen Sie auf Grundlage der Anforderungen entscheiden.

Das *List* Interface bietet eine Reihe überladener Fabrikmethoden *of* zur Erzeugung von immutable *List*s. Die so erzeugten Listen erlaube kein Hinzufügen, Löschen oder Ersetzen. Sie *null* werfen eine *NullPointerException*, wenn versucht wird *Null* einzufügen. Ferne sind diese Listen wertbasiert (value-based).

Diese Fabrikmethoden erweitern ein Angebot von *of*-Methoden, die es bereits in der Klassen *EnumSet* gab. Die dort erzeugten Sets von Enums sind aber mutable.

18.11 Comparable und Comparator

Das Interface *Comparable* definiert durch die (einzige) Methode *compareTo* eine vollständige Ordnung auf den Objekten einer Klasse, die *Comparable* implementiert:

$$x <= y \iff x.compareTo(y) <= 0$$

Das Interface *Comparator* deklariert die Methoden *compare* und *equals*. Ein *Comparator* macht einem z. B. das Leben einfacher, wenn man einen Heap implementiert: Man definiert einen Comparator, etwa für einen Min-Heap und definiert den anderen einfach als InverseComparator

```
import java.util.Comparator;

public class InverseComparator<T extends Comparable<T>> implements Comparator<T> {
public int compare(T t1, T t2) {
return t2.compareTo(t1);
}
}
```

Siehe hierzu auch das Thema „Anonyme Klassen“ in Abschn. 9.9.

In Abschn. 18.14 werden Sie eine weitere Implementierungsmöglichkeiten kennen lernen, die heute in vielen Fällen als die im Zweifel zu wählende gilt. In anderen Fällen verwenden Sie einfach einen Lambda-Ausdruck.

18.12 Geordnete Collections

Java stellt auch eine Reihe geordneter Collections zur Verfügung. So gibt es die Interfaces *SortedSet* *SortedMap*, die von Klassen wie *ConcurrentSkipListMap*, *TreeMap*, *ConcurrentSkipListSet*, *TreeSet* implementiert werden.

18.13 Generische Klassen und Methoden

In den vorstehenden Abschnitten wurde in die Nutzung der aktuellen Java Collection- oder Containerklassen eingeführt. In diesem Abschnitt führe ich nun in die Programmierung eigener generischer Elemente ein.

Innerhalb einer generischen Klasse kann es alle Elemente geben, die es in anderen Klassen auch geben kann:

- Attribute, auch Klassenattribute, allerdings keine Klassenattribute von einem Typ eines der Typparameter (type erasure).
- Methoden, auch Klassenmethoden.
- Innere Klassen.

Ein Beispiel für eine innere Klasse liefert ein verkettete Liste. Das Konzept zeigt Abb. 18.2

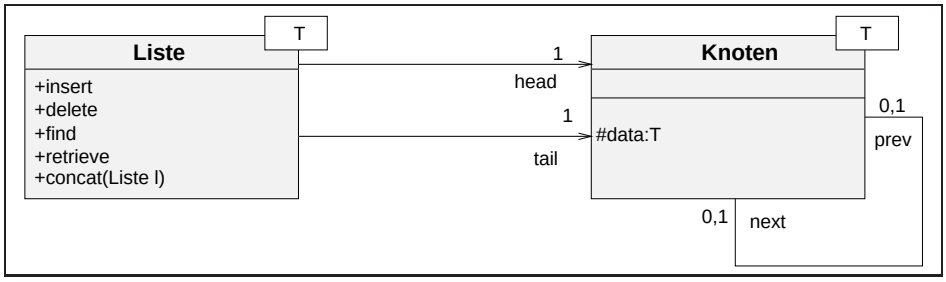


Abb. 18.2: Doppelt verkette Liste mit Head und Tail

Die Klasse *Knoten* wird üblicherweise als innere Klasse implementiert, da Sie nur innerhalb der Klasse *Liste* benötigt wird. Hier muss aber auf einige wenige Dinge geachtet werden. Ich beginne mit einer zunächst naheliegenden Variante:

```
public class LinkedList<E> {
   private Entry<E> head;
   private Entry<T> tail;
     ...
   private class Entry<E>{
      E element;
      Entry<E> next;
      Entry<E> previous;
      Entry(E element, Entry<E> next, Entry<E> previous) {
          this.element = element;
          this.next = next;
          this.previous = previous;
      }

   }
}
```

Dieser Codeausschnitt bringt noch jede Menge Warnungen, dass Elemente nicht benutzt werden. Diese interessieren mich hier nicht. Es gibt aber eine Warnung dort, wo die Definition der inneren Klasse *Entry<T>* beginnt. Die Warnung lautet: *The type parameter T is hiding the type T.* Damit ist die Typsicherheit potenziell gefährdet, die Generics bieten sollen. Deshalb ist die entsprechende Klasse in der Klasse *LinkedList* als *static* deklariert. Dann kann auch der Typparameter an dieser Stelle ohne Gefahr verwendet werden. Außerdem sind statische geschachtelte Klassen keine inneren

Klassen. Daher haben sie keine Referenz auf das umschließende äußere Objekt. Gibt es ein solches synthetisches Element, kann es zu Problemen im Zusammenhang mit Serialisierung kommen, siehe Abschn. 14.8.

18.14 Enumerations

Aufzählungstypen, sog. Enums behandele ich nur deshalb in diesem Kapitel, weil sie so deklariert sind:

```
public abstract class Enum<E extends Enum<E>> implements Comparable<E>, Serializable
```

Was leistet diese abstrakte Klasse? Konkrete Klassen werden mit dem Keyword *enum* deklariert:

```
enum WochenTage{
     MONTAG, DIENSTAG, MITTWOCH, DONNERSTAG, FREITAG, SONNABEND, SONNTAG
}
```

Den angegebenen symbolischen Namen werden in der Reihenfolge ihres Auftretens die Werte $0, 1, 2, \ldots$ zugewiesen.

Eine Spezialisierung von *Enum* durch den Programmierer ist nicht möglich, da eine Reihe hierfür notwendiger Methoden final sind. Der oben angegebene Aufzählungstyp bedeutet aber im Kern Folgendes:

```
public final class WochenTage extends Enum{
  private final String name;
  private final int value;
   WochenTage(String name, int ordinal){
      this.name = name;
      this.value = ordinal;
  }
}
```

Dies ist allerdings aufgrund der Deklaration von *Enum* syntaktisch falsch. Aber es gibt einen wichtigen Hinweis: *enums* sind vollwertige Klassen.

Im Code würde man dann mit so etwas wie

```
WochenTage [] wochentage = {new WochenTage("MONTAG"), new WochenTage("DIENSTAG"), ... }
```

verwenden.

Dies wird einem durch obige Deklaration als *enum* abgenommen. Die Objekte einer Enum sind genau die durch die enum Konstanten definierten.

Bemerkung 18.14.1 (Enum-Konstanten)
Dass es tatsächlich nur die als Konstanten definierten *Enum*-Objekte gibt, wird durch verschiedene Maßnahmen sichergestellt ([GJS+14], §8.9):

1. Der Aufruf von *new* für ein Enum liefert einen Compilerfehler.
2. Die Methode *clone* ist final. Sie wirft stets eine *CloneNotSupportedException*.
3. *Enums* werden vom Serialisierungsmechanismus so behandelt, dass durch Deserialisierung keine duplizierten Objekte entstehen können.
4. Reflexive Erzeugung (*newInstance*, siehe Kap. 19) von *Enum*-Objekten ist nicht erlaubt.

◀

Diese Enum-Konstanten können wie ganzzahlige Werte primitiver Typen in Vergleichen verwendet werden. Die Enum-Konstanten können also mit „==“ verglichen werden und der Vergleich liefert das gleiche Ergebnis wie der mittels *equals*.

Im Unterschied zu Konstanten, die etwa so deklariert sein könnten:

```
public static final int MONTAG = 1;
```

haben *enums* viele Vorteile:

1. Sie sind typsicher.
2. Sie sind trotzdem effizient.
3. Sie haben einen Namen, der die Bedeutung erkennen lässt.

Beispiel 18.14.2 (Äpfel und Birnen)
Aufzählungstypen sind typsicher, Sie können also nicht Äpfel mit Birnen vergleichen, für die ich entsprechende *enums Apfel* und *Birne* definiert habe:

```
 9 Apfel apfel = Apfel.BOSKOOP;
10 Birne birne = Birne.WILLIAMSCHRIST;
11 System.out.println("Nun ist");
12 System.out.println("Wert von apfel =" + apfel.ordinal());
13 System.out.println("Wert von birne =" + birne.ordinal());
14 System.out.println("Aber: apfel.equals(birne) liefert: "+ (apfel.equals(birne)));
15 System.out.println("Aber: apfel==birne liefert: "+ (apfel==birne));
```

Die Zeilen 12 und 13 liefern die Ausgaben

```
Wert von a = 2
Wert von b = 2
```

Zeile 14 liefert:

```
Aber: a.equals(b) liefert: false
```

Die letzte Zeile führt auf einen Compiler-Fehler: „Incompatible operand types Apfel and Birne“. ◀

Jeder *Enum* Typ *E* hat die implizit deklarierten Klassenmethoden (siehe [GJS+11]) *static E[] values()* und *static E valueOf(String name)*, die Sie *nicht* in der API-Dokumentation finden. Es geht also auch Folgendes:

```
System.out.println("Wert von " + Apfel.valueOf("BOSKOOP").name()
             + " ist " + Apfel.valueOf("BOSKOOP").ordinal());
for (Apfel a : Apfel.values()) {
    System.out.println(a.name());
}
```

Im Zusammenhang mit *enums* sind viele weitere interessante Dinge zu erwähnen. So gibt es insbesondere die Klassen *EnumSet* und *EnumMap*, die eine sehr effiziente Implementierung bieten. Beide ermöglichen Zugriff in konstanter Zeit, unabhängig von der Anzahl der Werte. Sie lohnen sich nach [GJS+11] sogar für einfache Iterationen. Ihre Implementierung ist sehr kompakt, so belegt ein *EnumSet* mit bis zu 64 Elementen nur den Platz einer *long*-Variablen. Genauer gilt nach [Blo08]: Bei bis zu 64 Elementen wird ein *RegularEnumSet* geliefert, das durch ein *long* repräsentiert wird. Ist das *EnumSet* größer, so wird ein *JumboEnumSet* geliefert, das durch ein Array von *long* repräsentiert wird. Hier ein einfaches Beispiel:

```
for(Apfel a:EnumSet.range(Apfel.BOSKOOP, Apfel.FUJI)) {
    System.out.println(a);
}
```

Die Klasse *EnumSet* hat einige Fabrikmethoden, mit denen Sie einfach *EnumSets* erzeugen können. Eine sehen Sie in obigem Code, weitere finden Sie in der API-Dokumentation.

Die Fabrikmethoden in *EnumSet* illustrieren auch einige Vorteile von Fabrikmethoden im Vergleich zu Konstruktoren:

- In einer Fabrikmethode können Sie Objekte einer internen (privaten) Klasse zurückgeben, die das Interface implementiert. In einem Konstruktor wird immer ein Objekt der Klasse zurückgegeben.
- In einer Fabrikmethode können Sie in Abhängigkeit von den Parametern entscheiden, welche Klasse Sie für das Rückgabeobjekt verwenden. In *EnumSet* wird z. B. für maximal 64 Einträge ein *RegularEnumSet* zurückgegeben, für mehr als 64 Einträge ein *JumboEnumSet*. Ersteres belegt für die Daten ein *long*, letzteres ein Array *long [ ]*.

Bemerkung 18.14.3 (Namen in Enums)
Die Mehrheitsmeinung über die Namen in *enums* scheint mir zu sein:

- *enum*-Namen werden in Upper Camel Case gebildet, wie Klassennamen.
- *enum*-Instanzen werden in Upper Case (Großbuchstaben) gebildet, wie Konstanten.

Ich hoffe dies inzwischen weitgehend zu berücksichtigen. ◀

Enums können aber noch wesentlich mehr. Im Vorgriff auf 23.3 hier ein als *Singleton* implementierter Comparator:

```
public enum ComparatorExample02 implements Comparator<String> {
    INSTANCE;
    @Override
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
}
```

Oder eine neue Version der Methode *reverseOrder* aus der Klasse *Collections*, die Sie in *generics.Collections2* finden.

```
    @SuppressWarnings("unchecked")
    public static <T> Comparator<? super T> reverseOrder() {
        return (Comparator<T>) ReverseComparator.REVERSE_ORDER;
    }

    /**
     * @serial include
     */
    private static enum ReverseComparator implements Comparator<Comparable<Object>>{
        REVERSE_ORDER;
        @Override
        public int compare(Comparable<Object> c1, Comparable<Object> c2) {
            return c2.compareTo(c1);
        }
    }
```

Würde eine solche Klasse als *class* und nicht als *enum* geschrieben, so müsste die Methode *readResolve* implementiert werden, falls die Objekte serialisierbar sein sollten: Ohne dies könnten bei Deserialisierung weitere Singleton-Objekte entstehen. Bei der Verwendung von *enum*s wird das schon in der Klasse *Enum* ausgeschlossen.

Enums können auch zur Implementierung endlicher Automaten verwendet werden Hier ein einfaches Beispiel von Cyrille Martraire:

```
public enum BabyState {

    POOP(null), SLEEP(POOP), EAT(SLEEP), CRY(EAT);

    private final BabyState next;

    private BabyState(BabyState next) {
        this.next = next;
    }

    public BabyState next(boolean discomfort) {
        if (discomfort) {
            return CRY;
        }
        return next == null ? EAT : next;
    }
}
```

Die Parameter in den Klammern nach den *enum* Konstanten sind Parameter, die an den Konstruktor weitergegeben werden. Durch den Konstruktor und die Methode *next* wird also die folgende Zustandsübergangsmatrix definiert:

State	next	
	comfort	discomfort
CRY	EAT	CRY
EAT	SLEEP	CRY
POOP	EAT	CRY
SLEEP	POOP	CRY

Visualisiert wird dies durch das folgende state chart diagram in Abb. 18.3.

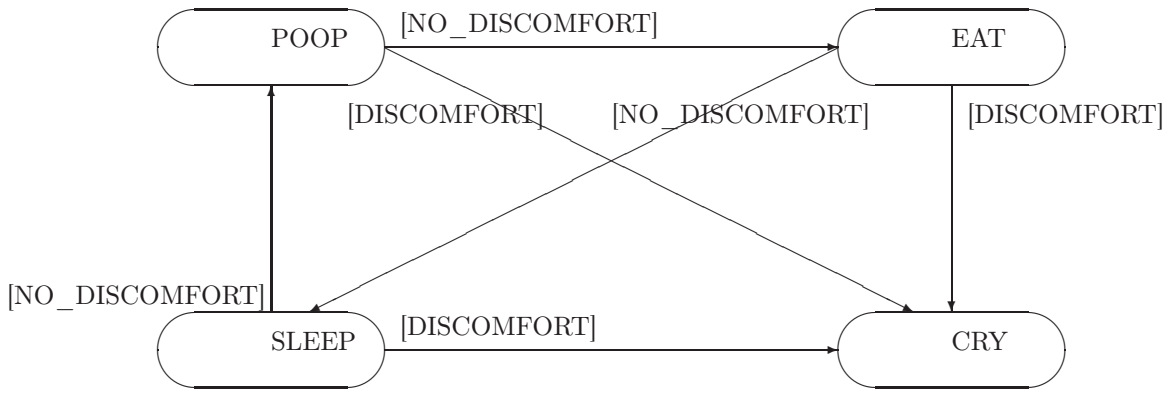


Abb. 18.3: Endlicher Automat BabyState

18.15 Historische Anmerkungen

Generics wurden in Java 1.5 (tiger release) 2004 eingeführt. Ich habe von manchen Kollegen die Ansicht gehört, seit der Einführung von Generics sei Java nicht mehr einfach. Aber was heißt „einfach“? Was gut erklärt wird, erscheint immer einfach. Wenn man es dann auch wirklich verstanden hat, ist es auch einfach. Solange man es nicht jemand anderem erklären kann, hat man es wahrscheinlich nicht ganz verstanden. Für mich haben Generics die Java Programmierung vereinfacht. Die Typsicherheit wird auf weitere Klassen ausgedehnt und viele Casts werden eliminiert. Entscheidend ist aber, das der Entwurfsentscheidungen direkt im Code abgebildet werden.

Sehr viele Einsatzmöglichkeiten gibt es für Enums (s. o.). Diese werden aber nach meinem Eindruck noch nicht in großem Maße genutzt.

In Java 7 wurde die automatische Erkennung von Typparametern (type inference) erweitert. So muss nach einer Deklaration mit einem Typ T bei der anschließenden Initialisierung der Typparameter nicht mehr angegeben werden, z. B. :

```
Jlist<String> namensliste = new JList<>();
```

Der sogenannte *diamond operator* $<>$ ist eine kleine, nützliche Erleichterung für Programmierer und Programmiererinnen.

18.16 Aufgaben

1. Schreiben Sie bitte eine Klassenmethode, die für eine variable Anzahl von Collections die Summe aller Anzahlen der Elemente der Collections liefert!

2. Schreiben Sie bitte ein Interface *Set*, das von Klassen implementiert werden kann, die zur Implementierung einer „zu viele“ Richtung einer binären Assoziation genutzt werden kann. Begründen Sie bitte Ihre Entscheidungen für die ausgewählten Operationen und deren Signaturen!

3. Implementieren Sie bitte zwei Varianten einer geordneten Liste: Eine für einen beliebigen Typparameter T und eine für einen durch *Comparable* beschränkten! Geordnet heiße dabei: Die Elemente werden entsprechend der definierten Ordnung abgespeichert, also $i < j \Rightarrow get(i) < get(j)$.

4. Sie sehen hier einen Ausschnitt aus einer generischen Klasse, die einen Stack mittels einer einfach verketteten Liste implementiert

```
10  public class StackAsSimpleList<E> implements Stack<E> {
20     private Knoten head;
    ...
30     public E peek() {
    ...
40        return head.daten;
50     }
   ...
60     private static class Knoten<T>{
70        T daten;
80        Knoten<T> next;
90        Knoten(T t, Knoten<T> k){
100         daten = t;
110         next = k;
120       }
130    }
140 }
```

Dieser Code enthält Fehler, die mit den Standardeinstellungen von Eclipse zu einem Compilerfehler und einigen Warnungen führen. In welchen Zeilen? Welcher Fehler bzw. Warnung ist es? Warum bemängelt der Compiler dies und was ist die Ursache? Wie machen Sie das richtig?

5. Hier folgt eine Methode aus einem Versuch eine einfach verkettete Liste zu implementieren:

```
public class EinfachVerketteListe<T> implements Liste<T> {
   public Knoten<T> head;
   public Knoten<T> tail;
   ...
   public Knoten<T> find(T elem) {
      Knoten<T> pos = head;
      while((pos.next.daten != elem) &&
            (pos.next.hashCode() != pos.next.next.next.hashCode())){
         pos = pos.next;
      }
      if (pos.next.daten.equals(elem)){
          return pos;
      }
      else{
         return null;
      }
   }
   ...
}
public class Knoten<T> {
   public T daten;
   public Knoten<T> next;
   public Knoten(){
      daten = null;
      next = null;
   }
}
```

Der Programmierer testete diesen Code und stellte fest: Für 1 . . . 127 Knoten ist der Test erfolgreich, für 128 und mehr Knoten schlägt der Test fehl: Es werden keine Elemente gefunden. Erklären Sie bitte, wie es zu dem Fehler kommt und korrigieren Sie den Code! Sie können dabei gleiche weitere Unzulänglichkeiten beseitigen. Geben Sie auch dazu bitte jeweils ein Begründung an!

6. Gegeben ist der folgende Code:

```
class Fu<T> {
   public static int classVar = 42;
}
```

Dann ist die Zuweisung in der folgende Zeile 10 illegal,

```
10 Fu<String>.classVar = 91;
20 Fu.classVar = 91;
```

während die in Zeile 20 legal ist. Warum ist das so? Warum ist hier die Verwendung eines raw types unkritisch? Was ist an diesem Beispiel noch auszusetzen?

7. Schreiben Sie bitte eine generische Klasse *Paar*, die zu einem Objekt, das als Schlüssel dient, ein weiteres Objekt enthält, den zugehörigen Wert. Zwischen den Schlüsselobjekten gibt es eine Ordnung. Paare sind entsprechend ihrer Schlüssel geordnet. Ferner kann für einen Schlüssel der zugehörige Wert ermittelt werden. *Hinweis*: Diese Klasse wird erst dann wirklich sinnvoll, wenn sie für die Implementierung einer sog. *HashTable* verwendet wird.

8. Hier folgt etwas Code:

```
100 public class Pair<T> {
110    private final T first;
120    private final T second;
130    public Pair(T first, T second) {
140        this.first = first;
150        this.second = second;
160    }
170    public T first() {
180        return first;
190    }
200    public T second() {
210        return second;
220    }
230    public List<String> stringList() {
240        return Arrays.asList(String.valueOf(first),
250                             String.valueOf(second));
260    }
270    public static void main(String[] args) {
280        Pair p = new Pair<Object>(23, "skidoo");
290        System.out.println(p.first() + " " + p.second());
300        for (String s : p.stringList())
310            System.out.print(s + " ");
320    }
330 }
```

   Ist dieser Code korrekt? Wenn ja: Was liefert er? Wenn nein: Was ist falsch?

9. Noch etwas Code:

```
010 public class LinkedList<E> {
020    private Node<E> head = null;
030    private class Node<E> {
040        E value;
050        Node<E> next;
060        Node(E value) {
070            this.value = value;
080            this.next = head;
090            head = this;
100        }
110    }
120    public void add(E e) {
130        new Node<E>(e);
140        // Link node as new head
150   }
160    public void dump() {
170        for (Node<E> n = head; n != null; n = n.next)
180           System.out.println(n.value + " ");
```

```
190    }
200    public static void main(String[] args) {
210        LinkedList<String> list = new LinkedList<String>();
220        list.add("world");
230        list.add("Hello");
240        list.dump();
250    }
260 }
```

9.1. Was liefert dieser Code?

9.2. Ist etwas falsch? Wenn ja, was und warum?

9.3. Wie machen Sie das richtig?

10. Betrachten Sie bitte folgenden Code-Schnipsel:

```
010  List<?> liste = new ArrayList<>();
020  list.add(null);
030  liste.add("moin");
```

Was passiert beim Compile und ggf. Ausführen des Codes? Begründen Sie bitte Ihre Antwort! (Es geht nicht um genaue Wortlaute etwaiger Compilermeldungen.)

11. Schreiben Sie bitte ein Enum, dass einen einfachen Schalter mit den Zuständen *ein* und *aus* repräsentiert, zwischen denen gewechselt werden kann!

12. Schreiben Sie bitte eine generische Klasse *Queue*, die das vorgegebene Interface *IQueue* implementiert!

```
import java.util.Optional;

/**
 * Ein sehr einfaches Interface für eine Queue (First In - First Out Speicher)
 * mit einer beschränkten
 * Kapazität.
 * @author Bernd Kahlbrandt
 *
 */
public interface IQueue<E> {
    int DEFAULT_CAPACITY = 42;
    /**
     * Fügt ein neues Element in die Queue ein. Wirft eine
     * {@link QueueFullException}, wenn die Kapazität
     * der Queue erschöpft ist.
     * @param element
     */
    void enqueue(E element) throws QueueFullException;
    /**
     * Entfernt das zuerst eingefügte Element aus der Queue. Wirft eine
     * {@link QueueEmptyException},
     * wenn die Queue leer ist.
     */
    void dequeue();
    /**
     * Liefert das zuerst eingefügte Element der Queue, das sich noch in der
```

```
     * Queue  befindet, via eines {@link Optional}.
     * @return {@link Optional}.
     */
    Optional<E> peek();
    /**
     * Tests if the Queue is full.
     * @return true, iff the Queue is full.
     */
    boolean isFull();
    /**
     * Tests if the Queue is empty.
     * @return true, iff the Queue is empty.
     */
    boolean isEmpty();
}
```

Die Einträge in der *Queue* verwalten Sie bitte über ein Array fester Länge. Beim Erzeugen eines Objekts kann eine ganzzahlige Kapazität der Queue angegeben werden. Wird keine Kapazität angegeben, so wird eine Queue mit der default-Kapazität von 42 angelegt. Wird eine nicht-positive Kapazität beim Erzeugen angegeben, so wird die (einzige) geeignete RuntimeException geworfen. Der erste Eintrag kommt auf Position 0 im Array, der nächste auf Position 1 usw. Ist das Ende des Arrays erreicht, so gibt es zwei Möglichkeiten:

12.1. Wurde keine Element am Anfang entnommen (*dequeue*). Dann ist die Queue voll und beim nächsten *enqueue* wird eine RuntimeException „QueueFullException“ geworfen.

12.2. Andernfalls geht es vorne mit 0 weiter. Bis der (sich rollierend verschiebende) Platz doch aufgebraucht ist.

Nun zu den einzelnen Methoden und der jeweiligen Fehlerbehandlung:

enqueue Bekommt ein Element übergeben und fügt es am Ende der Queue ein. Ist die Queue bereits voll, so wird eine Runtime Exception „QueueFullException“ geworfen.

dequeue Entfernt das erste Element aus der Queue. Ist die Queue leer, so wird eine Runtime Exception „QueueEmptyException“ geworfen.

peek Liefert ein Optional für das erste Element der Queue zurück. Ist die Queue leer, liefert peek ein leeres Optional.

Denken Sie bitte auch an etwaige weitere Methoden, die für diese Java-Klasse notwendig sein könnten! Begründen Sie bitte warum Sie weitere Methoden schreiben oder darauf verzichten.

13. Ein Stack ist eine Klasse, die folgende Methoden hat:

void push(E element) Legt das übergebene Element als Oberstes auf den Stack.

E pop() Entfernt das oberste Element vom Stack und liefert es zurück. Ist der Stack leer, so wird eine RuntimeException „EmptyStackException“ geworfen.

boolean isEmpty() Liefert *true*, wenn der Stack leer ist, andernfalls *false*.

Es gibt viele Möglichkeiten, einen Stack zu implementieren. Sie sollen hier bitte zwei typische realisieren, d. h. eine generische Klasse schreiben, die diese Methoden hat. Weitere Methoden, die eine Java-Klasse üblicherweise implementieren muss, kommen natürlich noch hinzu.

13.1. Die Elemente werden in einem Array gespeichert. Die Anfangsgröße wird in einem Konstruktor übergeben. Sehen Sie bitte eine default-Größe vor. Ist das Array voll, so vergrößern Sie es bitte um einen Prozentsatz, z. B. 100%.

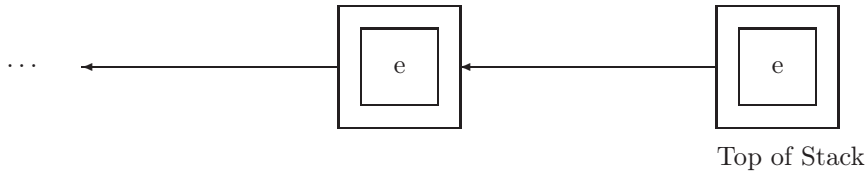


Abb. 18.4: Innere Klasse

13.2. Die Einträge im Stack werden über Objekte einer statischen geschachtelten Klasse verwaltet, wie in Abb. 18.4 skizziert. Ein Objekt dieser Klasse enthält das jeweils „oberste“ Element des Stacks und eine Referenz auf das vorhergende Objekt der inneren Klasse, das das vorhergehende Element enthält usw.

Schreiben Sie bitte einige aussagefähige Testfälle für beide Varianten!

Kapitel 19

Reflection

19.1 Übersicht

Die Eigenschaften von Java-Objekten können zur Laufzeit abgefragt und in einem gewissen Maße auch verändert werden. So kann etwa mittels des Ausdrucks Typ.*class* der Typ eines Referenz-Typs in Erfahrung gebracht werden oder der genauere Typ eine Objekts mit dem *instanceof*-Operator bestimmt werden. Die Klasse eine Objekts bekommen Sie mit der Methode *Class<?> getClass()*. Derartige Techniken braucht man nicht für jede Art von Anwendung. Sie sind aber für Enwicklungsumgebungen, (Eclipse, NetBeans), Frameworks (JUnit u. v. a. m) oder aspektorientierte Programmierung und dependency injection unverzichtbar. Sie können auch für Remote Procedure Calls sinnvoll sein und spezielle eingebettete Systeme. In allen anderen Fällen sollten Sie zunächst davon ausgehen, dass der Einsatz von Reflection keine gute Idee ist. Aber auch bei Testfällen (siehe Abschn. B.6,19.6) können Sie Bedarf für Reflection haben. In Java werden alle Dinge, die hiermit zu tun haben, mit den Schlagworten *Run Time Type Information (RTTI)* oder *Reflection* bezeichnet.

In objektorientierten Systemen erfolgen upcasts automatisch: Ist eine Klasse deklariert, so kann an dieser Stelle ein Objekt jeder Unterklasse eintreten. Andersherum gibt es keinen Automatismus und (down)casts gelten als unschön und potenziell gefährlich. Reflection ermöglicht es, den jeweiligen Typ zur Laufzeit zu ermitteln und die Gefahr so zu reduzieren.

Zwei Aspekte von Reflection streiche ich hier besonders heraus:

1. Hier wird das Meta-Modell von Java verwendet: Die Klasse *Class* ist der Ausgangspunkt. Von hieraus können alle Eigenschaften von Objekten dieser Klasse erkundet werden. So erhalten Sie Informationen über alle Elemente.

2. Hier werden Eigenschaften von Objekten überprüft, die sie zur Laufzeit haben.

Beiden Aspekten sind für Anfänger typische Schwierigkeiten zuzuordnen.

1. Abstraktionsvermögen: Es ist plötzlich die Rede von Klassen, die Klassen beschreiben: *Class*, *Field* (*Attribute*) *Method*, *Constructor* etc. Der Umgang mit der weiteren Abstraktionsebene „Metamodell“ macht manchen Schwierigkeiten.

2. Es gibt Vieles, das schief gehen kann: Fast alle Reflection nutzenden Aktivitäten können zu Exceptions führen, von denen viele keine *RuntimeExceptions* sind.

19.2 Lernziele

- Eigenschaften von Objekten und Klassen durch geeignete Methodenaufrufe ermitteln können.
- Zur Laufzeit Objekte von Klassen erzeugen können, die zur Compile-Zeit noch nicht bekannt sind.

- Das Metamodell für Java Klassen kennen.
- Bei Bedarf auch *private* oder *protected* Methoden testen können.
- Reflection sinnvoll einsetzen können.

19.3 Objekte, Klassen und Typen

Die hier zu beschreibenden Dinge können mindestens unter zwei Gesichtspunkten betrachtet werden:

1. Dynamisch Informationen über Objekte zu bekommen und mit diesen Eigenschaften weiter zu arbeiten, also Objekte zu erzeugen, zu verändern, Methoden aufzurufen etc.
2. Arbeiten mit dem Metamodell für Java-Klassen, von dem Sie einen Ausschnitt in Abb. 19.1.

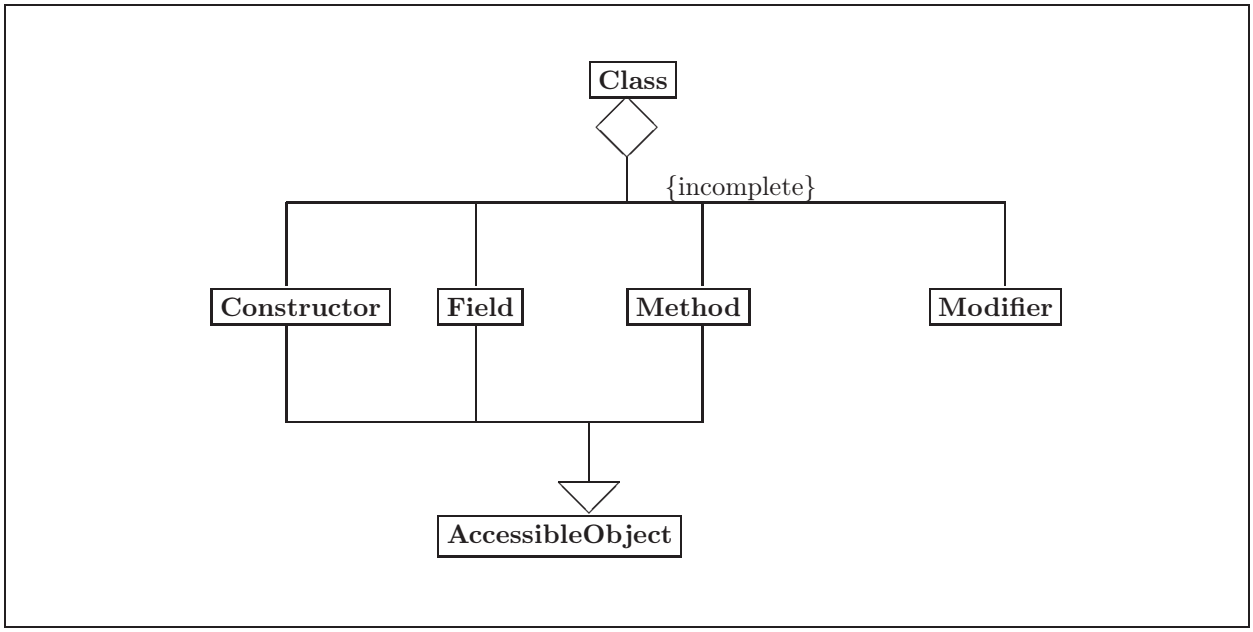


Abb. 19.1: Metamodell für Java-Klassen (Ausschnitt)

Mindestens eine Möglichkeit sich zur Laufzeit Informationen über den Typ eines Objekts zu beschaffen kennen Sie bereits: Den *instanceof*-Operator. Ein typisches Beispiel finden Sie in Bsp. 4.8.3 auf S. 56. Einige wichtige Dinge habe ich in früheren Kapiteln noch nicht genannt. Auf der linken Seite des *instanceof*-Operators steht ein Ausdruck, dessen Ergebnis ein Objekt von einem Referenz-Typ oder dem *null*-Typ ist. Der Referenz-Typ auf der rechten Seite des *instanceof*-Operators muss ein *reifiable* Referenz-Typ sein.

Definition 19.3.1 (Reifiable Types)
Ein Typ ist *reifiable*, wenn die Typinformation zur Laufzeit vollständig zur Verfügung steht. Genau die folgenden Typen sind *reifiable*:

1. Nicht-generische Klassen und Interfaces.
2. Parametrisierte Typen, in denen alle Typparameter unbeschränkte Wildcards sind.
3. Raw Types.

4. Primitive Typen.
5. Array-Typen, deren Elemente reifiable sind.
6. Geschachtelten Typen, wenn jeder Typ einzeln reifiable ist.

([GJS+14], Abschn. 4.7). ◄

Der *instanceof*-Operator liefert *true*, wenn ein Cast der linken Seite auf den rechts stehende Typ keinen Compiler-Fehler liefert. Insbesondere liefert er *false*, wenn links *null* steht.

Ferner gibt es einige Ausdrücke, die den Typ bzw. die Klasse liefern. Für die primitiven Typen etwa so: *.class* liefert die Klasse, *.TYPE* den Typ. Der folgende Code

```
Class<?> clazzWild = Void.TYPE;
System.out.println(clazzWild);
Class<Void> clazzTyped = Void.TYPE;
System.out.println(clazzTyped);
clazzTyped = Void.class;
System.out.println(clazzTyped);
clazzWild = Integer.TYPE;
System.out.println(clazzWild);
clazzWild = Integer.class;
System.out.println(clazzWild);
```

liefert die Ausgaben

```
void
void
class java.lang.Void
int
class java.lang.Integer
```

Ganz analoge Ergebnisse erhalten Sie für Arrays primitiver Typen. Hier handelt es sich aber um Klassen, also geht nur *.class*. Der folgende Code

```
Class<?> clazzWild = Void[][].class;
System.out.println(clazzWild);
Class<Void[][]> clazzTyped = Void[][].class;
System.out.println(clazzTyped);
clazzWild = Integer[][].class;
System.out.println(clazzWild);
clazzWild = Double[][][].class;
System.out.println(clazzWild);
```

liefert:

```
class [[Ljava.lang.Void;
class [[Ljava.lang.Void;
class [[Ljava.lang.Integer;
class [[[Ljava.lang.Double;
```

Den Kern der Java-Möglichkeiten um dynamisch Informationen über Klassen zu erhalten bilden aber die Klasse *Class<T>* im Paket *java.lang* und die Methode *getClass* der Klasse *Object*. *Class<T>* hat keinen öffentlichen Konstruktor. Objekte werden automatisch von der JVM erzeugt. Um ein Objekt reflexiv zu erzeugen, verwenden Sie die Methode *newInstance*. Gegebenenfalls können Sie auch Methodenreferenzen verwenden, siehe hierzu Kap. 17 zu λ-Ausdrücken und Streams.

Beginnen wir mit der Klasse *Class* und einem Beispiel aus dem Java API (reflection.GetClassExample01):

```
Number n = 0;
Class<? extends Number> cn = n.getClass();
```

Geben Sie sich *cn* aus, so erhalten Sie:

```
class java.lang.Integer
```

Bei

```
Number d = Math.PI;
Class<? extends Number> dn = d.getClass();
```

erhalten Sie dann ganz entsprechend

```
java.lang.Double
```

Das liegt einfach daran, dass das ganzzahlige Literale *int*s und dezimale Literale *double*s sind. (Bei der Ausgabe mittels *toString* wird noch `class␣` davorgestellt.)

Die Methode *getClass* liefert den vollqualifizierten Klassennamen, den das jeweilige Objekt hat. Im ersten Fall ist das *java.lang.Integer*, im zweiten *java.lang.Double*. Das Objekt, das zurückgeliefert wird, ist die Auslöschung (erasure) des Typs, für dessen Objekt *getClass* aufgerufen wird. Genau dieses Objekt wird von synchronisierten Klassenmethoden „gelocked“. Da die im Beispiel beteiligten Klassen *Integer* und *Double* nicht parametrisiert sind, ist der Typ gleich dem ausgelöschten.

Der genaue Typ, den *getClass* liefert ist Class<? extends |X|>, wobei |X| der statische Typ nach type erasure des Ausdrucks ist, für den sie aufgerufen wird.

Im folgende Fall

```
Class<?> cList = (new ArrayList<String>()).getClass();
```

erhalten wir also

```
java.util.ArrayList
```

ohne den Typparameter *String*. Wegen Typauslöschung kommen wir an den auch gar nicht heran. Wir können lediglich mittels *getTypeParameters()* die deklarierten Namen der Typparameter erhalten, hier *E*.

Mittels *getClass* erhalten Sie die Klasse eines Objekts. Damit können Sie dann weiterarbeiten. Abbildung 19.2 erinnert noch einmal an die Zusammenhänge:

- Der Compiler erzeugt aus eine .java-Datei eine oder mehrere .class Dateien, für jede Klasse eine.
- Die JVM lädt bei Bedarf die Klassendatei.
- Aus dieser Klassendatei wird ein Objekt der Klasse *Class* erzeugt. Dieses Objekt enthält alle Informationen über Objekte der Klasse, die zur Laufzeit verfügbar sind. Bei *Class* handelt es sich also auch um eine Metaklasse. Metadinge sind Dinge, die etwas über ein Ding aussagen. Eine Metaklasse beschreibt also eine Klasse. Um diese Beschreibung abzufragen, gibt es viele Methoden, die ich hier nicht alle aufzähle. Für jede Eigenschaft gibt es die entsprechende Methode, z. B.
  - *getDeclaredFields()* und *getFields()*, *getDeclaredField(String name)* und *getField(String name)*
  - *getDeclaredMethods()* und *getMethods()*, *getDeclaredMethod(String name, Class<?>... parameterTypes)* und *getMethod(String name, Class<?>... parameterTypes)*
  - *getDeclaredConstructors()* und *getConstructors()*
  - ...

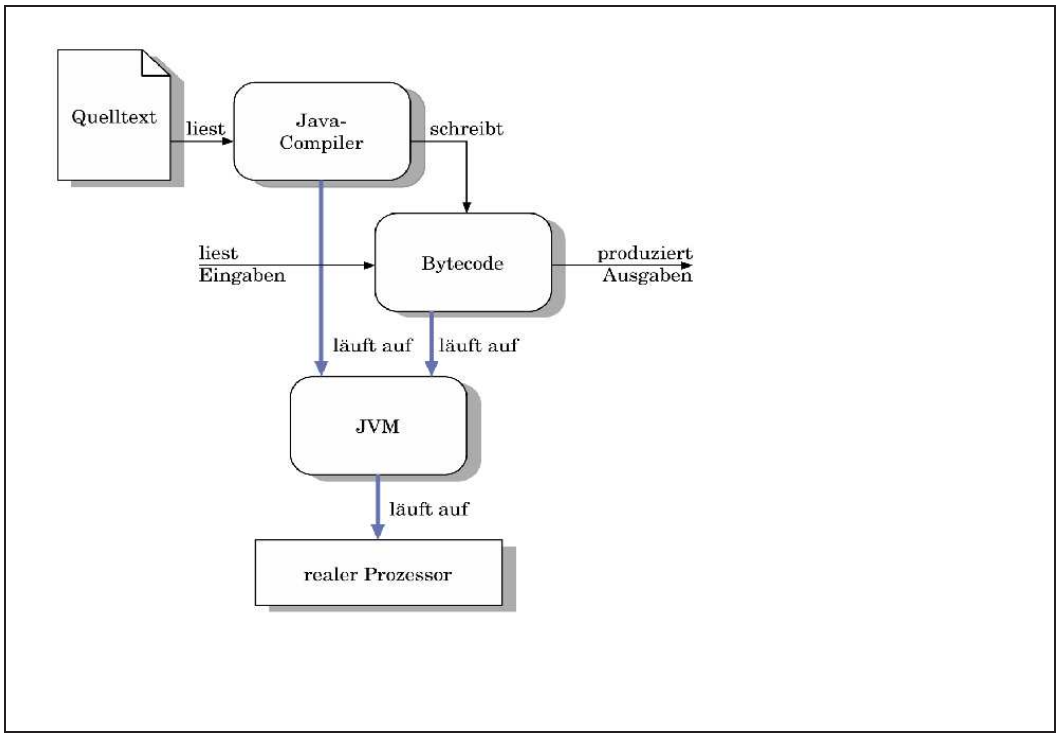


Abb. 19.2: Java Verarbeitungsmodell

Die Methode *getFields* etc. liefert für eine Klasse ein Array von *Field*s, d. h. alle *public* Attribute (Fields), die in dieser Klasse oder ihren Oberklassen definiert sind. Die Methode *getDeclaredFields* etc. liefert alle Attribute der Klasse, unabhängig von ihrer Sichtbarkeit, aber *nicht* die in Oberklassen definierten Attribute. Die Methode *getField(String name)* liefert das *public* Attribut mit dem angegebenen Namen (oder eine Exception). Ganz entsprechend *getDeclaredField(String name)* für ein Attribut mit beliebiger Sichtbarkeit. Ganz analog arbeiten die anderen erwähnten Methodensätze. Die Klassen *Field*, *Method* etc. finden Sie im Paket *java.lang.reflect*.

Die Beispiele *ClassProperties* und *ObjectProperties* zeigen den Einsatz einiger der Methoden.

Kurz noch einige Anmerkungen zur Darstellung der Methoden: Die Ausgabe mittels

```
System.out.println(m);
```

ist ganz einfach. Aber was passiert dabei genauer? Bekanntlich wird `m.toString()` auf der Konsole ausgegeben. Sehen Sie sich den Sourcecode von *toString()* in *Method* einmal an! Nachdem Sie sich durch die Methodenaufrufhierarchie gearbeitet haben, finden Sie in Klasse *Executable* Code wie diesen: Folgendes passiert dort:

1. Es wird mit einem *StringBuffer* gearbeitet. Es werden also insbesondere keine *String*s mit „+“ aneinandergehängt. Seit Java 8 haben Compiler-Hersteller aber die Freiheit, das erzeugen unnötiger String-Objekte zu unterbinden, so das der „Umweg“ über *StringBuffer* im Laufe der Zeit unnötig werden kann.

2. Die Modifier einer Methode sind in *int*'s verschlüsselt. Diese werden mittels

```
int mod = getModifiers() & Modifier.methodModifiers();
if (mod != 0) {
    sb.append(Modifier.toString(mod) + " ");
```

```
}
```

entschlüsselt. Die ganze Zahl *METHOD_ MODIFIERS*, die *Modifier.methodModifiers()* zurückliefert, ist das bitweise *oder* der verschiedenen Modifier für Methoden von *abstract* bis *synchronized*. Durch das bitweise *und* wird genau der gewünschte Wert „herausgeschnitten". Diesen „entschlüsselt" dann die Klassenmethode *toString* von *Modifier*.

3. usw.

Ich empfehle dringend, diese und weitere Methoden selber auszuprobieren!

Bemerkung 19.3.2 (Designentwicklung)
In Java 8 wurde eine neue abstrakte Oberklasse *Executable* von *Method* und *Constructor* eingeführt. Das ist ein gutes Beispiel für die Weiterentwicklung von Code. ◄

19.4 Dynamische Aufrufe

In Abschn. 19.3 wurden Methoden vorgestellt, mit denen Sie Informationen über Klassen erhalten können. Mit diesen Informationen können Sie aber auch etwas anfangen. So können Sie die Inhalte von Attributen auslesen und ändern:

- *get(Object o)* liefert den „Wert" des Fields (Attributs) für das angegebene Objekt.
- *getXxx(Object o)* liefert für die primitiven Typen wie etwa *char* für Xxx den Wert mit dem jeweiligen Typ, hier z. B. *char*.
- ...
- *set(Object obj, Object value)* weist dem Field für das Objekt *obj* das Objekt *value* zu. Zunächst geht das nur für public Attribute. Mittels der Methode
- *setAccessible(boolean flag)* aus der Oberklasse *AccessibleObject* können Sie aber Ihre Absicht kund tun, auf Elemente zuzugreifen, die private, protected oder package sichtbar sind. Anschließend können Sie ein Attribut lesen oder mittels *set* verändern. Ganz analog können Sie auf diese Weise Methoden mittels *invoke* aufrufen. Verhindert werden kann dies durch einen Security-Manager. Dieses Thema sprengt aber den Rahmen dieser Darstellung.
- *setXxx(xxx o)* setzt analog zu getXxx Werte primitiver Typen.

Wenn Sie schon einmal den Debugger Ihrer Entwicklungsumgebung (z. B. Eclipse) verwendet haben, sind Sie dieser Möglichkeit schon begegnet. Dort können Sie ja bei Bedarf fehlerhafte Variableninhalte korrigieren. Es sollte Ihnen jetzt klar sein, wie Eclipse das machen kann.

Dies funktioniert auch bei Attributen, die als *final* deklariert sind.

Ganz entsprechend können Sie mit Methoden arbeiten. So gibt es in der Klasse *Method* die Methode *invoke(Object obj, Object... args)*. Diese können Sie ohne Weiteres für öffentliche Methoden aufrufen. Um sie für andere Methoden aufrufen zu können, verwenden Sie wie bei der set-Methode für Attribute die Methode *setAccessible*.

Hier ein einfaches Beispiel für die praktische Anwendung:

Beispiel 19.4.1 (Methoden Ausführen)

```
MatheTest testObjekt = new MatheTest();
for(Method m:testObjekt.getClass().getMethods()){
   if(m.isAnnotationPresent(Test.class)){
      m.invoke(testObjekt);
   }
}
```

◄

Bemerkung 19.4.2 (Reflexiver Aufruf statische Methoden)
Ist die Methode, für die *invoke* aufgerufen wird *static*, so wird der erste Parameter ignoriert. Sie können dort also z. B. *null* übergeben. ◄

Bemerkung 19.4.3 (Reflexiver Methodenaufruf und Exceptions)
In Beispiel 19.4.1 habe ich die Fehlerbehandlung ausgeblendet. Aber dabei müssen Sie etwas beachten, wenn die reflexiv aufgerufene Methode eine Exception wirft. Der Aufruf von *invoke* wird wie folgt in einen *try-catch-block* ausgeführt:

```
try {
   m.invoke(null, minInts);
} catch (IllegalAccessException | IllegalArgumentException | InvocationTargetException e) {
    if(e.getCause() instanceof ArithmeticException){
       throw (ArithmeticException)e.getCause();
    }else{
       e.printStackTrace();
    }
}
```

Wird in der Methode m eine Exception geworfen wie hier eine *ArithmeticException*, so erhalten Sie in Ihrem Code eine *InvocationTargetException*. Diese wird geworfen, wenn eine reflexiv aufgerufene Methode eine Exception wirft. Den Grund bekommen Sie mittels *getCause()* heraus. Ich benötige solchen Code, wenn ich z. B. überprüfen möchte, ob Sie in einer Prüfung die geforderte Exception werfen. ◄

Ganz analog funktioniert das Erzeugen neuer Objekte. Dazu dient die Methode *newInstance(Object ... initargs)* von *Constructor*. Diese hat eine variable lange Argumentliste um die Parameter-Objekte für den Konstruktor zu übergeben. Sie holen sich also mittels der Methode *getConstructors* der Klasse *Class* die Konstruktoren, bekommen mittels *getParameterTypes()* die Parametertypen, entsprechend die generischen Parametertypen und können diese entsprechend bestücken. Den Default-Konstruktor können Sie auch direkt über *Class* mittels der Methode *newInstance()* aufrufen.

19.5 Umgang mit Arrays

Arrays spielen in Java eine Sonderrolle, da sie direkt in der Sprache verankert sind. Für den reflektiven Umgang mit ihnen benötigen Sie die Klasse *Array* aus dem Paket *java.lang.reflect*, nicht zu verwechseln mit einer anderen Utility-Klasse *java.util.Arrays*. Sie enthält Klassenmethoden zum dynamischenUmgang mit Arrays. Hier zwei einfache Beispiele: In einer Klasse seien zwei Attribute deklariert:

```
Object [] objects = new Object[10];
Object object = objects;
```

Für beide liefert die Methode *isArray* von *Class* true. Im ersten Fall stimmen der deklarierte Typ und der des zugewiesenen Objekts überein: Beides ist ein Array von *Object*. Im zweiten Fall ist *Object* deklariert, aber der dynamische Typ nach Zuweisung ist wieder ein Array von *Object*.

Als zweites Beispiel nehme ich eine triviale Klasse *BeispielKlasse01* mit einem Attribut von einem Array-Typ:

```
public class BeispielKlasse01 {
   private Object [] data = new Object[10];
   ...
}
```

An dieses Attribut komme ich mittels Reflection heran:

```
BeispielKlasse01 beispiel = new BeispielKlasse01();
Field dataField = beispiel.getClass().getDeclaredField("data");
```

Die Klasse von *dataField* ist kein Array, wohl aber der Typ: die von *Object* ererbte Methode *getClass* liefert die Laufzeitklasse des Objekts, dass ist hier natürlich *Field*. Die Methode *getType()* liefert den Typ, mit dem das Attribut (Field) in der Klasse *BeispielKlasse01* deklariert ist. Dies ist hier ein Array von *Object*.

Mache ich das Attribut *data* noch zugänglich, so komme ich auch an dessen Eigenschaften heran, z. B. :

```
dataField.setAccessible(true);
int laenge = dataField.getType().isArray()?
             Array.getLength(dataField.get(beispiel)):-1;
```

Dies zeigt mir nun, dass das Array die Länge 10 hat. Den Code finden Sie in *reflection.ArrayExamplenn*, nn=01,02.

Für den Zugriff auf von repeatable Annotationen gibt es die Methode *getAnnotationsByType*. Hier ein Beispiel:

Beispiel 19.5.1 (Repeatable Annotations)
Hat man eine repeatable Annotation *Schedule*, so bekommt man mittels der Methode *getAnnotationsByType(Schedule.class)* ein Array der Annotationen. Es besteht also kein Grund mehr in diesem Fall auf den zugehörige Container-Annotationstyp zuzugreifen.

```
Class<?> testClass = Class.forName("a06.UseSchedule");
for(Method m:testClass.getDeclaredMethods()){
   for(Annotation a : m.getAnnotationsByType(Schedule.class)){
      System.out.print(m);
      System.out.println(a);
   }
}
```

◀

19.6 Anwendungen

Reflection brauchen Sie nicht in jeder Anwendung. Beschäftigen Sie sich aber mit Werkzeugen für Entwickler, so brauchen Sie dies unbedingt. Oft werden Sie dabei auch weitere Java-Möglichkeiten nutzen. Als Beispiele nennen ich Annotationen (siehe Kap. 20) und Threads. Jede Entwicklungsumgebung (Eclipse, NetBeans, . . .) für Java verwendet Reflection. Ebenso benötigen Sie zumindest den *instanceof*-Operator, um in manchen Anwendungen die Typsicherheit zumindest zu verbessern.

Tools wie JUnit definieren Annotationen (@Test, @Before, . . .) und verwenden Code, wie ihn Beispiel 19.4.1 illustriert.

Sie selbst können beim Testen privater Methoden sinnvoll von Reflection Gebrauch machen. In vielen Fällen ist es sinnvoll nicht nur *public* oder *package* sichtbare Methoden zu testen, sondern auch solche, die *protected* oder *private* sind. Sie dürfen aber in der Praxis die Sichtbarkeit nicht auf *package* oder *public* erweitern und hinterher wieder auf den Ursprungszustand zurückändern.

Hier kennen Sie aber den Namen der Methode und die Parameter. Sie besorgen sich also ein Objekt der zu testenden Klasse, holen sich die zu testende Methode, nennen wir sie *methodToTest*, machen sie *accessible* und rufen mit dem Objekt und den Parametern für die Methode *invoke* auf. Sie gehen also nach folgendem Schema vor.

- ClassUnderTest testObject = new ClassUnderTest(. . .).
- Method mtt = textObject.getClass().getDeclaredMethod("methodToTest", parameter types).

- mtt.setAccessible(true).
- mtt.invoke(testObject, parameter values).

Das Ganze müssen Sie in einen *try-catch-block* kapseln. Die Ergebnisse der Aufrufe testen Sie wie üblich mittels der verschieden *assert...* Methoden.

Hier ein konkretes Beispiel: In einer Implementierung einer *Liste* mittels eines Arrays gibt es eine Methode *ensureCapacity()*. Diese prüft, ob im Array der Listeneinträge noch Platz ist. Ist dies nicht der Fall, so wird das Array vergrößert (z. B. auf die doppelte Größe).

Das können Sie so testen:

```
Method ensureCapacity = saas.getClass().getDeclaredMethod("ensureCapacity",
                                                          (Class<?> [])null);
ensureCapacity.setAccessible(true);
ensureCapacity.invoke(saas, (Object [])null);
assertEquals(0,saas.size());
saas.add("String One");
assertEquals(1,saas.size());
saas.add("String Two");
assertEquals(2,saas.size());
ensureCapacity.invoke(saas, (Object [])null);
assertEquals(2,saas.size());
```

Den Cast von *null* auf *Class<?> []* bzw. *Object []* empfahl Eclipse. Dies erscheint auf den ersten Blick unsinnig, da *null* von jedem Typ ist. Angesichts der Unterschiede beim Aufruf der Methode *isArray* erscheint dieser Hinweis aber durchaus sinnvoll.

19.7 Etwas über Interna

Vielleicht sind Sie bei der Lektüre der Java API-Dokumentation über eine Methode wie *isSynthetic* gestolpert. Nach Java Sprachspezifikation [GJS+11] Kap. 13.1, Nr. 7, müssen fast alle Konstrukte, die der Java-Compiler einführt, die keine Entsprechung im Source-Code haben, als *synthetic* gekennzeichnet werden. Lediglich folgende Elemente sind davon ausgenommen:

1. Default-Konstruktoren,
2. Die Klasseninitialisierungsmethode.
3. Die *values* und *valueOf*-Methode der Klasse *Enum*.

Für den ersten Fall finden Sie ein Beispiel in *reflection.Constructor01*. *reflection.UseConstructor01* zeigt, dass der generierte Default-Konstruktor nicht synthetisch ist. Die Klasseninitialisierungsmethode ist auch mit reflektiven Mitteln nicht zugänglich. Sie „sehen“ sie aber etwa im Beispiel *crash.CounterV06.class*. Sie heißt *<clinit>* und initialisiert das Klassenattribut. Hat die Klasse keine Klassenattribute, so wird diese Methode auch nicht generiert. Die entsprechende Methode zum Initialisieren der Instanzattribute heißt entsprechend *<init>*

Was für Methoden sind nun synthetisch? Ein Beispiel zeigt *reflection.Anonymous01*: Ich erzeuge dort ein *Comparable<String>* Objekt mittels einer anonymen Klasse. Das Entscheidende dabei ist die Tatsache, dass es sich um einen parametrisierten Typ handelt. Aufgrund der Typauslöschung steht in der Datei *Comparable.class* nur *java.lang.Object*:

```
00000000: cafe babe 0000 0033 000a 0100 1528 4c6a  .......3.....(Lj
00000010: 6176 612f 6c61 6e67 2f4f 626a 6563 743b  ava/lang/Object;
00000020: 2949 0100 0628 5454 3b29 4901 0009 5369  )I...(TT;)I...Si
00000030: 676e 6174 7572 6501 0009 636f 6d70 6172  gnature...compar
00000040: 6554 6f01 0014 6a61 7661 2f6c 616e 672f  eTo...java/lang/
```

```
00000050: 436f 6d70 6172 6162 6c65 0100 106a 6176  Comparable...jav
00000060: 612f 6c61 6e67 2f4f 626a 6563 7407 0005  a/lang/Object...
00000070: 0700 0601 0028 3c54 3a4c 6a61 7661 2f6c  .....(<T:Ljava/l
00000080: 616e 672f 4f62 6a65 6374 3b3e 4c6a 6176  ang/Object;>Ljav
00000090: 612f 6c61 6e67 2f4f 626a 6563 743b 0601  a/lang/Object;..
000000a0: 0007 0008 0000 0000 0001 0401 0004 0001  ................
000000b0: 0001 0003 0000 0002 0002 0001 0003 0000  ................
000000c0: 0002 0009                                ....
```

Die anonyme Klasse namens *Anonymous01$1* deklariert eine Methode

```
public int reflection.Anonymous01$1.compareTo(java.lang.String)
```

Der Compiler generiert aber eine synthetische Methode

```
public int reflection.Anonymous01$1.compareTo(java.lang.Object)
```

So wird dafür gesorgt, dass Spezialisierung wie erwartet funktioniert.

19.8 Historische Anmerkungen

19.9 Aufgaben

1. Warum benötigt man die Methoden *newInstance* von *Class* bzw. *Constructor*? Sie könnten doch einfach *new* verwenden, oder?

2. Welchen Wert (*true/false*) liefert der *instanceof*-Operator in den folgenden Ausdrücken? Welche der folgenden Ausdrücke führen zu einem Compiler-Fehler oder -Warnung und welche nicht? Begründen Sie bitte Ihre Antworten!

   2.1.
   ```
   Number in = 0;
   String s = in instanceof Integer?"":"k";
   ```

   2.2.
   ```
   String s = in instanceof Long?"":"k";
   ```

   2.3.
   ```
   String s = in instanceof Number?"":"k"
   ```

   2.4.
   ```
   Object io = 0;
   String s =io instanceof Integer?"":"k";
   ```

   2.5.
   ```
   String s =io instanceof Long?"":"k";
   ```

   2.6.
   ```
   String s = io instanceof Number?"":"k"
   ```

   2.7.
   ```
   public class Muehle extends AbstractRegularGame<Pair<Byte,Byte>>{...}
   Object muehle = new Muehle();
   String s = muehle instanceof AbstractRegularGame<?>?"":"k";
   ```

   2.8.
   ```
   String s = muehle instanceof AbstractRegularGame<Pair<Byte,Byte>>?"":"k";
   ```

   2.9.
   ```
   String s = muehle instanceof AbstractRegularGame<Pair<?,?>>?"":"k";
   ```

   2.10.
   ```
   Object ci = Integer.class;
   String s = ci instanceof Class<?>?"":"k:
   ```

   2.11.
   ```
   Object ci = Integer.class;
   String s = ci instanceof Class?"":"k";
   ```

   2.12.
   ```
   Object listInt = new LinkedList<Integer>();
   String s = listInt instanceof List<?>?"":"k";
   ```

3. Welche Elemente bilden das Metamodell zur Beschreibung von Java-Klassen?

4. Geben Sie bitte je ein Beispiel für die sechs Arten von reifiablen Typen!

5. Was gibt die folgende Methode auf der Console aus?

```
public class Reflector {
public static void main(String[] args) throws Exception {
Set<String> s = new HashSet<String>();
s.add("foo");
Iterator<String> it = s.iterator();
Method m = it.getClass().getMethod("hasNext");
System.out.println(m.invoke(it));
}
}
```

   5.1. Begründen Sie bitte, warum diese Ausgabe erfolgt!

   5.2. Wie muss der Code verändert werden um die (vielleicht) von Ihnen erwartete Ausgabe zu liefern?

6. Sie sollen eine Collection-Klasse *ExoticCollection<E>* testen, die ihre Elemente in einem Array

```
private Object [] entries;
```

   speichert. Die Ursache für bereits aufgefallene Fehler vermuten Sie in einer Methode

```
private void ensureCapacity();
```

   Diese wird von Einfüge-Methoden aufgerufen, wenn das Array *entries* voll ist und vergrößert es auf das Doppelte. Die Anfangskapazität ist 8.

   6.1. Geben Sie bitte Testfälle an, mit denen Sie diese Methode gerne testen würden. Begründen Sie bitte die Wahl Ihrer Testfälle!

   6.2. Schreiben Sie bitte Code, der die Methode *ensureCapacity* mit Ihren Testfällen testet. Ob Sie dabei *JUnit* verwenden oder nicht, ist Ihnen freigestellt.

Kapitel 20

Annotationen

20.1 Übersicht

*Annotation*en verlagern einen größeren Teil der Arbeit in die Spezifikation und ermöglichen es, diese direkt im Code zum Ausdruck zu bringen. Definiert werden sie als *Annotationstyp*en. Diese kann man ähnlich wie Interfaces charakterisieren. Sie dienen dazu, Deklarationen zu annotieren (ergänzen), stellen also Metainformation dar. Sie haben keinen Einfluss auf die Semantik des Java-Codes. Sie dienen als Input für verschiedene Tools. Wie auch generische Klassen, dienen Annotationen dazu, die Intentionen beim Programmieren explizit zu formulieren. In Verbindung mit der Möglichkeit in Java eigene Class-Loader zu schreiben sind Annotationen ein wichtiges Mittel, um Java weitere Funktionalität hinzuzufügen.

Annotationen bzw. Annotationstypen haben vielfältige Einsatzmöglichkeiten. Konkrete Anwendungen finden Sie in bewährten Frameworks, wie *JUnit*, *Hibernate* uvam. Für die tatsächliche Auswertung zur Laufzeit wird *Reflection* eingesetzt, siehe hierzu Kap. 19. Ich hoffe in späteren Auflagen im Abschn. 20.8 weitere Verwendungsmöglichkeiten vorstellen zu können.

20.2 Lernziele

- Die in Java definierten Annotationen kennen und verwenden können.
- Weitere Annotationen verstehen können.
- Eigene Annotationen schreiben können.
- Annotationen in eigenen Anwendungen verarbeiten können.

20.3 Einführung

Haben Sie diesen Kurs bzw. dieses Skript von Anfang an verfolgt bzw. gelesen, so sind Ihnen schon einige Annotationen begegnet. Aus Java direkt haben Sie die folgenden Annotationen kennengelernt:

@Override Signalisiert dem Compiler, dass tatsächlich eine Methode überschrieben werden soll. Ist dies nicht der Fall, so gibt es dann einen Compiler-Fehler.

@SuppressWarnings Unterdrückt gezielt Compiler-Warnungen in geeigneten Situationen.

Aus JUnit kennen Sie z. B.

@Test Charakterisiert Methoden, die als Test ausgeführt werden sollen.

@Before Charakterisiert die Methode oder Methoden, die vor jedem Testfall ausgeführt werden sollen.

@After Charakterisiert die Methode oder Methoden, die nach jedem Testfall ausgeführt werden sollen.

In diesem Kapitel gebe ich eine systematische Einführung in den Umgang mit Annotationen. Sie sind ein wichtiges Element um in Java-Code präzise und überprüfbar Designentscheidungen zu dokumentieren. Im Einzelnen:

- Designentscheidungen können explizit zum Ausdruck gebracht werden.
- Werkzeuge können diese Entscheidungen überprüfen.
- Werkzeuge, wie JUnit, Hibernate etc. werden dadurch einfacher oder überhaupt erst möglich.

20.4 Annotationstypen

Annotationstypen werden ähnlich wie Interfaces definiert: [GJS+14]

```
public @interface AnnotationTypeName AnnotationBody
```

Zulässige Sichtbarkeiten für Annotationstypen sind *public* und *package*. Direkter Obertyp jedes Annotationstyps ist das Interface *java.lang.annotation.Annotation*. Für Annotationstypen können Sie nur die Methoden aus diesem Interface verwenden:

- annotationType()
- equals
- hashCode
- toString

Die anderen Elemente aus *Object* stehen nicht zur Verfügung. Ansonsten gelten alle Regeln für Interfaces auch für Annotationstypen, insbesondere haben sie den gleichen Namensraum wie Interfaces, können überall deklariert werden, wo Interfaces deklariert werden können und haben den gleichen Gültigkeitsbereich und die gleichen Sichtbarkeiten.

Ein *Annotationstyp* kann Methodendeklarationen enthalten, die jeweils ein Element des Annotationstyps definieren:

```
AnnotationTypeBody:
{ {AnnotationTypeMemberDeclaration} }
```

Die Elemente eines *Annotationstypen* haben folgende Form:

```
AnnotationTypeMemberDeclaration
  AnnotationTypeElementDeclaration
  ConstantDeclaration
  ClassDeclaration
  InterfaceDeclaration

AnnotationTypeElementDeclaration:
  {AnnotationTypeElementModifier} UnannType Identifier ( ) [Dims] [DefaultValue]
```

Aufgrund der Definition kann eine *AnnotationTypeElementDeclaration* weder formale Parameter noch Typ-Parameter haben und auch keine *throws* Klausel. Für *Dims* gilt:

```
Dims:
   {Annotation} [ ] {{Annotation} [ ]}
```

Als *AnnotationTypeElementModifier* sind *public* oder *abstract* möglich.
Der Rückgabetyp *UnannType* kann folgende Werte haben:

- Primitive Typen
- *String*
- *Class* (oder eine Invocation of Class).
- Ein enum-Typ.
- Ein Annotationstyp.
- Ein Array-Typ einer der genannten Typen.

Definition 20.4.1 (Normale, Marker und Single Element Annotationstypen)
Normal Annotationstypen wie oben heißen *normale Annotationen*.

Marker Ein Annotationstyp, bei dem alle Elemente einen default-Wert haben. Insbesondere Annotationstypen ohne Elemente.

Single Element Ein Annotationstyp mit genau einem Element. Gemäß Konvention heißt dieses Element *value*.
◀

Hier nun einige Beispiele:
Java hat einige vordefinierte Annotationstypen, die zur Definition von Annotationstypen eingesetzt werden:

@Target Gibt an, für welche Arten von Elementen eine Annotation dieses Annotationstyps verwendet werden kann. Die zulässigen Arten von Elementen sind im Enum *ElementType* aus dem Paket *java.lang.annotation* definiert. In Beispiel 20.5.5 also *METHOD*. Ist kein Target angegeben, so ist die Annotation auf alle Elementtypen anwendbar.

@Retention Gibt an, in „wie lange“ Annotationen dieses Annotationsyps erhalten bleiben. Die zulässigen Werte sind im Enum *RetentionPolicy* aus dem Paket *java.lang.annotation* definiert.

@Repeatable Gibt an, das Annotationen dieses Annotationstyps mehrfach für ein zulässiges Element verwendet werden können. Hier muss ein zugehöriger Container-Annotationstyp angegeben werden.

@Inherited Gibt an, dass Annotationen dieses Annotationstyps an Unterklassen vererbt werden. Sie wirkt nur auf Klassen. Sie hat keine Wirkung bei implementierten Interfaces

Bemerkung 20.4.2 (Präzision und natürliche Sprache)
Beim Sprechen wird meistens nicht zwischen „Annotation“ und „Annotationstyp“ unterschieden. Wird diese Unterscheidung gemacht, so werden die Sätze aber etwas komplizierter, wie sich wieder an der Liste der vordefinierten Annotationstypen zeigt. Aber ich habe es oft erlebt, dass die Verwendung von nicht präzise definierten Begriffen zu mehr Problemen geführt hat als etwaige grammatikalische Verständnisschwierigkeiten. ◀

Die möglichen Werte für den Element-Typ, für den eine Annotation verwendet werden kann (Target), sind im Enum *ElementType* definiert:

- ANNOTATION_TYPE: Verwendbar für Annotationstypen.
- CONSTRUCTOR
- FIELD

- LOCAL_VARIABLE
- METHOD
- PACKAGE
- PARAMETER
- TYPE: Klasse, Interface oder Enum.
- TYPE_PARAMETER: Verwendbar für Typ-Parameter in Definitionen von generischen Elementen.
- TYPE_USE: Verwendbar für den konkreten Typ, der in einem parametrisierten Element verwendet wird.

Die Namen sind weitestgehend selbsterklärend, deshalb habe ich nur einige weiter erläutert.

Die möglichen *Retention*s sind im Enum *RetentionPolicy* definiert:

SOURCE Diese Annotationen werden nur vom Compiler verwendet und werden nicht Bestandteil der .class Datei.

CLASS Diese Annotationen werden in die .class Datei geschrieben. Dies ist die Default-Retention Policy. Annotationen dieser Art müssen von der VM nicht erhalten werden. Sie können von ClassLoadern verarbeitet werden.

RUNTIME Diese Annotationen werden in die .class Datei geschrieben und von der VM geladen. Sie können reflexiv gelesen werden.

Der default Wert für die *Retention* ist *CLASS*, [GJS+11].

Im Laufe der Entwicklung möchte ich für jede dieser Werte für `@Target` und `@Retention` sinnvolle Beispiele vorzustellen. Noch ist diese Sammlung nicht vollständig.

Beispiel 20.4.3 (Annotation Type Annotation)
`@Target`, `@Retention` und `@Inherited` sind zwei in Java vordefinierte Annotationstypen, die (nur) für Annotationstypen verwendbar sind. ◄

Beispiel 20.4.4 (Marker Annotation)
Typische Marker-Annotationen sind *@Override* und *@Serializable*. Mit den heutigen Möglichkeiten in Java wäre auch ein Annotationstyp *@Cloneable* gut als Alternative zum Interface *Cloneable* geeignet. ◄

Beispiel 20.4.5 (Single Element Annotation)
Hier ein Beispiel aus [GJS+14]:

```
interface Formatter {}
// Designates a formatter to pretty-print the annotated class
@interface PrettyPrinter {
Class<? extends Formatter> value();
}
```

◄

20.5 Annotationen

Verwendet werden Annotationen so, wie im Folgenden beschrieben.

AnnotationType:

@ TypeName (ElementValuePairs optional)

Handelt es sich um eine Markerannotation oder haben alle Elemente default Werte, die passen, so brauchen die Elemente-Wert-Paare nicht agegeben werden.

ElementValuePairs:

ElementValuePair

ElementValuePairs , ElementValuePair

ElementValuePair:

Identifier = ElementValue

ElementValue:

ConditionalExpression

Annotation

ElementValueArrayInitializer

ElementValueArrayInitializer:

ElementValues optional, optional

ElementValues:

ElementValue

ElementValues, ElementValue

Eine Annotation muss unmittelbar vor dem Element stehen, das sie annotiert. Es ist üblich, die Annotation vor alle anderen Modifier zu schreiben.

In Java können folgende Elemente annotiert werden: Alle Deklarationen, also *Paket*, *Klasse*, *Methode*, *Attribut (Field)*, *Parameter*, *Parameter*, *Konstruktor*, *Enum*, *Interface* und *Typ*, *Typ-Parameter* und *Typverwendung*.

Man kann also Deklarationsannotationen und Typannotationen unterscheiden. Um die beiden Dinge auseinander zu halten, dient die folgende Definition.

Definition 20.5.1 (Deklarations- und Typ-Annotation)
Eine *Deklarations-Annotation* ist eine Annotation, die sich auf eine Deklaration bezieht. Eine *Typ-Annotation* ist eine Annotation, die sich auf einen Typ oder einen seiner Bestandteile bezieht. [GJS+14] ◀

Erste beziehen sich auf eine Deklaration. Dies sind alle, bis auf die letzten beiden oben aufgezählten. Letztere beziehen sich auf einen Typ an einer Stelle, an der verwendet wird bzw. auf einen Typ-Parameter in einer generischen Klasse oder Methode bzw. einem generischen Interface.

Eine *package* Annotation darf höchsten in einer package Deklaration vorkommen. Es wird empfohlen dies in der Datei *package-info.java* in dem Verzeichnis zu tun, in dem sich die Source-Dateien des Pakets befinden. Diese legen Sie bei Bedarf an. Beispiele für *package-info.java* finden Sie in allen Paketen, in denen ich Beispiele zur Verfügung stelle.

Beispiel 20.5.2 (Marker Annotation)
Das JPA (Java Persistence API) definiert die Annotation *Entity*:

```
@Documented
@Target(value=TYPE)
@Retention(value=RUNTIME)
public @interface Entity {
     String name() default "";
}
```

Annotationen können selber wieder annotiert werden. *Entity* ist als *Dokumented* (s. u.) gekennzeichnet, kann für *Typen* verwendet werden und steht zur Laufzeit (*Retention RUNTIME*, s. u.) zur Verfügung.

Dies ist eine Marker Annotation. Deshalb kann das Klammerpaar „()“ nach *Entity* entfallen und sie kann einfach so verwendet werden:

```
@Entity
public class Address {
 ...
```

◀

Beispiel 20.5.3 zeigt, wie normale Assoziationen definiert und verwendet werden.

Beispiel 20.5.3 (Normale Annotation)
Für Attribute und Methoden definiert das JPA die Annotation

```
@Target(value={METHOD,FIELD})
@Retention(value=RUNTIME)
public @interface GeneratedValue  {
    /**
     * (Optional) The primary key generation strategy
     * that the persistence provider must use to
     * generate the annotated entity primary key.
     */
    GenerationType strategy() default AUTO;

    /**
     * (Optional) The name of the primary key generator
     * to use as specified in the {@link SequenceGenerator}
     * or {@link TableGenerator} annotation.
     * <p> Defaults to the id generator supplied by persistence provider.
     */
    String generator() default "";
}
```

Hier handelt es sich um eine „normale“ Annotation. Die Werte der Elemente müssen also angegeben werden, wenn ihre Auswahl nicht dem Persistence Provider überlassen werden soll: Hier ein Beispiel für Ihre Verwendung aus dem Kontext einer Entity-Klasse *Address*:

```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "ADDRESSGEN")
@SequenceGenerator(name = "ADDRESSGEN", sequenceName = "ADDRESSSEQ")
public long getId() {
        ...
```

◀

Beispiel 20.5.4 (Single Element Annotation)
Die Annotation *SuppressWarnings* aus *java.lang* ist eine Single Element Annotation. Ich kann z. B. einfach schreiben:

```
@SuppressWarnings("serial")
public class ClassTreeView extends JFrame{
...
```

◀

Beispiel 20.5.5 (@Override)
Dies Beispiel zeigt die Deklaration von *@override*

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
}
```

Diese Annotation weist den Compiler darauf hin, dass die folgende Methode eine andere überschreibt, die in der Klassenhierarchie höher steht. Vor Methoden, die Operationen aus einem Interface implementieren, ist sie überflüssig, dies erkennt der Compiler auch so. Die Annotation `@Target` gibt an, für welche Art Element die Annotation verwendet werden kann, hier also *Method*. Die Annotation `@Retention` gibt an, auf welcher Ebene die Annotation verfügbar ist. Hier ist dies im Source-Code.

Das klassische Beispiel für die `@Override` Annotation liefert die Methode *equals*. Diese Methode ist in Object so deklariert und implementiert:

```
public boolean equals(Object obj) {
   return (this == obj);
}
```

Ein unerfahrener Programmierer könnte in einer Unterklasse *Fu* nun folgende Methode schreiben:

```
public boolean equals(Fu obj) {
   return this.compareTo(obj)==0;
}
```

Damit wird die Methode *equals* aus *Object* aber nicht *überschrieben*, sondern *überladen*. Das muss kein Fehler sein, kann aber zu schwer zu entdeckenden Fehlern führen. Wird die Annotation *Override* verwendet, so macht der Programmierer seine Absicht explizit: Er will *equals* aus einer Oberklasse *überschreiben* und der Compiler liefert nun einen Fehler, da stattdessen *überladen* wird.

Die Annotation *@Override* verhindert auch einen anderen Fehler wirkungsvoll: Klassenmethoden können nicht überschrieben werden. Versuchen Sie dies trotzdem so wird dies erkannt oder auch nicht:

1. Deklarieren Sie die Methode mit *@Override*, so bekommen Sie einen Compilerfehler.
2. Andernfalls haben Sie einfach eine weitere, andere Klassenmethode in der Unterklassen. Überschrieben wird nicht.

Bei dieser Annotation sehen Sie auch wieder die Elemente *@Target* und *@Retention*.

Der Annotationstype *@Documented* spezifiziert, dass das jeweilige Element mit Javadoc oder ähnlichen Tools dokumentiert werden soll. Durch die Deklaration mit *@Documented* wird das annotierte Elemente Bestandteil der öffentlichen Schnittstelle der Klasse. ◀

Eine weitere Standard-Annotation ist *@Deprecated*

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(value={CONSTRUCTOR,FIELD,LOCAL_VARIABLE,METHOD,PACKAGE,PARAMETER,TYPE})
public @interface Deprecated {}
```

Mir ihr charakterisieren Sie Elemente, die nicht (mehr) verwendet werden sollten, weil sie gefährlich sind oder es bessere Alternativen gibt. Es gibt eine Compiler-Warnung, wenn ein *deprecated* Element verwendet wird oder in nicht *deprecated* Code überschrieben wird.

Die Annotation, die Sie als erste kennengelernt haben ist `@SuppressWarnings`, die ich zunächst verboten hatte. Hier nun die Einzelheiten und einige sinnvolle Anwendungsbeispiele.

```
@Target(value={TYPE,FIELD,METHOD,PARAMETER,CONSTRUCTOR,LOCAL_VARIABLE})
@Retention(value=SOURCE)
public @interface SuppressWarnings{
   String[] value();
}
```

Beispiel 20.5.6 (@SuppressWarnings)
Die Swing-Klassen kommen mit rudimentärer Serialisierungsunterstützung. Diese wird aber mit zukünftigen Swing-Versionen nicht kompatibel sein. Seit Java 1.4 ist die Unterstützung für alle Java Beans im Paket *java.beans* enthalten. Die Klasse XMLEncoder ermöglicht eine Speicherung einer Textdarstellung von Java Beans. Es ist daher legitim bei den Swing-Klassen die Annotation `@SuppressWarnings("serial")` zu verwenden.

Sie bringen damit eine Entscheidung explizit zum Ausdruck: Wer Objekte „Ihrer“ Swing-Klassen speichern will, sollte dies Mittels *XMLEncoder* machen. Wer es mit Serialisierung machen will, hat selber Schuld. ◄

Annotationen können wiederholt werden. Analog zum *javadoc* Tag *@author* kann also auch eine *Annotation @Author* mehrfach für ein Element verwendet werden, wenn Sie durch die *Annotation @Repeatable* gekennzeichnet ist. Im Einzelnen geht das so, wie hier an einem Beispiel gezeigt:

Beispiel 20.5.7 (Wiederholte Annotation)
Die folgende Annotation stellt in der Source analoge Information zur Verfügung, wie das *javadoc* tag *@author*:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.SOURCE)
@Repeatable(Authors.class)
public @interface Author {
    Name [] value;
}
```

Hier wird eine weitere Annotation verwendet, um Vor- und Nachnamen systematisch verwenden zu können:

```
public @interface Name {
   String firstName();
   String lastName();
}
```

Der Parameter für die *@Repeatable Annotation* ist die *.class*-Datei der „umschließenden Annotation“ *Authors*:

```
@Target(ElementType.TYPE)
public @interface Authors {
   Author [] value();
}
```

Ändern Sie die *Retention* auf *RUNTIME*, so stehen die Informationen auch zur Laufzeit zur Verfügung. ◄

20.6 Deklarationsannotationen und Typannotationen

In diesem Abschnitt beschreibe ich die Verwendung von Annotationen für Typ-Parameter und für die Verwendung für von Typen. Es geht also nicht um Annotationen für Klassen, enums oder Interfaces einschließlich Annotationen.

Hier einige Beispiele: Die folgende *Map*

```
Map<@NonNull String, @NonEmpty List<@Readonly Document> files;
```

soll als Schlüssel einen von *null* verschiedenen *String* haben und als Wert eine nicht-leere *List* von read-only *Document*s.

Bei der Annotation von Array-Typen müssen Sie genau hinsehen. In der folgenden Zeile

```
@Readonly Document [] docs0;
```

wirnd ein Array von read-only *Documents* deklariert. Die Annotation *@ReadOnly* bezieht sich auf das Element, vor dem sie direkt steht und das ist hier der Typ *Document*. Die nächste Zeile

```
Document @Readonly [] docs1;
```

deklariert ein eine read-only Array von *Documents*. Sie steht „direkt vor“ dem Array-Typ *Document [ ]*.

Nun noch ein Beispiel für ein Beispiel für eine Annotation bei einem zweidimensionalem Array:

```
Document @NonEmpty[] @Readonly [] docs2;
```

@NonEmpty bezieht sich auf den Typ *Document [ ][ ]*, deklariert also ein nicht-leeres Array von *Document [ ]*s. *@Readonly* bezieht sich auf *Document [ ]*. Insgesamt wird also ein nicht-leeres Array von read-only *Document [ ]* deklariert. Bei den hier zur Illustration verwendeten Annotationen, sollten Sie sich aber überlegen, ob das Beispiel wirklich sinnvoll ist!

20.7 Weitere in Java definierte Annotatonen

Mit *funktionalen Interfaces* (siehe Def. 16.3.1) kommt die ensprechende Annotation *@Functional-Interface*:

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface FunctionalInterface {}
```

Mit dieser Annotation mache Sie klar, dass das von Ihnen geschriebene Interface genau eine abstrakte Methode hat und nicht mehr haben soll. Es ist also ein SAM (Single Abstract Method) Interface und soll dies auch bleiben. Programmierer, die weiter an diesem Interface arbeiten, wissen dann, dass sie keine weitere abstrakte Methode hinzufügen dürfen. Versuchen Sie es doch, so gibt es einen Compiler-Fehler.

Hier folgt eine Erläuterung einiger (vieler) der *Annotation*en, die in Java vordefiniert sind (geordnet nach Paket):

java.lang

@Deprecated Kennzeichnet ein Element, das nicht mehr verwendet werden soll. Der Grund ist meistens, dass, die Nutzung gefährlich ist oder es (inzwischen) bessere Alternativen gibt.

@FunctionalInterface Kennzeichnet ein funktionales Interface, ist aber nicht notwendig, um Interfaces zu charakterieren, die das Ziel von *λ-Ausdrücken* sein können.

@Override Diese *Annotation* bringt explizit zum Ausdruck, dass eine *Methode* eine *Methode* einer Oberklasse *überschreiben* soll. Durch sie wird irrtümliches *Überladen* wirkungsvoll vermieden.

@SafeVarargs Unter bestimmten Bedingungen ([GJS+14], §4.12.2, 5.1.9) können variabel lange Parameterlisten zu „Verschmutzung“ des Heaps und Compiler-Warnungen führen. Sind Sie sich ganz sicher, dass diese Warnungen ignoriert werden können, so bringen Sie dies durch diese *Annotation* zum Ausdruck.

@SuppressWarnings Diese Annotation unterdrückt gezielt Warnungen des Compilers bei bestimmten möglicherweise problematischen Konstrukten. Setzen Sie dies bitte ausschließlich dann ein, wenn Sie sich ganz sicher sind, das die Warnung unberechtigter Weise erfolgt.

java.lang.annotation

@Documented Gibt an, das dieses Element per default mit javadoc oder einem ähnlichen Werkzeug dokumentiert werden muss. So annotierte Elemente werden Bestandteil des öffentlichen APIs dieser Elemente.

@Inherited Gibt an, das *Annotationen* dieser Klasse von Unterklassen geerbt werden sollen.

@Native Eine *Annotation* für Attribute (Fields), die Konstanten definieren. Sie gibt an, das native Code auf diese Attribute zugreifen kann. Sie ist gedacht für Werkzeuge, die Header-Dateien erzeugen.

@Repeatable Gibt an, das eine *Annotation* mehrfach für ein Element verwendet werden kann und gibt die Annotation an, die die Informationen enthält.

@Retention Gibt an, ob die *Annotation* im Source-Code, in der Klassendatei oder zur Laufzeit verfügbar ist.

@Target Gibt an, für welche Elemente eine *Annotation* verwendet werden kann.

javax.annotation

@Generated Kennzeichnet Code-Elemente, die generiert wurden, z. B. von einem GUI-Builder.

@PostConstruct Dient der Unterstützung von *dependency injection*. Für Einzelheiten verweise ich auf die API-Dokumentation dieser Annotation.

@PreDestroy Das Analogon zu *@PostConstruct* kennzeichnet die Methode, die vor Entfernen des eines Objekts aus dem Container noch aufgerufen werden muss.

@Resource Markiert eine Ressource, die eine Anwendung benötigt.

@Resources Die umschließende *Annotation*, die die Resourcen enthält.

javax.annotation.processing

@SupportedAnnotationTypes Markiert die *Annotationen*, die ein Annotations-Prozessor unterstützt.

@SupportedOptions Gibt an, welche Optionen ein Annotations-Prozessor unterstützt.

@SupportedSourceVersion Gibt an, welche Java-Versionen ein Annotations-Prozessor unterstützt.

javax.persistence Siech auch Kap. 27.

@Entity Objekte dieser Klasse entsprechen Entities im Sinne eines Entity Relationship Modells. D

@Table Spezifiziert die Tabelle, deren Zeilen den Objekten der Entity entsprechen.

@Column Spezifiziert die Spalte für eine persistente Eigenschaft oder Attribut.

@Basic Definiert die Abbildung von einem Datenbank-Typ auf einen Java-Typ. Lässt sich anwenden auf primitive Typen, die zugehörigen Wrapperklassen, String, BigInteger, BigDecimal, java.sql.Date, java.sql.Time, java.sql.Timestamp, byte[], Byte[], char[], Character[], enums und alle anderen Typen, die Serializable implementieren. Also auch für die in Java 8 neuen Klassen aus dem Paket java.time, wie LocalDate etc.

Ein Beispiel:

Beispiel 20.7.1 (SafeVarargs)

```
public class HeapPollution01 {
    static void m(List<String>... stringLists) {
        Object[] array = stringLists;
        List<Integer> tmpList = Arrays.asList(42);
        array[0] = tmpList; // (1)
        String s = stringLists[0].get(0); // (2)
        System.out.println(s);
    }

    public static void testM() {
        List<String> list1 = Arrays.asList("A");
        List<String> list2 = Arrays.asList("B");
        m(list1, list2);
    }

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

Hier gibt es die Warnungen „Type safety: Potential heap pollution via varargs parameter stringLists“, „Type safety: A generic array of List<String> is created for a varargs parameter“. Zur Laufzeit gibt es eine *ClassCastException*. Hier ist die Annotation *SafeVarargs* definitiv nicht angebracht. [GJS+14] ◄

20.8 Annotationsprozessoren

Im Zusammenhang mit den Reflection-Fähigkeiten von Java können Annotationen einfach genutzt werden. Ich empfehle die Nutzung von JUnit. Auch in den Oracle-Dokumenten zu Java finden Sie entsprechende Beispiele, etwa in http://docs.oracle.com/javase/1.5.0/docsslash guides/language/annotations.html. Dieses Dokument stammt aus der Java Version 1.5, in der Annotationen wingeführt wurden.

Eine Annotation wie Test kann so aussehen:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test {
}
```

Hier eine einfache Klasse, analog zu den JUnit-Testklassen:

```
public class CounterTest {
   simple.Counter counter = new Counter();
   @Test
   public void testIncrement(){
      counter.increment();
```

```
      if(counter.show()!=1){
         throw new RuntimeException("Expected "+1+" but was " + counter.show());
      }
   }
   @Test
   public void testDecrement(){
      counter.decrement();
         if(counter.show()!=1){
            throw new RuntimeException("Expected "+1+" but was " + counter.show());
         }
   }
}
```

Mittels Reflection (siehe Kap. 19) können die mit dieser Annotation versehenen Methoden aufgerufen werden, etwa so:

```
public class RunTests {
   public static void main(String [] args){
      int passed = 0;
      int failed = 0;
      CounterTest ct = new CounterTest();
      for (Method m : ct.getClass().getMethods()) {
         if (m.isAnnotationPresent(Test.class)) {
            try {
               m.invoke(ct);
               passed++;
            } catch (Throwable ex) {
               System.out.printf("Test %s failed: \n%s %n", m, ex.getCause());
               failed++;
            }
         }
      }
      System.out.printf("Passed: %d, Failed %d%n", passed, failed);
   }
}
```

Dem gesamten Code finden Sie im Paket *annotations* in meinem Pub.

Analog zu Marker-Interfaces gibt es auch Marker-Annotationen. Dies sind Annotationen ohne Elemente. Die Java Sprachspezifikation [GJSB05] nennt auf S. 275 als Beispiel eine Annotation *Preliminary*

```
/**
* Annotation with this type indicates that the specification of the
* annotated API element is preliminary and subject to change.
*/
public @interface Preliminary { }
```

20.9 Historische Anmerkungen

Annotationen kamen mit Java 5 im Herbst 2004. *@FunctionalInterface* und wiederholbare Annotationen kamen mit Java 8 im März 2014. Ebenfalls mit Java 8 kamen Annotationen für Parameter, Typ-Parameter und Typen hinzu.

Die Wirkung der Annotation *@Override* änderte sich nach Java SE 5.0. Seitdem gibt es einen Compiler-Fehler, wenn eine so annotierte keine Methode eines Supertyps oder nicht override-äquivalent zu einer Methode in *Objekt* ist. Der zweite Halbsatz fehlte in der ursprünglichen Implementierung.

20.10 Aufgaben

1. Stellen Sie die Vor- und Nachteile der Verwendung der Annotation *@Deprecated* im Vergleich mit Javadoc-Kommentar *@depricated* dar!

2. Was ist an der Deklarationen dieser beiden Annotationen falsch?

   ```
   @interface Ping { Pong value(); }
   @interface Pong { Ping value(); }
   ```

3. Schreiben Sie bitte eine Annotation um einen Copyright-Vermerk in Code einzufügen, der nicht nur dort steht, sondern auch im compilierten Byte-Code verfügbar ist und zur Laufzeit abgefragt werden kann!

4. Schreiben Sie bitte eine Annotation um den Autor einer Klasse festzuhalten!

Kapitel 21

Konfigurationen

21.1 Übersicht

Es gibt viele Möglichkeiten, Java Code an verschiedene Situationen anzupassen. In diesem Kapitel werden einige erläutert. Dazu gehören sehr verschiedene Dinge.

In vielen Fällen wollen Sie eine Anwendung z. B. in verschiedene Sprachen verfügbar machen.

21.2 Lernziele

- Einige JVM-Parameter kennen.
- Einige Einzelheiten des Java Compilers *javac* kennen und verwenden können.
- Mit Resourcen umgehen können, wie z. B. *String*s für Nachrichten.

21.3 Virtualmachine

Beim Starten der JVM können viele Parameter gesetzt werden. Ein Beispiel ist der heap space. Hier dieses und einige weitere Beispiele.

1. -XmxNNNNM Maximale Heap Größe
2. -XmsNNNNM Initale Heap Größe
3. -XmnNNNNM Heap Größe for the *young generation*.
4. -XssNNNNk|m Stack space
5. -Xnoclassgc No Class Garbage Collection.
6. -XX:StringTableSize Setzt die Größe der Tabelle für die internerned Strings (Verfügbar seit Java 7).
7. -XX:DisableExplicitGC. Verhindert die Ausführung von System.gc. Oft sinnvoll, da diese Methoden zu unvorhersehbaren Performanceinbußen führen kann.

Nun die Erläuterungen, was das im Einzelnen bedeutet.

-XmnNNNNM Die *young generation* umfasst neu erzeugte Objekte, die seit maximal 2–3 Zyklen des *garbage collectors* existieren.

-Xnoclassgc Standardmäßig entlädt die JVM eine Klasse aus dem Speicher, wenn keine Live-Objekte dieser Klasse mehr vorhanden sind. Dies kann jedoch die Leistung verringern. Wenn Sie Garbage-Collection für Klassen inaktivieren, entfällt der Aufwand für das mehrfache Laden und Entladen derselben Klasse.

Falls die Klasse nicht mehr benötigt wird, wird der Platz, den Klasse im Heap-Speicher belegt, normalerweise für die Erstellung neuer Objekte verwendet. Wenn Sie jedoch eine Anwendung haben, die für Anforderungen jeweils ein neues Objekt einer Klasse erstellt, und Anforderungen für diese Anwendung zufällig eingehen, kann es passieren, dass sobald die eine Anforderung abgeschlossen ist, die normale Garbage-Collection für Klassen diese Klasse bereinigt und den belegten Heap-Speicherplatz freigibt, nur um beim Eingang der nächsten Anforderung die Klasse sofort erneut instanzieren zu müssen. In einem solchen Fall können Sie die Garbage-Collection für Klassen mit dieser Option inaktivieren:

Standardeinstellung:	Garbage-Collection für Klassen aktiviert
Empfohlene Einstellung:	Garbage-Collection für Klassen inaktiviert
Verwendung:	Xnoclassgc inaktiviert die Garbage-Collection für Klassen

(Quelle: http://pic.dhe.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=%2Fcom.ibm.websphere.express.doc /%2Finfo%2Fexp%2Fae%2Ftprf_tunejvm.html)

Weitere JVM-Parameter:

1. -Xrunhprof: Liefert ein Protokoll über den Programmablauf. Dies wird allerdings nicht mehr offiziell unterstützt. Die aktuellee Schnittstelle für derartige Informationen ist Java Virtual Machine Tool Interface (JVM TI).

21.4 Der Java Compiler javac

Der Java Compiler *javac* dient insbesondere der Umwandlung von Java-Sourcecode in Bytecode. Der Aufruf erfolgt mittels:

```
javac <options> <source files>
```

Die Optionen und ihre Parameter beschreibe ich hier im Einzelnen.

Option und Parameter	Erläuterung
@<filename>	Einlesen von Optionen und Dateien aus einer Datei (filename)
-Akey[=value]	Optionen zur Weitergabe an einen Annotationsprocessor
–add-modules <module>(,<module>)*	Aufzulösenden Root-Module, zusätzlich zu den initialen Modulen, oder allen im Module-Pfad, wenn der <module> ALL-MODULE-PATH ist.
–boot-class-path, -bootclasspath <path>	Überschreibt den Standardort der bootstrap .class files.
–class-path, -classpath, -cp <path>	Gibt an, wo Benutzerklassen und Annotations-Prozessoren sind.
-d <directory>	Verzeichnis, in dem generierte .class Dateien gespeichert werden sollen.
-deprecation	Ausgabeverzeichnisse für Source-Dateien, in denen deprecated APIs verwendet werden.
-encoding <encoding>	Spezifiziert den Zeichensatz für die Source-Dateien
-endorseddirs <dirs>	Überschreibt den Speicherort für unterstützte Standard-Pfade.
-extdirs <dirs>	Überschreibt den Speicheror für installierte Erweiterungen.
-g	Generiert alle Debug-Informationen.
-g:lines,vars,source	Generiert ausgewählte Debug-Informationen.

Option und Parameter	Erläuterung
-g:none	Generiert keine Debug-Informationen.o
-h <directory>	Verzeichnis, in dem Header-Dateien für native methods (JNI) gespeichert werden sollen.
–help, -help	Erzeugt die Basis-Informationen über die Nutzung von javac.
–help-extra, -X	Erzeugt die Informationen über die erweiterten Optionen für javac.
-implicit:none,class	Gibt an, ob für implizit referenziert Dateien .class-Dateien erzeugt werden sollen.
-J<flag>	Gibt <flag> direkt an's Laufzeit-System weiter.
–limit-modules <module>(,<module>)*	Grenzt die zu beachtenden Module auf diese ein.
–module <module-name>, -m <module-name>	Compile nur der angegeben Modules, prüfe dabei die timestamps.
–module-path <path>, -p <path>	Gibt den Pfad an, in dem sich die Module der Anwendung befinden.
–module-source-path <module-source-path>	Gibt an, wo die Source-Dateien zu zu verwendeten Module stehen.
–module-version <version>	Gibt die Version der zu verwendenden Module an.
-nowarn	Unterdrückt alle Warnungen.
-parameters	Erzeugt Metadaten für Reflection für Methoden-Parameter.
-proc:none,only	Steuert, ob Annotations-Verarbeitung und/oder Compilierung geschehen soll.
-processor <class1>[, <class2>, . . .]	Namen der Annotations-Prozessoren; übergeht die default Suche.
–processor-module-path <path>	Gibt einen Module-Pfad an, in dem die Annotations-Prozessoren stehen.
–processor-path <path>, -processorpath <path>	Gibt einen Pfad an, in dem die Annotations-Prozessoren stehen.
-profile <profile>	Prüft ob, das verwendete API im angegebenen profile verfügbar ist.
–release <release>	Compile für die angegebenen VM version (6, 7, 8 oder 9)
-s <directory>	Speicherort für generierte Source-Dateien.
-source <release>	Stelle Source-Code-Kompabilität mit der angegebenen Version sicher (6, 7,8 ofer 9).
–source-path <path>, -sourcepath <path>	Gibt an, wo die Input-Source-Dateien stehen.
–system <jdk>|none	Überschreibt Speicherplatz der System-Module.
-target <release>	Erzeugt .class-Dateien for angegebene JVM-Version
–upgrade-module-path <path>	Überschreibt Speicherplatz von upgradeable modules.
-verbose	Umfangreichere Nachrichten über den Compiler-Lauf.
–version, -version	Gibt Versions-Information aus.
-Werror	Beendet javac, wenn eine Warnung auftritt.

Sie werden in der Praxis mit wenigen dieser Optionen auskommen.

21.5 Ressourcen

In vielen Anwendungen benötigt man statische (also nicht-dynamische) Elemente, z. B. Texte. Diese können direkt in den Code geschrieben werden. Dies birgt aber eine Reihe von Problemen:

- Die Konsistenz ist nur organisatorisch sicher zu stellen: Werden an verschiedenen Stellen die

selben Texte benötigt, werden auch die gleichen verwendet? Wenn es in Menus Abkürzungen gibt, z. B. Tastaturkürzel, werden diese konsistent verwendet?

- Die Übersetzung in andere Sprachen ist aufwändig und fehlerträchtig.

Wie bereits in Kap. 12 erläutert, haben diese statischen Elemente höchstens einen Vorteil: Sie zeigen direkt im Code, welche Fehlermeldung an dieser Stelle ggf. ausgegeben wird. Mehr Positives ist darin aber nicht zu sehen.

Um diese Probleme zu vermeiden bietet Java u. a. die Klasse *ResourceBundle* (und *ResourceBundle.Control* mit ihren Unterklassen *ListResourceBundle* und *PropertiesResourceBundle*.

Aber zunächst einmal ganz pragmatisch: Bisher haben Sie wahrscheinlich an vielen Stellen String-Literale in Ihrem Code verwendet. Eclipse bietet Ihnen mittels *Source→Externalize Strings* eine einfache Möglichkeit diese String-Literale an einer Stelle zusammenzufassen. Als Ergebnis erhalten Sie drei Dinge:

1. In Ihrem Code werden alle String-Literale wie

   ```
   String s = "Hallo";
   ```

   durch einen Aufruf der Art

   ```
   String s = Messages.getString("I18NIntro.0"); //$NON-NLS-1$
   ```

   ersetzt. Der Zeilenende-Kommentar „`//$NON-NLS-1$`“ ist nur ein Hinweis an Eclipse, diesen String *nicht* zu externalisieren.

2. Es wird eine Klasse *Messages* generiert:

   ```
   public class Messages {
      private static final String BUNDLE_NAME = "i18n.messages"; //$NON-NLS-1$

      private static final ResourceBundle RESOURCE_BUNDLE = ResourceBundle
                                                            .getBundle(BUNDLE_NAME);

      private Messages() {
      }

      public static String getString(String key) {
         try {
              return RESOURCE_BUNDLE.getString(key);
             } catch (MissingResourceException e) {
                      return '!' + key + '!';
       }
         }
   }
   ```

   Dies ist im Wesentlichen eine Utility-Klasse. Den privaten default Konstruktor können Sie zunächst getrost „vergessen“. Er wird erst dann benötigt, wenn Sie viele String-Literale effizient externalisieren wollen. Ansätze dazu versuche ich in diesem Kapitel unterzubringen, aber eine vollständige Diskussion benötigt mehr als, die Vorlesungen bisher vermitteln wollten.

   Da bisher keine eigene Unterklasse der abstrakten Klasse *ResourceBundle* verwendet wird liefert die Klassenmethode *getBundle* ein Objekt der Klasse *PropertyResourceBundle*. Für das erste Verständnis genügt es zu wissen, dass eine solches Bundle die Werte aus einer Datei mit der Endung .properties liest.

3. Eine Datei *messages.properties* der Form

```
I18NIntro.0=Hallo
I18NIntro.1=Bye Bye
I18NIntro.2=Auf Wiedersehen
```

Dies ist einfach ein sequentielle Datei, aus der die Strings entsprechend der Schlüssel ausgelesen werden. Vor dem Gleichheitszeichen steht der Schlüssel, nach dem Gleichheitszeichen der String, der an Stelle des Schlüssels eingesetzt werden soll. Die Schlüssel sind Strings. Hier wurden Sie generiert. Sie können selbstverständlich andere Namen wählen.

Die Klasse *PropertyResourceBundle* wird nicht spezialisiert. Die Hierarchie

- Basis
  - Country
    - Lanugage
      - Dialect

wird durch die Namenskonventionen der Properties-Dateien hergestellt:

Basis_Country_Language_Dialect

Wird ein String nicht in der jeweiligen *Locale* gefunden, so wird in der nächst höheren Ebene gesucht bis gefunden wird oder — wenn auch auf der obersten Ebene nicht gefunden wird — eine Exception geworfen wird.

Mit ResourceBundles können Anwendungen internationalisiert werden. Sie können damit aber auch einen flexiblen Frame entwickeln. Alle Ressourcen sind hierfür geeignet, die mittels Strings erzeugt werden können. Sie können etwa die Menüs in einer Liste speichern und aus der *properties*-Datei einlesen. Mittels Reflection könnte man das komplett flexibilisieren, dies bietet aber keine weiteren Vorteile.

Die Code-Beispiele *Messages* wie in *I18NIntro* verwendet bieten noch nicht Alles, was Sie brauchen werden.

Hier habe ich die einfachste Variante von ResourceBundle verwendet, *PropertyResourceBundle*. Es gibt auch die abstrakte Klasse *ListResourceBundle*, die zum Speichern der Properties eine Liste verwendet. Für umfangreiche Meldungssammlungen werden Sie die Meldungen nach Bereichen aufgliedern und mittels einer Hash-Struktur verwalten.

Sie können die Meldungen auch durch Parameter individualisieren. Alles was Sie dazu benötigen ist die Methode *format* der Klasse *MessageFormat* aus dem Paket *java.text*. Die Methode *format* erwartet als ersten Parameter eine *String*, der die Formatierung bestimmt. Die Formatierung kann komplex sein, aber auch einfach aus Strings mit eingebetteten Parameternnummern in geschweiften Klammern bestehen. Die Nummern der Parameter beginnen bei 0 und an der jeweiligen Stelle werden die Werte aus dem folgenden Array, genauer deren *String*-Darstellung, eingesetzt.

An der Stelle, wo ein Parameter in der Nachricht erwartet wird, steht im einfachsten Fall einfach „{n}“, z. B. in „Spieler {0} hat in {1} Zügen gewonnen.“. Um diese Nachricht auch mit Werten zu füllen überladen Sie in Ihrer Nachrichtenklasse *Messages* die Methode *getString*:

```
public static String getString(String key, Object[] params) {
    try {
        return MessageFormat.format(this.RESOURCE_BUNDLE.getString(key),
                params);
    } catch (MissingResourceException e) {
        return '!' + key + '!';
    }
}
```

In der API-Dokumentation der Klasse *MessageFormat* finden Sie weitere Details zur Spezifikation der Parameter und dem korrekten Umgang mit Singular und Plural (Klasse *ChoiceFormat*).

21.6 Ausführbare .jar-Dateien

Eine jar-Datei (Java Archive) ist ein Datei im .zip-Format. Darüber hinaus kann sie direkt von einer JVM ausgeführt werden. Dazu müssen Sie alle benötigten .class-Datein und weitere Ressourcen in dieser Datei zusammenfassen. In Eclipse geht das mit der Export-Funktion. Diese Technik funktioniert unter Windows für GUI-Anwendungen, nicht für reine Konsol-Anwendungen.

21.7 Werkzeuge: Troubleshooting

21.7.1 jcmd

You use the jcmd utility to send diagnostic command requests to a running Java Virtual Machine (JVM).

21.7.2 jdb

You use the jdb command and it's options to find and fix bugs in Java platform programs.

21.7.3 jhsdb

You use the jhsdb tool to attach to a Java process or to launch a postmortem debugger to analyze the content of a core dump from a crashed Java Virtual Machine (JVM).

21.7.4 jinfo

Experimental

You use the jinfo command to generate Java configuration information for a specified Java process. This command is experimental and unsupported.

21.7.5 jmap

Experimental

You use the jmap command to print details of a specified process. This command is experimental and unsupported.

21.7.6 jstack

Experimental You use the jstack command to print Java stack traces of Java threads for a specified Java process. This command is experimental and unsupported.

21.8 Historische Anmerkungen

21.9 Aufgaben

1.

Kapitel 22

Java und XML

22.1 Übersicht

XML steht für eXtensible Markup Language. Dies ist eine Auszeichnungssprache, die als Erweiterung von HTML und als Spezialisierung von SGML angesehen werde kann. XML kommt heute in vielen Bereichen zum Einsatz und so müssen Sie auch bei der Java-Programmierung damit umgehen können.

Die Darstellung hier ist noch sehr knapp. Für eine etwas genauere Behandlung verweise ich auf das im Praktikum verwendete Buch von Panitz [Pan08].

22.2 Lernziele

- Grundzüge von XML kennen.
- Einige Konzepte zum Umgang mit XML in Java kennen.

22.3 Einführung

Ein XML-Dokument beginnt mit der Angabe der Version:

```
<?xml version="1.0" ?>
```

Als Parameter kann die Codierung angegeben werden, hier z. B. ISO-8859-1, die ISO-Codierung für westeuropäische Sprachen:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
```

Des weiteren enthält ein XML-Dokument Tags, die immer paarweise auftreten müssen: Der Start wird durch `<einTag>` gekennzeichnet, das Ende durch `</einTag>`. Zwischen Start- und Ende-Tag steht der eigentlich Inhalt des XML-Dokuments.

Des weiteren muss ein XML-Dokument einigen einfachen Regeln genügen:

1. Hierarchischer Aufbau.
2. Es gibt genau ein oberstes Element: `<topTag>` ... `</topTag>` zwischen dem alles Andere steht.
3. Start-Tag, Ende-Tag Paare überlappen nicht: Das heißt, bei Schachtelung muss das letzte Start-Tag als erstes geschlossen werden usw.

Da die Tags über die im Standard vorgegebenen frei definiert werden können, muss deren Bedeutung erläutert werden. Dies kann in einer Document Type Definition (DTD) und einem Schema (xsd) geschehen. Eine Schema-Datei ist eine XML-Datei mit dafür festgelegten Tags. Weitere Informationen finden Sie beim W3C http://www.w3c.org/, in [Ull11], http://de.selfhtml.org/xml/uvam.

22.4 Java Beans

Beans werden in Java an einigen Stellen verwendet. Ich erinnere hier nur an die Swing-Klassen, bei denen zum Serialisieren auf das Paket *java.beans* verwiesen wurde. Java Beans sind Klassen, die folgenden Konventionen genügen:

Definition 22.4.1 (Java Bean)
Eine *Java Bean* ist eine Java-Klasse mit folgenden Eigenschaften:

1. Einfache Eigenschaften (properties): Für eine Eigenschaft vom Typ T mit Namen „hugo“ gibt es Methoden „T getHugo()“ und „setHugo(T t)“.
2. Indizierte Eigenschaften (indexed properties): Gibt es eine Eigenschaft „fu“ vom Typ T mit mehreren Ausprägungen, so gibt es Methoden „T [] getFu()“ und „setFu(T [] t)“.
3. Gebundene Eigenschaften (bound properties): Dies sind Eigenschaften, über die andere Objekte informiert werden können. Das ist das Ihnen schon bekannte Beobachter-Muster (Observer Pattern), auch als Model-View-Controller bekannt (MVC).
4. Eigenschaften mit Vorbehalt (vetoable oder constraint properties)
5. Öffentlicher Default-Konstruktor.
6. Auf Attribute wird über Operationen der Art get..., set... bzw. is... anstelle von get... bei booleschen Attributen zugegriffen. Bei Beans wird anstelle von Attributen oder Fields von Eigenschaften (Properties) gesprochen.
7. Eine Bean kann ihren Zustand durch Serialisierung speichern und wieder herstellen.
8. Zu einer Eigenschaft kann es Listener geben.

◀

Ein triviales Beispiel zeigt die Klasse *Person1* im Paket *beans*. Die getter und setter wurden von Eclipse entsprechend der Namenskonventionen für Beans erzeugt.

Die Umsetzung des Beobachter-Musters in Beans illustriert die Klasse *Person2* im Paket *beans*. Das ist das gleiche Prinzip, dass Sie bereits aus den Listener in Swing kennen sollten. Hier die wichtigsten Teile der Klasse Person2:

```
public class Person2 {
        ...
private PropertyChangeSupport propertyChangeSupport = new PropertyChangeSupport(this);
/**
 * Diese Methode illustriert die Implementierung des Beobachter Musters in Java Beans.
 * @param name Neuer name der Person
 */
public void setName(String name) {
String nameAlt = this.name;
this.name = name;
this.propertyChangeSupport.firePropertyChange("name", nameAlt,this.name);
}
```

```
        ...
public void removePropertyChangeListener(PropertyChangeListener listener) {
 propertyChangeSupport.removePropertyChangeListener(listener);
}
public void addPropertyChangeListener(PropertyChangeListener listener) {
this.propertyChangeSupport.addPropertyChangeListener(listener);
}

}
```

Die Klasse *Person3* illustriert eine weitere Möglichkeit von Java *Beans*: *vetoable changes*.

Nun zur Serialisierung von Beans und Swing-Objekten. Dazu gibt es im Paket *java.beans* die Klasse XMLEncoder zum Schreiben der Daten eines Objekts in eine XML-Datei und die Klasse XMLDecoder zum Einlesen. Letztendlich passiert dabei nichts anderes als beim Verwenden eines ObjectOutput- bzw InputStreams. Allerdings wird Ihnen dabei einiges abgenommen. Am Beispiel *WriteXML* sehen Sie ein einfaches Beispiel. Dies zeigt auch die Vorteile des Interfaces *AutoCloseable*.

22.5 DOM

DOM steht für Document Object Model. Hier wird das XML-Dokument in seiner Sicht als Baum komplett im Speicher gehalten. Die notwendigen Interfaces finden Sie im Paket *org.w3c.dom*, implementierende Klassen u. a. im Paket *java.xml.parsers*.

Ein Beispiel für die Verwendung zeigt die Klasse *DOMExample01*.

22.6 SAX

SAX steht für Simple API for XML Parsing. Hier wird das XML-Dokument im Wesentlichen als sequentielle Datei betrachtet. Von dieser können auch nur Teile im Speicher gehalten werden.

Ein Beispiel finden Sie in [Pan08] im Umfeld der Klasse *SAXTournamentReader*.

22.7 StAX

Streaming API for XML. Hier wird die XML Datei als Stream behandelt. Es wird also im Stil einer *foreach*-Schleife über die Datei iteriert. Ein Beispiel finden Sie in [Pan08] im Umfeld der Klasse *StAXTournamentReader*.

22.8 JDOM

Java Document Object Model. Entwicklung schon lange eingestellt, aber wohl immer noch populär.

22.9 JAXB

Java API for XML Parsing. Für Abweichungen von den Standard-Abbildung von Java-Datentypen auf XML oder die JAXB nicht unterstützt gibt es im Paket *javax.xml.bind.annotation.adapters* die abstrakte Klasse *XmlAdapter<ValueType,BoundType>* mit den Methoden *ValueType marshal(BoundType v)* und *BoundType unmarshal(ValueType v)*.

22.10 Anwendungen

22.11 Historische Anmerkungen

22.12 Aufgaben

Kapitel 23

Entwurfsmuster

23.1 Übersicht

Entwurfsmuster sammeln Erfahrungen. Als Anfänger haben Sie noch keine Erfahrungen im Programmieren. Viele Entwurfsmuster erfordern Erfahrung, um sie zu verstehen. Aber selbst Anfänger im Programmieren müssen das Beobachter-Muster, das Fabrik-Muster (Abstrakte Fabrik, Fabrikmethode) und vielleicht noch einige weitere kennen. Ich konzentriere mich in diesem Kapitel auf diejenigen Muster, die in Java oft oder an zentraler Stelle verwendet werden.

23.2 Lernziele

- Wichtige Muster der Java Programmierung kennen und anwenden können.
- Das Singleton-Muster (singleton pattern) kennen und in Java professionell implementieren können.
- Fabrik-Muster in Java kennen und anwenden können.
- Beobachter bzw. Model-View-Controller in Java kennen und anwenden können.
- Das Iterator-Muster in verschiedenen Varianten beherrschen.

23.3 Singleton

Singleton ist das allereinfachste Muster. Es sorgt dafür, dass nur ein Objekt einer Klasse erzeugt werden kann, dass dann von allen Nutzern gemeinsam verwendet wird. Dazu müssen Sie nur wenig tun:

- Alle Konstruktoren werden als *private* deklariert. Soll die Klasse spezialisiert werden können, so können Sie Konstruktoren auch *protected* deklarieren.
- Ein Klassenattribut speichert eine Referenz auf das (einzige) Objekt der Klasse. Soll es eine begrenzte Anzahl größer eins geben, so nehmen Sie einen geeigneten Container.
- Eine öffentliche Klassenmethode (*Fabrikmethode*) liefert das Objekt für Nutzer.

Ein ganz einfache Modell zeigt Abb. 23.1 Der aktuelle Stand für die Implementierung eines *Singletons* in Java ist oft:

```
public enum Singleton {
    INSTANCE;
    private boolean state;
```

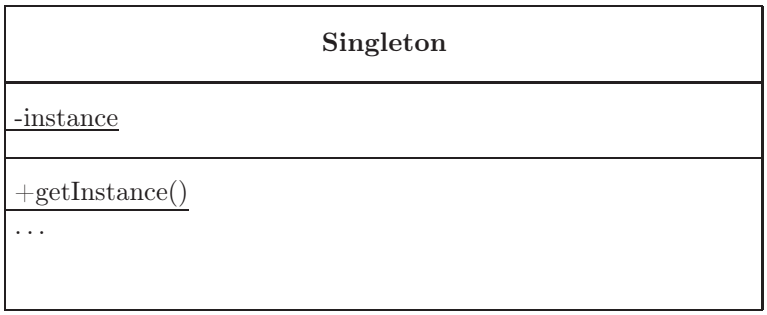


Abb. 23.1: Singleton pattern

```
    public boolean isState() {
        return this.state;
    }

    public void setState(boolean state) {
        this.state = state;
    }
```

Die unterschiedlichen Namen sind java-Konventionen geschuldet: Enum Werte werden in *SCREAMING_ SNAKE_ CASE* gesetzt. Eine einfache Anwendung hierfür ist ein *reverse comparator*

```
public enum ComparatorExample02 implements Comparator<String> {
    INSTANCE;
    @Override
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
}
```

Hier brauchen Sie gar keine *Fabrikmethode*. Da *Singleton.INSTANCE final* ist, kann einfach direkt darauf zugegriffen werden.

Bemerkung 23.3.1 (Warnung)
Die Implementierung von obiger Methode *compare* ist nur akzeptabel, da *String* nur nicht-negative Längen haben. Bei der Addition von *int*'s kann es zu einem Überlauf kommen. ◄

Aber so etwas werden Sie nicht schreiben müssen: *Comparator* ist eine generische Klasse und Sie können ganz einfach eine generische Klassenmethode schreiben, die das leistet. Hier ein Beispiel aus einer Utility-Klasse *Collections2*, in der ich das eben präsentierte Schema aus der Utility-Klasse *Collections* umgesetzt habe:

```
    @SuppressWarnings("unchecked")
    public static <T> Comparator<? super T> reverseOrder() {
        return (Comparator<T>) ReverseComparator.REVERSE_ORDER;
    }

    private static enum ReverseComparator implements Comparator<Comparable<Object>>,
```

```
            Serializable {
        REVERSE_ORDER;
        @Override
        public int compare(Comparable<Object> c1, Comparable<Object> c2) {
            return c2.compareTo(c1);
        }
    }
```

Die weiter unten beschriebene Methode *readResolve()* brauchen Sie Sie bei *enums* nicht, da *enums* nicht de-serialisierbar sind. Die Methoden *readObject* und *readObjectNoData* sind wie folgt überschrieben:

```
private void readObject(ObjectInputStream in) throws IOException,
    ClassNotFoundException {
    throw new InvalidObjectException("can't deserialize enum");
}

private void readObjectNoData() throws ObjectStreamException {
    throw new InvalidObjectException("can't deserialize enum");
}
```

So erhalten Sie eine Exception, wenn Sie versuchen ein Enum zu deserialisieren. Dadurch ist die Singleton-Eigenschaft auch für serialisierbare Singletons gewährleistet, wenn Sie diese Technik verwenden. Die Klasse *EnumSet* hat einige solche Klassenmethoden, um ein *EnumSet* zu erzeugen.

Wollen Sie das Objekt (bzw. die Objekte) einer Singleton-Klasse serialisierbar machen, so müssen Sie die API-Dokumentation genau lesen oder das Ende von Abschn 14.8.

Wenn eine serialisierbare Klasse eine Methode *readResolve* hat, so wird diese aufgerufen, nachdem das Objekt deserialisiert wurde. Die könnte hier etwa so aussehen (Siehe *pattern.Singleton*):

```
    private Object readResolve() {
        return reverseOrder();
    }
```

So wird sichergestellt, dass tatsächlich das wegserialisierte Singleton-Objekt genommen wird und nicht ein neues erstellt wird. Dieser Mechanismus ist immer notwendig, wenn Klassen vom ObjectInputStream gelesene Objekte durch andere ersetzen müssen. Instanz-Attribute machen bei einem Singleton auf den ersten Blick keinen wirklichen Sinn. Im Zusammenspiel mit anderen Java-Mechanismen können Sie aber Sinn machen. Sie sollten dann aber auf jeden Fall als transient deklariert sein. Sie könnten als separate Klassenattribute deklariert werden. Aber das macht keinen Sinn, auch nicht unter Kapselungsgesichtspunkten.

Das Gegenstück zur Methode *readResolve* beim Serialisieren ist die Methode *writeReplace*. Eine serialisierbare Klasse muss diese Methode haben, wenn ein anderes Objekt, als das im Parameter übergebene auf den Stream geschrieben werden soll. Für beide Methoden gilt: Sie können *private*, *protected* oder sichtbar sein.

23.4 Beobachter

Das Beobachtermuster ist Ihnen vielleicht zuerst in Form der verschiedenen Listener in JavaFX begegnet. Auch in Java Beans wird dieses Prinzip systematisch eingesetzt. Die Abb. 6.4 auf S. 103. zeigt dies schematisch. Der Operation *notify* entspricht bei Java Beans die Methode *firePropertyChange* der Klasse *propertyChangeSupport*.

Ferner gibt es in *java.util* das Interface *Observer* und die Klasse *Observable*. Beide sind ab Java 9 Deprecated.

23.5 Fabrik

Jedes Muster soll ein Problem lösen. Mit Generics wurden in Java Probleme der Typsicherheit gelöst. Ein Problem wurde dadurch aber nicht gelöst: Attribute mit dem Typ des Typ Parameters können nicht mittels des *new*-Operators erzeugt werden. Um dieses Problem zu lösen, gibt es die in 18.8 beschriebenen Möglichkeiten. Hier betrachte ich den Einsatz des Factory Pattern (Fabrikmuster) genauer. Das Problem, das hier gelöst werden soll ist nicht (nur) eine technische Unzulänglichkeit in Java. Wegen der Typauslöschung (type erasure) ist zur Laufzeit nicht der Typparameter sondern nur *Object* bzw. die engste Bound bekannt. Etwas allgemeiner formuliert: Es sollen Objekte erzeugt werden, deren Typ zur Compile-Zeit noch nicht bekannt ist. Dies ist genau die Situation, in der eine Fabrik nützlich ist [GHJV95]. Genauer handelt es sich um eine Fabrikmethode, wie in Abb. 23.2 skizziert. Bei der Verwendung im Zusammenhang mit generi-

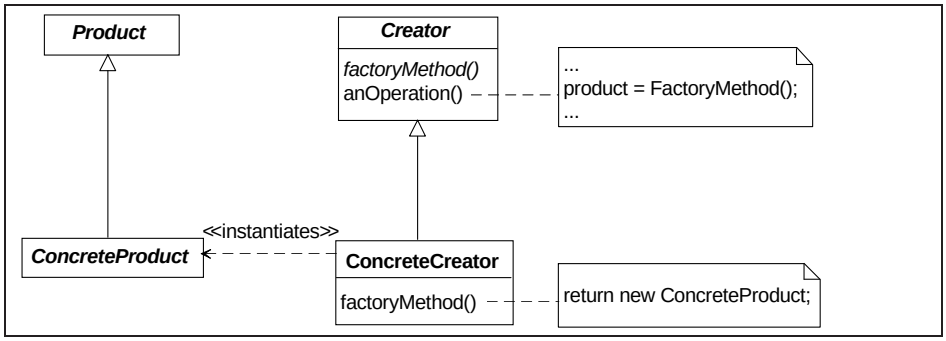


Abb. 23.2: Factory Method pattern: Struktur

schen Attributen in Java in Abschn. 18.8 ist die *factoryMethod* die Methode *create* im Interface *Factory*, die Nutzer z. B. durch eine anonyme Klasse implementieren können.

23.6 Visitor

Wichtig für Streams (Consumer hat accept Methode)

23.7 Composite

Das Composite Pattern kennen Sie zumindest aus einem der Minimodelle (siehe z. B. Aufgabe 12.5 in Abschn. 1.8). Das allgemeine Schema zeigt Abb. 23.3.

Ein Composite Objekt hat also weitere Bestandteile. Hier kommt ein weiteres Muster zum Einsatz, dass Sie bereits kennen. Die Elemente eines Composites werden mittels eines Iterators durchlaufen.

23.8 Iterator

Wichtig für: for each (externer Iterator) und Streams (interner Iterator) Iteratoren externalisieren das durchlaufen von Aggregaten. Typische Beispiele hierfür liefern die Container-Klassen in *java.util*. Das allgemeine Schema zeigt Abb. 23.4. Ein Beispiel für *Aggregate* ist das Interface *List<T>*, für *ConcreteAggregate* die Klassen *ArrayList<T>* und *LinkedList<T>*. *Iterator* entspricht z. B. dem Java-Interface *Iterator*. Iteratoren haben viele Vorteile, von denen ich nur einige nenne:

- Sie können für ein Aggregat mehrere Iteratoren haben, die sich jeweils eine Position merken.

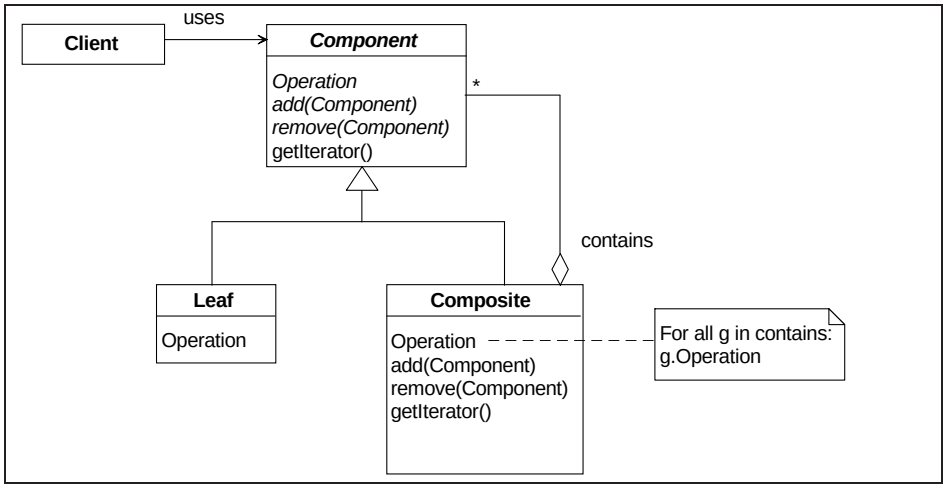


Abb. 23.3: Composite pattern

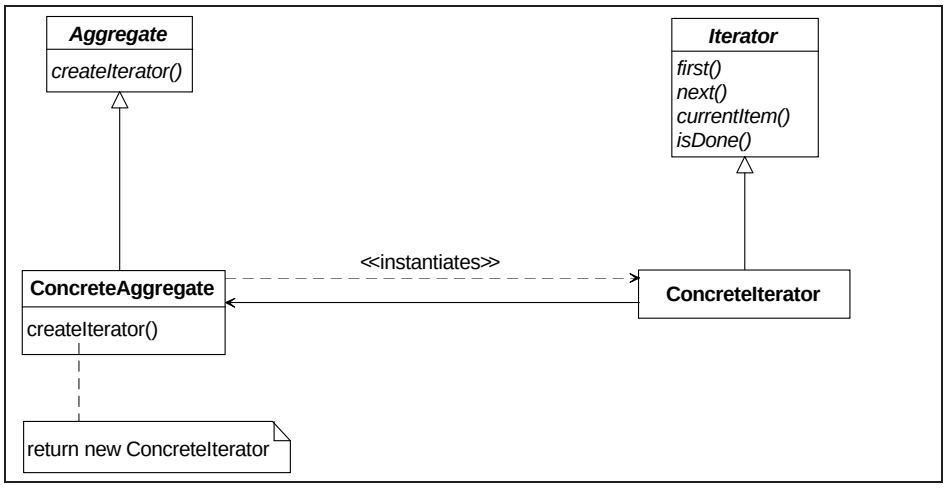


Abb. 23.4: Iterator pattern: Struktur

- Die Schnittstelle des Aggregats wird schmaler.
- Wie das Beispiel der *LinkedList* zeigt, kann eine *foreach*-Schleife mit einem Iterator effizienter sein, als die entsprechende *for*-Schleife mit ganzzahliger Laufvariablen.

Wenn Sie sich das *Iterator pattern* genauer ansehen, so werden Sie feststellen, dass es nicht nur externe Iteratoren gibt, sondern auch interne Iteratoren geben kann. Letztere werden in Java mit dem neuen Streaming API (Java 8) generell verfügbar. Ein Beispiel hierfür bietet die default Methode *forEach(Consumer<? super T> action)* des *Interface>IterableIterable*Interface *Iterable*.

23.9 Flyweight Pattern

Das Flyweight Pattern wird eingesetzt, wenn eine große Anzahl von Objekten durch wenige gemeinsam genutzte ersetzt werden können [GHJV95]. Abbildung 23.5 zeigt das allgemeine Schema.

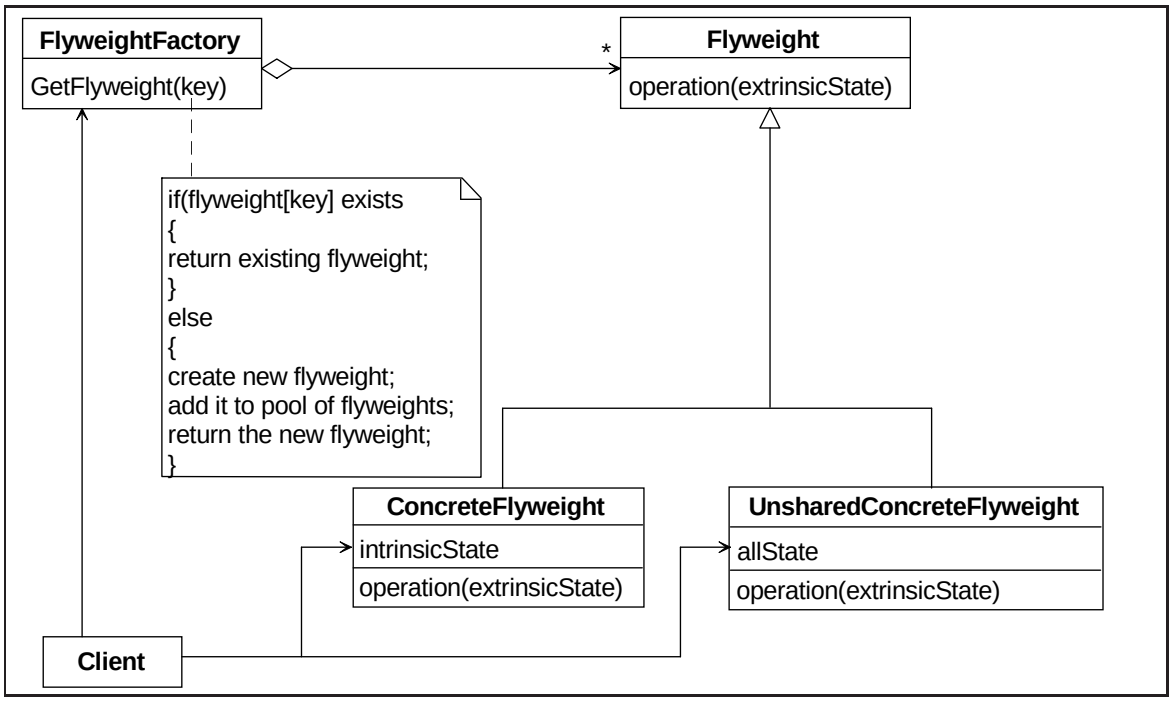


Abb. 23.5: Flyweight

In Java wird dieses Muster für *String-Literale*, genauer konstante Ausdrücke von *String*s verwendet. Java verwendet (seit 7.40+) hierfür Platz auf dem Heap. Die Zugriffszeit ist konstant, d. h. $\mathcal{O}(1)$, also unabhängig von der Anzahl von *Flyweight-Strings*. Im Einzelnen funktioniert das so:

- Ein *String-Literal* ist eine Referenz auf ein *String*-Objekt und verweist immer auf das gleiche *String*-Objekt.
- Ein *String-Literal* wird einem von der Klasse *String* verwalteten Pool hinzugefügt. Allgemeiner gilt dies für konstante *String*-Ausdrücke. Die Klasse *String* ist dabei die *FlyweightFactory* aus Abb. 23.5. Die Methode *intern()* ist die Methode *GetFlyweight*. Der Key ist der String des Literals für das sie aufgerufen wird. Es handelt sich um eine *native* Methode, deren API-Dokumentation genau das beschreibt, was die Notiz in Abb. 23.5 angibt.
- Da der Vergleich mit *equals* teurer ist, als der mittels „==“ können Sie bei entsprechender Sorgfält hier etwas einsparen. Aber Achtung: Wie immer müssen Sie wissen, was sie tun!

23.10 Null Object Pattern

In vielen Anwendungen muss oft überprüft werden, ob ein Objekt *null* ist. Das kann Code unübersichtlich und fehleranfällig machen. Martin Fowler nennt in [Fow99] das Beispiel einer Klasse *Customer* zu der es eine Unterklasse *NullCustomer* gibt. Ein *Null Object*[Woo98] ist ein Stellvertreter für ein anderes Objekt mit der gleichen Schnittstelle, das nichts tut. Es schirmt die Implementierung also von Entscheidungen ab, wie ein Nichtstun realisiert wird und versteckt derartige Details vor Nutzern. Ein *Null Object* liefert Ergebnisse, mit denen eine Anwendung weiterarbeiten kann, falls das eigentlich benötigte Elemente nicht existiert.

Abbildung 23.6 zeigt das allgemeine Schema. In Java 8 wurde dazu die Klasse *Optional* ein-

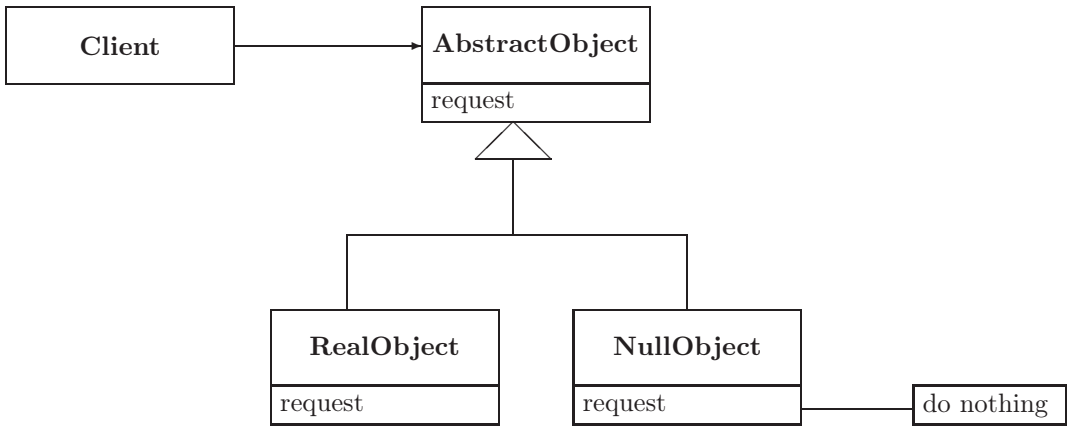


Abb. 23.6: Null Object Pattern

geführt. Diese Klasse stellt Methoden zur Verfügung, mit denen überprüft werden kann, ob ein Objekt vorhanden ist oder nicht und was dann geschehen soll. *Optional* hat drei Fabrikmethoden: *empty()* liefert ein *Optional* ohne default Wert, *of(T value* mit einem vorgegebenen nicht-*null* Wert, *ofNullable(T value)* ein *Optional* das den Wert liefert, wenn er nicht-*null* ist und andernfalls ein leeres *Optional*, wie es auch *empty* liefert.

boolean isPresent()

void ifPresent(Consumer<? super T> consumer)

T orElse(T other)

T orElseGet(Supplier<? extends T> other)

<X extends Throwable> orElseThrow(Supplier<? extends X> exceptionSupplier)

23.11 Decorator pattern

23.12 Historische Anmerkungen

Muster wurden zuerst von der „Viererbande“ in [Gam92] systematisch beschrieben und popularisiert. Die jetzt empfohlene Implementierung eines *Singletons* mittels *enum* wurd mit der EInführung von generischen Enums in Java 1.5 möglich.

23.13 Aufgaben

1. Halten Sie es für sinnvoll, in einer *Singleton*-Klasse die Methode *equals* zu überschreiben? Begründen Sie bitte Ihre Antwort!

Kapitel 24

Nebenläufige und asynchrone Programmierung

24.1 Übersicht

Java unterstützt die Programmierung parallel ablaufender Verarbeitung durch Threads.

24.2 Lernziele

- Threads in Java schreiben können.
- Die Funktion von synchronized kennen und verwenden können.
- Threadpools und Executors kennen und verwenden können.
- Kommunikationsmöglichkeiten zwischen Threads kennen und nutzen können.

24.3 Einführung

Ein Rechner kann einen oder viele Prozessoren enthalten. Enthält er nur einen Prozessor, so kann er nur eine Sache zur Zeit tun. Trotzdem können Nutzer den Eindruck haben, dass mehrere Dinge parallel erledigt werden. Um dies zu erreichen gibt es verschiedene Techniken. Einige davon werden in Vorlesungen über Betriebssysteme vermittelt. Ich beschränke mich in diesem Kapitel ausschließlich auf die Mechanismen, die Ihnen in Java hierfür zur Verfügung stellt. Die JVM stellt für viele Anforderungen adäquate Lösungen zur Verfügung. Verwenden Sie nur die Klasse *Thread* oder das Interface *Runnable*, so werden Ihre Threads in einer *Thread-Queue* vom Scheduler der JVM verwaltet (links in Abb. 24.1)

Ohne auf technische Details einzugehen, kann dieser Mechanismus einfach beschrieben werden: Die JVM hat eine *Queue*, in die Threads eingestellt werden. Diese Queue wird bis zu einer bestimmten Tiefe abgearbeitet. Ist diese z. B. 10, so können 10 Threads parallel laufen.

Verwenden Sie *Threadpools* und *ExecutorServices*, so können Sie für verschiedene Gruppen von Threads verschiedene Threadpools mit situationsangepassten Eigenschaften verwenden (rechts in Abb. 24.1).

24.4 Implementierung

Im einfachsten Fall erzeugen Sie sich eine Klasse, die die Klasse *Thread* spezialisiert, überschreiben die Methode *run()* und rufen die Methode *start()* auf. Die *start()*-Methode übergibt das Objekt der JVM und diese ruft dessen *run()*-Methode auf.

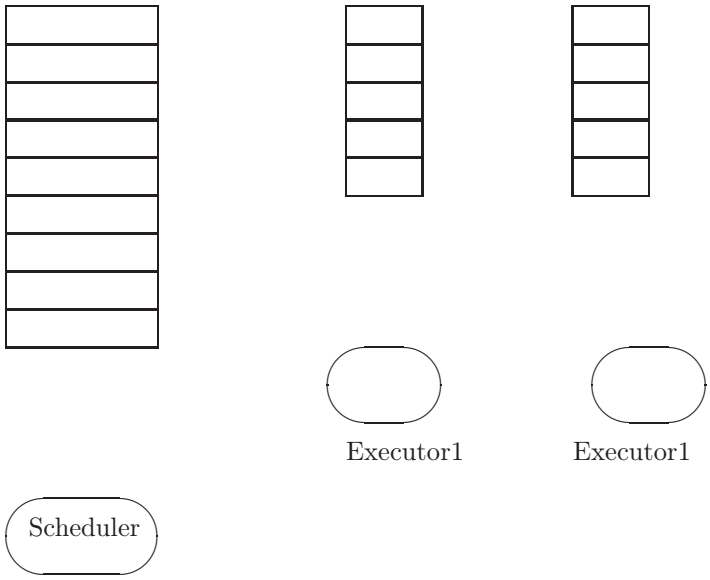


Abb. 24.1: Thread Queue und Threadpools

Es gibt zwei Möglichkeiten in Java mit den Nebenläufigkeitsmechanismen parallele Vorgänge zu implementieren:

1. Sie können die Klasse *Thread* spezialisieren. In diesem Fall werden Sie die Methode *run* überschreiben müssen, es sei denn Sie haben ein separates *Runnable* Objekt erzeugt. Dann ruft die Methode *run* von *Thread* die *run*-Methode dieses Objekts auf. Die *start*-Methode erben Sie einfach von *Thread*.

2. Sie können eine Klasse schreiben, die das Interface *Runnable* implementiert. Sie müssen dann die Methode *run* implementieren. Innerhalb ihrer Klasse, die *Runnable* implementiert brauchen Sie ein Objekt, z. B. *thread* der Klasse *Thread*. In Ihrer Klasse implementieren Sie eine Methode *start*, die einfach *thread.start()* aufruft.

Im Paket *java.util.concurrent* finden Sie weitere Interfaces, wie z. B. ExecutorServices. Mit diesen können Sie die Thread-Verwaltung an Ihre Anforderungen anpassen

Alle diese Möglichkeiten stelle ich in den folgenden Abschnitten im Detail dar.

24.4.1 Thread

Steht dies nicht in Konflikt mit anderen Vererbungsmöglichkeiten, die Sie nutzen wollen, so können Sie eine Klasse, deren Objekte einen eigenen Thread haben müssen, von der Klasse Thread spezialisieren. In der Klasse *ThreadDerived* wird die Methode *run* überschrieben. Für ein Objekt der Klasse *ThreadDerived* wird mittels Aufruf der von *Thread* geerbten Methode *start* das Objekt an die JVM übergeben, wie in Abb. 24.3 skizziert. Code finden Sie in der Klasse *ThreadDerived*.

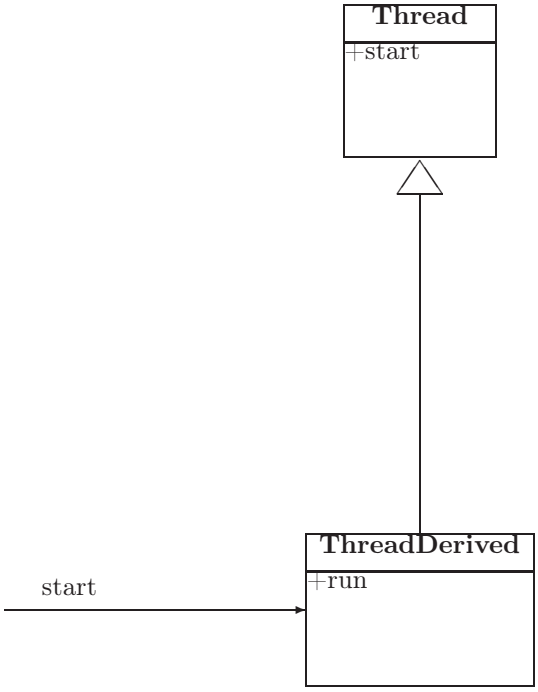


Abb. 24.2: Spezialisierung von Thread

24.4.2 Runnable

Flexibler und genauso einfach ist die Implementierung des Interfaces *Runnable*. Analog zum Vorgehen in Absch. 24.4.1 muss nun die Methode *run* implementiert werden. Zum Starten wird mittels eines Konstruktors der Klasse Thread, der als Parameter ein *Runnable* erhält, ein neuer Thread erstellt und mit der *start*-Methode an die JVM übergeben. Dies ist in Abb. 24.3 skizziert. Code finden Sie in der Klasse *ThreadRunnable*.

24.4.3 Threadpools

Die aktuelle und jetzt empfohlene Technik zur Implementierung von Threads verwendet neben dem Interface Runnable die Interfaces und Klassen aus dem Paket *java.util.concurrent*. In Abb. 24.4 wird mittels der Klassenmethode *newCachedThreadPool* der Klasse *Executors* ein neuer Threadpool beschafft. Nun wird mittels dessen Methode *submit* wie in Abschn. 24.4.2 ein neuer Thread an die JVM übergeben. Dies sieht zunächst nur etwas komplizierter, aber nicht wesentlich anders aus, als die Technik aus Abschn. 24.4.2. In Abschn. 24.6 werde ich zeigen, welche zusätzlichen Möglichkeiten durch diese Technik erschlossen werden.

24.5 Thread-Synchronisation

Ein typisches Problem im Umgang mit Threads illustriert das Leser/Schreiber-Problem: Ein Schreiber-Thread erstellt Daten und ein Leser-Thread liest sie aus. Die Klassen *SchlechteFigur*, *Leser* und Schreiber illustrieren dies. Der Schreiber „stellt“ Figuren auf die Diagonale eines Schachbretts und der Leser list sie aus. Dabei werden immer wieder ungültige Positionen gelesen, die nicht auf der Diagoanle stehen. Dieser Efekt kommt dadurch zu Stande, dass der Leser zum Zug kommen kann, obwohl der Schreiber erst die Spalte und noch nicht die Zeile gesetzt hat.

Abb. 24.3: Implementierung von Runnable

Abb. 24.4: Implementierung mit java.util.concurrent

Zur Lösung dieses Problems werden die Methoden setPosition und getPosition synchronisiert, d.h. durch das Schlüsselwort *synchronized* gekennzeichnet (siehe *GuteFigur*. Dies bedeutet: Für ein Objekt der Klasse *GuteFigur* kann nur eine der beiden Methoden zur Zeit aufgerufen werden. Java stellt dazu einen Monitor zur Verfügung (Abb. 24.5). Nur ein Thread zur Zeit kann eine *synchroni-*

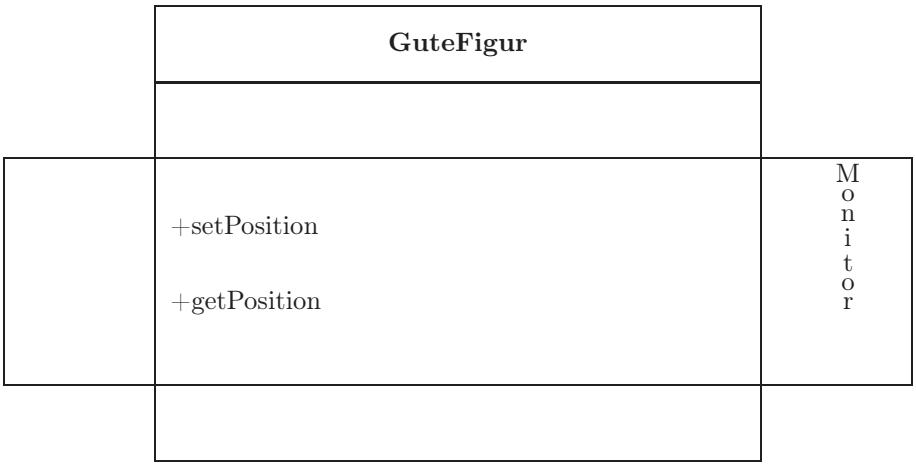


Abb. 24.5: Monitor

zed Methode aufrufen. Andere warten im Monitor. So ist nun sichergestellt, dass der Schreiber die *setPosition*-Methode vollständig ausführen kann und ebenso der Leser die *getPosition*-Methode.

24.6 Kommunikation zwischen Threads

Mittels *synchronized* lassen sich aber nicht alle Probleme dieser Art aus der Welt schaffen, wie das Erzeuger/Verbraucher-Problem zeigt: Hier haben wir es im Beispiel mit der Klasse *Wert* mit einem einfachen Puffer zu tun, in den ein *Erzeuger* Werte einstellt (*put*) und ein *Verbraucher* Werte entnimmt (*get*). Die am Anfang bzw. am Ende der Erzeuger bzw. Verbraucher-Methode *run* eingebauten *sleep*-Aufrufe stehen als Simulation aufwändiger Aktivitäten. Auch mit *synchronized* kann der Verbraucher hier Werte mehrfach auslesen oder Werte verpassen. Hierzu müssen die beiden Threads — Erzeuger und Verbraucher — miteinander kommunizieren.

In diesem Fall kann dies die Klasse *Wert* übernehmen: Die bekommt ein boolesches Attribut *verfügbar*, das in den Methoden *put* und *get* verwendet korrekt und gesetzt werden muss. Das passiert in der Klasse *GuterWert* so:

```
public synchronized int get() {
   if (!verfuegbar)
      try {
        wait();
      }
```

```
      catch (InterruptedException ie) {
       ie.printStackTrace();
      }
    verfuegbar = false;
    notify();
    System.out.println("Verbraucher get: " + wert);
    return wert;
  }
public synchronized void put (int w) {
    if (verfuegbar)
      try {
        wait();
      }
      catch (InterruptedException ie) {
       ie.printStackTrace();
      }
    wert = w;
    System.out.println("Erzeuger    put: " + wert);
    verfuegbar = true;
    notify();
  }
```

Das Attribut *verfuegbar* gibt an, ob ein Wert erzeugt und noch nicht verbraucht wurde. Will ein Erzeuger einen neuen Wert einstellen, so wird in der *put*-Methode überprüft, ob ein Wert (noch) verfügbar ist. Ist dies der Fall, so wartet der Wert. Anderfalls setzt er den (neuen) Wert und *verfuegbar* auf *true*. Anschließend informiert er mittels *notify* einen im Monitor wartenden Thread. Versucht ein Verbraucher einen Wert zu entnehmen, so wird in der *get*-Methode von Wert überprüft, ob ein Wert verfügbar ist. Ist dies nicht der Fall, so wartet die *get*-Methode. Andernfalls wird ein (eventuell) wartender Thread im Monitor informiert und der Wert zurückgegeben.

24.7 Fork und Join ab Java 7

Um einen Algorithmus auf mehrere Prozessoren eines Rechners zu verteilen, dient die Klasse ForkJoinTask. Konkrete Unterklassen sind *RecursiveAction* und *RecursiveTask*. *RecursiveAction* liefert kein Ergebnis zurück, während *RecursiveTask* ein Ergebnis zurückliefert.

Das Grundprinzip ist „teile und herrsche“: Ist das Problem klein genug, so löse man es in einem Thread. Anderfalls verkleinere man es und löse zwei Teilprobleme durch parallele Threads und füge das Ergebnis anschließend zusammen. Beispiele hierfür liefert z. B. [LeaoJ].

Ich gebe hier ein Beispiel aus dem Bereich Algorithmen und Datenstrukturen: *Quicksort* ist bis auf den unangenehmen *worst case* ein gutes Sortierverfahren. Für ein Array eines Typs *T*, der *Comparable* implementiert kann es grob so beschrieben werden:

```
quicksort(T [] a,int start, int ende){
    findSplitPosition();
    quicksort(a, start, l);
    quicksort(a, r, ende);
}
```

Der Algorithmus ist um so besser, je näher die Position, an der das Array aufgesplittet wird, an der Mitte liegt. Genauer bricht man das aufsplitten am Besten ab, wenn eine Schranke erreicht ist, z. B. 30. Anschließend sortiert man einfach das Array mit einem Verfahren, wie Insertion- oder Selectionsort.

Die wichtigen Klassen finden Sie im Paket *java.util.concurrent*.

Hier typischer Code

```
public class ForkJoinQuicksortTask extends RecursiveAction {
    private static final int SERIAL_THRESHOLD = 0x1000;

    private final int[] a;
    private final int left;
    private final int right;

    public ForkJoinQuicksortTask(int[] a) {
        this(a, 0, a.length - 1);
    }

    private ForkJoinQuicksortTask(int[] a, int left, int right) {
        this.a = a;
        this.left = left;
        this.right = right;
    }

    @Override
    protected void compute() {
        if (serialThresholdMet()) {
            Arrays.sort(a, left, right + 1);
        } else {
            int pivotIndex = Quicksort.partition(a, left, right);
            ForkJoinTask<?> t1 = null;

            if (left < pivotIndex)
                t1 = new ForkJoinQuicksortTask(a, left, pivotIndex).fork();
            if (pivotIndex + 1 < right)
                new ForkJoinQuicksortTask(a, pivotIndex + 1, right).invoke();

            if (t1 != null)
                t1.join();
        }
    }

    private boolean serialThresholdMet() {
        return right - left < SERIAL_THRESHOLD;
    }
}
```

Sie sollten mit der Anzahl der *ForkJoin*-Task experimentieren. Ich sehe vor allem zwei Ansätze:

1. Keine Einschränkung der Taskanzahl.
2. Nur so viele Tasks, wie der Rechner Prozessoren hat. Diese Anzahl bekommen Sie mittels *Runtime.getRuntime().availableProcessors()*.

24.8 Historische Anmerkungen

Das Paket java.util.concurrent kam mit Java 1.5. In Java 7 kamen *ForkJoinPool*s hinzu.

24.9 Aufgaben

1. Nennen Sie die Möglichkeiten in Java Threads zu erzeugen und deren Vor- und Nachteile!

2. Erläutern Sie die Funktion und die Nutzung des Schlüsselworts *synchronized*!

3. Warum wird empfohlen Threads über die Implementierung von *Runnable* zu realisieren (und nicht über die Spezialisierung von *Thread*)?

4. Schreiben Sie bitte eine Klasse, deren Objekte Würfe mit einen Würfel simulieren. Mit zwei Objekten dieser Klasse soll gegeneinander gewürfelt werden. Der Spieler hat gewonnen, der die höhere Augenzahl geworfen hat. Die Klasse soll *Runnable* implementieren. Verwenden Sie bitte Hilfmittel aus java.util.concurrent! Jeder der beiden Würfe soll in einem eigenen Thread erfolgen und anschließend das Ergebnis ermittelt werden. Die Spieler können Sie etwa Spieler1 und Spieler2 nennen. Exception Behandlung ist nicht gefordert. Sie können das Starten in einer main-Methode Ihrer Klasse schreiben.

5. Die folgende Aufgabe stammt aus einer Betriebssysteme-Klausur von Wolfgang Fohl (SS 2010) und wurde auf Java umgebaut. Jedes Mal, wenn im Controller-Thread die Return-Taste gedrückt wird, soll der WorkerThread zwischen den Zuständen blockiert und freigegeben wechseln. Das Blockieren soll mit Hilfe eines klassischen binären Semaphoren realisiert werden. Sie dürfen ausschließlich mit den beiden Methoden sem_wait und sem_post auf den Semaphoren zugreifen. Fügen Sie Ihren Code zur Koordinierung der Threads in das Programmlisting ein.

```
/* Signaturen Semaphor-Methoden */
void sem_wait(sema* s);
void sem_post(sema* s);
/* Semaphoredeklarationen
* in Pseudocode. Z.B.:
* sema mysema = 3;
*/
/* Globale Variablen */
int main(void) {
    start_controller();
    start_worker();
    return 0;
} /* end main */
/* Der Controller-Thread */
void controller(void) {
   while (1) {
      wait_for_return_key();
      printf("Worker-Thread "
      "wird umgeschaltet\n");
      /* IHR KOORDINATIONSCODE: */
      } /* end while */
   } /* end controller */
/* Der Worker-Thread */
void worker(void) {
   while (1){
   do_work();
     /* IHR KOORDINATIONSCODE: */
   } /* end while */
} /* end worker */Name:
```

Kapitel 25

Netzwerkprogrammierung

25.1 Übersicht

[1] Dieses Kapitel beschreibt gängige Implementierungsmöglichkeiten zur Kommunikation zwischen Systemen. Es wird beschrieben, wie in Java auf entfernte Objekte und Ressourcen zugegriffen werden kann. Zunächst werden Client-Server-Kommunikation, Message Exchange Pattern und Vergleichskriterien festgelegt und beschrieben; im weiteren Verlauf die Funktionsweisen und Implementierungen verschiedener Java-Standards. Vor- und Nachteile jener werden zusammengefasst und verglichen. Die Client-Server-Architektur bezeichnet die kooperative Informationsverarbeitung in offenen Systemen oder mit Hilfe offener Systeme. In diesem Kontext sind mit Servern zumeist „Anbieter“ gemeint, Clients bezeichnen die „Konsumenten“. Dienstanbieter warten darauf, dass Dienstnutzer ihre Dienste in Anspruch nehmen (vgl. [HM05]).

Client-Server-Verbindungen werden in der Kommunikationstechnik topologisch unterschieden. Eine Client-Server-Kommunikation besitzt zwei Endpunkte. Der Anbieter stellt eine Funktionalität über ein Übertragungsprotokoll zur Verfügung und der Konsument verwendet diese. Bei der Übertragung zwischen einem Konsumenten und einem Anbieter wird in Punkt-zu-Punkt und Ende-zu-Ende Verbindung unterschieden. Bei einer Punkt-zu-Punkt Verbindung sind genau zwei Kommunikationspartner (Anbieter und Konsument) beteiligt. Bei einer Ende-zu-Ende-Verbindung können keine oder beliebige viele Kommunikationspartner zwischen dem Anbieter und Konsument beteiligt sein.

Die Kommunikation zwischen Konsument und Anbieter kann auf unterschiedliche Arten erfolgen. Diese werden im Allgemeinen als „Message-Exchange-Pattern“ (MEP) bezeichnet. Das wohl bekannteste MEP ist „Request-Response“, eine synchrone Kommunikation zwischen dem Konsumenten und dem Anbieter. Die Übermittlung ist erst abgeschlossen, wenn der Konsument eine Rückantwort vom Anbieter erhalten hat. Diese Kommunikationsart liegt auch dem „HTTP-Protokoll“ zugrunde. Beim MEP „Fire-and-Forget“ besitzt der Konsument keine Information über die erfolgreiche Übermittlung der Nachricht. Die Nachricht wird einmalig an den Anbieter gesendet. Eine Kombination aus Request-Response und Fire-and-Forget ist das „Publish-and-Subscribe“ MEP. Die Kommunikation zwischen dem Konsumenten und dem Anbieter erfolgt über das Request-Response MEP. Nach erfolgreicher Verarbeitung der Nachricht im Konsumenten, werden die Benachrichtigungen an jene über das Fire-and-Forget MEP versendet. Der Konsument besitzt somit keine Informationen darüber, welche Konsumenten die Nachricht erhalten haben. (s.a. [GHM+15], s.a. insbesondere [Erl05]).

Webapplikationen können über das Intra-/Internet in einem Browser ausgeführt werden, welcher als plattform- und ortsunabhängiger Client dient. Webanwendungen generieren den Großteil ihrer bereitgestellten Inhalte dynamisch und werden in [Ora14] als Erweiterung eines Web- oder Applicationservers betrachtet.

[1] Wichtige Teile dieses Version dieses Kapitels lieferten Christopher Addo und Jan-Tristan Rudat im Rahmen einer Hausarbeit im WS 2014/15

Weiter wird beschrieben, dass sich Webapplikationen in zwei Schichten unterteilen lassen würden. Präsentationsorientierte Webapplikationen generieren interaktive HTML, XML, XHTML-Dokumente in Abhängigkeit von Anfragen. Die serviceorientierten Webapplikationen stellen den Endpunkt eines abrufbaren Webservices bereit. Diese Services könnten von eingebetteten Systeme, Desktopapplikationen oder Mobileapplikationen zum Abrufen von Daten verwendet werden.

Dieses Kapitel beschreibt einige der Möglichkeiten zur Konzeption serviceorientierter Webapplikationen.

25.2 Lernziele

-

25.3 Einführung

Das Transmission Control Protocol (TCP) erledigt Transportaufgaben für verteilte Applikationen. IP dient als Mantel für die Paketdaten, gewährleistet seit jeher Adressierung. Mit dem User Datagram Protocol (UDP) lassen sich Datagramme verschicken, die keine Bestätigung erfordern. Die UDP-Paketübertragung ist verbindungslos. Das MEP Request-Response lässt sich auf TCP abbilden, UDP ist vergleichbar mit dem MEP Fire-and-Forget.

An den Endpunkten einer Netzwerk-Verbindung finden sich offene Sockets. Ein Socket ist mit einer Portnummer verbunden, so dass die TCP-Schicht in der Lage ist, Dienste und Services, welche an einer IP-Adresse zu finden sind, auseinanderzuhalten. Per IP werden nur Zielrechner adressiert und keine einzelnen Programme. Die direkte Prozessadressierung wäre ohne ein Protokoll-Port-Konzept problematisch. Ports lassen sich weiter zwischen den Contact Ports im Bereich 0 bis 1012 und den User Ports im Ziffernbereich 1024 bis 65535 unterscheiden. Aus den Tripeln (Internet-Adresse, Protokoll und Port) von Sender und Empfänger werden eindeutige Kommunikationseckpunkte gebildet. Verbindungen über eine URL werden ebenfalls per Sockets realisiert.

Für einen JAVA-Entwickler sind Sockets Kanäle auf denen Daten empfangen und gesendet werden können. Das java.net-Package stellt Werkzeuge für das Arbeiten mit Socketverbindungen zur Verfügung und ist fester Bestandteil seit JAVA 1.0. Socketprogrammierung ist plattformunabhängig und de facto Standard. Die Methodenaufrufe an java.net-Klassen werden durch die Java Virtual Machine (JVM) zu Aufrufen der Betriebssystem-API übersetzt. Grundsätzlich haben sich als Programmierschnittstelle die Berkeley Sockets etabliert, deren Bibliotheken für viele Betriebssysteme vorhanden sind.

Auf der Grundlage von TCP und UDP können eigene Protokolle entwickelt werden. Hierfür stellt JAVA Klassen wie SocketImpl, SocketImplFactory und DatagramSocketImpl zur Verfügung.

Eine ausführliche Beschreibung der Socket-Theorie finden Sie in [Tan03] und [MS09], ein Oracle-Tutorial zu Netzwerkgrundlagen in [Doc15].

Mit den Klassen Socket, InetAddress und ServerSocket lassen sich Stream- oder TCP-Sockets erzeugen. Im folgenden Beispiel wird eine Verbindung vom Konsumenten zum Anbieter (Host „haw-hamburg.de“) an Port 80 aufgebaut.

```
[caption={Socket},label=lst:SocketLst1]
Socket socket = new Socket("haw-hamburg.de", 80);
```

Auf dieser Verbindung können nun, ähnlich wie bei Daten aus Dateien, Streams aufgebaut werden. Das folende Beispiel angelehnt an [jav] zeigt den Verbindungsaufbau und die Datenübertragung per Streams:

25.4

25.5 Historische Anmerkungen

25.6 Aufgaben

Kapitel 26

Entfernter Methodenaufruf

26.1 Übersicht

Entfernte Objekte sind solche, die sich nicht im gleichen „Adressraum“ befinden. Methoden solcher Objekte können mittels Remote Method Invocation, kurz RMI, aufgerufen werden. Ziel von RMI ist es, den Zugriff auf Methoden entfernter Objekte und damit die Programmierung verteilter Anwendung mit der gleichen Syntax zu ermöglichen, wie die Programmierung lokaler Anwendungen. Technisch handelt es sich dabei um Objekte von Klassen, die das Interface *java.rmi.Remote* implementieren.

26.2 Lernziele

- Klassen schreiben können, die das Interface remote implemntieren.
- Die Reihenfolge der Schritte kennen, mit denen remote Objekte verfügbar gemacht werden können.
- Die Tools *rmic* und *rmiregistry* kennen.
- RMI-Server und -Clients verwenden können.

26.3 Einführung

Entfernte Objekte kann man auch als nicht-lokale Objekte bezeichnen. Um solche Objekte aus anderen Adressräumen zugreifbar zu machen, müssen sie ein Interface implementieren, die das Interface *java.rmi.Remote* erweitern. Letzteres ist ein Marker-Interface.

Die für RMI erforderliche Infrastruktur besteht aus Server, Client und Registry. Die Registry ist dabei für die Herstellung der Verbindung zwischen Client und Server verantwortlich. Das folgende Sequenzdiagramm in Abb. 26.1 zeigt den Ablauf der Kommunikation Eine Registry muss existieren oder neu erzeugt werden. Eine Standardanwendung für ein Singleton.

Bemerkung 26.3.1
An die Stelle der Registry können andere Mechanismen treten, wie etwa ein Corba (Common Object Request Broker Architecture). ORB. ◄

Ein Registry-Objekt kann mit verschiedenen Mechanismen erstellt werden:

1. Java kommt mit einem Utility *rmiregistry*. Dies wird unter Windows mittels

   ```
   start rmiregistry
   ```

Abb. 26.1: RMI-Sequenzdiagramm

aufgerufen und unter Unix ganz analog mittels

```
rmiregistry &
```

Dadurch wird ein Registry-Objekt in dem Adressraum (Rechner) erzeugt, in dem das Utility aufgerufen wurde. Ohne Parameter wird der Default-Port 1099 verwendet. Ein anderer Port kann als Parameter übergeben werden.

2. Ein Registry-Objekt kann vom Server selbst erstellt werden mittels der Klassenmethode *LocateRegistry.createRegistry(1099)*. Der Port 1099 ist der Default-Port für ein Registry-Objekt.

Sie brauchen sich wegen der oben genannten Techniken aber gar nicht darum zu kümmern. Ein Registry-Objekt erstellen Sie einfach auf einem der beiden Wege.

Das Serverobjekt muss sich bei einem Registry-Objekt registrieren. Das Registry-Objekt bekommt man mir der Klassenmethode

```
LocateRegistry.getRegistry();
```

Beispiel 26.3.2 (Entfernte Nachricht)
Als erstes Beispiel betrachten wir eine Klasse, die „Hello World"-artig eine Nachricht ausgeben kann, die das Objekt von einem Objekt auf einem Server holt. Den ersten Code, weitgehend am Java Tutorial download.oracle.com/javase/6/docs/technotes/guides/rmi/hello/hello-world.html orientiert, finden Sie in *rmi.Server.java* und *rmi.Client.java*. Hier haben Server und Client eine *main*-Methode. Diese Kann natürlich genausogut und vielleicht sogar besser in eine eigene Klasse ausgelagert werden.

Wesentlich an diesem Code sind die folgenden Dinge:

1. Der Server implementiert ein Unter-Interface von *Remote*:

   ```
   public interface IAnswer extends Remote {
       String answer() throws RemoteException;
   }
   ```

   Jede entfernte (remote) Methode muss eine RemoteException deklarieren. Dies ist notwendig, da diese eine solche werfen, wenn es zu Netzwerkproblemen irgendwelcher Art kommt.

2. In der main-Methode des Servers sind die folgenden vier Zeilen entscheidend:

   ```
   24   Server server = new Server();
   25   IAnswer stub = (IAnswer) UnicastRemoteObject.exportObject(server, 0);
   26
   27   server.registry = LocateRegistry.getRegistry();
   28   server.registry.bind("Answer", stub);
   ```

   In Zeile 24 wird ein neues Serverobjekt erstellt. Damit die Methoden des Servers von entfernten Objekten aufgerufen werden kann, wird ein sog. *Stub* erstellt. Dies ist ein Objekt, dass alle entfernten Operationen des Servers implementiert. Das heißt genauer: das Stub-Objekt hat hier den Typ *IAnswer*. Die Methode *stub.answer* delegiert an die answer-Methode des Servers und liefert dem Aufrufer das Ergebnis. Das Stub-Objekt wird von der Methode *UnicastRemoteObject.exportObject* erstellt. Eine Referenz auf ein solches Stub-Objekt erhält jeder Client (s. u.). Für die Einzelheiten verweise ich auf die ausführliche API-Dokumentation der Klasse *UnicastRemoteObject*.

   In Zeile 27 über die Klassenmethode *getRegistry()* von *LocateRegistry* eine Referenz auf Registry-Objekt beschafft. Bei diesem Objekt wird mit der *bind*-Methode in Zeile 28 der Server mit einem eindeutigen Namen und der Referenz auf den Stub registriert. Hierbei müssen Sie aber Folgendes beachten: Die Methode *getRegistry* versucht keinen Zugriff auf den Host. Sie erstellt lediglich eine lokale Referenz auf das (möglicherweise nicht existierende) Registry-Objekt auf dem Host, ist also immer erfolgreich, selbst wenn es kein Registry-Objekt auf dem Host gibt. Vergessen Sie also, vor dem Starten des Servers, ein Registry-Objekt zu erstellen (Siehe rmiregistry), so erhalten Sie eine Exception.

   Der Rest ist rudimentäre Fehlerbehandlung.

3. Die Methode *answer* ist hier trivial:

   ```
   public String answer() throws RemoteException {
       return "Hallo von Server!";
   }
   ```

Der Client ist hier ähnlich einfach. Den wesentlichen Code enthalten die folgenden Zeilen.

```
18   String host = (args.length < 1) ? null : args[0];

20   Registry registry = LocateRegistry.getRegistry(host);
21   IAnswer stub = (IAnswer) registry.lookup("Answer");
22   String response = stub.answer();
```

Der Client erhält den Host, auf dem der Server läuft, als Kommandozeilenparameter (Zeile 18). In Zeile 20 wird eine Referenz auf ein Registry-Objekt auf dem Host beschafft. Die funktioniert immer, Fehler treten ggf. bei der Verwendung auf (s. o.). In der nächsten Zeile 21 wird für den Client ein Stub-Object mit den Methoden des Interfaces *IAnswer* beschafft. Dessen Methode *answer* kapselt den Zugriff auf das entfernte Serverobjekt.

Es kann nun in Zeile 22 die *answer*-Methode des Stubs aufgerufen werden, um den entfernten Server zu erreichen.

Um den Code auszuprobieren müssen Sie jetzt folgende Schritte machen:

1. Utility rmiregistry ausführen.
2. Den Server starten.
3. Den Client starten.

◀

Ein weiteres Beispiel zeigt die einfache Date-Client-Server Anwendung aus [Pan08].

26.4 Historische Anmerkungen

26.5 Aufgaben

Kapitel 27

Datenbankzugriff aus Java

27.1 Übersicht

Ich stelle in diesem Kapitel einige Möglichkeiten vor, um *Java* und (relationale) Datenbanken zu verbinden. In einer objekt-orientierten Sprache programmieren Sie objekt-orientiert. Auf relationale Datenbankengreifen Sie mengen-orientiert zu. Das ist ein grundlegender Unterschied. Um von dem einen System in das andere zu kommen und zurück, müssen Sie etwas tun. Konzepte wie Iteratoren und eingebettete SQL in C oder C++ können dabei hilfreich sein, führen aber zu umfangreichem SQL-Code in Ihren entsprechenden Klassen, können dynamisches SQL erzwingen und zu anderen Unannehmlichkeiten führen. Für den Zugriff aus Java gibt es *JDBC*. Der direkte Umgang mit JDBC ist aber nicht besonders komfortabel. Da ist dann ein Framework wie *Hibernate* sehr hilfreich. Das werden ich in Ihnen diesem Kapitel vorstellen.

27.2 Lernziele

- Aus *Java* mittels *JDBC* auf *relationale Datenbanken* zugreifen können.
- Hibernate mittels Annotationen oder XML konfigurieren können.
- Unter Verwendung von *Hibernate* auf relationale Datenbanken zugreifen können.

27.3 Einführung

27.4 JDBC - Grundlagen

SQL sollte Ihnen bekannt sein. Ich erläutere daher hier nur bei Bedarf einige Dinge, die mir wichtig und Ihnen vielleicht nicht so geläufig sind.

JDBC (Java Data Base Connectivity) erfordert den geringsten Konfigurationsaufwand, um auf eine relationale Datenbank zuzugreifen. Das einzige, was Sie brauchen, ist eine Datenbank (URL), zu der Sie eine Benutzerkennung und ein Password haben und ein passender Treiber. Ich verwende ein ganz einfaches Beispiel einer Tabelle *Customer* mit vier Spalten: Einem ganzzahligen Primärschlüssel, zwei String-Spalten und einer Datumsspalte.

Columnname	Datatype	nullable
ID	NUMBER(32,0)	No
FIRSTNAME	VARCHAR2(50 BYTE)	Yes
FAMILYNAME	VARCHAR2(50 BYTE)	Yes
ENTRYDATE	DATE	Yes

Ich lege diese Tabelle an und trage einen ersten Satz ein, den ich dann lesen und ausgebe. Hier der einfache Code, den ich anschließend erläutern werde.

```
100     OracleDataSource ods = new OracleDataSource();
110     ods.setUser(Messages.getString("DBAccessSimple01.user")); //$NON-NLS-1$
120     ods.setPassword(Messages.getString("DBAccessSimple01.psw")); //$NON-NLS-1$
130     ods.setURL(Messages.getString("DBAccessSimple01.connstring")); //$NON-NLS-1$
140     Connection conn = ods.getConnection();
150     try (Statement stmt = conn.createStatement();
160          ResultSet rset = stmt.executeQuery("select FIRSTNAME, FAMILYNAME, ENTRYDATE
                                                 from CUSTOMER")) { //$NON-NLS-1$
170         while (rset.next())
180             System.out.printf("%-50s %-50s %s\n",  //$NON-NLS-1$
190                       rset.getString(1),
200                       rset.getString(2),
210                       rset.getDate(3).toLocalDate().
220                            format(DateTimeFormatter.ofPattern("dd.MM.YYYY"))); //$NON-NLS-1$
230     }
```

1. In Zeile 100 erzeuge ich ein Objekt der Klasse *OracleDataSource*.

2. Um die Verbindung herzustellen brauche ich drei Dinge, die ich in eine Properties Datei ausgelagert habe (siehe Kap. 21, Abschn. 21.5). Schließlich will ich Ihnen zumindest das Passwort nicht offenlegen. Dies sind die Zeilen 110 – 130.

   Benutzerkennung Diese steht unter dem Schlüssel „user“ in der Properties-Datei und wird mittels der Klassenmethode *getString* der Klasse *Messages* gelesen. Gesetzt wird sie mittels der Methode *setUser*.

   Passwort Dies steht ganz analog unter „psw“ in der Properties-Datei. Dies wird durch *setPassword* gesetzt.

   DB Connection String Der Connection String lautet für die Installation, die ich nutze:

   ```
   jdbc:oracle:thin:@ora14.informatik.haw-hamburg.de:1521:inf14
   ```

   Der Connection String besteht aus folgenden Teilen, die jeweils durch einen Doppelpunkt getrennt werden:

   - Dem Treibernamen.
   - Einem @ gefolgt von der URL der Datenbank.
   - Dem Port, hier 1521
   - Der System Id, hier inf14

   Er wird durch *setURL* gesetzt.

   Näheres dazu etwas später.

3. Die Kommentare`//$NON-NLS-1$` wurden von Eclipse beim Externalisieren der Strings eingefügt. Sie verhindern, dass diese Strings ggf. nochmals externalisiert werden. In Zeile 160, 180 und 220 stehen sie, weil ich diese Strings nicht externalisieren wollte.

4. Die Verbindung zur Datenquelle wird durch ein in Zeile 140 erstelltes *Connection* Objekt hergestellt. *Connection* ist ein Interface aus java.sql. Wollen Sie Oracle-Spezifika nutzen, so erzeugen Sie ein *OracleConnection* Objekt.

5. Durch getConnection wird versucht eine Verbindung herzustellen. Dabei kann eine *SQLException* geworfen werden.

6. In Zeile 150 beginnt ein try-with-resources. Hier werden ein SQL-*Statement* und ein *ResultSet* erzeugt. Beide sind *AutoCloseable*, daher ist dies die einzig sinnvolle Vorgehensweise.

7. Über das ResultSet iteriere ich mittels while-Schleife und gebe die Treffer mittels *printf* formatiert aus. Um Platz zu sparen, habe ich leichtsinnigerweise bei diesem Einzeiler das geschweifte Klammerpaar wegelassen.

Nun zum elementaren Einfügen von Sätzen:

```
100     try (PreparedStatement pstmt = conn.prepareStatement("insert into CUSTOMER"
110             + "(ID, FIRSTNAME, FAMILYNAME,ENTRYDATE)"
120             + "values (CUSTOMERSEQ.nextval, ?, ?, ?)")) {
130         pstmt.setString(1, "Max");
140         pstmt.setString(2, "Motte");
150         pstmt.setDate(3, Date.valueOf(LocalDate.now()));
160         pstmt.execute();
170         System.out.println(pstmt.getUpdateCount() + " rows updated");
180     } catch (SQLException e) {
190         e.printStackTrace();
200     }
```

Hierzu die folgenden Erläuterungen

1. Auch hier verwende ich natürlich wieder try-with-resources (Zeile 100). In diesem Fall erstelle ich ein *PreparedStatement*. Ein solches kann effizient mehrfach verwendet werden. Die erste Postion beim insert fülle ich mit dem nächsten Primärschlüssel aus der zugehörigen Sequence. Die mit „?“ versehenen fülle ich nmit den entsprechenden setXXX Methoden.

2. Für die primitiven Typen und einige weitere in Java gibt es setXXX-Methoden, von denen ich hier *setLong* und *setString* verwende (Zeilen $130 - 150$).

3. Setzen des Datums erfolgt mittels *setDate*. Java arbeitet mit LocalDate. Hierfür hat die Klasse *java.sql.Date* die Klassenmethode *valueOf*, die aus einem *LocalDate* ein *jva.sql.Date* erzeugt.

4. Um den Code einfach zu halten verwende ich eine default-Einstellung von Connection: Autocommit. Das ist nicht unbedingt sinnvoll, kürzt aber den Code, den ich hier erläutern muss.

Will ich solche Dinge Testen, so ist JUnit nützlich. Dabei ist folgendes zu beachten bzw. vielleicht zu empfehlen:

- Vieles ist sinnvoll in einer *@BeforeClass* Methode zu erledigen. Dazu gehört auf jeden Fall das Herstellen der Verbindung zur Datenbank. Manchmal ist es auch sinnvoll, die Tabellen jedesmal zu löschen, neu anzulegen und mit geeigneten Testdaten zu befüllen.

- Je nach Autocommit Einstellung kann es sinnvoll sein, nach jedem Testfall ein commit abzusetzen (@After)

Eine Benutzerkennung und ein Password werden Sie aber kaum in Source-Code schreiben wollen. Zum Testen ist es natürlich sehr bequem das zu tun. So müssen Sie nicht für jeden Testfall die Daten eintippen. Für die DB_URL bietet sich eine Property-Datei an. Auch die Auslagerung von Benutzerkenung und Passwort in eine Properties-Datei ist ein richtiger Schritt, den ich oben gewählt habe.

Wenn die Umgebung entsprechend konfiguiert ist, können Sie die Daten auch mit Java-Mitteln direkt erhalten, am AIL z. B. so:

```
System.getenv("HAW_USER");
System.gentenv("HAW_PASSWORD");
```

Vielen Dank an Sebastian Wagner für diesen Hinweis. Die Auslagerung in eine Properties-Datei mag für Testzwecke akzeptabel erscheinen, ist aber auch nicht professionell. Zum Testen können die Daten auch in eine verschlüsselte Datei geschrieben werden und dann immer wieder daraus eingelesen werden. Siehe hierzu die Java Cryptography Architecture (JCA). Die Klassen finden Sie in *javax.crypto* und den Unterpaketen.

Eine einfache Möglichkeit bietet hier Swing, etwa so:

```
/**
 * Gathers userdata from user.
 * @return Array of {@link String} with data to sign in to the database.
 */
   private static String[] getUserData() {
       String [] userData = new String[2];
       JPasswordField pswField = new JPasswordField();
       pswField.setEchoChar('*');
       userData[0] = JOptionPane.showInputDialog("Bitte Benutzerkennung eingeben");
       Object[] message = { "Bitte Passwort eingeben!\n", pswField };
       int resp = JOptionPane.showConfirmDialog(null, message,
                                                "Retrieve Password",
                                                JOptionPane.OK_CANCEL_OPTION,
                                                JOptionPane.QUESTION_MESSAGE);
       if(resp==JOptionPane.OK_OPTION){
           userData[1] = new String(pswField.getPassword());
       }else{
           System.exit(-1);
       }

       return userData;
   }
```

Um auf eine Datenbank zugreifen zu können, müssen Sie diese kennen und Ihrem Programm mitteilen können, wo und wie es an diese Datenbank herankommt. Was Sie dazu benötigen und wie Sie dies angeben müssen hängt vom jeweiligen DBMS, dem verwendeten Treiber (driver) und dem für den Zugriff verwendeten System ab. Hier ein Beispiel, wie sich das für Sie darstellen kann:

Den JDBC Connection String (siehe http://www.orafaq.com/wiki/JDBC, zuletzt besucht am 23.08.2015), gibt es in zwei Versionen (Vielen Dank an Michael Brodersen für diese Recherche):

> Oracle's JDBC Thin driver uses Java sockets to connect directly to Oracle. It provides its own TCP/IP version of Oracle's SQL*Net protocol. Because it is 100% Java, this driver is platform independent and can also run from a Web Browser (applets). There are 2 URL syntax, old syntax which will only work with SID and the new one with Oracle service name. Old syntax
>
> ```
> jdbc:oracle:thin:@[HOST][:PORT]:SID
> ```
>
> New syntax
>
> ```
> jdbc:oracle:thin:@//[HOST][:PORT]/SERVICE
> ```
>
> On new syntax SERVICE may be a oracle service name or a SID. There are also some drivers that support an URL syntax which allow to put Oracle user id and password in URL.
>
> ```
> jdbc:oracle:thin:[USER/PASSWORD]@[HOST][:PORT]:SID
> jdbc:oracle:thin:[USER/PASSWORD]@//[HOST][:PORT]/SERVICE
> ```

Letzteres habe ich noch nicht weiter recherchiert und mit dem aktuellen Oracle-Treiber noch nicht ausprobiert.

Beispiel 27.4.1 (Datenbankverbindung)
Um die Datenbank zu identifzieren gibt es mehrere Parameter, die ich hier für beide Varainten des Oracle Treibers angebe:

Treiber Der verwendete Datenbanktreiber

Host Die *URL* unter der das DBMS zu erreichen ist, z. B. ora14.informatik.haw-hamburg.de.

SID Service ID: Eine Kennzeichnung,unter das DBMS auf dem Host angesprochen werden kann.

SERVICE SID oder Service-Name.

Portnumber Der *Port* über den auf das DBMS zugegriffen werden kann.

Für ein System an der HAW ist z. B. der Host

Info	Name	Bemerkung
Treiber	jdbc:oracle:thin	
Host	ora14.informatik.haw-hamburg.de	
Port	1521	
SID	inf14	
SERVICE	inf14.informatik.haw-hamburg.de	

Für einen Zugriff aus Eclipe EE mit JPA-Unterstützung müssten Sie z. B. Von der SID nur inf14 angeben. Für einen Zugriff mit Hibernate müssen Sie die SID komplett angeben. In jedem Fall wird daraus folgender String zusammengesetzt: *jdbc:oracle:thin:@ora14.informatik.haw-hamburg.de:1521:inf14.informatik.haw-hamburg.de*

◀

Welche Java Datentypen unterstützt werden hängt von der Version des Treibers ab. Java 8 Datentypen werden auf einfache Art und Weise ab JDBC 4.2 unterstützt.

Direktes Arbeiten mit JDBC ist aber etwas mühselig. So müssen Sie die Datentypen selber durch Aufruf der entsprechenden Methoden von dem SQL-Typ in den Java-Typ umsetzen und zurück.

27.5 JPA und Hibernate - Übersicht

Die Java Persistence Architecture (JPA) definiert ein API, um aus Java auf (relationale) Datenbanken zuzugreifen. *Hibernate* ist eine Implementierung dieses APIs. Hibernate verwendet

- *Annotation*en für persistente Klassen und deren Elemente.
- *XML*-Dateien, in denen die benötigten Datenbankressourcen definiert werden.
- JPA-QL als Abfragesprache.

In beiden Fällen erfolgt eine Abbildung von Datenbanktabellen auf Java-Klassen. Außerdem werden die Datentypen der Tabellenspalten auf Java-Datentypen abgebildet und zurück.

Insgesamt müssen Sie drei Aspekte berücksichtigen:

Java	JPA-QL	DB
Klasse	Entity	Tabelle

Der default ist, dass alle drei den gleichen Namen haben (bis auf Groß/Kleinschreibung bei Tabelle). Es können aber für Entity bzw. Tabelle jeweils eigenen Namen gewählt werden und eine Klasse kann auf mehrere Tabellen abgebildet werden. **Hinweis:** Die XML-Dateien müssen - zumindest bei der Arbeit mit Eclipse - im Verzeichnis src/META_INF/ liegen. Dieses muss dann im Source-Tab des Java Build Path stehen. Generell müssen diese XML-Dateien im Classpath liegen.

27.6 Hibernate und XML

Der Einstiegspunkt für Hibernate in die Datenbankverbindung ist die XML-Datei *persistence.xml*. Diese wird vom EntityManager verwendet, sie konfigurieren ihn also durch diese. Für das Beispiel aus Abschn. 27.4 kann diese Datei etwa so aussehen:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                            http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
                            version="2.0">
   <persistence-unit name="de_hawh_kahlbrandt">
      <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
      <properties>
          <property name="hibernate.connection.driver_class"
                    value="oracle.jdbc.driver.OracleDriver"/>
          <property name="hibernate.connection.url"
                    value="jdbc:oracle:thin:@//ora14.informatik.haw-hamburg.de:
                           1521/inf14.informatik.haw-hamburg.de"/>
          <property name="hibernate.connection.username" value="user"/>
          <property name="hibernate.connection.password" value="psw"/>
          <property name="hibernate.dialect" value="org.hibernate.dialect.Oracle10gDialect"/>
          <property name="hibernate.default_schema" value="khb"/>
          <property name="hibernate.show_sql" value="true"/>
          <property name="hibernate.format_sql" value="true"/>
      </properties>
   </persistence-unit>
</persistence>
```

Auch diesen Code erläutere ich tag für tag:

1. Die erste Zeile enthält die processing instruction `<?xml>`. Groß- und Kleinschreibung wird dabei nicht beachtet. Es gibt kein schließendes tag `</?xml>`. Außer den hier verwendeten beiden Attributen gibt es noch `standalone`, für das no oder yes angegeben werden kann. Ich verwende in der Regel UTF-8 in Programmcode. Verwenden Sie einen anderen Zeichensatz, so müssen Sie natürlich diesen angeben, etwas ISO-8859-1.

2. `<persistence>` definiert in den Attributen die grundlegenden Dinge für JPA.

3. Die `persistence-unit` definieren Sie eine Art Namensraum für ihre persistenten Dinge. Den Namen wählen Sie und Sie können eine Beschreibung mittels `<description>`angeben. Auf letzteres habe ich verzichtet.

4. Für die persistence-unit müssen Sie den (vollqualifizierte) Namen der Klasse des zu verwendenden Treibers angeben.

5. Die beiden Zeilen mit username und password gefallen mir gar nicht. Ich suche noch noch einer Möglichkeit, sie zu eliminieren.

6. Der dialect spezifiziert den (propriäteren) SQL-Dialekt, den „Ihre“ Datenbank „spricht“. Die Dokumentation dafür erscheint dürftig. Auf stackoverflow habe ich die Aussage gefunden: „Copy-paste whatever you can find on google and pray to God.“ Sollte Ihnen der hier verwendete entgangen sein: Googlen nach „oracle hibernate.dialect“ liefert den Treffer auf der ersten Seite.

7. show_sql sollte für Produktionsumgebungen aus *false* gesetzt werden. Der Wert *true* sorgt dafür dass die Queries von log4j gelogged werden.

Der nächste Schritt ist das Erzeugen des *EntityManagers*.

27.7 Hibernate und Annotationen

Java-Klassen und Datenbanktabellen können über Annotationen in den Java-Klassen zugeordnet und damit für Hibernate verfügbar gemacht werden, siehe Abschn. 27.7. Das ist manchmal praktisch, wenn Sie die Klassen selber schreiben. Bekommen Sie Klassen im Rahmen einer Software ohne Source-Code, so haben Sie diese Möglichkeit nicht. Dann können Sie dies mittels XML tun, siehe 27.6.

27.8 Mappings

Hier einige wichtige Zuordnungen zwischen Java- und DB-Datentypen[1].

Mappings - Auswahl

SQL	Java
DECIMAL	BigDecimal
DATE	LocalDate
BIGINT	Duration
TIME	LocalTime
Converter verwenden	Period
Converter verwenden	Duration

```
@Converter
public class PeriodStringConverter implements AttributeConverter<Period, String> {
    @Override
    public String convertToDatabaseColumn(Period period) {
        return period.toString();
    }

    @Override
    public Period convertToEntityAttribute(String period) {
        return Period.parse(period);
    }
}
```

27.9 Spring

27.10 Historische Anmerkungen

Hibernate und *JBoss*

27.11 Aufgaben

1. Welche Vor- bzw. Nachteile haben die Konfigurationsmöglichkeiten von Hibernate über XML-Dateien bzw. über Annotationen in Java-Klassen?

[1] Siehe Hibernate 5.1.0 Mapping Guide

Kapitel 28

Das Java Native Interface

28.1 Übersicht

Das *Java Native Interface* (*JNI*) ermöglicht es, aus Java Code heraus Programme in anderen Programmiersprachen aufzurufen. Tatsächlich handelt es sich dabei um Funktionen, für die entsprechende Methoden in Java-Klassen definiert werden müssen. *C* und *C++* Programme lassen sich direkt einbinden. Für andere Sprachen müssen Sie ggf. einen *C* oder *C++* Wrapper schreiben. Java erwartet den „Fremdcode“ in Windows als *DLL* (Dynamic Link Library) und in Unix als *Shared Object* (.so Datei). Das JNI bietet auch ein Mapping für die Datentypen in Java auf die in C bzw. C++. Dazu später mehr.

Auch die andere Richtung funktioniert: Sie können von anderen Programmen aus Java-Methoden aufrufen. Dazu müssen Sie aus Ihrem Programm eine *JVM* starten. Auch dafür bietet Java Ihnen Methoden.

28.2 Lernziele

- Die Bedeutung des Schlüsselworts *native* kennen.
- Das *Java Native Interface (JNI)* kennen.
- Auf Basis von C Code in Java Programmen nutzen können, die in anderen Programmiersprachen geschrieben wurden.

28.3 Grundlagen

Eine Methode kann in Java als *native* deklariert werden.

Beispiel 28.3.1 (native Methoden)
Die Klasse *RandomAccessFile* deklariert etwa einige native Methoden

```
package java.io;
public class RandomAccessFile
  implements DataOutput, DataInput
  { ...
      public native void open(String name, boolean writeable)
             throws IOException;
      public native int readBytes(byte[] b, int off, int len)
             throws IOException;
      public native void writeBytes(byte[] b, int off, int len)
             throws IOException;
```

```
        public native long getFilePointer() throws IOException;
        public native void seek(long pos) throws IOException;
        public native long length() throws IOException;
        public native void close() throws IOException;
}
```

◀

Eine als native deklarierte Methode hat keinen Body. Die Deklaration endet also mit dem Semikolon. Implementiert werden native Methoden in C, C++ oder FORTRAN etc.

Hier eine Art „Hello World“ Code für JNI:

```
public class HelloJNI {
  static {
    System.loadLibrary("hello");
  }

  /**
   * Declare a native method sayHello() that receives nothing and returns void.
   */
  private native void sayHello();

  /**
   * Runs the app, to demonstrate functionality.
   */
  public static void main(String[] args) {
    new HelloJNI().sayHello();
  }
}
```

Die *native method* ist *sayHello*. Sie hat keine Implementierung in Java, die müssen Sie in C-Code haben bzw. schreiben. Der statische Initialisierungsblock (siehe Abschn. 4.3). Um damit irgend etwas anfangen zu können, müssen Sie eine C Headerdatei erzeugen. Das geht mit dem Java Compiler *javac* mit der Option -h:

```
javac -h outputdirectory HelloJNI.java
```

Für eine Übersicht der Optionen von *javac* verweise ich auf Abschn. 21.4.

Ich habe mich hier entschieden, die Header-Datei im Paket jni erzeugen zu lassen. Eine gute Alterntive ist im Kontext der Unterrichtsbeispiele ein Verzeichnis „c“ unter jni. Hier das Ergebnis:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class de_hawh_kahlbrandt_jni_HelloJNI */

#ifndef _Included_de_hawh_kahlbrandt_jni_HelloJNI
#define _Included_de_hawh_kahlbrandt_jni_HelloJNI
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     de_hawh_kahlbrandt_jni_HelloJNI
 * Method:    sayHello
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_de_hawh_kahlbrandt_jni_HelloJNI_sayHello
```

```
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Hier haben Sie es mit viel Code des C-Preprozessors zu tun. Den können Sie zunächst ignorieren, wenn Sie C nicht kennen. Wichtig ist für Sie nur die Definition der Methode

```
JNIEXPORT void JNICALL Java_de_hawh_kahlbrandt_jni_HelloJNI_sayHello(JNIEnv *, jobject)
```

Hierzu diese Erläuterungen zum generierten Code:

- `JNIEXPORT`
- `void`: Keine Rückgabe; gleiche Bedeutung, wie in Java.
- `JNICALL`
- Der Methodenname beginnt mit `Java`. Die Punkte im vollqualifizierten Klassennamen und der Punkt zwischen Klassenname und Methodenname wurden durch Unterstriche ersetzt.

Um nun tatsächlich die notwenidgen ausführbaren Dateien erzeugen zu können, brauchen Sie noch einige weitere Informationen zu notwenfigen Dateien für den C und den Java Compiler.

28.4 Aufruf von C-Code

28.5 Historische Anmerkungen

Seit Java 8 wird *javac* mit der Option *-h* zum Erzeugen von Header-Dateien verwendet, vorher *javah*.

28.6 Aufgaben

1.

Kapitel 29

Graphische Oberflächen

29.1 Übersicht

Graphische Oberflächen werden heute zu einem großen Teil in Browsern eingesetzt bzw. für Browser entwickelt. Für manche Anwendung benötigt man sie aber auch ohne Browser. Dieses Kapitel führt in die Programmierung graphischer Client-Oberflächen in Java ein. Als Basis habe ich mich für Swing entschieden. Die Grundprinzipien sind aber für viele Frameworks gleich. Ob es nun Observer Pattern, Model-View-Controller oder Document Object Model heißt: Das Grundprinzip ist immer das gleiche. Ein Objekt wird von einem oder mehreren anderen verändert und es werden jeweils alle Beteiligte informiert, damit sie geeignet reagieren können (Siehe Kap. 23).

29.2 Lernziele

- Basisklassen für graphische Oberflächen benutzen können.
- Grundlegende Strukturen von GUI Frameworks kennen.
- Den Umgang mit Listener beherrschen.
- Mit Layout-Managern arbeiten können.

29.3 Einführung

Als Einführung beginnen wir mit einem einfachen Dialog um Dateien zu öffnen oder zu suchen. Dies ist professioneller, als die Abfrage über die Console. Als Basis für die Oberfläche verwende ich Swing. Swing-Oberflächen werden mit einem einheitlichen Erscheinungsbild gestaltet, dem sogenannten *Look & Feel*. Die Standard Look & Feels sind

Metal

Windows

Motif

Voreingestellt ist das Metal Look & Feel. Möchte man mit dem arbeiten, so muss man gar nichts tun. Ich entscheide mich für das Windows Look & Feel: Dies setze ich als erstes mittels einer Klassenmethode der Klasse *UIManager*:

```
UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
```

In diesem Fall wird der Dialog also unabhängig von der Umgebung mit dem Windows Look & Feel ausgestattet. Will man das Look & Feel an das der jeweiligen Laufzeitumgebung anpassen, so verwende man:

```
UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
```

Welche Look & Feels zur Verfügung stehen findet man mit einer weiteren Klassenmethode der Klasse *UIManager* heraus: Der folgende Code

```
for(UIManager.LookAndFeelInfo laf: UIManager.getInstalledLookAndFeels())
   System.out.println(laf.getClassName());
```

liefert für das System an dem ich jetzt gerade sitze

javax.swing.plaf.metal.MetalLookAndFeel
com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel
com.sun.java.swing.plaf.motif.MotifLookAndFeel
com.sun.java.swing.plaf.windows.WindowsLookAndFeel
com.sun.java.swing.plaf.windows.WindowsClassicLookAndFeel

Will man die Auswahl des Look & Feels ganz flexibel gestalten, so kann man so vorgehen:

- Definiere einen Container, z. B. eine Liste oder ein Array von UIManager.LookAndFeelInfo oder Strings mit UIManager.LookAndFeelInfo.getClassName().
- Schreibe einen Dialog, der diese Auswahlmöglichkeiten, z. B. über Checkboxen anbietet.
- Passe das Look & Feel jeweils der Wahl des Benutzers an.

Der Rest des Codes ist ganz einfach

```
 UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
JFileChooser chooser = new JFileChooser();
chooser.setDialogTitle("Datei öffnen");
chooser.showOpenDialog(null);
System.out.printf("%s%s%s\n", "Sie haben: ",
chooser.getSelectedFile() == null ? "keine Datei " : chooser
.getSelectedFile().getAbsolutePath(), " ausgewählt");
```

Wichtig sind hier zunächst die ersten drei Zeilen. Die letzte dient nur dazu, sich davon zu überzeugen, dass soweit Alles geklappt hat.

Wir müssen nun nur noch dafür sorgen, dass der *JFileChooser* so lange ausgeführt wird, bis eine Datei oder abschließend keine, ausgwählt wurde.

29.4 Swing Schritt für Schritt

In diesem Abschnitt führe ich Schritt für Schritt einige Swing-Elemente ein.

29.4.1 Schritt 1: Ein einfaches Fenster

Die Folgende einfache Klasse zeigt lediglich ein kleines Fenster, genaue nur den Kopfbalken (caption) an.

```
public class Schritt01 extends JFrame {
/**
 * @author Bernd Kahlbrandt
 */
public Schritt01(){
this.setTitle("Swing Schritt für Schritt, Schritt 01");
this.setDefaultCloseOperation(EXIT_ON_CLOSE);
this.pack();
```

```
this.setVisible(true);
}
public static void main(String[] args) {
new Schritt01();
}
}
```

Dieses kann aber schon verschoben, in der Größe verändert, minimiert , maximiert und geschlossen werden.

Bemerkung 29.4.1 (Serialization)
Früher wurden Swing-Objekte bei Bedarf mittels Serialisierung weggeschrieben und wiederhergestellt. Inzwischen wird hierfür die Nutzung aus dem Paket *java.beans* empfohlen. Insofern kann bei Swing-Klassen auf ein Klassenattribut *serialVersionUID* verzichtet werden. Die dadurch verursachte Warnung kann durch eine *@SuppressWarning* Annotation unterdrückt werden. ◄

29.4.2 Schritt 2: Hinzufügen einer Menüleiste

Im nächsten Schritt füge ich eine Menüleiste hinzu. Dazu erst einmal einige Infos zur Struktur eines JFrames. Aus der Java API Dokumention sehen Sie, dass ein JFrame ein Attribut rootPane der Klasse JRootPane besitzt. Eine JRootPane wiederum hat u. a. ein Attribut menuBar der Klasse JMenuBar und entsprechende Methoden setJMenuBar und getMenuBar. Zunächst einmal ergänze ich also typische Windows Menüs wie Datei, Bearbeiten, Fenster und Hilfe.

```
public class Schritt02 extends JFrame {
/**
 * @author Bernd Kahlbrandt
 * Einfachstes Fenster mit Menüleiste
 */
private JMenu file = new JMenu("Datei");
private JMenuItem open = new JMenuItem("Öffnen");
private JMenuItem exit = new JMenuItem("Beenden");
private JMenu edit = new JMenu("Bearbeiten");
private JMenu window = new JMenu("Fenster");
private JMenu help = new JMenu("Hilfe");
private JMenuItem helpcontents = new JMenuItem("Hilfe Inhalt");
private JMenuItem about = new JMenuItem("Über");

public Schritt02(){
this.setTitle("Swing Schritt für Schritt, Schritt 01");
this.setDefaultCloseOperation(EXIT_ON_CLOSE);
this.createMenus();
this.createMenuBar();
this.pack();
this.setVisible(true);
}
private void createMenus() {
file.add(open);
file.add(exit);
help.add(helpcontents);
help.add(about);
}
private void createMenuBar() {
this.rootPane.setJMenuBar(new JMenuBar());
this.rootPane.getJMenuBar().add(file);
```

```
this.rootPane.getJMenuBar().add(edit);
this.rootPane.getJMenuBar().add(window);
this.rootPane.getJMenuBar().add(help);
}
public static void main(String[] args) {
new Schritt02();
}
}
```

Die Menüs und die Menüpunkte habe ich als Attribute definiert. So kann ich später leicht aus anderen Methoden darauf zugreifen. Die Anwendung tut immer noch nichts. Wir haben uns aber auch noch gar nicht überlegt, was dass für eine Anwendung werden soll. In den nächsten Schritten werde ich nun an weitgehend anwendungsneutralen Elementen zeigen, wie man auf Aktionen des Benutzers mit Maus oder Tastatur reagiert.

29.4.3 Schritt 3: Reagieren auf Menüauswahl

Um auf Aktionen des Benutzers (events), sei es über Tastatur oder Maus, zu reagieren, müssen ActionListener definiert werden. Dies kann auf verschiedenen Wegen geschehen:

- Definition einer inneren Klasse in der Klasse unseres JFrame, Schritt*nn*.
- Verwendung einer anonymen Klasse an der jeweiligen Stelle.
- Implementierung der Schnittstelle ActionListener in der Klasse Schritt*nn*.
- Implementierung einer (oder mehrerer) Klassen, die ActionListener implementieren.

Ich entscheide mich hier für die letztgenannte Möglichkeit. Sie entspricht guten objektorientierten Entwurfsprinzipien. Wie wir sie genau nutzen, wird sich in den nächsten Schritten zeigen. In der ersten Version definiere ich eine Klasse MenuListener03.

Die drei erstgenannten Möglichkeiten zur Implementierung der ActionListener werden in den Klassen Schritt*03*a,b,c gezeigt.

29.4.4 Schritt 4

Um die Elemente *JTabbedPane* zu üben und ein oft wiederverwendbares Teil zu demonstrieren zeige ich nun noch den Einbau eines Dialoges um weitgehend beliebige Einstellungen vornehmen zu können. Die Grundidee des Aufbaus zeigt Abb. 29.1 Um so etwas einfach zu machen können Sie GUI-Builder verwenden. Auch wenn es hier um Java geht, können Sie sich mit Produkten wie Forms aus Microsoft Visual Studio einen Entwurf machen. Ich zeige hier grob, wie man das „zu Fuß“ machen kann.

Um ein solches Fenster aufzubauen überlege ich mir zunächst Folgendes:

- Es soll ein Dialog werden. Ich nenne die Klasse deshalb *OptionsDialog*.
- Die Karteikarten sind das wesentliche Element. Dazu kommen noch die drei Buttons. Karteikarten werden in einer *JTabbedPane* plaziert (siehe Java API Dokumentation).
- Ich wähle deshalb das *Borderlyout* und setze eine JTabbedPane auf *NORTH*.
- Die Buttons setze ich in einen *JPanel*. Um sie von links nach rechts anzuordnen wähle ich das *FlowLayout* mit der Option *LEFT*. Alternativen sind *CENTER* oder *RIGHT*.

Jede der Karteikarten in der TabbedPane hat Ihre eigene Logik. Wir werden später noch weitere Einstellungen benötigen. Ich demonstriere dies zunächst an einer Karte, um das Look & Feel zu wählen.

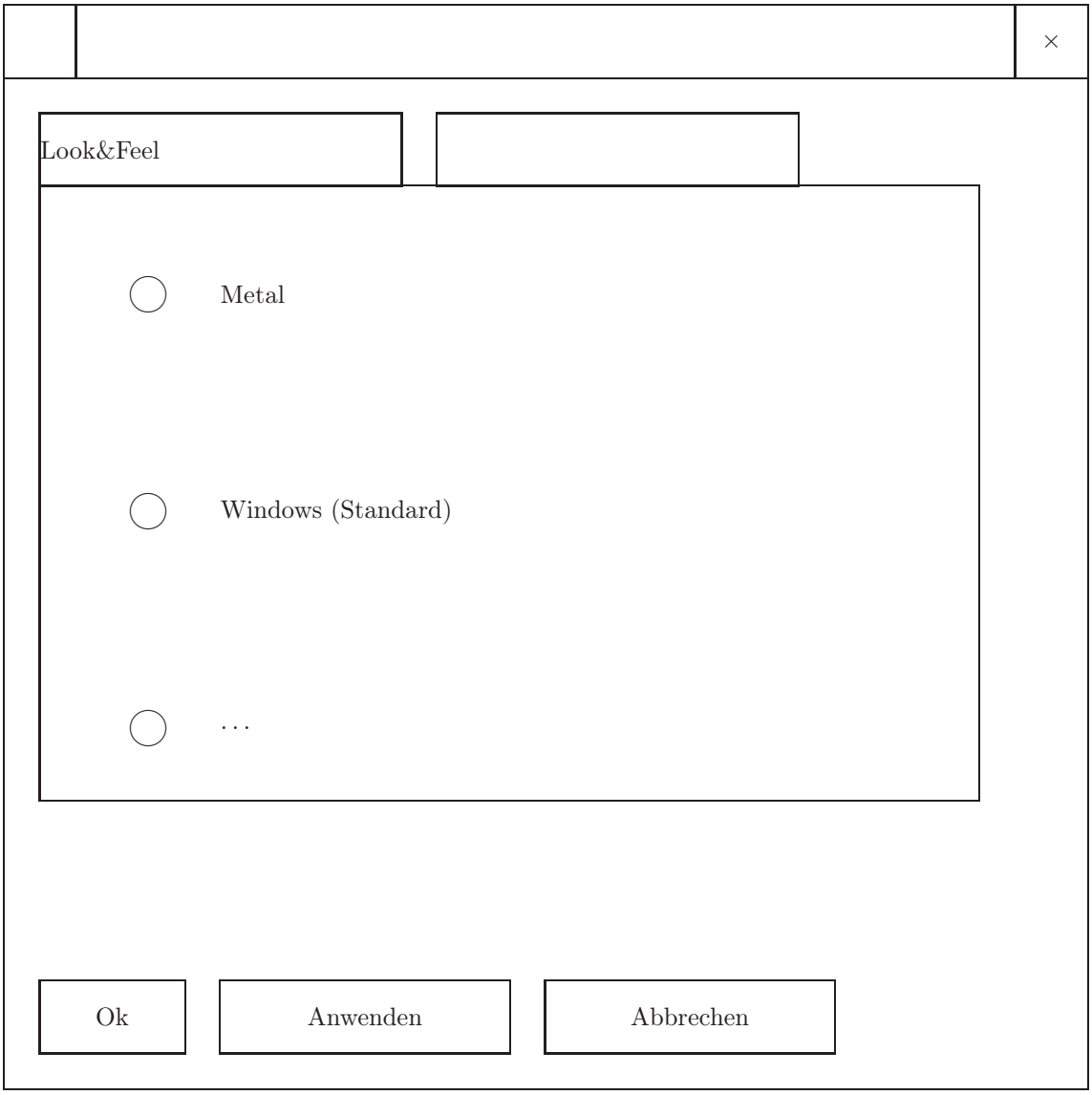


Abb. 29.1: Einstellungsdialog

Wie er aussehen soll zeigt bereits Abb. 29.1. Ich definiere mir eine Klasse *LookAndFeelSelection* als Unterklasse von *JPanel*. Dieser verpasse ich ein *GridLayout* mit so vielen Zeilen, wie Look & Feels vorhanden sind, und einer Spalte. Hier nun dieser Stand des Codes (bis auf package und import Befehle):

```
public class LookAndFeelSelection extends JPanel {
   private UIManager.LookAndFeelInfo[] lookAndFeelsAvailable = UIManager
      .getInstalledLookAndFeels();
   private Window[] toUpdate = new Window[0];
   private JRadioButton[] buttons;

   public LookAndFeelSelection() {
      this.setLayout(new GridLayout(lookAndFeelsAvailable.length, 1));
      buttons = new JRadioButton[lookAndFeelsAvailable.length];
      ButtonGroup group = new ButtonGroup();
      for (int i = 0; i < lookAndFeelsAvailable.length; i++) {
         this.buttons[i] = new JRadioButton(lookAndFeelsAvailable[i]
                           .getName());
         if (lookAndFeelsAvailable[i].getName().equals(
               UIManager.getLookAndFeel().getName()))
             this.buttons[i].setSelected(true);
         group.add(buttons[i]);
      this.add(buttons[i]);
      }
   }
}
```

Den Dialog, der mittels des Menüpunkts Optionen→ aufgerufen wird nenne ich *OptionsDialog*. Nach den vorstehenden Überlegungen sieht er zur Zeit so aus:

```
public class OptionsDialog extends JDialog {
Option [] options;
public OptionsDialog(Schritt04 frame, boolean modal){
this.options = frame.getOptions();
this.setTitle("Einstellungen");
Container content = this.getContentPane();
content.setLayout(new BorderLayout());
JTabbedPane optionen = new JTabbedPane();
JPanel lookAndFeel = new LookAndFeelSelection();
optionen.addTab("Look & Feel", lookAndFeel);
content.add(optionen, BorderLayout.NORTH);
JPanel buttons = new JPanel();
content.add(buttons, BorderLayout.SOUTH);
buttons.setLayout(new FlowLayout(FlowLayout.LEFT ));
JButton ok = new JButton("Ok");
ok.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent e){
setOptions();
setVisible(false);
}
});
JButton apply = new JButton("Anwenden");
apply.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent e){
setOptions();
```

```
}
});
JButton cancel = new JButton("Abbrechen");
cancel.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent e){
setVisible(false);
}
});
buttons.add(ok);
buttons.add(apply);
buttons.add(cancel);
this.pack();
}

}
```

Nun muss ich noch dafür sorgen, dass die Wahl des Benutzers auch entsprechend des gedrückten Buttons *Ok*,*Anwenden*, bzw. *Abbrechen* umgesetzt wird bzw. nicht. Dazu verwende ich ein Interface *Setting*, dass alle Options Karteikarten implementieren müssen, wie etwa *LookAndFeelSelection*.

Nachdem wir nun Menüauswahlen auch durch sinnvolle Reaktionen beantworten können, will ich diese ersten Demonstrationsobjekte nun zu einer kleinen Anwendung ausbauen. Es sollen Objekte ausgewählter Klassen angezeigt werden können.

Da wir Objekte bisher nur gemäß der Schnittstelle Serializable speichern bzw. laden können, beginne ich damit einfach eine solche Datei anzuzeigen. Mittels der bereits rudimentär implementierten Menüauswahl „Datei→Öffnen“ soll also eine solche Datei geöffnet und in einem Fenster innerhalb des Frames angezeigt werden.

Dies soll in Karteikarten, wie in Eclipse geschehen. Grundsätzlich habe ich bereits beim Einstellungsdialog gezeigt wie das geht:

- Wird eine Datei geöffnet, so wird eine neue Karteikarte eingetragen.
- Je nach Art der Darstellung brauchen wir eine andere Karteikarte, z. B.
  - Textanzeige in ASCII, Unicode oder Hex
  - Darstellung eines Objekts in Form seiner Attribute
  - Darstellung einer Liste von Objekten mit ihren Attributen
  - Darstellung von Objekten in 1:*-Assoziationen in Baumdarstellung
  - Darstellung eines oder mehrerer Objekte mit den verfügbaren Operationen, geeignet für einen Anwender aufbereitet; dazu brauchen wir RTTI/Reflection. Diesem Thema wende ich mich später zu.

29.5 Layout Manager

Die Positionierung der Elemente eines Swing-Containers wird durch LayoutManager gesteuert. Mit diesen haben Sie es bei *JPanel* und bei ContentPanes zu tun. Ein *JFrame* hat eine *JRootPane*. Diese verwaltet u. a. die ContentPane des Frames und einen etwaigen *JMenuBar*. Dazu an geeigneter Stelle mehr.

Ein LayoutManager bestimmt die Anordnung von Elementen innerhalb eines *JPanel* oder einer ContentPane. Hier einige wichtige:

BorderLayout Ein Borderlayout hat fünf Bereiche: *CENTER*, *NORTH*, *SOUTH*, *WEST*, *EAST*, deren Namen ihrer Position entsprechen. Abbildung 29.2 zeigt das Schema. Dieser Layoutmanager ist in vielen Fällen gut für die oberste Komponente, z. B. einen *JPanel* geeignet.

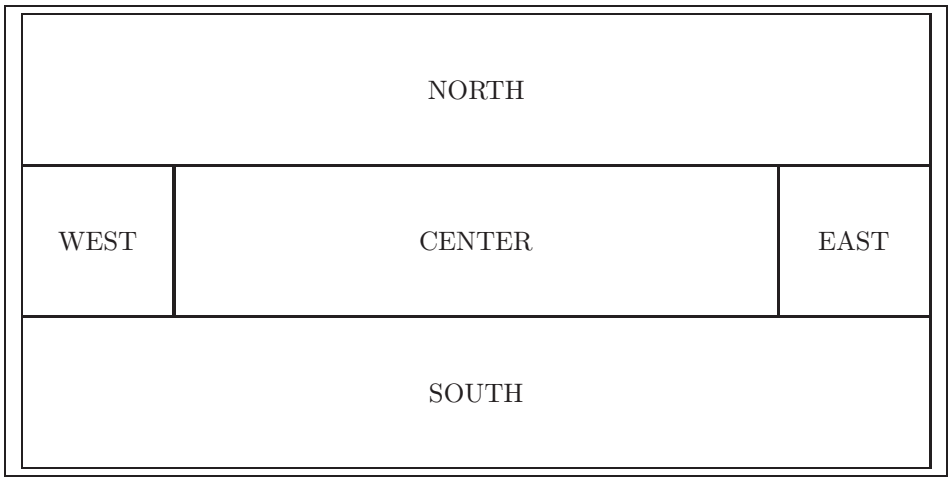


Abb. 29.2: BorderLayout

Die Methode add erhält die entsprechende Position als Parameter. Die hier verwendeten Konstanten unterstellen eine westliche Orientierung, also u. a. lesen von links nach rechts. Unahbhängig davon sind die folgenden Konstanten (In Klammern die entsprechende oben verwendete): PAGE_START (NORTH), PAGE_END (SOUTH), LINE_START (WEST), LINE_END (EAST). CENTER ist unverändert.

FlowLayout Ein Layoutmanager, der die Komponenten von links nach rechts anordnet und beim Zeilenende umbricht. Das Schema zeigt Abb. 29.3. Ist die Größe des Fensters veränderbar,

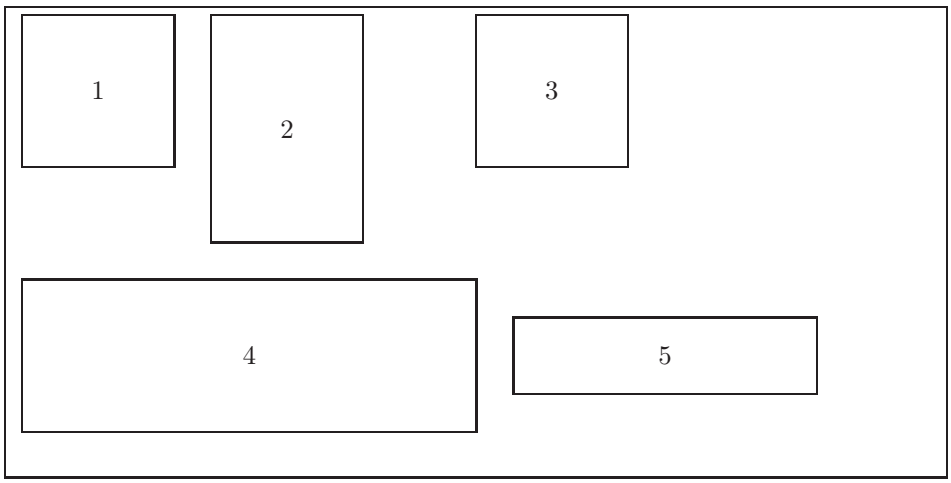


Abb. 29.3: FlowLayout

so ändert sich mit einem FlowLayout ggf. auch die Anordnung: Die Zeilenumbrüche hängen von der Länge der Zeilen ab. Von daher eignet sich ein FLowLayout keineswegs immer.

BoxLayout Ein Layoutmanager, bei dem Sie entscheiden können ob, die Komponenten von links nach rechts oder von oben nach unten angeordent werden sollen. Im Konstruktor geben Sie dafür die Achse an (senkrecht oder waagerecht). Bei senkrechter Anordnung kann das Ergebnis aussehen wie in Abb. 29.4. BoxLayout ist allgemein und kann für eine Anordnung von Komponenten verwendet werden, für die andere LayoutManager zu starr sind.

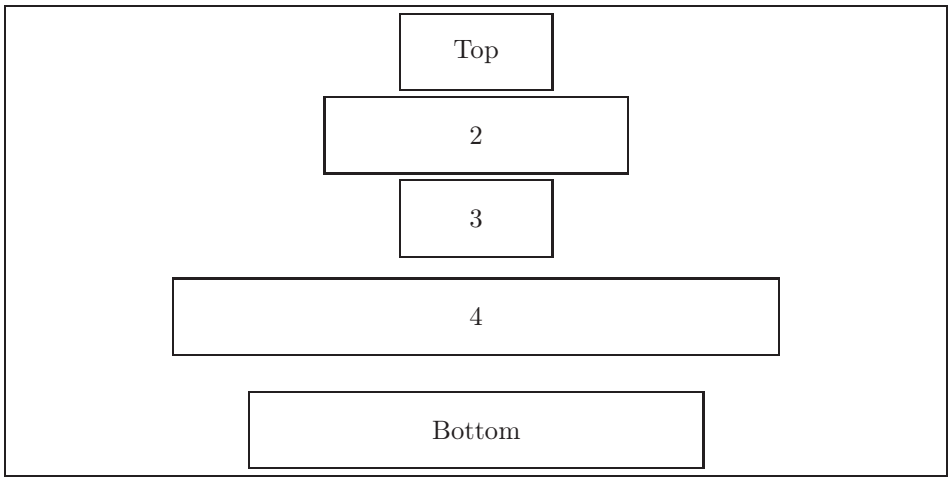


Abb. 29.4: BoxLayout

GridLayout Ordnet die Komponenten in Zeilen und Spalten an. Im Konstruktor werden die Anzahl Zeilen und Spalten und die Abstände zwischen den Zeilen und Spalten angegeben. Abbildung 29.5 zeigt das Schema. Für gleichgroße Komonenten ist das prima. Brauchen Sie

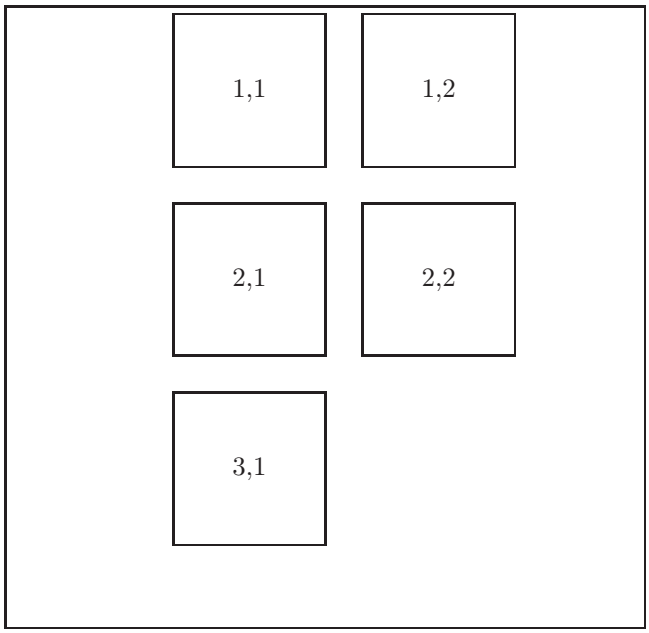


Abb. 29.5: GridLayout

mehr Flexibilität, so verwenden Sie eine Hierarchie von verschiedenen *JPanel*s mit geeigneten Layout pro Panel oder ein GridBagLayout, wenn Ersteres nicht sinnvoll erscheint.

GridBagLayout Ein flexibler Layoutmanager, bei dem die einzelnen Elemente unterschiedlich groß sein können, wie etwa der Button für 0 auf vielen numerischen Tastaturblöcken größer als die anderen Ziffern ist. Das Schema zeigt Abb. 29.6. Sowohl GridLayout als auch GridBagLayout kommen oft in geeigneten Teilen eines Fensters zum Einsatz.

LayoutManager werden Sie nicht immer direkt verwenden wollen. Einfacher ist es oft, einen GUI-

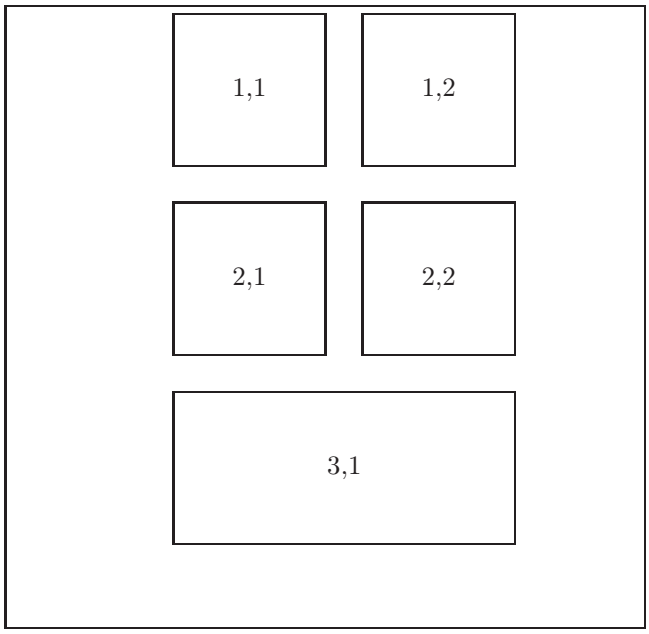


Abb. 29.6: GridBaglayout

Builder zu verwenden. Für Eclipse gibt es plugins für diesen Zweck. Die Entwicklungsumgebung NetBeans besitzt einen leistungsfähigen GUI-Builder.

29.6 Swing Components

In diesem Abschnitt behandle ich die einzelnen Swing-Komponenten systematisch. Dies Darstellung wird nie vollständig sein. Eine erste Übersicht gibt Abb. 29.7 Weitgehend isolierte Beispiele für einige der Komponenten finden Sie im Verzeichnis swing.components. Dabei mache ich Gebrauch von einigen Hilfsklassen, die im Paket *swing.helper* zusammengefasst sind.

Für allgemeine Informationen zu den verwendeten Mustern, insbesondere dem Beobachtermuster (observer pattern) verweise ich auf Kap. 23.

29.6.1 JLabel

Die Klasse *JLabel* Mittels der Methode *setBackground* wird die Farbe des Hintergrunds gesetzt. Diese ist nur sichtbar, wenn das *JLabel opaque*, d. h. durchsichtig, ist. Ob dies der Fall ist bestimmen Sie mittels *setOpaque*. Die Methode *setForeground* bestimmt die Farbe des Vordergrunds. Diese Methoden stammen aus der Oberklasse *JComponent*. Bei Bedarf werden sie in den konkreten Komponenten überschrieben. Für ein *JLabel* wird so z. B. die Farbe des Text gesetzt.

29.6.2 Font

Um Fonts systematisch zu nutzen, hilft es eine Begriffe aus dem Schriftsatz zu kennen.

Font Ein Font ist eine Abbildung von Zeichen eines Zeichensatzes auf Glyphen, die Zeichen darstellen. Dieser Text ist im Font Computer Modern gesetzt. in diesem Font enspricht der Buchstabe `a` dem Zeichen „a“ und die Buchstabenfolge `fi` dem Glyph „fi“, einer Ligatur.

Schriftfamilie Eine Gruppe von Fonts, z. B. Monospaced (jeder Buchstabe hat die gleiche Breite, wie Courier) , Font mit Serifen (Times New Roman) oder ohne Serifen (Helvetica) usw.

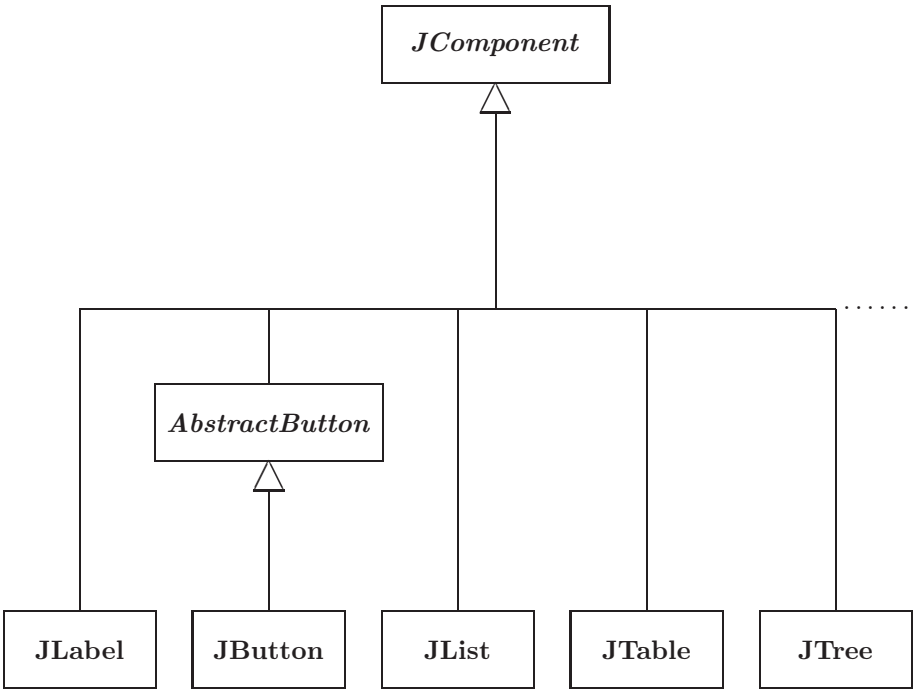


Abb. 29.7: Swing-Komponenten

Schriftgrad Größe der Buchstaben. Diese wird oft in Punkt (pt) angeben. $1pt = (1/72, 27)Zoll$

Schriftschnitt Attribute von Schriften, wie geneigt (slanted) etc., die aus einer senkrechten Basisversion systematisch abgeleitet werden.

Dementsprechend gibt es Konstruktoren, die einen Font mit default Schriftgrad und Schriftschnitt. Ein Array mit den im jeweiligen System verfügbaren Fonts erhalten Sie mit der Methode *getAllFonts* der AWT-Klasse *GraphicsEnvironment*.

29.6.3 JButton

Ein Objekt der Klasse *JButton* ist ein push-Button. Da es ein Container ist, kann ein *JButton* verschiedene Komponenten enthalten. Für *JButton* snd dies Text (*String*) und Bild (*Icon*) Ein Button kann verschiedene Listener haben, am wichtigsten ist wohl eine Implementierung des Interfaces *ActionListener*, siehe Abb. 29.8.

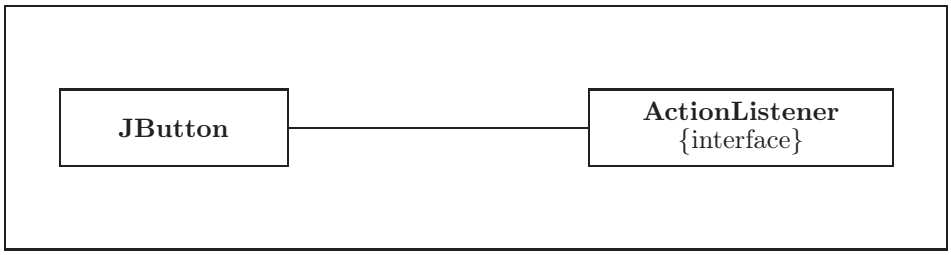


Abb. 29.8: JButton

29.6.4 JList

Die Klasse *JList<E>* nutzt ein Standard-Muster, das auch in Swing durchgängig angewandt wird: Das Beobachtermuster (observer pattern, model - view - controler). Abbildung 29.9 zeigt die beim

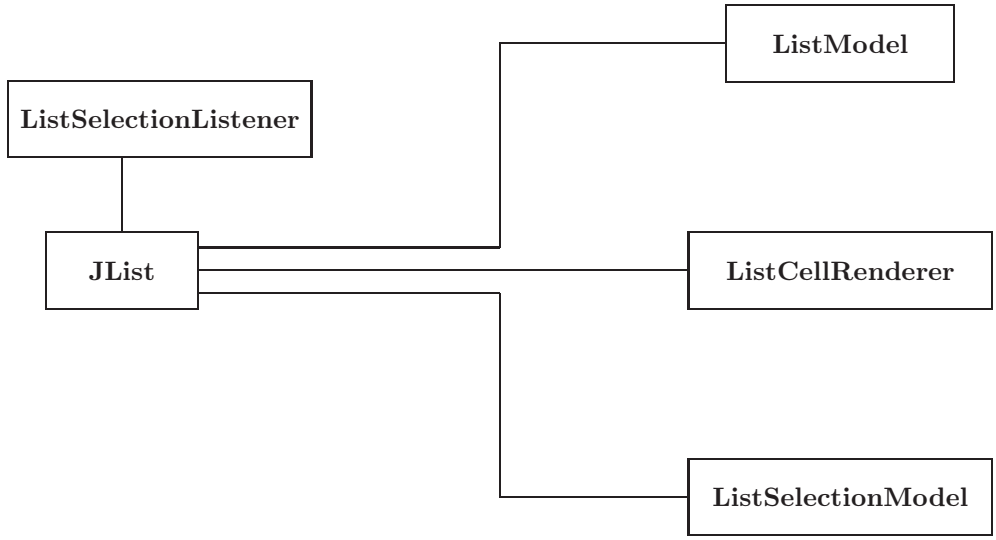


Abb. 29.9: JList, ListModel, ListSelectionModel, CellRenderer und ListSelectionListener

Beobachtermuster hier beteiligten Klassen schematisch. In dieser Abb. sind bis auf *JList* die Namen in allen Klassensymbolen Namen von *Interfaces*. Im folgenden Beispiel habe ich nur das *Interface ListCellRenderer* durch eine nicht-*anonyme Klasse* implementiert, alle anderen durch *anonyme Klassen*.

Um eine Reihe von Einträgen in der *JList* zu haben, wähle ich die in meinem System verfügbaren *Fonts*. Eine einfache Liste ist aber noch nicht besonders praktisch. So werden Sie in einer längeren Liste blättern wollen. Für diesen Zweck gibt es die Klasse *JScrollPane*. Diese stellt bei Bedarf automatisch einen horizontalen oder vertikalen ScrollBar zur Verfügung. Im Konstruktor wird einfach das Element übergeben, dass mit diesne ScrollBars dekoriert werden soll (s. a. *decorator pattern*. In diesem Fall also:

```
JList<Font> jList = new JList<>(GraphicsEnvironment.
                                getLocalGraphicsEnvironment().
                                getAllFonts());
JScrollPane jsp = new JScrollPane(jList);
```

Mittels einer einfachen Hilfklasse zeige ich die jeweilige Komponente dann an:

```
ShowInFrame.show("Scrollable List aller Fonts im jeweiligen Font", jsp);
```

Sie finden diese Klasse und einige weitere im Paket *gui.swing.helper*.

Eine *JList* hat immer ein *Model*. Wenn Sie nichts Weiteres tun, ist die ein Objekt der Klasse *DefaultListModel*. Um mit der Auswahl von Listenelementen umzugehen brauche ich einen *ListSelectionListener*. Zusammen mit dem *ListSelectionModel* bildet dieser den *Controler*-Teil.

Der Code sollte bis auf einen kleinen Teil beinahe selbsterklärend sein:

```
100 jList.addListSelectionListener(new ListSelectionListener() {
110    @Override
120    public void valueChanged(ListSelectionEvent lse) {
130       if (!lse.getValueIsAdjusting()) {
140          System.out.println("You selected Font: " + jList.getSelectedValue());
150       }
160    }
180 });
```

Das *Interface ListSelectionListener* deklariert nur eine *Operation*: *valueChanged*. Ändert sich die Auswahl, so ruft der *ListSelectionListener* diese Methode auf. Dies passiert beim z. B. Drücken und beim Loslassen der rechten Maustaste. Ohne die *if*-Bedingung in Zeile 130 würde in obigem Code die Ausgabe des selektierten Codes die Ausgabe auf der Konsole also zweimal erfolgen.

Hier kommt nun die Methode *getValueIsAdjusting* von *ListSelectionEvent* zum Einsatz: Wenn ein *ListSelectionEvent* zu einer zsammengehörigen Folge von Ereignissen gehört, liefert diese Methode *true*. Dies ist beium loslassen der rechten Maustaste der Fall, also erfolgt die Ausgabe nicht noch ein zweites Mal, wenn die Maustaste losgelassen wird.

Das *ListModel* bildet hier den *Model*-Teil des *MVC*-Musters. Zum *View* gehört bisher die JList. Nun füge ich noch ein weiteres *View*-Element hinzu: Ich möchte die Beschreibung des jeweiligen *Fonts* in diesem *Font* gesetzt sehen, um besser auswählen zu können. Hier der Code dazu:

```
100 public class FontListCellRenderer implements ListCellRenderer<Font>{
110    protected ListCellRenderer<? super Font> defaultRenderer;
120    public FontListCellRenderer(ListCellRenderer<? super Font> listCellRenderer){
130        this.defaultRenderer = listCellRenderer;
140    }
150    @Override
160    public Component getListCellRendererComponent(
170            JList<? extends Font> list, Font value, int index,
180            boolean isSelected, boolean cellHasFocus) {
190        JLabel cell = (JLabel) defaultRenderer.
200                      getListCellRendererComponent(list, value, index,
210                                                   isSelected, cellHasFocus);
220        cell.setFont(new Font(list.getModel().getElementAt(index).getName(),
230                              Font.PLAIN,24));
240        return cell;
250    }
260 }
```

Der *FontListCellRenderer* soll das weiter aufbereiten, was eventuell ein anderer *ListCellRenderer* bereits geleistet hat. Deshalb wird im Konstruktor ein *ListCellRenderer* erwartet und in einem *Attribut* gespeichert. Jeder Eintrag in einer *JList* ist ein *JLabel*, daher die lokale Variable *cell* in Zeile 190. Hier wird einfach die *getListCellRendererComponent* des ursprünglichen *ListCellRenderers* aufgerufen. Der jeweilige *Font* wird in Zeilen 220–230 aud em zugehörigen *ListModel* besorgt.

Nach Abb. 29.9 gehört zu einer *JList* ein *ListSelectionModel*. Sie können selber eines schreiben. Für viele Fälle reichen aber die *ListSelectionModel* aus, die mit *Swing* ausgeliefert werden. So gibt es die Klasse *DefaultListSelectionModel*. Im *Interface ListSelectionModel* sind drei Selektionsmodi vordefiniert, deren Namen selbsterklärend ist:

SINGLE_SELECTION Nur ein Listenelement zur Zeit kann ausgewählt werden.

SINGLE_INTERVAL_SELECTION Ein zusammenhängedes Intervall von Listenelementen kann ausgewählt werden. Das kann unter Windows mit den üblichen Mitteln erreicht werden: Anklicken des ersten Elements und mit gedrückter Umschalttaste das nächste Element

anklicken. Dann wird das durch diese beiden Elemente begrenzte Intervall einschließlich dieser beiden ausgewählt. Über die Tastatur geht das ausgehend von einem ELement durch betätigen der Pfeil-auf- bzw. Pfeil-ab-Taste bei gleichzeitig gesrückter Umschalttaste.

MULTIPLE_INTERVAL_SELECTION Mehrere zusammenhängende Intervalle können ausgewählt werden. Auch das geht unter windows gemäß Windowsstandard: Steuerungstaste gedrückt halten und nach und nach die Elemente anklicken.

Hier Beispielcode für die letztgenannte Variante:

```
100     final JList<Font> jList = new JList<>(GraphicsEnvironment
110             .getLocalGraphicsEnvironment().getAllFonts());
120     jList.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
130     jList.setCellRenderer(new FontListCellRenderer(jList.getCellRenderer()));
140     jList.addListSelectionListener(new ListSelectionListener() {
150         @Override
160         public void valueChanged(ListSelectionEvent lse) {
170             if (!lse.getValueIsAdjusting()) {
180                 System.out.println("You selected Fonts: ");
190                 for (Font f : jList.getSelectedValuesList()) {
200                     System.out.println("-" + f.getName());
210                 }
220             }
230         }
240     });
250     JScrollPane jsp = new JScrollPane(jList);
260     ShowInFrame.show("Scrollable List aller Fonts mit multiple Selection", jsp);
270 }
```

Bemerkung 29.6.1 (Anonyme Klasse und final)
Im Beispiel *List06* u. a. (s. o.) müssen die Variablen für die *JList* (oben in Zeile 100) und den *ListCellRenderer* als final deklariert werden. Nur dadurch kann sichergestellt werden, dass das Objekt der anonymen Klassen sie nicht verändern kann. Andernfalls könnten sie dort z. B. auf *null* gesetzt werden. Im Beispiel *List05* war dies nicht erforderlich, da statische innere Klassen nicht auf die Elemente der äußeren Klasse zugreifen können. ◄

Um zu demonstrieren, dass die Mehrfach-Intervall-Selektion funktioniert, gebe ich in Zeilen 190–210 alle gerade ausgewählten Elemente (hier Fonts) aus.

29.6.5 JTable

Eine einfache Möglichkeit, Tabellen mit einheitlichen Spalten darzustellen. Die Klasse *JTable* implementiert die wesentlichen Listener, die für die Arbeit mit Tabellen benötigt werden.

29.6.6 JTree

Die meisten der Dinge, die ich hier vermitteln möchte, kann ich an der Baumdarstellung erläutern. Deshalb arbeite ich diese als erste aus. Um eine flexible Struktur innerhalb dieser Anwendung zu schaffen, will ich dafür sorgen, dass verschiedene Fenster darin nach den Vorgaben des Anwenders angeordnet werden können. Die hierfür geeignete Klasse ist *JDesktopPane*. Eine *JDesktopPane* enthält Objekte der Klasse *JInnerFrame*

Für die Baumdarstellung verwende ich die Klasse *JTree*. Diese Klasse ist ein gutes Beispiel für eine Struktur, die InformatikerInnen immer wieder begegnet.

De Klasse *JTree* kommt mit einem *DefaultTreeModel* aus dem Paket *javax.swing.tree*. Die Klasse *DefaultTreeModel* verwaltet die Knoten eines Baumes (tree). Dies sind Objekte deren Klassen

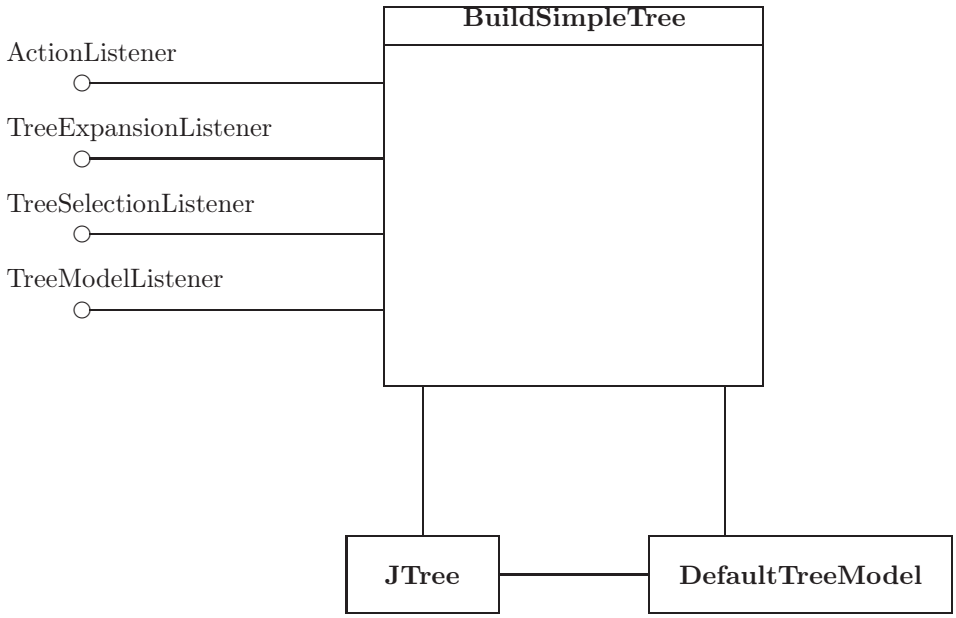


Abb. 29.10: Klassenmodell für JTree-Verwendung (noch unvollständig)

das Interface *TreeNode* implementieren. Für einfache Zwecke genügt die Klasse *DefaultMutableTreeNode*. Da diese Klasse u. a. die Operationen *getUserObject* und *setUserObject* besitzt kann vieles darüber abgehandelt werden. Um auf die Aktionen des Benutzers reagieren zu können werden verschiedene Listener eingesetzt werden, insbesondere *TreeExpansionListener*, *TreeSelectionListener*, *TreeModelListener*. Wird etwa ein weiterer Knoten in einen Baum eingefügt, so wird die Methode *treeNodesInserted(TreeModelEvent e)* aufgerufen. Um den oder die neuen Knopten anzuzeigen, verwenden Sie *expandPath(e.getTreePath())*. Ein einfaches Beispiel, in dem Knoten eingefügt, allerdings nicht wieder gelöscht werden können, finden Sie in *swing.BuildSimpleTree*. Diese Klasse implementiert auch alle benötigten Listener-Interfaces. Die Abb. 29.10 zeigt das Modell von *BuildSimpleTree*. Hieran sind einige Dinge diskussionsbedürftig:

1. Die Klasse *BuildSimpleTree* implementiert alle Listener selbst. Dies dient nur dazu,dass Modell klein und kompakt zu halten. Für komplexere Dinge werden Sie diese herausfaktorisieren.

2. Es sind redundante Assoziationen vorhanden. Es bedüfte nur der Assoziation von *BuildSimpleTree* zu *JTree*, denn *JTree* hat eine Methode *getModel()*. Diese liefert ein Objekt vom Typ *TreeModel*, einem Interface. Also müssten Sie nach *DefaultTreeModel* casten. Durch die Definition der entsprechenden Attribute wird der Code kürzer und m. E. leichter lesbar.

3. Das Grundschema *Komponente—Modell* finden Sie in Swing durchgehend. Das dahinter liegende Muster wird im Kap. 23 beschrieben. Außerdem verweise ich auf die Veranstaltungen zum Softwareengineering.

29.7 Java FX

29.7.1 Übersicht

JavaFX ist die aktuelle Bibliothek zum Schreiben von GUIs in Java. Ich habe vieles aus [Sha15] und [VGC+14] gelernt.

29.7.2 Lernziele

29.7.3 Einführung

Java FX Anwendungen können aus mehreren Arten von Java-Klassen und Dateien bestehen:

1. Einer Startklasse. In dieser oder einer weiteren Klasse kann auch die Oberfläche mit Java Mitteln definiert werden

2. Einer Controller Klasse. Diese fungiert dann als Controller im Sinne des MVC-Patterns.

3. Einer .css Datei (Cascading Style Sheet). Diese definiert das allgemeine Aussehen der Anwendung, z. B. Schrifttype etc.

4. Einer .fxml Datei. Für diese gibt es den Scenebuilder

5. Einer .fxgraph Datei bei Verwendung von e(fx)clipse. In dieser Datei wird in JSON ähnlicher Syntax die Oberfläche beschrieben, aus der dann eine zugehörige .fxml Datei erzeugt wird.

Parameter können einfache an eine JavaFX Anwendung übergeben werden. Dabei können sie benannte und unbenannte Parameter übergeben. Der Name eines benannten Parameters muss mit `--` beginnen:

```
Bernd --width=200
```

Einem benannten Parameter wird wie eben gezeigt ein Wert zugewiesen. Die Klasse *Application* hat eine Methode *getParameters*, die ein Objekt der statischen inneren Klasse *Application.Parameters* liefert. Beide Parameter-Arten zusammen heißen raw-Parameter.

Eine JavaFX Application muss einen default-Kontruktor (also einen ohne Parameter) haben, Sonst gibt es beim Launch eine runtime-Exception.

29.7.4 Knotenstruktur in JavaFX

Der oberste (top-level) Container ist die *Stage*. Die primary Stage wird von der Plattform erzeugt. Die Anwendung kann weitere Stage-Objekte erzeugen.

29.7.5 Observable

Die Basis für das Beobachter-Muster (observer pattern) ist das Interface *Observable* aus dem Paket javafx.beans.

Bemerkung 29.7.1
Verwechseln Sie das Interface *javafx.beans.Observable* bitte nicht mit der Klasse *java.util.Observable*! Letzte wird ebenso wie das Interface *java.util.Observer* in Java 9 deprecated werden. ◄

Weak Listener (*WeakInvalidationListener*, *WeakChangeListener*) werden vom Garbage Collector entfernt, Strong Listener nicht. Verwendung ersterer reduziert die Gefahr von Memory Leaks.

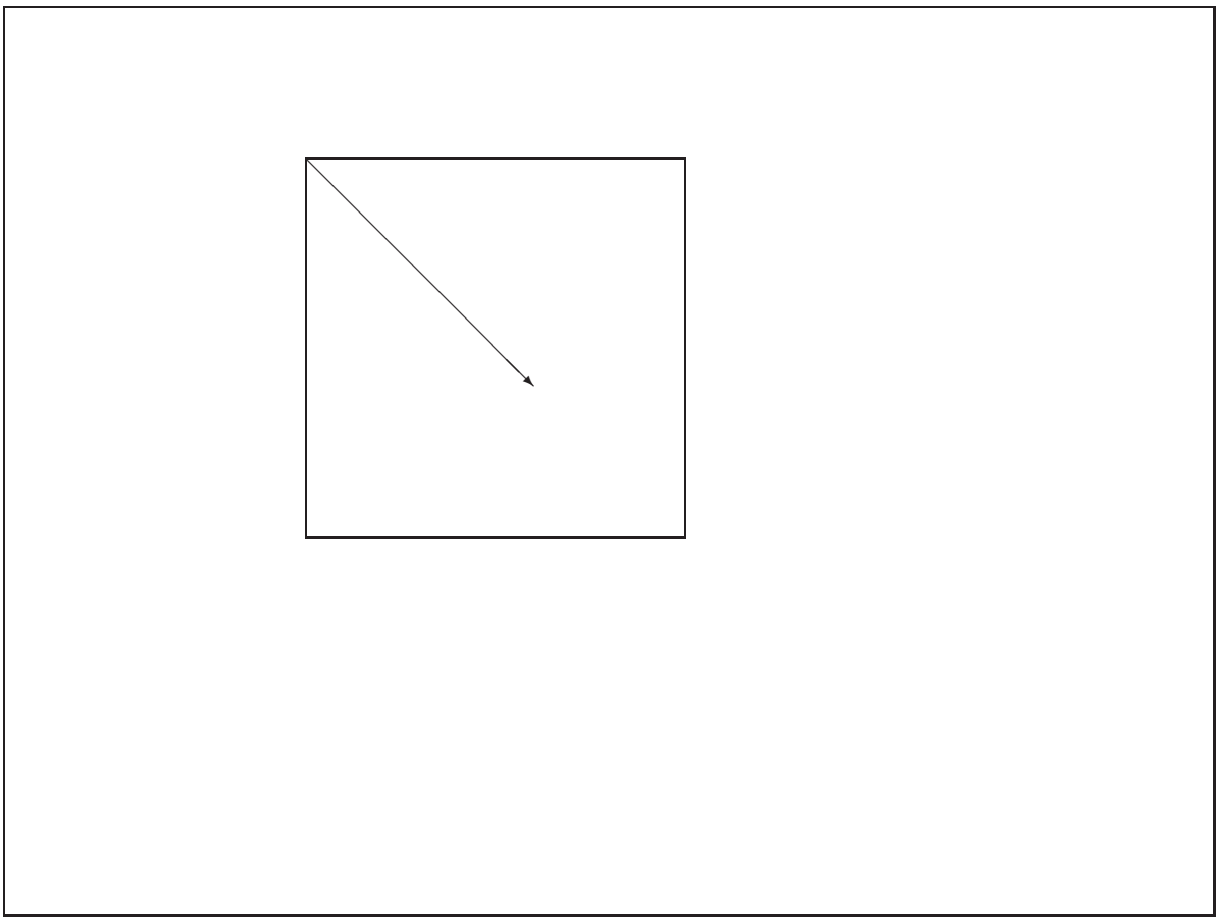


Abb. 29.11: Auswahl eines Rechtecks auf Bildschirm

29.8 Auf Ereignisse reagieren

Einige Arten von sog. „Listenern“ oder „Handlern“ stelle ich Ihnen an einem Beispiel vor: Auswahl eines Rechtecks auf einem Bildschirm. Das Schema zeigt Abb. 29.11 Die Aufgabe besteht darin links oben zu beginnen und eine Rechteck zu markieren, auf dem dann weitere Aktionen (z. B. Zoom) aufgebaut werden können.

- Eine Ecke muss markiert werden. In Abb. 29.11 ist das links oben. Für den Fall, dass Benutzer andere Ecken als Anfang wählen, müssne wir uns auch noch etwas überlegen. In jedem Fall müssen wir uns die Position merken.
- Eine Ecke kann durch den Klick mit einer Maustaste ausgewählt werden. Primäre Maustaste könnte auswählen, sekundäre auswählen und ein Popup-Menu mit weiteren Optionen anbieten.
- Wenn die Maustaste wieder losgelassen wird, müssem wir uns den anderen Eckpunkt des Rechtecks merken.
- Wenn die Maus mit gedrückter Taste zur anderen Ecke gezogen, müssen wir verhindern, das die ursprünglich ausgewählte Ecke „vergessen“ wird.

Diese Überlegungen führen auf folgende Methoden:

29.8.1 Diverses

Analog zum ternären Operator gibt es in JavaFX die Klasse *When*. Sie ermöglicht einen Ausdruck wie:

```
new When(bedingung).then(wert1).otherweise(wert2);
```

Dies dient als „Ersatz“ für den ternären Operator für *ObservableBooleanExpression*s. Sie ähnelt der Klasse *Optional*. Das Ergebnis kann z. B. einem *StringBinding* zugewiesen werden.

29.8.2 Observable Collections

JavaFX provides *ObservableList<E>*, *ObservableSet<E>*, *ObservableMap<K,V>* und *FilteredList<E>*, *SortedList<E>*. Erzeugt werden sie durch Klassenmethoden der Utility-Klasse *FXCollections*.

29.8.3 JavaFX CSS

Das Aussehen („Look& Feel“) von Java FX Anwendungen kann durch Style Sheets (.css-Dateien) gestaltet werden. Das entspricht dem üblichen Vorgehen in Textsatzsystemen, bei denen der Text von der Formatierung getrennt wird, wie z. B. in MS-Word durch eine Dokumentvorlage, in LaTeXeine Documentclass und in HTML ein Cascading Style Sheet.

Der default Stil ist im *Modena* Stylesheet mondena.css definiert. Sie setzen es mit der Klassenmethode *setUserAgentStylesheet(String url)*. Ein weiterer Stil ist *Caspian*. Sie finden beide in der Java FX Bibliothek jfxrt.jar in com.sun.javafx/scene/control/skin/. Setzen sie es auf *null*, so gilt der default Stil *Modena*.

Die einzelnen Möglichkeiten den Stil zu setzen haben die folgenden Prioritäten (Höchste zu erst):

1. Inline style
2. Parent style sheets
3. Scene style sheets

4. Werte, die durch das JavaFX API gesetzt werden
5. User Agent Style sheets

JavaFX CSS unterstützen folgende Typen von Werten:

- inherit
- boolean
- string
- number
- angle
- point
- color-stop
- URI
- effect
- font
- paint

29.8.4 fxml und Scenebuilder

Der Universal Selector ist „*“. Er wählt alle Knoten aus:

```
* {
     -fx-text-fill: blue;
}
```

Descendend Selectors bestehen aus zwei oder mehr durch Blanks getrennten Selektoren:

```
.hbox .button {
        -fx-text-fill: green;
}
```

betrifft z. B. alle Knoten vom Stil button und Abkömmlinge eines Knoten vom Stil hbox sind.
State-based Selectors betreffen Knoten in Abhängigkeit von ihrem Zustand:

```
.button:focus {
        -fx-text-fill: red;
}
```

Hier eine noch nicht vollständige Liste dieser „Pseudo Klassen“:

Pseudo Klasse	Anwendbar auf	Beschreibung
disabled	Node	It applies when the node is disabled.
focused	Node	It applies when the node has the focus.
hover	Node	It applies when the mouse hovers over the node.
pressed	Node	It applies when the mouse button is clicked over the node.
show-mnemonic	Node	It applies when the mnemonic should be shown.
cancel	Button	It applies when the Button would receive VK_ESC if the event is not consumed.
default	Button	It applies when the Button would receive VK_ENTER if the event is not consumed.
empty	Cell	It applies when the Cell is empty.
filled	Cell	It applies when the Cell is not empty.
selected	Cell, CheckBox	It applies when the node is selected.
determinate	CheckBox	It applies when the CheckBox is in a determinate state.
indeterminate	CheckBox	It applies when the CheckBox is in an indeterminate state.
visited	Hyperlink	It applies when the Hyperlink has been visited.
horizontal	ListView	It applies when the node is horizontal.
vertical	ListView	It applies when the node is vertical.

Mittels *javapacker* kann eine .css-Datei in ein binäres Format umgewandelt werden. Für Einzelheiten rufen Sie javapacker -help auf.

29.8.5 Erste Schritte mit Scene Builder

Ich beginne mit einem ganz einfachen Beispiel, das wichtige Prinzipien illustriert. Basis ist ein leeres Fenster. Sie finden den Code hierfür in de.kahlbrandt.javafx.controls.MenuExample00.java und MenuExample00.fxml im gleichen Paket. Hier der Code zwecks Erläuterung.

```
public class MenuExample01 extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {
        Parent root = FXMLLoader.load(getClass().getResource("MenuExample01.fxml"));
        Scene scene = new Scene(root, 600, 550);
        primaryStage.setTitle("Menü Beispiel");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

1. JavaFX Anwendungsklassen müssen die Klasse *Application* aus dem paket javafx.application spezialisieren.

2. Diese Klasse hat eine abstrakte Methode: start. Diese muss also überschrieben werden. Die start-Methode wird aus der launch-methode aufgerufen.

3. Basis ist hier das Objekt root der Klasse Parent. Dies ist der Basis *Node* für alle *Node*s einer *Scene*, die Children haben. In diesem Beispiel wird dieses Objekt aus der fxml-Datei MenuExample01.fxml durch FXMLLoader.load erstellt.

4. Dieses *Parent* Objekt root ist die Basis für die Scene dieses kleinen Beispiels.

5. Das Objekt scene der Klasse *Scene* ist die Wurzel des Scene-Graph. Es wird unter der Stage „eingehängt“ und hier mit einer initialen Größe erzeugt. Die umfassende Stage kann weitere Elemente (z. B. Dekorationen) enthalten, deren Größe vom GUI-System (Betriebssystem, Windows, Motif etc.) abhängt. Es ist deshalb sinnvoll, die Größe der Scene zu planen. Die Stage-Größe ergibt sich dann automatisch.

6. Die Methode *launch* erhält ein *Stage* übergeben. Dies ist der Top-Level Container, in den alle untergeordneten Objekt kommen.

7. Den Titel setzen Sie mit setTitle, die Scene wird mittels setScene der Stage hinzugefügt.

8. Als letztes mache ich das Fenster mittels primaryStage.show() sichtbar.

Die Basisstruktur ist in der folgenden fxml-Datei MenuExample01.fxml definiert:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.layout.*?>
<?import java.lang.*?>
<?import javafx.scene.control.*?>
<BorderPane maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity" minWidth="-Infinity"
            xmlns="http://javafx.com/javafx/8" xmlns:fx="http://javafx.com/fxml/1">
   <top>
      <Pane prefHeight="200.0" prefWidth="200.0" BorderPane.alignment="CENTER" />
   </top>
</BorderPane>
```

1. Die erste Zeile ist der optionale xml-Prolog. `encoding = "UTF-8"` ist der default und wird als Zeichensatz empfohlen. Alle Verarbeitungsnweisungen haben die Form `<?tag ... ?>`

2. Der xml-Prolog ist eine Verarbeitungsanweisung. In den nächsten drei Zeilen folgen weitere Verarbeitungsanweisungen. In diesem Fall für JavaFX, um die benötigten import-Statements zu generieren.

3. Danach folgt der einzige für das Verständnis wichtige Teil: Innerhalb eines Containers (Stage, Scene) können Sie weitere Elemente platzieren. Um die Platzierung zu erleichtern, gibt es Unterstützung in der Form von Panes. Auf der obersten Ebene ist oft eine *BorderPane* sinnvoll: Norden, Süden, Osten, Westen, Zentrum. In einer *BorderPane* können Sie gezielt weitere Container oder elementare Elemente einfügen.

Die Datei habe ich mitt dem JavaFX Scene Builder erzeugt. Dessen Bedienung eignen Sie sich am Besten durch Üben mit dem System an.

Als erste Beispiel füge ich diesem Fenster nun ein typisches Menü hinzu und die zugehörigen Aktionen. Als erstes füge ich im Top Teil der BorderPane einen *MenuBar* ein. Als Default enthält dieser die üblichen Windows Menus File, Edit und Help. Die haben schon die *Standard MenuItem*s Close bzw. About. Diese „tun“ aber noch nichts. Das wollen wir jetzt ergänzen.

29.8.6 Event handling

- Event source
- Event target
- Event Type

Name	Art	Beschreibung
Event	Class	An instance of this class represents an event. Several subclasses of the Event class exist to represent specific types of events.
EventTarget	Interface	An instance of this interface represents an event target.
EventType	Class	An instance of this class represents an event type, for example, mouse pressed, mouse released, mouse moved.
EventHandler	Interface	An instance of this interface represents an event handler or an event filter. Its handle() method is called when the event for which it has been registered occurs.

29.9 Layouts

Eine Region kann einen Rand haben (Border). Borders bestehen aus Strokes, Bildern oder beiden. Ein Stroke hat eine Farbe (Color), einen Stil (Style), eine Breite (Width), Radien für die vier Ecken, Insets für die vier Seiten.

29.9.1 Historische Anmerkungen

Ist das jetzt der nächste (vielleicht letzte) Versuch, Java im Client-Bereich zu etablieren.

29.9.2 Aufgaben

29.10 Historische Anmerkungen

Nach [Zak] ist der name „Swing“ so entstanden:

> The story is: The team went to Hobees for lunch, and the topic turned to what to name the new toolkit we were writing. Up till then the name was code named KFC, which was chosen by our manager (Rick Levenson) as a way to ensure we'd come with with a better name before shipping; he knew there was no way „KFC“, aka Kentucky Fried Chicken, would be allowed by the lawyers.
>
> Some names that were tossed around included Juliet and Carousel. There were many more, but none felt „just right“.
>
> Finally after lunch, while driving back to Sun, Amy Fowler (lead engineer of the team) asked our most hip team member, Georges Saab, „Georges, you know what's up and coming... what's the new happening thing in San Francisco?“
>
> Georges responded with „Swing dancing is getting to be really big.“ And that was it, we all knew it was perfect. When we got back to the office I did a global search and replace of „kfc“ with „swing“, and the rest is history.

29.11 Aufgaben

In den folgenden Aufgaben sollen Sie den Einsatz der einzelnen Swing Komponenten weitgehend isoliert üben. Ein Schema, auf dem Sie aufbauen können (aber nicht müssen!) finden Sie in *ws2010/a00*. Dort wird eine Klasse *ShowInFrame* verwendet.

1.

Kapitel 30

Javascript — Nashorn Engine

[1]

30.1 Übersicht

Am 18. September 1995 bietete der Navigator Browser von Netscape das erste mal die Möglichkeit eingebetteten Script Code aus HTML Dateien auszuführen. Die Scriptsprache, von Brendan Eich entwickelt und damals noch unter dem Namen LiveScript bekannt, wurde in Zusammenarbeit mit Sun Microsystems weiterentwickelt und ist nun unter dem Namen JavaScript in jedem populären Browser implementiert. Da der Eingebettete Code in einer Script Engine im Browser des Clients ausgeführt wird, bietete es nie zuvor dagewesene Möglichkeiten wie die dynamische Manipulation und Generierung von HTML Code und damit interaktive Webseiten.

Andere Unternehmen, wie zum Beispiel Microsoft, erkannten das Potential und entwickelten ihre eigenen Dialekte (JScript) welche kompatibel waren und fügten eigene Funktionalitäten hinzu. Im Jahr 1996 wurde daraufhin ein Standart für Clientseitige Scriptsprachen im Web entwickelt und 1997 unter dem Namen ECMAScript veröffentlicht [ECM15b].

Jedoch nicht nur im Web sondern auch in anderen Applikationen, welche Scripting Funktionalität zur Verfügung stellen, werden Sprachen nach dem ECMAScript Standard entworfen. Beispiele hierfür sind das KDE für Linux, welches gleich mehrere Dialekte unterstützt, Adobe Produkte mit ExtendScript und das Microsoft .NET Framework das die JScript .NET Engine zur verfügung stellt. Die noch immer populärste Realisierung von ECMAScript ist JavaScript, das größte Anwendungsgebiet das Web, wobei fast jeder moderne Browser eine eigene implementierung der Script-Engine enthält. Einige dieser Engines können auch eigenständig verwendet werden wie zum Beispiel die JavaScript Engine V8 von Googles Chrome, welche sich auch auf der Serverseite in Platformen wie Node.js wiederfindet[ecm15a]. Oder die von der Mozilla Foundation entwickelte Engine Rhino die im Rahmen des Packages javax.script erstmals in Java 6 enthalten war, welches hinzugefügt wurde um Scripting Funktionalitäten in Java zu ermöglichen.[scr14] Ein Neuerung der Java Version 8, welche im März 2014 erschien, ist die neue JavaScript Engine mit dem Namen Nashorn. Rhino und Nashorn haben zwar mit ihrem Namen etwas gemeinsam, unterscheiden sich jedoch Erheblich bei der Implementierung. Während Rhino in Java 5 geschrieben ist, basiert Nashorn auf dem Projekt der DaVinci Machine, welches zum Ziel hat die Architektur der Java Virtual Machine so zu erweitern dass auch andere Sprachen als Java zu effizientem Bytecode für die JVM kompiliert werden können. Der Focus ist hierbei auf dynamisch typisierten Sprachen gerichtet [dav] und so wurden mit Java 7 die JVM Instruktion invokedynamic sowie MethodHandles hinzugefügt um performanteren Bytecode für dynamische Sprachen erzeugen zu können und die Implementierung deren Compiler zu vereinfachen.[jsr11] Ziel des Nashorn Projektes ist es diese neuen Möglichkeiten so gut wie möglich zu Nutzen um eine Performanz für Kompilierten JavaScript Code zu erreichen die an den von Java Code herankommt.[pro14]

[1] Die Basis dieses Kapitels bildet eine Hausarbeit von Jonas Schäufler aus dem WS 2014/15.

30.2 Lernziele

-

30.3 Nashorn in Java

An der Java Scripting API hat sich seit Java 7 nicht viel verändert[sgu15]. Weiterhin werden die ScriptEngines vom ScriptEngineManager verwaltet und können mittels der Methode getEngineByName instanziiert werden. Beispiele für deren Verwendung befinden sich in dem zu dieser Hausarbeit gehörenden Eclipse Workspace[git15]. Dieser beinhaltet außerdem den hier gezeigten Quellcode, weitere Beispiele, den Quellcode um die Benchmarks durchzuführen, die Ergebnisse, sowie den Quellcode der verglichenen Engines. Hier ist anzumerken dass diese mit *ant* kompiliert werden müssen.

Nashorn bietet die Möglichkeit der Engine Parameter zu übergeben. Hierfür wird, statt dem Manager, ein Factory Objekt erstellt welches die Instanz der Engine mit den gewünschten Parametern zurückgibt:

```
%%\lstinputlisting[caption=EngineParameterExample.java,title=Listing 2.1: EngineParameterExample.
```

Die den meisten Parameter lassen sich debugging Informationen über die Engine erhalten. Man kann aber auch verschiedene Einstellungen vornehmen, wie die größe des Cache für die generierten .class Dateien zu ändern, oder bestimmte Funktionalitäten der Engine an-/ausschalten. Eine komplette Liste aller möglichen Parameter befindet sich im Nashorn Quellcode im Package *jdk.nashorn.internal.runtime.resources* in der *Options.properties* Datei. Die meisten Parameter sind nicht dokumentiert, woran man merkt dass Nashorn ein Projekt in entwicklung ist.

Eine Neuerung der Scripting API ist der Umgang mit Objekten die im Kontext der Engine erstellt wurden. Hierfür wurde die Klasse ScriptObjectMirror eingeführt. Sie ist eine Wrapper-Klasse und repräsentiert Objekte in Java die zum Kontext der Engine gehören. Dazu zählen JavaScript Objekte sowie Java Klassen Instanzen die mit Nashorns Java API erstellt wurden. Um zu zeigen wie mit verschiedenen Typen ungegangen wird ist in Listing 30.3 die Ausgabe des Programmes GetExample.java dargestellt.

```
\lstinputlisting[caption=GetExampleOutput, title=Listing 2.2: GetExampleOutput,label=lst:getout,b
```

Listing 30.3 zeigt die erstellung der Objekte. In diesem Programm werden mittels der *get* Methode der *ScriptEngine* Instanz werden Referenzen auf die in JavaScript erstellten Objekte gespeichert und deren Typen ausgegeben.

```
\lstinputlisting[caption=GetExample.java,title=Listing 2.3: GetExample.java,language=Java,label=l
```

Wie man in Listing 30.3 sehen kann werden JavaScript und Java Objekte, und gegebenenfalls deren Attribute, sowie Funktionen mit dem ScriptObjectMirror Wrapper zurückgegeben. Die Klasse ScriptObjectMirror stellt Methoden zur verfügung um Attribute und Funktionen dieser Objekte zu verwenden und sie wenn nötig zu *unwrappen* um an das darunterliegende Objekt zu gelangen[som15]. JavaScript Typen *number* und *string* werden in ihren, aus Java Sicht, 'nativen' Gegenstücken Integer und String referenziert.

30.4 Nashorn Java API

Mit Nashorn können Java Klassen auf einfache Weise verwendet werden. Es muss jedoch der volle Pfad der Pakete zur Klasse verwendet werden. Wie das Beispiel in Listing 30.4 zeigt ist der Code dadurch jedoch anfälliger für Fehler.

```
\begin{lstlisting}[caption=String Beispiel,label=lst:examplerino]
jjs> java.lang.String
```

```
[JavaClass java.lang.String]
jjs> var s1 = new java.lang.String("test");
jjs> var s2 = new String("test");
jjs> print("typeof s1: " + typeof s1 + " typeof s2: " + typeof s2);
typeof s1: string typeof s2: object
jjs> s1.getClass()
class java.lang.String
jjs> s2.getClass()
<shell>:1 TypeError: Cannot call undefined
jjs> Java
[object Java]
\end{lstlisting}
```

Um den Umgang zu erleichtern ist das globale Objekt *Java* eingeführt worden. So können mit der *type* Methode Referenzen auf Klassen in einer Variablen gespeichert werden, um sie im Code mit dieser Referenz zu Instantiieren. Dies funktioniert auch mit primitiven Typen und Arrays. Man könnte auch die Referenz ohne *type* Methode speichern, jedoch würde dann nicht überprüft ob es sich wirklich um einen Java Typen handelt. Um die Verwendung der *type* und der *to* Methode zu zeigen ist in Listing 30.4 ein Beispielscript dargestellt.

```
\lstinputlisting[caption=javaTypeExample.js,title=Listing 3.2: javaTypeExample.js,label=lst:jte,l
```

Es werden in Variablen Referenzen auf Java Typen gespeichert und deren Objekte instantiiert. Das Java *Integer* Array wird mittels der *to* Methode erstellt. Diese konvertiert JavaScript Arrays in deren komplementären Java Typ. Ihr äquivalent ist die Methode *from*. Ein weiterer Vorteil ist dass das Array beim erstellen Initialisiert werden kann, wie in Zeile 7 in Listing 30.4 zu sehen ist. Aus zwei Arrays der Typen *Integer* und *int* wird versucht mit der *asList* Methode eine *ArrayList* zu erstellen. An der Ausgabe in Listing 30.4 sieht man dass beim primitiven *int* Typen das Array Objekt zur Liste hinzugefügt wird. Bei einem *Integer* jedoch dessen Elemente. Dies ist ein Beispiel dafür, dass dieser Weg immernoch anfällig für Fehler ist, da für jeden Typ ein beliebigen Bezeichner vergeben werden kann.

```
\lstinputlisting[caption=javaTypeExample Output,title=Listing 3.3: javaTypeExample Output,label=l
```

Eine Lösung dafür ist das *JavaImporter* Objekt. Anders als das globale *Java* Objekt, muss der *JavaImporter* davor instantiiert werden. Mit diesem können Java Pakete importiert werden ohne dass ein neuer Bezeichner dafür vergeben werden muss und ohne den Kontext mit unnötigen Imports zu verschmutzen. Wenn sich Bezeichner überschneiden, wie bei *String*, muss für die Java Klasse weiterhin der Pfad angegeben werden.

```
\lstinputlisting[caption=javaImporterExample.js,title=Listing 3.4: javaImporterExample.js,label=l
```

Das in Listing 30.4 ist die verwendung des JavaImporters dargestellt. Das Script erzeugt folgenden Output:

```
\lstinputlisting[caption=javaImporterExample Output,title=Listing 3.5: javaImporterExample Output
```

Im Beispielscript in Listing 30.4 werden Strings auf drei verschiedene Weisen erzeugt. In Zeile 4 ein JavaScript Objekt mit der *String* funktion. Und in Zeile 5 und 10 Java Strings mittels des vollständigen Pfades und einem Literal. Alle Nummern sind vom Java Typ *Integer*. Würde man auf den JavaScript String die *getClass* Methode aufrufen, würde das zu einer Exception führen.

Mit *Java.extend* können Java Klassen abgeleitet werden. Als Beispiel wird diese Java Klasse verwendet:

```
\lstinputlisting[caption=ExtendExampleClass.java,title=Listing 3.6: ExtendExampleClass.java,langu
```

Ist die *class* Datei im Klassenpfad kann mit *extend* eine Ableitung erstellt werden und die *greetings* Methode überschrieben werden.

```
\lstinputlisting[caption=extendingClass.js,title=Listing 3.7: extendingClass.js,label=lst:ext,lar
```

Am Beispiel in Listing 30.4 wird in Zeile 13 die *super* Methode verwendet um aus dem Objekt einer Abgeleiteten Klasse einen *JavaSuperAdapter* zu erstellen, welche mit den Methoden der Superklasse gelinkt wird aber auf den Attributen der Ableitung arbeitet. Das Programm gibt folgendes aus:

```
\begin{lstlisting}[caption=extendingClass.js Output,label=lst:exto]
Hello from Java
Greetings from JavaScript
Hello JavaScript
jdk.nashorn.internal.runtime.linker.JavaSuperAdapter@2f490758
jdk.nashorn.javaadapters.ExtendExampleClass@101df177
\end{lstlisting}
```

Im Beispiel 30.4 verwendet die Methode *greetings* keine Klassenattribute oder Methoden der Superklasse. Will man mit der neuen Funktion darauf zugreifen so muss der Code leicht abgeändert werden, wie in Listing 30.4 dargestellt ist. Hier wird die Implementation der Funktion nicht der *extend* Funktion übergeben sondern dem Konstruktor bei der Instanziierung.

```
\lstinputlisting[caption=extendingClass2.js,title=Listing 3.9: extendingClass2.js,label=lst:ext2,
```

Mit der *extend* Funktion könnte man auch das *Runnable* Interface implementieren. Mit Java 8 ist es möglich eine anonyme Klasse mit einem Lambda-Ausdruck zu ersetzen. In Nashorn wird der Lambda-Ausdruck durch eine Funktion dargestellt.

```
\lstinputlisting[caption=lambdaExample2.js,title=lambdaExample2.js,label=lst:lex2,language=JavaSc
```

So kann man, wie in Listing 30.4 dargestellt, Interfaces implementieren oder Java Streams verwenden (Siehe 30.4).

```
\lstinputlisting[caption=lambdaExample.js,title=lambdaExample.js,label=lst:lex,language=JavaScrip
```

Nashorn ermöglicht es JavaScript Code zu schreiben der fließend in Java übergeht und auch die Möglichkeit neue Features von Java 8 wie Streams und Lambda-Ausdrücke auszunutzen.

30.5 Nashorn in der Shell

Eine weiteres Produkt des Nashorn Projektes sind die Interpreter *jrunscript* und *jjs*. *jrunscript* ist ein Sprachunabhängiger Interpreter welcher mit Engines die nach der Spezifikation im JSR 223 entworfen wurden verwendet werden kann. Da auch Rhino der Spezifikation folgt kann auch diese mit *jrunscript* verwendet werden. Dafür benötigt man eine kompilierte Version von Rhino mit welche die jar der JSR-223 Script Engine erstellt wird. Befinden sich beide jar Dateien im Klassenpfad kann man, mit dem Parameter -l, Rhino als Engine auswählen. In Listing 30.5 ist die Ausgabe eines Aufrufes, mit den Parametern *-cp js-engine.jar:js.jar -q* um die verfügbaren Engines anzuzeigen, dargestellt.

```
\lstinputlisting[caption=jrunscript Output,title=Listing 4.1: jrunscript Output,label=lst:jrso,sh
```

Die default Engine ist, wie bei *jjs*, Nashorn. Jedoch stellt dieses noch weitere Optionen zur Verfügung um mit JavaScript möglich zu machen was auch andere interpretierte Scriptsprachen, wie zum Beispiel Perl, PHP oder Python, bieten. Eine der Stärken von JavaScript mit Nashorn ist die gute Kopplung mit Java. So kann mit der Option *-fx* das Paket JavaFX verwendet werden, welches seit Java 8 das empfohlene Paket zum erstellen von Benutzerobeflächen ist.

Mit der -scripting Option von jjs wird die Skript-Erweiterung aktiviert. Nashorn unterstützt die Shebang Notation mit welcher zu Begin eines Scriptes der Interpreter und Parameter festgelegt werden können. Startet eine Datei mit einem Hash-Zeichen (#), wird die Skript-Erweiterung

automatisch aktiviert. Alle weiteren Hash-Zeichen werden, in diesem Modus, wie in UNIX Shells als Kommentar interpretiert. Um die UNIX Shell mit JavaScript in einer Kommandozeileninterpreter Umgebung voll ausnutzen zu können werden von der Engine Objekte bereitgestellt mit welchen auf Argumente und Umgebungsvariablen der Shell-Umgebung zugegriffen werden kann. Zusätzlich ist es möglich Kommandos der Shell oder andere Ausführbare Dateien wie in einem Bash-Script zu verwenden. Als Beispiel ist in Listing 30.5 ein Script dargestellt, dass den *ls* Befehl verwendet und dessen Ausgabe mit dem übergebenen Argument gefiltert wird.

```
\lstinputlisting[caption=shellInvocation.js,title=Listing 4.2: shellInvocation.js,label=lst:enve
```

Man beachte dass in Zeile 2 in Listing 30.5 das Gravis, oder *Backtick*, verwendet wird um einen Befehl in der Shell auszuführen. Die Argumente an das Script müssen beim Aufrufen nach einem doppelten Bindestrich (-) folgen:

```
\lstinputlisting[caption=shellInvocation Output,title=Listing 4.3: shellInvocation Output,label=
```

Eine andere möglichkeit ist das, von der Engine im Scripting Modus zur verfügung gestellte, globale Objekt *$EXEC*. Mit diesem lassen sich *Shell-Invocations* durchführen welche Input benötigen welcher nicht hard Kodiert ist und man hat Zugang zu den Standart Input-/Output und Errorstreams.

```
\lstinputlisting[caption=shellInvocation2.js,title=Listing 4.4: shellInvocation2.js,label=lst:env
```

In Listing 30.5 ist ein Script dargestellt dessen Funktionalität die selbe ist wie in 30.5. Nur verwendet dieses die Objekte *$EXEC* und *$OUT* und das Programm *grep*; Zusätzlich werden im Script-Modus die Funktionalitäten String Interpolation und Heredocs aktiviert, wie sie auch in der Shell zur Verfügung stehen [nhh15]. Nashorn bietet außerdem einige zusätzliche Funktionen welche im Interpreter Modus nützlich sind. Zum Beispiel die Funktion *load* beziehungsweise *loadWithNewGlobal* mit welcher Scripte aus dem Dateisystem, als String oder von einer URL, im aktuellen oder einem neuen Kontext, ausgeführt werden können.

30.6 Historische Anmerkungen

30.7 Aufgaben

Kapitel 31

Refactoring

Nichts ist überflüssig. Es kann zumindest noch als abschreckendes Beispiel dienen.

31.1 Übersicht

Elemente von Sourcecode sind oft verbesserungsfähig. Beim Entstehen des Codes sind diese Teile nicht immer sofort zu erkennen. Aber hier gilt wie in vielen anderen Lebensbereichen:

„Wehret den Anfängen“ oder auf englisch „Do it right first time“.

In diesem Kapitel stelle ich einige bewährte Techniken oder Strategien vor, um dies auch praktisch umzusetzen. Als Programmiersprache habe ich dabei Java vor Augen, aber alles, was ich hier erläutere, ist überall in der Informatik anwendbar.

Refactoring zielt darauf, Unzulänglichkeiten in Code (oder sonst wo) sofort zu beheben. Dies ist ganz im Sinne der Theorie, dass ein Fehler um so teurer ist, je früher im Entwicklungsprozess er begangen wird.

31.2 Lernziele

- Den Begriff *Refactoring* definieren können.
-
-
- Erläutern können, warum schon ein Umbenennen eines Elements nicht trivial ist.

31.3 Grundbegriffe

Definition 31.3.1 (Refactoring)
Unter *Refactoring* versteht man das Verändern von Sourcecode mit zwei Ergebnissen:

- Die Qualität des Codes nach akzeptierten Kriterien wird verbessert.
- Der Code liefert garantiert das gleiche Ergebnis wie vor der Refaktorisierung.

Ganz allgemein kann man Refactoring für Modelle definieren, indem man Sourcecode durch Modell ersetzt. ◄

Bemerkung 31.3.2 (Refactoring)
Der Ausdruck *Refactoring* hat sich auch im Deutschen durchgesetzt. Trotzdem spreche ich von *refaktorisieren*, wenn ich von dem Prozess spreche. „Akzeptierte Kriterien“ können sich ändern, sind aber in diesem Kontext seit Jahrzehnten unverändert: Es wird geringe Kopplung und hoher Zusammenhalt angestrebt. Eng damit verbunden ist das Kriterium der *Verständlichkeit*. Auch einen falsch geschriebenen Namen eines Elements sollten Sie unverzüglich korrigieren! In der angemessenen Konkretisierung werden Sie diese Kriterien auch auf allgemeinere Modelle anwenden können. ◄

Dieses Kapitel soll keine Liste von Refactorings werden, wie Sie sie z. B. in [Fow99] finden. Ich habe dieses Buch übersetzt [Fow00]. Das ist aber etwas ganz anderes, als ein Buch zu lesen. Nach meiner Erinnerung werden durch die Refactorings bewährte Desingprinzipien nachträglich in COde hineingebracht, die in der Hektik(?) der Entwicklung zunächst vegessen oder zunmindest vernachlässigt wurden. Ich werde mich bemühen, einige Grundprinzipien herauszuarbeiten. Weitere Details können Sie in der Literatur nachlesen.

31.4 Ein kleines Beispiel

Das folgende Beispiel ist echt! Aus verständlichen Gründen nenne ich den Autor nicht bzw. „N .N“. Tatsächlich erinnere ich ihn oder sie auch nicht mehr und auch wenn ich es in meinen Archiven noch finden könnte, habe ich zu der Recherche keine Lust. Ich beginne mit der Aufgabenstellung:

1. Sie sollen eine Klasse schreiben, die einen Temperaturwert enthält. Die Klasse soll die Temperaturen in verschiedenen Einheiten liefern können. Einige Einheiten, die Sie unterstützen sollen, finden Sie im Interface ITemperatur.
2. Überlegen Sie sich, in welcher Form Sie die Temperaturwerte festhalten wollen! Begründen Sie bitte Ihre Entscheidungen! Das gilt für die Attribute, die Methoden und die Konstruktoren!
3. Schreiben Sie bitte eigene JUnit-Tests, die demonstrieren, dass Ihre Implementierung funktioniert!
4. Verwenden Sie Ihre Klasse um eine Tabelle der Temperaturen in verschiedenen Einheiten auf der Konsole auszugeben: Ich erwarte, dass ein Anfangswert und ein Endwert sowie eine Schrittweite für die Temperaturen in Celsius angegeben werden kann. Als Ergebnis erwarte ich eine Tabelle der Temperaturen, die in jeder Zeile die Werte in den im Interface verwendeten Einheiten zeigt. Das könnte etwa so aussehen:

Kelvin	Celsius	Reaumur	Fahrenheit
0	-273,160	-218,528	-459,670
10	-263,160	-210,528	-441,670
...	...	...	...

5. Denken Sie bitte daran, die verwendeten Quellen korrekt anzugeben!

Hinweis: Der Einfachheit halber brauchen Sie sich noch nicht darum kümmern, dass die Temperaturen gültig sind, also größer oder gleich 0 Grad Kelvin.

Das erwähnte Interface ITemperatur finden Sie in meinem pub unter *Loesungsvorschlaege/a01* und hier

```
package a01;

/**
 * Ein Interface für verschiedene Darstellungen von Temperaturen
 * @author Bernd Kahlbrandt
 */
```

```
public interface ITemperatur {
/**
 * Liefert die Temperatur in Grad Kelvin.
 * @return Temperatur in Grad Kelvin.
 */
double getKelvin();
/**
 * Liefert die Temperatur in Grad Celsius
 * @return Temperatur in Grad Celsius.
 */
double getCelsius();
/**
 * Liefert die Temperatur in Grad Fahrenheit.
 * @return Temperatur in Grad Fahrenheit.
 */
double getFahrenheit();
/**
 * Liefert die Temperatur in Grad Reaumur.
 * @return Temperatur in Grad Reaumur.
 */
double getReaumur();
}
```

Nun der Lösungsvorschlag, den ich in diesem Abschnitt sezieren werden:

```
public class SoNicht {
static {
   System.out.println("Dieses Programm rechnet Celsius in Kelvin, Reaumur und Fahrenheit um.\n");
   double kelvin;
   double reaumor;
   double fahrenheit;
   double start=0;
   double end=0;
   double schritt=0;
   BufferedReader bin = new BufferedReader(new InputStreamReader(System.in));
   System.out.println("Startwert Celsius: ");
   String eingabe;
   try {
         eingabe = bin.readLine();
         start = Double.parseDouble(eingabe);
       } catch (IOException e) {
         e.printStackTrace();
       }
   System.out.println("Endwert Celsius: ");
   try {
      eingabe = bin.readLine();
      end = Double.parseDouble(eingabe);
      } catch (IOException e) {
               e.printStackTrace();
         }
   System.out.println("Schrittweite: ");
   try {
      eingabe = bin.readLine();
      schritt = Double.parseDouble(eingabe);
       } catch (IOException e) {
```

```
                e.printStackTrace();
            }

    System.out.println("\n" + "Celsius:" + "\t" + "Kelvin:" + "\t\t"
                        + "Reaumor:" + "\t" + "Fahrenheit:");
            for (double celsius = start; celsius <= end; celsius += schritt) {
                  kelvin = (double) celsius + 273.15;
                  reaumor = (double) Math.round(((kelvin - 273.15) * 0.8) * 100.00) / 100.00;
                  fahrenheit = (double) Math.round((celsius * 1.8 + 32) * 100.00) / 100.00;
                  // fahrenheit = (double) celsius * 1.8 + 32; //Ausgeklammert um
                  // einen Fehler reproduzierbar zu machen
                  System.out.println(celsius + "\t\t" + kelvin + "\t\t" + reaumor
                                    + "\t\t" + fahrenheit);
            }
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
    }
}
```

Durch diesen Lösungsversuch bin ich überhaupt erst darauf aufmerksam geworden, dass es einen *statischen* Initialisierungsblock gibt. In der Vorlesung habe ich so etwas aufgrund meines damaligen (Un-) Kenntnis gar nicht präsentieren können. Eigentlich kann man solchen Code nur „in die Tonne treten“. Aber ich habe eine grobe Vorstellung, wie jemand dazu kommen kann, solchen Code zu schreiben:

1. Es beginnt mit einer Klasse die eine *static main*-Methode hat.
2. Alles, was man da machen will muss *static* sein.
3. Vor lauter Verzweiflung über die vielen Fehlermeldungen macht man alles *static*.
4. Irgendwannkommt man irgendwie auf einen Block $\{\ldots\}$. Da gibt es plötzlich weniger Fehlermeldungen, abder eine bleibt noch und die kriegt man dann weg, indem man den Quicktipp von Eclips verwendet und *static* davor schreibt. Und fertig ist das abschreckende Beispiel.

Die Entscheidung, was hier als Erstes verbessert werden muss, fällt mir schwer, da der Code so völlig „verwarzt“ ist. Ich fange deshalb mit einer ganz einfachen Maßnahme an: ich benenne die Variable *reaumor* in die korrekte Schreibweise *reaumur* um. Ändern Sie einfach die Deklaration *double reaumor;* in *double reaumur;*, so erzeugen Sie viele Compiler-Fehler. Verwenden Sie die Refactoring Möglichkeiten von Eclipse, so stellen Sie sicher, das die Namensänderung an allen Stellen nachvollzogen wird. *alt+shift+r*. Weitere Tastenürzel für Eclipse finden Sie in B.9. Das ändert aber nichts an der Qualität der Code-Struktur.

31.5 Eine etwas größere Fallstudie

In Eclispe erreichen Sie dies einfach durch Im WS 2002/2003 wurde mir eine Lösung einer Übungsaufgabe präsentiert, die sich durch zwei Dinge auszeichnete:

1. An der Oberfläche sah sie richtig gut aus. De fakto hatte sie von allen die ansprechendste Oberfläche.
2. Die Innere Struktur war abstrus: Sie bestand aus zwei Java-Klassen, von der die eine nur aus einer `main`-Methode, die ein Objekt der anderen Klasse erzeugt.

Also eine Lösung die dem Spruch „Außen hui, innen pfui“ entspricht. Aber da nichts wirklich überflüssig ist, es kann schließlich notfalls noch als abschreckendes Beispiel dienen, verwende ich dies, um ein Beispiel einer Refaktorisierung vorzuführen.

31.5.1 Ausgangssituation

Die Javadoc Dateien der Originalklassen findet man hier: .

Ich beginne mit der Diskussion einiger Qualitätsüberprüfungen und Metriken. Die Audit-Komponente von Together liefert zwei Hinweise, die als schwerwiegend eingestuft werden: An zwei Stellen wird eine Variable mit dem gleichen Namen wie ein Attribut definiert. Die Vermutung liegt nahe, dass hier ein Programmierfehler vorliegt.

Ich werde versuchen, dies zu testen.

Des weiteren werden einige weitere Dinge bemängelt:

1. Komplexe Zuweisungen: left=links=rechts=right=0;
2. Mehrere Variablendeklarationen in einer Zeile.
3. Nicht standardkonforme Operationsnahmen: Underscore, Großschreibung erster Buchstabe.
4. Attribut Temp beginnend mit einem Großbuchstaben.
5. Nicht-finales Klassenattribut `animation`
6. Öffentliche Attribute
7. String Literale (Man sollte nichts derartige „hardcodieren“.
8. Festdefiniert Font-Größen

31.6 Refactoring zu λ-Ausdrücken

Der folgende Code-Ausschnitt

```
@Override
 public void start(Stage primaryStage) {
     final Timeline timeline = new Timeline(new KeyFrame(Duration.ZERO, new EventHandler() {
         @Override
         public void handle(Event event) {
             iterateBoard();
         }
     }), new KeyFrame(Duration.millis(100)));
```

bringt in einer Java 8 Umgebung die Compiler Warnungen

```
EventHandler is a raw type. References to generic type EventHandler<T> should be parameterized
Type safety: The expression of type new EventHandler(){} needs unchecked conversion to conform to
```

Die Ursache ist klar, die Behebung (in Eclipse Luna) nicht unmittelbar. Der Typ-Parameter *Object* wird von Eclipse als Quick Fix vorgeschlagen, aber nicht vom Compiler akzeptiert. *ActionEvent* wird auch nicht akzeptiert. ABer der Ersatz durch einen λ-Ausdruck wird akzeptiert.

31.7 Refaktorisierungen

31.8 Werkzeuge

Das Ruby refactorin plugin für vim unterstützt folgende Refactorings:

```
:RInlineTemp             - Inline Temp
:RConvertPostConditional - Convert Post Conditional
:RExtractConstant        - Extract Constant
:RExtractLet             - Extract to Let (Rspec)
:RExtractLocalVariable   - Extract Local Variable
:RRenameLocalVariable    - Rename Local Variable
:RRenameInstanceVariable - Rename Instance Variable
:RExtractMethod          - Extract Method
```

31.9 Historische Anmerkungen

31.10 Aufgaben

Kapitel 32

Miniprojekt: Rechner

32.1 Übersicht

Als kleines Projekt oder als Folge von Übungsaufgaben soll hier ein „Taschenrechner“ entwickelt werden. Der sollte in der ersten Version etwas so aussehen, wie die bekannten Rechner aus Windows (calc.exe) oder Unix in der Standardansicht, siehe Abb. 32.1. Diese Basisversion soll dann weiterentwickelt werden, um verschiedene Java Konstrukte praktisch einzusetzen. So soll es verschiedene Ansichten geben. Der Windows Rechner hat u. a. die wissenschaftliche Ansicht und kann darüberhinaus u. a. noch auf hexadezimal umstellen. So kann auch das Rechnen mit verschiedenen Basen eingeübt werden. Dies lässt sich auch weiter skalieren: So können statistische und andere mathematische Funktionen ergänzt werden, wie die Suche von Nullstellen, Differenzieren, Integrieren oder das Zeichnen von Graphen. Darüberhinaus kann man die Anwendung als Applet oder Service über das Internet verfügbar machen.

Dies Beispiel kann zur Illustration verschiedener Klassenbibliotheken verwendet werden, z. B. Swing, JavaFX uvam.

32.2 Lernziele

- Mit Zahlen in unterschiedlichen Basen arbeiten können.
- Einfache Entwurfsprinzipien kennen.
- Einige nützliche Datenstrukturen anwenden können.
- Grundlagen der Internationalisierung kennen.
- Grundsätze der GUI Programmierung in Java beherrschen.

32.3 Rechner: Erste Schritte

Mit allem, was in diesem Kurs behandelt wird, sollten Sie am Ende in der Lage sein, die Grundlage zu einem Rechner zu legen, der wie die üblichen *Rechner* in Windows oder Unix, mit Dezimalzahlen rechnen kann, aber auch mit beliebigen anderen Basen zwischen 2 und 36. Mit etwas mehr Eigenarbeit auch mit ganz anderen Basen, etwa negativen ganzen Zahlen, $2i$ für komplexe Zahlen, gemischten Basen (früheres britisches Pfund, Währung in der Zaubererwelt bei Harry Potter, Gregorianischer Kalender, Maya Kalender. . .).

Nichts desto trotz sind auch hier noch einige Dinge zu analysieren:

1. In einem System Rechnen und für Eingabe und Ausgabe aus bzw. in das vom Benutzer gewählte System umrechnen? Welche Kriterien sollen für die Entscheidung herangezogen werden?

2. Im vom Benutzer gewählten System auch rechnen?

Zunächst einmal überlegen wir uns, wie die Oberfläche des Rechners aussehen soll. Abbildung 32.1 gibt schon mal einen guten Eindruck, von dem was wir machen wollen. Wir sehen dort bereits

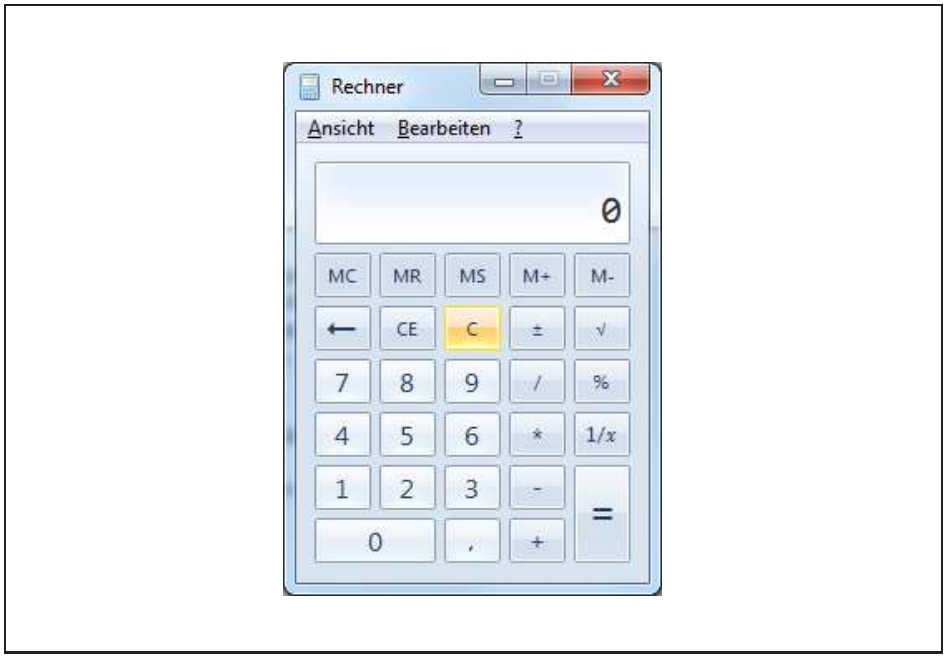


Abb. 32.1: Rechner aus Windows

Folgendes:

1. Wir haben ein Fenster mit einem Balken an der oberen Seite mit einem Titel (hier *Rechner*) und links einem Icon, hinter dem sich ein Menü verbirgt und rechts der der Möglichkeit zu minimieren und zu schließen, maximieren wird nicht angeboten. Das können wir uns für unseren Rechner natürlich auch noch anders überlegen.

2. Wir haben eine Menüleiste

Bereits die erste Eigenschaft legt die Wahl eines JFrames nahe, klar wird das durch das Menü.

Das Gerüst kriegen können Sie damit schon einmal in einer Klasse *Calculator01* zusammenbauen.

Die Klasse *Calculator01* ist einfach unsere erste Version. Sind wir später mit einer Entwicklung zufrieden, können wir immer noch zu einem aussagefähigen Namen refaktorisieren. Diese Klasse erweitert *JFrame*. *JFrame* implementiert *Serializable*, daher benötigen wir das Klassenattribut *serialVersionUID*. Diese Warnung können wir aber ignorieren und stattdessen die Serialisierungsmöglichkeiten von Beans verwenden (s. u.).

Bisher kann diese Klasse noch gar nichts, außer den ererbten Methoden. In der *main*-Methode erzeugen wir ein neues Calculator1 Objekt sorgen mit *pack()* sorgt dafür, dass alle Inhalte des Frames mindestens ihre bevorzugte Größe haben und in das Fenster passen. *setVisible* sorgt dafür, dass das Fenster auch sichtbar ist.

Im Ergebnis haben wir eine leeres Fenster, dass wir verschieben, größer oder kleiner ziehen können und schließen können.

Einen Titel bekommen wir sofort beim Entwickeln des Konstruktors. Dort rufe ich als erstes den Konstruktor JFrame(String Title) auf (mittels super). Genausogut kann man den Titel über die Operation setTitle setzen, die *Calculator01* über JFrame von awt.Frame erbt.

Als Nächstes wenden wir uns einer weiteren Trivialität zu und fügen einen Menubar ein.

Hier muss aber gewarnt werden. Auch wenn dies in vielen Tutorials, Lehrbüchern (z. B.) so beschrieben wird: Das tut man so nicht. Solche statischen Texte haben in Code nichts zu suchen.

Deshalb wird dies auch sofort geändert! Eclipse bietet hierfür die Funktion „Externalize Strings“, siehe Abschn. 21.5.

Für die ersten Implementierungsstufen bietet sich eine kompakte Implementierung mit wenigen Klasse an, vielleicht sogar nur einer Klasse.

32.4 Ein ganzzahliger Rechner

Ein einfacher Rechner mit der mit ganzen Zahlen im Dezimalsystem rechnet, kann so aussehen, wie in Abb. 32.1, wobei wir außer den Zifferntasten nur vier Tasten für die Symbole der Grundrechenarten und die Ergebnistaste benötigen.

32.5 Verschiedene Basen

Wollen wir mit verschiedenen Basen Rechnen, so müssen wir das Tastenfeld anpassen. Im Binärsystem brauchen wir eigentlich nur die Tasten 0 und 1. Im Hexadezimalsystem aber 0 bis F. Hier ist zu entscheiden, ob immer alle Symbole verfügbar sind. Dies ist im Windows Rechner der Fall: Dort sind immer alle Tasten sichtbar und die für die jeweilige Basis (2, 8, 10, 16) benötigten auswählbar. Sollen alle Basen verfügbar gemacht werden, die mit Java einfach zu unterstützen sind, so muss man sich allerdings etwas mehr Gedanken über die Gestaltung machen.

Für die Auswahl der Basen könnte eine Combo-Box verwendet werden. Eine andere Idee wäre ein Doppelbelegung der Tasten.

32.6 Fließkommarechnung

Im nächsten Schritt soll die Funktionalität um die Rechung mit Fließkommazahlen erweitert werden. Hier muss über die Genauigkeit entschieden werden. So hat der Windows-Rechner eine höhere Genauigkeit, als der Datentyp *double* in Java bietet. Hier könnte man intern mit *BigDecimal* arbeiten. So könnte die Genauigkeit auch konfigurierbar gemacht werden.

32.7 Weitere Funktionen

Wie in jeder Windows-Anwendung wird man Funktionalität wie rückgängig machen einer Eingabe, cut& paste etc. erwarten. Diese sollen in diesem Schritt hinzugefügt werden.

32.8 Speichern

In einem weiteren Schritt werden Speicher und Operationen für deren Manipulation ergänzt. Wie bei früheren, für damalige Verhältnisse leistungsfähigen Taschenrechnern, kann man auch mehrere Speicher vorsehen. Die hatten oft die Möglichkeit eine Zahl zu der im Speicher zu addieren. Wir können hier gleich mehr Fumktionen realisieren, analog zu den Java-Operatoren +=, -=, *=,. . . . Außerdem soll der gesamte aktuelle Rechner-Zustand gespeichert werden können.

32.9 Mathematische Funktionen

In diesem Schritt sollen mathematische Funktionen und Konstanten, wie x^y, $log_x y$,e, π etc. ergänzt werden.

32.10 Erweiterungen

Diverse statistische, ökonometrische etc. Funktionalität ergänzen. Einige Ideen hierzu:

1. Tilgung von Annuitäten Darlehen.
2. Investitionsrechnung:
   2.1. Interner Zinsfuß,
   2.2. Barwert,
   2.3. Vollständige Finanzpläne.
3. Einheitenumrechnung
4. ...

32.11 Internationalisierung

Externalisieren von Strings und die Oberfläche in mehreren Sprachen anbieten.

32.12 Historische Anmerkungen

32.13 Aufgaben

Anhang A

Programmierrichtlinien (Java)

Nearly everybody is convinced that every style but their own is ugly and unreadable. Leave out the „but their own“ and they're probably right... Jerry Coffin (on indentation), zitiert nach [Batff].

A.1 Übersicht

Ich stelle hier eine Reihe bewährter Regeln für das Schreiben von Java-Code zusammen. Dabei orientiere ich mich an der Empfehlungen von Oracle und bewährten Prinzipien des Software-Engineering. Hier finden Sie nicht Alles, aber das was Sie finden habe ich mich bemüht überzeugend zu motivieren. Sie sollten sehr überzeugende Gründe nennen können, wenn Sie gegen diese Prinzipien verstoßen! Außerdem berücksichtige inzwischen ich viele Punkte der detaillierten Konvention von Google. die Sie in [Goo] finden.

Ich sehe regelmäßig, dass Verstöße gegen diese elementaren Regeln zu Fehlern führen, die bei ihrer Einhaltung gar nicht erst entstehen würden.

A.2 Lernziele

- Wissen, wie Namen von Java-Elementen gebildet werden.
- Die Reihenfolge kennen, in der Elemente in Java-Klassendateien geschreiben werden.
- Grundlegende Richtlinien zum Schreiben guten Java Codes kennen und anwenden können.
- Grundprinzipien der Kopplungsvermeidung einsetzen können.
- Zusammenhalt von Code systematisch stärken können.

A.3 Struktur von Klassendateien

Eine solche Datei *.java*-Datei beginnt üblicherweise mit einem mehrzeiligen (C-Style) Kommentar, der den Klassennamen, den Programmierer, Version, ggf. Copyright (©) etc. enthält.

```
/*
* ClassName
*
* Version info
*
* Copyright notice
*/
```

Liefern Sie tatsächlich Code aus, so sollten Sie dieser Konvention folgen. Für den Rahmen von Java-Programmiervorlesungen verzichte ich auf Kommentare dieser Art.

Anschließend folgt das *package*-Statement und etwaige benötigte *import*-Statements.

Anschließend kommt der Inhalt der Datei, also die Deklaration eines *Interface*s, einer *Interface* oder *Annotation*. Vor dem Element steht ggf. eine Annotation oder ein Javadoc-Kommentar. Ich erwarte vor einer Klasse, einem Interface oder einer Annotation immer einen Javadoc-Kommentar mit einer kurzen Beschreibung (eine Zeile) der Klasse und einen @author tag mit dem Namen jedes der beteiligten Autoren. Ich bevorzuge ein @author tag pro Autor. In Beispielcode, den ich von Anderen übernehme, pflege ich meinen Namen in einem @author tag zu ergänzen und meine Änderungen knapp anzugeben.

Anschließend folgt die Spezifikation der zugehörigen Elemente in der Reihenfolge:

1. Klassenattribute nach absteigender Sichtbarkeit.
2. (Instanz-) Attribute nach absteigender Sichtbarkeit.
3. Konstruktoren nach absteigender Sichtbarkeit.
4. Methoden, logisch gruppiert.

A.4 Namen

Mit dem Begriff *CamelCase* kann man die Namenskonventionen von Java kurz und knapp formulieren. *CamelCase* bedeutet, dass die Zeichen eines Namens Kleinbuchstaben sind und nur an Wortgrenzen das erste Zeichen des Teilwortes ein Großbuchstabe ist. *lowerCamelCase* bedeutet, dass das erste Zeichen ein Kleinbuchstabe ist, *UpperCamelCase*, dass das erste Zeichen ein Großbuchstabe ist. Ferner kennen Sie sicher den Begriff *lowercase* (nur Kleinbuchstaben) und vielleicht auch den Begriff *SCREAMING_SNAKE_CASE* (nur Großbuchstaben und an Wortgrenzen ein Unterstrich. Damit gelten für Java (und viele weitere Programmiersprachen) folgende Namenskonventionen:

1. Typen, also Interfaces, Klassen, Annotationen etc. werden in *UpperCamelCase* gebildet.
2. Attribute und Methoden werden in *lowerCamelCase* gebildet.
3. Namen von lokalen Variablen und Parametern werden in *lowerCamelCase*
4. Namen von Paketen werden in *lowercase* gebildet.
5. Namen von Konstanten werden in *SCREAMING_SNAKE_CASE* gebildet. Dies umfasst auch *Enum*-Werte.

Im Einzelnen kann dies natürlich noch weiter differenziert werden.

Klassen

1. Klassennamen beginnen mit einem Großbuchstaben. An Wortgrenzen wird ebenfalls ein Großbuchstabe verwendet (UpperCamelCase).
2. Klassennamen werden aus Substantiven im Singular gebildet.
3. Für Namen von Utility-Klassen wird ggf. auch der Plural verwendet.
4. Der Name der Klasse soll die Verantwortung der Klasse erkennen lassen.
5. Klassennamen werden in *einer* Sprache gebildet, also konsequent Deutsch, Englisch etc. aber nicht gemischt. Ausnahmen gelten bei Verwendung der Klassenbibliotheken. In diesen werden englische Namen für Klassen und ihre Elemente verwendet.

Beispiel A.4.1 (Utility-Klasse)
Zwei wichtige Utility-Klassen, deren Namen nach obiger Regel gebildet worden sind enthält das Paket *java.util*: *Collections* und *Arrays*. ◄

Hier eine tabellarische Übersicht zur Namensgebung:

Format	Element	Beispiel	Weiteres
UpperCamelCase	Klasse	Person	Substantiv im Singular
	Utility-Klasse	Colleections	oft Plural
	Interface	List	
lowerCamelCase	Attribut (field	name	
	Variable	input	
	Parameter	name	
	Methode	doSonething	oft ein Verb
lowercase	Paket	java.lang	Umgekehrte Domain-Reihenfolge
	Module	java.base	
	JavaFX css	buttonstyle.css	Sonst wie Element
lower_snake_case	JavaFX style class	Sonst wie Klassenname	
SCREAMING_SNAKE_CASE	Konstanten	Math.PI	

Innerhalb einer Klasse werden die Elemente in dieser Reihenfolge deklariert:

1. Javadoc Kommentar.
2. Klassen oder Interface Deklaration.
3. Klassenattribute, *public, protected, package, private*.
4. Instanzattribute, *public, protected, package, private*.
5. Konstruktoren, *public, protected, package, private*.
6. Methoden, in einer logisch sinnvollen Reihenfolge.

Attribute Attribute werden am Anfang der Spezifikation einer Klasse geschrieben, beginnend mit den Klassenattributen.

1. Attributnamen beginnen mit einem Kleinbuchstaben. An Wortgrenzen wird ein Großbuchstabe verwendet (lower camel case).
2. Namen für Konstanten werden ausschließlich aus Großbuchstaben (SCREAMING_SNAKE_CASE) gebildet. Dies gilt auch für enums, wie etwa in[1]

```
public enum Ampel {
   GRÜN, GELB, ROT;
}
```

3. Attribute sind in der Regel *private* oder *protected*.
4. Wenn möglich, verwende man als Typ ein Interface.

Methoden

1. Methodennamen beginnen mit einem Kleinbuchstaben. An Wortgrenzen wird ein Großbuchstabe verwendet (lowerCcamelCase).

[1] Der Google Styleguide [Goo] fordert Zeichen nur aus dem ASCII Code.

2. Methodennamen werden aus Verben hergeleitet.
3. Methoden können nicht alle private sein. Aber es ist immer einfacher ein Element „sichtbarer“ zu machen als „unsichtbarer“.

Typparameter „Kernige“ Namen, möglichst ein Buchstabe. Üblich sind:

E Typparameter für generische Container-Klassen.

K Typparameter für Schlüssel in generischen Maps.

V Typparameter für Werte in generischen Maps.

X Typparameter für beliebige Exception-Typen.

T Typparameter, wenn andere Kriterien nicht greifen, z. B. in generischen Methoden.

Lokale Variablen, Parameter Namen von lokalen Variablen und Parametern werden in lower Camel Case gebildet.

Das einzige Element einer Single element Annotation heißt *value*.

A.5 Methoden - Stilfragen

Operations- und Methodennamen sind oft Verben oder werden aus solchen abgeleitet. Die Namen werden in lowerCamelCase gebildet. Namen von Parametern und lokalen Variablen werden in lowerCamelCase gebildet.

lesende, verändernde Methoden

1. Operationen mit einem anderen Rückgabetyp als *void* lesen nur und verändern das Objekt nicht.
2. Operationen, die ein Objekt verändern, haben den Rückgabetyp *void*.

Diese Regel ist sehr empfehlenswert und weit verbreitet [Mey97]. Es gibt aber auch Situationen in denen dagegen aus guten Gründen verstoßen wird. Überlegen Sie bitte sorgfältig, wenn Sie gegen diese Regel verstoßen und begründen Sie bitte derartige Verstöße!

return statements Eine Methode mit Rückgabe sollte genau ein *return*-Statement haben. Es gibt Ausnahmen, in denen die Lesbarkeit gegen dieses Kriterium gewinnt, aber die sind eher selten.

boolean Operationen, die boolean zurückliefern, also die Frage beantworten, ob eine Eigenschaft vorliegt oder nicht, heißen *isSomething*, z. B. *isEmpty* oder *isFull*.

Jede Klasse erbt *equals* von *Object*. Implementiert sie auch das Interface *Comparable*, so muss sie die Methode *compareTo* konsistent mit *equals* implementieren.

compareTo Die einzige Methode des Interfaces *Comparable* muss konsistent mit *equals* implementiert werden. Dies geht z. B. so: Zunächst wird die Methode *comparTo* implementiert.

equals kann für Klassen, die *Comparable* implementieren, am einfachsten so implementiert werden (Die Klasse heiße *Clazz*):

```
public boolean equals(Object obj){
   return obj instanceof Clazz ? this.compareTo((Clazz)obj)==0:false;
}
```

So ist die Kompatiblität mit *equals* gewährleistet. Ein Beispiel finden Sie in *crash.CounterV12*. Aus Gründen der Performance wird dies manchmal in mehreren Schritten getan, von denen der hier gezeigte dann nur der letzte ist.

hashCode Wird *equals* überschrieben, so muss auch *hashCode* konsistent überschrieben werden.

statische Fabrikmethoden : Für diese haben sich zwei Namen als oft verwendet und leicht verständlich herausgestellt:

valueOf Beispiele hierfür liefern die Methoden *valueOf* Wrapper-Klassen der primitiven Typen und der Klasse *String*.

instance() um das einzige Objekt einer Singleton-Klasse zu bekommen ([GHJV95], Ruby Module Singleton) oder auch *getInstance*, wie in StackWalker.

Variablen Für lokale Variable und Parameter gelten die gleichen Namenskonventionen wie für Attribute: *lowerCamelCase*.

A.6 Vererbungshierarchien

Für die Bildung von Vererbungshierarchien gibt es Faustregeln.

1. Vererbungshierarchie sollen möglichst flach gehalten werden. Dies steht in Übereinstimmung mit den Erkenntissen der Organisationslehre.

2. Nur auf der untersten Ebene der Vererbungshierarchie sollten die Klassen konkret sein. Dies ist in Java nur durchzuhalten, wenn man die Klasse Object ausnimmt, oder die Regeln auf die Klassen beschränkt, die explizit mittels *extends* angegeben werden.

Sie werden aber viele Beispiele sehen, in denen aus mehr oder weniger guten Gründen gegen diese Regel verstoßen wird.

A.7 Interfaces

Deklaration Die Sichtbarkeit von Operationen (Elementen) eines Interfaces ist immer *public*. Die Sichtbarkeit wird deshalb nicht explizit angegeben.

Attribute Attribute in Interfaces sind automatisch *static final* und es ist durchaus üblich dies nicht explizit zu deklarieren, wie auch die Sichtbarkeit.

Verwendung Attribute, Variablen und Rückgabetypen werden mit dem Typ eines Interfaces deklariert, wenn dies möglich ist.

Container De Empfehlung der Deklaration mit dem Typ gilt insbesondere für die *Collection Classes*.

A.8 Lokale Variablen

Lokale Variablen werden möglichst unmittelbar vor der ersten Verwendung deklariert und initialisiert. In Java ist es üblich lokale Variablen wie Attribute am Anfang eines Blocks zu deklarieren und zu initialisieren. Rein formal lässt sich diese Regel immer einfach durch ein weiteres Paar geschweifter Klammern einhalten. Benötigte lokale Variablen können also ohne Probleme am Anfang eines Blocks deklariert und initialisiert werden. Sie müssen auch vom Programmierer initialisiert werden, während primitive Attribute einer Klasse mit mehr oder weniger sinnvollen Standardwerten automatisch initialisiert werden

Sind die Methoden gut faktorisiert, so geht der Code einer Methode nicht über eine Bildschirmseite hinaus. Nach meinen Erfahrungen schreiben die Studenten mit dem kleineren Bildschirm besseren Code, als die mit einem großen Bildschirm.

A.9 Kommentare

Über Kommentare gehen die Ansichten weit auseinander. Das Ziel sollte immer sein, Code zu schreiben, der ohne Kommentare verständlich ist. Trotzdem gibt es gute Gründe für manche Kommentare. Ich gebe hier aber nur einige wenige an.

1. In jede Klasse gehört ein Javadoc-Kommentar, der den Zweck der Klasse nennt und den oder die Programmierer/in mittels @author angibt.
2. Eine öffentliche Methode benötigt unbedingt einen aussagefähigen Javadoc-Kommentar, wenn die Klasse im Rahmen einer Bibliothek (ohne Sourcecode) ausgeliefert wird.

A.10 Historische Anmerkungen

Die Auffassung über *guten* Programmierstil veränderten sich und verändern sich weiter. So vertrat man in den sechziger und siebziger Jahren des letzten Jahrhunderts die Ansicht, dass alle Deklarationen und Initialisierungen von Variablen in einem Codeteil zusammengefasst werden sollten.

Während man heute meist modularisierten Code mit kleinen Modulen empfiehlt, wurde in den achtziger Jahren des letzten Jahrhunderts etwa für die IBM /38 empfohlen, Programme so auszulegen, dass sie die maximale Größe von 64K erreichten. Davon versprach man sich eine effizientere Hauptspeicherauslastung.

Die hier beschriebenen Konventionen sind zur Zeit in vielen Programmiersprachen üblich.

A.11 Aufgaben

1. Welche der folgenden Klassennamen sind gemäß diesen Konventionen gebildet? Was lässt der Name erkennen? Begründen Sie bitte Ihre Ansicht!
   1.1. EierlegendeWollmilchSau
   1.2. Kunde
   1.3. Lieferanten
2. In welcher Reihenfolge werde die Elemente in Java-Klassendateien deklariert?
3. Wie werden die Namen von Klassen in Java gebildet?
4. Wie werden die Namen von Interfaces in Java gebildet?
5. Wie werden die Namen von Attributen in Java gebildet?
6. Wie werden die Namen von Variablen in Java gebildet?
7. Wie werden die Namen von Parametern in Java gebildet?
8. Wie werden die Namen von Typ-Parametern in Java gebildet?
9. Wie werden die Namen von Konstanten in Java gebildet?
10. Wie werden die Namen von Paketen in Java gebildet?
11. Wie werden die Namen von Fabrikmethoden in Java gebildet?

Anhang B

Eclipse

B.1 Übersicht

Dieses Kapitel gibt eine pragmatische Einführung in Eclipse. Ich stelle hier die Dinge zusammen, die sich für mich bewährt haben und die ich den Studierenden in den Veranstaltungen Programmierung, Algorithmen und Datenstrukturen und Software-Engineering empfehle.

B.2 Lernziele

- Wissen, woher man Eclipse beziehen kann.
- Eclipse installieren können.
- Projekte in Eclipse verwalten können.
- Plugins in Eclipse integrieren können.
- Fehlermeldungen und Warnungen verstehen können.
- Hinweise (z. B. Quickfix) umsetzen können.
- Systematisch mit JUnit testen können.

B.3 Download und Installation

Sie erhalten Eclipse über www.eclipse.org. Ich empfehle für Praktikumsaufgaben die Version aus dem AIL zu verwenden. Aber Sie sollten sich auch mit der jeweils aktuellen Version vertraut machen. Ich verwende zur Zeit (28.01.2018) Eclipse Oxygen (4.7.2) in dewr IDE und EE Version. Zum Installationsumfang empfehle ich Folgendes:

1. Installation von Java für Ihr jeweiliges Betriebssystem (z. B. von java.com).
2. Installation der Version von Eclipse für Ihr jeweiliges Betriebssystem. Ob Sie Classic oder EE wählen, ist zunächst nicht entscheidend.
3. Installation des Java Source-Codes (finden Sie ebenfalls unter der angegeben Java URL). In der dortigen readme-Datei finden Sie die Anleitungen hierfür. Bei mir stand er nach Installation von Eclipse IDE 4.7.2 mit Java 9 direkt zur Verfügung.
4. Wollen Sie auch offline arbeiten, sollten Sie sich auch die API-Dokumentation lokal installieren.

5. Zu Ihren Java Projekten sollten Sie unter Projekt→Eigenschaften→Java Build Path unter Libraries zum Classpath JUnit hinzufügen.

Achtung: So mir weitere Hinweise bekannt werden, die Sie bei der Installation beachten sollten, gebe ich diese in der Vorlesung. Sie ändern sich aber zu häufig, als dass ich Sie nicht hier festhalten mag.

Nach diesen Schritten sollten Sie eine funktionsfähige Entwicklungsumgebung haben.

B.4 Projekt und Einstellungen

Starten Sie Eclipse, so müssen Sie einen Workspace auswählen. Ich habe z. B. einen Workspace für die studentischen Lösungen jeder Veranstaltung pro Semester. Außerdem habe ich für meine Java-Programmiervorlesungen einen Workspace für alle Beispielprogramme.

Sobald Sie einen Workspace angelegt haben, empfehle ich, vor Anlegen des ersten Projekts, die Einstellungen vorzunehmen, die ich in Abschn B.10 vorzunehmen. Diese werden zunächst für jedes Projekt übernommen, Sie können sie aber pro Projekt an Ihre jeweiligen Anforderungen anpassen.

Legen Sie in Eclipse ein neues Projekt an, so werden Ihnen verschiedene Möglichkeiten angeboten.

1. Sie können ein neues Projekt im aktuellen *Workspace* anlegen oder ein Projekt aus bestehenden Java-Dateien erstellen. Haben Sie schon programmiert und wollen jetzt mit Eclipse weiterarbeiten, so kommt die letztere Möglichkeit in Frage.

   Für die Praktikumsaufgaben in meinen Veranstaltungen bitte ich Sie jeweils *ein* Projekt für das Praktikum anzulegen.

2. Unter Project→Properties können Sie einige wichtige Einstellungen vornehmen. Ich weise vor allem auf den Java Build Path hin.

   - Als JRE (Java Runtime Environment) rate ich zu Folgendem: Für Ihre selbstständige Arbeit, in der Sie frei entscheiden können, rate ich zur jeweils aktuellsten JRE. Für die bereits oben erwähnten Praktika verwenden Sie sicherheitshalber die gerade verwendete. Fügen Sie unter *Libraries* mittels *Add Library* JUnit hinzu. Ich empfehle dringend JUnit, wie es in Eclipse Standard ist.

3. Für die Organisation Ihrer Dateien haben Sie vor allem zwei Optionen:

   - Sie können alle Dateien direkt in Ihrem Projektordner speichern.
   - Sie speichern Quellcode-Dateien (.java) in einem *src* (Source, Quell-)Ordner und Java-Bytecode (.class) in einem anderen, z. B. *bin*-Verzeichnis.
   - Für (javadoc-) Dokumentation wird ein weiterer Ordner *doc* angelegt.
   - Für das Praktikum legen Sie bitte ein Projekt an, in dem *src* und *bin* Verzeichnis für *.java* bzw. *.class*-Dateien angelegt wird. Wenn Sie beim Exportieren Ihres Lösungsvorschlags die richtigen Dateien auswählen, ist dies aber zweitrangig.

Die weiteren Optionen besprechen wir, wenn sie benötigt werden. Einige finden Sie später in diesem Kapitel.

B.5 Erste Schritte

Nachdem Sie nun eine funktionsfähige Java und Eclipse-Umgebung haben, hier nun einige Hinweise für das praktische Arbeiten.

Neue Elemente legen Sie an, in dem Sie im Menu *File* (*Datei*) den Punkt *new* oder im jeweiligen Element die rechte Maustaste drücken. Als Erstes legen Sie sich ein *Paket*, englisch *package* an. Tun Sie dies auf jeden Fall! Verwenden Sie bitte *nicht* das *default*-Package!

Innerhalb eines *package*s können Sie nun weitere Elemente ebenfalls mittels *new* anlegen. Zunächst einmal interessant sind Interfaces und Klassen. Die weiteren Optionen werden Sie im Rahmen der Arbeit mit Eclipse und der Programmierausbildung kennenlernen.

Interface Hierfür wählen Sie nach *new* die Option *Interface* aus. Haben Sie die rechte Maustaste über einem Paket (package) gedrückt, so müssen Sie einen Namen angeben. Wenn Ihr Interface ein anderes erweitern soll, so können, Sie die ebenfalls hier unter *extended Interfaces* angeben. Wie fast immer in Eclipse, fangen Sie in letzterem Fall einfach an, den Namen des Superinterfaces einzutippen. Sobald Sie den Namen des gesuchten Interfaces sehen, können Sie diesen auswählen.

Klasse Hierfür wählen Sie nach *new* die Option *class*. Hier haben Sie eine Reihe von Optionen zur Auswahl:

Name Klassennamen beginnen mit einem Großbuchstaben (upper camel case).

Superclass Hier geben Sie die direkte Oberklassen an. Oft werden Sie den default *java.lang.Object* stehen lassen können.

Interfaces Wie bei Interfaces können Sie auch bei Klassen Interfaces auswählen, die von Ihrer Klasse implementiert werden müssen. Die Auswahl ist zunächst leer, mittels Klick auf „add“ können Sie ein oder mehrer Interfaces hinzufügen. Verklicken Sie sich in der Folge nicht irgendwo, so werden Ihnen dann die entsprechenden Methodenrümpfe automatisch generiert.

method stubs Hier können Sie einige weitere Methoden generieren lassen, z. B. wenn Sie Methoden aus einer Oberklasse überschreiben wollen. Eine *main*-Methode brauchen Sie für den Start einer Anwendung oder für einfache Tests. Ersteres brauchen Klassen selten und Letzteres machen Sie aber besser mit *JUnit* Die restlichen Optionen können Sie später kennen lernen.

Wenn Sie nun eine Datei in Eclipse angelegt haben, schreiben Sie Java-Code. Eclipse unterstützt Sie dabei durch verschiedene Dinge:

1. Konnten Sie bereits Methodenrümpfe (Siehe die Tags *// TODO Auto-generated method stub*) generieren lassen, so rate ich Ihnen als erstes einen JUnit Testfall zu schreiben. Einige grundlegende Informationen zu JUnit finden Sie in Abschn. B.6.

2. Automatische Ergänzung: Tippen Sie irgendetwas ein und drücken dann, *strg+space*, so erhalten Sie Vorschläge für sinnvolle, syntaktisch korrekte Ergänzungen.

3. Haben Sie etwas geschrieben, das syntaktisch potenziell oder sicher falsch ist, wird die entsprechende Zeile mit einem gelben Ausrufungszeichen (Warnung) oder Kreuz in rotem Kreis (Fehler) gekennzeichnet. Steht da auch noch eine *Lampe*, so kann Ihnen Eclipse einen Hinweis auf mögliche Korrekturen geben (*Quickfix*). Ein häufiger Quickfix besteht einfach im importieren der fehlenden Klassen oder Interfaces. Dies erreichen Sie oft auch einfach, in dem Sie in Ihrer Source-Codedatei die rechte Maustaste klicken und dort *Source→Organize Imports* (strg+↑+O) auswählen.

4. Sie sollten sich auf jeden Fall die Meldungen im Tab *Problems* ansehen. In meinen Veranstaltungen erwarte ich Lösungen ohne Warnungen!

5. Außerdem weise ich bereits jetzt auf die Technik des Refactoring hin. Eclipse hat einige Refaktorisierungen (refactorings) fertig implementiert. Am häufigsten werden Sie sicher *rename* verwenden, um Namen von Elementen zu verbessern oder einfach irritierende Schreibfehler zu beheben.

6. Sie können in Eclipse eine Rechtschreibprüfung auch auf Deutsch verwenden. Ich habe mit einem Wörterbuch mit 10.000 Einträgen begonnen und dies bei Bedarf ergänzt. Das Original

kam von www.wortschatz.uni-leipzig.de/Papers/top10000de.txt. Sie stellen dies in Eclipse unter Window→ Preferences → Editors→ Text Editors→ Spelling ein.

Für Deutsch habe ich zu letzt ein Wörterbuch für Eclipse verwendet, das ich hier bezogen habe:

http://freefr.dl.sourceforge.net/project/germandict/german.7z.

Inzwischen habe ich aber entschieden neuen Code ausschließlich englisch zu schreiben und zu kommentieren. Ich verwende amerikanisches Englisch.

7. Was Sie im Consolen-Fenster in Eclipse bei der Ausgabe sehen, hängt ebenfalls von den Einstellungen ab. Neben Schriftgröße und Schriftart ist der Zeichensatz u. U. interessant: Unter *RunAs → Run Configurations* wählen Sie den Tab *Common* aus und stellen z. B. UTF-8 ein. Dann sehen Sie auch Unicodezeichen wie `\u221E` als ∞ und nicht als `Infinity`.

8. Wollen Sie Schriften lesbar ausgeben, die *double byte character support* benötigen, so setzen Sie UTF-16 statt UTF-8.

Um Ihren Code auszuführen klicken Sie mit der rechten Maus-Taste auf das Element und wählen aus dem Pop-up Menu die gewünschte Option aus. Für Sie wird diese oft *Java Application* oder *JUnit Test* sein. Wollen Sie Parameter mitgeben, so wählen Sie *Run Configurations*. Dort haben Sie unter *(x) = Arguments* einen Tab, in dem sie das tun können. Die Parameter werden einfach durch einen oder mehrere Blanks getrennt.

B.6 JUnit

Eine aktuelle Version von *JUnit* erhalten Sie bereits mit *Eclipse*. Unter www.junit.org finden Sie weitere Informationen. Um mit *JUnit* produktiv zu arbeiten, brauchen Sie keine weiteren Vorkenntnisse. Alles was Sie zum Start brauchen, passt auf diese bzw. die folgende Seite.

Um einen Testfall für eine Klasse anzulegen markieren Sie am einfachsten eine Klasse im *Package Explorer* von Eclipse, drücken die rechte Maustaste, wählen *new* und dann *JUnit Test Case*. Ihr Testfall hat nun bereits einen Namen. Der Default-Name ist *KlassenNameTest*. Gibt es überladene Methoden, so werden die Typen der Parameter an den Namen der Testmethode angehängt.

Bitte drücken Sie jetzt noch *nicht finish*!

Überlegen Sie sich, was Sie testen wollen. Wenn Sie z. B. vor jedem Test einen bestimmten, immer gleichen, Ausgangszustand herstellen wollen, so kreuzen Sie bitte *setUp()* an. Die anderen Optionen können Sie später nach Bedarf nutzen.

Anschließend wählen Sie dann *next*. Im nun folgenden Dialogschritt können Sie auswählen, welche Methoden Sie testen wollen. Angeboten werden alle Methoden aus Ihrer Klasse und die von Oberklassen bzw. Interfaces ererbten.

Haben Sie sich für eine *setUp*-Methode entschieden, so sollten Sie dort die entsprechenden Testdaten für die zu testenden Methoden verfügbar machen. Oft wird es sinnvoll sein, entsprechende Attribute in der Testklasse zu deklarieren und in der setUp-Methode zu initialisieren. Die größte Flexibilität erreichen Sie, wenn Sie die Testdaten jeweils aus einer Datei einlesen können.

Haben Sie die Voraussetzungen hierfür geschaffen, so können Sie sich überlegen, was Sie testen wollen. Überlegen Sie sich sinnvolle Eingaben und das Ergebnis, dass nach Spezifikation herauskommen sollte. Ob dies auch so erfolgt, können Sie mit einer Reihe überladener Operationen überprüfen:

assertEquals (Object erwartet, Object tatsächlich) Diese Operation ist u. a. für long, double, String überladen.

assertArrayEquals (Object [] erwartet, Object [] tatsächlich) Diese Operation ist u. a. für die numerischen primitiven Typen überladen.

assertTrue Prüft ab, ob eine Bedingung wahr oder falsch ist und ist erfolgreich, wenn die Bedingung *true* ist.

assertFalse Analog zu assertTrue.

assertThat (T tatsächlich, org.hamcrest.Matcher<T> matcher) Prüft, dass „tatsächlich“ den Bedingungen des „matcher“ genügt.

Haben Sie dies getan, so können Sie mittels *run as→JUnit Test* ausführen. Je nach dem, ob die Zusicherungen erfüllt sind oder nicht, zeigt JUnit Ihnen einen grünen oder roten Balken.

Bemerkung B.6.1 (assertTrue vs assertEquals)
Wenn Sie Werte vergleichen wollen, haben Sie die Wahl zwischen *assertTrue(a.equals(b))* o. ä. und *assertEquals(a,b)*. Ein kleiner Vorteil der zweiten Variante besteht darin, dass Sie dann im Fehlerfalle die Werte angezeigt bekommen. Das kann Ihnen in manchen Fällen schon Hinweise auf die Ursache geben, etwa bei Rundungsfehlern. Beim Vergleich von Fließkommazahlen müssen Sie eine akzeptierte Toleranz angeben. ◀

Um zu testen, ob eine *Exception* (genauer ein *Throwable*), die geworfen werden soll, auch tatsächlich geworfen wird, verwenden Sie den Parameter expected der Annotation @Test. Natürlich kann *JUnit* noch mehr, aber diese Informationen reichen wahrscheinlich zum Anfang völlig aus. Etwas mehr finden Sie in Anhamg C.

B.7 Javadoc

Für Einzelheiten zu Javadoc verweise ich auf Kap. 13. Um aus den Javadoc-Kommentaren erzeugt javadoc einen Satz von verlinkten HTML-Seiten. In Eclipse rufen Sie javadoc über die Export-Funktion auf. Dort wählen Sie *Javadoc* aus. Beim ersten Mal müssen Sie noch angeben, wo Eclipse javadoc findet bzw. welche Version von javadoc Sie verwenden wollen. Außerdem müssen Sie das Zielverzeichnis angeben. Üblich ist ein Verzeichnis *doc* in Ihrem Projekt, also auf der gleichen Ebene, wie *src* und *bin*.

B.8 jar-Dateien

Ein *jar*-Datei ist zunächst einmal einfach eine zip-Datei. Sie werden zunächst jar-Dateien für die JUnit Tests kennenlernen, die ich für einige Aufgaben vorgeben werde. *jar*-Dateien werden in Eclipse über die Export-Funktion erstellt. Sie können mit dieser Funktion auch die Abgaben für die Praktikumsaufgaben erzeugen. Ich habe das Format *zip* vorgegeben, da das einigen vielleicht schon bekannt sein könnte.

Wichtig sind für Sie zunächst *jar*- und *executable jar*-Dateien. Für letztere müssen Sie noch eine *Manifest*-Datei erzeugen. Übernehmen Sie zunächst einmal die default-Einstellungen, die Eclipse hier vorgibt.

Wenn Sie einfach nur Source-Code aus einem Projekt exportieren wollen, so exportieren sie einen Archive File. Für Sourcen, die Sie im Praktikum abgeben wollen, bitte ich Sie nur das Paket und seine Inhalte zu exportieren. Das erreichen Sie durch die Auswahl „Create only selected directories“.

B.9 Nützliche Tastaturkürzel und andere Abkürzungen

Java Editor	
alt+enter	Properties
alt+shift+r	rename
strg+space	vervollständigen
syso+strg+space	System.out.println()
strg+alt+↑	kopieren des Bereichs nach oben[a]
strg+alt+↓	kopieren des Bereichs nach unten
strg+7	Zeile, Bereich ein/aus kommentieren
strg+d	Zeile löschen
strg+s	Aktuelle Datei speichern
strg+shift+s	Alles speichern
strg+shift+f	Source formatieren
strg+shift+O	Organize Imports
shift+F2	Java API Dokumentation
F3	Java Source
Debug/Run	
strg+F11	run last launched
alt+shift+X, T	Run as JUnit Test
strg+shift+D, Z	Debug as JUnit Test
alt+shift+X, Z	Run Ruby Script
F5	step into
F6	step over

[a] Sie müssen eventuell Abkürzungstasten (Hotkeys) für den Desktop deaktivieren.

B.10 Konfigurationen

Basis der Verwaltung Ihres Source-Code in Eclipse ist der Workspace. Viele Eigenschaften können Sie auf der Ebene des Workspaces konfigurieren. Ich berichte hier über einige, die ich meistens vornehme. Dazu wählen Sie Window→Preferences aus.

1. Als Zeichensatz wähle ich UTF-8. Unter General→Workspace können Sie dies auswählen. Als Zeilentrennzeichen wähle ich UNIX. Diese Werte werden dann für alle Projekte (siehe unten).

2. Unter General→Editors→Text Editors kreuze ich „Insert spaces for tabs“. Dadurch werden beim Einfügen von Tabs stattdessen Spaces eingefügt. Unter Java→Code Style→Formatter lege ich mir ein eigenes Profile an, um das default Profil nicht zu verändern. Hier wähle ich im Tab Indentation die Tab Policy Only Spaces aus. Das ist für mich bequem, wenn ich Code in eine Folie oder ein Skript übernehme. Auch der Google Style Guide [Goo] empfiehlt dies so.

3. Unter General→Web Browser wähle ich einen externen Webbrowser und belasse es beim default Webbrowser.

4. Verwende ich den Workspace in einer Veranstaltung so vergößere ich die Schriftgrößen geeignet. Das geht in Eclipse seit Längerem einfach mit *Strg +* bzw. *Strg* -, wie in vielen Browsern.

5. Von den Einstellungen für den Java Compiler verändere ich nur wenige.

   Method with a constructor name Hier stelle ich „Warning“ ein.

   Resource not managed with try-with-resource Hier stelle ich „Warning“ ein.

   Warning „@Override“ annotation Hier stelle ich „Warning“ ein.

Value of method parameter is not used: Hier gebe ich „Warning“ an und ignoriere dies weder bei überschreibenden bzw. implementierenden Methoden noch bei mittels „@param“ dokumentierten Parametern. So werde ich darauf hingewiesen, wenn ich etwas vergessen oder einen überflüssigen Parameter definiert habe.

6. Haben Sie den jdk intalliert, so bekommen Sie auch gleich den Source-Code mitgeliefert. Auf diesen können Sie einfach zugreifen, indem Sie die Taste F3 drücken, wenn Sie auf einem dokumentierten Element stehen.

7. Die Java API-Dokumentation ist per default online verfügbar, Sie können Sie aber auch pro Projekt lokal verfügbar machen.

Innerhalb eines Workspaces legen Sie Projekte an. Wollen Sie Code aus einem Projekt *Base* in einem anderen Projekt nutzen, so geben Sie das Projekt *Base* unter den Eigenschaften des Projekts als Project Reference an. Die auf der Workspace-Ebene definierten Eigenschaften können bei Bedarf pro Projekt verändert werden.

B.11 Historische Anmerkungen

Die Eclipse-Initiative wurde im November 2001 gegründet. Von IBM iniziiert, traten bis Ende 2002 auch HP, Oracle, SAP bei.

B.12 Aufgaben

1. Legen Sie bitte in Eclipse ein Java-Projekt an, in dem Source-Dateien (.java) in einem Verzeichnis src und Byte-Code Dateien (.class) in einem Verzeichnis bin gespeichert werden.

Anhang C

JUnit

C.1 Übersicht

JUnit ist ein *Framework* für Komponententest (Unit-Tests). Für andere Sprachen gibt es ähnliche Werkzeuge, etwa für *Smalltalk*, *C++*, *Ruby* u. a. In Java baut es auf Annotationen auf. Für seine Benutzung müssen Sie aber nicht viel über Annotationen wissen. Es ist überflüssig den Useguide (https://junit.org/junit5/docs/current/user-guide/) zu reproduzieren. Aber einige Hinweise sind m. E. nützlich.

C.2 Lernziele

- Aus einer Klasse einen JUnit Testfall (test case) erzeugen können (mit Eclipse).
- Den Ablauf eines JUnit Testfalls kennen und beschreiben können.
- Verschiedenen Varianten der *assert*-Methoden verwenden können.
- Testfälle ausführen können.

C.3 Einführung

Mit JUnit testen Sie Methoden von Klassen. Oft werden Sie eine Testklasse pro zu testender Klasse haben. Das ist aber nicht zwingend und nicht immer sinnvoll. Zum Testen brauchen Sie:

1. Eine Testumgebung aus korrekt initialisierten Objekten. Diese wird manchmal *Fixture* genannt.
2. Werte für die Parameter der zu testenden Methoden.
3. Korrekte Werte für die Rückgaben der zu testenden Methoden.
4. Gegebenenfalls die Exceptions, die von den zu testenden Methoden geworfen werden sollen.

Dieses alles zusammen definiert Ihre Testfälle.

Alle Methoden der Testfälle sind parameterlos und haben Rückgabetyp *void*. Einige sind *static*.

Testfälle können zu TestSuites zusammengefasst werden. Hierfür verweise ich auf die Dokumentation. Für die Testautomatisierung sind diese weiterhin wichtig. Arbeiten Sie in Eclipse, so können Sie aber auch ohne eine TestSuite alle Testfälle in eine Paket auf einmal mittels *run as JUnit Test Case* ausführen.

C.4 Annotationen

JUnit steuert die Tests über *Annotation*en. Alle JUnit-Annotationen sind Marker-Annotationen. Die *@Test*-Annotation hatte für JUnit4 zwei Parameter, die inzwischen entfallen sind. Der Parameter *expected*, mit dem Sie eine erwartete Exception spezifizieren konnten ist sozusagen ersetzt worden durch *assertThrows* ersetzt worden. Diese Parameter sind also optional. Alle Annotationen bis auf *@Ignore* sind Annotationen für Methoden, nur *@Ignore* kann auch für einen *TYPE* (meist eine Klasse) verwendet werden.

Annotation	Erläuterung
@BeforeClass	Annotation für statische Methoden, die vor dem Ausführen aller Testfälle ausgeführt werden sollen.
@Before	Methoden mit dieser Annotation werden vor jedem Testfall ausgeführt. Sie stellen also die „Fixture“ für jede Testmethode wieder her.
@Test	Methoden, die die eigentlichen Tests enthalten. @Test hat die beiden Parameter • expected: Hier geben Sie die Exceptions an, die in den Testfällen erwartet werden: `expected = ExceptionName.class`. Der default ist *None*, eine in der Annotation definierte Dummy Exception. • timeout: Hier geben Sie eine maximale Laufzeit der Methode in Millisekunden an. Wird diese überschritten, so ist der Test fehlgeschlagen. Der default ist $0L$, d. h. keine Einschränkung der Laufzeit.
@Ignore	Markiert einen Testklasse oder eine Testmethode, die (temporär) nicht ausgeführt werden soll.
@After	Methoden mit dieser Annotation werden nach jedem Testfall ausgeführt und dienen der Freigabe externer Ressourcen, die während eines Tests oder in einer *@Before* Methode geschaffen wurden.
@AfterClass	Annotation für statische Methoden, in denen Ressourcen freigegeben werden, die in *@BeforeClass* Methoden angelegt wurden.
@ClassRule	Annotiert statische Attribute, die Regeln referenzieren oder statische Methoden, die sie zurückgeben.
@FixMethodOrder	Ermöglicht die Festlegung der Ausführungsreihenfolge von Testmethoden für eine TestCase Klasse: DEFAULT, JVM, NAME_ASCENDING

C.5 Testmethoden aus Assert

Die *JUnit* Testmethoden sind statische Methoden der Utility-Klasse *org.junit.Assert*. Sie haben einen *Boolean* Parameter oder zwei bis drei Parameter. In den letzteren Fällen enthält der erste Parameter das erwartete Ergebnis (expected) und der zweiter das tatsächliche Ergebnis (actual). Diese Methoden sind mehrfach überladen. Es gibt Sie für die primitiven Typen und für *Object*.

Methode	Erläuterung

assertEquals	`assertEquals(Object expected, Object actual)`
	`assertEquals(String message, Object expected, Object actual)`
	`assertEquals(long expected, long actual)`
	`assertEquals(String message, long expected, long actual)`
	`assertEquals(double expected, double actual, double delta)`
	`assertEquals(float expected, float actual, float delta)`
	`assertEquals(String message, double expected, double actual, double delta)`

assertArrayEquals	`assertArrayEquals(boolean [] expecteds, boolean [] actuals)`

Methode	Erläuterung

	`assertArrayEquals(byte [] expecteds, byte [] actuals)`
	`assertArrayEquals(char [] expecteds, char [] actuals)`
	`assertArrayEquals(int [] expecteds, int [] actuals)`
	`assertArrayEquals(Object [] expecteds, Object [] actuals)`
	`assertArrayEquals(short [] expecteds, short [] actuals)`
	`assertArrayEquals(double [] expecteds, double [] actuals, double delta)`
	`assertArrayEquals(float [] expecteds, float [] actuals, float delta)`
	In den letzten beiden Fällen ist delta der maximale Absolutbetrag von expecteds[i] und actuals[i]
	`assertArrayEquals(String message, ...)` ... wie oben

assertNotEquals	`assertNotEquals(Object unexpected, Object actual)`
	`assertNotEquals(long unexpected, long actual)`
	`assertNotEquals(double unexpected, double actual, double delta)`
	`assertNotEquals(float unexpected, float actual, float delta)`
	`assertNotEquals(String message, Object unexpected, Object actual)`
	`assertNotEquals(String message, long unexpected, long actual)`
	`assertNotEquals(String message, double unexpected, double actual, double delt`
	`assertNotEquals(String message, float unexpected, float actual, float delta)`

assertFalse	`assertFalse(boolean condition)` `assertFalse(String message, boolean condition)`
assertNotNull	`assertNotNull(Object obj)` `assertNotNull(String message, Object obj)`
assertNull	`assertNull(Object obj)` `assertNull(String, Object obj)`

assertThat	`<T>assertThat(T actual, Matcher<? super T> matcher)`
	`<T>assertThat(String reason, T actual, Matcher<? super T> matcher)`
	Matcher ist ein Interface aus dem Paket org.hamcrest.

assertThrows	`assertThrows(Class<T> expectedType, Executable executable)`
assertTrue	`assertTrue(boolean condition)` `assertTrue(String message, boolean condition)`
assertSame assertSame	`assertSame(Object expected, Object actual)` `assertSame(String message, Object expected, Object actual)`
assertNotSame assertNotSame	`assertNotSame(Object unexpected, Object actual)` `assertNotSame(String message, Object unexpected, Object actual)`
fail	`fail()` `fail(String message)` Wird z. B. verwendet, wenn Sie Testfälle erzeugen lassen.

Die Methoden assertEquals etc. sind schon lange Bestandteil von *JUnit*. Die Methoden *assertThat* und die Matcher sind in JUnit 4.12 hinzugekommen. Ihre Verwendung gilt als der aktuelle Stil.

Bemerkung C.5.1 (expected und actual)
Achten Sie darauf, die Parameter *expected* und *actual* nicht zu vertauschen! Passen Sie dabei nicht auf, so suchen Sie vielleicht an der falschen Stelle nach einem Fehler! ◀

C.6 Historische Anmerkungen

Automatisierte Unit Tests wurden zunächst in Smalltalk populär. Weit verbreitet wurde dann JUnit. MIt JUnit 5 sind Matcher hinzugekommen (siehe *assertThat*

C.7 Aufgaben

Anhang D

Tabellen und Grenzen

D.1 Übersicht

Dieses Kapitel enthält einige Werte für häufig benötigte Ausdrücke, die ein Informatiker ohne langes Nachdenken wissen sollte.

D.2 Lernziele

- Die Werte einiger wichtiger Ausdrücke kennen.

D.3 Ganze Zahlen

D.4 Ausdrücke

Abbildung D.1 zeigt die Zweierpotenzen von 2^0 bis 2^{30} und die Abb. D.2

D.5 Fließkommazahlen

D.6 Historische Anmerkungen

Alle die hier aufgeführten Werte finden Sie auch anderswo, z. B. auch im Internet. Aber einige Dinge sollte ein Informatiker schon im Kopf haben. Da denke ich ähnlich konservativ wie Don Knuth, der entsprechendes auch in [Knuff] immer wieder im Anhang aufführt.

D.7 Aufgaben

i	Dezimal	Binär	Oktal	Hex
0	1	1	1	1
1	2	10	2	2
2	4	100	4	4
3	8	1000	10	8
4	16	10000	20	10
5	32	100000	40	20
6	64	1000000	100	40
7	128	10000000	200	80
8	256	100000000	400	100
9	512	1000000000	1000	200
10	1.024	10000000000	2000	400
11	2.048	100000000000	4000	800
12	4.096	1000000000000	10000	1000
13	8.192	10000000000000	20000	2000
14	16.384	100000000000000	40000	4000
15	32.768	1000000000000000	100000	8000
16	65.536	10000000000000000	200000	10000
17	131.072	100000000000000000	400000	20000
18	262.144	1000000000000000000	1000000	40000
19	524.288	10000000000000000000	2000000	80000
20	1.048.576	100000000000000000000	4000000	100000
21	2.097.152	1000000000000000000000	10000000	200000
22	4.194.304	10000000000000000000000	20000000	400000
23	8.388.608	100000000000000000000000	40000000	800000
24	16.777.216	1000000000000000000000000	100000000	1000000
25	33.554.432	10000000000000000000000000	200000000	2000000
26	67.108.864	100000000000000000000000000	400000000	4000000
27	134.217.728	1000000000000000000000000000	1000000000	8000000
28	268.435.456	10000000000000000000000000000	2000000000	10000000
29	536.870.912	100000000000000000000000000000	4000000000	20000000
30	1.073.741.824	1000000000000000000000000000000	10000000000	40000000

Abb. D.1: Die Zweierpotenzen von 2^0bis 2^{30} dezimal, binär, oktal und hex

31	2.147.483.648	10000000000000000000000000000000	20000000000	80000000
32	4.294.967.296	100000000000000000000000000000000	40000000000	100000000
33	8.589.934.592	100000000000000000000000000000000	100000000000	200000000
34	17.179.869.184	100000000000000000000000000000000	200000000000	400000000
35	34.359.738.368	100000000000000000000000000000000	400000000000	800000000
36	68.719.476.736	100000000000000000000000000000000	1000000000000	1000000000
37	137.438.953.472	100000000000000000000000000000000	2000000000000	2000000000
38	274.877.906.944	100000000000000000000000000000000	4000000000000	4000000000
39	549.755.813.888	100000000000000000000000000000000	10000000000000	8000000000
40	1.099.511.627.776	100000000000000000000000000000000	20000000000000	10000000000
41	2.199.023.255.552	100000000000000000000000000000000	40000000000000	20000000000
42	4.398.046.511.104	100000000000000000000000000000000	100000000000000	40000000000
43	8.796.093.022.208	100000000000000000000000000000000	200000000000000	80000000000
44	17.592.186.044.416	100000000000000000000000000000000	400000000000000	100000000000
45	35.184.372.088.832	100000000000000000000000000000000	1000000000000000	200000000000
46	70.368.744.177.664	100000000000000000000000000000000	2000000000000000	400000000000
47	140.737.488.355.328	100000000000000000000000000000000	4000000000000000	800000000000
48	281.474.976.710.656	100000000000000000000000000000000	10000000000000000	1000000000000
49	562.949.953.421.312	100000000000000000000000000000000	20000000000000000	2000000000000
50	1.125.899.906.842.624	100000000000000000000000000000000	40000000000000000	4000000000000
51	2.251.799.813.685.248	100000000000000000000000000000000	100000000000000000	8000000000000
52	4.503.599.627.370.496	100000000000000000000000000000000	200000000000000000	10000000000000
53	9.007.199.254.740.992	100000000000000000000000000000000	400000000000000000	20000000000000
54	18.014.398.509.481.984	100000000000000000000000000000000	1000000000000000000	40000000000000
55	36.028.797.018.963.968	100000000000000000000000000000000	2000000000000000000	80000000000000
56	72.057.594.037.927.936	100000000000000000000000000000000	4000000000000000000	100000000000000
57	144.115.188.075.855.872	100000000000000000000000000000000	10000000000000000000	200000000000000
58	288.230.376.151.711.744	100000000000000000000000000000000	20000000000000000000	400000000000000
59	576.460.752.303.423.488	100000000000000000000000000000000	40000000000000000000	800000000000000
60	1.152.921.504.606.846.976	100000000000000000000000000000000	100000000000000000000	1000000000000000
61	2.305.843.009.213.693.952	100000000000000000000000000000000	200000000000000000000	2000000000000000
62	4.611.686.018.427.387.904	100000000000000000000000000000000	400000000000000000000	4000000000000000

Abb. D.2: Die Zweierpotenzen von 2^{31}bis 2^{62} dezimal, binär, oktal und hex

Anhang E

Von Ruby zu Java

E.1 Übersicht

Ich gebe in diesem Kapitel eine kurze Übersicht über Unterschiede und Gemeinsamkeiten von *Ruby* und *Java*. Ich möchte Ihnen dabei helfen, Kenntnisse aus Ruby möglichst einfach nach Java zu transferieren und die gröbsten Fehler zu vermeiden.

Abbildung E.1 gibt einen Überblick über die Unterschiede bzw. Entsprechungen und damit

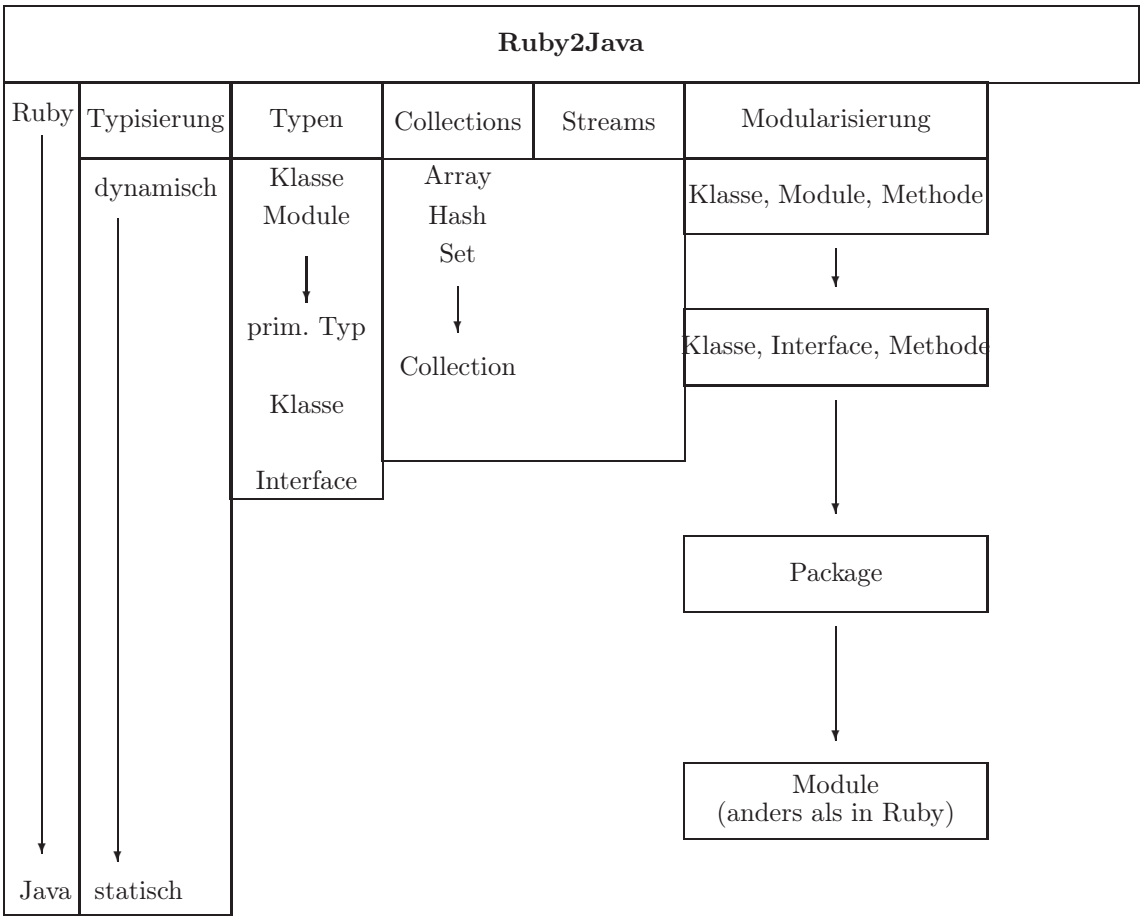


Abb. E.1: Ruby und Java — Übersicht

auch schon auf die Vorgehensweise, die ich bei der Darstellung plane.

E.2 Lernziele

- Die wichtigsten Unterschiede zwischen Ruby und Java kennen.
- Die wichtigsten Gemeinsamkeiten zwischen Ruby und Java kennen.
- Ruby-Kenntnisse nach Java transferieren können.

E.3 Grundlagende Unterschiede

Ohne Anspruch auf Vollständigkeit zunächst eine kurze Liste:

1. Ruby-Code wird zur Laufzeit interpretiert. Java-Code wird in Byte-Code compiliert, der dann zur Laufzeit interpretiert wird. Byte-Code ist aber näher an Maschinencode als Ruby-Code.
2. Ruby ist dynamisch typisiert, Java ist statisch typisiert.
3. In Ruby gibt es nur Objekte, in Java auch primitive Datentypen und generische Typen.
4. In Ruby erfolgt die Parameterübergabe per Referenz (call by reference). Ausnahmen sind Integer und einige Floats (call-by-value). In Java gibt es call-by-value für primitive Typen und Parameterübergabe per Wert (call-by-value) für Referenztypen.

E.4 Klassen und Typen

Ruby kennt nur Klassen und Objekte. Zusätzlich gibt es Module, die verschiedene Aspekte haben. In Java gibt es primitive Typen (boolean, char, int, long, float, double) und Referenztypen. Letztere sind Klassen oder Interfaces. Die Funktion von Interfaces übernehmen in Ruby zum Teil Module. Außerdem gibt es in Java generische Typen.

E.5 Tabellarische Übersicht

Ursprünglich diente mir die folgende Quelle als Muster: http://introcs.cs.princeton.edu/java/faq/c2java.html, zuletzt besucht am 25.12.2014. Von dem damaligen Muster für einen Vergleich von C und Java ist nicht mehr viel übrig geblieben, aber die Idee finde ich weiterhin ganz gut.

Thema	Ruby	Java
Allgemeines		
Art der Sprache	Interpretiert	Compiliert in Byte-Code, benötigt Laufzeitumgebung (JVM)
Paradigmen	objekt-orientiert, prozedural, funktional	objekt-orientiert, prozedural, funktional
Typisierung	dynamisch	statisch
Namen	case sensitive	case sensitive
Basiselement	Klasse, Module, Skript	Klasse, Interface, primitiver Typ, Module
Portierbarkeit (Source)	ja	ja
Portierbarkeit (Objectcode)	kein Objectcode	ja, Bytecode ist „write once, run anywhere“

Thema	Ruby	Java
Kommentare	# bis Zeilenende, rdoc generiert Dokumentation Mit YARD: tags ähnlich Javadoc	/* ... */, // bis Zeilenende /** ... */ hieraus generiert Javadoc Dokumentation
Befehlsende	newline, „;“z. B. bei Einzeilern	;
Sichtbarkeit (accessibility)	Für Methoden. public[:] Alle private[:] das Objekt protected[:] Objekte der Klasse und von Unterklassen	Für Klassen, Methoden, Attribute Für alle Elemente innerhalb einer Klassen außer anonyme public, alle private, alle Objekte der Klasse protected, alle Objekte von Klassen des Pakets und von Unterklassen default ist package (kein Keyword), alle Objekte von Klassen des Pakets
Attribute	immer private	üblicherweise private
Zugriff auf Attribute	attr_reader :attribut_name attr_writer: attribut_name= attr_accessor: attribut_name, beides	getAttributName() setAttributName(AttributTyp att) getter und setter
Methoden	Rückgabe boolean: name? z. B. empty? Gibt es eine lesende und eine verändernde: name und name! (bang)	Rückgabe boolean: isName, z. B. isEmpty n/a
Compiler	Interpreter	javac Hello.java erzeugt JVM Bytecode javac Main.java, alle abhängigen Dateien werden bei Bedarf neu compiliert.
Interaktive Ausführung	irb (InterActiveRuby)	jshell
Gruppierung von Elementen	Ordner, aber kein Namensraum, Module	Paket und Modul definiert Namensraum, Klasse, Interface ebenfalls

Klassendefinition	def KlassenName
	
	...
	end
		[public|abstract|final] class KlassenName {
		...
		}
		Für innere nicht-anonyme Klasse auch modifier private, static

Name der Klassendatei	Klassenname in (lower_)snake_case	Wie KlassenName
Class Object	Eines pro KlassenName, Monkey Patching möglich	Eines pro vollqualifiziertem Klassennamen.
Closure, funktionale Programmierung	Block, Proc, Lambda, Method(reference)	Lambda, Methodenreferenz
Mathematische Funktionen	Module Math	import [static] java.lang.Math
Zugriff auf Bibliothek	require tk	import java.io.File;
Verwendung Bibliotheksfunktion	require "math"	import static java.lang.Math.sqrt;

Thema	Ruby	Java
	x = sqrt(2.2);	x = sqrt(2.2); oder import java.lang.Math.sqrt; x = Math.sqrt(2.2);
Ausführung	Interpreter ausführen	Java führt Byte-Code aus

hello, world	`puts "Hello, World"`
	p
		`public class HelloWorld {`
		`... void main(String [] args) {`
		`System.out.println("Hello, World");`
		`}`
		`}`

Block	{...} für Einzeiler
	do ...
	end für Mehrzeiler
		{
		...
		}

Erzeugung von Objekten		
new	Klassenmethode der jeweiligen Klasse Initialisierung durch private Instanzmethode initialize	Operator Initialisierung durch Konstruktor
Anonyme Klasse	n/a	new InterfaceName...
Verkettung	Nicht direkt anwendbar, aber default Parameter	constructor chaining
Typen		
Ganze Zahlen	Integer Unbegrenzter Bereich kein unsigned Typ	int 32 bit 2'er Komplement BigInteger long 64 bit 2'er Komplement char, byte, short kein unsigned Typ, aber unsigned Arithmetik-Methoden Wrapperklassen Integer, Long etc. Math.addExact, Math.subtractExact
Fließkomma Zahlen		float: 32 bit IEEE 754
	Float 64 bit	double:64 bit IEEE 754
Mit hoher Genauigkeit	BigDecimal	BigDecimal
boolean type	Nein, stattdessen Klassen *FalseClass*, *TrueClass*	*boolean* ist primitiver Datentyp
character type	nein	*char* ist 16 bit Unicode
String	mutable Klasse String	immutable Klasse String 16 Bit Unicode StringBuilder, StringBuffer
Reguläre Ausrücke	Direkt unterstützt	Klasse Pattern
Array	Array	Interface List, implementierende Klassen
Array Deklaration	a = [1, 2, 3]	a = {1, 2, 3} int [] a = new int[n];
Array Größe	dynamisch, a.length Wie Java List	fest, a.length()
Datum/Uhrzeit	Module Time	LocalDate, LocalTime, LocalDateTime, ZonedDateTime,

Thema	Ruby	Java
		Period, Instant, Duration
Wrapperklassen	n/a	Für die primitiven Typen
	Literale	
Format ganzzahliger numerischer Literale	1_000_000	1_000_000 (dez), 0Xnnnn (hex), 0Bnnnn (binär), 0nn_mmm (oktal), \unnnn
Format Fließkomma Literale	12,345,67, 3.1415, 0.31415E1	3.1415, 0.31415E1
String Literal	double quoted "...", Stringinterpolation möglich.	"..."
	single quoted '...'	nur char
Comparable	Module Comparable	Interface Comparable
Iterierbar	Module Enumerable	Interface Iterable
Iterator	Klasse Enumerator	Interface Iterator
	Schleifen	
unbedingt	loop	-/-
for	for i in a..b (über Range)	for (int i = 0; i < n; i++) for(T t:anIterable) („for-each“)
each	Basisiterator each für Enumerable n.times,m.upto(n), n.downto(m), n, m Integer	Methode forEach von Iterable. n/a
while	vor- und nachprüfend auch als Statement Modifier	vorprüfend
until	Vor- und nachprüfend auch als Statement Modifier	do ... while nachprüfend
	Entscheidungen	
	if, auch als Statement Modifier	if
	if-else	if-else
	if-elsif-else	
	unless, auch als Statement Modifier	
	case - when - else	switch - case - default
	Ternärer Operator	Ternärer Operator
Variable		
Zulässige Zeichen	Grundsätzlich UTF	Grundsätzlich UTF Character.isJavaIdentifierStart(char c) Character.isJavaIdentifierPart(char c)
Gültigkeitsbereiche	Methoden und Variable sind lokal im Block	Methoden und Variable sind lokal im Block
Schreiben auf Standardausgabe	`puts`, `p`	`System.out.println("sum = " + x);` u. a.
Formattierte Ausgabe	`printf("avg = %3.2f", avg);`	`System.out.printf("avg = %3.2f", avg)`
Lesen von Standardeingabe	`scanf("%d", &x);`	Java library support, einfach zu verwenden, Scanner, Reader u a.
Speicheradresse	Referenz	Referenz
	Funktionen	
Bezeichnung	Methode	Methode
Definition in	Skript, Klasse, Module	Klasse, Interface

Thema	Ruby	Java
Methodenname	(lower_)snake_case	lowerCamelCase
Methodensignatur	def [self.]max a, b	public [static] int max(int a, int b)
Parameterübergabe	by-reference: Übergabe Kopie der Referenz, ∃ Ausnahmen	Primitive Typen: by-value: Übergabe Kopie des Werts Referenz-Typen: by-reference: Übergabe Kopie der Referenz
Definition Datenstruktur	class	class mit Attributen (fields) und Metnoden u. a.
	Struct	bean
Attribut	Attribut, Instanzvariable	Attribut, Feld (field)
Methode	Methode	Methode
Zugriff auf Elemente von Datenstruktur	obj.att_name für Elemente	obj.getAttributName für Attribut (field)
Methoden	obj.methode	a.methode
„Pointer Chasing“	x.left.right	x.left.right
allocating memory	Klassenmethode new, ruft initialize auf	new Operator ruft Konstruktor auf.
de-allocating memory	automatic garbage collection	automatic garbage collection
buffer overflow	Exception zur Laufzeit	checked run-time error exception
declaring constants	Erstes Zeichen Großbuchstabe, Warnung bei Veränderung, Konvention: SCREAMING_SNAKE_CASE	final, nur einmal zuweisbar, Konvention: SCREAMING_SNAKE_CASE
variable auto-initialization	nil	Attribute von primitivem Typ 0, 0.0, false, Referenztype *null*, Compiler-Fehler nicht initialisierte Variablen zu verwenden. Andere Variable nicht automatisch initialisiert.
interface Methode	Module Methode	public Methode in Interface
Interface	include (mixin)	implements
Generischer Datenyp	Object	aktueller Typparamter
casting	-/-	Compiler-Fehler oder checked Exception zur Laufzeit
Typeerweiterung/-verengung		Automatische Typerweiterung, int zu long etc., Typverengung: expliziter cast
Polymorphismus	Vererbung, include, extend, prepend class Clazz < OberKlasse...	Vererbung, extends, implements class Clazz extends OberKlasse... class Clazz implements EinInterface
mixin	Module	abstrakte Klasse, Interface
überladen	nein, aber default Parameter	Methoden ja, Operatoren nein
variable Anzahl Parameter	∗args	args...
Monkey Patching	ja, aber nicht immer empfohlen	Nein
graphics	Externe Bibliotheken, z. B. Tk	Java Bibliotheken (AWT, Swing, FX, ...)
null	nil Objekt	null Literal
Enumeration	Enumerable	Iterable

Thema	Ruby	Java
preprocessor	nein	nein, aber Annotationsprozessoren sind möglich
Variablendeklaration	überall	Vor Benutzung, empfohlen: Am Anfang eines Blocks
Serialisierung		
Serialisierung	Module Marshal	Interface Serializeable
Namenskonventionen		
Klasse	UpperCamelCase	UpperCamelCase
Module	UpperCamelCase	lowercase, aber ganz anderer Begriff
Interface	n/a	UpperCamelCase
Variablen	snake_case	lowerCamelCase
Methode	snake_case	lowerCamelCase
Parameter	snake_case	lowerCamelCase
Konstante	SCREAMING_SNAKE_CASE	SCREAMING_SNAKE_CASE
Datei	Klasse/Module: Name in snake_case Skript: snake_case	Wie Klasse, Interface
Kommentar	#	/* */, // Javadoc: /** */
callbacks	pointers to global functions	use interfaces for command dispatching
assertions	gem solid_assert	assert, -ea für VM zur Aktivierung notwendig
	assert_xxx in Test::Unit	Klassenmethoden assert_xxx in org.junit.Assert
exit and return value to OS	exit	System.exit
Textdarstellung für Objekte	to_s, p	toString()
Exceptions		
Arten		Throwable: Wurzel der Fehlerhierarchie Error: Fehler in der JVM
	Exception	Exception
	begin raise rescue resume end	try-catch-finally
	begin ... end	try
	raise	throw
	rescue	catch
	ensure	finally
Testen		
Komponententest	RUnit	JUnit
Gleichheit		
Objektidentität	equal?	Referenzgleichheit: ==
Gleichheit, klassenspezifisch	==	equals
	für Hash: eql?, oft alias für ==	hashCode()
	=== für case	n/a
Ordnungsvergleich	<=> für Module Comparable liefert -1, 0 oder 1	compareTo aus Interface Comparable compare aus Interface Comparator Klassenmethode compare in diversen Wrapperklassen u.a.

Thema	Ruby	Java
	nil, wenn nicht vergleichbar	ClassCastException, wenn nicht vergleichbar und nicht durch Compiler erkennbar
	konsistent mit == implementieren	Konsistent mit equals implementieren
	Entwurfsmuster	
Null Object	NilClass	Optional
Iterator	Enumerator	Iterator
	Enumerable	Iterable
Fabrik		Fabrikmethode
Singleton	Module Singleton	Verwendung von Enum

E.6 Aufgaben

1. In Ruby gibt es eine Klasse *Rational*, in Java keine entsprechende Klasse. Schreiben Sie bitte eine entsprechende Klasse in Java.

2. In Ruby gibt es eine Klasse *Complex*, in Java keine entsprechende Klasse. Schreiben Sie bitte eine entsprechende Klasse in Java.

3. In Ruby gibt es eine Klasse *Range*. Schreiben Sie etwas Entsprechendes in Java!

4. Welche Arten von Vergleichen von Objekten auf Gleichheit gibt es Ruby bzw. in Java? Erläutern Sie bitte die Unterschiede und Gemeinsamkeiten!

5. Verbessern und ergänzen Sie bitte diese Übersicht.

Anhang F

Von C oder C++ zu Java

F.1 Übersicht

Ich gebe in diesem Kapitel eine kurze Übersicht über Unterschiede und Gemeinsamkeiten von *C* bzw. *C++* und *Java* zu geben. Ich möchte dabei helfen, Kenntnisse aus C und C++ möglichst einfach nach Java zu transferieren und die gröbsten Fehler zu vermeiden.

Die vielleicht wichtigsten Hinweise zu den Unterschieden C und C++ sind für mich diese, die ich mal bei Bjarne Stroustrup gelesen habe:

- Alle Beispiele in [KR88] sind gültiger C-Code.
- Die default Sichtbarkeit von struct in C ist public und die von class in C++ ist private.

Damit klärt sich auch schon vieles für den Übergang von C zu Java.

F.2 Lernziele

- Die wichtigsten Unterschiede zwischen C bzw. C++ und Java kennen.
- Die wichtigsten Gemeinsamkeiten von C bzw. C++ und Java kennen.
- Wissen, welche Syntax in Java äquivalent zur gleichlautenden Syntax in C bzw. C++ ist.
- Einige der gleichlautenden Befehle mit unterschiedlicher Wirkung kennen.

F.3 Klassen und Typen

In C haben Sie einige eingebaute Datentypen:

1. char, short, int, long (signed, unsigned)
2. float, double

Außerdem gibt es *struct* und *union* als Konstrukte, die eine weitgehende Erweiterung ermöglichen.

In C++ haben sie diese und es kommen Klassen hinzu. Letztere unterscheiden sich von den eingebauten Datentypen dadurch, das Sie Referenztypen sind.

In Java gibt es die signed Typen ebenfalls. Der primitive Typ char hat 16 Bit, ist numerisch und unsigned, es gibt keine signed char. Zusätzlich gibt es den primitiven Typ boolean und weitere (s. u.). Allerdings gibt es in Java kein unsigned int oder long. Es gibt in den Klassen *Integer* und *Long* Klassenmethoden für unsigned Rechnung und Vergleich.

Hinzu kommen als Referenztypen Klassen und Interfaces, außerdem Annotationen. In C und C++ können Sie Interfaces am ehersten mit Header-Dateien vergleichen.

F.4 Von Structs und Unions zu Klassen

Der Übergang von *C* zu *C++* lässt sich einfach zusammenfassen: Zusätzlich zu *struct* gibt es *class* und die default-Sichtbarkeit bei Klassen (class) ist private, während sie bei struct public ist. Außerdem kommt der new Operator hinzu und ensprechend delete. Ebenso kommen in C++ Exceptions hinzu.

Bei Java gibt es ein etwas anderes Schema, aber die wichtigsten Punkte sind Klassen und Interfaces. Die Beseitigung nicht mehr benötigter Objekte übernimmt der Garbage Collector. In C und C++ sind Sie als Programmierer für die Freigabe von Speicher selbst verantwortlich.

F.5 Grundlagende Unterschiede

Das Grundprinzip der Parameterübergabe in C ist call-by-value, in Java gibt es das auch für primitive Typen, aber oft haben Sie es mit call-by-reference zu tun. In beiden Fällen wird eine Kopie des Parameters übergeben, aber Achtung: Bekommen Sie eine Referenz auf ein Objekt, so können Sie destruktive Methoden aufrufen. Die können dann tatsächlich das Objekt ändern. Bei call-by-value sind Sie vor solchen Seiteneffekten geschützt.

F.6 Pointer und Referenzen

Manche meinen Pointer seien an C eines der schwierigsten Dinge. Ich sehe das anders, aber wer mit Pointern Probleme hatte, wird sich in Java wohler fühlen. Pointer werden Sie in Java nur in der Form von „NullPointerExceptions“ sehen und das auch nur, wenn Sie gegen den sinnvollen, guten Stil der Java-Programmierung verstoßen. Dann haben Sie selber Schuld!

F.7 Syntax und Konstrukte

Viele syntaktische Merkmale von C und C++ finden Sie auch in Java. Ich nenne hier nur wenige Punkte: Blöcke werden durch geschweifte Klammern {...} begrenzt, Statents enden mit „ ; “, Entscheidungen werden mittels if-then-else, dem ternären Operator oder über ein case Konstrukt getroffen, es gibt eine for-Schleife etc.

Die Unterschiede kommen von C zu C++ genau wie zu Java durch Vererbung (Polymorphismus) und funktionale Programmierung hinzu, weniger durch neue Syntax.

F.8 Tabellarische Übersicht

Hier nun eine tabellarische Übersicht. Ursprünglich diente mir die folgende Quelle als Muster: http://introcs.cs.princeton.edu/java/faq/c2java.html, zuletzt besucht am 25.12.2014. Von dem damaligen Muster für einen Vergleich von C und Java ist nicht mehr viel übrig geblieben, aber die Idee finde ich weiterhin ganz gut.

Thema	C,C++	Java
Allgemeines		
Art der Sprache	Compiliert, gelinked, ausführbare Dateien	Compiliert in Byte-Code, benötigt Laufzeitumgebung (JVM), seit Java 9 auch Linker mit anderer Funktion
Paradigmen	C: funktionsorientiert, prozedural, C++: zusätzlich objekt-orientiert, funktional in neueren C++ Versionen	objekt-orientiert, prozedural, funktionale Elemente
Typisierung	statisch, aber void *	statisch, aber Polymorphismus
cast	beliebig, auf eigene Gefahr	weitgehend geprüft, sonst auf eigene Gefahr
Basiselement	Funktion	Klasse, Interface
Portierbarkeit (Source)	Mit viel Disziplin möglich	ja
Portierbarkeit (Objectcode)	nein, compile für jedes Ziel	ja, Bytecode ist „write once, run anywhere“
Kommentare	/* */, // /** */ für Doxygen	/* */, // /** */ für Javadoc
Sichtbarkeit (accessibility), Kapselung	Für Elemente von Objekten und Klassen	für Elemente von Klassen, durch Security Manager weiter steuerbar
public	(C,C++: struct default) alle Objekte	alle Objekte
private	Objekte der Klasse	Objekte der Klasse
protected	Objekte der Klasse und Unterklassen	Objekte der Klasse, des Pakets und von Unterklassen
package	N/A	default ist package (kein Keyword), nur Objekte von Klassen des Pakets
friend	Objekte der Klassen und der angegebenen opaque pointer	N/A
Compiler	gcc hello.c erzeugt Maschinencode (.obj)	javac Hello.java erzeugt JVM Bytecode
Linker	Erzeugt ausführbaren Code	Neu in Java 9, andere Funktion, als in C oder C++
linking in the Math library	gcc -lm calculate.c	Keine Compiler Optionen erforderlich
Compilation	gcc main.c helper1.c helper2.c	javac Main.java, andere Dateien werden bei Bedarf automatisch compiliert
execution	a.out loads and executes program	java Hello interpretiert byte code

hello, world	`#include<stdio.h>`
	`int main(void) {`
	`  printf("Hello\n");`
	`  return 0;`
	`}`
		`public class HelloWorld {`
		`  public static void main(String[] args) {`
		`  System.out.println("Hello");`
		`  }`
		`}`

Thema	C,C++	Java
Block	{...}	{...}
Typen		
Ganze Zahlen	int üblicherweise 32 oder Bitlänge OS 2's complement	int ist 32 bit 2's complement
	signed, unsigned	kein unsigned Typ, aber unsigned Arithmetik-Methoden
	long üblicherweise 64 bit 2's complement	long ist 64 bit 2's complement
Fließkomma Zahlen	float üblicherweise 32 bit	float ist 32 bit IEEE 754 binary floating point;
	double üblicherweise 64 bit	double ist 64 bit IEEE 754 BigDecimal
boolean type	Verwende int: 0 für false, nonzero für true oder ein enum	boolean ist eigener Typ mit Werten true oder false
character type	char, signed, unsigned, Zeichensatz der Implementierung	char is 16 bit UNICODE, numerisch unsigned
String	\0 terminated char*	immutable Klasse String
Array Deklaration	C: int *a = malloc(n * sizeof(*a)); C++: int a[n]; char *s = new char[n]; int a[3] ={1, 2, 3};	int[ ] a = new int[n]; int [] a ={1, 2, 3};
Array Größe	Arrays kennen ihre Länge nicht	a.length
Datum/Uhrzeit	stdlib: time.h	LocalDate, LocalTime, LocalDateTime, ZonedDateTime Duration, Period
Literale		
int	*nnnn*,*ooo* (oktal),*xnn* (hex)	nnn_mmm (dez.) 0Xnnnn (hex), 0Bnnnn (binär), 0nn_mmm (oktal), `\unnnn` (Unicode)
long	*nnnnL*	*nnn_mmmmL*
Fließkomma	x.yEnn	x.yEnn[F]
String	`"Hello, world"`	`"Hello, world"`
Schleifen		

for loop	for (i = 0; i < n; i++)	for (int i = 0; i < n; i++)
		for(T t:anIterable) („for-each“)
		forEach

while	vorprüfend	vorprüfend
	do - while, nachprüfend	do - while, nachprüfend
Entscheidungen		
	if	if
	if else	if else
	switch case default	switch case default
	Ternärer Operator	Ternärer Operator
Variable		
Zulässige Zeichen	ASCII	Unicode (UTF-8) Character.isJavaIdentifierStart Character.isJavaIdentifierPart
Deklaration	Type name	Type nameInLowerCamelCase Type [ ] nameInLowerCamelCase (Array)

Thema	C,C++	Java
Namenskonventionen		
Variable	snake_case	lowerCamelCase
Konstante	SCREAMING_SNAKE_CASE	SCREAMING_SNAKE_CASE
Klasse	n/a, UpperCamelCase	UpperCamelCase
Namensraum	lowercase	package, module: lowercase
Datei	stack.c, stack.h	Stack.java - Dateiname wie Klasse bw. Interface
Teilsystem	namespace, lower case	package. lower case
Lesen/Schreiben		
Schreiben auf Standardausgabe	`printf("sum = %d", x)`	`System.out.println("sum = " + x);`
Formattierte Ausgabe	`printf("avg = %3.2f", avg);`	`System.out.printf("avg = %3.2f", avg)`
Lesen von Standardeingabe	`scanf("%d", &x);`	Java library support
memory address	pointer	Referenz
Pointers Manipulation	*, &, +	Keine Manipulation von Referenzen
Funktionen		
Begriff	function	method
Deklaration	int [static] max(int a, int b)	public [static] int max(int a, int b)
Parameterübergabe	by-value: Übergabe Kopie des Werts	Primitive Typen: by-value: Übergabe Kopie des Werts Referenz-Typen: by-reference: Übergabe Kopie der Referenz
Variable Anzahl Argumente	varargs	type args oder type [ ]...
Zugriff auf Bibliothek	#include <stdio.h>	import java.io.File;
Zugriff auf Bibliotheksfunktion	`#include "math.h"`	import static java.lang.Math.*;
	x = sqrt(2.2);	x = sqrt(2.2); oder x = Math.sqrt(2.2);
Definition Datenstruktur	struct	class, Klassen haben Felder (Attribute) und Funktionen (Methode)
Zugriff auf Datenstruktur	a.numerator für elements	a.numerator für instance variables, c = a.plus(b) for methods
pointer chasing	x->left->right	x.left.right
allocating memory	malloc new	new Operator ruft Konstruktor auf.
de-allocating memory	free	automatic garbage collection
buffer overflow	segmentation fault, core dump, unpredicatable program	checked run-time exception
Deklaration von Konstanten	const und #define	final

Thema	C,C++	Java
Initialisierung von Variablen	nicht garantiert	Attribute (und Array-Elements) werden mit 0, null, or false initialisiert. compile-time Fehler beim Zugriff auf nicht initialisierte Variablen
Kapselung	opaque pointers and static	private, protected
Interface method	non-static function	public method
data type for generic item	void *	Object
casting	anything goes	checked exception at run-time or compile-time error
demotions	automatisch, aber ggf. Genauigkeitsverlust	Expliziter cast notwendig, z. B. von long zu int
Polymorphismus	union	Vererbung: extends, implements
überladen	C: nein C++: Funkionen und Operatoren	ja für Methoden, nein für Operatoren
Graphik	use external libraries	Java library support
null	NULL	null
enumeration	C: enum C++: ähnlich wie Java	typsichere enum, vollwertige Klasse
preprocessor	Ja	Nein
Variablen Deklaration	Am Anfang eines Blocks	Vor Benutzung, empfohlen: Am Anfang eines Blocks
callbacks	pointers to global functions	Verwende Interface z. B. für Swing, FX
funktionale Programmierung	function pointer, C++: Lambdas	Lambda, Functional Interface
assertions	assert	assert, JVM Parameter -ea
exit and return value to OS	exit(1)	System.exit(1)
Textdarstellung für Objekte	n/a	Methode toString()
Komponententests	CUnit, cppUnit	JUnit
Modularisierung		
	struct (C) class (C++)	class
	namespace	package, module

F.9 Historische Anmerkungen

Gerüchte über die Entstehung von C gibt es viele. Zur Entwicklung von C++ verweise ich gerne auf [Str94a]. Dieses Buch habe ich mit viel Vergnügen gelesen. Für Java könnte Joshua Bloch sicher viel zur Geschichte beitragen, ich hoffe er hat Besseres zu tun.

F.10 Aufgaben

1. Schreiben Sie bitte ein C-Programm, dass eine Datei mit einer main-Methode in C einliest und in syntaktisch korrekten und semantisch äquivalente Java-Code als .java Datei ausgibt.
2. Schreiben Sie bitte ein Java-Programm, dass das gleiche leistet!
3. Schreiben Sie bitte ein Java-Programm, dass eine in C geschriebene struct einliest und daraus eine Java-Klasse mit gettern und settern erzeugt!

Anhang G

Aufgaben

> Wir stehen selbst enttäuscht und sehn betroffen
> Den Vorhang zu und alle Fragen offen.
>
> Bertolt Brecht. Der gute Mensch von Sezuan.[Bre70]

G.1 Wahr oder Falsch?

1. Jeder Typ in einem Java-Programm ist ein Objekt.
2. Jeder Typ in einem Java-Programm ist eine Klasse.
3. Die Methode toString wird automatisch aufgerufen, wenn ein Objekt ausgegeben wird.
4. In Java gibt es keine Pointer.
5. In Java gibt es keine Polymorphie.
6. In Java gibt es niemals Programmierfehler.
7. null, NEGATIVE_INFINITY und POSITIVE_INFINITY sind vom Typ float.
8. Das Schlüsselwort *static* erzwingt, das ein Objekt statisch eingefroren wird also z. B. seine Werte nicht ändern darf.
9. Wartbarer und lesbarer Code ist wichtig.
10. Klassen sind besondere Methoden, die sich dadurch abgrenzen, dass sie reduzierte Möglichkeiten haben.
11. Eine long-Variable ist 4 Byte groß.
12. Eine long-Variable ist 8 Byte groß.
13. Eine float-Variable ist 4 Byte groß.
14. Eine float-Variable ist 8 Byte groß.
15. Eine double-Variable ist 4 Byte groß.
16. Eine double-Variable ist 8 Byte groß.
17. Es hat keine Wirkung das Schlüsselwort public wegzulassen, wenn dafür kein anderes verwendet wird.

18. Die default-Sichtbarkeit in Java ist *public* wie bei *structs* in C.
19. Die default-Sichtbarkeit in Java ist *private* wie bei Klassen in C++.
20. Die default-Sichtbarkeit in Java ist *package*.
21. Mit dem Schlüsselwort super(...) mit geeigneten Parametern kann man im Konstruktor einen Konstruktor der Superklasse aufrufen.
22. Rekursive Programme sind komplizierter und beherbergen schwer zu findende Fehler. Rekursion ist daher soweit möglich zu meiden.
23. Der Garbage-Collector ist für die Speicherfreigabe zuständig, wenn Speicher(bereiche) nicht mehr benötigt wird (werden).
24. Ein großer Vorteil der JVM ist, dass es nie zu wenig Speicher geben kann.
25. Unter Java kann man gezielt auf die Bits der in Variablen gespeicherten Werte zugreifen sofern man die Rechte hat.
26. Die Java API-Dokumentation gibt es in jeder Sprache für die es eine *Locale* gibt (siehe *static* Attribute (fields)) in der Klasse *Locale*.
27. Java ist anerkannt die beste Programmiersprache und macht den Einsatz anderer Programmiersprachen überflüssig.
28. Java unterstützt die Definition von eigenen (Daten-)Typen.
29. Java unterstützt Einfachvererbung, d. h. eine Klasse hat genau eine direkte Oberklasse.
30. Java unterstützt Mehrfachvererbung für Interfaces.
31. Java unterstützt Mehrfachvererbung für Interfaces und abstrakte Klassen.
32. Java unterstützt Einfach-Vererbung, d. h. eine Klasse hat genau eine direkte Oberklasse.
33. Wenn ein Objekt mit *new* erzeugt wurde, muss es mit *delete* zerstört werden, wenn es nicht mehr benötigt wird.
34. Package-Namen werden nach Konvention aus Kleinbuchstaben gebildet.
35. Klassennamen in Java beginnen nach Konvention mit einem Großbuchstaben.
36. Klassennamen in Java bestehen nur aus Großbuchstaben und an Wortgrenzen einem Unterstrich (underscore).
37. Kommentare in Java beginnen mit // oder /*.
38. Java Kommentare enden mit `\n` oder */.
39. Javadoc Kommentare beginnen mit /** und enden mit */
40. Javadoc Kommentare können HTML Tags enthalten.
41. Jede If-Anweisung kann durch eine äquivalente switch-Anweisung ersetzt werden.
42. Jede switch-Anweisung kann durch eine äquivalente if-Anweisung ersetzt werden.
43. Java generiert für eine Klasse immer einen default-Konstruktor.
44. Der Rückgabetyp kann für das Überladen von Methoden nicht verwendet werden.
45. Durch *this(...)* kann in einer Methode einer Klasse der geeignete Konstruktor aufgerufen werden.

46. Ein Aufruf eines anderen Konstruktors mittels *this(...)* muss die erste Anweisung in einem Konstruktor sein.

47. Durch *super(...)* kann in einer Methode einer Klasse der geeignete Konstruktor der direkten Oberklasse aufgerufen werden.

48. Ein Aufruf eines Konstruktors der Oberklasse mittels *super(...)* muss die erste Anweisung in einem Konstruktor sein.

49. Wird eine lokale Variable nicht intialisiert, so gibt es eine RuntimeException.

50. Wird eine lokale Variable nicht initialisiert, so gibt es bei ihrer Verwendung einen Compiler-Fehler.

51. Wird eine lokale Variable nicht initialisiert, so gibt einen Compiler-Fehler.

52. Attribute einer Klasse werden in der Reihenfolge ihrer Deklaration mit den entsprechenden default-Werten initialisiert.

53. Klassenattribute (statische Attribute) werden beim Laden einer Klasse initialisiert.

54. Hat man ein Array `int [] a = new int[4]`, so kann man nach dem letzten Element noch Eins einfügen: `a[4]=1;`

55. Ein Array von primitivem Typ wird automatisch vergrößert, wenn die Länge nicht ausreicht.

56. Konstruktoren können in Unterklassen überschrieben werden.

57. Auf die als protected deklarierten Elemente einer Klasse können alle Methoden von Unterklassen zugreifen.

58. Auf die als protected deklarierten Elemente einer Klasse können ausschließlich die Methoden von Unterklassen zugreifen.

59. Das Keyword *final* kennzeichnet Konstanten, die zur Compile-Zeit festgelegt werden und danach nicht mehr geändert werden können.

60. Das Keyword *final* vor Attributen kennzeichnet Werte, die beim Erzeugen von Objekten festgelegt werden und danach nicht mehr geändert werden können.

61. Das Keyword *final* vor nicht-statischen Attributen kennzeichnet Werte, die beim Erzeugen von Objekten festgelegt werden und danach nicht mehr geändert werden können.

62. Wenn eine Klasse *compareTo* implementiert, dann muss sie auch *equals* konsistent implementieren.

63. Wenn eine Klasse *equals* implementiert, dann muss sie auch *compareTo* implementieren.

64. Attribute in Interfaces sind automatisch *static* und *final*.

65. Innere Klassen können auf alle Elemente der umschließenden Klasse zugreifen.

66. Innere Klassen können nur auf die als *public* oder *protected* deklarierten Elemente der umschließenden Klasse zugreifen.

67. Jede eigenständige Java Anwendung muss eine *public class* mit einer *Methode main* als Startpunkt enthalten.

68. Mit dem Schlüsselwort *private* wird der Zugriff auf Methoden/Attribute einer Klasse beschränkt auf Klassen innerhalb desselben package.

69. *char*-Variablen bestehen aus 2 Bytes.

70. *char*-Variablen bestehen aus 1 Byte.

71. Mit *null* bezeichnet man in Java das mathematische Ergebnis einer Operation, z.B ergibt $4 - 3 - 1 = null$

72. Das Interface *Comparable*<*T*> spezifiziert, dass eine Methode public int compareTo(T) implementiert wird.

73. Einen String muss man so erzeugen wie jedes andere Objekt: `String a = new String("hallo");`

74. Die *File*-Klasse dient dazu, Daten in Dateien zu schreiben bzw. Daten aus Dateien zu lesen.

75. Alle Operatoren werden bei Java von links nach rechts ausgeführt, will man das nicht, muss man Klammern setzen.

76. Wrapper-Klassen hüllen primitive Typen in Objekte ein.

77. String-Objekte sind „immutable“. Wenn man Strings kombiniert, werden daraus immer neue String-Objekte erzeugt.

78. Das Löschen eines Objekts aus einer LinkedList ist schneller als die entsprechende Operation bei einer ArrayList.

79. Die Schlüssel-Objekte und die Werte-Objekte einer Map bilden jeweils ein Set.

80. Wrapper-Klassen dienen dazu, um mit primitiven Typen Rechnungen durchführen zu können.

81. Stream-Klassen benutzt man für die Behandlung eines Eingangs- oder Ausgangsdatenstrom von Bytes.

82. Die *main* Methode muss wie folgt aussehen : public static void main(String [ ] args) {. . . }

83. Eine Klassenattribut wird durch das Schlüsselwort *static* spezifiziert, der Wert des Attributs gilt dann für alle Objekte einer Klasse.

84. Eine innere Klasse kann innerhalb eine anderen Klasse definiert werden, Java benutzt dafür das Schlüsselwort *inner*.

85. Ein Interface enthält alle Operationen, die eine Klasse implementieren muss.

86. Eine Klassenmethode kann auf Instanzattribute eines Objekts der Klasse zugreifen.

87. Eine Variable in Java ist in dem Block gültig, in dem sie deklariert wird.

88. Ein Array ist ein Objekt, dass auf dem Heap erzeugt wird.

89. Ein Interface kann durch eine anonyme Klasse implementiert werden.

90. Eine Assoziation stellt eine schnelle Verbindung zwischen Objekten her.

91. Ein Interface enthält mindestens eine Operation.

92. Java nimmt Typ-Konvertierungen automatisch vor, wann immer dies möglich ist.

93. Eine Variable vom Typ *byte* ist unsigned, hat also kein Vorzeichen.

94. Jede Java-Applikation ist auch automatisch ein Java-Applet.

95. Ein SWT-Composite beinhaltet einen Container für Widgets.

96. Mit dem Schlüsselwort *protected* wird der Zugriff auf Methoden/Attribute einer Klasse beschränkt auf Klassen innerhalb des selben Pakets.

97. Form-Layout ist das komplizierteste SWT-Layout.
98. Die Widget-Klasse ist die Basisklasse für alle SWT-Fenster-Elemente.
99. Ein Dialog heißt modal, wenn mehrere Dialog-Antworten möglich sind.
100. Das interface *Comparable* verlangt, dass eine Methode *public int compareTo(Object o)* implementiert wird.
101. Autoboxing ist eine Neuerung von Java 5.0, primitive Typen werden automatisch bei Bedarf in die entsprechenden Objekt-Typen konvertiert.
102. Die *Scanner*-Klasse ist ein Iterator über einen Datenstrom, z.B. kann damit leicht über die durch Leerzeichen getrennten Strings eines Eingabestroms iteriert werden.
103. Eine generische Klasse akzeptiert nur Objekte des gegebenen Typs und Objekte von Subtypen dieses Typs.
104. Seit Java 5.0 dürfen *raw* Collections nicht mehr benutzt werden.
105. Ein Typ $C<A>$ nennt man auch einen parametrisierter Typ.
106. Generische Typen entsprechen genau den sogenannten Templates bei C++.
107. Die *clone*-Methode von Objekt erzeugt eine flache Kopie des Objekts.
108. Mit Hilfe von Reflection kann man die Eigenschaften eines Objekts zur Laufzeit untersuchen - z. B. die public-Methoden und Instanzvariablen.
109. Die Schlüssel-Objekte einer *Map* bilden ein *Set*.
110. Das Observer-Pattern wird nur bei Swing benutzt, nicht bei SWT.
111. Die *stop*-Methode der Klasse *Thread* ist *deprecated* und sollte nicht mehr benutzt werden.

G.2 Geschlossene Fragen

1. Wie viele *double* werte gibt es im Intervall $(0, 1)$ (also auschließlich 0 und 1)? **Ab hier Fragen aus einem Quiz der Computerwoche. Sie beziehen sich höchstens teilweise auf Java**
2. Wie ist die Beziehung zwischen Java und JavaScript?
   2.1. Netscape erfand JavaScript als eine abgespeckte, leichte Version von Java.
   2.2. Java lieh sich Ideen von JavaScript aus und entwickelte daraus eine vollwertige Programmiersprache.
   2.3. Beides stammt ursprünglich von Sun Microsystems.
   2.4. Es gibt keine; es ist alles nur Marketing.
3. Die ungarische Notation ist ein Konvention, um Variablen zu benennen. Wie bekam sie ihren Namen?
   3.1. Sie wurde so benannt wegen ihrer Ähnlichkeit zur umgekehrten polnischen Notation (UPN)
   3.2. Ihr Erfinder war Ungar
   3.3. Wenn Sie die Bezeichnungen laut aussprechen, klingt es so, als ob Sie ungarisch sprechen würden
   3.4. Sie wurde an der polytechnischen Universität zu Bukarest, Ungarn, erfunden

4. Just-in-time-Compilierung (JIT) verbesserte die Leistungsfähigkeit von Sprachen, die in Programme in Bytecode übersetzen. Welche Sprache besaß den ersten JIT-Compiler?

   4.1. Java

   4.2. C#

   4.3. Smalltalk

   4.4. COBOL

5. Wenn ich Ihnen die Erzeugung von „Threaded Code“ als Schlüsseleigenschaft der Programmiersprache meiner Wahl mitteilen würde, über welche Sprache rede ich dann höchstwahrscheinlich?

   5.1. Pascal

   5.2. Java

   5.3. Forth

   5.4. Python

6. Einstmals sehr beliebt und häufig eingesetzt brachte Pascal eine Reihe von abgeleiteten Sprachen hervor. Welche ist *kein* Erbe von Pascal?

   6.1. Python

   6.2. Ada

   6.3. Oberon

   6.4. Modula-2

7. Im nahegelegenen Supermarkt sind Energydrinks wie Red Bull oder Jolt Cola ausverkauft. Welches Getränk ist durch seinen hohen Gehalt an Koffein eine gute Alternative?

   7.1. Fritz-Kola

   7.2. Coca Cola Classic

   7.3. Pepsi Cola

   7.4. Afri Cola

8. Was ist der beste Weg Typsicherheit bei Assembler zu wahren?

   8.1. Linken Sie Ihre Module nicht mit dem Modulen von anderen Sprachen

   8.2. Stellen Sie sicher, dass Sie alle Variablen rechtzeitig deklarieren

   8.3. Addieren Sie keine Variablen eines Typs zu einem anderen Typ

   8.4. Beten Sie andächtig

9. Welche der folgenden Grundsätze gehören *nicht* zur Extremprogrammierung (extreme programing, kurz XP)?

   9.1. Restrukturieren Sie Ihren Code oft

   9.2. Verwerfen Sie veralteten Code

   9.3. Debugging ist für Weicheier

   9.4. Sprechen Sie häufig mit dem Kunden

10. Warum sind Wettlaufsituationen ein Problem bei moderner Softwareentwicklung?

    10.1. Unter Entwicklern sind Minderheiten unterrepräsentiert

    10.2. Entwickler können nicht schnell genug entwickeln, um die Anforderungen zu erfüllen

10.3. Software kann mit der Geschwindigkeit moderner Prozessoren nicht mithalten

10.4. Prozesse, die sich den gleichen Speicher teilen, können unerwartete Ergebnisse erzeugen

11. Warum betrachten manche Ruby als die sauberere objektorientierte Sprache als andere OO-Sprachen wie Java und C++?

11.1. Weil Ruby Objektorientierung unterstützt und nicht erlaubt, prozedural zu programmieren

11.2. Weil Ruby nicht zwischen Objekten und primitiven Datentypen unterscheidet

11.3. Weil sich die Syntax von Ruby an Perl anlehnt

11.4. Weil Ruby-Programmierer eingebildet sind

12. Fehler bei der Validierung von Benutzereingaben gehören zu den Hauptgründen für Software-Sicherheitslecks. Wann ist es sicher, Benutzereingaben zu akzeptieren?

12.1. Wenn die Anwendung hinter einer Firewall läuft

12.2. Niemals. Jedes Programm, dass Eingaben entgegennimmt, muss sie validieren

12.3. Wenn die Anwendung in Perl geschrieben wurde und den sogenannten „taint mode“ verwendet

12.4. Wenn der Benutzer Ihre eigene Mutter ist

13. Welche der folgenden Antworten ist richtig, um wiederverwendbaren Code zu erzielen, der leicht zu warten ist?

13.1. Verwende mehr globale Variablen

13.2. Vergebe Variablen- und Funktionennamen mit ein bis zwei Buchstaben

13.3. Füge Kommentare in den Sourcecode ein

13.4. Verwende Pointer-Arithmetik, wo immer möglich

14. Welche der folgenden Personen ist *kein* Erfinder einer Programmiersprache, die noch verwendet wird?

14.1. Andrew S. Tanenbaum

14.2. Guido van Rossum

14.3. Niklaus Wirth

14.4. Bjarne Stroustrup

15. Auf welches Konzept bezieht sich der „Mythos des Mann-Monats“?

15.1. Der Code der letzten Monat geschrieben worden wäre, wenn das Projekt fristgerecht beendet worden wäre

15.2. Die Menge Code, die jemand in einem Monat für sein Gehalt schreiben kann

15.3. Auf das Gesetz, dass das Hinzufügen von Arbeitskräften zu einem verspäteten Projekt die Verspätung erhöht

15.4. Auf die Tatsache, dass man mit dem Debugging wie Sisyphos mit seinem Felsblock niemals fertig werden wird

16. Ist P gleich NP?

16.1. Ja

16.2. Nein

16.3. Manchmal

16.4. Ich weiß es nicht

17. Ein Kunde bittet sie, eine einfache Software zur Verwaltung von Bankkonten in C zu entwickeln. Welche Datenstruktur ist am besten für geeignet, Euros und Cents abzubilden?

17.1. Float

17.2. Double

17.3. Integer

17.4. Boolean

18. Für welche Leistung ist Brian Kernighan bestens bekannt

18.1. Er war Miturheber des Unix-Betriebssystems

18.2. Er war Miturheber von AWK, einer Programmiersprache für Textverarbeitung

18.3. Er war Miturheber der Programmiersprache C

18.4. Er war Miterfinder der objektorientierten Programmierung

19. Eine Programmiersprache ist Turing-vollständig, wenn sie dazu verwendet werden kann, sämtliche Algorithmen zu berechnen. Welche der folgenden Sprache ist nicht Turing-komplett in ihrer Standardform?

19.1. PostScript

19.2. BASIC

19.3. C#

19.4. SQL

20. Welche Gruppe hat den größten Einfluss auf die moderne objektorientierte Programmierung?

20.1. The Gang of Four

20.2. The Party of Five

20.3. Die Erfrischungsgetränkeindustrie

20.4. AC/DC

21. Was von den folgenden Antworten ist KEINE Datenstruktur einer modernen Programmiersprache?

21.1. Linked List

21.2. Twisted Pair

21.3. Circular buffer

21.4. Sparse matrix

22. Wen bezeichnet man allgemein als den ersten Programmierer?

22.1. Blaise Pascal

22.2. Ada Lovelace

22.3. Jean-Benoit Fortran

22.4. William Henry Gates III

23. Welche der folgenden Sprachen ist, genau genommen, keine Programmiersprache?

23.1. SQL

23.2. PostScript

23.3. AppleScript

23.4. Tcl

24. Welche der folgenden Aussagen ist wahr bezogen auf die Programmiersprache C?

24.1. Methoden dürfen nur einen einen Wert zurückgeben, aber Objekte verfügen über eine Reihe von Methoden

24.2. Der Referenzzähler eines Objekts muss null erreichen, bevor die Garbage Collection greift

24.3. Alle Klassen müssen in Headerdateien deklariert werden, um sie als Objekte zu erzeugen

24.4. Keine der genannten Antworten

25. Was ist ein Singleton?

25.1. Eine Klasse, von der nur eine einziges Objekt erzeugt werden kann

25.2. Ein komplettes Programm, das nur eine einzige Zeile Code einnimmt

25.3. Ein Datenstruktur wie eine verkettete Liste, die nur ein Feld beinhaltet

25.4. Ein typischer Programmierer Freitag nachts

26. Warum ist Parallel Computing eine der schwierigsten Disziplinen für Programmierer?

26.1. Wettlaufsituationen können unerwartet Daten zerstören

26.2. Mangelhafte Speicherverwaltung kann zu Anwendungs-Deadlocks führen

26.3. Zu viele Threads oder Prozesse können die Performance verringern

26.4. Alle genannten

27. Welcher der folgenden Sprachen verlangt *nicht*, dass jede Codezeile mit einem Semikolon beendet wird?

27.1. Java

27.2. C++

27.3. Python

27.4. Alle genannten

28. Welche der folgenden Antworten trifft *nicht* auf eine funktionale Programmiersprache zu?

28.1. Funktionen können andere Funktionen als Argumente verwenden und als Ergebnis zurückgeben

28.2. Algorithmen codiert man in natürlichen, mathematischen Ausdrücken statt in Verfahrensschritten

28.3. Jede Funktion muss eine Art des Zustands ändern - entweder innerhalb der Maschine oder innerhalb der Außenwelt

28.4. Rekursion wird gefördert

29. Was ist der erste Schritt beim Design eines rekursiven Algorithmus?

29.1. Stellen Sie sicher, dass alle Unterprogramme thread-sicher sind

29.2. Geben Sie allen allokierten Speicher frei

29.3. Normalisieren Sie das Datenmodell

29.4. Entscheiden Sie sich für eine Antwort auf die Frage ? oberhalb

30. C# ist eine Sprachplattform, die sehr ähnlich zu Java ist. Wenn ein Programmierer Software in C# mit Microsofts Visual Studio entwickelt, welches Tool ist das Gegenstück in Java?

30.1. Subversion

30.2. Apache Ant

30.3. NetBeans

30.4. Spring

31. Was ist eine typische Aktivität beim Refactoring?

31.1. Buffer-Überlauf wird verhindert

31.2. Variablen werden umbenannt, damit das Programm leichter lesbar ist

31.3. Exception Handler werden hinzugefügt, um Laufzeitfehler abzufangen

31.4. Neue Funktionen werden hinzufügt, die auf den neuesten iterativen Designänderungen basieren

32. Was ist eine Turingmaschine?

32.1. Ein abstraktes Gerät, das die Logik von jedem denkbaren Computeralgorithmus simulieren kann

32.2. Jeder Mechanismus, der menschliche Intelligenz täuschend echt simulieren kann

32.3. Ein Gerät, das früher verwendet wurde, um Lochkarten in den Computer einzugeben

32.4. Jede der Maschinen, die von den ersten mechanischen Computern abgeleitet wurden

33. Pointer sind ein Relikt der C-Programmierung, die bei fehlerhafter Pointer-Arithmetik zu schweren Fehlern führen können. Bei welchem der folgenden C-Nachfolgern hat man daher entschieden, keine derartigen Pointer zu verwenden?

33.1. 1.) C# 2.) Java 3.) Objective-C 4.) C++

34. APL ist eine betagte Programmiersprache, die schnell durch ihre Eigentümlichkeiten in Ungnade gefallen ist. Warum hebt sich APL gegenüber heutigen Standards so ab?

34.1. Sie können APL-Code nicht mit den Zeichen einer üblichen Tastatur eingeben

34.2. Als Mac-OS-Sprache können Sie nicht ohne einen GUI-Editor programmieren

34.3. APL kennt keine Zeichenketten. Text muss jeweils Zeichen für Zeichen verarbeitet werden

34.4. Als einzige Albanische Programmiersprache basieren alle Schlüsselwörter auf dem Albanischen

35. Welche Programmiersprache ist ein Dialekt von ECMAScript?

35.1. ActionScript

35.2. AppleScript

35.3. PostScript

35.4. VBScript

36. Was haben Brücke, Fassade, Fabrik und Fliegengewicht gemeinsam?

36.1. Sie sind Versionskontrollsysteme für verschiedene Plattformen

36.2. Sie sind populäre Frameworks für die GUI-Entwicklung

36.3. Sie sind Entwurfsmuster für OO-Softwareentwicklung

36.4. Sie beginnen alle mit dem Buchstaben „F“

37. Was ist der Hauptnachteil von statischen Bibliotheken in Bezug auf Anwendungen?

37.1. Statische Bibliotheken erhöhen die Dateigröße von Anwendungen und führen zu Coderedundanzen

37.2. Ohne dynamisches Binden kann Software nicht die Vorteile von höheren Funktionen ausnutzen

37.3. Die Anwendung steht unter der Lizenz der dazu gebundenen Bibliotheken, was Lizenzkosten nach sich ziehen kann

37.4. Das Fehlen der dynamischen Bindung führt zu einer Zunahme von Bibliotheken und zur DLL-Hölle

38. Für welche Programmiersprache wurde der erste optimierende Compiler geschrieben?

38.1. Cobol

38.2. Fortran

38.3. C

38.4. LISP

39. Was ist kein Werkzeug für das Softwaredesign?

39.1. Flussdiagramme

39.2. Pseudocode

39.3. Bytecode

39.4. UML

40. Weswegen serialisiert man ein Objekt bei der OO-Programmierung?

40.1. Um seine Datenstrukturen im Speicher zu sortieren, damit man schneller darauf zugreifen kann

40.2. Um sicherzustellen, dass es durch Eliminierung von Deadlocks oder Wettlaufsituationen thread-sicher ist

40.3. Um es in einfachere Subobjekte (Episoden) aufzuteilen

40.4. Um sie in eine Form zu bringen, die dauerhaft gespeichert werden kann

Ende Quiz Computerwoche

41. In *java.io* gibt es die Klasse *File*. Worin besteht der Unterschied zwischen den Methoden *getCanonicalPath* und *getAbsolutePath*?

42. Gegeben seien zwei Unterklassen von *Exception*: *A* und *B*, die innerhalb eines *try*-Blocks geworfen werden können. Unter welcher Bedingung ist es egal, in welcher Reihenfolge je ein *catch*-Block für jede dieser Exception-Klassen deklariert wird?

43. Gegeben seien zwei Unterklassen von *Exception*: *A* und *B*, die innerhalb eines *try*-Blocks geworfen werden können. Dabei gilt *A extends B*. In welcher Reihenfolge müssen die catch-Blöcke deklariert werden?

44. Welche Exceptions müssen Sie in Java nicht explizit selbst behandeln?

45. Wodurch kann es in Java dazu kommen, dass eine nicht-Runtime Exception geworfen wird?

46. Gegeben Sei die folgende Deklaration mit Zuweisung:

```
int [][] a = {{1,0,1},{0,1,0},{0,0,1}};
```

„Wie groß ist a?“ Bitte geben Sie alle Ihrer Ansicht nach sinnvollen Antworten mit jeweils einer kurzen Begründung an.

G.3 Offene Fragen

1. Nennen Sie bitte alle Möglichkeiten Attribute in Java zu initialisieren!
2. Wofür ist eine Instanz-Initialisierungsblock nützlich oder sogar notwendig?
3. Nennen Sie knapp die Unterschiede und Gemeinsamkeiten zwischen abstrakten Klassen und Interfaces in Java.
4. Nennen Sie bitte die Vorteile der parametrisierten Collection Klassen (generische) gegenüber den prä Java 1.5 angebotenen, die nur Objekte organisieren konnten!
5. Die Verwendung des Schlüsselworts *final* ist vor allem eine Performancefrage und sollte deshalb höchstens als letztes Mittel in Erwägung gezogen werden. Ist diese Behauptung wahr oder falsch?
6. Was würde es bedeuten, wenn Sie einen Konstruktor *final* deklarieren?
7. Welche Arten von Typen kennen Sie in Java? Nennen Sie alle, die Ihnen einfallen!

   Hier einige Fragen von Birgit Wendholt aus dem WS 2009/10 (Satzzeichen und Tipppfehler korrigiert, wenn erkannt).
8. Welche Typsysteme kennen Sie in Java? Was sind die wesentlichen Unterschiede?
9. Welche Typen begründen in Java Typfamilien?
10. Was versteht man unter Wrapperklassen?
11. Welches Compilerverfahren ermöglicht die Zuweisung von Wrappertypen zu den zugehörigen Basisdatentypen und umgekehrt? Erläutern Sie das Verfahren an Beispielen!
12. Was versteht man unter dem statischen Typ einer Variable?
13. Was versteht man unter dem dynamischen Typ einer Variable?
14. Wodurch erhält man in Java trotz statischen Typs von Variablen polymorphes Verhalten beim Aufruf von Methoden auf diesen Variablen?
15. Welche Formen von Polymorphie kennen Sie in Java?
16. Was ist der Unterschied zwischen *Überladen* und *Überschreiben* von *Methoden*? Zeigen Sie bitte am Beispiel den Unterschied zwischen *Überladen* und *Überschreiben* von Methoden!
17. Was versteht man unter ungeschützter Kovarianz von Arrays? Geben Sie ein Beispiel!
18. Erläutern Sie bitte einen der Begriffe *mutable* bzw *immutable*!
19. Welche Objekte sind insbesondere in Java immutable?
20. Erklären Sie die Begriffe Interface und abstrakte Klasse und nennen Sie die Unterschiede!
21. Was ist eine generische Klasse / Interface? Was ist ein generischer Typ?
22. Welche Arten generischer Typen kennen Sie? Erläutern Sie bitte deren Unterschiede!
23. Nennen Sie bitte drei Einschränkungen in der Verwendung generischer Typen!
24. Was versteht man unter Type-Erasure?
25. Was sind Enums bzw. wann verwenden Sie Enums?
26. Was versteht man unter Typsicherheit von Enums?

27. Welche Formen der Initialisierung von Objektvariablen kennen Sie in Java?

28. Welche Arten von Sichtbarkeitsattributen für Methoden kennen Sie in Java. Erläutern Sie für jedes Attribut, wer Zugriff auf die Methode hat!

29. Was ist eine anonyme innere Klasse? Welche Einschränkungen gelten bei anonymen inneren Klassen für lokale Variablen des Aufrufkontextes?

30. Erklären Sie das Singleton-Pattern!

31. Erklären Sie das Composite-Pattern!

32. Erklären Sie das Observer-Pattern! Wann wird dieses Pattern eingesetzt?

33. Was versteht man unter Exception-Chaining?

34. Welche zwei Möglichkeiten kennen Sie in Java, um Objekte vergleichbar zu machen? Geben Sie ein Beispiel für deren Anwendung!

    Ende Fragen von Birgit

35. Gegeben Sie die folgende Deklaration:

    ```
    int [][] matrix = new int [256][256];
    ```

    Wie lang bzw. groß ist *matrix*? Geben Sie bitte alle Antworten, die Sie für sinnvoll halten und begründen Sie die Antwort jeweils kurz!

36. Wann heißt ein Objekt immutable?

37. Welche Vorteile hat es, wenn die Objekte einer Klasse immutable sind?

38. Schreiben sie eine rekursive Methode sum(x, y) um die Summe von x und y zu berechnen. x und y sind natürliche nicht negative Zahlen inklusive 0. Da Sie damit die Addition sozusagen erst definieren sollen, dürfen Sie in ihrem Code als Addition nur +1 und -1 verwenden (Vorgänger- und Nachfolgerfunktion. sum(x, y)

39. Schreiben Sie eine iterative Methode um zu bestimmen, ob der gegebene String von vorne und hinten gelesen gleich ist. Beispiel: „aba“ und „aa“ sind Palindrome, „ab“ ist keins. Verwenden Sie eine while- Schleife!

40. Schreiben Sie eine rekursive Methode um die höchste Zahl in einem ZahlenArray zu finden, das Zahlen oder weitere Zahlenarrays enthält. z.B. [1,3,4,2,7,5,[3],[1,2,[7,2,9,[1]],5],3] highest_number(arr)

41. Warum kann ein Interface in Java keine Klassenoperation enthalten? Es gibt einen logischen Grund dafür!

42. Was ist die default Sichtbarkeit eines Elements in Java (wenn also keine Sichbtarkeit spezifiziert wird)? Welche Elemente können auf ein solches Element zugreifen?

43. Sie sollen für Java X eine Methode *isNull()* schreiben in der Klasse *Object* schreiben, die es ermöglicht zu prüfen, ob ein Objekt null ist. Die Methode soll *true* liefern, wenn das Objekt *null* ist, andernfalls *false*.

    Nehmen Sie den Auftrag an?

    - Wenn ja: Wie lautet der Code der Methode?
    - Wenn nein: Warum nicht?

G.4 Fehler finden und korrigieren

G.4.1 compareTo

Finden Sie alle Fehler oder Unzulänglichkeiten in folgendem Codeausschnitt:

```
public class Dollar implements Comparable<Dollar> {
        private double value;
        public int compareTo(Object d) {
                if (d.value == value)
                        return 0;
                else if (d.value < value)
                        return 1;
                else
                        return -1;
        }
}
```

G.4.2 Nochmals Comparable und Cloneable

Hier eine Klasse aus einem Übungsbuch für Java 9:

```
public class Objekt implements Cloneable, Comparable {
  private int zahl;

  public Objekt(int n) {
    zahl = n;
  }

  public boolean equals(Object object) {
    if (!(object instanceof Objekt))
      return false;
    System.out.println("Vergleich der Referenzen auf geklonte " + "Objekte : " + super.equals(obj
    Objekt kopie = (Objekt) object;
    return (zahl == kopie.zahl);
  }

  public int compareTo(Object object) {
    Objekt kopie = (Objekt) object;
    return kopie.zahl - this.zahl;
  }
}
```

Finden Sie alle Fehler und ggf. sonstige Unzulänglichkeiten dieses Codes. Schreiben Sie bitte Testfälle, die diese Fehler aufdecken und schreiben Sie eine korrekte und bessere Version.

G.4.3 Eine abstruse Klasse

Finden Sie alle Fehler oder Unzulänglichkeiten in folgendem Codeausschnitt:

```
public class schrotti {
int SchrottNummer;
public Schrotti(){
SchrottNummer++;
}
public int equals(schrotti schrotti){
```

```
if(SchrottNumber - schrotti.SchrottNummer <10)
return 0;
else
return SchrottNummer;
}
}
```

G.4.4 Kunde - Auftrag

Gegeben sei eine 1:*-Assoziation zwischen Kunde und Auftrag wie in folgendem Klassendiagramm:

Der folgende Code soll diese implementieren:

```
public class Kunde {
   string name;
   string adresse;

   Auftrag []auftrag;
   Kunde(string name,
         string adresse){
      auftrag=new auftrag[42];
   }
   void addAuftrag (auftrag a)
   auftrag [anzahlauftraege ++]
   storniere Auftrag (auftrag a);
}
```

```
public class Auftrag<kunde> {
   Date datum;
   kunde[] kunde;
   {
      assert (kunde != null);
      this.kunde = this.kunde;
      Kunde.addAuftrag(this);
   }
}
```

Identifizieren Sie bitte alle Fehler und sonstigen Unzulänglichkeiten in diesem Code. Sie müssen den Code nicht verbessern bzw. korrigieren, sondern nur angeben, was an diesem Code zu Compiler-Fehlern oder -Warnungen führt und nicht den Codekonventionen entspricht. Javadoc und andere Kommentarkonventionen sollen Sie *nicht* berücksichtigen.

G.5 Schleifen

1. Erläutern Sie bitte, was die folgende Methode ausgibt!

   ```
   10 public void loop(){
   20    int j = 0;
   30    for (int i = 0; i < 100; i++)
   40         j = j++;
   50    System.out.println(j);
   60 }
   ```

2. Analog mit `j--`.

3. Geben Sie bitte möglichst viele einfache Möglichkeiten an, eine Endlosschleife zu schreiben!

4. Was passiert in folgendem Code?

   ```
   public class Shifty {
   ```

```
    public static void main(String[] args) {
        int i = 0;
        while (-1 << i != 0)
            i++;
        System.out.println(i);
    }
}
```

G.6 Datum und Uhrzeit

Hier folgen einige Programmieraufgaben, die etwas aufwändiger sein können, bzw. bei denen ich den Aufwand noch nicht abschätzen kann.

G.6.1 Maya Kalender

Im Zusammenhang mit einigen Weltuntergangsprophezeiungen wurde Ende des Jahres 2011 bzw. Anfang des Jahres 2012 der 21.12.2012 genannt. An diesem Tag endet nämlich angeblich der Maya-Kalender. Zumindest der Maya-Kalender in emacs geht allerdings ganz einfach weiter. Sie sollen Code schreiben, der es ermöglicht Daten im Maya-Kalender anzuzeigen. Dieser basiert auf folgendem Schema:

1	Tag	1	kin
20	kin	1	Uinal
18	Uinal	1	Tun
20	Tun	1	Katun
20	Katun	1	Baktun

1 Tun ist also fast ein Jahr (360 Tage). Um mit der Umlaufzeit der Erde um die Sonne ungefähr synchron zu bleiben, wurden im Maya Kalender noch 5 weitere Tage eingeschoben [Scr68].

Die Aufgabe lässt sich ausbauen: Weiter Kalender sind etwa der Julianische, der Hebräische, Persisch, Bahai, Mad ....

G.7 Wahr oder Falsch

Punkte: 20 Bitte kreuzen Sie für die folgenden Aussagen an, ob sie wahr (true) oder falsch (false) sind.

		true	false
1.	Ein Interface enthält alle Operationen, die eine Klasse implementieren muss.		
2.	Eine Klassenmethode kann auf Instanzattribute eines Objekts der Klasse zugreifen.		
3.	Das Schlüsselwort `this` kann benutzt werden, um durch `this(...)` mit geeigneten Parametern einen anderen Konstruktor aufzurufen.		
4.	Eine Variable in Java ist in dem Block gültig, in dem sie deklariert wird.		
5.	Eine long-Variable ist 8 Byte groß.		
6.	Eine double-Variable ist 4 Byte groß		
7.	Mit dem Schlüsselwort super(...) mit geeigneten Parametern kann man im Konstruktor einen Konstruktor der Superklasse aufrufen.		
8.	Java unterstützt die Definition von eigenen Datentypen.		
9.	Java unterstützt Mehrfachvererbung für Interfaces.		
10.	Wenn ein Objekt mit *new* erzeugt wurde, muss es mit *delete* zerstört werden, wenn es nicht mehr benötigt wird.		

		true	false
11.	Package-Namen werden nach Konvention aus Kleinbuchstaben gebildet.		
12.	Kommentare in Java beginnen mit // oder /*.		
13.	Java generiert für eine Klasse immer einen default-Konstruktor.		
14.	Durch *this(...)* kann in einer Methode einer Klasse der geeignete Konstruktor aufgerufen werden.		
15.	Wird eine lokale Variable nicht intialisiert, so gibt es eine Runtime-Exception.		
16.	Attribute einer Klasse werden in der Reihenfolge ihrer Deklaration mit den entsprechenden default-Werten initialisiert.		
17.	Konstruktoren können in Unterklassen überschrieben werden.		
18.	Das Keyword *final* vor nicht-statischen Attributen kennzeichnet Werte, die beim Erzeugen von Objekten festgelegt werden und danach nicht mehr geändert werden können.		
19.	Wenn eine Klasse *equals* implementiert, dann muss sie auch *compareTo* implementieren.		
20.	Eine Klassenattribut wird durch das Schlüsselwort *static* spezifiziert, der Wert des Attributs gilt dann für alle Objekte einer Klasse.		

G.8 Zuweisungen und Rechnungen

Geben Sie bitte an, welche Werte die folgenden (lokalen Variablen) nach der Initialisierung haben! Begründen Sie Ihre Antwort.

1. long a = 42 / 11; **1 P**

   a = _____

2. float b = (float)(42 / 11); **1 P**

   b = _____

3. float c = (float)42 / 11; **1 P**

   c = _____

4. double d = (double)42 / 11; **1 P**

   d = _____

5. float e = 42.00 -11.1; **2 P**

   e = _____

6. byte f = (1*1 - 1/1) /* + 1.0 */ * 13; **2 P**

   f = _____

G.9 Puzzels

1. Schreiben Sie bitte eine Klasse *A*, so dass der Konstruktor der folgenden Klasse *B* „Win“ ausgibt!

```
class B extends A {
  B(Long i){
    new B(i/Long.compare(i,i));
    System.out.println("Win");
  }
}
```

Quelle: Wouter Coekaert, http://t.co/dabnwJIlnP, 2. Feb. 2015

Literaturverzeichnis

[Ada81] Douglas Adams. *Per Anhalter durch die Galaxis*. Roger & Bernhard, München, 1981. Ein Kultbuch der 80er Jahre. Der erste Band einer Trilogie, die inzwischen aus 5 Bänden besteht.

[AS85] Harold Abelson und Gerald Jay Sussmann. *Structure and Interpretation of Computer Programs*. MIT Press, McGraw-Hill, Cambridge, MA, London, England, New York, NY, St. Louis, MO, San Francisco, CA, Montreal, Toronto, 1985.

[ASU86] Alfred V. Aho, Ravi Sethi und Jeffrey D. Ullman. *Compilers – Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1986. Das „Drachenbuch".

[Batff] Bozhidar Batsov. Ruby Style Guide. https://github.com/bbatsov/ruby-style-guide, 2011ff. Zuletzt besucht am 27.07.2015.

[BG05] Joshua Bloch und Neal Gafter. *Java™Puzzlers: Traps, Pitfalls, and Corner Cases*. Addison-Wesley, Reading, MA, 2005, 378 Seiten. ISBN 0-321-33678-X.

[Bir10] Richard Bird. *Pearls of Functional Algorithm Design*. Cambridge University Press, Cambridge, UK, 2010, xi+277 Seiten. 98 Artikel zu diesem Thema stehen in der elektronischen Bibliothek der HAW zur Verfügung (Journal of Functional Programming).

[Blo08] Joshua Bloch. *Effective Java: A Programming Language Guide*. Addison-Wesley Longman, Amsterdam, 2. Auflage, 2008, xix+327 Seiten. ISBN 0-321-35668-3.

[Blo18] Joshua Bloch. *Effective Java: A Programming Language Guide*. Addison-Wesley, Boston,Columbus, Indianapolis, 3. Auflage, 2018, xx+392 Seiten.

[Bre70] Bertolt Brecht. *Der gute Mensch von Sezuan*. Nr. 73 in edition suhrkamp. Suhrkamp, Frankfurt a. M., 12. Auflage, 1970, 144 Seiten. 431.-480. Tausend.

[BRJ98] Grady Booch, James Rumbaugh und Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, Reading, MA, 1. Auflage, 1998, xxii+482 Seiten. ISBN 0-201-57168-4. Eine gut zu lesende Einführung in die UML und ihre Anwendung. Es gibt eine deutsche Übersetzung [BRJ99].

[BRJ99] Grady Booch, James Rumbaugh und Ivar Jacobson. *Das UML Benutzerhandbuch*. Addison-Wesley, Bonn, Paris, Reading, MA, 1. Auflage, 1999. ISBN 3-8237-1486-0. Deutsche Übersetzung von [BRJ98].

[CN93] Peter Coad und Jill Nicola. *Object-Oriented Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1993. Die drei Bücher [CY90a], [CY90b] und dieses stellen das Vorgehen für objektorientierte Analyse, Design und Programmierung dar, das von Coad und Yourdon propagiert wird. Nach der Intention der Autoren sollen sie in beliebiger Reihenfolge gelesen werden können. Die ersten beiden habe ich nach schneller Lektüre beiseite gelegt. Das dritte Buch hat mir besser gefallen: Instruktive Beispiele mit C++ und Smalltalk Programmen auf Diskette. Auch die im Buch abgedruckten Beispiele

funktionierten bei mir! Sehr locker geschrieben und viele Redundanzen (mit den anderen beiden). Die Prinzipien sind nicht sehr präzise, sondern mehr dem gesunden Menschverstand entsprechend formuliert. Manche werden von anderen Methodikern abgelehnt. Am Ende ist eine Liste von 69 (!) OOP Prinzipien zusammengestellt. Etwas mehr Struktur und Präzision hätte gut getan. Die Dicke des Buches trügt etwas: Viel Platz wird für die Durchführung der Beispiele in C++ und Smalltalk verwandt. Aber nichts desto trotz: Zum Zeitpunkt des Erscheinens lesenswert als Einführung, auch in C++ und Smalltalk. Heute ist die Notation überholt und würde mehr verwirren als erklären.

[CY90a] Peter Coad und Edward Yourdon. *Object Oriented Analysis*. Prentice-Hall, Englewood Cliffs, NJ, 1990.

[CY90b] Peter Coad und Edward Yourdon. *Object Oriented Design*. Prentice-Hall, Englewood Cliffs, NJ, 1990.

[dav] the Da Vinci Machine Project. Zuletzt besucht: 14.01.2015.

[DG04] Jeffrey Dean und Sanjay Gihemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI 2004: 6_{th} Symposium On Operating Systems Design & Implementation*, San Francisco, CA, Dec. 5_{th} 2004.

[Doc15] Docs.oracle.com. Networking Basics, 2015. Zuletzt besucht : 09.02.2015.

[ecm15a] List of ECMAScript engines, 2015. Zuletzt besucht: 14.01.2015.

[ECM15b] Wikipedia Artikel: ECMAScript, 2015. Zuletzt besucht: 14.01.2015.

[Erl05] Thomas Erl. *Service-oriented architecture*. Prentice Hall Professional Technical Reference, Upper Saddle River, NJ, 2005.

[Ess08] Friedrich Esser. *Java 6 Core Techniken. Essentielle Techniken für Java-Apps*. Oldenbourg Wissenschaftsverlag, München, 2008, xii+455 Seiten.

[Fow99] Martin Fowler. *Refactoring. Improving the Design of Existing Code*. Object Technology Series. Addison Wesley Longman, Reading, MA, 1. Auflage, 1999, xxi+431 Seiten. ISBN 0-201-48567-2. Ein Katalog von Refaktorisierungen in Java. Bewährte Techniken zur Verbesserung von Code, die zum Teil auf die altbekannten Mechanismen des Modul- oder Komponentendesigns zurückgehen und hier im objektorientierten Gewandt präsentiert werden. Bei einigen der beschriebenen Refaktorisierungen sollte man meines Erachtens genauer auf die Vor- und Nachteile eingehen.

[Fow00] Martin Fowler. *Refactoring. Wie Sie das Design vorhandener Software verbessern.* Professionelle Softwareentwicklung. Addison-Wesley, Bonn, Paris, Reading, MA, 1. Auflage, 2000. ISBN 3-8273-1630-8. Deutsche Übersetzung des 3. korrigierten Nachdrucks von [Fow99].

[Gam92] Erich Gamma. *Objektorientierte Software-Entwicklung am Beispiel von ET++*. Springer-Verlag, Berlin, Heidelberg, New York, NY, 1992. Nach Ansicht von Heinz Züllighoven „eines der besten Bücher zur objektorientierten Konstruktion mit C++. Ein Muß für die objektorientierten Programmierung.“ Dieses Buch basiert auf der Dissertation des Autors an der ETH Zürich. Ich weiß nicht, wieweit dieses Buch durch das neuere [GHJV95] von Gamma et al. überholt ist.

[Gar90] Harry Garms. *Pflanzen und Tiere Europas. Ein Bestimmungsbuch.* dtv, München, 11 Auflage, 1990, vii+343 Seiten. ISBN 3-423-03013-5.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson und Vlissides, John M. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994, Fourth printing, September 1995. ISBN 0-201-63361-2. Ein Katalog von Mustern, mit denen häufig vorkommenden Designaufgaben in C++ oder anderen objektorientierten Programmiersprachen gelöst werden können. Vieles davon ist an anderer Stelle beschrieben, aber dies ist eine kompakte Zusammenfassung, die auch die Anwendung einiger dieser Patterns an einer Fallstudie zeigt. Die Lösungen sind praxiserprobt und helfen bei der guten Strukturierung auch (aber keineswegs nur) von C++ Software. Sicherlich eines der einflussreichsten Informatik Bücher der letzten Jahre. Ein großer Teil der Arbeit wurde bei der Firma Taligent durchgeführt. Es gibt eine deutsche Übersetzung [GHJV96].

[GHJV96] Erich Gamma, Richard Helm, Ralph Johnson und Vlissides, John M. *Entwurfsmuster - Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, Bonn, Paris, Reading, MA, 1996. ISBN 3-89319-950-0. Deutsche Übersetzung von [GHJV95].

[GHM+15] Martin Gudgin, Marc Hadley, Noah Mendelssohn, Jean-Jacques Moreau, Henrik Nielsen, Anish Karmarkar und Yves Lafon. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition), 2015. Zuletzt besucht: 09.02.2015.

[git15] Bitbucket: git Repository, 2015.

[GJS+11] James Gosling, Bill Joy, Guy Steele, Gilad Bracha und Alex Buckley. *The Java Language Specification. Java SE 7 Edition.* Oracle, http://download.oracle.com/otn-pub/jcp/jls-7-mr3-fullv-oth-JSpec/JLS-JavaSE7-Full.pdf, 2011, 586 Seiten. Besucht: 23.12.2011.

[GJS+14] James Gosling, Bill Joy, Guy Steele, Gilad Bracha und Alex Buckley. *The Java Language Specification. Java SE 8 Edition.* Oracle, http://download.oracle.com/otn-pub/jcp/jls-7-mr3-fullv-oth-JSpec/JLS-JavaSE7-Full.pdf, 2014, 760 Seiten. Zuletzt besucht: 09.06.2014.

[GJS+17] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley und Daniel Smith. *The Java Language Specification. Java SE 9 Edition.* Oracle, Redwood City, CA, 2017, 808 Seiten. Zuletzt besucht: 05.01.2018.

[GJS+18] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley und Daniel Smith. *The Java Language Specification. Java SE 10 Edition.* Oracle, Redwood City, CA, 2018, 808 Seiten. Zuletzt besucht: 05.01.2018.

[GJSB05] James Gosling, Bill Joy, Guy Steele und Gilad Bracha. *The Java Language Specification. Third Edition*. Addison-Wesley, Reading, MA, 2005, 688 Seiten. ISBN 0-321-24678-0. Online Verfügbar als html und als pdf (zum einmaligen Druck) von java.sun.com/docs/books/jls.

[Goo] Google Corporation. Google Java Code Style. Zuletzt besucht: 30.01.2018.

[Hel61] Joseph Heller. *Catch-22*. Simon and Schuster, New York, 2. printing Auflage, 1961, 463 Seiten. ISBN 0-671-12805-1.

[HM05] Steffen Heinzl und Markus Mathes. *Middleware in Java*. Springer-Verlag, Berlin Heidelberg New York, 2005. Aufl. Auflage, 2005. ISBN 978-3-528-05912-5.

[HMHG07] Cornelia Heinisch, Frank Müller-Hofmann und Joachim Goll. *JAVA als erste Programmiersprache: Vom Einsteiger zum Profi*. B. G. Teubner, GWV Fachverlage GmbH, Wiesbaden, 5. überarbeitete und erweiterte Auflage, 2007, 1235 Seiten. ISBN 3-8351-0147-1.

[HS92] Brian Henderson-Sellers. *A Book of Object-Oriented Knowledge. Object Oriented Analysis, Design and Implementation: A New Approach to Software Engineering.* Prentice-Hall, Englewood Cliffs, NJ, 1992. ISBN 0-13-059445-8. Eine kurze Einführung. Der Umfang trügt etwas, da auch Folien für einen Kurs enthalten sind. Die Entwicklung objektorientierte Analyse- und Design-Methoden ist inzwischen deutlich fortgeschritten, so dass eine überarbeitete Neuauflage nützlich wäre. Nichts desto trotz eine lesenwerte Einführung.

[HV04] Peter A. Henning und Holger Vogelsang. *Taschenbuch Programmiersprachen.* Fachbuchverlag Leipzig im Carl hanser Verlag, Leipzig, 2004, 538 Seiten. ISBN 3-446-22580-3.

[jav] Java - Networking (Socket Programming) Tutorial. Zuletzt besucht: 09.02.2015.

[Job06] Fritz Jobst. *Programmieren in Java.* Carl Hanser Verlag, München, Wien, 5., überarbeitete Auflage, 2006, 1 CD, xvi+544 Seiten. ISBN 3-446-40401-5.

[jsr11] JSR 292: Supporting Dynamically Typed Languages on the JavaTM Platform, Juli 2011. Zuletzt besucht: 14.01.2015.

[Knu57] Donald E. Knuth. The Potrzebie System of Weights and Measures. *MAD*, 1(33):36–37, Juni 1957.

[Knu73] Donald E. Knuth. *The Art of Computer Programming I: Fundamental Algorithms.* Addison-Wesley, Reading, MA, 2. Auflage, 1973, xix+634 Seiten. ISBN 0-201-03821-8. Ein Klassiker, den jeder Informatiker gelesen haben sollte. Der Stil bis hin zu den Aufgaben ist legendär. Dieser Band ist im Gegensatz zu den weiteren auch als Paperback lieferbar.

[Knu97a] Donald E. Knuth. *The Art of Computer Programming I: Fundamental Algorithms.* Addison-Wesley, Reading, MA, 3. Auflage, 1997, xix+650 Seiten. ISBN 0-201-89683-4. Neue, erheblich überarbeitete Auflage von [Knu73].

[Knu97b] Donald E. Knuth. *The Art of Computer Programming II, Seminumerical Algorithms.* Addison-Wesley, Reading, MA, 3. Auflage, 1997. ISBN 0-201-89684-2. Die letzte Revision vor Fertigstellung der auf den dritten folgenden Bände.

[Knu07] Donald Ervin Knuth. *The Art of Computer Programming Volume 4, pre-Fascicle 0B. A Draft of Section 7.1.2: Boolean Evaluation.* Addison-Wesley, 2007, v+61 Seiten. Zeroth printing (revision 7), 20 May 2007. www-cs-faculty.stanford.edu/ knuth/, seit dem 01.05.2008 nicht mehr online Verfügbar.

[Knu08a] Donald Ervin Knuth. *The Art of Computer Programming Volume 4, pre-Fascicle 0C. A Draft of Section 7.1.1: Boolean Basics.* Addison-Wesley, 2008, iv+84 Seiten. Zeroth printing (revision 13), 28 April 2007. www-cs-faculty.stanford.edu/ knuth/, seit dem 01.05.2008 nicht mehr online Verfügbar.

[Knu08b] Donald Ervin Knuth. *The Art of Computer Programming Volume 4, pre-Fascicle 1A. A Draft of Section 7.1.3: Bitwise Tricks and Techniques.* Addison-Wesley, 2008, v+118 Seiten. Zeroth printing (revision 3), 16 February 2008. www-cs-faculty.stanford.edu/ knuth/,Seit dem 01.05.2008 nicht mehr online Verfügbar.

[Knu11] Donald Ervin Knuth. *Selected Papers on Fun & Games.* CSLI, Stanford, 2011, xvii+741 Seiten. Viele interessante Artikel, u. a. der erste, den Don publizierte.

[Knuff] Donald E. Knuth. *The Art Of Computer Programming.* Addison-Wesley, Reading, MA, 1973ff. Das Lebenswerk von Don Knuth. Ich hoffe er bleibt solange gesund bis er es abgeschlossen hat und darüber hinaus! Ich erinnere mich aber an die Besprechung von Jerrold E. Marsden[Mar80] von Jean Dieudonnés *Foundations of Analysis*:

This... illustrates a basic theorem in mathematical writing which even experts find hard to swallow: *to find the true length (measured in pages or years) of a writing project multiply your initial estimate by a factor of at least three.* In fact, if one takes a project with large enough scope, the writing process can become stationary: *even though you write feverishly, your project is at all times half-completed.* Dieudonnés position is probably almost as bad as the theorem indicates and could even be as bad as stationary, ... Diese Beschreibung könnte auch auf dieses Werk zutreffen.

[KR88] Brian Kernighan und Dennis Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, 2. Auflage, 1988, vii+272 Seiten. ISBN 0-13-110370-9.

[Krü07] Guido Krüger. *Handbuch der Java-Programmierung.* Addison-Wesley, Bonn, Paris, Reading, MA, 5. Auflage, 2007, 1280 Seiten. ISBN 3-8273-2373-8. DVD, html verfügbar über www.javabuch.de.

[Käs67] Erich Kästner. *kurz und bündig. Epigramme.* Deutsche Buchgemeinschaft, Berlin, Darmstadt, Wien, 1967, 112 Seiten.

[LeaoJ] Doug Lea. A Java Fork/Join Framework. http://gee.cs.oswego.edu/dl/papers/fj.pdf, oJ. Nachgeschlagen: 11.11.2011.

[Lek52] Cornelis Gerrit Lekkerkerker. Voorstelling van natuurlijke getallen door een som van getallen van Fibonacci. *Simon Stevin*, 27:190–195, 1952.

[Lin61] Astrid Lindgren. *Kalle Blomquist lebt gefährlich.* Oettinger, Hamburg, 1961.

[LYBB14] Tim Lindholm, Frank Yellin, Gilad Bracha und Alex Buckley. *Java^{TM} Virtual Machine Specification, Java SE 8 Edition.* Oracle Corparation, Redwood City, CA, 2014, 612 Seiten. Zuletzt besucht: 09.06.2014.

[Mar80] Jerrold E. Marsden. Book Review: Treatise on analysis. *Bulletin of the American Mathematical Society*, 3(1, July 1980):719–724, 1980.

[Mes71] Herbert Meschkowski. *Mathematisches Begriffswörterbuch.* Nr. 99 in Hochschultaschenbücher. Bibliographisches Institut, Mannheim, Wien, Zürich, 3. erweiterte Auflage, 1971. Inzwischen natürlich etwas antik. Ich benutze es aber immer noch, wenn ich etwas mathematisches nachschlagen möchte, über das ich keine Spezialliteratur habe.

[Mey88a] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Mey88b] Bertrand Meyer. *Objektorientierte Software Entwicklung.* Hanser, Prentice-Hall, München, Wien, London, 1988. Deutsche Übersetzung von [Mey88a]. Einer der Klassiker über objektorientierte Entwicklung. Stark mit Blick auf EIFFEL geschrieben.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall PTR, Upper Saddle River, NJ, 2. Auflage, 1997. ISBN 0-13-629155-4. Eine dramatisch erweiterte Ausgabe von [Mey88b].

[Mis98] Frederick C. Mish, Hrsg. *Webster's New Collegiate Dictionary.* Merriam-Webster, Springfield, MA, 10. Auflage, 1998, xxx+1557 Seiten. ISBN 0-87779-708-0. Nicht nur nach meiner Ansicht das beste Dictionary der amerikanischen (und auch der englischen) Sprache. Preis in den USA unter US$ 20. Einige Hinweise sind aber am Platz: Nicht mit anderen „Websters“'s verwechseln, die z.T noch dicker und noch billiger, aber nicht unbedingt besser sind. Die verwendete Lautschrift ist nicht die international übliche.

[Mös05] Hanspeter Mössenböck. *Sprechen Sie Java. Eine Einführung in das systematische Programmieren.* Verlag für digitale Technologie GmbH, Heidelberg, 3. überarbeitete und erweiterte Auflage, 2005, 327 Seiten. ISBN 3-8986-4362-X.

[MS09] Christoph Meinel und Harald Sack. *Digitale Kommunikation - Vernetzen, Multimedia, Sicherheit.* Springer Science und Business Media, Berlin Heidelberg, 2009. Aufl. Auflage, 2009.

[nhh15] Nashorn and Shell Scripting, 2015. Zuletzt besucht 23.01.2015.

[OK99] Bernd Owsnicki-Klewe. *Algorithmen und Datenstrukturen*, Band 5 von *Vorlesungen zum Informatik- und Ingenieurstudium.* Wißner Verlag, 3. veränderte Auflage, 1999, xiv+498 Seiten. ISBN 3-446-22075-5.

[Ora14] Oracle. Getting Started with Web Applications - The Java EE 6 Tutorial, 2014. Zuletzt besucht: 10.02.2015.

[OW07] Andy Oram und Greg Wilson, Hrsg. *Beautiful Code. Leading Programmers Explain How They Think.* O'Reilly & Associates, Sebastopol, CA, 2007, xxi+593 Seiten. ISBN 0-596-51004-7.

[Pan08] Sven Eric Panitz. *Java will nur spielen. Programmieren mit Spaß und Kreativität.* Vieweg+Teubner, Wiesbaden, 2008, xi+245 Seiten. Der Ansatz, mit motivierenden Spielen in die Programmierung einzuführen hat mir gut gefallen. Allerdings kommen dabei einige wichtige Dinge etwas zu kurz. Ich versuche Teile des Buches in Praktika zu nutzen. Die Sourcen stellt der Autor über www.sveneric.de/ludens/index.html zur Verfügung.

[pro14] Project Nashorn, January 2014.

[RJB99] James Rumbaugh, Ivar Jacobson und Grady Booch. *The Unified Modeling Language Reference Manual.* Object Technology Series. Addison-Wesley, Reading, MA, 1. Auflage, 1999, xxii+482 Seiten. ISBN 0-201-30998-X. CD-ROM mit den Dokumenten zur UML 1.3 wird nachgeliefert, wenn man den Berechtigungsnachweis einschickt.

[Rot60] Karl Rottmann. *Mathematische Formelsammlung*, Band 13 von *Hochschultaschenbücher.* Bibliographisches Institut, Mannheim, Wien, Zürich, 2. Auflage, 1960, 176 Seiten. ISBN 3-411-00013-9.

[RSSW06a] Dietmar Ratz, Jens Scheffler, Detlef Sees und Jan Wiesenberger. *Grundkurs Java. Band 1: Der Einstieg in Programmierung und Objektorientierung.* Carl Hanser Verlag, München, Wien, 3. Auflage, 2006, 1 CD, 489 Seiten. ISBN 3-446-40493-7.

[RSSW06b] Dietmar Ratz, Jens Scheffler, Detlef Sees und Jan Wiesenberger. *Grundkurs Java. Band 2: Einführung in die Programmierung kommerzieller Systeme.* Carl Hanser Verlag, München, Wien, 2., aktualisierte und überarbeitete Auflage, 2006, 448 Seiten. ISBN 3-446-40494-5.

[Sag02] Carl Sagan. *Cosmos.* Random House, NY, 2002, 384 Seiten.

[Scr68] Christoph J. mit Unterstützung von Dormer Ellis Scriba. *The Concept Of Number.* Nr. 925/825a in Hochschulskripten. Bibliographisches Institut, Mannheim, Zürich, 1968, i+216 Seiten.

[scr14] JavaTM Scripting Enhancements, 2014. Zuletzt besucht: 14.01.2015.

[Sea17] Robert Seacord. NCC Group Whitepaper Combating Java Deserialization Vulnerabilities with Look-Ahead Object Input Streams (LAOIS). https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2017/june/ncc_group_combating_java_deserialization_vulnerabilities_with ahead_object_input_streams1.pdf, June 15, 2017. Zuletzt bescuht am 30.04.2018. pdf in vorlesungen/pre/material/java.

[sgu15] The Java Scripting API, 2015. Zuletzt besucht: 14.01.2015.

[Sha15] Kishori Sharan. *Learn JavaFX: Building User Experience and Interfaces with Java 8.* Apres Media, New York, NY, 2015, 1210 Seiten. ISBN 978-1-4842-1142-7. Electronic Edition.

[som15] ScriptObjectMirror, 2015. Zuletzt besucht: 20.01.2015.

[Str94a] Bjarne Stroustrup. *The Design and Evolution of C++.* Addison-Wesley, Reading, MA, 1994, x+461 Seiten. ISBN 0-201-54330-3. Das „Eichenbuch“. Eine hervorragende Darstellung, warum was in C++ wie funktioniert und nicht anders. An vielen Stellen wird der Bezug zwischen Designprinzipien und Grundprinzipien der Programmiersprache hergestellt. Viele andekdotische Details aus der Entstehungsgeschichte der Sprache. Es gibt eine deutsche Übersetzung [Str94b].

[Str94b] Bjarne Stroustrup. *Design und Entwicklung von C++.* Addison-Wesley, Bonn, Paris, Reading, MA, 1994. Deutsche Übersetzung von [Str94a].

[Str13] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, 4. Auflage, 2013, 1368 Seiten.

[Svo16] David Svoboda. Exploiting Java Deserialization for Fun and Profit, 2016.

[Tan03] Andrew S. Tanenbaum. *Computernetzwerke.* Pearson Studium, München, 4. aktualisierte Auflage, 2003.

[Ull11] Christian Ullenboom. *Java ist auch eine Insel. Programmieren mit der Java Standard Edition Version 6.* Galileo, Bonn, 10., aktualisierte Auflage, 2011, 1308 ,mit DVD Seiten.

[Ull14] Christian Ullenboom. *Java SE 8 Standard-Bibliothek. Das Handbuch für Java-Entwickler.* Galileo, Bonn, 2. Auflage, 2014, 1448 Seiten.

[VGC+14] Johan Vos, Weiqi Gao, Stephen Chin, Dean Iverson und James Weaver. *Pro JavaFX 8. A Definite Guide to Building Desktop, Mobile ,and Embedded Java Clients.* Apress, New York, NY, 2014, 604 Seiten. ISBN 978-1-4302-6575-7. Electronic Edition.

[War02] Henry S. jr. Warren. *Hacker's Delight.* Addison-Wesley, Reading, MA, 2002, 320 Seiten. ISBN 0-201-91465-4.

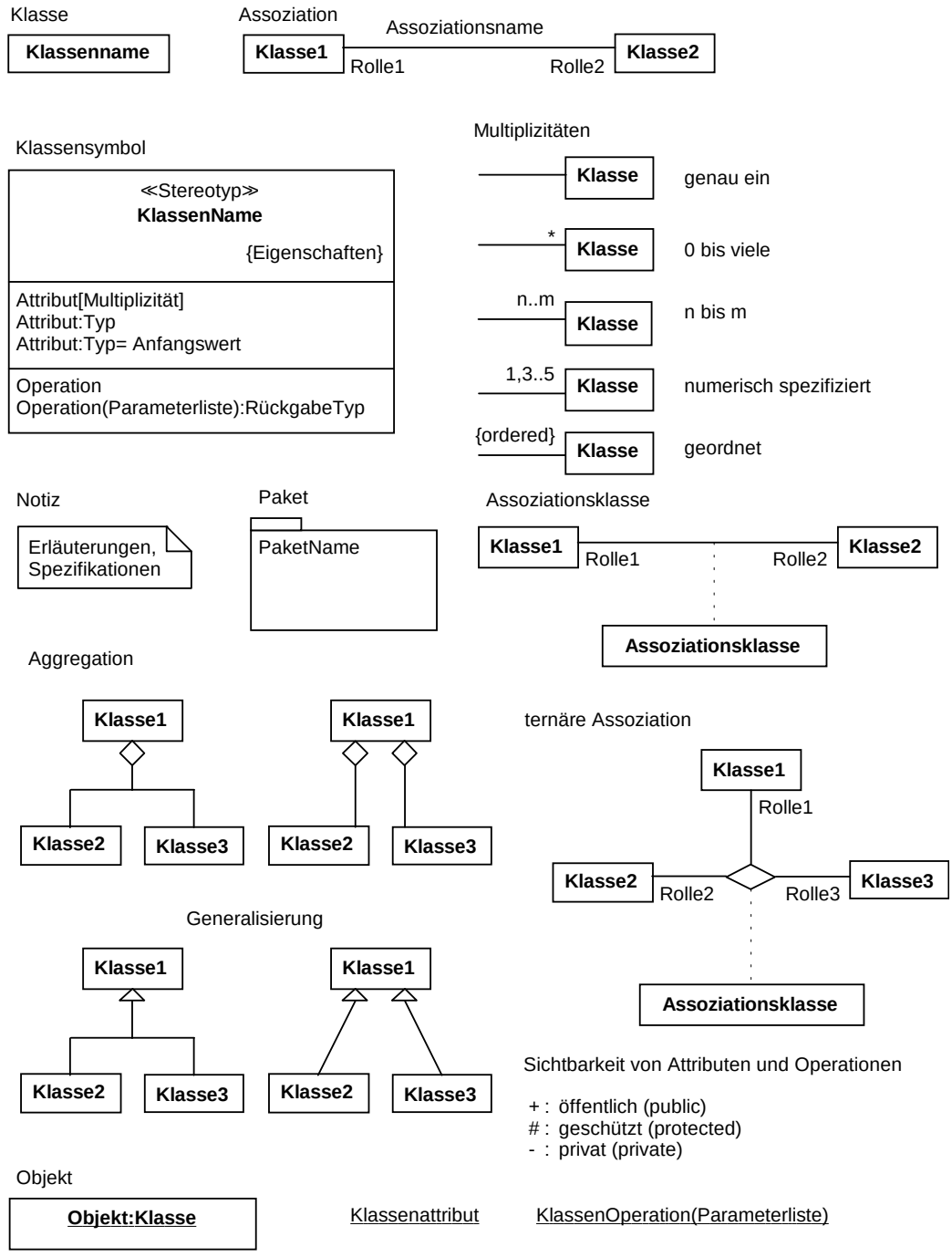
[Wes01] Ralph Westfall. Hello, World Considered Harmful. It all starts at the beginning: OO programming learned naturally, not procedurally. *Communications of the ACM*, 44(10):129–130, Oktober 2001. Der Punkt, den der Autor herausstreicht, ist m. E. völlig richtig beobachtet, die Verbesserung aber kaum wesentlich und mach einen anderen Fehler. Auch Klassenoperationen sollten sinnvoll verwendet werden.

[Woo98] Bobby Woolf. *Null Object*, Kapitel 1, S. 5–18. Addison-Wesley, 1998.

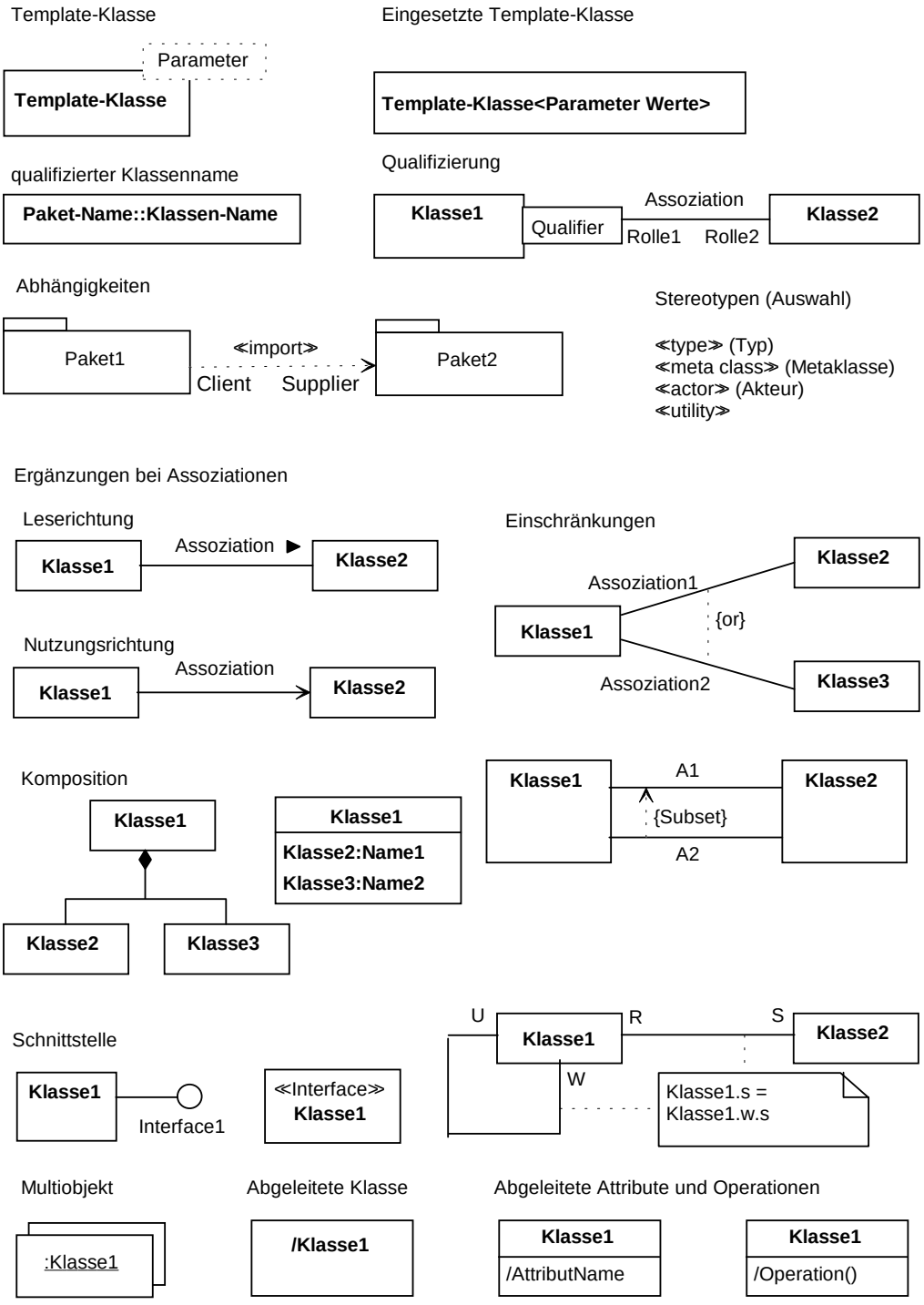
[Zak] Sharon Zakhour. Why is Swing Called Swing? http://blogs.oracle.com/thejavatutorials/entry/why_is_swing_called_swing.

[Zec72] Edouard Zeckendorf. A Generalized Fibonacci Numeration. *Fibonacci Quarterly*, 10:365–372, 1972.

Notation für Klassen und Assoziationen - 1 -



Notation für Klassen und Assoziationen - 2 -



Index