
Java 7

Christian Ullenboom
1. Ausgabe, Januar 2012

Fortgeschrittene Programmierung

U_BW_8605-2



HERDT

Impressum

Matchcode: U_BW_8605-2

Autor: Christian Ullenboom

Redaktion: Tina Wegener

Produziert im HERDT-Digitaldruck

1. Ausgabe, Januar 2012

Herausgeber: HERDT-Verlag für Bildungsmedien GmbH
Am Kümmerling 21-25
55294 Bodenheim
Internet: www.herd़t.com
E-Mail: info@herdt.com

© Galileo Press GmbH, Bonn



Alle Rechte vorbehalten. Kein Teil des Werkes darf in irgendeiner Form (Druck, Fotokopie, Mikrofilm oder einem anderen Verfahren) ohne schriftliche Genehmigung des Verlags reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

Dieses Buch wurde mit großer Sorgfalt erstellt und geprüft. Trotzdem können Fehler nicht vollkommen ausgeschlossen werden. Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen.

Wenn nicht explizit an anderer Stelle des Werkes aufgeführt, liegen die Copyrights an allen Screenshots beim HERDT-Verlag. Sollte es trotz intensiver Recherche nicht gelungen sein, alle weiteren Rechteinhaber der verwendeten Quellen und Abbildungen zu finden, bitten wir um kurze Nachricht an die Redaktion.

Die in diesem Buch und in den abgebildeten bzw. zum Download angebotenen Dateien genannten Personen und Organisationen, Adress- und Telekommunikationsangaben, Bankverbindungen etc. sind frei erfunden. Eventuelle Übereinstimmungen oder Ähnlichkeiten sind unbeabsichtigt und rein zufällig.

Die Bildungsmedien des HERDT-Verlags enthalten Verweise auf Webseiten Dritter. Diese Webseiten unterliegen der Haftung der jeweiligen Betreiber, wir haben keinerlei Einfluss auf die Gestaltung und die Inhalte dieser Webseiten. Bei der Bucherstellung haben wir die fremden Inhalte daraufhin überprüft, ob etwaige Rechtsverstöße bestehen. Zu diesem Zeitpunkt waren keine Rechtsverstöße ersichtlich. Wir werden bei Kenntnis von Rechtsverstößen jedoch umgehend die entsprechenden Internetadressen aus dem Buch entfernen.

Die in den Bildungsmedien des HERDT-Verlags vorhandenen Internetadressen waren zum Zeitpunkt der Erstellung der jeweiligen Produkte gültig. Sollten Sie die Inhalte nicht mehr unter den angegebenen Adressen finden, sind diese eventuell inzwischen komplett aus dem Internet genommen worden oder unter einer neuen Adresse zu finden.

1 Dokumentationskommentare mit JavaDoc	4	6.4 Alternative Datenaustauschformate..... 135 6.5 Tokenizer 138
1.1 Zu diesem Buch.....	4	
1.2 Einführung in Dokumentations- kommentare.....	5	
1.3 Einen Dokumentationskommentar setzen	5	
1.4 Mit dem Werkzeug javadoc eine Dokumentation erstellen	7	
1.5 HTML-Tags in Dokumentations- kommentaren	7	
1.6 Generierte Dateien	7	
1.7 Dokumentationskommentare im Überblick.....	8	
1.8 JavaDoc und Doclets	9	
2 Techniken für den Umgang mit Zeichenketten.....	10	
2.1 Reguläre Ausdrücke.....	10	
2.2 Zerlegen von Zeichenketten.....	16	
2.3 Ausgaben formatieren	24	
2.4 Format-Klassen.....	26	
3 Generics<T>	32	
3.1 Einführung in Java Generics.....	32	
3.2 Umsetzen der Generics, Typlöschung und Raw-Types.....	42	
3.3 Einschränken der Typen über Bounds.....	49	
3.4 Typparameter in der throws-Klausel	51	
3.5 Generics und Vererbung, Invarianz	54	
4 Preferences (Voreinstellungen)	62	
4.1 Benutzereinstellungen	62	
4.2 Die Utility-Klasse System und Properties	68	
5 Threads und nebenläufige Programmierung	74	
5.1 Nebenläufigkeit	74	
5.2 Threads erzeugen	77	
5.3 Thread-Eigenschaften und -Zustände	80	
5.4 Der Ausführer (Executor) kommt.....	85	
5.5 Synchronisation über kritische Abschnitte	88	
5.6 Synchronisation über Warten und Benachrichtigen	100	
5.7 Datensynchronisation durch besondere Concurrency-Klassen	105	
5.8 Atomare Operationen und frische Werte mit volatile	109	
5.9 Threads in einer Thread-Gruppe	111	
5.10 Einen Abbruch der virtuellen Maschine erkennen	117	
6 Spezielle Streams und Serialisierung....	118	
6.1 Kommunikation zwischen Threads mit Pipes.....	118	
6.2 Prüfsummen	121	
6.3 Persistente Objekte und Serialisierung	123	
7 Netzwerkprogrammierung.....	140	
7.1 Grundlegende Begriffe	140	
7.2 URI und URL.....	141	
7.3 Host- und IP-Adressen	146	
7.4 Mit dem Socket zum Server.....	149	
7.5 Client-Server-Kommunikation.....	153	
8 Verteilte Programmierung mit RMI.....	158	
8.1 Entfernte Objekte und Methoden.....	158	
8.2 Java Remote Method Invocation	160	
8.3 Auf der Serverseite	162	
8.4 Auf der Clientseite.....	168	
8.5 Entfernte Objekte übergeben und laden.....	169	
8.6 Weitere Eigenschaften von RMI.....	170	
8.7 Daily Soap und das SOAP-Protokoll	171	
9 Datenbankmanagement mit JDBC.....	176	
9.1 Relationale Datenbanken	176	
9.2 Einführung in SQL	177	
9.3 Datenbanken und Tools	180	
9.4 JDBC und Datenbanktreiber	183	
9.5 Eine Beispielabfrage	185	
9.6 Mit Java an eine Datenbank andocken	189	
9.7 Datenbankabfragen	194	
9.8 Elemente einer Datenbank hinzufügen und aktualisieren.....	200	
9.9 Vorbereitete Anweisungen (Prepared Statements).....	203	
9.10 Transaktionen	205	
9.11 Metadaten	206	
9.12 Vorbereitete Datenbankverbindungen	209	
9.13 JPA-Beispiel mit der NetBeans-IDE.....	211	
10 Java Native Interface (JNI)	216	
10.1 Java Native Interface und Invocation-API.....	216	
10.2 Eine C-Funktion in ein Java-Programm einbinden	217	
10.3 Nativ die Stringlänge ermitteln	221	
10.4 Erweiterte JNI-Eigenschaften	222	
10.5 Einfache Anbindung von existierenden Bibliotheken.....	225	
10.6 Invocation-API.....	226	
11 Sicherheitskonzepte.....	228	
11.1 Zentrale Elemente der Java-Sicherheit	228	
11.2 Der Sandkasten (Sandbox)	229	
11.3 Sicherheitsmanager (Security Manager)....	229	
11.4 Signierung	236	
11.5 Verschlüsseln von Daten(-strömen).....	237	
Stichwortverzeichnis	240	

1 Dokumentationskommentare mit JavaDoc

In diesem Kapitel erfahren Sie

- ✓ wie Sie mit diesem Buch arbeiten
- ✓ warum Dokumentationskommentare benötigt werden
- ✓ wie Sie Dokumentationskommentare erstellen

Voraussetzungen

- ✓ Grundlagen zu Java

1.1 Zu diesem Buch

Aufbau und inhaltliche Konventionen des Buches

- ✓ Das Buch ist in verschiedene Bereiche unterteilt, um Ihnen so einen guten Überblick über den Inhalt des Buches zu geben und zudem das Festlegen der Lernschwerpunkte zu erleichtern.
- ✓ Am Anfang jedes Kapitels finden Sie die Lernziele und die empfohlenen Voraussetzungen.
- ✓ Die meisten Kapitel enthalten Übungen, mit deren Hilfe Sie die jeweiligen Inhalte einüben können.

Typografische Konventionen

Im Text erkennen Sie bestimmte Programmelemente an der Formatierung. So werden beispielsweise Ordnernamen immer *kursiv* geschrieben.

KAPITÄLCHEN kennzeichnen alle vom Anwendungsprogramm vorgegebenen Bezeichnungen

Kursivschrift kennzeichnet alle vom Anwender zugewiesenen Namen wie Dateinamen, Ordnernamen etc. sowie Hyperlinks und Pfadangaben.

Courier New kennzeichnet Programmcode und Zeichenfolgen, die vom Anwendungsprogramm ausgegeben oder in das Programm eingegeben werden.

Symbole



Besondere **Information** zur Programmbedienung



Besonders **praktische** und einfache Möglichkeit, eine Aktion durchzuführen; **Tipp**, wie eine Arbeit erleichtert werden kann



Warnhinweis bei Aktionen, die **unerwünschte** Wirkungen haben könnten

Beispiel-, Übungs- und Ergebnisdateien

Die Beispieldateien sowie die in den Übungen verwendeten Übungs- und Ergebnisdateien können Sie **kostenlos** von der Homepage des HERDT-Verlages unter <http://www.herd़t.com> herunterladen. Tragen Sie hierzu auf der Webseite in das Eingabefeld zur Suche den Matchcode `jav7f` ein, klicken Sie in der Trefferliste auf den Titel des Buches und starten Sie den Download, indem Sie auf der anschließend angezeigten Webseite den Link **Übungsdateien** anklicken.



Hinweise zu Soft- und Hardware

In den Beschreibungen des Buches wird von der Erstinstallation der Software Oracle Java 2 Platform, Standard Edition 7.0 (J2SE 7.0 JDK, Version 1.0) ausgegangen (<http://www.oracle.com/technetwork/java/javase/downloads/java-se-jdk-7-download-432154.html>).

Empfohlene Vorkenntnisse

Um sich problemlos fortgeschrittene Themen der Programmierung mit Java aneignen zu können, sollten Sie bereits Grundkenntnisse in Java 7 mitbringen, wie Sie im Buch **Java 7 - Grundlagen Programmierung** vermittelt werden.

1.2 Einführung in Dokumentationskommentare

Die Dokumentation von Softwaresystemen ist ein wichtiger, aber oft vernachlässigter Teil der Softwareentwicklung. Leider, denn Software wird im Allgemeinen öfter gelesen als geschrieben. Während des Entwicklungsprozesses müssen die Entwickler Zeit in Beschreibungen der einzelnen Komponenten investieren, besonders dann, wenn weitere Entwickler diese Komponenten in einer öffentlichen Bibliothek anderen Entwicklern zur Wiederverwendung zur Verfügung stellen. Um die Klassen, Schnittstellen, Aufzählungen und Methoden sowie Attribute gut zu finden, müssen sie sorgfältig beschrieben werden. Wichtig bei der Beschreibung sind der Typname, der Methodename, die Art und die Anzahl der Parameter, die Wirkung der Methoden und das Laufzeitverhalten. Da das Erstellen einer externen Dokumentation (also einer Beschreibung außerhalb der Quellcode datei) fehlerträchtig und deshalb nicht gerade motivierend für die Beschreibung ist, werden spezielle Dokumentationskommentare in den Java-Quelltext eingeführt. Ein spezielles Programm generiert aus den Kommentaren Beschreibungsdateien (im Allgemeinen HTML) mit den gewünschten Informationen. (Die Idee ist nicht neu. In den 1980er-Jahren verwendete Donald E. Knuth das WEB-System zur Dokumentation von TeX. Das Programm wurde mit den Hilfsprogrammen weave und tangle in ein Pascal-Programm und eine TeX-Datei umgewandelt.)

1.3 Einen Dokumentationskommentar setzen

In einer besonders ausgezeichneten Kommentarumgebung werden die **Dokumentationskommentare ("Doc Comments")** eingesetzt. Die Kommentarumgebung erweitert einen Blockkommentar und ist vor allen Typen (Klassen, Schnittstellen, Aufzählungen) sowie Methoden und Variablen üblich. Im folgenden Beispiel gibt JavaDoc Kommentare für die Klasse, Attribute und Methoden an:

```
com/tutego/insel/javadoc/Room.java
package com.tutego.insel.javadoc;

/**
 * This class models a room with a given number of players.
 */
public class Room
{
    /** Number of players in a room. */
    private int numberOfPersons;

    /**
     * A person enters the room.
     * Increments the number of persons.
     */
    public void enterPerson() {
        numberOfPersons++;
    }
}
```

```

/**
 * A person leaves the room.
 * Decrements the number of persons.
 */
public void leavePerson() {
    if ( numberOfPersons > 0 )
        numberOfPersons--;
}

/**
 * Gets the number of persons in this room.
 * This is always greater equals 0.
 *
 * @return Number of persons.
 */
public int getNumberOfPersons() {
    return numberOfPersons;
}
}

```

Kommentar	Beschreibung	Beispiel
@param	Beschreibung der Parameter	@param a A Value.
@see	Verweis auf ein anderes Paket, einen anderen Typ, eine andere Methode oder Eigenschaft	@see java.util.Date @see java.lang.String#length()
@version	Version	@version 1.12
@author	Schöpfer	@author Christian Ullenboom
@return	Rückgabewert einer Methode	@return Number of elements.
@exception/@throws	Ausnahmen, die ausgelöst werden können	@exception NumberFormatException
{@link Verweis}	Ein eingebauter Verweis im Text im Code-Font. Parameter wie bei @see	{@link java.io.File}
{@linkplain Verweis}	Wie {@link}, nur im normalen Font	{@linkplain java.io.File}
{@code Code}	Quellcode im Code-Zeichensatz - auch mit HTML-Sonderzeichen	{@code 1 ist < 2}
{@literal Literale}	Maskiert HTML-Sonderzeichen. Kein Code-Zeichensatz	{@literal 1 < 2 && 2 > 1}
@category	Für Java 7 oder 8 geplant: Vergabe einer Kategorie	@category Setter

Die wichtigsten Dokumentationskommentare im Überblick



Die Dokumentationskommentare sind so aufgebaut, dass der erste Satz in der Auflistung der Methoden und Attribute erscheint und der Rest in der Detailansicht:

```

/**
 * Ein kurzer Satz, der im Abschnitt "Method Summary" stehen wird.
 * Es folgt die ausführliche Beschreibung, die später im
 * Abschnitt "Method Detail" erscheint, aber nicht in der Übersicht.
 */
public void foo() { }

```

Weil ein Dokumentationskommentar `/**` mit `/*` beginnt, ist er für den Compiler ein normaler Block-kommentar. Die JavaDoc-Kommentare werden oft optisch aufgewertet, indem am Anfang jeder Zeile ein Sternchen steht - dieses ignoriert JavaDoc.

1.4 Mit dem Werkzeug javadoc eine Dokumentation erstellen

Aus dem mit Kommentaren versehenen Quellcode generiert ein externes Programm die Zieldokumente. Das JDK liefert das Konsolen-Programm **javadoc** mit aus, dem als Parameter ein Dateiname der zu kommentierenden Klasse übergeben wird; aus kompilierten Dateien können natürlich keine Beschreibungsdateien erstellt werden. Wir starten javadoc im Verzeichnis, in dem auch die Klassen liegen, und erhalten unsere HTML-Dokumente.

Beispiel: Möchten wir Dokumentationen für das gesamte Verzeichnis erstellen, so geben wir alle Dateien mit der Endung `.java` an:

```
$ javadoc *.java
```

JavaDoc geht durch den Quelltext, parst die Deklarationen und zieht die Dokumentation heraus. Daraus generiert das Tool eine Beschreibung, die in der Regel als HTML-Seite zu uns kommt.

In Eclipse lässt sich eine Dokumentation mit JavaDoc sehr einfach erstellen: Im Menü FILE - EXPORT ist der Eintrag JAVADOC zu wählen, und nach einigen Einstellungen ist die Dokumentation generiert.

Die Sichtbarkeit spielt bei JavaDoc eine wichtige Rolle. Standardmäßig nimmt JavaDoc nur öffentliche Dinge in die Dokumentation auf.



1.5 HTML-Tags in Dokumentationskommentaren

In den Kommentaren können HTML-Tags verwendet werden, beispielsweise `bold` und `<i>italic</i>`, um Textattribute zu setzen. Sie werden direkt in die Dokumentation übernommen und müssen korrekt geschachtelt sein, damit die Ausgabe nicht falsch dargestellt wird. Die Überschriften-Tags `<h1>..</h1>` und `<h2>..</h2>` sollten jedoch nicht verwendet werden. JavaDoc verwendet sie zur Gliederung der Ausgabe und weist ihnen Formatvorlagen zu.

In Eclipse zeigt die Ansicht javadoc in einer Vorschau das Ergebnis des Dokumentationskommentars an.

1.6 Generierte Dateien

Für jede öffentliche Klasse erstellt JavaDoc eine HTML-Datei. Sind Klassen nicht öffentlich, muss ein Schalter angegeben werden. Die HTML-Dateien werden zusätzlich mit Querverweisen zu den anderen dokumentierten Klassen versehen. Daneben erstellt JavaDoc weitere Dateien:

- ✓ **index-all.html** liefert eine Übersicht aller Klassen, Schnittstellen, Ausnahmen, Methoden und Felder in einem Index.
- ✓ **overview-tree.html** zeigt in einer Baumstruktur die Klassen an, damit die Vererbung deutlich sichtbar ist.
- ✓ **allclasses-frame.html** zeigt alle dokumentierten Klassen in allen Unterpaketen auf.
- ✓ **deprecated-list.html** bietet eine Liste der veralteten Methoden und Klassen.
- ✓ **serialized-form.html** listet alle Klassen auf, die Serializable implementieren. Jedes Attribut erscheint mit einer Beschreibung in einem Absatz.
- ✓ **help-doc.html** zeigt eine Kurzbeschreibung von JavaDoc.
- ✓ **index.html:** JavaDoc erzeugt eine Ansicht mit Frames. Das ist die Hauptdatei, die die rechte und linke Seite referenziert. Die linke Seite ist die Datei `allclasses-frame.html`. Rechts im Frame wird bei fehlender Paketbeschreibung die erste Klasse angezeigt.
- ✓ **stylesheet.css** ist eine Formatvorlage für HTML-Dateien, in der sich Farben und Zeichensätze einstellen lassen, die dann alle HTML-Dateien nutzen.
- ✓ **packages.htm** ist eine veraltete Datei. Sie verweist auf die neuen Dateien.

1.7 Dokumentationskommentare im Überblick

Einige JavaDoc-Kommentare kann der Entwickler in den Block setzen, so wie `@param` oder `@return` zur Beschreibung der Parameter oder Rückgaben, andere auch in den Text, wie `{@link}` zum Setzen eines Verweises auf einen anderen Typ oder eine andere Methode. Tags der ersten Gruppe heißen **Block-Tags**, die anderen **Inline-Tags**. Bisher erkennt das JavaDoc-Tool die folgenden Tags (ab welcher Version, steht in Klammern; <http://tutego.de/go/javadoctags>):

- ✓ **Block-Tags:** `@author (1.0)`, `@deprecated (1.0)`, `@exception (1.0)`, `@param (1.0)`, `@return (1.0)`, `@see (1.0)`, `@serial (1.2)`, `@serialData (1.2)`, `@serialField (1.2)`, `@since (1.1)`, `@throws (1.2)`, `@version (1.0)`
- ✓ **Inline-Tags:** `{@code} (1.5)`, `{@docRoot} (1.3)`, `{@inheritDoc} (1.4)`, `{@link} (1.2)`, `{@linkplain} (1.4)`, `{@literal} (1.5)`, `{@value} (1.4)`

In Java 6 und Java 7 ist kein Tag hinzugekommen. Es gab einiges auf der Liste, was als JavaDoc-Tag in Java 8 hinzukommen soll.

Beispiele

Eine externe Zusatzquelle geben wir wie folgt an:

```
@see <a href="spec.html#section">Java Spec</a>.
```

Verweis auf eine Methode, die mit der beschriebenen Methode verwandt ist:

```
@see String#equals(Object) equals
```

Von `@see` gibt es mehrere Varianten:

```
@see #field
@see #method(Type, Type,...)
@see #method(Type argname, Type argname,...)
@see #constructor(Type, Type,...)
@see #constructor(Type argname, Type argname,...)
@see Class#field
@see Class#method(Type, Type,...)
@see Class#method(Type argname, Type argname,...)
@see Class#constructor(Type, Type,...)
@see Class#constructor(Type argname, Type argname,...)
@see Class.NestedClass
@see Class
@see package.Class#field
@see package.Class#method(Type, Type,...)
@see package.Class#method(Type argname, Type argname,...)
@see package.Class#constructor(Type, Type,...)
@see package.Class#constructor(Type argname, Type argname,...)
@see package.Class.NestedClass
@see package.Class
@see package
```

Dokumentiere eine Variable. Gib einen Verweis auf eine Methode an:

```
/**
 * The X-coordinate of the component.
 *
 * @see #getLocation()
 */
int x = 1263732;
```

Eine veraltete Methode, die auf eine Alternative zeigt:

```
/**  
 * @deprecated As of JDK 1.1,  
 * replaced by {@link #setBounds(int,int,int,int)}  
 */
```

Anstatt HTML-Tags wie <tt> oder <code> für den Quellcode zu nutzen, ist {@code} viel einfacher.

```
/**  
 * Compares this current object with another object.  
 * Uses {@code equals()} an not {@code ==}}.  
 */
```

1.8 JavaDoc und Doclets

Die Ausgabe von JavaDoc kann den eigenen Bedürfnissen angepasst werden, indem Doclets eingesetzt werden. Ein Doclet ist ein Java-Programm, das auf der Doclet-API aufbaut und die Ausgabedatei schreibt. Das Programm liest dabei wie das bekannte JavaDoc-Tool die Quelldateien ein und erzeugt daraus ein beliebiges Ausgabeformat. Dieses Format kann selbst gewählt und implementiert werden. Wer also neben dem von JavaSoft beigefügten Standard-Doclet für HTML-Dateien Framemaker-Dateien (MIF) oder RTF-Dateien erzeugen möchte, der muss ein eigenes Doclet programmieren oder kann auf Doclets unterschiedlicher Hersteller zurückgreifen. Die Webseite <http://www.doclet.com/> listet zum Beispiel Doclets auf, die Docbook generieren oder UML-Diagramme mit aufnehmen.

Daneben dient ein Doclet aber nicht nur der Schnittstellendokumentation. Ein Doclet kann auch aufzeigen, ob es zu jeder Methode eine Dokumentation gibt oder ob jeder Parameter und jeder Rückgabewert korrekt beschrieben ist. Vor dem Durchbruch der Annotationen in Java 5 waren Doclets zur Generierung zusätzlicher Programmdateien und XML-Deskriptoren populär. XDoclet (<http://xdoclet.sourceforge.net/>) generiert aus Anmerkungen in den JavaDoc-Tags Mappings-Dateien für relationale Datenbanken oder Dokumente für Enterprise JavaBeans.

2 Techniken für den Umgang mit Zeichenketten

In diesem Kapitel erfahren Sie

- ✓ wie Sie Zeichenketten mit Mustern vergleichen und zerlegen können
- ✓ wie Sie formulierte Ausgaben erzielen

Voraussetzungen

- ✓ Arbeiten mit Strings

2.1 Reguläre Ausdrücke

Ein regulärer Ausdruck (engl. regular expression) ist eine Beschreibung eines Musters (engl. pattern). Reguläre Ausdrücke werden bei der Zeichenkettenverarbeitung beim Suchen und Ersetzen eingesetzt. Für folgende Szenarien bietet die Java-Bibliothek entsprechende Unterstützung an:

- ✓ **Frage nach einer kompletten Übereinstimmung:** Passt eine Zeichenfolge komplett auf ein Muster? Wir nennen das **match**. Die Rückgabe einer solchen Anfrage ist einfach wahr oder falsch.
- ✓ **Finde Teilstrings:** Das Pattern beschreibt einen Teilstring, und gesucht sind alle Vorkommen dieses Musters in einem Suchstring.
- ✓ **Ersetze Teilstichenfolgen:** Das Pattern beschreibt Wörter, die durch andere Wörter ersetzt werden.
- ✓ **Zerlegen einer Zeichenfolge:** Das Muster steht für Trennzeichen, sodass nach dem Zerlegen eine Sammlung von Zeichenfolgen entsteht.

Ein **Pattern-Matcher** ist die "Maschine", die reguläre Ausdrücke verarbeitet. Zugriff auf diese Mustermaschine bietet die Klasse `Matcher`. Dazu kommt die Klasse `Pattern`, die die regulären Ausdrücke in einem vorcompilierten Format repräsentiert. Beide Klassen befinden sich im Paket `java.util.regex`. Um die Sache etwas zu vereinfachen, gibt es bei `String` zwei kleine Hilfsmethoden, die im Hintergrund auf die Klassen verweisen, um eine einfachere API anbieten zu können; diese nennen sich auch **Fassaden-Methoden**.

`Pattern.matches()` bzw. `String#matches()`

Die statische Methode `java.util.regex.Pattern.matches()` und die Objektmethode `matches()` der Klasse `String` testen, ob ein regulärer Ausdruck eine Zeichenfolge komplett beschreibt.

Wir wollen testen, ob eine Zeichenfolge in einfache Hochkommata eingeschlossen ist:

Ausdruck	Ergebnis
<code>Pattern.matches(".*\"", "'Hallo Welt'")</code>	true
<code>"'Hallo Welt'".matches(".*\"")</code>	true
<code>Pattern.matches(".*\"", "!!!")</code>	true
<code>Pattern.matches(".*\"", "Hallo Welt")</code>	false
<code>Pattern.matches(".*\"", "'Hallo Welt'")</code>	false

Einfache reguläre Ausdrücke und ihr Ergebnis

Der Punkt im regulären Ausdruck steht für ein beliebiges Zeichen, und der folgende Stern ist ein Quantifizierer, der wahllos viele beliebige Zeichen erlaubt.

Zeichenklasse	Enthält
\p{Alpha}	einen Buchstaben: [\p{Lower}\p{Upper}]
\p{Alnum}	ein alphanumerisches Zeichen: [\p{Alpha}\p{Digit}]
\p{Punct}	ein Punkt-Zeichen: !"#\$%&'()*+,-./:;=>?@[\]^_`{ }~
\p{Graph}	ein sichtbares Zeichen: [\p{Alnum}\p{Punct}]
\p{Print}	ein druckbares Zeichen: [\p{Graph}]

Bei den Wortzeichen handelt es sich standardmäßig um die ASCII-Zeichen und nicht um deutsche Zeichen mit unseren Umlauten oder allgemeine Unicode-Zeichen. Eine umfassende Übersicht liefert die API-Dokumentation der Klasse `java.util.regex.Pattern`.

RegExDemo.java, main(), Ausschnitt

```
System.out.println( Pattern.matches( "\\\d*", "112" ) );           // true
System.out.println( Pattern.matches( "\\\d*", "112a" ) );          // false
System.out.println( Pattern.matches( "\\\d*.", "112a" ) );         // true
System.out.println( Pattern.matches( ".\\\d*.", "x112a" ) );       // true
```



Die Methode `contains()` der `String`-Klasse testet nur Teilzeichenfolgen, aber diese Zeichenfolge ist kein regulärer Ausdruck (sonst würde so etwas wie `contains(".")` auch eine völlig andere Bedeutung haben). Wer ein `s.contains("pattern")` sucht, kann es als `s.matches(".*pattern.*")` umschreiben.

Die Klassen Pattern und Matcher

Der Aufruf der Objektmethode `matches()` auf einem `String`-Objekt beziehungsweise das statische `Pattern.matches()` ist nur eine Abkürzung für die Übersetzung eines Patterns und Anwendung von `matches()`:

String#matches()	Pattern.matches()
<pre>public boolean matches(String regex) { return Pattern.matches(regex, this); }</pre>	<pre>public static boolean matches(String regex, CharSequence input) { Pattern p = Pattern.compile(regex); Matcher m = p.matcher(input); return m.matches(); }</pre>

Implementierungen der beiden matches()-Methoden

Während die `String`-Mitläufer-Methode `matches()` zur `Pattern.matches()` delegiert, steht hinter der statischen Fassadenmethode `Pattern.matches()` die wirkliche Nutzung der beiden zentralen Klassen `Pattern` für das Muster und `Matcher` für die Mustermaschine. Für unser erstes Beispiel `Pattern.matches(".*'", "'Hallo Welt'")` hätten wir also äquivalent schreiben können:

```
Pattern p = Pattern.compile( ".*'" );
Matcher m = p.matcher( "'Hallo Welt'" );
boolean b = m.matches();
```



Bei mehrmaliger Anwendung des gleichen Patterns sollte es kompiliert gesucht werden, denn das immer wieder nötige Übersetzen über die Objektmethode `String#matches()` beziehungsweise die Klassenmethode `Pattern.matches()` kostet Speicher und Laufzeit.

java.util.regex.Pattern
+ compile(regex: String): Pattern
+ compile(regex: String, flags: int): Pattern
+ pattern(): String
+ toString(): String
+ matcher(input: CharSequence): Matcher
+ flags(): int
+ matches(regex: String, input: CharSequence): boolean
+ split(input: CharSequence, limit: int): String[]
+ split(input: CharSequence): String[]
+ quote(s: String): String

```
final class java.util.regex.Pattern
implements Serializable
```

UML-Diagramm von Pattern

- ✓ static Pattern compile(String regex)
Übersetzt den regulären Ausdruck in ein Pattern-Objekt.

- ✓ static Pattern compile(String regex, int flags)
Übersetzt den regulären Ausdruck in ein Pattern-Objekt mit Flags. Als Flags sind CASE_INSENSITIVE, MULTILINE, DOTALL, UNICODE_CASE und CANON_EQ erlaubt. In Java 7 kommt UNICODE_CHARACTER_CLASS hinzu.
- ✓ int flags()
Liefert die Flags, nach denen geprüft wird.
- ✓ Matcher matcher(CharSequence input)
Liefert ein Matcher-Objekt, das prüft.
- ✓ static boolean matches(String regex, CharSequence input)
Liefert true, wenn der reguläre Ausdruck regex auf die Eingabe passt.
- ✓ static String quote(String s)
Maskiert die Metazeichen/Escape-Sequenzen aus. So liefert Pattern.quote ("*. [\d]") den String \Q*. [\d]\E.
- ✓ String pattern()
Liefert den regulären Ausdruck, den das Pattern-Objekt repräsentiert.

Pattern-Flags

Die Flags sind in speziellen Situationen ganz hilfreich, etwa wenn die Groß-/Kleinschreibung keine Rolle spielt oder sich die Suche über eine Zeile erstrecken soll. Doch Java zwingt uns nicht, die Pattern-Klasse zu nutzen, um die Flags einzusetzen zu können, sondern erlaubt es, mit einer speziellen Schreibweise die Flags auch im regulären Ausdruck selbst anzugeben, was die Nutzung bei String#matches() ermöglicht.

Flag in der Pattern-Klasse	Eingebetteter Flag-Ausdruck
Pattern.CASE_INSENSITIVE	(?i)
Pattern.COMMENTS	(?x)
Pattern.MULTILINE	(?m)
Pattern.DOTALL	(?s)
Pattern.UNICODE_CASE	(?u)
Pattern.UNICODE_CHARACTER_CLASS	(?U)
Pattern.UNIX_LINES	(?d)

In einem regulären Ausdruck sind die Varianten rechts sehr praktisch, da sie an unterschiedlichen Positionen ein- und ausgeschaltet werden können. Ein nach dem Fragezeichen platziertes Minus stellt die Option wieder ab, etwa "(?i) jetzt insensitive(?-i) wieder sensitive". Mehrere Flag-Ausdrücke lassen sich auch zusammensetzen, etwa zu "(?ims)".

In der Praxis häufiger im Einsatz sind Pattern.DOTALL/(?s), Pattern.CASE_INSENSITIVE/(?i) und Pattern.MULTILINE/(?m). Es folgen Beispiele, wobei wir MULTILINE bei den Wortgrenzen vorstellen. Standardmäßig matcht der "." kein Zeilenendezeichen, sodass ein regulärer Ausdruck einen Zeilenumbruch nicht erkennt. Das lässt sich mit Pattern.DOTALL-Flag beziehungsweise (?s) ändern.

Beispiel: Die Auswirkung vom DOTALL beziehungsweise (?s):

```
System.out.println( "wau wau miau".matches( "wau.+wau.*" ) );      // true
System.out.println( "wau\nwau miau".matches( "wau.+wau.*" ) );      // false
System.out.println( "wau wau miau".matches( "(?s)wau.+wau.*" ) );  // true
System.out.println( "wau\nwau miau".matches( "(?s)wau.+wau.*" ) ); // true
```

Quantifizierer und Wiederholungen

Neben den Quantifizierern ? (einmal oder keinmal), * (keinmal oder beliebig oft) und + (einmal oder beliebig oft) gibt es drei weitere Quantifizierer, die es erlauben, die Anzahl eines Vorkommens genauer zu beschreiben:

- ✓ X{n}. X muss genau n-mal vorkommen.
- ✓ X{n, }. X kommt mindestens n-mal vor.
- ✓ X{n, m}. X kommt mindestens n-, aber maximal m-mal vor.

Beispiel: Eine E-Mail-Adresse endet mit einem Domain-Namen, der 2 oder 3 Zeichen lang ist:

```
Static Pattern p = Pattern.compile( "[\\w|-]+@[\\w[\\w|-]*\\. [a-z]{2,3}" );
```

Ränder und Grenzen testen

Die bisherigen Ausdrücke waren nicht ortsgebunden, sondern haben geprüft, ob es irgendwo im String eine Übereinstimmung gibt. Dateiendungen zum Beispiel sind aber - wie der Name schon sagt - am Ende zu prüfen, genauso wie ein URL-Protokoll wie "http://" am Anfang stehen muss. Um diese Anforderungen mit berücksichtigen zu können, können bestimmte Positionen mit in einem regulären Ausdruck gefordert werden. Die Pattern-API erlaubt folgende Matcher:

Matcher	Bedeutung
^	Beginn einer Zeile
\$	Ende einer Zeile
\b	Wortgrenze
\B	Keine Wortgrenze
\A	Beginn der Eingabe
\Z	Ende der Eingabe ohne Zeilenabschlusszeichen wie \n oder \r
\z	Ende der Eingabe mit allen Zeilenabschlusszeichen
\G	Ende des vorherigen Matches. Sehr speziell für iterative Suchvorgänge

Wichtig ist zu verstehen, dass diese Matcher keine "Breite" haben, also nicht wirklich ein Zeichen oder eine Zeichenfolge matchen, sondern lediglich die Position beschreiben.

Die Matcher ^ und \$ lösen gut das Problem mit den Dateiendungen und HTTP-Protokollen und leisten gute Dienste bei bestimmten Löschanweisungen.

Beispiel: Die String-Methode trim() schneidet den Weißraum vorne und hinten ab. Mit replaceAll() und den Matchern für den Beginn und das Ende einer Zeile ist schnell ein Ausdruck gefunden, der nur den Weißraum vorne oder nur hinten entfernt:

```
String s = "\t\two ist die Programmiersprache des Lächelns?\t\t ";
String ltrim = s.replaceAll( "^\\s+", "" );
String rtrim = s.replaceAll( "\\s+$", "" );
System.out.printf( "'%s'%n", ltrim ); // 'Wo ist die Programmiersprache des
                                         // Lächelns? '
System.out.printf( "'%s'%n", rtrim ); // ' Wo ist die Programmiersprache des
                                         // Lächelns? '
```

Der Matcher \b ist nützlich, wenn es darum geht, ein Wort umrandet von Weißraum in einer Teilzeichenkette zu finden. In der Zeichenkette "Spaß in China innerhalb der Grenzen" wird die Suche nach "in" drei Fundstellen ergeben, aber \bin\b nur eine und \bin\B auch eine, und zwar "innerhalb". Es matcht demnach ein \b genau die Stelle, bei der ein \w auf ein \W folgt (beziehungsweise andersherum).

Multiline-Modus

Normalerweise sind ^ und \$ nicht zeilenorientiert, das heißt, es ist ihnen egal, ob im String Zeilenumbruchzeichen wie \n oder \r vorkommen oder nicht. Mitunter soll der Test aber lokal auf einer Zeile stattfinden - hierzu muss der Multiline-Modus aktiviert werden.

Beispiel: Teste, ob eine E-Mail die Zeile "Hi," enthält:

```
System.out.println( "Hi,".matches( ".*^Hi,$.*" ) );
System.out.println( "Fwd:\nHi,mir geht's gut!".matches( ".*^Hi,$.*" ) );
System.out.println( "Fwd:\nHi,\nmir geht's gut!".matches( "(?sm).*^Hi,$.*" ) );
```

Der Test auf ".*^Hi,\$.*" gibt im ersten Fall true zurück, da der String wirklich matcht und wir auch überhaupt keinen Zeilentrenner haben, der uns Probleme bereiten könnte. Die zweite Zeile aber liefert false, da sie global mit "Fwd" und nicht mit "Hi" beginnt und mit "!" endet statt mit einem Komma. Führen wir den Test mit der Option (?sm) zeilenweise durch und überspringen wir die Zeilentrenner, dann ist das Ergebnis true, denn die 2. Zeile in

```
Fwd:  
Hi,  
mir geht's gut!
```

passt genau auf unseren regulären Ausdruck.

Der Multiline-Modus erklärt auch den Grund, warum es gleich mehrere Grenz-Matcher gibt. Die Matches \A und \Z beziehungsweise \z sind im Prinzip wie ^ und \$, unterscheiden sich aber dann, wenn der Multiline-Modus aktiviert ist. Dann arbeiten (wie im Beispiel) ^ und \$ zeilenorientiert, \A und \Z beziehungsweise \z aber nie - die letzten drei Matcher kennen Zeilentrenner überhaupt nicht. Damit ist "Fwd:\nHi,\nalles OK!".matches("(?sm).*\\AHi,\\Z.*") auch trotz (?sm) ganz einfach false.

Es bleiben \z und \Z. Sie unterscheiden, ob bei Zeilen, die abschließende Zeilentrenner wie \n oder \r besitzen, diese Zeilentrenner mit zum Match gehören oder nicht. Das \z ist wie \$ ein Matcher auf das absolute Ende inklusive aller Zeilentrenner. Das große \Z ignoriert am Ende stehende Zeilentrenner, sodass sozusagen der Match schon vorher zu Ende ist.

Beispiel: Das Trennzeichen beim split() soll einmal \z und einmal \Z sein:

```
String[] tokens1 = "Lena singt\r\nn".split( "\\\z" );
String[] tolens2 = "Lena singt\r\nn".split( "\\\Z" );
System.out.printf( "%d %s%n", tokens1.length, Arrays.toString( tokens1 ) );
System.out.printf( "%d %s%n", tolens2.length, Arrays.toString( tolens2 ) );
```

Bei \z gehören alle Zeilentrenner zum String, und daher ist die Ausgabe:

```
1 [Lena singt  
]
```

Die zweite Ausgabe ist:

```
2 [Lena singt,  
]
```

Und die abschließenden Zeilentrenner sind ein zweites Token.

Finden und nicht matchen

Bisher haben wir mit regulären Ausdrücken lediglich festgestellt, ob eine Zeichenfolge vollständig auf ein Muster passt. Die Matcher-Klasse kann jedoch auch feststellen, ob sich eine durch ein Muster beschriebene Teilfolge im String befindet. Dazu dient die Methode find(). Sie hat zwei Aufgaben: Zunächst sucht sie nach einer Fundstelle und gibt bei Erfolg true zurück. Das Nächste ist, dass jedes Matcher-Objekt einen Zustand mit Fundstellen besitzt, den find() aktualisiert. Einem Matcher-Objekt entlockt die Methode group() den erkannten Substring, und start()/end() liefert die Positionen. Wiederholte Aufrufe von find() setzen die Positionen weiter:

RegExAllNumbers.java, main()

```
String s = "'Demnach, welcher verheiratet, der tut wohl; welcher aber " +
          "nicht verheiratet, der tut besser.' 1. Korinther 7, 38";
Matcher matcher = Pattern.compile( "\\d+" ).matcher( s );
while ( matcher.find() )
    System.out.printf( "%s an Position [%d,%d]\n",
                       matcher.group(),
                       matcher.start(), matcher.end() );
```

Die Ausgabe des Zahlenfinders ist:

```
1 an Position [94,95]
7 an Position [107,108]
38 an Position [110,112]
```

Beispiel: Da es in der String-Klasse zwar ein `contains()`, aber kein `containsIgnoreCase()` gibt, lässt sich für diesen Zweck entweder ein Ausdruck wie `s1.toLowerCase().contains(s2.toLowerCase())` formen oder ein Pattern-Flag verwenden:

```
String s1 = "Prince Michael I, Paris, Prince Michael II (Blanket)";
String s2 = "PARIS";
boolean in = Pattern.compile( Pattern.quote( s2 ),
                             Pattern.CASE_INSENSITIVE ).matcher( s1 ).find();
System.out.println( in ); // true
```

2.2 Zerlegen von Zeichenketten

Die Java-Bibliothek bietet einige Klassen und Methoden, um nach bestimmten Mustern große Zeichenketten in kleinere zu zerlegen. In diesem Kontext sind die Begriffe **Token** und **Delimiter** zu nennen: Ein Token ist ein Teil eines Strings, der durch bestimmte Trennzeichen (engl. delimiter) von anderen Tokens getrennt wird. Nehmen wir als Beispiel den Satz "Moderne Musik ist Instrumentespielen nach Noten" (Peter Sellers). Wählen wir Leerzeichen als Trennzeichen, lauten die einzelnen Tokens "Moderne", "Musik" und so weiter.

Die Java-Bibliothek bietet eine Reihe von Möglichkeiten zum Zerlegen von Zeichenfolgen, von denen einige in den nachfolgenden Abschnitten vorgestellt werden:

- ✓ `split()` von `String`: Aufteilen mit einem Delimiter, der durch reguläre Ausdrücke beschrieben wird.
- ✓ `Scanner`: Schöne Klasse zum Ablauen einer Eingabe.
- ✓ `StringTokenizer`: Der Klassiker aus Java 1.0. Delimiter sind nur einzelne Zeichen.
- ✓ `BreakIterator`: Findet Zeichen-, Wort-, Zeilen- oder Satz-Grenzen.

Splitten von Zeichenketten mit `split()`

Die Objektmethode `split()` eines `String`-Objekts zerlegt die eigene Zeichenkette in Teilzeichenketten. Die Trenner sind völlig frei wählbar und als regulärer Ausdruck beschrieben. Die Rückgabe ist ein Feld der Teilzeichenketten.

Beispiel: Zerlege einen Domain-Namen in seine Bestandteile:

```
String path = "www.tutego.com";
String[] segs = path.split( Pattern.quote( "." ) );
System.out.println( Arrays.toString(segs) ); // [www, tutego, com]
```

Da der Punkt als Trennzeichen ein Sonderzeichen für reguläre Ausdrücke ist, muss er passend mit dem Backslash auskommentiert werden. Das erledigt die statische Methode `quote()`. Andernfalls liefert `split(".")` auf jedem String ein Feld der Länge 0.

Ein häufiger Trenner ist `\s`, also Weißraum.

Beispiel: Zähle die Anzahl der Wörter in einem Satz:

```
String string = "Hört es euch an, denn das ist mein Gedudel!";
int nrOfWords = string.split( "(\\s|\\p{Punct})+" ).length;
System.out.println( nrOfWords ); // 9
```

Der Trenner ist entweder Weißraum oder ein Satzeichen.

String.split() geht auf Pattern#split()

Die `split()`-Methode aus der `String`-Klasse delegiert wie auch bei `match()` an das `Pattern`-Objekt:

```
public String[] split( String regex, int limit )
{
    return Pattern.compile( regex ).split( this, limit );
}
public String[] split( String regex )
{
    return split( regex, 0 );
}
```

Am Quellcode ist zu erkennen, dass für jeden Methodenaufruf von `split()` auf dem `String`-Objekt ein `Pattern` übersetzt wird. Das ist nicht ganz billig, und so soll bei mehrmaligem Split mit dem gleichen Zerlege-Muster gleich ein `Pattern`-Objekt und dort das `split()` verwendet werden:

```
final class java.lang.String
implements CharSequence, Comparable<String>, Serializable
```

- ✓ `String[] split(String regex)`
Zerlegt die aktuelle Zeichenkette mit dem regulären Ausdruck.
- ✓ `String[] split(String regex, int limit)`
Zerlegt die aktuelle Zeichenkette mit dem regulären Ausdruck, liefert jedoch maximal begrenzt viele Teilzeichenfolgen.

```
final class java.util.regex.Pattern
implements Serializable
```

- ✓ `String[] split(CharSequence input)`
Zerlegt die Zeichenfolge `input` in Teilzeichenketten, wie es das aktuelle `Pattern`-Objekt befiehlt.
- ✓ `String[] split(CharSequence input, int limit)`
Wie `split(CharSequence)`, doch nur höchstens `limit` viele Teilzeichenketten.

Die Klasse Scanner

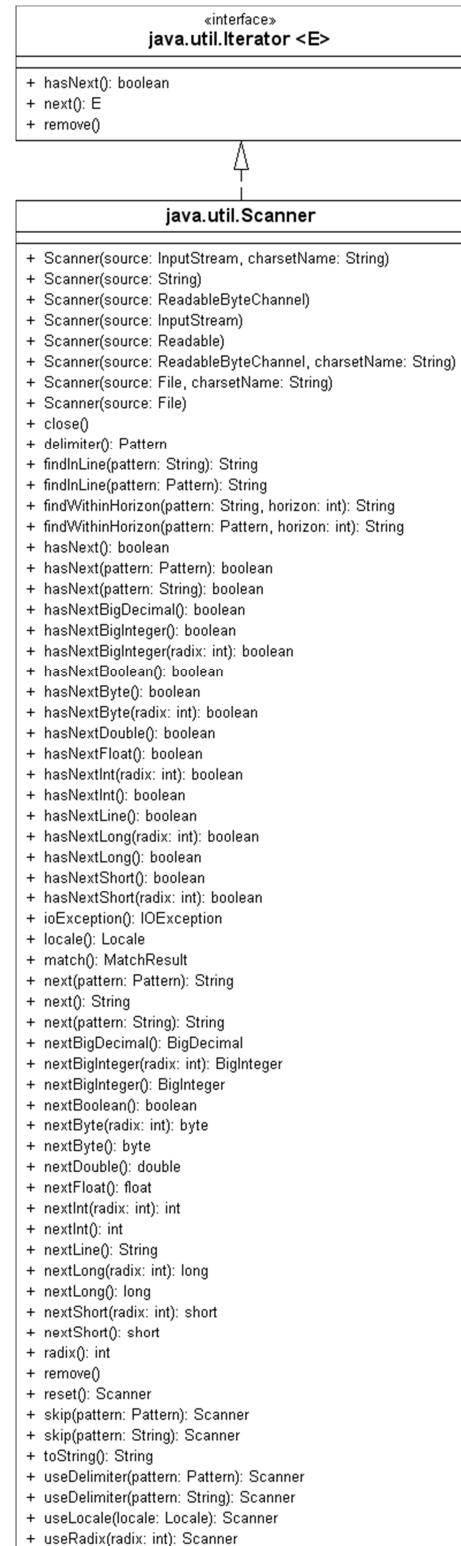
Die Klasse `java.util.Scanner` kann eine Zeichenkette in Tokens zerlegen und einfach Dateien zeilenweise einlesen. Bei der Zerlegung kann ein regulärer Ausdruck den Delimiter beschreiben. Damit ist `Scanner` flexibler als ein `StringTokenizer`, der nur einzelne Zeichen als Trenner zulässt.

Zum Aufbau der `Scanner`-Objekte bietet die Klasse einige Konstruktoren an, die die zu zerlegenden Zeichenfolgen unterschiedlichen Quellen entnehmen, etwa einem `String`, einem Datenstrom (beim Einlesen von der Kommandozeile wird das `System.in` sein), einem `File`-Objekt oder diversen NIO-Objekten. Falls ein Objekt vom Typ `Closeable` dahintersteckt, wie ein `Writer`, sollte mit `close()` der `Scanner` geschlossen werden, der das `close()` zum `Closeable` weiterleitet. Beim `String` ist das nicht nötig, und bei `File` schließt der `Scanner` selbstständig.

```
final class java.util.Scanner
implements Iterator<String>, Closeable
```

- ✓ Scanner(String source)
- ✓ Scanner(File source)
- ✓ Scanner(File source, String charsetName)
- ✓ Scanner(Path source)
- ✓ Scanner(Path source, String charsetName)
- ✓ Scanner(InputStream source)
- ✓ Scanner(InputStream source, String charsetName)
- ✓ Scanner(Readable source)
- ✓ Scanner(ReadableByteChannel source)
- ✓ Scanner(ReadableByteChannel source, String charsetName)

Erzeugt ein neues Scanner-Objekt aus diversen Quellen.



Zeilenweises Einlesen einer Datei

Ist das Scanner-Objekt angelegt, lässt sich mit dem Paar `hasNextLine()` und `nextLine()` einfach eine Datei zeilenweise auslesen:

ReadAllLines.java

```
import java.io.*;
import java.util.Scanner;

public class ReadAllLines
{
    public static void main( String[] args ) throws FileNotFoundException
    {
        Scanner scanner = new Scanner( new File("EastOfJava.txt") );
        while ( scanner.hasNextLine() )
            System.out.println( scanner.nextLine() );
        scanner.close();
    }
}
```

UML-Diagramm der Scanner-Klasse

Da der Konstruktor von `Scanner` mit der Datei eine Ausnahme auslösen kann, müssen wir diesen möglichen Fehler behandeln. Wir machen es uns einfach und leiten einen möglichen Fehler an die Laufzeitumgebung weiter. Auch sollte immer die Kodierung angegeben werden, doch auch das sparen wir uns für das kleine Beispiel.

```
final class java.util.Scanner
implements Iterator<String>, Closeable
```

- ✓ `boolean hasNextLine()`
Liefert `true`, wenn eine nächste Zeile gelesen werden kann.
- ✓ `String nextLine()`
Liefert die nächste Zeile.

Der Nächste, bitte

Nach dem Erzeugen des `Scanner`-Objekts liefert die Methode `next()` die nächste Zeichenfolge, wenn denn ein `hasNext()` die Rückgabe `true` ergibt. (Das sind dann auch die Methoden der Schnittstelle `Iterator`, wobei `remove()` nicht implementiert ist.)

Beispiel: Von der Standardeingabe soll ein `String` gelesen werden:

```
Scanner scanner = new Scanner( System.in );
String s = scanner.next();
```

Neben der `next()`-Methode, die nur einen `String` als Rückgabe liefert, bietet `Scanner` diverse `next<Typ>()`-Methoden an, die das nächste Token einlesen und in ein gewünschtes Format konvertieren, etwa in ein `double` bei `nextDouble()`. Über gleich viele `hasNext<Typ>()`-Methoden lässt sich erfragen, ob ein weiteres Token von diesem Typ folgt.

Beispiel: Die einzelnen `nextXXX()`- und `hasNextXXX()`-Methoden in einem Beispiel:

ScannerDemo.java, main()

```
Scanner scanner = new Scanner( "tutego 12 1973 12,03 True 123456789000" );
System.out.println( scanner.hasNext() );           // true
System.out.println( scanner.next() );             // tutego
System.out.println( scanner.hasNextByte() );       // true
System.out.println( scanner.nextByte() );          // 12
System.out.println( scanner.hasNextInt() );        // true
System.out.println( scanner.nextInt() );           // 1973
System.out.println( scanner.hasNextDouble() );      // true
System.out.println( scanner.nextDouble() );          // 12.03
System.out.println( scanner.hasNextBoolean() );     // true
System.out.println( scanner.nextBoolean() );         // true
System.out.println( scanner.hasNextLong() );         // true
System.out.println( scanner.nextLong() );            // 123456789000
System.out.println( scanner.hasNext() );             // false
```

Sind nicht alle Tokens interessant, überspringt `Scanner skip(Pattern pattern)` beziehungsweise `Scanner skip(String pattern)` sie - Delimiter werden nicht beachtet.

```
final class java.util.Scanner
implements Iterator<String>, Closeable
```

- ✓ `boolean hasNext()`
- ✓ `boolean hasNextBigDecimal()`
- ✓ `boolean hasNextBigInteger()`
- ✓ `boolean hasNextBigInteger(int radix)`
- ✓ `boolean hasNextBoolean()`
- ✓ `boolean hasNextByte()`

- ✓ boolean hasNextByte(int radix)
- ✓ boolean hasNextDouble()
- ✓ boolean hasNextFloat()
- ✓ boolean hasNextInt()
- ✓ boolean hasNextInt(int radix)
- ✓ boolean hasNextLong()
- ✓ boolean hasNextLong(int radix)
- ✓ boolean hasNextShort()
- ✓ boolean hasNextShort(int radix)

Liefert true, wenn ein Token des gewünschten Typs gelesen werden kann.

- ✓ String next()
- ✓ BigDecimal nextBigDecimal()
- ✓ BigInteger nextBigInteger()
- ✓ BigInteger nextBigInteger(int radix)
- ✓ boolean nextBoolean()
- ✓ byte nextByte()
- ✓ byte nextByte(int radix)
- ✓ double nextDouble()
- ✓ float nextFloat()
- ✓ int nextInt()
- ✓ int nextInt(int radix)
- ✓ long nextLong()
- ✓ long nextLong(int radix)
- ✓ short nextShort()
- ✓ short nextShort(int radix)

Liefert das nächste Token.

Die Methode `useRadix(int)` ändert die Basis für Zahlen und `radix()` erfragt sie.

Trennzeichen definieren

`useDelimiter()` setzt für die folgenden Filter-Vorgänge den Delimiter. Um nur lokal für das nächste Zerlegen einen Trenner zu setzen, lässt sich mit `next(String)` oder `next(Pattern)` ein Trennmuster angeben. `hasNext(String)` beziehungsweise `hasNext(Pattern)` liefern true, wenn das nächste Token dem Muster entspricht.

Beispiel: Der String `s` enthält eine Zeile wie `a := b`. Uns interessieren der linke und der rechte Teil:

```
String s = "Url := http://www.tutego.com";
Scanner scanner = new Scanner( s ).useDelimiter( "\\s*:=\\s*" );
System.out.printf( "%s = %s", scanner.next(), scanner.next() );
// Url = http://www.tutego.com
```

Mit `findInLine(String)` beziehungsweise `findInLine(Pattern)` wird der Scanner angewiesen, nach dem Muster nur bis zum nächsten Zeilenendezeichen zu suchen; Delimiter ignoriert er.

Beispiel: Suche mit `findInLine()` nach einem Muster:

```
String text = "Hänsel-und-Gretel\ngingen-durch-den-Wald";
Scanner scanner = new Scanner( text ).useDelimiter( "-" );
System.out.println( scanner.findInLine( "Wald" ) ); // null
System.out.println( scanner.findInLine( "ete" ) ); // "ete"
System.out.println( scanner.next() ); // "1" "gingen"
System.out.println( scanner.next() ); // "durch"
```

Mit `findWithinHorizon(Pattern, int)` beziehungsweise `findWithinHorizon(String, int)` lässt sich eine Obergrenze von Code-Points (vereinfacht ausgedrückt, von Zeichen) angeben. Liefert die Methode in dieser Grenze kein Token, liefert sie `null` und setzt auch den Positionszeiger nicht weiter.

Landessprachen

Auch ist die `Scanner`-Klasse in der Lage, die Dezimalzahlen unterschiedlicher Sprachen zu erkennen.

Beispiel: Mit dem passenden `Locale`-Objekt erkennt der Scanner bei `nextDouble()` auch Fließkomma-zahlen mit Komma, etwa "12,34":

```
Scanner scanner = new Scanner( "12,34" ).useLocale( Locale.GERMAN );
System.out.println( scanner.nextDouble() ); // 12.34
```

Das klingt logisch, funktioniert aber bei einem deutschsprachigem Betriebssystem in der Regel auch ohne `useLocale(Locale.GERMAN)`. Der Grund ist einfach: Der Scanner setzt das `Locale` vorher standardmäßig auf `Locale.getDefault()`, und bei auf Deutsch eingestellten Betriebssystemen ist das eben `Locale.GERMAN`. Andersherum bedeutet das, dass eine in englischer Schreibweise angegebene Zahl wie 12.34 nicht erkannt wird und der Scanner eine `java.util.InputMismatchException` meldet.

```
final class java.util.Scanner
implements Iterator<String>, Closeable
```

- ✓ `Scanner useLocale(Locale locale)`
Setzt die Sprache zum Erkennen der **lokalisierten Zahlen**, insbesondere der Fließkommazahlen.
- ✓ `Locale locale()`
Liefert die eingestellte Sprache.

IO-Fehler während des Parsens

Bezieht der Scanner die Daten von einem `Readable`, kann es Ein-/Ausgabefehler in Form von `IOExceptions` geben. Methoden wie `next()` geben diese Fehler nicht weiter, sondern fangen sie ab und speichern sie intern. Die Methode `ioException()` liefert dann das letzte `IOException`-Objekt oder `null`, falls es keinen Fehler gab.

Die Klasse StringTokenizer

Die Klasse `StringTokenizer` zerlegt ebenfalls eine Zeichenkette in Tokens. Der `StringTokenizer` ist jedoch auf **einzelne** Zeichen als Trennsymbole beschränkt, während die Methode `split()` und die Klassen um `Pattern` einen regulären Ausdruck zur Beschreibung der Trennsymbole erlauben. Es sind keine Zeichenfolgen wie ":"= denkbar.

Beispiel: Um einen String mithilfe eines `StringTokenizer`-Objekts zu zerlegen, wird dem Konstruktor der Klasse der zu unterteilende Text als Argument übergeben:

```
String s = "Faulheit ist der Hang zur Ruhe ohne vorhergehende Arbeit";
StringTokenizer tokenizer = new StringTokenizer( s );
while ( tokenizer.hasMoreTokens() )
    System.out.println( tokenizer.nextToken() );
```

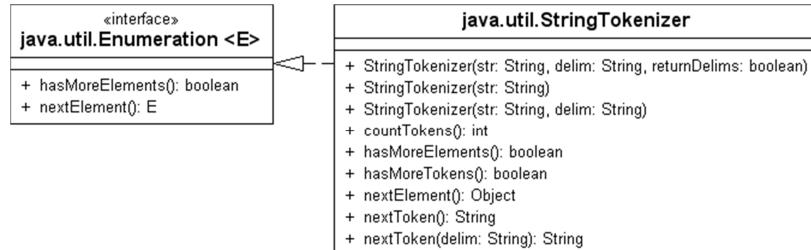
Der Text ist ausschließlich ein Objekt vom Typ `String`.

Um den Text abzulaufen, gibt es die Methoden `nextToken()` und `hasMoreTokens()`. (Die Methode `hasMoreElements()` ruft direkt `hasMoreTokens()` auf und wurde nur implementiert, da ein `StringTokenizer` die Schnittstelle `Enumeration` implementiert.) Die Methode `nextToken()` liefert das nächste Token im String. Ist kein Token mehr vorhanden, wird eine `NoSuchElementException` ausgelöst. Damit wir frei von diesen Überraschungen sind, können wir mit der Methode `hasMoreTokens()` nachfragen, ob noch ein weiteres Token vorliegt.

In der Voreinstellung sind Tabulator, Leerzeichen und Zeilentrenner die Delimiter. Sollen andere Zeichen als die voreingestellten Trenner den Satz zerlegen, kann dem Konstruktor als zweiter String eine Liste von Trennern übergeben werden. Jedes Zeichen, das in diesem String vorkommt, fungiert als einzelnes Trennzeichen:

```
 StringTokenizer st = new StringTokenizer( "Blue=0000ff\nRed:ff0000\n", "=:\n" );
```

Neben den beiden Konstruktoren existiert noch ein dritter, der auch die Trennzeichen als eigenständige Bestandteile bei `nextToken()` übermittelt.



UML-Diagramm vom StringTokenizer

```
class java.util.StringTokenizer
  implements Enumeration<Object>
```

- ✓ `StringTokenizer(String str, String delim, boolean returnDelims)`
Ein String-Tokenizer für `str`, wobei jedes Zeichen in `delim` als Trennzeichen gilt. Ist `returnDelims` gleich true, so sind auch die Trennzeichen Tokens der Aufzählung.
- ✓ `StringTokenizer(String str, String delim)`
Ein String-Tokenizer für `str`, wobei alle Zeichen in `delim` als Trennzeichen gelten. Entspricht dem Aufruf von `this(str, delim, false)`;
- ✓ `StringTokenizer(String str)`
Ein String-Tokenizer für `str`. Entspricht dem Aufruf von `this(str, "\t\n\r\f", false)`; Die Trennzeichen sind Leerzeichen, Tabulator, Zeilenende und Seitenvorschub.
- ✓ `boolean hasMoreTokens()`
- ✓ `boolean hasMoreElements()`
Testet, ob ein weiteres Token verfügbar ist. `hasMoreElements()` implementiert die Methode der Schnittstelle `Enumeration`, aber beide Methoden sind identisch.
- ✓ `String nextToken()`
- ✓ `Object nextElement()`
Liefert das nächste Token vom String-Tokenizer. `nextElement()` existiert nur, damit der Tokenizer als `Enumeration` benutzt werden kann. Der weniger spezifische Ergebnistyp `Object` macht eine Typumwandlung erforderlich.
- ✓ `String nextToken(String delim)`
Setzt die Delimiter-Zeichen neu und liefert anschließend das nächste Token.
- ✓ `int countTokens()`
Zählt die Anzahl der noch möglichen `nextToken()`-Methodenaufrufe, ohne die aktuelle Position zu berühren. Der Aufruf der Methode ist nicht billig.

BreakIterator als Zeichen-, Wort-, Zeilen- und Satztrenner

Benutzer laufen Zeichenketten aus ganz unterschiedlichen Gründen ab. Ein Anwendungsszenario ist das Ablaufen eines Strings Zeichen für Zeichen. In anderen Fällen sind nur einzelne Wörter interessant, die durch Wort- oder Satztrenner separiert sind. In wieder einem anderen Fall ist eine Textausgabe auf eine bestimmte Zeilenlänge gewünscht.

Zum Zerlegen von Zeichenfolgen sieht die Standardbibliothek im Java-Paket `java.text` die Klasse `BreakIterator` vor. Einen konkreten Iterator erzeugen diverse statische Methoden, die optional auch nach speziellen Kriterien einer Sprache trennen. Wenn keine Sprache übergeben wird, wird automatisch die Standardsprache verwendet.

```
abstract class java.text.BreakIterator
implements Cloneable
```

- ✓ static BreakIterator getCharacterInstance()
- ✓ static BreakIterator getCharacterInstance(Locale where)
 Trennt nach Zeichen. Vergleichbar mit einer Iteration über charAt().
- ✓ static BreakIterator getSentenceInstance()
- ✓ static BreakIterator getSentenceInstance(Locale where)
 Trennt nach Sätzen. Delimiter sind übliche Satztrener wie ".", "!", "?".
- ✓ static BreakIterator getWordInstance()
- ✓ static BreakIterator getWordInstance(Locale where)
 Trennt nach Wörtern. Trenner wie Leerzeichen und Satzzeichen gelten ebenfalls als Wörter.
- ✓ static BreakIterator getLineInstance()
- ✓ static BreakIterator getLineInstance(Locale where)
 Trennt nicht nach Zeilen, wie der Name vermuten lässt, sondern ebenfalls nach Wörtern. Nur werden Satzzeichen, die am Wort "hängen", zum Wort hinzugezählt. Praktisch ist dies für Algorithmen, die Textblöcke in eine bestimmte Breite bringen wollen. Ein Beispiel für die drei Typen zeigt das gleich folgende Programm.

Auf den ersten Blick ergibt ein BreakIterator von getCharacterInstance() keinen großen Sinn, denn für das Ablaufen einer Zeichenkette ließe sich viel einfacher eine Schleife nehmen und mit charAt() arbeiten. Der BreakIterator kann jedoch korrekt mit Unicode 4 umgehen, wo zwei char ein Unicode 4-Zeichen bilden. Zum zeichenweisen Iterieren über Strings ist auch CharacterIterator eine gute Lösung.



Beispiel für die drei BreakIterator-Typen

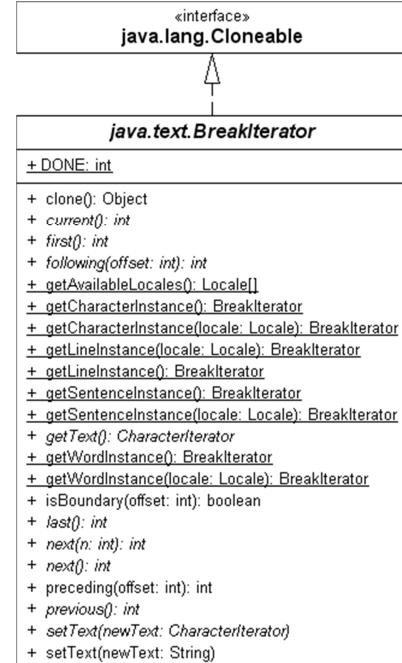
Das nächste Beispiel zeigt, wie ohne großen Aufwand durch Zeichenketten gewandert werden kann. Die Verwendung eines String-Tokenizers ist nicht nötig. Unsere statische Hilfsmethode out() gibt die Abschnitte der Zeichenkette bezüglich eines BreakIterator aus:

BreakIteratorDemo.java, out()

```
static void out( String s, BreakIterator iter )
{
    iter.setText( s );

    for ( int last = iter.first(), next = iter.next();
          next != BreakIterator.DONE;
          last = next, next = iter.next() )
    {
        CharSequence part = s.subSequence( last, next );

        if ( Character.isLetterOrDigit( part.charAt( 0 ) ) )
            System.out.println( part );
    }
}
```



Klassendiagramm vom BreakIterator

Einmal sollen die Wörter und einmal die Sätze ausgegeben werden:

```
BreakIteratorDemo.java, main()

public static void main( String[] args )
{
    String helmutKohl1 = "Ich weiß, dass ich 1945 fünfzehn war und 1953 achtzehn.",
           helmutKohl2 = "Das ist eine klassische journalistische Behauptung. " +
                           "Sie ist zwar richtig, aber sie ist nicht die Wahrheit.";

    BreakIterator sentenceIter = BreakIterator.getSentenceInstance();
    BreakIterator wordIter      = BreakIterator.getWordInstance();
    BreakIterator lineIter      = BreakIterator.getLineInstance();

    out( helmutKohl1, sentenceIter );
    out( helmutKohl2, sentenceIter );

    System.out.println( "-----" );

    out( helmutKohl1, wordIter );
    out( helmutKohl2, wordIter );

    System.out.println( "-----" );

    out( helmutKohl1, lineIter );
    out( helmutKohl2, lineIter );
}
```

Die Ausgabe enthält (skizziert):

```
Ich weiß, dass ich 1945 fünfzehn war und 1953 achtzehn.
Das ist eine klassische journalistische Behauptung.
Sie ist zwar richtig, aber sie ist nicht die Wahrheit.
-----

```

```
Ich
weiß
...
die
Wahrheit
-----
Ich
weiß,
...
die
Wahrheit.
```

Im letzten Beispiel ist gut zu sehen, dass die Wörter am Ende ihre Leer- und Satzzeichen behalten.

2.3 Ausgaben formatieren

Immer wieder müssen Zahlen, Datumsangaben und Text auf verschiedenste Art und Weise formatiert werden. Zur Formatierung bietet Java mehrere Lösungen:

- ✓ Seit Java 5 realisieren die `format()`- und `printf()`-Methoden eine Ausgabe, so wie sie unter C mit `printf()` gesetzt wurde.
- ✓ Formatieren über Format-Klassen: Allgemeines Formatierungsverhalten wird in einer abstrakten Klasse `Format` fixiert; konkrete Unterklassen, wie `NumberFormat` und `DateFormat`, nehmen sich spezielle Datenformate vor.
- ✓ Umsetzung eines Strings nach einer gegebenen Maske mit einem `MaskFormatter`.
- ✓ Die Format-Klassen bieten nicht nur landes- beziehungsweise sprachabhängige Ausgaben per `format()`, sondern auch den umgekehrten Weg, Zeichenketten wieder in Typen wie `double` oder `Date` zu zerlegen. Jede Zeichenkette, die vom Format-Objekt erzeugt wurde, kann auch mit dem Parser wieder eingelesen werden.

Formatieren und Ausgeben mit `format()`

Die Klasse `String` stellt mit der statischen Methode `format()` eine Möglichkeit bereit, Zeichenketten nach einer Vorgabe zu formatieren.

Beispiel

```
String s = String.format( "Hallo %s. Es gab einen Anruf von %s.",  
                         "Chris", "Joy" );  
System.out.println( s ); // Hallo Chris. Es gab einen Anruf von Joy.
```

Der erste an `format()` übergebene String nennt sich **Format-String**. Er enthält neben auszugebenden Zeichen weitere sogenannte **Format-Spezifizierer**, die dem Formatierer darüber Auskunft geben, wie er das Argument formatieren soll. `%s` steht für eine unformatierte Ausgabe eines Strings. Nach dem Format-String folgt ein Vararg (oder alternativ das Feld direkt) mit den Werten, auf die sich die Format-Spezifizierer beziehen.

Spezifizierer	Steht für...	Spezifizierer	Steht für...
<code>%n</code>	neue Zeile	<code>%b</code>	Boolean
<code>%%</code>	Prozentzeichen	<code>%s</code>	String
<code>%c</code>	Unicode-Zeichen	<code>%d</code>	Dezimalzahl
<code>%x</code>	Hexadezimalschreibweise	<code>%t</code>	Datum und Zeit
<code>%f</code>	Fließkommazahl	<code>%e</code>	wissenschaftliche Notation

Der Zeilenvorschub ist vom Betriebssystem abhängig, und `%n` gibt uns ein gutes Mittel an die Hand, um an dieses Zeilenvorschubzeichen (oder diese Zeichenfolge) zu kommen. Dann kommt der `format()`-Aufruf auch mit einem Argument aus, und es lautet `String.format("%n")`.



```
final class java.lang.String  
implements CharSequence, Comparable<String>, Serializable
```

- ✓ static String format(String format, Object... args)
Liefert einen formatierten String, der aus dem String und den Argumenten hervorgeht.
- ✓ static String format(Locale l, String format, Object... args)
Liefert einen formatierten String, der aus der gewünschten Sprache, dem String und den Argumenten hervorgeht.

Intern werkeln `java.util.Formatter` (keine `java.text.Format`-Objekte), die sich auch direkt verwenden lassen; dort ist auch die Dokumentation festgemacht.

`System.out.printf()`

Soll eine mit `String.format()` formatierte Zeichenkette gleich ausgegeben werden, so muss dazu nicht `System.out.print(String.format(format, args))`; angewendet werden. Praktischerweise findet sich zum Formatieren und Ausgeben die aus `String` bekannte Methode `format()` auch in den Klassen `PrintWriter` und `PrintStream` (das `System.out`-Objekt ist vom Typ `PrintStream`). Da jedoch der Methodenname `format()` nicht wirklich konsistent zu den anderen `printXXX()`-Methoden ist, haben die Entwickler die `format()`-Methoden auch unter dem Namen `printf()` zugänglich gemacht (die Implementierung von `printf()` ist eine einfache Weiterleitung zur Methode `format()`).

Beispiel: Gib die Zahlen von 0 bis 16 hexadezimal aus:

```
for ( int i = 0x0; i <= 0xf; i++ )  
    System.out.printf( "%x%n", i ); // 0 1 2 ... e f
```

Auch bei `printf()` ist als erstes Argument ein `Locale` möglich.

Positionsangaben

Beispiel: `System.out.printf("%te. %1$tb%n", t); // 28. Okt`

Die Angabe mit `Position$` ist eine Positionsangabe, und so bezieht sich `1$` auf das erste Argument, `2$` auf das zweite und so weiter (interessant ist, dass hier die Nummerierung nicht bei null beginnt).

Die Positionsangabe im Formatstring ermöglicht zwei Dinge:

- ✓ Wird, wie in dem Beispiel, das gleiche Argument mehrmals verwendet, ist es unnötig, es mehrmals anzugeben. So wiederholt `printf("%te. %tb%n", t, t)` das Argument `t`, was die Angabe einer Position vermeidet. Statt `%te. %1$tb%n` lässt sich natürlich auch `%1$te. %1$tb%n` schreiben, also auch für das erste Argument ausdrücklich die Position 1 vorschreiben.
- ✓ Die Reihenfolge der Parameter kann immer gleich bleiben, aber der Formatstring kann die Reihenfolge später ändern.

Der zweite Punkt ist wichtig für lokalisierte Ausgaben. Dazu ein Beispiel: Eine Bildschirmausgabe soll den Vor- und Nachnamen in unterschiedlichen Sprachen ausgeben. Die Reihenfolge der Namensbestandteile kann jedoch unterschiedlich sein, und nicht immer steht in jeder Sprache der Vorname vor dem Nachnamen. Im Deutschen heißt es im Willkommenstext dann "Hallo Christian Ullenboom", aber in der (erfundenen) Sprache Bwatuti hieße es "Jambo Ullenboom Christian":

FormatPosition.java, main()

```
Object[] formatArgs = { "Christian", "Ullenboom" };

String germanFormat = "Hallo %1$s %2$s";
System.out.printf( germanFormat, formatArgs );
System.out.println();

String bwatutiFormat = "Jambo %2$s %1$s";
System.out.printf( bwatutiFormat, formatArgs );
```

Die Aufrufreihenfolge für Vor-/Nachname ist immer die gleiche, aber der Formatstring, der zum Beispiel extern aus einer Konfigurationsdatei oder Datenbank kommt, kann diese Reihenfolge ändern und so der Landessprache anpassen.



Bezieht sich ein nachfolgendes Formatelement auf das vorangehende Argument, so kann ein `<` gesetzt werden:

```
Calendar c1 = new GregorianCalendar( 1973, 2, 12 );
Calendar c2 = new GregorianCalendar( 1985, 8, 2 );
System.out.printf( "%te. %<tb %<ty, %2$te. %<tb %<ty%n",
                  c1,                      c2 );      // 12. Mrz 73, 2. Sep 85
```

Die Angaben für Monat und Jahr beziehen sich jeweils auf die vorangehenden Positionen. So muss nur einmal `c1` und `c2` angegeben werden.

2.4 Format-Klassen

Die Methode `format()` formatiert Zahlen, Datumswerte und sonstige Ausgaben und benötigt wegen ihrer Komplexität eine Beschreibung von mehreren Bildschirmseiten. Dabei gibt es noch eine andere Möglichkeit, für unterschiedliche Typen von zu formatierenden Werten eigene Klassen zu haben:

- ✓ `DateFormat`: Formatieren von Datums-/Zeitwerten
- ✓ `NumberFormat`: Formatieren von Zahlen
- ✓ `MessageFormat`: Formatieren für allgemeine Programmmeldungen

Die Klassen haben gemeinsam, dass sie die abstrakte Klasse `Format` erweitern und so eine gemeinsame Schnittstelle haben. Jede dieser Klassen implementiert auf jeden Fall die Methode `format()` zur Ausgabe und zum Parsen, also zur Konvertierung vom String in das Zielobjekt, die Methode `parseObject()`.


```
abstract class java.text.Format
    implements Serializable, Cloneable
```

- ✓ `String format(Object obj)`
Formatiert das Objekt `obj` und gibt eine Zeichenkette zurück.
- ✓ `abstract StringBuffer format(Object obj, StringBuffer toAppendTo, FieldPosition pos)`
Formatiert ein Objekt und hängt den Text an den angegebenen `StringBuffer` an (eine Methode mit `StringBuilder` gibt es nicht). Kann die Zeichenkette nicht mit `format()` nach den Regeln des Format-Objekts ausgegeben werden, löst die Methode eine `IllegalArgumentException` aus. Ist die Formatierungsanweisung falsch, so gibt `format()` das Unicode-Zeichen `\uFFFD` zurück.
- ✓ `Object parseObject(String source)`
Analysiert den Text von Anfang an.
- ✓ `abstract Object parseObject(String source, ParsePosition pos)`
Der Text wird ab der Stelle `pos` umgewandelt. Konnte `parseObject()` die Zeichenkette nicht zurück-übersetzen, so folgt eine `ParseException`. `parseObject(String, ParsePosition)` verändert das `ParsePosition`-Objekt nicht und gibt die null-Referenz zurück.
- ✓ `Object clone()`
Gibt eine Kopie zurück.

Die Mehrzahl der `Format`-Unterklassen implementiert statische Fabrikmethoden der Art:

- ✓ `static XXXFormat getYYYInstance()`
Liefert ein Formatierungsobjekt mit den Formatierungsregeln für das voreingestellte Land.
- ✓ `static XXXFormat getYYYInstance(Locale l)`
Liefert ein Formatierungsobjekt mit den Formatierungsregeln für das angegebene Land. So erlauben die Unterklassen von `Format` es dem Benutzer auch, weitere Objekte zu erzeugen, die an die speziellen Sprachbesonderheiten der Länder angepasst sind.

Zahlen, Prozente und Währungen mit `NumberFormat` und `DecimalFormat` formatieren

`NumberFormat` widmet sich der Ausgabe von Zahlen. Dabei unterstützt die Klasse vier Typen von Ausgaben, für die es jeweils eine statische Fabrikmethode gibt.

```
abstract class java.text.NumberFormat
    extends Format
```

- ✓ `static NumberFormat getInstance()`
Liefert den einfachen Formatierer für Zahlen.
- ✓ `static NumberFormat getIntegerInstance()`
Liefert einen Formatierer, der den Nachkommateil abschneidet und rundet.
- ✓ `static NumberFormat getPercentInstance()`
Liefert einen Formatierer, der Fließkommazahlen über die `format()`-Methode im Bereich von 0 bis 1 annimmt und dann als Prozentzahl formatiert. Nachkommastellen werden abgeschnitten.
- ✓ `static NumberFormat getCurrencyInstance()`
Liefert einen Formatierer für Währungen, der ein Währungszeichen zur Ausgabe hinzufügt.

Die genannten vier statischen Methoden gibt es jeweils in der parameterlosen Variante und in der Variante mit einem `Locale`-Objekt, um etwa das Währungszeichen oder das Kommazeichen anzupassen.

Dezimalzahlformatierung mit DecimalFormat

`DecimalFormat` ist eine Unterklasse von `NumberFormat` und ermöglicht individuellere Anpassungen an die Ausgabe. Dem Konstruktor kann ein Formatierungsstring übergeben werden, sozusagen eine Vorlage, wie die Zahlen zu formatieren sind. Die Formatierung einer Zahl durch `DecimalFormat` erfolgt mit Rücksicht auf die aktuell eingestellte Sprache:

`DecimalFormatDemo.java, main()`

```
double d = 12345.67890;
DecimalFormat df = new DecimalFormat( "###,##0.00" );
System.out.println( df.format(d) ); // 12.345,68
```

Der Formatierungsstring kann eine Menge von Formatierungsanweisungen vertragen; im Beispiel kommen `#`, `0` und das Komma vor. Die beiden wichtigen Symbole sind jedoch `0` und `#`. Beide repräsentieren Ziffern. Der Unterschied tritt erst dann zutage, wenn weniger Zeichen zum Formatieren da sind, als im Formatierungsstring genannt werden.

Symbol	Bedeutung
0	Repräsentiert eine Ziffer - ist die Stelle nicht belegt, wird eine Null angezeigt.
#	Repräsentiert eine Ziffer - ist die Stelle nicht belegt, bleibt sie leer, damit führende Nullen und unnötige Nullen hinter dem Komma nicht angezeigt werden.
.	Dezimaltrenner. Trennt Vor- und Nachkommastellen.
,	Gruppert die Ziffern (eine Gruppe ist so groß wie der Abstand von "," zu ".").
;	Trennzeichen. Links davon steht das Muster für positive Zahlen, rechts davon das Muster für negative Zahlen.
-	Das Standardzeichen für das Negativpräfix
%	Die Zahl wird mit 100 multipliziert und als Prozentwert ausgewiesen.
\u2030	Die Zahl wird mit 1.000 multipliziert und als Promillewert ausgezeichnet.
\u00A4	Nationales Währungssymbol (€ für Deutschland)
\u00A4\u00A4	Internationales Währungssymbol (EUR für Deutschland)
X	Alle anderen Zeichen - symbolisch X - können ganz normal benutzt werden.
'	Ausmaskieren von speziellen Symbolen im Präfix oder Suffix

Hier sehen wir ein Beispiel für die Auswirkungen der Formatieranweisungen auf einige Zahlen:

Format	Eingabezahl	Ergebnis
##	12.3456	12
##	123456	123456
.00	12.3456	12,35
.00	.3456	,35
0.00	.789	0,79
#.000000	12.34	12,340000
,###	12345678.901	12.345.679
#.#+(#.)	12345678.901	12345678,9
#.#+(#.)	-12345678.901	(12345678,9)
,###.## \u00A4	12345.6789	12.345,68 €
,#00.00 \u00A4\u00A4	-12345678.9	-12.345.678,90 EUR
,#00.00 \u00A4\u00A4	0.1	00,10 EUR

Währungen angeben und die Klasse Currency

Die NumberFormat-Klasse liefert mit `getCurrencyInstance()` ein Format-Objekt, das neben der Dezimalzahl auch noch ein Währungssymbol mit anzeigt. So liefert `NumberFormat.getCurrencyInstance().format(12345.6789)` dann 12.345,68 €, also automatisch mit einem Euro-Zeichen. Dass es ein Euro-Zeichen ist und kein Yen-Symbol, liegt einfach daran, dass Java standardmäßig das eingestellte Land "sieht" und daraus die Währung ableitet. Wenn wir explizit den Formatter mit einem Land initialisieren, etwa wie in

```
NumberFormat frmt1 = DecimalFormat.getCurrencyInstance( Locale.FRANCE );
System.out.println( frmt1.format( 12345.6789 ) ); // 12 345,68 €
```

so ist die Währung automatisch Euro (denn Frankreich nutzt den Euro); schreiben wir `DecimalFormat.getCurrencyInstance(Locale.JAPAN)`, ist sie Yen, und wir bekommen ¥12,346 (es gibt standardmäßig keine Nachkommastellen beim Yen). `Locale`-Objekte repräsentieren immer eine Sprachregion.

`DecimalFormat` beziehungsweise schon die Oberklasse `NumberFormat` ermöglicht die explizite Angabe der Währung. In der Java-Bibliothek wird sie durch die Klasse `java.util.Currency` repräsentiert.

`NumberFormat` liefert mit `getCurrency()` die eingestellte `Currency`, die zur Formatierung verwendet wird, und `setCurrency()` setzt sie neu. Das löst Szenarios, in denen etwa ein Euro-Zeichen die Währung darstellt, aber die Zahlenformatierung englisch ist, wie die folgenden Zeilen zeigen:

```
NumberFormat frmt = DecimalFormat.getCurrencyInstance( Locale.ENGLISH );
frmt.setCurrency( Currency.getInstance( "EUR" ) );
System.out.println( frmt.format( 12345.6789 ) ); // EUR12,345.68
```

Die `Currency`-Klasse bietet drei statische Methoden, die `Currency`-Objekte liefern. Da ist zum einen `getAvailableCurrencies()`, was ein `Set<Currency>` liefert, und zum anderen gibt es die beiden Fabrikfunktionen `getInstance(Locale locale)` und `getInstance(String currencyCode)`. `Currency`-Objekte besitzen eine ganze Reihe von Objektfunktionen, die etwa den ISO-4217-Währungscode liefern oder den ausgeschriebenen Währungsnamen (und das auch noch in verschiedenen Sprachen, wenn gewünscht).

Folgendes Programm geht über alle Währungen und gibt die zentralen Informationen aus:

```
for ( Currency currency : Currency.getAvailableCurrencies() )
{
    System.out.printf( "%s, %s, %s (%s)%n",
        currency.getCurrencyCode(),
        currency.getSymbol(),
        currency.getDisplayName(),
        currency.getDisplayName( Locale.ENGLISH ) );
}
```

Wir bekommen dann mehr als 200 Ausgaben, und die Ausgabe beginnt mit:

```
EGP, EGP, Ägyptisches Pfund (Egyptian Pound)
IQD, IQD, Irak Dinar (Iraqi Dinar)
GHS, GHS, Ghana Cedi (Ghana Cedi)
AFN, AFN, Afghani (Afghani)
MUR, MUR, Mauritius Rupie (Mauritius Rupee)
SGD, SGD, Singapur Dollar (Singapore Dollar)
...
```

MessageFormat und Pluralbildung mit ChoiceFormat

`MessageFormat` ist eine besondere Unterklasse von `Format`, die für die Formulierung von Nachrichten gedacht ist. Die Klasse ist in etwa mit den Formatierungen über `printf()`, nur werden bei `MessageFormat` die Platzhalter immer per Index in geschweiften Klammern angesprochen.

Beispiel: Formuliere einen Nachrichtenstring mit drei Feldern:

```
int soldCars = 10;
double sum      = 1234534534;
String s = MessageFormat.format( "{0} Auto(s) verkauft am {1,date} zum "+
                           "Gesamtpreis von {2,number,currency} .",
                           soldCars, new Date(), sum );
System.out.println( s );
```

Die Ausgabe ist:

"10 Auto(s) verkauft am 01.07.2011 zum Gesamtpreis von 1.234.534,534,00 €."

Sie ist automatisch lokalisiert, die Sprache lässt sich jedoch wieder als `Locale`-Objekt übergeben.

ChoiceFormat

Eine besondere Möglichkeit ist die Verbindung von `MessageFormat` und `ChoiceFormat`, um das Problem zu lösen, das unser Beispiel im Fall von verkauften Autos mit "Auto(s)" löst. Im Deutschen ist die Pluralbildung anspruchsvoll, da es "0 Autos, 1 Auto, 2 Autos, 3 Autos" usw. heißt aber nur "0 Koffer, 1 Koffer, 2 Koffer, ...". Das in Software zu modellieren ist nicht ganz einfach, aber mit `ChoiceFormat` lässt es sich lösen. Dem Konstruktor werden zum Generieren der Ausgabe zwei Felder mitgegeben: Ein Limit-Array kodiert Bereiche, und ein zweites Feld enthält die zugeordneten Elemente für den Bereich.

Beispiel: Löse das Problem mit "0 Autos, 1 Auto, 2 Autos" usw.:

```
MessageFormat formatter = new MessageFormat( "Du hast {0} {1} verkauft." );
double[] limits   = { 0.,      1.,      2.      };
String[] formats = { "Autos", "Auto", "Autos" };
ChoiceFormat choices = new ChoiceFormat( limits, formats );
formatter.setFormatByArgumentIndex( 1, choices );
int size = 4;
Object[] params = { size, size };
System.out.println( formatter.format( params ) ); // Du hast 4 Autos verkauft.
```

Das Feld `{0.., 1.., 2..}` interpretiert sich so: Liegt der Wert zwischen größer gleich 0 und echt kleiner 1 (also bei Ganzzahlen ist er effektiv 0), wird das erste Element des Feldes {"Autos", "Auto", "Autos"}, also "Autos", gewählt. Liegt es zwischen größer gleich 1 und echt kleiner 2, dann ist es das "Auto". Alles was echt größer 2 ist, wird auf "Autos" abgebildet. Für die 2 im Feld ist `Math.nextUp(1.)` eine Alternative, und wenn auf "Komma-Autos" gewechselt wird, ist es gleich korrekt, denn es heißt zwar "1 Auto" aber "1,5 Autos". (Genau genommen sind es sogar "1,0 Autos", wodurch die ganze Pluralbildung wegfällt.)

Die API-Dokumentation der Klasse `ChoiceFormat` gibt weitere Beispiele und zeigt insbesondere, wie sich die Bereiche auch in den Strings selbst kodieren lassen, was für externe Übersetzungsdateien optimal ist.

3 Generics<T>

In diesem Kapitel erfahren Sie

- ✓ was generische Datentypen sind
- ✓ wie Sie generische Datentypen realisieren und einsetzen

Voraussetzungen

- ✓ Datentypen

3.1 Einführung in Java Generics

Generics zählen zu den komplexesten Sprachkonstrukten in Java. Wir wollen uns Generics in zwei Schritten nähern: von der Seite des Nutzers und von der Seite des API-Designers. Das Nutzen von generisch deklarierten Typen ist deutlich einfacher, sodass wir diese niedrig hängende Frucht zuerst pflücken wollen. Das Java-Buch für Fortgeschrittene dokumentiert sehr detailliert Generics aus der Sicht des API-Designers; die gepflückten Früchte werden dann veredelt.

Mensch versus Maschine: Typprüfung des Compilers und der Laufzeitumgebung

Eine wichtige Eigenschaft von Java ist, dass der Compiler die Typen prüft und so weiß, welche Eigenschaften vorhanden sind und welche nicht. Hier unterscheidet sich Java von dynamischen Programmiersprachen wie Python oder PHP, die erst spät eine Prüfung zur Laufzeit vornehmen.

In Java gibt es zwei Instanzen, die die Typen prüfen, und diese sind unterschiedlich schlau. Wir haben die JVM mit der absoluten Typ-Intelligenz, die unsere Anwendung ausführt und als letzte Instanz prüft, ob wir ein Objekt nicht einem falschen Typ zuweisen. Dann haben wir noch den Compiler, der zwar gut prüft, aber teilweise etwas zu gutgläubig ist und dem Entwickler folgt. Macht der Entwickler Fehler, kann dieser die JVM ins Verderben stürzen und zu einer `Exception` führen. Alles hat mit der expliziten Typanpassung zu tun. Ein zunächst unkompliziertes Beispiel:

```
Object o = "String";
String s = (String) o;
```

Dem Compiler wird über den expliziten Typecast das `Object` `o` für ein `String` verkauft. Das ist in Ordnung, weil ja `o` tatsächlich ein `String`-Objekt referenziert. Problematisch wird es, wenn der Typ `nicht` auf `String` gebracht werden kann, wir dem Compiler aber eine Typanpassung anweisen:

```
Object o = Integer.valueOf( 42 );           // oder mit Autoboxing: Object o = 42;
String s = (String) o;
```

Der Compiler akzeptiert die Typanpassung, und es folgt kein Fehler zur Übersetzungszeit. Es ist jedoch klar, dass diese Anpassung von der JVM nicht durchgeführt werden kann - daher folgt zur Laufzeit eine `ClassCastException`, da eben ein `Integer` nicht auf `String` gebracht werden kann.

Bei Generics geht es nun darum, dem Compiler mehr Informationen über die Typen zu geben und `ClassCastException`-Fehler zu vermeiden.

Taschen

In unseren vorangehenden Beispielen drehte sich alles um Spieler und in einem Raum platzierte Spielobjekte. Stellen wir uns vor, der Spieler hat eine Tasche (engl. **pocket**), die etwas enthält. Da nicht bekannt ist, was genau er in der Tasche hat, müssen wir einen Basistyp nehmen, der alle möglichen Objekttypen repräsentiert. Das soll in unserem ersten Beispiel der allgemeinste Basistyp `Object` sein, sodass der Benutzer alles in seiner Tasche tragen kann. (Primitive Datentypen können über Wrapper-Objekte gespeichert werden, was seit Java 5 dank Autoboxing leicht möglich ist.)

com/tutego/insel/nongeneric/Pocket.java, Pocket

```
public class Pocket
{
    private Object value;
    public Pocket() {}
    public Pocket( Object value ) { this.value = value; }
    public void set( Object value ) { this.value = value; }
    public Object get() { return value; }
    public boolean isEmpty() { return value == null; }
    public void empty() { value = null; }
}
```

Es gibt einen Standard- sowie einen parametrisierten Konstruktor. Mit `set()` lassen sich Objekte in die Tasche setzen und über die Zugriffsmethode `get()` wieder auslesen.

Geben wir einem Spieler eine rechte und eine linke Tasche:

com/tutego/insel/nongeneric/Player.java, Player

```
public class Player
{
    public String name;
    public Pocket rightPocket;
    public Pocket leftPocket;
}
```

Zusammen mit einem Spieler, der eine rechte und eine linke Tasche hat, ist ein Beispiel schnell geschrieben. Unser Spieler `michael` soll sich in beide Taschen Zahlen legen. Dann wollen wir sehen, in welcher Tasche er die größere Zahl versteckt hat.

com/tutego/insel/nongeneric/PlayerPocketDemo.java, main()

```
Player michael = new Player();
michael.name = "Omar Arnold";
Pocket pocket = new Pocket();
Long aBigNumber = 11111111111111L;
pocket.set( aBigNumber );                                // (1)
michael.leftPocket = pocket;
michael.rightPocket = new Pocket( 2222222222222222L );

System.out.println( michael.name + " hat in den Taschen " +
                    michael.leftPocket.get() + " und " + michael.rightPocket.get() );

Long val1 = (Long) michael.leftPocket.get();   // (2)
Long val2 = (Long) michael.rightPocket.get();

System.out.println( val1.compareTo( val2 ) > 0 ? "Links" : "Rechts" );
```

Das Beispiel hat keine besonderen Fallen, allerdings fallen zwei Sachen auf, die prinzipiell unschön sind. Die haben damit zu tun, dass die Klasse `Pocket` mit dem Typ `Object` zum Speichern der Tascheninhalte sehr allgemein deklariert wurde und alles aufnehmen kann:

- ✓ Beim Initialisieren wäre es gut, zu sagen, dass die Tasche nur einen bestimmten Typ (etwa `Long`) aufnehmen kann. Wäre eine solche Einschränkung möglich, dann lassen sich wie in Zeile (1) auch wirklich nur `Long`-Objekte in die Tasche setzen und nichts anderes, etwa `Integer`-Objekte.
- ✓ Beim Entnehmen (2) des Tascheninhalts mit `get()` müssen wir uns daran erinnern, was wir hineingelegt haben. Fordern Datenstrukturen besondere Typen, dann sollte dies auch dokumentiert sein. Doch wenn der Compiler wüsste, dass in der Tasche auf jeden Fall ein `Long` ist, dann könnte die Typanpassung wegfallen und der Programmcode wäre kürzer. Auch könnte uns der Compiler warnen, wenn wir versuchen würden, das `Long` als `Integer` aus der Tasche zu ziehen. Unser Wissen möchten wir gerne dem Compiler geben! Denn wenn in der Tasche ein `Long`-Objekt ist, wir es aber als `Integer` annehmen und eine explizite Typanpassung auf `Integer` setzen, meldet der Compiler zwar keinen Fehler, aber zur Laufzeit gibt es eine böse `ClassCastException`.

Um es auf den Punkt zu bringen: Der Compiler berücksichtigt im oberen Beispiel die Typsicherheit nicht ausreichend. Explizite Typanpassungen sind in der Regel unschön und sollten vermieden werden. Aber wie können wir die Taschen typsicher machen?

Eine Lösung wäre, eine neue Klasse für jeden in der Tasche zu speichernden Typ zu deklarieren, also einmal eine `PocketLong` für den Datentyp `long`, dann vielleicht `PocketInteger` für `int`, `PocketString` für `String` usw. Das Problem bei diesem Ansatz ist, dass viel Code kopiert wird - fast identischer Code. Das ist keine vernünftige Lösung; wir können nicht für jeden Datentyp eine neue Klasse schreiben, und die Logik bleibt die gleiche. Wir wollen wenig schreiben, aber Typsicherheit beim Compilieren bekommen und nicht erst die Typsicherheit zur Laufzeit, wo uns vielleicht eine `ClassCastException` überrascht. Es wäre gut, wenn wir den Typ bei der Deklaration frei, allgemein, also "generisch" halten könnten, und sobald wir die Tasche benutzen, den Compiler dazu bringen könnten, auf diesen dann angegebenen Typ zu achten und die Korrektheit der Nutzung sicherzustellen.

Die Lösung für dieses Problem heißt **Generics**. (In C++ werden diese Typen von Klassen parametrisierte Klassen oder Templates (Schablonen) genannt.) Diese Technik wurde in Java 5 eingeführt. Sie bietet Entwicklern ganz neue Möglichkeiten, um Datenstrukturen und Algorithmen zu programmieren, die von einem Datentyp unabhängig, also **generisch** sind.

Generische Typen deklarieren

Wollen wir `Pocket` in einen generischen Typ umbauen, so müssen wir an den Stellen, an denen `Object` vorkam, einen Typstellvertreter, einen sogenannten **formalen Typparameter** einsetzen, der durch eine **Typvariable** repräsentiert wird. Der Name der Typvariablen muss in der Klassendeklaration angegeben werden.

Die Syntax für den generischen Typ von `Pocket` ist folgende:

`com/tutego/insel/generic/Pocket.java, Pocket`

```
public class Pocket<T>
{
    private T value;
    public Pocket() {}
    public Pocket( T value ) { this.value = value; }
    public void set( T value ) { this.value = value; }
    public T get() { return value; }
    public boolean isEmpty() { return value != null; }
    public void empty() { value = null; }
}
```

Wir haben die Typvariable `T` definiert und verwenden diese jetzt anstelle von `Object` in der `Pocket`-Klasse. Bei generischen Typen steht die Angabe der Typvariable nur einmal zu Beginn der Klassendeklaration in spitzen Klammern hinter dem Klassennamen. Der Typparameter kann nun fast überall dort genutzt werden, wo auch ein herkömmlicher Typ stand. (`T t = new T();` ist zum Beispiel nicht möglich.) In unserem Beispiel ersetzen wir direkt `Object` durch `T`, und fertig ist die generische Klasse.

Namenskonvention

Formale Typparameter sind in der Regel einzelne Großbuchstaben wie `T` (steht für Typ), `E` (Element), `K` (Key/Schlüssel), `V` (Value/Wert). Sie sind nur Platzhalter und keine wirklichen Typen. Möglich wäre etwa auch Folgendes, doch davon ist absolut abzuraten, da `Elf` viel zu sehr nach einem echten Klassentyp als nach einem formalen Typparameter aussieht:

```
public class Pocket<Elf>
{
    private Elf value;
    public void set( Elf value ) { this.value = value; }
    public Elf get() { return value; }
}
```

Es dürfen nicht nur Elfen in die Klasse, sondern alle Typen.

Wofür Generics noch gut ist

Es gibt eine ganze Reihe von Beispielen, in denen Speicherstrukturen wie unsere Tasche nicht nur für einen Datentyp `Long` sinnvoll sind, sondern grundsätzlich für alle Typen, wobei aber die Implementierung (relativ) unabhängig vom Typ der Elemente ist. Das gilt zum Beispiel für einen Sortieralgorithmus, der mit der Ordnung der Elemente arbeitet. Wenn zwei Elemente größer oder kleiner sein können, muss ein Algorithmus lediglich diese Eigenschaft nutzen können. Es ist dabei egal, ob es Zahlen vom Typ `Long`, `Double` oder auch `Strings` oder Kunden sind - der Algorithmus selbst ist davon nicht betroffen. Der häufigste Einsatz von Generics sind Container, die typsicher gestaltet werden sollen.

Die Idee, Generics in Java einzuführen, ist schon älter und geht auf das Projekt *Pizza* beziehungsweise das Teilprojekt *GJ* (A Generic Java Language Extension) von Martin Odersky (der auch der Schöpfer der Programmiersprache *Scala* ist), Gilad Bracha, David Stoutamire und Philip Wadler zurück. *GJ* wurde dann die Basis des JSR 14, "Add Generic Types To The Java Programming Language".



Generics nutzen

Um die neue `Pocket`-Klasse nutzen zu können, müssen wir sie zusammen mit einem Typparameter angeben; es entstehen hier zwei **parametisierte Typen**:

com/tutego/insel/generic/PocketPlayer.java, main() Teil 1

```
Pocket<Integer> intPocket = new Pocket<Integer>();
Pocket<String> stringPocket = new Pocket<String>();
```

Der konkrete Typ steht immer hinter dem Klassen-/Schnittstellennamen in spitzen Klammern. (Dass auch XML in spitzen Klammern daherkommt und XML als groß und aufgebläht gilt, wollen wir nicht als Parallele zu Javas Generics sehen.) Die Tasche `intPocket` ist eine Instanz eines generischen Typs mit dem konkreten Typargument `Integer`. Diese Tasche kann jetzt offiziell nur `Integer`-Werte enthalten, und die Tasche `stringPocket` enthält nur Zeichenketten. Das prüft der Compiler auch, und wir benötigen keine Typanpassung mehr:

com/tutego/insel/generic/PocketPlayer.java, main() Teil 2

```
intPocket.set( 1 );
int x = intPocket.get();           // Keine Typanpassung mehr nötig
stringPocket.set( "Selbstzerstörungsauslösungsschalterhintergrundbeleuchtung" );
String s = stringPocket.get();
```

Der Entwickler macht so im Programmcode sehr deutlich, dass die Taschen einen `Integer` enthalten und nichts anderes. Da Programmcode häufiger gelesen als geschrieben wird, sollten Autoren immer so viele Informationen wie möglich über den Kontext in den Programmcode legen. Zwar leidet die Lesbarkeit etwas, da insbesondere beim Instanziieren der Typ sowohl rechts wie auch links angegeben werden muss und die Syntax bei geschachtelten Generics lang werden kann, doch wie wir später sehen werden, lässt sich das ab Java 7 abkürzen.

Das Schöne für die Typsicherheit ist, dass nun alle Eigenschaften mit dem angegebenen Typ geprüft werden. Wenn wir etwa aus `intPocket` mit `get()` auf das Element zugreifen, ist es vom Typ `Integer` (und durch Unboxing gleich `int`), und `set()` erlaubt auch nur ein `Integer`. Das macht den Programmcode robuster und durch den Wegfall der Typanpassungen kürzer und lesbarer.

Zusammenfassung der bisherigen Generics-Begriffe

Begriff	Beispiel
Generischer Typ (engl. <i>generic type</i>)	<code>Pocket<T></code>
Typvariable oder formaler Typparameter (engl. <i>formal type parameter</i>)	<code>T</code>
Parametrisierter Typ (engl. <i>parameterized type</i>)	<code>Pocket<Long></code>
Typparameter (engl. <i>actual type parameter</i>)	<code>Long</code>
Originaltyp (engl. <i>raw type</i>)	<code>Pocket</code>



Typparameter können in Java Klassen, Schnittstellen, Aufzählungen und Arrays davon sein, aber keine primitiven Datentypen. Das schränkt die Möglichkeiten zwar ein, doch da es Autoboxing gibt, lässt sich damit leben. Und wenn `null` in der `Pocket<Integer>` liegt, führt ein Unboxing zur Laufzeit zur `NullPointerException`.

Geschachtelte Generics

Ist ein generischer Typ wie `Pocket<T>` gegeben, gibt es erst einmal keine Einschränkung für `T`. So beschränkt sich `T` nicht auf einfache Klassen- oder Schnittstellentypen, sondern kann auch wieder ein generischer Typ sein. Das ist logisch, denn jeder generische Typ ist ja ein eigenständiger Typ, der (fast) wie jeder andere Typ genutzt werden kann:

`com/tutego/insell/generic/PocketPlayer.java, main() Teil 3`

```
Pocket<Pocket<String>> pocketOfPockets = new Pocket<Pocket<String>>();
pocketOfPockets.set( new Pocket<String>() );
pocketOfPockets.get().set( "Inner Pocket<String>" );
System.out.println( pocketOfPockets.get().get() ); // Inner Pocket<String>
```

Hier enthält die Tasche eine Innentasche, die eine Zeichenkette "Inner Pocket<String>" speichert. Bei Dingen wie diesen ist schnell offensichtlich, wie hilfreich Generics für den Compiler (und uns) sind. Ohne Generics sähen eben alle Taschen gleich aus.

Präzise mit Generics	Unpräzise ohne Generics
<code>Pocket<String> stringPocket;</code>	<code>Pocket stringPocket;</code>
<code>Pocket<Integer> intPocket;</code>	<code>Pocket intPocket;</code>
<code>Pocket<Pocket<String>> pocketOfPockets;</code>	<code>Pocket pocketOfPockets;</code>

Nur ein gut gewählter Name und eine präzise Dokumentation können bei nicht-generisch deklarierten Variablen helfen. Vor Java 5 haben sich Entwickler damit geholfen, mithilfe eines Blockkommentars Generics anzudeuten, etwa in `Pocket/*<String>*/ stringPocket`.

Keine Arrays von parametrisierten Typen

Die folgende Anweisung bereitet eine einzige Tasche mit einem Feld von Strings vor:

```
Pocket<String[]> pocketForArray = new Pocket<String[]>();
```

Aber lässt sich auch ein Array von mehreren Taschen, die jeweils Strings enthalten, deklarieren? Ja. Doch während die Deklaration noch möglich ist, ist die Initialisierung schlichtweg ungültig:

```
Pocket<String>[] arrayOfPocket;
arrayOfPocket = new Pocket<String>[2]; // ☹ Compilerfehler
```

Der Grund liegt in der Umsetzung in Bytecode verborgen, und die beste Lösung ist, die komfortablen Datenstrukturen aus dem `java.util`-Paket zu nutzen. Für Entwickler ist ein `List<Pocket<String>>` sowieso sexier als ein `Pocket<String>[]`. Laufzeiteinbußen sind kaum zu erwarten. Da Arrays aber vom Compiler automatisch bei variablen Argumentlisten eingesetzt werden, gibt es ein Problem, wenn die Parametervariable eine Typvariable ist. Bei Signaturen wie `f(T... params)` hilft die Annotation `@SafeVarargs`, die Compiler-Meldung zu unterdrücken.

Diamonds are forever

Bei der Initialisierung einer Variablen, deren Typ generisch ist, fällt auf, dass der Typparameter zweimal angegeben werden muss. Bei geschachtelten Generics fällt die Mehrarbeit unangenehm auf. Nehmen wir eine Liste, die `Maps` enthält, wobei der Assoziativspeicher Datumswerte mit `Strings` verbindet:

```
List<Map<Date, String>> listOfMaps;
listOfMaps = new ArrayList<Map<Date, String>>();
```

Der Typparameter `Map<Date, String>` steht einmal auf der Seite der Variablen-deklaration und einmal beim `new`-Operator.

Seit Java 7 erlaubt der Compiler eine verkürzte Schreibweise, denn er kann aus dem Kontext den Typ ableiten; diese Eigenschaft - **Typ-Inferenz** (engl. **type inference**) genannt - wird uns noch einmal über den Weg laufen.

Der Diamantoperator

Verfügt der Compiler über alle Typinformationen, so können beim new-Operator die Typparameter entfallen, und es bleibt lediglich ein Pärchen spitzer Klammern:

Beispiel

Ab Java 7 wird aus

```
List<Map<Date, String>> listOfMaps = new ArrayList<Map<Date, String>>();
```

einfach:

```
List<Map<Date, String>> listOfMaps = new ArrayList<>();
```

Wegen des Aussehens der spitzen Klammern <> nennt sich der Typ, für den die spitzen Klammern stehen, auch **Diamanttyp** (engl. **diamond type**). Das Pärchen <> wird auch **Diamantoperator** (engl. **diamond operator**) genannt, und es ist ein Operator, weil er den Typ herausfindet, weshalb er auch **Diamant-Typ-Inferenz-Operator** genannt wird.

Es ist ungewöhnlich, dass der Java-Compiler hier den Typ der linken Seite betrachtet - denn bei long val = 10000000000; macht er das auch nicht. Doch darüber müssen wir uns keine so großen Gedanken machen, denn dies ist nicht das einzige Problem in der Java-Grammatik ...



Einsatzgebiete des Diamanten

Der Diamant in unserem Beispiel ersetzt den gesamten Typparameter Map<String, String>. Es ist nicht möglich, ihn nur zum Teil bei geschachtelten Generics einzusetzen. So schlägt new ArrayList<Map<>>() fehl. Auch ist nur bei new der neue Diamant-Operator erlaubt, und es wäre falsch, ihn auch auf der linken Seite bei der Variablen Deklaration einzusetzen und ihn etwa auf der rechten Seite bei der Bildung des Exemplars zu nutzen. Eine Deklaration wie List<> listOfMaps; führt somit zum Compilerfehler, denn der Compiler würde nicht bei jeder folgenden Nutzung irgendwelche Typen ableiten können.

Da der Diamant bei new eingesetzt wird, kann er - bis auf einige Ausnahmen, die wir uns später anschauen - immer dort eingesetzt werden, wo Exemplare gebildet werden. Das nächste Nonsense-Beispiel zeigt vier Einsatzgebiete:

```
import java.util.*;

public class WereToUseTheDiamond
{
    public static List<String> foo( List<String> list )
    {
        return new ArrayList<>();
    }

    public static void main( String[] args )
    {
        List<String> list = new ArrayList<>();
        list = new ArrayList<>();
        foo( new ArrayList<>( list ) );
    }
}
```

Die Einsatzorte sind:

- ✓ bei Deklarationen und der Initialisierung von Attributen und lokalen Variablen
- ✓ bei der Initialisierung von Attributen, lokalen Variablen bzw. Parametervariablen
- ✓ als Argument bei Methoden-/Konstruktoraufrufen
- ✓ bei Methodenrückgaben

Ohne Frage ist der erste und zweite Fall der sinnvollste. Fast überall kann der Diamant die Schreibweise abkürzen. Besonders im ersten Fall spricht nichts Grundsätzliches gegen den Einsatz, bei den anderen drei Punkten muss berücksichtigt werden, ob nicht vielleicht die Lesbarkeit des Programmcodes leidet. Wenn zum Beispiel in mitten in einer Methode eine Datenstruktur mit `list = new ArrayList<>()` initialisiert wird, aber die Variablendeclaration nicht auf der gleichen Bildschirmseite liegt, ist mitunter für den Leser nicht sofort sichtbar, was denn genau für Typen in der Liste sind. Um den Diamanten zu testen, haben die Entwickler ein Tool geschrieben, das durch das JDK läuft und schaut, welche generisch genutzten news durch den Diamanten vereinfacht werden könnten. (Heraus kamen etwa 5000 Stellen.) Nicht jedes Team hat jede erlaubte Konvertierung hin zum Diamanten akzeptiert. So wollte das Team, das die Java-Security-Bibliotheken pflegt, weiterhin die explizite Schreibweise der Generics bei Zuweisungen beibehalten.

Diamant nicht möglich

Es gibt Situationen, in denen die Typableitung nicht so funktioniert wie erwartet. Oftmals hat das mit dem Einsatz des Diamanten bei Methodenaufrufen zu tun, sodass anzuraten ist - auch schon aus Gründen der Programmverständlichkeit -, bei Methodenaufrufen grundsätzlich auf Diamanten zu verzichten.

Ein Beispiel mit zwei für den Compiler unlösbar Fällen:

```
class NoDiamondsForYou
{
    static void out( List<String> list ) { }

    public static void main( String[] args )
    {
        out( new ArrayList<>() ); // ☹ Compilerfehler
        List<String> list = new ArrayList<>().subList(0, 1); // ☹ Compilerfehler
    }
}
```

Die Typinferenz ist komplex, und glücklicherweise muss ein Entwickler sich nicht um die interne Arbeitsweise kümmern. Wenn der Diamant, wie im Beispiel, nicht möglich ist, löst eine explizite Typangabe das Problem. (Für Java 7 standen zwei Algorithmen zur Typauswahl zur Auswahl: simpel und komplex. Der komplexe Ansatz bezieht neben den Typeninformationen, die eine Zuweisung liefert, noch den Argumenttyp mit ein. Zunächst verwendete das Team den einfachen Algorithmus, wechselte ihn jedoch später, da der komplexe Ansatz auf Algorithmen zurückgreift, die der Compiler auch an deren Stellen einsetzt. Ein paar mehr Details geben die Präsentation http://blogs.oracle.com/darcy/resource/JavaOne/J1_2010-ProjectCoin.pdf und Beiträge in der Mailingliste <http://mail.openjdk.java.net/pipermail/coin-dev/2009-November/002393.html>.)

Generische Schnittstellen

Eine Schnittstelle kann genauso als generischer Typ deklariert werden wie eine Klasse. Werfen wir einen Blick auf die Schnittstellen `java.lang.Comparable` und einen Ausschnitt von `java.util.Set` (Schnittstelle, die Operationen für Mengenoperationen vorschreibt), die beide seit Java 5 mit einer Typvariablen ausgestattet sind.

Generische Deklaration der Schnittstellen Comparable und Set

<code>public interface Comparable<T></code>	<code>public interface Set<E> extends Collection<E></code>
{ <code>public int compareTo(T o);</code> }	{ <code>int size();</code> <code>boolean isEmpty();</code> <code>boolean contains(Object o);</code> <code>Iterator<E> iterator();</code> <code>Object[] toArray();</code> <code><T> T[] toArray(T[] a);</code> <code>boolean add(E e);</code> <code>...</code> }

Wie bekannt, greifen die Methoden auf die Typvariablen `T` und `E` zurück. Bei `Set` ist weiterhin zu erkennen, dass sie selbst eine generisch deklarierte Schnittstelle erweitert.

Beim Einsatz von generischen Schnittstellen lassen sich die folgenden zwei Benutzungsmuster ableiten:

- ✓ Ein nicht-generischer Klassentyp löst Generics bei der Implementierung auf.
- ✓ Ein generischer Klassentyp implementiert eine generische Schnittstelle und gibt die Parametervariable weiter.

Nicht-generischer Klassentyp löst Generics bei der Implementierung auf

Im ersten Fall implementiert eine Klasse die generisch deklarierte Schnittstelle und gibt einen konkreten Typ an. Alle numerischen Wrapper-Klassen implementieren zum Beispiel Comparable und füllen den Typparameter genau mit dem Typ der Wrapper-Klasse:

```
public final class Integer extends Number implements Comparable<Integer>
{
    public int compareTo( Integer anotherInteger ) { ... }
    ...
}
```

Durch diese Nutzung wird für den Anwender die Klasse Integer Generics-frei.

Generischer Klassentyp implementiert generische Schnittstelle und gibt die Parametervariable weiter

Die Schnittstelle Set schreibt Operationen für Mengen vor. Eine Klasse, die Set implementiert, ist zum Beispiel HashSet. Der Kopf der Typdeklaration ist folgender:

```
public class HashSet<E>
    extends AbstractSet<E>
    implements Set<E>, Cloneable, java.io.Serializable
```

Es ist abzulesen, dass Set eine Typvariable E deklariert, die HashSet nicht konkretisiert. Der Grund ist, dass die Datenstruktur Set vom Anwender als parametrisierter Typ verwendet wird und nicht aufgelöst werden soll.

In manchen Situationen wird auch Void als Typparameter eingesetzt. Wenn etwa interface I<T> { T foo(); } eine Typvariable T deklariert, ohne dass es bei der Implementierung von I etwas zurückzugeben gibt, dann kann der Typparameter Void sein:

```
class C implements I<Void> {
    @Override public Void foo() { return null; }
```

Allerdings sind void und Void unterschiedlich, denn bei Void muss es eine Rückgabe geben, was ein return null notwendig macht.



Generische Methoden/Konstruktoren und Typ-Inferenz

Die bisher genannten generischen Konstruktionen sahen im Kern wie folgt aus:

- ✓ class Klassenname<T> { ... }
- ✓ interface Schnittstellename<T> { ... }

Eine an der Klassen- oder Schnittstellendeklaration angegebene Typvariable kann in allen nicht-statischen Eigenschaften des Typs angesprochen werden.

Beispiel: Folgendes führt zu einem Fehler:

```
class Pocket<T> {
    static void foo( T t ) { };           // ☹ Compilerfehler
}
```

Der Eclipse-Compiler meldet: "Cannot make a static reference to the non-static type T".

Doch was machen wir, wenn

- ✓ statische Methoden eine eigene Typvariable nutzen wollen?
- ✓ unterschiedliche statische Methoden unterschiedliche Typvariablen nutzen möchten?

Eine Klasse kann auch ohne Generics deklariert werden, aber **generische Methoden** besitzen. Ganz allgemein kann jeder Konstruktor, jede Objektmethode und jede Klassenmethode einen oder mehrere formale Typparameter deklarieren. Sie stehen dann nicht mehr an der Klasse, sondern an der Methoden-/Konstruktor-deklaration und sind "lokal" für die Methode beziehungsweise den Konstruktor. Das allgemeine Format ist:

Modifizierer <**Typvariable (n)**> Rückgabetyp Methodename (Parameter) throws-Klausel

Ganz zufällig das eine oder andere Argument

Interessant sind generische Methoden insbesondere für Utility-Klassen, die nur statische Methoden anbieten, aber selbst nicht als Objekt vorliegen. Das folgende Beispiel zeigt das anhand einer Methode `random()`:

com/tutego/insel/generic/GenericMethods.java, GenericMethods

```
public class GenericMethods
{
    public static <T> T random( T m, T n )
    {
        return Math.random() > 0.5 ? m : n;
    }

    public static void main( String[] args )
    {
        String s = random( "Analogkäse", "Gel-Schinken" );
        System.out.println( s );
    }
}
```

Dabei deklariert `<T> T random(T m, T n)` eine generische Methode, wobei der Rückgabetyp und Parametertyp durch eine Typvariable `T` bestimmt wird. Die Angabe von `<T>` beim Klassennamen ist bei dieser Syntax entfallen und wurde auf die Deklaration der Methode verschoben.



Natürlich kann eine Klasse als generischer Typ und eine darin enthaltene Methode als generische Methode mit unterschiedlichem Typ deklariert werden. In diesem Fall sollten die Typvariablen unterschiedlich benannt sein, um den Leser nicht zu verwirren. So bezieht sich im Folgenden `T` bei `sit()` eben nicht auf die Parametervariable der Klasse `Lupilu`, sondern auf die der Methode:

```
class Lupilu<T> { <T> void sit( T val ); } // Verwirrend
class Lupilu<T> { <V> void sit( V val ); } // Besser
```

Der Compiler auf der Suche nach Gemeinsamkeiten

Den Typ (der wichtig für die Rückgabe ist) leitet der Compiler also automatisch aus dem Kontext, das heißt aus den Argumenten, ab. Diese Eigenschaft nennt sich **Typ-Inferenz** (engl. **type inference**) Das hat weitreichende Konsequenzen.

Bei der Deklaration `<T> T random(T m, T n)` sieht es vielleicht auf den ersten Blick so aus, als ob die Variablentypen `m` und `n` absolut gleich sein müssen. Das stimmt aber nicht, denn bei den Typen geht der Compiler in der Typhierarchie so weit nach oben, bis er einen gemeinsamen Typ findet.

Aufruf	Identifizierte Typen	Gemeinsame Basistypen
random("Essen", 1)	String, Integer	Object, Serializable, Comparable
random(1L, 1D)	Long, Double	Object, Number, Comparable
random(new Point(), new StringBuilder())	Point, StringBuilder	Object, Serializable, Cloneable

Es fällt auf, aber überrascht nicht, dass `Object` immer in die Gruppe gehört.

Die Schnittmenge der Typen bildet im Fall von `random()` die gültigen Rückgabetypen.

Erlaubt sind demnach für die Parametertypen `String` und `Integer`:

```
Object      s1 = random( "Essen", 1 );
Serializable s2 = random( "Essen", 1 );
Comparable   s3 = random( "Essen", 1 );
```

Knappe Fabrikmethoden

Der bei Java 7 eingeführte Diamant-Typ kürzt Variablen Deklarationen mit Initialisierung einer Referenzvariablen angenehm ab. Heißt es unter Java 6 noch zwingend

```
Pocket<String> p = new Pocket<String>();
```

und musste `<String>` zweimal angegeben werden, so erlaubt der Diamant ("<>") ab Java 7 Folgendes:

```
Pocket<String> p = new Pocket<>();
```

Mit der Typ-Inferenz gibt es eine alternative Lösung auch für Java 5 und Java 6. Geben wir unserer Klasse `Pocket` eine Fabrikmethode

```
public static <T> Pocket<T> newInstance()
{
    return new Pocket<T>();
}
```

so ist folgende Alternative möglich:

```
Pocket<String> p = Pocket.newInstance();
```

Aus dem Ergebnistyp `Pocket<String>` leitet der Compiler den tatsächlichen Typparameter `String` für die Tasche ab. Und ist bei einem einfachen Typ wie `String` die Schreibersparnis noch gering, wird der Code bei verschachtelten Datenstrukturen kürzer. Soll die Tasche einen Assoziativspeicher aufnehmen, der eine Zeichenkette mit einer Liste von Zahlen assoziiert, so schreiben wir kompakt:

```
Pocket<Map<String, List<Integer>>> p = Pocket.newInstance();
```

Natürlich müssen entweder die Klassen selbst oder Utility-Klassen diese Fabrikmethoden anbieten, denn von selbst sind sie nicht vorhanden. Da unter Java 7 der Diamant aber den Programmcode bei den Initialisierungen verkürzt, ist es unwahrscheinlich, dass Bibliotheksautoren sie jetzt nachrüsten.

Die populäre Google-Collections-Bibliothek (<http://code.google.com/p/guava-libraries/>) bietet Utility-Klassen wie `Lists`, `Maps`, `Sets` mit genau den statischen Fabrikfunktionen wie `newArrayList()`, `newLinkedList()`, `newTreeMap()`, `newHashSet()`, die von der Typ-Inferenz leben.

Generische Methoden mit explizitem Typparameter

Es gibt Situationen, in denen der Compiler nicht aus dem Kontext über Typ-Inferenz den richtigen Typ ableiten kann. Zum Beispiel ist Folgendes nicht möglich:

```
boolean hasPocket = true;
Pocket<String> pocket = hasPocket ? Pocket.newInstance() : null;
```

Der Eclipse-Compiler meldet "Type mismatch: cannot convert from `Pocket<Object>` to `Pocket<String>`".

Die Lösung: Wir müssen bei `Pocket.newInstance()` den Typparameter `String` explizit angeben:

```
Pocket<String> pocket = hasPocket ? Pocket.<String>newInstance() : null;
```

Die Syntax ist etwas gewöhnungsbedürftig, doch in der Praxis ist die explizite Angabe selten nötig.

Beispiel: Ist das Argument der statischen Methode `Arrays.asList()` ein Feld, dann ist der explizite Typparameter nötig, da der Compiler nicht erkennen kann, ob das Feld selbst das eine Element der Rückgabekommt ist oder ob das Feld die Vararg-Umsetzung ist und alle Elemente des Feldes in die Rückgabeliste kommen:

```
List<String> list11 = Arrays.asList( new String[] { "A", "B" } );
List<String> list12 = Arrays.asList( "A", "B" ); // Parameter ist als vararg definiert
System.out.println( list11 ); // [A, B]
System.out.println( list12 ); // [A, B]
List<String> list21 = Arrays.<String>asList( new String[] { "A", "B" } );
List<String> list22 = Arrays.<String>asList( "A", "B" );
System.out.println( list21 ); // [A, B]
System.out.println( list22 ); // [A, B]
List<String[]> list31 = Arrays.<String[]>asList( new String[] { "A", "B" } );
// List<String[]> list32 = Arrays.<String[]>asList( "A", "B" );
System.out.println( list31 ); // [[Ljava.lang.String;@69b332]
```

Zunächst gilt es festzuhalten, dass die Ergebnisse für `list11`, `list12`, `list21` und `list22` identisch sind. Der Compiler setzt ein Vararg automatisch als Feld um und übergibt das Feld der `asList()`-Methode. Im Bytecode sehen daher die Aufrufe gleich aus.

Bei `list21` und `list22` ist der Typparameter jeweils explizit angegeben, aber nicht wirklich nötig, da ja das Ergebnis wie `list11` bzw. `list12` ist. Doch der Typparameter `String` macht deutlich, dass die Elemente im Feld, also die Vararg-Argumente, Strings sind. Spannend wird es bei `list31`. Zunächst zum Problem: Ist `new String[] {"A", "B"}` das Argument einer Vararg-Methode, so ist das mehrdeutig, weil genau dieses Feld das erste Element des vom Compiler automatisch aufgebauten Varargs-Feldes sein könnte (dann wäre es ein Feld im Feld) oder - und das ist die interne Standardumsetzung - der Java-Compiler das übergebene Feld als die Vararg-Umsetzung interpretiert. Diese Doppeldeutigkeit löst `<String[]>`, da in dem Fall klar ist, dass das von uns aufgebaute String-Feld das einzige Element eines neuen Varargs-Feldes sein muss. Und `Arrays.<String[]> asList()` stellt heraus, dass der Typ der Feldelemente `String[]` ist. Daher funktioniert auch die letzte Variablendeclaration nicht, denn bei `asList("A", "B")` ist der Elementtyp `String`, aber nicht `String[]`.

3.2 Umsetzen der Generics, Typlösung und Raw-Types

Zum Verständnis der Generics und um zu erfahren, was zur Laufzeit an Informationen vorhanden ist, lohnt es sich, sich anzuschauen, wie der Compiler Generics in Bytecode übersetzt.

Realisierungsmöglichkeiten

Im Allgemeinen gibt es zwei Möglichkeiten, um generische Typen zu realisieren:

- ✓ **Heterogene Variante:** Für jeden Typ (etwa `String`, `Integer`, `Point`) wird individueller Code erzeugt, also drei Klassendateien. Die Variante nennt sich auch Code-Spezialisierung.
- ✓ **Homogene Übersetzung:** Aus der parametrisierten Klasse wird eine Klasse erzeugt, die anstelle des Typparameters nur `Object` einsetzt. Für den konkreten Typparameter werden Typanpassungen in die Anweisungen eingebaut.
- ✓ Java nutzt die homogene Übersetzung, und der Compiler erzeugt nur eine Klassendatei. Es gibt keine multiplen Kopien der Klasse - weder im Bytecode noch im Speicher.

Typlösung (Type Erasure)

Übersetzt der Java-Compiler die generischen Anwendungen, so löscht er dabei alle Typinformationen, da die Java-Laufzeitumgebung keine Generics im Typsystem hat. Das nennt sich **Typlösung** (engl. **type erasure**). Wir können uns das so vorstellen, dass alles wegfällt, was in spitzen Klammern steht, und dass jede Typvariable zu `Object` wird. (Sind Bounds im Spiel - eine Typeinschränkung, die später noch vorgestellt wird -, wird ein präziserer Typ statt `Object` genutzt.)

Mit Generics	Nach der Typlösung
<pre>public class Pocket<T> { private T value; public void set(T value) { this.value = value; } public T get() { return value; } }</pre>	<pre>public class Pocket { private Object value; public void set(Object value) { this.value = value; } public Object get() { return value; } }</pre>

Generische Klasse im Quellcode und wie sie nach der Typlösung aussieht

So entspricht der Programmcode nach der Typlösung genau dem, was wir selbst auch ohne Generics am Anfang programmiert haben. Auch bei der Nutzung wird gelöscht:

Mit Generics	Nach der Typlösung
<pre>Pocket<Integer> p = new Pocket<Integer>(1); p.set(1); Integer i = p.get();</pre>	<pre>Pocket p = new Pocket(1); p.set(1); Integer i = (Integer) p.get();</pre>

Nutzung generischer Klassen und wie es nach der Typlösung aussieht

Beim Herausholen über `get()` fügt der Compiler genau die explizite Typanpassung ein, die wir in unserem ersten Beispiel noch von Hand eingesetzt haben.

Wenn der Compiler Bytecode erzeugt, der auch für ältere JVMs keine Probleme bereitet, so stellt sich die Frage, wo denn die Informationen abgespeichert sind, dass ein Typ generisch deklariert wurde oder nicht. Irgendwo muss das stehen, denn der Compiler weiß das ja. Die Antwort ist, dass der Compiler diese Typinformationen, die nicht Teil des Typsystems der JVM sind, als Signatur-Attribute in den Konstantenpool des Bytecodes legt. Das Attribut ist ein UTF-8 Text, der von älteren Compilern als Kommentar überlesen wird. Mit dem Diassembler `javap` und dem Schalter `-verbose` lassen sich diese Informationen anzeigen. Interessierte bekommen weitere Informationen unter: http://java.sun.com/docs/books/jvms/second_edition/jvms-clarify.html.



Das große Ziel: Interoperabilität

Interoperabilität stand bei der Einführung der Generics ganz oben auf der Wunschliste. Zwei wichtige Anforderungen sind:

- ✓ Die neuen mit Generics deklarierten Klassen - wie `List<E>` - müssen auf jeden Fall noch von altem Programmcode, der zum Beispiel mit einem Java 1.4-Compiler erzeugt wurde, nutzbar sein. Das funktioniert so, dass generische deklarierte Klassen im Bytecode für einen "alten" Compiler so aussehen, als gäbe es keine Generics. Wir sprechen von **Typlösung**. Hätte Sun sich nicht dieses Kompatibilitätsziel auf die Fahnen geschrieben, hätte die Umsetzung auch anders ausfallen können. Denn die Konsequenz der Typlösung ist, dass es keine Informationen über den Typparameter zur Laufzeit gibt. Das führt zu Überraschungen und Einschränkungen (insbesondere bei Arrays), die wir uns gleich anschauen werden. Was wir hier vor uns haben, ist der Wunsch nach **Bytecode-Kompatibilität**.
- ✓ Auf der anderen Seite gibt es neben der Bytecode-Kompatibilität auch noch die **Quellcode-Kompatibilität**. Alter Programmcode, der zum Beispiel Listen als `List list;` statt zum Beispiel als `List<String> list;` nutzt, soll immer noch übersetzbare sein, auch wenn er die überarbeiteten Datenstrukturen nicht generisch nutzt. Warnungen sind akzeptabel, aber keine Compilerfehler. Es gibt Millionen Zeilen alten Quellcodes, die Listen ohne Generics nutzen, ohne dass sofort ein Team alle Programmstellen anfasst und Typparameter einführt.

Java Generics gehen bei den Typbeschreibungen weit über das hinaus, was C++-Templates bieten. In C++ kann ein beliebiger Typparameter eingesetzt werden - was zu unglaublichen Fehlermeldungen führt. Der C++-Compiler führt somit eher eine einfache Ersetzung durch. Doch durch die heterogene Umsetzung generiert der C++-Compiler für jeden genutzten Template-Typ unterschiedlichen (und wunderbar optimierten) Maschinencode. Im Fall von Java würde die heterogene Variante zu sehr vielen sehr ähnlichen Klassen führen, die sich nur in ein paar Typanpassungen unterschieden. Und da in Java sowieso nur Referenzen als Typvariablen möglich sind und keine primitiven Typen, ist auch eine besondere Optimierung an dieser Stelle nicht möglich.



Durch die Code-Spezialisierung sind aber andere Dinge in C++ machbar, die in Java unmöglich sind, zum Beispiel Template-Metaprogramming. Der Compiler wird in diesem Fall als eine Art Interpreter für rekursive Template-Aufrufe genutzt, um später optimalen Programmcode zu generieren. Das ist funktionale Programmierung mit einem Compiler ...

Probleme aus der Typlösung

Typlösung ist für die Laufzeitumgebung praktisch, weil sie überhaupt nicht an die Generics angepasst werden muss. So sehen zum Beispiel die seit Java 5 generisch deklarierten Datenstrukturen nach dem Übersetzungsvorgang genauso aus wie unter Java 1.4 und sind damit voll kompatibel. Sonst aber stellt die Typlösung ein riesiges Problem dar, weil die Typinformationen zur Laufzeit nicht vorhanden sind. (Dass diese Typinformationen nicht vorliegen, wird auch damit begründet, dass die Laufzeit leiden könnte. Microsoft war das hingegen egal, dort besteht Generizität in der Common Language Runtime (CLR), also auch in der Laufzeitumgebung. Microsoft ist damit einen klaren Schritt voraus. Doch gab es Generics (Parametric Polymorphismus ist der offizielle Name) auch wie in Java nicht von Anfang an; es zog erst in Version 2 in die Sprache und CLR ein. Die alten Datenstrukturen wurden einfach als veraltet markiert, und die Entwickler waren gezwungen, auf die neuen generischen Varianten umzusteigen.)



Für Java 7 stand auf der Aufgabenliste, die generischen Parameter auch zur Laufzeit zugänglich zu machen. Das wurde jedoch verschoben und kommt vielleicht irgendwann, in Java 8, Java 9, Java 2020, ... Das Stichwort dazu ist **Reified Generics**, also generische Informationen, die auch zur Laufzeit komplett zugänglich sind.

Kein new T

Da durch die Typlösung bei Deklarationen wie `Pocket<T>` die Parametervariable durch `Object` ersetzt wird, lässt sich zum Beispiel in der Tasche **nicht** Folgendes schreiben, um ein neues Exemplar eines Tascheninhalts zu erzeugen:

Gedacht: Mit Generics (Compilerfehler!)	Konsequenz aus Typlösung
<pre>class Pocket<T> { T newPocketContent() { return new T(); } }</pre>	<pre>class Pocket<T> { Object newPocketContent() { return new Object(); } }</pre>

Warum "new T()" nicht funktionieren kann: nur ein "new Object()" würde gebildet

Als Aufrüfer von `newPocketContent()` erwarten wir aber nicht immer ein lächerliches `Object`, sondern ein Objekt vom Typ `T`.

Kein instanceof

Der `instanceof`-Operator ist bei parametrisierten Typen ungültig, auch wenn das praktisch wäre, um zum Beispiel aufgrund der tatsächlichen Typen eine Fallunterscheidung vornehmen zu können:

```
void printType( Pocket<?> p )
{
    if ( p instanceof Pocket<Number> )           // ☹ illegal generic type for instanceof
        System.out.println( "Pocket mit Number" );
    else if ( p instanceof Pocket<String> ) // ☹ illegal generic type for instanceof
        System.out.println( "Pocket mit String" );
}
```

Der Compiler meldet zu Recht einen Fehler - nicht nur eine Warnung -, weil es die Typen `Pocket<String>` und `Pocket<Number>` zur Laufzeit gar nicht gibt: Es sind nur typgelöschte `Pocket`-Objekte. Nach der Typlösung würde unsinniger Code entstehen:

```
void printType( Pocket p )
{
    if ( p instanceof Pocket )
        ...
    else if ( p instanceof Pocket )
        ...
}
```

Keine Typanpassungen auf parametrisierten Typ

Typanpassungen wie

```
Pocket<String> p = (Pocket<String>) new Pocket<Integer>(); // ☣ Compilerfehler
```

sind illegal. Wir haben ja extra Generics, damit der Compiler die Typen testet. Und durch die Typlöschung verschwindet der Typparameter, sodass der Compiler Folgendes erzeugen würde:

```
Pocket p = (Pocket) new Pocket();
```

Kein .class für generische Typen und keine Class-Objekte mit Typparameter zur Laufzeit

Ein hinter einen Typ gesetztes .class liefert das Class-Objekt zum jeweiligen Typ.

```
Class<Object> objectClass = Object.class;
Class<String> stringClass = String.class;
```

Class selbst ist als generischer Typ deklariert.

Bei generischen Typen ist das .class nicht erlaubt. Zwar ist noch (mit Warnung) Folgendes gültig:

```
Class<Pocket> pocketClass = Pocket.class;
```

aber dies nicht mehr:

```
Class<Pocket<String>> pocketClass = Pocket<String>.class; // ☣ Compilerfehler
```

Der Grund ist die Typlöschung: Alle Class-Objekte für einen Typ sind gleich und haben zur Laufzeit keine Information über den Typparameter:

```
Pocket<String> p1 = new Pocket<String>();
Pocket<Integer> p2 = new Pocket<Integer>();
System.out.println( p1.getClass() == p2.getClass() ); // true
```

Alle Exemplare von generischen Typen werden zur Laufzeit vom gleichen Class-Objekt repräsentiert. Hinter Pocket<String> und Pocket<Integer> steckt also immer nur Pocket. Kurz gesagt: Alles in eckigen Klammern verschwindet zur Laufzeit.

Keine generischen Ausnahmen

Grundsätzlich ist eine Konstruktion wie class MyClass<T> extends SuperClass erlaubt. Aber der Compiler enthält eine spezielle Regel, die verhindert, dass eine generische Klasse Throwable (Exception und Error sind Unterklassen von Throwable) erweitern kann. Wäre zum Beispiel

```
class MyException<T> extends Exception { } // ☣ Compilerfehler
```

erlaubt, könnte im Quellcode vielleicht ein

```
try { }
catch ( MyException<Typ1> e ) { }
catch ( MyException<Typ2> e ) { }
```

stehen, doch durch die Typlöschung würde das auf zwei identische catch-Blöcke hinauslaufen, was nicht erlaubt ist.

Keine statischen Eigenschaften

Statische Eigenschaften hängen nicht an einzelnen Objekten, sondern an Klassen. Pocket kann zum Beispiel einmal als parametrisierter Typ Pocket<String> und einmal als Pocket<Integer> auftauchen, also als zwei Instanzen. Aber kann Pocket auch eine statische Methode deklarieren, die auf den formalen Typparameter der Klasse zurückgreift? Nein, das geht nicht.

Würden wir in `Pocket` etwa die folgende statische Methode einsetzen

```
public static boolean isEmpty( T value ) { return value == null; } // ☹
```

so gäbe es bei `T` die Fehlermeldung "Cannot make a static reference to the non-static type `T`".

Statische Variablen und die Parameter/Rückgaben von statischen Methoden sind nicht an ein Exemplar gebunden. Eine Typvariable jedoch, so wie wir sie bisher verwendet haben, ist immer mit dem Exemplar verbunden. Das `T` für den `value` ist ja erst immer dann festgelegt, wenn wir zum Beispiel `Pocket<String>` oder `Pocket<Integer>` mit einem Exemplar verbinden. Bei `Pocket.isEmpty("")`; zum Beispiel kann der Compiler nicht wissen, was für ein Typ gemeint ist, da für statische Methodenaufrufe ja keine Exemplare nötig sind, also nie ein parametrisierter Typ festgelegt werde. Das Nutzen von Code wie `Pocket<String>.isEmpty("")` führt zu einem Compilerfehler, denn die Syntax ist nicht erlaubt.

Statische generische Methoden sind natürlich möglich, wie wir schon gesehen haben; sie haben dann eine eigene Typvariable.

Kein Überladen mit Typvariablen

Kommt nach der Typlösung einfach nur `Object` heraus, kann natürlich keine Methode einmal mit einer Typvariablen und einmal mit `Object` parametrisiert sein. Folgendes ist nicht erlaubt:

```
public class Pocket<T>
{
    public T value;
    public void set( T value ) { this.value = value; }
    public void set( Object value ) { this.value = value; } // ☹ Compilerfehler!
}
```

Der Compiler liefert: "Method `set(T)` has the same erasure `set(Object)` as another method in type `Pocket<T>`". Ist der Typ spezieller, also etwa `String`, sieht das wieder anders aus. Dann taucht die Frage auf, welche Methode bei `Pocket<String>` aufgerufen wird. Die Leser dürfen das gerne prüfen.

Es lassen sich keine Arrays generischer Klassen bilden

Die Nutzung von Generics bei Arrays schränkt der Compiler ebenfalls ein. Während

```
Pocket[] pockets = new Pocket[1];
```

gültig ist und mit einer Warnung versehen wird, führt bei

```
Pocket<String>[] pockets; // (1)
pockets = new Pocket<String>[1]; // (2) ☹ Compilerfehler
```

nicht die erste, aber die zweite Zeile zum Compilerfehler "Cannot create a generic array of `Pocket<String>`".

Typsicher kann das nicht genutzt werden, aber drei schnelle Lösungen sind denkbar:

- ✓ auf Generics ganz zu verzichten und ein `@SuppressWarnings("unchecked")` an die Feldvariable zu setzen
- ✓ den Typ durch ein Wildcard ersetzen, sodass es etwa zu einem `Pocket<?>[] pockets = new Pocket<?>[1];` kommt. Wildcards sind Platzhalter, die später noch detaillierter vorgestellt werden.
- ✓ gleich auf Datenstrukturen der Collection-API umsteigen, bei denen ein `Collection<String> pockets = new ArrayList<String>();` keine Probleme bereitet

Als Zusammenfassung lässt sich festhalten, dass Array-Variablen von generischen Typen zwar deklariert (1), dass aber keine Array-Objekte gebaut werden können (2). Mit einem Trick funktioniert es:

```
class PocketFullOfMoney extends Pocket<BigInteger> {}
Pocket<BigInteger>[] pockets = new PocketFullOfMoney[1];
```

Hübsch ist das nicht, denn es muss extra eine temporäre Klasse angelegt werden.

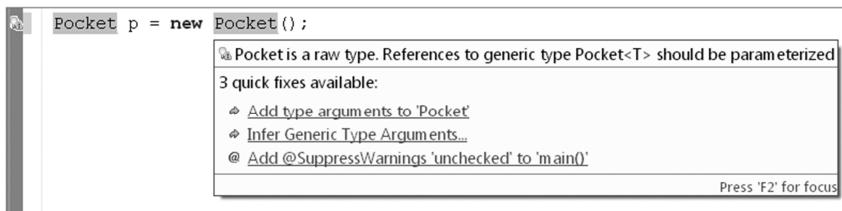
Raw-Type

Generische Klassen müssen nicht unbedingt parametrisiert werden, doch es ist einleuchtend, dass wir dem Compiler so viel Typinformation wie möglich geben sollten. Auf die Typparameter zu verzichten ist nur für die Rückwärtskompatibilität wichtig, da sonst viele parametrisierte neue Klassen nicht mehr mit altem Programmcode verwendet werden könnten. Wenn zum Beispiel `Pocket` unter Java 1.4 deklariert und mit den Sprachmitteln von Java 5 zu einem generischen Typ verfeinert wurde, kann es immer noch alten Programmcode geben, der wie folgt aussieht:

```
Pocket p = new Pocket();           // Gefährlich, wie wir gleich sehen werden
p.set( "Drei Pleitegeier, die Taschen voller Sand" );
String content = (String) p.get();
```

Ein generischer Typ, der nicht als parametrisierter Typ, also ohne Typargument, genutzt wird, heißt Raw-Type. In unserem Beispiel ist `Pocket` der Raw-Type von `Pocket<T>`. Bei einem Raw-Type kann der Compiler die Typkonformität nicht mehr prüfen, denn es ist der Typ nach der Typlösung; `get()` liefert `Object`, und `set(Object)` kann alles annehmen.

Ein unter Java 1.4 geschriebenes Programm nutzt also nur Raw-Types. Trifft ein Java 5-Compiler auf Programmcode, der einen generischen Typ nicht als parametrisierten Typ nutzt, fängt er an zu meckern, denn er wünscht, dass der Typ generisch verwendet wird.



Eclipse warnt Raw-Types standardmäßig an

Auch bei `set()` gibt der Compiler eine Warnung aus, denn er sieht eine Gefahr für die Typsicherheit. Die Methode `set()` ist so entworfen, dass sie ein Argument von dem Typ akzeptiert, mit dem sie parametrisiert wurde. Fehlt durch die Verwendung des Raw-Types der konkrete Typ, bleibt `Object`, und der Compiler gibt bei den sonst mit einem Typ präzisierten Methoden eine Warnung aus:

```
p.set( "Type safety: The method set(Object) belongs to the " +
       "raw type Pocket. References to generic type " +
       "Pocket<T> should be parameterized" );
```

Der Hinweis besagt, dass die Tasche hätte typisiert werden müssen. Wenn wir nicht darauf achten, kann das schnell zu Problemen führen:

```
Pocket<String> p1 = new Pocket<String>();
Pocket p2 = p1;                                // Compiler-Warnung
p2.set( new java.util.Date() );                // Compiler-Warnung
String string = p1.get();                      // ☹ ClassCastException
System.out.println( string );
```

Der Compiler gibt keinen Fehler, aber Warnungen aus. Die dritte Zeile ist hochgradig problematisch, denn über die nicht parametrisierte Tasche können wir beliebige Objekte eintüten. Da aber das Objekt hinter `p2` und dem typgelöschten `p1` identisch ist, haben wir ein Typproblem, das zur Laufzeit zu einer `ClassCastException` führt:

```
Exception in thread "main" java.lang.ClassCastException: java.util.Date cannot be cast to java.lang.String
```

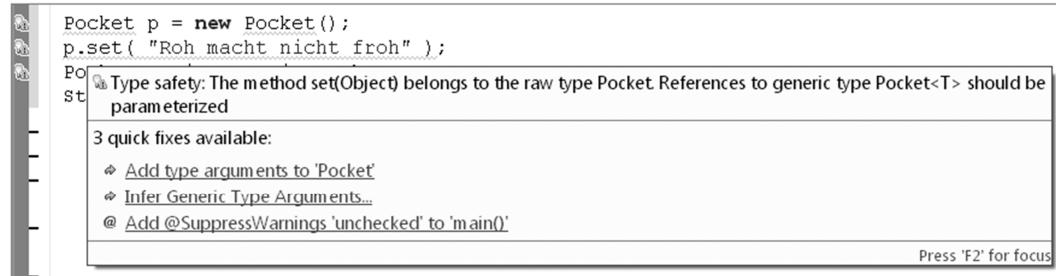
Es kann also nur die Empfehlung ausgesprochen werden, Raw-Types in neuen Programmen zu vermeiden, da ihre Verwendung zu Ausnahmen führen kann, die erst zur Laufzeit auffallen.

Typanpassungen

Ein Raw-Type lässt sich automatisch in eine speziellere Form bringen, wobei es natürlich Warnungen vom Compiler gibt.

```
Pocket p = new Pocket();                                // (1) Warnung
p.set( "Roh macht nicht froh" );                      // (2) Warnung
Pocket<String> stringPocket = p;                      // (3) Warnung
String result = stringPocket.get();                    // (4)
```

Bei der Variablen `p`, die wir über den Raw-Type nutzen (2), prüft der Compiler gar keine Typen in `set()`, denn er hat sie ja nie kennengelernt. Zeile (3) verkauft dem Compiler den Raw-Type als parametrisierten Typ. Eine explizite Typanpassung ist nicht nötig, denn Casts sind nur zwischen "echten" Typen gültig, wie `Object` auf `Pocket`, nicht aber von `Pocket` auf `Pocket<String>`, da `Pocket<String>` ja der gleiche Class-Typ ist (siehe "Kein .class für generische Typen und keine Class-Objekte mit Typparameter zur Laufzeit"). Eine Anweisung wie (4), die einen Nicht-String-Typ in die Tasche setzt, bringt keinen Fehler zur Übersetzungszeit, und so kann auch über diese Hintertür ein falscher Typ in die Tasche kommen.



Warnung von Eclipse bei Raw-Types

Annotation SuppressWarnings

In seltenen Fällen muss in den Typ konvertiert werden. Als Beispiel soll `cast()` dienen:

```
public <T> T cast( Object obj )
{
    return (T) obj; // Compilerwarnung: Type safety: Unchecked cast from Object to T
}
```

Lässt sich der Cast nicht vermeiden, um dem Compiler den Typ zu geben und ihn somit glücklich zu machen, setzen wir eine `@SuppressWarnings`-Annotation:

```
@SuppressWarnings("unchecked")
public <T> T cast( Object obj )
{
    return (T) obj;
}
```

Die Generics bieten uns Möglichkeiten, den Quellcode sicherer zu machen. Wir sollten diese Sicherheit nicht durch Raw-Types kaputt machen.

Unter den **Preferences** von Eclipse können drei Typen von Hinweisen für die Nutzung von Raw-Types angegeben werden: Der Compiler gibt einen harten Compilerfehler aus, eine Warnung, oder er ignoriert sie. Dass er Warnungen ausgibt, ist voreingestellt, und diese Vorgabe ist ganz gut.

3.3 Einschränken der Typen über Bounds

Bei generischen Angaben können die Typen weiter eingeschränkt werden. Das ist nützlich, da ein beliebiger Typ oft zu allgemein ist. Unsere Deklaration von `random()` sah keine Einschränkungen für die Typen vor:

```
public static <T> T random( T m, T n )
{
    return Math.random() > 0.5 ? m : n;
}
```

So ist auch Folgendes möglich:

```
Object o1 = new Object();
Object o2 = new Point();
System.out.println( random( o1, o2 ) );
```

Da der Typ beliebig ist, können auch Objekte übergeben werden, die vielleicht wenig Sinn ergeben, insbesondere in ihrer Kombination.

Einfache Einschränkungen mit extends

Bei der Deklaration eines generischen Typs kann vorgeschrieben werden, dass der spätere parametrisierte Typ eine bestimmte Klasse erweitert oder eine konkrete Schnittstelle implementiert. Soll unsere statische `random()`-Methode zum Beispiel nur Objekte vom Typ `CharSequence` (also Zeichenfolgen wie `String` und `StringBuffer/StringBuilder`) akzeptieren, so schreiben wir das in die Deklaration mit hinein:

com/tutego/insell/generic/BondageBounds.java, BondageBounds

```
public class BondageBounds
{
    public static <T extends CharSequence> T random( T m, T n )
    {
        return Math.random() > 0.5 ? m : n;
    }

    public static void main( String[] args )
    {
        String random1 = random( "Shinju", "Karada" );
        System.out.println( random1 );

        CharSequence random2 = random( "Ushiro", new StringBuilder("Takatekote") );
        System.out.println( random2 );
    }
}
```

Einen Aufruf mit zwei `Strings` lässt der Compiler korrekterweise durch, genauso wie mit `String` und `StringBuilder`, wobei der Rückgabetyp dann nur noch `CharSequence` ist.

Ein Fehler ist leicht zu provozieren. Dazu muss der Methode `random()` nur etwa ein `Point` übergeben werden - `Point` ist nicht vom Typ `CharSequence`. So führt

```
System.out.println( random( "", new Point() ) ); // ☹ Compilerfehler
                                                // "Bound mismatch"
```

zu der Fehlermeldung: "Bound mismatch: The generic method `random(T, T)` of type `BondageBounds` is not applicable for the arguments (`String, Point`). The inferred type `Serializable` is not a valid substitute for the bounded parameter `<T extends CharSequence>`".

Der Compiler führt eine Typ-Inferenz durch; das heißt, er schaut sich an, welche gemeinsamen Typen die Argumente `" "` und `Point` haben, und kommt auf `Serializable`. Der Typ hilft jedoch nicht weiter, denn wir wollten bloß `CharSequences` einsetzen können.

Typeeinschränkung für gemeinsame Methoden

Eine Typeeinschränkung wie `<T extends CharSequence>` ist interessant, da wir so wissen, dass ein konkreter Typparameter (etwa `String` oder `StringBuffer`) mindestens die Methoden der Schnittstelle `CharSequence` hat. Das ist logisch, denn bei einer Einschränkung des Typs wird der Compiler sicherstellen, dass die konkreten Typen die vorgeschriebene Schnittstelle implementieren (oder die Klasse erweitern) und damit die Methoden existieren.

Nehmen wir an, wir wollten ein typsicheres `max()` implementieren. Es soll den größeren der beiden Werte zurückgeben. Vergleiche lassen sich einfach tätigen, wenn die Objekte `Comparable` implementieren, denn `compareTo()` liefert einen Rückgabewert, der aussagt, welches Objekt nach der definierten Metrik kleiner, größer oder gleich ist.

```
com/tutego/inse1/generic/BondageBounds.java, max()

public static <T extends Comparable<T>> T max( T m, T n )
{
    return m.compareTo( n ) > 0 ? m : n;
}
```

Die Nutzung ist einfach:

```
System.out.println( max( "Kino", "Lesen" ) );           // Lesen
System.out.println( max( 12, 100 ) );                   // 100
```



Ohne Typ-Bound können nur die Methoden von `Object` verwendet werden, aber immerhin noch Methoden wie `equals()`, `hashCode()` und `toString()`.

Betrachten wir noch einen Fehlerfall. Intuitiv ist anzunehmen, dass alles, was vom Typ `Comparable` ist, auch ein gültiges Argument für `max()` ist. Das ist aber nicht ganz präzise, denn wir schreiben `<T extends Comparable<T>> T max(T m, T n)`, was bedeutet, dass der gemeinsame Typ für `m` und `n` laut Typ-Inferenz `Comparable` sein muss, nicht nur jeder einzelne. Was das bedeutet, zeigt die folgende Anweisung, die der Compiler mit einem Fehler ablehnt:

```
System.out.println( max( 12L, 100F ) ); // ☤ Compilerfehler "Bound mismatch"
```

Nach dem Boxing leitet der Compiler aus dem `Long` 12 und dem `Float` 100 den gemeinsamen Typ `Number` ab. (Zur Erinnerung: Der Compiler geht in der Typhierarchie so lange nach oben, bis ein gemeinsamer Typ gefunden wurde, der für `T` eingesetzt werden kann. Das ist `Number`.) Aber `Number` ist nicht vom Typ `Comparable`, und so folgt eine Fehlermeldung: "Bound mismatch: The generic method max(T, T) of type BondageBounds is not applicable for the arguments (Long, Float). The inferred type Number&Comparable<?> is not a valid substitute for the bounded parameter <T extends Comparable<T>>".



Bei `<T extends Comparable<T>>` handelt es sich um einen sogenannten *rekursiven Type-Bound*. Er kommt selten vor und soll hier auch nicht weiter vertieft werden. Bei Interesse gibt <http://tutego.de/go/getthistrick> weitere Hinweise. Wird die Methode `max()` zum Beispiel fälschlicherweise mit

```
static <T extends Comparable/*Hier fehlt was*/> T max( T m, T n ) { ... }
```

deklariert, so gibt der Compiler die Warnung "Comparable is a raw type. References to generic type Comparable<T> should be parameterized" aus. Ignorieren wir die Warnung, so lässt sich `max(12L, 100F)` tatsächlich aufrufen, doch es folgt eine `ClassCastException` mit "java.lang.Float cannot be cast to java.lang.Long".

Weitere Obertypen mit &

Soll der konkrete Typ zu mehreren Typen passen, lassen sich mit einem & weitere Obertypen hinzunehmen. Wichtig ist aber, dass nur eine Klassen-Vererbungsangabe stattfinden kann, also nur ein `extends` stehen darf, da Java keine Mehrfachvererbung auf Klassenebene unterstützt. Bei dem Rest muss es sich um implementierte Schnittstellen handeln. Die allgemeine Notation (für eine Klasse `C` und Schnittstellen `I1` bis `In`) ist:

```
T extends C & I1 & I2 & ... & In
```

Nehmen wir eine fiktive Oberklasse `Endeavour` und die Schnittstellen `Serializable` und `Comparable` an. (`Comparable` bekommt selbst einen Typparameter, was das Beispiel aus Gründen der Übersichtlichkeit auslässt.) Dann sind die folgenden Deklarationen prinzipiell erlaubt:

- ✓ `<T extends Endeavour>`
- ✓ `<T extends Serializable & Comparable>`
- ✓ `<T extends Endeavour & Serializable>`
- ✓ `<T extends Endeavour & Comparable & Serializable>`

Syntaktisch falsch wäre etwa `<T extends Endeavour & T extends Comparable>`, da das Schlüsselwort `extends` nur einmal vorkommen darf.

3.4 Typparameter in der throws-Klausel

Wir haben im Abschnitt "Keine generischen Ausnahmen" gesehen, dass durch die Typlösung eine Konstruktion wie `class MyException<T> extends Exception` nicht möglich ist. Allerdings ist ein Typparameter in der `throws`-Klausel erlaubt. Das gibt interessante Möglichkeiten für Klassen, die je nach Anwendungsfall einmal geprüfte oder ungeprüfte Ausnahmen auslösen können.

Deklaration einer Klasse mit Typvariable `<E extends Exception>`

Unsere Schnittstelle `CharIterable` soll von Klassen implementiert werden, die einen Strom von Zeichen liefern. `CharIterable` ist ein generischer Schnittstellentyp mit einem formalen Typparameter, der später eine Unterklasse von `Exception` sein muss:

```
com/tutego/insel/generic/CharIterable.java, CharIterable
public interface CharIterable<E extends Exception>
{
    boolean hasNext() throws E;
    char next() throws E;
}
```

Zeichen können etwa aus einer Datei, von einer Internetressource oder von einem String kommen, doch die Nutzung sieht immer gleich aus:

```
while ( iter.hasNext() )
    System.out.print( iter.next() );
```

Parametrisierter Typ bei Typvariable `<E extends Exception>`

Kommen wir zu den Klassen, die `CharIterable` implementieren, sodass Nutzer mit der gerade vorgestellten Schleife die Zeichen ablaufen können. Die Deklaration der Schnittstelle `CharIterable<E extends Exception>` enthält eine auf `Exception` eingeschränkte Typvariable, was zum Beispiel die folgenden Implementierungen zulässt:

- ✓ `class StringIterable implements CharIterable<RuntimeException>`
- ✓ `class WebIterable implements CharIterable<IOException>`

Kommen im Fall von `StringIterable` die Zeichen aus einem String, ist keine Ein-/Ausgabe-Ausnahme zu erwarten, daher ist der Typparameter `RuntimeException`. Beim Lesen aus Dateien oder Internetressourcen kann es jedoch zu `IOExceptions` kommen, sodass `WebIterable` den Typparameter `IOException` wählt.

Beispielimplementierungen für den parametrisierten Typ

Implementieren wir die beiden Klassen `StringIterable` und `WebIterable`. Da `StringIterable` bei der Implementierung der Schnittstelle den Typparameter `RuntimeException` wählt, führt das zu einem `throws RuntimeException`, was wiederum optional ist und weggelassen werden kann.

com/tutego/insel/generic/StringIterable.java, StringIterable

```
public class StringIterable implements CharIterable<RuntimeException>
{
    private final String string;
    private int pos;

    public StringIterable( String string )
    {
        this.string = string;
    }

    @Override public boolean hasNext()
    {
        return pos < string.length();
    }

    @Override public char next()
    {
        return string.charAt( pos++ );
    }
}
```

Bei `WebIterable` sieht das anders an. Hier ist der Typparameter `IOException`, und somit ist ein `throws IOException` an der Methodensignatur nötig.

com/tutego/insel/generic/WebIterable.java, WebIterable

```
public class WebIterable implements CharIterable<IOException>
{
    private final Reader reader;

    public WebIterable( String url ) throws IOException
    {
        reader = new InputStreamReader( new URL( url ).openStream() );
    }

    @Override public boolean hasNext() throws IOException
    {
        return reader.ready();
    }

    @Override public char next() throws IOException
    {
        return (char) reader.read();
    }
}
```

Nutzen von `StringIterable` und `WebIterable`

Das folgende Beispiel zeigt, dass beim Ablauen eines Strings keine Ausnahmebehandlung nötig ist, beim Lesen von Zeichen aus dem Internet aber schon:

```
com/tutego/insel/generic/CharReadableExample.java, main()

StringIterable iter1 = new StringIterable( "Shasha" ); // try ist unnötig
while ( iter1.hasNext() )
    System.out.print( iter1.next() );

System.out.println();

try
{
    WebIterable iter2 = new WebIterable( "http://java-tutor.com/aufgaben/bond.txt" );
    while ( iter2.hasNext() )
        System.out.print( iter2.next() );
}
catch ( IOException e )
{
    e.printStackTrace();
}
```

Statt `StringIterable iter1 = new StringIterable...` **hätten wir natürlich auch** `CharIterable<RuntimeException> iter1...` **schreiben können und analog** `CharIterable<IOException> iter2` **statt** `WebIterable iter2.`

Zusammenfassung

Das Beispiel macht deutlich, dass ein Typparameter `RuntimeException` selbst so elementare Dinge wie geprüfte Ausnahmen ausschaltet. Die Besonderheit liegt beim Compiler, dass er Dinge wie `throws E` zulässt und dass `E` dann einmal eine geprüfte oder ungeprüfte Ausnahme sein kann. Exakt so hatten wir `CharIterable` deklariert:

```
public interface CharIterable<E extends Exception>
{
    boolean hasNext() throws E;
    char     next()      throws E;
}
```

Das ist sehr praktisch, denn unser Anwendungsfall macht deutlich, dass es gut ist, einmal geprüfte Ausnahmen zu verwenden, denn geprüfte Aufnahmen verlangen ja immer etwas mehr Aufwand und sind immer nötig, wie das Ablaufen von Strings zeigt.

API-Design der Klasse Scanner

Scanner ist ein Beispiel für eine Klasse, in der die Java-API-Designer geprüfte Ausnahmen bei den `next()`-Methoden nicht haben wollten. Die Klasse kann normale Strings zerlegen, bei denen `next()` keine `IOException` auslösen kann. Aber Scanner kann auch einen Eingabestrom bekommen, und dann sind Ein-/Ausgabeausnahmen durchaus möglich.



Was also tun? Entweder bei den `next()`-Methoden immer eine `IOException` auslösen oder nie? Lösen sie keine Ausnahme aus (das ist das Design jetzt), so bleiben die Fehler auf der Strecke, die beim Einlesen aus dem Datenstrom auftreten können. Trügen die `next()`-Methoden jedoch ein `throws IOException`, dann wäre das lästig beim Zerlegen von puren Strings - und das wollten die Entwickler nicht. Daher fällt die `IOException` bei `next()` unter den Tisch und muss explizit über die Methode `ioException()` erfragt werden. Das steht so ganz im Gegensatz zu der Idee, bei Ein-/Ausgabefehlern immer geprüfte Ausnahmen zu verwenden. Beim `PrintWriter` ist das übrigens genauso, die `write()` und `printXXX()`-Methoden lösen keine `IOException` aus, sondern Entwickler fragen später mit `checkError()` nach, ob es Probleme gab. Leser können überlegen, ob `Scanner<E extends Exception>` und Methoden wie `next() throws E` das Problem lösen würden.

3.5 Generics und Vererbung, Invarianz

Vererbung und Substitution ist für Java-Entwickler alltäglich, sodass diese Eigenschaft nicht weiter verwunderlich ist. Die `toString()`-Methode zum Beispiel wird ganz natürlich auf allen Objekten aufgerufen, und Entwicklern ist klar, dass der Aufruf dynamisch gebunden ist. Genauso lässt sich bei `String.toString(Object o)` jedes Objekt übergeben, und die statische Methode ruft die Objektmethode `toString()` auf.

Arrays sind invariant

Nehmen wir als folgendes Beispiel die Hierarchie der bekannten Wrapper-Klassen. Natürlich steht `Object` oben. Die numerischen Wrapper-Klassen implementieren alle `Number`. Darunter stehen dann etwa `Integer`, `Double` und die anderen numerischen Wrapper. Folgendes bereitet keine Kopfschmerzen:

```
Number number = Integer.valueOf( 10 );
number = Double.valueOf( 1.1 );
```

Einmal zeigt `number` auf ein `Integer`, dann auf ein `Double`-Objekt.

Wie verhält es sich nun mit Arrays? Da ist ein `Number`-Array der Basistyp eines `Double`-Arrays:

```
Number[] numbers = new Double[ 100 ];
numbers[ 0 ] = 1.1;
```

Dass ein Array vom Typ `Double[]` ein Untertyp von `Number[]` ist, nennt sich **Invarianz**. Doch lässt sich das auch auf Generics übertragen?

Generics sind kovariant

Es funktioniert, Folgendes zu schreiben:

```
Set<String> set = new HashSet<String>();
```

Ein `HashSet` mit `Strings` ist eine Art von `Set` mit `Strings`. Aber ein `HashSet` mit `Strings` ist kein `HashSet` mit `Objects`. Damit wäre Folgendes falsch:

```
HashSet<Object> set = new HashSet<String>(); // ☹ Compilerfehler!
```

Generics sind nicht invariant, sie sind **kovariant**. Diese Eigenschaft ist auf den ersten Blick gegen die Intuition, doch ein Beispiel rückt diesen Eindruck schnell gerade. Bleiben wir bei unserem `Pocket` und den Wrapper-Klassen. Auch wenn `Number` die Oberklasse von `Integer` ist, so gilt dennoch nicht, dass `Pocket<Number>` ein Obertyp von `Pocket<Integer>` ist. Wäre es das, wäre Folgendes möglich und zur Laufzeit ein Problem:

```
Pocket<Number> p;
p = new Pocket<Integer>(); // Ist das OK?
p.set( 2.2 );
```

Das Argument `2.2` ist über Autoboxing ein `Double`, und daher scheint es auf `Number` zu passen. Allerdings sollte `Double` aber gar nicht erlaubt sein, da wir mit `Pocket<Integer>` ja eine Tasche für `Integer` aufgebaut haben, und ein `Double` darf nicht in die `Integer`-Tasche. Daher folgt: Die Ableitungsbeziehung zwischen Typen überträgt sich nicht auf generische Klassen. Ein `Pocket<Number>` ist also keine Oberklasse, die alle erdenklichen numerischen Typen in der Tasche erlaubt. Der Compiler meckert bei diesem Versuch sofort:

```
Pocket<Number> p;
p = new Pocket<Integer>(); // ☹ Type mismatch: cannot convert from Pocket<Integer>
                           // to Pocket<Number>
```

Auch durch eine alternative Schreibweise lässt sich der Compiler nicht in die Irre führen:

```
Pocket<Integer> p = new Pocket<Integer>();
Pocket<Number> p2 = p; // ☹ Type mismatch: cannot convert
                       // from Pocket<Integer> to Pocket<Number>
```

Im Fall von immutable Objekten mit Nur-lese-Zugriff bestünde eigentlich kein Grund für Kovarianz. Nehmen wir an, die folgende Deklaration wäre korrekt:

```
Pocket<Number> p = new Pocket<Integer>( 1 );
Number n = p.get();
```

Dann haben wir gezeigt, dass `p.set(2.2)` zum Beispiel nicht in Ordnung ist, da `Double` nicht mit `Integer` kompatibel ist. Aber wenn das Objekt etwa über den Konstruktor initialisiert würde, spräche nichts dagegen, mit einem kleineren Typ, also hier `Number`, daraus zu lesen. Jedoch kann Java nicht erkennen, ob ein Typ immutable ist, und kann daher auch solche Ausnahmen bei den Generics nicht machen. Der Compiler nimmt immer an, Zugriffe wären lesend und schreibend.



Wildcards mit ?

Wir wollen eine Methode `isOnePocketEmpty()` schreiben, die eine variable Anzahl von beliebigen Tascheninhalten bekommt und testet, ob eine davon leer ist. Ein Aufruf könnte so aussehen:

```
Pocket<String> p1 = new Pocket<String>( "Bad-Bank" );
Pocket<Integer> p2 = new Pocket<Integer>( 1500000 );
System.out.println( isOnePocketEmpty( p1, p2 ) ); // false
```

Die erste Idee für den Methodenkopf sieht so aus:

```
public static boolean isOnePocketEmpty( Pocket<Object>... pockets )
```

Doch halt! Da `Pocket<Object>` nicht Taschen mit allen Typen umfasst, sondern nur exakt eine Tasche trifft, die ein `Object`-Objekt enthält, ist das keine sinnvolle Parametrisierung für `isOnePocketEmpty()`. Das hatten wir im oberen Abschnitt schon festgestellt. Denn wäre das möglich, würde es die Typsicherheit gefährden. Denn wenn diese Methode tatsächlich Taschen mit allen Inhalten akzeptieren würde, so könnte einer Tasche leicht ein Wert mit falschem Typ untergeschoben werden. Denn wird wie in unserem Beispiel die Methode `isOnePocketEmpty()` mit einem `Pocket<String>` aufgerufen, so würde wegen `isOnePocketEmpty(Pocket<Object>... pockets)` dann auch der Aufruf von `set(12)` auf dem Pocket gültig sein, und dann stände plötzlich statt des gewünschten Inhalts vom Taschentyp `String` nun ein `Integer` in der Tasche. Das darf nicht gültig sein!

Ist der Typ egal, könnten wir an den Original-Typ (Raw-Type) denken. Doch die Raw-Types haben den Nachteil, dass bei ihnen der Compiler überhaupt nichts prüft, wir aber eine gewisse Prüfung möchten. So soll die Methode `isOnePocketEmpty()` beliebige Taschen entgegennehmen, aber gleichzeitig soll es der Methode auch verboten sein, falsche Dinge in die Taschen zu setzen. Ein `isOnePocketEmpty(Pocket... pockets)` ist also keine gute Idee und führt außerdem zu diversen Warnungen.

Die Lösung besteht im Einsatz des **Wildcard-Typs** `?`. Er repräsentiert dann eine Familie von Typen. Wenn schon `Pocket<Object>` nicht der Basistyp aller Tascheninhalte ist, dann ist es `Pocket<?>`. Es ist wichtig zu verstehen, dass `?` nicht für `Object` steht, sondern für einen unbekannten Typ! Damit lässt sich `isOnePocketEmpty()` realisieren:

```
com/tutego/insell/generic/PocketsEmpty.java

public static boolean isOnePocketEmpty( Pocket<?>... pockets )
{
    for ( Pocket<?> pocket : pockets )
        if ( pocket.isEmpty() )
            return true;

    return false;
}

public static void main( String[] args )
{
    Pocket<String> p1 = new Pocket<String>( "Bad-Bank" );
    Pocket<Integer> p2 = new Pocket<Integer>( 1500000 );
    System.out.println( isOnePocketEmpty( p1, p2 ) ); // false
    System.out.println( isOnePocketEmpty( p1, p2, new Pocket<Byte>() ) ); // true
}
```

Dass der Aufruf von `isOnePocketEmpty()`, also bei keiner übergebenen Tasche, zu `false` führt, soll an dieser Stelle als gegeben gelten.

Wir müssen Wildcards von Typvariablen gedanklich streng trennen. Instanziierungen mit Wildcards sind nicht erlaubt, da eine Wildcard ja eben nicht für einen konkreten Typ, sondern für eine ganze Reihe von möglichen Typen steht. Wildcards können auch nicht wie Typvariablen in Methoden genutzt werden, auch wenn der Typ beliebig ist.

Korrekt mit Typvariable	Falsch mit Wildcard (Compilerfehler!)
<code>Pocket<?> pocket = new Pocket<Byte>();</code>	<code>Pocket<?> pocket = new Pocket<?>();</code>
<code>static <T> T random(T m, T n) { ... }</code>	<code>static ? random(? m, ? n) { ... }</code>

Auswirkungen auf Lese-/Schreiboperationen

Ist der Wildcard-Typ bei `Pocket<?>` im Einsatz, wissen wir nichts über den Typ, und dem Compiler gehen alle Informationen verloren. Deklarieren wir etwa

```
Pocket<?> p1 = new Pocket<Integer>();
```

oder

```
Pocket<Integer> p2 = new Pocket<Integer>();
Pocket<?> p3 = p2;
```

dann ist über die wirklichen Typparameter bei `p1` und `p3` nichts bekannt. Das hat wichtige Auswirkungen auf die Methoden, die wir auf `Pocket` aufrufen können.

- ✓ Ein Aufruf von `p1.get()` ist legal, denn alles, was die Methode liefern wird, ist immer ein `Object`, auch wenn es `null` ist. Die Anweisung `Object v = p1.get();` ist dementsprechend korrekt.
- ✓ Ein `p1.set(value)` kann nicht erlaubt sein, da über den Typ von `value` nichts bekannt ist. In `p1` dürfen wir kein `Double` einsetzen, da `Pocket` nur `Integer` speichern soll. Die einzige Ausnahme ist `null`, da `null` jeden Typ hat. `p1.set(null)` ist also eine zulässige Anweisung. Das heißt ebenso, dass mit `<?>` aufgebaute Objekte nicht automatisch immutable sind.

Bounded Wildcards

Die Angabe des konkreten Typparameters wie `Pocket<Integer>` und die Wildcard-Form `Pocket<?>` bilden Extreme. Die Tasche `Pocket<Integer>` nimmt nur Ganzzahlen auf, `Pocket<?>` auf der anderen Seiten alles. Es muss aber auch etwas dazwischen geben, um zum Beispiel auszudrücken, dass die Tasche nur eine Zahl oder eine Zeichenkette enthalten soll.

Daher sind Typ-Einschränkungen mit `extends` und `super` möglich. Damit ergeben sich drei Arten von Wildcards:

Wildcard	Bezeichnung	Typparameter
<code>?</code>	Wildcard-Typ	ist beliebig
<code>? extends Typ</code>	Upper-bound Wildcard-Typ	muss Typ erweitern
<code>? super Typ</code>	Lower-bound Wildcard-Typ	muss über Typ stehen

Die Wildcard beschreibt also die Eigenschaft eines Typparameters. Wenn es

```
Pocket<? extends Number> p;
```

heißt, dann können in der Tasche `p` alle möglichen `Number`-Objekte sein.

Machen wir `extends` und `super` noch an einem anderen Beispiel deutlich, das zeigt, welche Familie von Typen die Syntax beschreibt:

<code>? extends CharSequence</code>	<code>? super String</code>
<code>CharSequence</code>	<code>String</code>
<code>String</code>	<code>CharSequence</code>

? extends CharSequence	? super String
StringBuffer	Object
StringBuilder	
...	

Einige eingeschlossene Typen bei `extends` und `super`

Die erste Tabellenzeile (nach dem Tabellenkopf) macht deutlich, dass `extends` und `super` den angegebenen Typ selbst mit einschließen. In `<? extends CharSequence>` ist `CharSequence` genau der Upper-Bound der Wildcard, und in `<? super String>` ist `String` der Lower-Bound der Wildcard. Während die Anzahl der Typen beim Lower-Bound beschränkt ist (die Anzahl der Oberklassen kann sich nicht erweitern), ist die Anzahl der Typen mit Upper-Bound im Prinzip unbekannt, da es immer wieder neue Unterklassen geben kann.

Einsatzgebiete

Jeder der drei Wildcard-Typen hat seine Einsatzgebiete. Weitere Anwendungen der **Upper-bound Wildcard** und der **Lower-bound Wildcard** zeigen die Sortiermethoden der Datenstrukturen und Algorithmen.

Beispiel	Bedeutung
<code>Pocket<?> p;</code>	Taschen mit beliebigem Inhalt
<code>Pocket<? extends Number> p;</code>	Taschen nur mit Zahlen
<code>Comparator<? super String> comp = String.CASE_INSENSITIVE_ORDER;</code>	Entweder String-, Object- oder CharSequence-Comparator. Idee: Ein Comparator, der allgemeine Object-Objekte vergleichen kann, kann (irgendwie) auch Strings vergleichen, denn durch die Vererbung ist ein String eine Art von Object.

Beispiel mit Upper-bound-Wildcard-Typ

Die Upper-bound Wildcard ist häufiger zu finden als die Lower-bound-Variante. Daher wollen wir ein Beispiel aufführen, an dem gut der übliche Einsatz für den Upper-bound abzulesen ist. Unser Player hatte eine rechte und eine linke Tasche. Die Taschen sollen aber nun nicht alles Mögliche speichern können, sondern nur besondere Spielobjekte vom Typ Portable (engl. für tragbar). Portable ist eine Schnittstelle, die ein Gewicht für die tragbaren Objekte vorschreibt. Zwei Typen sollen tragbar sein: Pen und Cup. Die Implementierung sieht so aus:

```
com/tutego/insel/generic/PortableDemo.java, Ausschnitt

interface Portable
{
    double getWeight();
    void setWeight( double weight );
}

abstract class AbstractPortable implements Portable
{
    double weight;

    @Override public double getWeight() { return weight; }

    @Override public void setWeight( double weight ) { this.weight = weight; }

    @Override public String toString() { return getClass().getName() +
        "[weight=" + weight + "]"; }
}

class Pen extends AbstractPortable { }

class Cup extends AbstractPortable { }
```

Um zu testen, ob der Spieler nicht zu viele Sachen trägt, soll eine Methode `areLighterThan()` testen, ob das Gewicht einer Liste von tragbaren Dingen unter einer gegebenen Grenze bleibt. Der erste Versuch könnte so aussehen:

```
boolean areLighterThan( List<Portable> collection, double maxWeight )
```

Moment! Das würde wieder ausschließlich Portable-Objekte akzeptieren, denn Kovarianz gilt ja nicht. Selbst wenn es gehen würde, könnte das bedeuten, dass in einer Methode dann vielleicht über `collection.add()` ein Pen hinzugefügt werden kann, auch wenn die übergebene Liste mit Cup deklariert wurde. Dann stände in der Liste plötzlich etwas Falsches. Außerdem ist Portable eine Schnittstelle, sodass die Methode wirklich überhaupt keinen Sinn ergibt. So ist die korrekte Schreibweise nur mit einem Upper-bound-Wildcard-Typ möglich:

```
boolean areLighterThan( List<? extends Portable> list, double maxWeight )
```

Somit nimmt die Methode nur Listen mit Portable-Objekte an, und das ist auch nötig, denn Portable-Objekte haben ein Gewicht, und diese Eigenschaft brauchen wir.

com/tutego/insel/generic/PortableDemo.java, Ausschnitt

```
class PortableUtils
{
    public static boolean areLighterThan( List<? extends Portable> list, double maxWeight )
    {
        double accumulatedWeight = 0.0;

        for ( Portable portable : list )
            accumulatedWeight += portable.getWeight();

        return accumulatedWeight < maxWeight;
    }
}

public class PortableDemo
{
    public static void main( String[] args )
    {
        Pen pen = new Pen();
        pen.setWeight( 10 );
        Cup cup = new Cup();
        cup.setWeight( 100 );
        System.out.println( PortableUtils.areLighterThan( Arrays.asList( pen, cup ),
                                                       10 ) ); //false
        System.out.println( PortableUtils.areLighterThan( Arrays.asList( pen, cup ),
                                                       120 ) ); //true
    }
}
```

Wie schon besprochen wurde, kann aus der mit den Upper-bound-Wildcards deklarierten Datenstruktur `List<? extends Portable>` nur gelesen, aber nicht verändert werden.

Bounded-Wildcard-Typen und Bounded-Typvariablen

Zwischen Bounded-Wildcard-Typen und Bounded-Typvariablen gibt es natürlich einen Zusammenhang, und bei der Deklaration sind zwei Varianten wählbar. Warum das so ist, kann unsere Methode `areLighterThan()` demonstrieren. Statt

```
boolean areLighterThan( List<? extends Portable> list, double maxWeight )
```

hätten wir auch einen formalen Typparameter lokal für die Methode deklarieren können:

```
<T extends Portable> boolean areLighterThan( List<T> list, double maxWeight )
```

Beide Varianten erfüllen den gleichen Zweck. Doch ist die erste Variante der zweiten vorzuziehen.

Immer dann, wenn der formale Typparameter (etwa `T`) nur in der Signatur auftaucht und es in der Methode selbst keinen Rückgriff auf den Typ `T` gibt, wähle die Variante mit der Wildcard.



Mit Typparametern lassen sich gut Abhängigkeiten zwischen den einzelnen Argumenten oder dem Rückgabetypr herstellen. Das zeigt das folgende Beispiel (mit einigen Methoden, die bisher noch nicht vorgestellt wurden), das das leichteste Objekt in einer Sammlung von Taschen zurückgeben soll:

```
com/tutego/insell/generic/PortableDemo.java, PortableUtils

public static <T extends Portable> T lightest( Collection<T> collection )
{
    Iterator<T> iterator = collection.iterator();
    T lightest = iterator.next();

    while ( iterator.hasNext() )
    {
        T next = iterator.next();

        if ( next.getWeight() < lightest.getWeight() )
            lightest = next;
    }

    return lightest;
}
```

Der Compiler achtet darauf, dass der Typ der Rückgabe mit dem Typ der Sammlung übereinstimmt.

Auf Bounded-Wildcard-Typen in Rückgaben verzichten

Wenn es möglich ist, Bounded-Wildcard-Typen oder Bounded-Typvariablen zu nutzen, sind Bounded-Typvariablen immer vorzuziehen. Wildcard-Typen liefern keine Typinformation, und es ist immer besser, sich vom Compiler über die Typ-Inferenz einen genaueren Typ geben zu lassen.

Nehmen wir eine statische Methode `leftSublist()` an, die von einer Liste eine Unterliste zurückgibt. Die Unterliste geht von der ersten Position bis zur Hälfte.

Versuch 1:

```
public static List<?> leftSublist( List<? extends Portable> list )
{
    return list.subList( 0, list.size() / 2 );
}
```

Der Rückgabetypr `List<?>` ist so ziemlich der schlechteste, den wir wählen können, denn der Aufrufer der Methode kann mit der Rückgabe überhaupt nichts anfangen: Er weiß nichts über den Inhalt der Liste.

Versuch 2:

```
public static List<? extends Portable> leftSublist( List<? extends Portable> list )
```

Das ist schon ein wenig besser, denn hier bekommt der Empfänger wenigstens die Information zurück, dass die Liste irgendwelche tragbaren Dinge enthält.

Noch besser ist natürlich, auf die Typ-Inferenz des Compilers zu setzen und dem Aufrufer genau den Typ wieder zurückzugeben, mit dem er den Parametertyp spickte. Dazu müssen wir aber eine Typvariable einsetzen. Der Grund ist: Deklariert eine Methode Parameter oder eine Rückgabe mit mehreren Wildcard-Typen, so sind die wirklichen Typargumente völlig frei wählbar und ohne Zusammenhang.

```
com/tutego/insell/generic/PortableDemo.java, PortableUtils
```

```
public static <T extends Portable> List<T> rightSublist( List<T> list )
{
    return list.subList( 0, list.size() / 2 );
}
```

Nun ist der Typ der Liste, die reinkommt, gleich dem Typ der Liste, die rauskommt. Mit `extends` ist die Liste zwar nur lesbar, aber das liegt in der Natur der Sache.



Insbesondere in der Klasse `Collections` aus der Java-Standard-API könnten viele Methoden auch anders geschrieben werden. Ein Beispiel: Statt `<T extends E> boolean addAll(Collection<T> c)` wählten die Autoren `boolean addAll(Collection<? extends E> c)`.

Das LESS-Prinzip

Während die mit `extends` eingeschränkten Familien Leseoperationen zulassen, gilt für `super` das Gegenteil. Hier ist Lesen nicht erlaubt, aber Schreiben. Als Merkhilfe lässt sich das als LESS-Prinzip festhalten:

Lesen = Extends, Schreiben = Super (LESS)

(Im Englischen ist auch der Ausdruck PECS (**producer-extends, consumer-super**) im Umlauf.)

Ein Beispiel ist auch hier hilfreich. Eine statische Methode `copyLighterThan()` soll nur die Elemente aus einer Liste in eine andere kopieren, die leichter als eine bestimmte Obergrenze sind. Der erste Versuch:

```
public static void copyLighterThan( List<? extends Portable> src,
                                    List<? extends Portable> dest, double maxWeight )
{
    for ( Portable portable : src )
        if ( portable.getWeight() < maxWeight )
            dest.add( portable );           // ❌ Compilerfehler !!
}
```

Auf den ersten Blick sieht es gut aus, aber das Programm lässt sich nicht übersetzen. Das Problem ist die Zeile `dest.add(portable);`. Wir erinnern uns: Mit einer Upper-bound-Wildcard lässt sich nicht schreiben. Das ergibt Sinn, denn die Liste `src` kann ja zum Beispiel eine Liste von `Cup`-Objekten sein und `dest` eine Liste von `Pen`-Objekten. Beide sind `Portable`, aber dennoch inkompatibel, da `Cups` nicht in `Pens` kopiert werden können. Die Frage ist also, wie der Typ der Ergebnisliste aussehen soll. Beginnen wir bei der Quelliste. Hier ist `List<? extends Portable>` schon korrekt, denn die Liste kann ja alles enthalten, was tragbar ist. Doch welche Anforderungen gibt es an die Zielliste? Wie muss der Typ sein, sodass sich alles vom Typ `Portable`, wie `Cup` oder `Pen`, oder sogar noch Unterklassen speichern lassen? Die Antwort ist einfach: Jeder Typ, der über `Portable` liegt! Das sind `Portable` selbst und `Object`, also alle Obertypen. Dies ist aber der Lower-bound-Wildcard-Typ, den wir mit `super` schreiben. Damit folgt:

```
com/tutego/insel/generic/PortableDemo.java, PortableUtils

public static void copyLighterThan( List<? extends Portable> src,
                                    List<? super Portable> dest, double maxWeight )
{
    for ( Portable portable : src )
        if ( portable.getWeight() < maxWeight )
            dest.add( portable );
}
```

Ein Beispiel für den Aufruf:

```
com/tutego/insel/generic/PortableDemo.java, Ausschnitt main()

List<? extends Portable> src = Arrays.asList( pen, cup );
List<? super Portable> dest = new ArrayList<Object>();
PortableUtils.copyLighterThan( src, dest, 20 );
System.out.println( dest.size() ); // 1
Object result = dest.get( 0 );
System.out.println( result );     // com.tutego.insel.generic.Pen[weight=10.0]
```

Die Liste `dest` ist schreibbar, aber der lesbare Typ ist lediglich `Object` - der Compiler weiß nicht, was hier tatsächlich in der Liste steckt, er weiß nur, dass es beliebige Obertypen von `Portable` sein können. Und da bleibt als allgemeinsten Typ eben nur `Object`.

Wildcard-Capture

Das LESS-Prinzip hat eine wichtige Konsequenz, die insbesondere bei Listen-Operationen auffällt. Eine mit einer Wildcard parametrisierte Liste kann nicht verändert werden. Doch wie lässt sich zum Beispiel eine Methode schreiben, die eine Liste umdreht? Vom API-Design her könnte eine Methode `reverse()` wie folgt aussehen:

```
public static void reverse( List<?> list );
```

oder so:

```
public static void <T> reverse( List<T> list );
```

Nach unserem Verständnis, dass wir bei völlig freien Typen die Wildcard-Schreibweise bevorzugen wollen, stehen wir vor einem Dilemma.

```
public static <T> void reverse( List<?> list )
{
    for ( int i = 0; i < list.size() / 2; i++ )
    {
        int j = list.size() - i - 1;
        ? tmp = list.get( i );                      // ☹ Compilerfehler
        list.set( i, list.get( j ) );
        list.set( j, tmp );
    }
}
```

Es bleibt uns nichts anderes, als doch die Variante mit der Typvariablen zu wählen, sodass wir Zugriff auf den Typ `T` haben.

Da nun vom API-Design `reverse(List<?> list)` bevorzugt wird, aber `reverse(List<T> list)` in der Implementierung nötig ist, stellt sich die Frage, ob beides miteinander vereinbar ist. Die gute Nachricht: Ja, denn `reverse(List<?> list)` kann auf eine Umdrehmethode `reverse_(List<T>)` weiterleiten. Zwar müssen die Methoden anders benannt werden, aber wegen des sogenannten **Wildcard-Capture** funktioniert die Abbildung von einer Wildcard auf eine Typvariable.

com/tutego/insell/generic/WildcardCapture, `WildcardCapture`

```
public class WildcardCapture
{
    private static <T> void reverse_( List<T> list )
    {
        for ( int i = 0; i < list.size() / 2; i++ )
        {
            int j = list.size() - i - 1;
            T tmp = list.get( i );
            list.set( i, list.get( j ) );
            list.set( j, tmp );
        }
    }

    public static void reverse( List<?> list )
    {
        reverse_( list );
    }
}
```

Der Compiler "fängt" bei `reverse(list)` den unbekannten Typ der Liste ein und "füllt" die Typvariable bei `reverse_(list)`.

4 Preferences (Voreinstellungen)

In diesem Kapitel erfahren Sie

- ✓ wie Sie Benutzereinstellungen bearbeiten
- ✓ wie Sie Systemeigenschaften nutzen und einstellen

Voraussetzungen

- ✓ Betriebssystem-Kenntnisse
- ✓ Die Klasse `HashMap`

4.1 Benutzereinstellungen

Einstellungen des Benutzers - wie die letzten vier geöffneten Dateien oder die Position eines Fensters - müssen abgespeichert und erfragt werden können. Dafür bietet Java eine Reihe von Möglichkeiten. Sie unterscheiden sich unter anderem in dem Punkt, ob die Daten lokal beim Benutzer oder zentral auf einem Server abgelegt sind.

Im lokalen Fall lassen sich die Einstellungen zum Beispiel in einer Datei speichern. Das Dateiformat kann in Textform oder binär sein. In Textform lassen sich die Informationen etwa in der Form Schlüssel=Wert oder im XML-Format ablegen. Welche Unterstützung Java in diesem Punkt gibt, zeigen die `Properties`-Klasse und die XML-Fähigkeiten der Java-API. Werden Datenstrukturen mit den Benutzereinstellungen serialisiert, kommen in der Regel binäre Dateien heraus. Unter Windows gibt es eine andere Möglichkeit der Speicherung: die Registry. Auch sie ist eine lokale Datei, nur kann das Java-Programm keinen direkten Zugriff auf die Datei vornehmen, sondern muss über Betriebssystemaufrufe Werte einfügen und erfragen.

Sollen die Daten nicht auf dem Benutzerrechner abgelegt werden, sondern zentral auf einem Server, so gibt es auch verschiedene Standards. Die Daten können zum Beispiel über einen Verzeichnisdienst oder Namensdienst verwaltet werden. Bekanntere Dienste sind hier LDAP oder Active Directory. Zum Zugriff auf die Dienste lässt sich das Java Naming and Directory Interface (JNDI) einsetzen. Natürlich können die Daten auch in einer ganz normalen Datenbank stehen, auf die dann die eingebaute JDBC-API Zugriff gewährt. Bei den letzten beiden Formen können die Daten auch lokal vorliegen, denn eine Datenbank oder ein Server, der über JNDI zugänglich ist, kann auch lokal sein. Der Vorteil von nicht-lokalen Servern ist einfach der, dass sich der Benutzer flexibler bewegen kann und immer Zugriff auf seine Daten hat.

Zu guter Letzt lassen sich Einstellungen auch auf der Kommandozeile übergeben. Das lässt die Option `-D` auf der Kommandozeile zu, wenn das Dienstprogramm `java` die JVM startet. Nur lassen sich dann die Daten nicht einfach vom Programm ändern, aber zumindest lassen sich so sehr einfach Daten an das Java-Programm übertragen.

Benutzereinstellungen mit der Preferences-API

Mit der Klasse `java.util.prefs.Preferences` können Konfigurationsdateien gespeichert und abgefragt werden. Für die Benutzereinstellungen stehen zwei Gruppen zur Verfügung: die Benutzerumgebung und die Systemumgebung. Die Benutzerumgebung ist individuell für jeden Benutzer (jeder Benutzer hat andere Dateien zum letzten Mal geöffnet), aber die Systemumgebung ist global für alle Benutzer. Je nach Betriebssystem verwendet die `Preferences`-Implementierung unterschiedliche Speichervarianten und Orte:

- ✓ Unter Windows wird dazu ein Teilbaum der Registry reserviert. Java-Programme bekommen einen Zweig, `SOFTWARE\JavaSoft\Prefs` unter `HKEY_LOCAL_MACHINE` beziehungsweise `HKEY_CURRENT_USER` zugewiesen. Es lässt sich nicht auf die gesamte Registry zugreifen!
- ✓ Unix und Mac OS X speichern die Einstellungen in XML-Dateien. Die Systemeigenschaften landen bei Unix unter `/etc/java/systemPrefs` und die Benutzereigenschaften lokal unter `$HOME/.java/userPrefs`. Mac OS X speichert Benutzereinstellungen im Verzeichnis `/Library/Preferences`.

java.util.prefs.Preferences
+ MAX_KEY_LENGTH: int
+ MAX_NAME_LENGTH: int
+ MAX_VALUE_LENGTH: int
+ <i>absolutePath()</i> : String
+ <i>addNodeChangeListener(nc: NodeChangeListener)</i>
+ <i>addPreferenceChangeListener(pcl: PreferenceChangeListener)</i>
+ <i>childrenNames()</i> : String[]
+ <i>clear()</i>
+ <i>exportNode(os: OutputStream)</i>
+ <i>exportSubtree(os: OutputStream)</i>
+ <i>flush()</i>
+ <i>get(key: String, def: String)</i> : String
+ <i>getBoolean(key: String, def: boolean)</i> : boolean
+ <i>getByteArray(key: String, def: byte[])</i> : byte[]
+ <i>getDouble(key: String, def: double)</i> : double
+ <i>getFloat(key: String, def: float)</i> : float
+ <i>getInt(key: String, def: int)</i> : int
+ <i>getLong(key: String, def: long)</i> : long
+ <i>importPreferences(is: InputStream)</i>
+ <i>isUserNode()</i> : boolean
+ <i>keys()</i> : String[]
+ <i>name()</i> : String
+ <i>node(pathName: String)</i> : Preferences
+ <i>nodeExists(pathName: String)</i> : boolean
+ <i>parent()</i> : Preferences
+ <i>put(key: String, value: String)</i>
+ <i>putBoolean(key: String, value: boolean)</i>
+ <i>putByteArray(key: String, value: byte[])</i>
+ <i>putDouble(key: String, value: double)</i>
+ <i>putFloat(key: String, value: float)</i>
+ <i>putInt(key: String, value: int)</i>
+ <i>putLong(key: String, value: long)</i>
+ <i>remove(key: String)</i>
+ <i>removeNode()</i>
+ <i>removeNodeChangeListener(nc: NodeChangeListener)</i>
+ <i>removePreferenceChangeListener(pcl: PreferenceChangeListener)</i>
+ <i>sync()</i>
+ <i>systemNodeForPackage(c: Class<?>)</i> : Preferences
+ <i>systemRoot()</i> : Preferences
+ <i>toString()</i> : String
+ <i>userNodeForPackage(c: Class<?>)</i> : Preferences
+ <i>userRoot()</i> : Preferences

UML-Diagramm Preferences

Preferences-Objekte lassen sich über statische Methoden auf zwei Arten erlangen:

- ✓ Die erste Möglichkeit nutzt einen absoluten Pfad zum Registry-Knoten. Die Methoden sind am Preferences-Objekt befestigt und heißen für die Benutzerumgebung `userRoot()` und für die Systemumgebung `systemRoot()`.
- ✓ Die zweite Möglichkeit nutzt die Eigenschaft, dass automatisch jede Klasse in eine Paketstruktur eingebunden ist. `userNodeForPackage(Class)` oder `systemNodeForPackage(Class)` liefern ein Preferences-Objekt für eine Verzeichnisstruktur, in der die Klasse selbst liegt.

Beispiel: Erfrage ein Benutzer-Preferences-Objekt über einen absoluten Pfad und über die Paketstruktur der eigenen Klasse:

```
Preferences userPrefs = Preferences.userRoot().node( "/com/tutego/insel" );
Preferences userPrefs = Preferences.userNodeForPackage( this.getClass() );
```

Eine Unterteilung in eine Paketstruktur ist anzuraten, da andernfalls Java-Programme gegenseitig die Einstellung überschreiben könnten; die Registry-Informationen sind für alle sichtbar. Die Einordnung in das Paket der eigenen Klasse ist eine der Möglichkeiten.

abstract class java.util.prefs.Preferences
--

- ✓ static Preferences `userRoot()`
Liefert ein Preferences-Objekt für Einstellungen, die lokal für den Benutzer gelten.
- ✓ static Preferences `systemRoot()`
Liefert ein Preferences-Objekt für Einstellungen, die global für alle Benutzer gelten.

Einträge einfügen, auslesen und löschen

Die Klasse `Preferences` hat große Ähnlichkeit mit den Klassen `Properties` beziehungsweise `HashMap`. Schlüssel/Werte-Paare lassen sich einfügen, löschen und erfragen. Allerdings ist die Klasse `Preferences` kein Mitglied der Collection-API, und es existiert auch keine Implementierung von Collection-Schnittstellen.

```
abstract class java.util.prefs.Preferences
```

- ✓ abstract void put(String key, String value)
- ✓ abstract void putBoolean(String key, boolean value)
- ✓ abstract void putByteArray(String key, byte[] value)
- ✓ abstract void putDouble(String key, double value)
- ✓ abstract void putFloat(String key, float value)
- ✓ abstract void.putInt(String key, int value)
- ✓ abstract void putLong(String key, long value)

Bildet eine Assoziation zwischen den Schlüsselnamen und dem Wert. Die Varianten mit den speziellen Datentypen nehmen intern eine einfache String-Umwandlung vor und sind nur kleine Hilfsmethoden; so steht in `putDouble()` nur `put(key, Double.toString(value))`. Die Hilfsmethode `putByteArray()` konvertiert die Daten nach der Base64-Kodierung und legt sie intern als String ab.

- ✓ abstract String get(String key, String def)
- ✓ abstract boolean getBoolean(String key, boolean def)
- ✓ abstract byte[] getByteArray(String key, byte[] def)
- ✓ abstract double getDouble(String key, double def)
- ✓ abstract float getFloat(String key, float def)
- ✓ abstract int getInt(String key, int def)
- ✓ abstract long getLong(String key, long def)

Liefert den gespeicherten Wert typgerecht aus. Fehlerhafte Konvertierungen werden etwa mit einer `NumberFormatException` bestraft. Der zweite Parameter erlaubt die Angabe eines Alternativwerts, falls es keinen assoziierten Wert zu dem Schlüssel gibt.

- ✓ abstract String[] keys()
- Liefert alle Knoten unter der Wurzel, denen ein Wert zugewiesen wurde. Falls der Knoten keine Eigenschaften hat, liefert `keys()` ein leeres Feld.
- ✓ abstract void flush()

Die Änderungen werden unverzüglich in den persistenten Speicher geschrieben.

Unser folgendes Programm richtet einen neuen Knoten unter `/com/tutego/insel` ein. Aus den über `System.getProperties()` ausgelesenen Systemeigenschaften sollen alle Eigenschaften, die mit "user." beginnen, in die Registry übernommen werden:

```
com/tutego/insel/prefs/PropertiesInRegistry.java, Ausschnitt 1

static Preferences prefs = Preferences.userRoot().node( "/com/tutego/insel" );

static void fillRegistry()
{
    for ( Object o : System.getProperties().keySet() )
    {
        String key = o.toString();

        if ( key.startsWith("user.") && System.getProperty(key).length() != 0 )
            prefs.put( key, System.getProperty(key) );
    }
}
```

Um die Elemente auszulesen, kann ein bestimmtes Element mit `getXXX()` erfragt werden. Die Ausgabe aller Elemente unter einem Knoten gelingt am besten mit `keys()`. Das Auslesen kann eine `BackingStoreException` auslösen, falls der Zugriff auf den Knoten nicht möglich ist. Mit `get()` erfragen wir anschließend den mit dem Schlüssel assoziierten Wert. Wir geben "---" aus, falls der Schlüssel keinen assoziierten Wert besitzt:

```
com/tutego/insel/prefs/PropertiesInRegistry.java, Ausschnitt 2

static void display()
{
    try
    {
        for ( String key : prefs.keys() )
            System.out.println( key + ": " + prefs.get(key, "---") );
    }
    catch ( BackingStoreException e )
    {
        System.err.println( "Knoten können nicht ausgelesen werden: " + e );
    }
}
```

Die Größen der Schlüssel und Werte sind beschränkt! Der Knoten- und Schlüsselname darf maximal `Preferences.MAX_NAME_LENGTH/MAX_KEY_LENGTH` Zeichen umfassen, und die Werte dürfen nicht größer als `MAX_VALUE_LENGTH` sein. Die aktuelle Belegung der Konstanten gibt 80 Zeichen und 8 KiB (8.192 Zeichen) an.



Um Einträge wieder loszuwerden, gibt es drei Methoden: `clear()`, `remove()` und `removeNode()`. Die Namen sprechen für sich.

Auslesen der Daten und Schreiben in einem anderen Format

Die Daten aus den `Preferences` lassen sich mit `exportNode(OutputStream)` beziehungsweise `exportSubtree(OutputStream)` im UTF-8-kodierten XML-Format in einen Ausgabestrom schreiben. `exportNode(OutputStream)` speichert nur einen Knoten, und `exportSubtree(OutputStream)` speichert den Knoten inklusive seiner Kinder. Und auch der umgekehrte Weg funktioniert: `importPreferences(InputStream)` importiert Teile in die Registrierung. Die Schreib- und Lesemethoden lösen eine `IOException` bei Fehlern aus, und eine `InvalidPreferencesFormatException` ist beim Lesen möglich, wenn die XML-Daten ein falsches Format haben.

Auf Ereignisse horchen

Änderungen an den `Preferences` lassen sich mit Listenern verfolgen. Zwei sind im Angebot:

- ✓ Der `NodeChangeListener` reagiert auf Einfüge- und Löschoperationen von Knoten.
- ✓ Der `PreferenceChangeListener` informiert bei Wertänderungen.

Es ist nicht gesagt, dass, wenn andere Applikationen die Einstellungen ändern, diese Änderungen vom Java-Programm auch erkannt werden.

Eine eigene Klasse `NodePreferenceChangeListener` soll die beiden Schnittstellen `NodeChangeListener` und `PreferenceChangeListener` implementieren und auf der Konsole die erkannten Änderungen ausgeben.

```
com/tutego/insel/prefs/ NodePreferenceChangeListener.java, NodePreferenceChangeListener

class NodePreferenceChangeListener implements
    NodeChangeListener, PreferenceChangeListener
{
    /* (non-Javadoc)
     * @see java.util.prefs.NodeChangeListener#childAdded(java.util.prefs.NodeChangeEvent)
     */
    @Override public void childAdded( NodeChangeEvent e )
    {
        Preferences parent = e.getParent(), child = e.getChild();

        System.out.println( parent.name() + " hat neuen Knoten " + child.name() );
    }

    /* (non-Javadoc)
     * @see java.util.prefs.NodeChangeListener#childRemoved
     * (java.util.prefs.NodeChangeEvent)
     */
    @Override public void childRemoved( NodeChangeEvent e )
    {
        Preferences parent = e.getParent(), child = e.getChild();

        System.out.println( parent.name() + " verliert Knoten " + child.name() );
    }

    /* (non-Javadoc)
     * @see java.util.prefs.PreferenceChangeListener#preferenceChange
     * (java.util.prefs.PreferenceChangeEvent)
     */
    @Override public void preferenceChange( PreferenceChangeEvent e )
    {
        String key = e.getKey(), value = e.getNewValue();

        Preferences node = e.getNode();

        System.out.println( node.name() + " hat neuen Wert " + value + " für " + key );
    }
}
```

Zum Anmelden eines Listeners bietet Preferences **zwei addXXXChangeListener()**-Methoden:

com/tutego/insel/prefs/PropertiesInRegistry.java, addListener()

```
NodePreferenceChangeListener listener = new NodePreferenceChangeListener();
prefs.addNodeChangeListener( listener );
prefs.addPreferenceChangeListener( listener );
```

Zugriff auf die gesamte Windows-Registry

Wird Java unter MS Windows ausgeführt, so ergibt sich hin und wieder die Aufgabe, Eigenschaften der Windows-Umgebung zu kontrollieren. Viele Eigenschaften des Windows-Betriebssystems sind in der Registry versteckt, und Java bietet als plattformunabhängige Sprache keine Möglichkeit, diese Eigenschaften in der Registry auszulesen oder zu verändern. (Die Schnittstelle `java.rmi.registry.Registry` ist eine Zentrale für entfernte Aufrufe und hat mit der Windows-Registry nichts zu tun. Auch das Paket `java.util.prefs` mit der Klasse `Preferences` erlaubt nur Modifikationen an einem ausgewählten Teil der Windows-Registry.) Um von Java aus auf alle Teile der Windows-Registry zuzugreifen, gibt es mehrere Möglichkeiten, unter anderem:

- ✓ Um auf allen Werten der Windows-Registry, die dem Benutzer zugänglich sind, operieren zu können, lässt sich mit einem Trick ab Java 1.4 eine Klasse nutzen, die `Preferences` unter Windows realisiert: `java.util.prefs.WindowsPreferences`. Damit ist keine zusätzliche native Implementierung - und damit eine Windows-DLL im Klassenpfad - nötig. Die Bibliothek <https://sourceforge.net/projects/jregistrykey/> realisiert eine solche Lösung.
- ✓ eine native Bibliothek, wie das Windows Registry API Native Interface (<http://tutego.com/go/jnireg>), die frei zu benutzen ist und unter keiner besonderen Lizenz steht
- ✓ das Aufrufen des Konsolenregistrierungsprogramms `reg` zum Setzen und Abfragen von Schlüsselwerten

Registry-Zugriff selbst gebaut

Für einfache Anfragen lässt sich der Registry-Zugriff schnell auch von Hand erledigen. Dazu rufen wir einfach das Kommandozeilenprogramm `reg` auf, um etwa den Dateinamen für den Desktop-Hintergrund anzugeben:

```
$ reg query "HKEY_CURRENT_USER\Control Panel\Desktop" /v Wallpaper
! REG.EXE VERSION 3.0

HKEY_CURRENT_USER\Control Panel\Desktop
    Wallpaper    REG_SZ    C:\Dokumente und Einstellungen\tutego\Anwendungsdaten\Hg.bmp
```

Wenn wir `reg` von Java aufrufen, haben wir den gleichen Effekt:

```
com/tutego/insell/lang/JavaWinReg.java, main()

ProcessBuilder builder = new ProcessBuilder(
    "reg", "query",
    "\\" + HKEY_CURRENT_USER + "\\Control Panel\\Desktop\"", "/v", "Wallpaper" );
Process p = builder.start();
Scanner scanner = new Scanner( p.getInputStream() )
    .useDelimiter( "    \\\\w+\\\\s+\\\\w+\\\\s+" );
scanner.next();
System.out.println( scanner.next() );
```

4.2 Die Utility-Klasse System und Properties

In der Klasse `java.lang.System` finden sich Methoden zum Erfragen und Ändern von Systemvariablen, zum Umlenken der Standard-Datenströme, zum Ermitteln der aktuellen Zeit, zum Beenden der Applikation und noch für das ein oder andere. Alle Methoden sind ausschließlich statisch, und ein Exemplar von `System` lässt sich nicht anlegen. In der Klasse `java.lang.Runtime` - die Schnittstelle `RunTime` aus dem CORBA-Paket hat hiermit nichts zu tun - finden sich zusätzlich Hilfsmethoden, wie etwa das Starten von externen Programmen oder Methoden zum Erfragen des Speicherbedarfs. Anders als `System` ist hier nur eine Methode statisch, nämlich die Singleton-Methode `getRuntime()`, die das Exemplar von `Runtime` liefert.

<code>java.lang.System</code>	<code>java.lang.Runtime</code>
<pre>+ err: PrintStream + in: InputStream + out: PrintStream + arraycopy(src: Object, srcPos: int, dest: Object, destPos: int, length: int) + clearProperty(key: String): String + console(): Console + currentTimeMillis(): long + exit(status: int) + gc() + getProperties(): Properties + getProperty(key: String, def: String): String + getProperty(key: String): String + getSecurityManager(): SecurityManager + getenv(name: String): String + getenv(): Map + identityHashCode(x: Object): int + inheritedChannel(): Channel + load(filename: String) + loadLibrary(libname: String) + mapLibraryName(libname: String): String + nanoTime(): long + runFinalization() + runFinalizersOnExit(value: boolean) + setErr(err: PrintStream) + setIn(in: InputStream) + setOut(out: PrintStream) + setProperties(props: Properties) + setProperty(key: String, value: String): String + setSecurityManager(s: SecurityManager)</pre>	<pre>+ addShutdownHook(hook: Thread) + availableProcessors(): int + exec(cmdarray: String[], envp: String[], dir: File): Process + exec(command: String, envp: String[], dir: File): Process + exec(command: String): Process + exec(cmdarray: String[]): Process + exec(cmdarray: String[], envp: String[]): Process + exec(command: String, envp: String[]): Process + exit(status: int) + freeMemory(): long + gc() + getLocalizedInputStream(in: InputStream): InputStream + getLocalizedOutputStream(out: OutputStream): OutputStream + getRuntime(): Runtime + halt(status: int) + load(filename: String) + loadLibrary(libname: String) + maxMemory(): long + removeShutdownHook(hook: Thread): boolean + runFinalization() + runFinalizersOnExit(value: boolean) + totalMemory(): long + traceInstructions(on: boolean) + traceMethodCalls(on: boolean)</pre>

Eigenschaften der Klassen `System` und `Runtime`



Insgesamt machen die Klassen `System` und `Runtime` keinen besonders aufgeräumten Eindruck; sie wirken irgendwie so, als sei hier alles zu finden, was an anderer Stelle nicht mehr hineingepasst hat. Auch wären Methoden einer Klasse genauso gut in der anderen Klasse aufgehoben.

Dass die statische Methode `System.arraycopy()` zum Kopieren von Feldern nicht in `java.util.Arrays` stationiert ist, lässt sich nur historisch erklären. Und `System.exit()` leitet an `Runtime.getRuntime().exit()` weiter. Einige Methoden sind veraltet beziehungsweise anders verteilt: Das `exec()` von `Runtime` zum Starten von externen Prozessen übernimmt eine neue Klasse `ProcessBuilder`, und die Frage nach dem Speicherzustand oder der Anzahl der Prozessoren beantworten MBeans, wie etwa `ManagementFactory.getOperatingSystemMXBean().getAvailableProcessors()`. Aber API-Design ist wie Sex: Eine unüberlegte Aktion, und es lebt mit uns für immer.

Systemeigenschaften der Java-Umgebung

Die Java-Umgebung verwaltet Systemeigenschaften wie Pfadtrenner oder die Version der virtuellen Maschine in einem `java.util.Properties`-Objekt. Die statische Methode `System.getProperties()` erfragt diese Systemeigenschaften und liefert das gefüllte `Properties`-Objekt zurück. Zum Erfragen einzelner Eigenschaften ist das `Properties`-Objekt aber nicht unbedingt nötig: `System.getProperty()` erfragt direkt eine Eigenschaft.

Beispiel: Gib den Namen des Betriebssystems aus:

```
System.out.println( System.getProperty("os.name") );
Gib alle Systemeigenschaften auf dem Bildschirm aus:
System.getProperties().list( System.out );
```

Die Ausgabe beginnt mit:

```
-- listing properties --
java.runtime.name=Java(TM) SE Runtime Environment
sun.boot.library.path=C:\Program Files\Java\jdk1.7.0\jre\bin
java.vm.version=21.0-b17
java.vm.vendor=Oracle Corporation
java.vendor.url=http://java.oracle.com/
path.separator=;
```

Eine Liste der wichtigen Standard-Systemeigenschaften:

Schlüssel	Bedeutung
java.version	Version der Java-Laufzeitumgebung
java.class.path	Klassenpfad
java.library.path	Pfad für native Bibliotheken
java.io.tmpdir	Pfad für temporäre Dateien
os.name	Name des Betriebssystems
file.separator	Trenner der Pfadsegmente, etwa / (Unix) oder \ (Windows)
path.separator	Trenner bei Pfadangaben, etwa : (Unix) oder ; (Windows)
line.separator	Zeilenumbruchzeichen(-folge)
user.name	Name des angemeldeten Benutzers
user.home	Home-Verzeichnis des Benutzers
user.dir	Aktuelles Verzeichnis des Benutzers

API-Dokumentation

Ein paar weitere Schlüssel zählt die API-Dokumentation bei `System.getProperties()` auf. Einige der Variablen sind auch anders zugänglich, etwa über die Klasse `File`.

```
final class java.lang.System
```

- ✓ `static String getProperty(String key)`
Gibt die Belegung einer Systemeigenschaft zurück. Ist der Schlüssel null oder leer, gibt es eine `NullPointerException` beziehungsweise eine `IllegalArgumentException`.
- ✓ `static String getProperty(String key, String def)`
Gibt die Belegung einer Systemeigenschaft zurück. Ist sie nicht vorhanden, liefert die Methode die Zeichenkette def, den Default-Wert. Für die Ausnahmen gilt das Gleiche wie bei `getProperty(String)`.
- ✓ `static String setProperty(String key, String value)`
Belegt eine Systemeigenschaft neu. Die Rückgabe ist die alte Belegung - oder null, falls es keine alte Belegung gab.
- ✓ `static String clearProperty(String key)`
Löscht eine Systemeigenschaft aus der Liste. Die Rückgabe ist die alte Belegung - oder null, falls es keine alte Belegung gab.
- ✓ `static Properties getProperties()`
Liefert ein mit den aktuellen Systembelegungen gefülltes `Properties`-Objekt.

`line.separator`

Um nach dem Ende einer Zeile an den Anfang der nächsten zu gelangen, wird ein **Zeilenumbruch** (engl. **new line**) eingefügt. Das Zeichen für den Zeilenumbruch muss kein einzelnes sein, es können auch mehrere Zeichen nötig sein. Zum Leidwesen der Programmierer unterscheidet sich die Anzahl der Zeichen für den Zeilenumbruch auf den bekannten Architekturen:

- ✓ Unix: Line Feed (Zeilenvorschub)
- ✓ Windows: beide Zeichen (Carriage Return und Line Feed)
- ✓ Macintosh: Carriage Return (Wagenrücklauf)

Der Steuercode für Carriage Return (kurz CR) ist 13 (0x0D), der für Line Feed (kurz LF) 10 (0x0A). Java vergibt obendrein eigene Escape-Sequenzen für diese Zeichen: `\r` für Carriage Return und `\n` für Line Feed (die Sequenz `\f` für einen Form Feed - Seitenvorschub - spielt bei den Zeilenumbrüchen keine Rolle).

Bei der Ausgabe mit einem `println()` oder der Nutzung des Formatspezifizierers `%n` in `format()` beziehungsweise `printf()` haben wir bei Zeilenumbrüchen keinerlei Probleme. So ist es oft gar nicht nötig, das Zeilenumbruchzeichen vom System über die Property `line.separator` zu erfragen.

Eigene Properties von der Konsole aus setzen

Eigenschaften lassen sich auch beim Programmstart von der Konsole aus setzen. Dies ist praktisch für eine Konfiguration, die beispielsweise das Verhalten des Programms steuert. In der Kommandozeile werden mit `-D` der Name der Eigenschaft und nach einem Gleichheitszeichen (ohne Leerzeichen) ihr Wert angegeben. Das sieht dann etwa so aus:

```
$ java -DLOG=DUSER=Chris -DSIZE=100 com.tutego.insel.lang SetProperty
```

Die Property `LOG` ist einfach nur "da", aber ohne zugewiesenen Wert. Die nächsten beiden Properties, `USER` und `SIZE`, sind mit Werten verbunden, die erst einmal vom Typ String sind und vom Programm weiterverarbeitet werden müssen.

Die Informationen tauchen nicht bei der Argumentliste in der statischen `main()`-Methode auf, da sie vor dem Namen der Klasse stehen und bereits von der Java-Laufzeitumgebung verarbeitet werden.

Um die Eigenschaften auszulesen, nutzen wir das bekannte `System.getProperty()`:

```
com/tutego/insel/lang/SetProperty.java
package com.tutego.insel.lang;

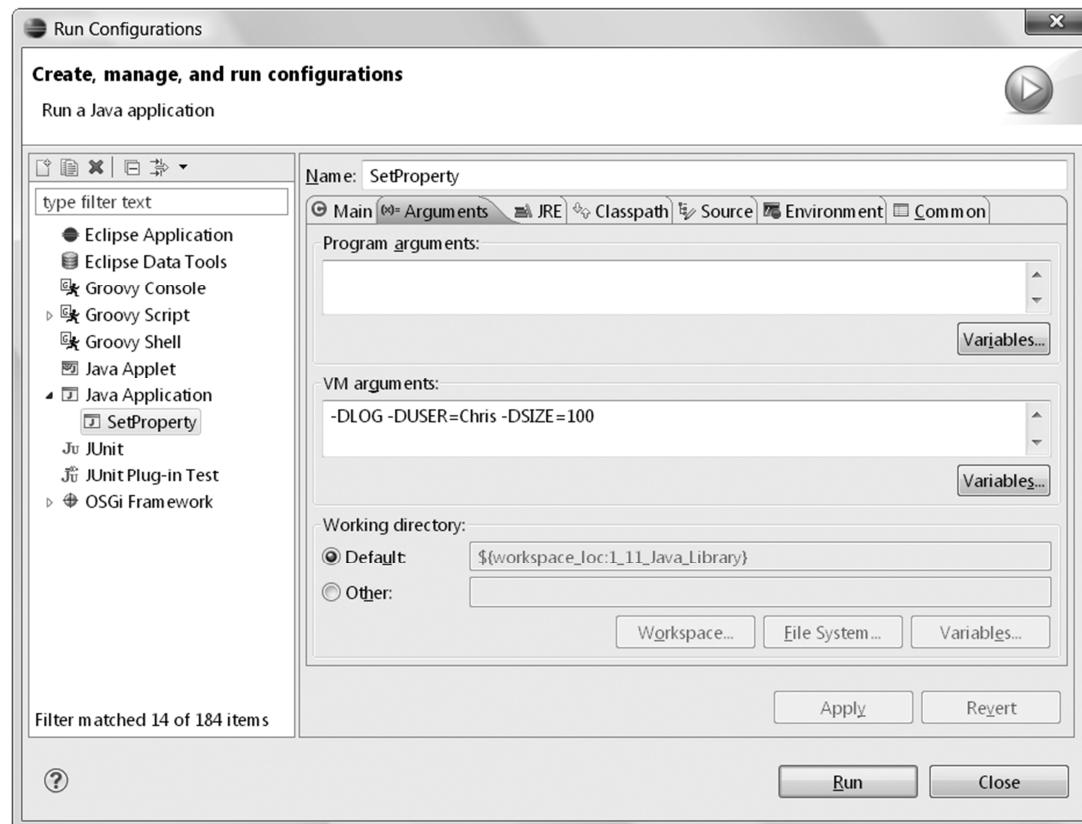
class SetProperty
{
    static public void main( String[] args )
    {
        String logProperty      = System.getProperty( "LOG" );
        String usernameProperty = System.getProperty( "USER" );
        String sizeProperty     = System.getProperty( "SIZE" );

        System.out.println( logProperty != null );                                // true
        System.out.println( usernameProperty );                                     // Chris

        if ( sizeProperty != null )
            System.out.println( Integer.parseInt( sizeProperty ) );                // 100

        System.out.println( System.getProperty( "DEBUG", "false" ) );             // false
    }
}
```

Wir bekommen über `getProperty()` einen String zurück, der den Wert anzeigt. Falls es überhaupt keine Eigenschaft dieses Namens gibt, erhalten wir stattdessen `null`. So wissen wir auch, ob dieser Wert überhaupt gesetzt wurde. Ein einfacher Test wie bei `logProperty != null` sagt also, ob `logProperty` vorhanden ist oder nicht. Statt `-DLOG` führt auch `-DLOG=` zum gleichen Ergebnis, denn der assoziierte Wert ist der Leerstring. Da alle Properties erst einmal vom Typ `String` sind, lässt sich `usernameProperty` einfach ausgeben, und wir bekommen entweder `null` oder den hinter `=` angegebenen String. Sind die Typen keine `Strings`, müssen sie weiterverarbeitet werden, also etwa mit `Integer.parseInt()`, `Double.parseDouble()` usw. Nützlich ist die Methode `System.getProperty()`, der zwei Argumente übergeben werden, denn das zweite steht für einen Default-Wert. So kann immer ein Standardwert angenommen werden.



Entwicklungsumgebungen erlauben es, die Kommandozeilenargumente in einem Fenster zu setzen. Unter Eclipse gehen wir dazu unter RUN - RUN CONFIGURATIONS, dann zu ARGUMENTS.

`Boolean.getBoolean()`

Im Fall von Properties, die mit Wahrheitswerten belegt werden, kann Folgendes geschrieben werden:

```
boolean b = Boolean.parseBoolean( System.getProperty( property ) ); // (*)
```

Für die Wahrheitswerte gibt es eine andere Variante. Die statische Methode `Boolean.getBoolean(name)` sucht aus den System-Properties eine Eigenschaft mit dem angegebenen Namen heraus. Analog zur Zeile (*) ist also:

```
boolean b = Boolean.getBoolean( property );
```

Es ist schon erstaunlich, diese statische Methode in der Wrapper-Klasse `Boolean` anzutreffen, weil Property-Zugriffe nichts mit den Wrapper-Objekten zu tun haben und die Klasse hier eigentlich über ihre Zuständigkeit hinausgeht.

Gegenüber einer eigenen, direkten System-Anfrage hat `getBoolean()` auch den Nachteil, dass wir bei der Rückgabe `false` nicht unterscheiden können, ob es die Eigenschaft schlichtweg nicht gibt oder ob die Eigenschaft mit dem Wert `false` belegt ist. Auch falsch gesetzte Werte wie `-DP=false` ergeben immer `false`. (Das liegt an der Implementierung: `Boolean.valueOf("false")` liefert genauso `false` wie `$_ret.Boolean.valueOf("false")`, `Boolean.valueOf("")` oder `Boolean.valueOf(null)`.)

```
final class java.lang.Boolean
implements Comparable<Boolean>
```

- ✓ static boolean getBoolean(String name)

Liest eine Systemeigenschaft mit dem Namen `name` aus und liefert `true`, wenn der Wert der Property gleich dem String "true" ist. Die Rückgabe ist `false`, wenn entweder der Wert der Systemeigenschaft "false" ist oder er nicht existiert oder `null` ist.

Umgebungsvariablen des Betriebssystems

Fast jedes Betriebssystem nutzt das Konzept der **Umgebungsvariablen** (engl. **environment variables**); bekannt ist etwa `PATH` für den Suchpfad für Applikationen unter Windows und unter Unix. Java macht es möglich, auf diese System-Umgebungsvariablen zuzugreifen. Dazu dienen zwei statische Methoden:

```
final class java.lang.System
```

- ✓ static Map<String, String> getEnv()

Liest eine Menge von `<String, String>`-Paaren mit allen Systemeigenschaften.

- ✓ static String getenv(String name)

Liest eine Systemeigenschaft mit dem Namen `name`. Gibt es sie nicht, ist die Rückgabe `null`.

Beispiel: Was ist der Suchpfad? Den liefert `System.getenv("path")`;

Auswahl einiger unter Windows verfügbarer Umgebungsvariablen

Name der Variablen	Beschreibung	Beispiel
COMPUTERNAME	Name des Computers	MOE
HOMEDRIVE	Laufwerksbuchstabe des Benutzerverzeichnisses	C
HOMEPATH	Pfad des Benutzerverzeichnisses	\Dokumente und Einstellungen\Christian Ullenboom
OS	Name des Betriebssystems	Windows_NT
PATH	Suchpfad	C:\WINDOWS\system32; C:\WINDOWS
PATHEXT	Dateiendungen, die für ausführbare Programme stehen	.COM;.EXE;.BAT;.CMD;.WSH
SYSTEMDRIVE	Laufwerksbuchstabe des Betriebssystems	C
TEMP und auch TMP	Temporäres Verzeichnis	C:\DOKUME~1\CHRIST~1\LOKALE~1\Temp
USERDOMAIN	Domäne des Benutzers	MOE
USERNAME	Name des Nutzers	Christian Ullenboom
USERPROFILE	Profilverzeichnis	C:\Dokumente und Einstellungen\Christian Ullenboom
WINDIR	Verzeichnis des Betriebssystems	C:\WINDOWS

Einige der Variablen sind auch über die `System-Properties` (`System.getProperties()`, `System.getProperty()`) erreichbar.

Beispiel: Gib die Umgebungsvariablen des Systems aus. Um die Ausgabe etwas übersichtlicher zu gestalten, ist bei der Aufzählung jedes Komma durch ein Zeilenvorschubzeichen ersetzt worden:

```
Map<String, String> map = System.getenv();
System.out.println( map.toString().replace(',', '\n') );
```

5 Threads und nebenläufige Programmierung

In diesem Kapitel erfahren Sie

- ✓ was Threads bewirken
- ✓ wie Sie mit Threads arbeiten
- ✓ wozu Dämonen benötigt werden

Voraussetzungen

- ✓ Betriebssystem-Kenntnisse
- ✓ Java-Kenntnisse

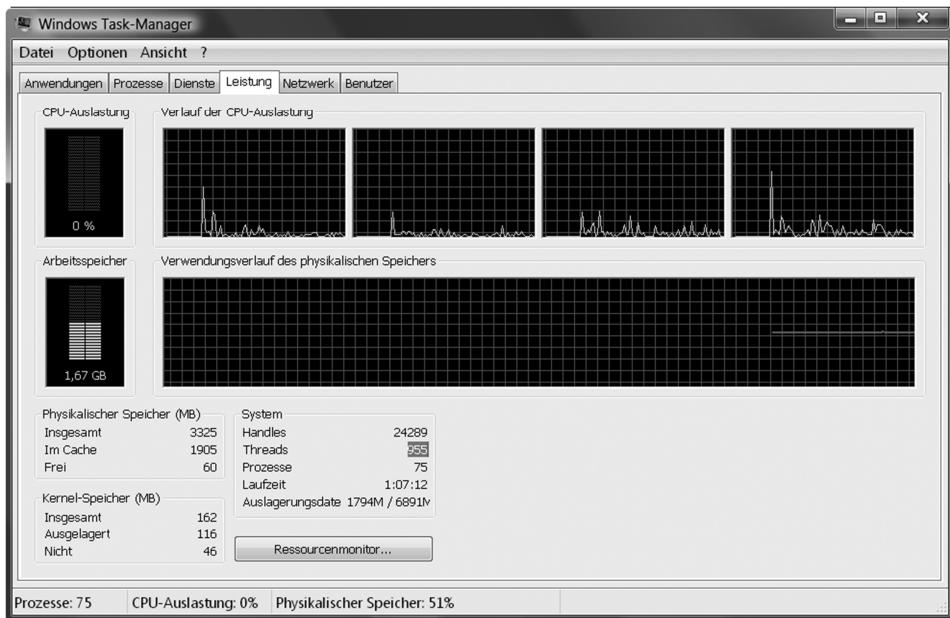
5.1 Nebenläufigkeit

Moderne Betriebssysteme geben dem Benutzer die Illusion, dass verschiedene Programme gleichzeitig ausgeführt werden - die Betriebssysteme nennen sich **multitaskingfähig**. Was wir dann wahrnehmen, ist eine Quasiparallelität, die im Deutschen auch Nebenläufigkeit genannt wird. (Mitunter sind die Begriffe parallel und nebenläufig nicht äquivalent definiert. Wir wollen sie in diesem Zusammenhang aber synonym benutzen.) Diese Nebenläufigkeit der Programme wird durch das Betriebssystem gewährleistet, das auf Einprozessormaschinen die Prozesse alle paar Millisekunden umschaltet. Daher ist das Programm nicht wirklich parallel, sondern das Betriebssystem gaukelt uns dies durch eine verzahnte Bearbeitung der Prozesse vor. Wenn mehrere Prozessoren oder mehrere Prozessor-Kerne am Werke sind, werden die Programmteile tatsächlich parallel abgearbeitet. Aber ob nur ein kleines Männchen oder beliebig viele im Rechner arbeiten, soll uns egal sein.

Der Teil des Betriebssystems, der die Umschaltung übernimmt, heißt **Scheduler**. Die dem Betriebssystem bekannten aktiven Programme bestehen aus Prozessen. Ein Prozess setzt sich aus dem Programmcode und den Daten zusammen und besitzt einen eigenen Adressraum. Des Weiteren gehören Ressourcen wie geöffnete Dateien oder belegte Schnittstellen dazu. Die virtuelle Speicherverwaltung des Betriebssystems trennt die Adressräume der einzelnen Prozesse. Dadurch ist es nicht möglich, dass ein Prozess den Speicherraum eines anderen Prozesses korrumptiert; er sieht den anderen Speicherbereich nicht. Damit Prozesse untereinander Daten austauschen können, wird ein besonderer Speicherbereich als Shared Memory markiert. Amok laufende Programme sind zwar möglich, werden jedoch vom Betriebssystem gestoppt.

Threads und Prozesse

Bei modernen Betriebssystemen gehört zu jedem Prozess mindestens ein **Thread** (zu Deutsch **Faden** oder **Ausführungsstrang**), der den Programmcode ausführt. Damit werden also genau genommen die Prozesse nicht mehr parallel ausgeführt, sondern nur die Threads. Innerhalb eines Prozesses kann es mehrere Threads geben, die alle zusammen in demselben Adressraum ablaufen. Die einzelnen Threads eines Prozesses können untereinander auf ihre öffentlichen Daten zugreifen.



Windows zeigt im Task-Manager die Anzahl laufender Threads an.

Die Programmierung von Threads ist in Java einfach möglich, und die quasi parallel ablaufenden Aktivitäten geben dem Benutzer den Eindruck von Gleichzeitigkeit. In Java ist auch **multithreaded** Software möglich, wenn das Betriebssystem des Rechners keine Threads direkt verwendet. In diesem Fall simuliert die virtuelle Maschine die Parallelität, indem sie die Synchronisation und die verzahnte Ausführung regelt. Unterstützt das Betriebssystem Threads direkt, bildet die JVM die Thread-Verwaltung in der Regel auf das Betriebssystem ab. Dann haben wir es mit **nativen Threads** zu tun. Die 1:1-Abbildung ermöglicht eine einfache Verteilung auf Mehrprozessorsystemen, doch mit dem Nachteil, dass das Betriebssystem in den Threads auch Bibliotheksaufrufe ausführen kann, zum Beispiel, um das Ein- und Ausgabesystem zu verwenden oder für grafische Ausgaben. Damit dies ohne Probleme funktioniert, müssen die Bibliotheken jedoch thread-sicher sein. Damit hatten die Unix-Versionen in der Vergangenheit Probleme: Insbesondere die grafischen Standardbibliotheken *X11* und *Motif* waren lange nicht thread-sicher. Um schwerwiegenden Problemen mit grafischen Oberflächen aus dem Weg zu gehen, haben die Entwickler daher auf eine native Multithreading-Umgebung zunächst verzichtet.

Ob die Laufzeitumgebung native Threads nutzt oder nicht, steht nicht in der Spezifikation der JVM. Auch die Sprachdefinition lässt bewusst die Art der Implementierung frei. Was die Sprache jedoch garantieren kann, ist die korrekt verzahnte Ausführung. Hier können Probleme auftreten, die Datenbankfreunde von Transaktionen her kennen. Es besteht die Gefahr konkurrierender Zugriffe auf gemeinsam genutzte Ressourcen. Um dies zu vermeiden, kann der Programmierer durch synchronisierte Programmblöcke einen gegenseitigen Ausschluss sicherstellen. Damit steigt aber auch die Gefahr von **Verklemmungen** (engl. **deadlocks**), die der Entwickler selbst vermeiden muss.

Wie parallele Programme die Geschwindigkeit steigern können

Auf den ersten Blick ist es nicht ersichtlich, warum auf einem Einprozessorsystem die nebenläufige Abarbeitung eines Programms geschwindigkeitssteigernd sein kann. Betrachten wir daher ein Programm, das eine Folge von Anweisungen ausführt. Die Programmsequenz dient zum Visualisieren eines Datenbank-Reports. Zunächst wird ein Fenster zur Fortschrittsanzeige dargestellt. Anschließend werden die Daten analysiert und der Fortschrittsbalken kontinuierlich aktualisiert. Schließlich werden die Ergebnisse in eine Datei geschrieben. Die Schritte sind:

1. Baue ein Fenster auf.
2. Öffne die Datenbank vom Netz-Server, und lies die Datensätze.
3. Analysiere die Daten, und visualisiere den Fortschritt.
4. Öffne die Datei, und schreibe den erstellten Report.

Was auf den ersten Blick wie ein typisches sequenzielles Programm aussieht, kann durch geschickte Parallelisierung beschleunigt werden.

Damit dies besser zu verstehen ist, ziehen wir noch einmal den Vergleich mit Prozessen. Nehmen wir an, auf einer Einprozessormaschine sind fünf Benutzer angemeldet, die im Editor Quelltext tippen und hin und wieder den Java-Compiler bemühen. Die Benutzer bekämen vermutlich die Belastung des Systems durch die anderen nicht mit, denn Editor-Operationen lasten den Prozessor nicht aus. Wenn Dateien kompiliert und somit vom Hintergrundspeicher in den Hauptspeicher transferiert werden, ist der Prozessor schon besser ausgelastet, doch geschieht dies nicht regelmäßig. Im Idealfall übersetzen alle Benutzer nur dann, wenn die anderen gerade nicht übersetzen - im schlechtesten Fall möchten natürlich alle Benutzer gleichzeitig übersetzen.

Übertragen wir die Verteilung auf unser Problem, nämlich wie der Datenbank-Report schneller zusammengestellt werden kann. Beginnen wir mit der Überlegung, welche Operationen parallel ausgeführt werden können:

- ✓ Das Öffnen des Fensters der Ausgabedatei und das Öffnen der Datenbank kann parallel geschehen.
- ✓ Das Lesen neuer Datensätze und das Analysieren alter Daten kann gleichzeitig erfolgen.
- ✓ Alte analysierte Werte können während der neuen Analyse in die Datei geschrieben werden.

Wenn die Operationen wirklich parallel ausgeführt werden, lässt sich bei Mehrprozessorsystemen ein enormer Leistungszuwachs verzeichnen. Doch interessanterweise ergibt sich dieser auch bei nur einem Prozessor, was in den Aufgaben begründet liegt. Denn bei den gleichzeitig auszuführenden Aufgaben handelt es sich um unterschiedliche Ressourcen. Wenn die grafische Oberfläche des Fenster aufbaut, braucht sie dazu natürlich Rechenzeit. Parallel kann die Datei geöffnet werden, wobei weniger Prozessorleistung gefragt ist, da die vergleichsweise träge Festplatte angesprochen wird. Das Öffnen der Datenbank wird auf den Datenbank-Server im Netzwerk abgewälzt. Die Geschwindigkeit hängt von der Belastung des Servers und des Netzes ab. Wenn anschließend die Daten gelesen werden, muss die Verbindung zum Datenbank-Server natürlich stehen. Daher sollten wir zuerst die Verbindung aufzubauen.

Ist die Verbindung hergestellt, lassen sich über das Netzwerk Daten in einen Puffer holen. Der Prozessor wird nicht belastet, vielmehr der Server auf der Gegenseite und das Netzwerk. Während der Prozessor also vor sich hin döst und sich langweilt, können wir ihn besser beschäftigen, indem er alte Daten analysiert. Wir verwenden hierfür zwei Puffer: In den einen lädt ein Thread die Daten, während ein zweiter Thread die Daten im anderen Puffer analysiert. Dann werden die Rollen der beiden Puffer getauscht. Jetzt ist der Prozessor beschäftigt. Er ist aber vermutlich fertig, bevor die neuen Daten über das Netzwerk eingetroffen sind. In der Zwischenzeit können die Report-Daten in den Report geschrieben werden; eine Aufgabe, die wieder die Festplatte belastet und weniger den Prozessor.

Wir sehen an diesem Beispiel, dass durch hohe Parallelisierung eine Leistungssteigerung möglich ist, da die bei langsamen Operationen anfallenden Wartezeiten genutzt werden können. Langsame Arbeitsschritte lasten den Prozessor nicht aus, und die Wartezeit, die für den Prozessor beim Netzwerzkzugriff auf eine Datenbank anfällt, kann für andere Aktivitäten genutzt werden. Die Tabelle gibt die Elemente zum Kombinieren noch einmal an:

Ressource	Belastung
Hauptspeicherzugriffe	Prozessor
Dateioperationen	Festplatte
Datenbankzugriff	Server, Netzwerkverbindung

Parallelisierbare Ressourcen

Das Beispiel macht auch deutlich, dass die Nebenläufigkeit gut geplant werden muss. Nur wenn verzahnte Aktivitäten unterschiedliche Ressourcen verwenden, resultiert daraus auf Einprozessorsystemen ein Geschwindigkeitsvorteil. Daher ist ein paralleler Sortieralgorithmus nicht sinnvoll. Das zweite Problem ist die zusätzliche Synchronisation, die das Programmieren erschwert. Wir müssen auf das Ergebnis einer Operation warten, damit wir mit der Bearbeitung fortfahren können.

Was Java für Nebenläufigkeit alles bietet

Für nebenläufige Programme sieht die Java-Bibliothek eine Reihe von Klassen, Schnittstellen und Aufzählungen vor:

- ✓ **Thread:** Jeder laufende Thread ist ein Exemplar dieser Klasse.
- ✓ **Runnable:** Beschreibt den Programmcode, den die JVM parallel ausführen soll.
- ✓ **Lock:** Dient zum Markieren von kritischen Abschnitten, in denen sich nur ein Thread befinden darf.
- ✓ **Condition:** Threads können auf die Benachrichtigung anderer Threads warten.

5.2 Threads erzeugen

Die folgenden Abschnitte verdeutlichen, wie der nebenläufige Programmcode in einen `Runnable` verpackt und dem Thread zur Ausführung vorgelegt wird.

Threads über die Schnittstelle `Runnable` implementieren

Damit der Thread weiß, was er ausführen soll, müssen wir ihm Anweisungsfolgen geben. Diese werden in einem Befehlsobjekt vom Typ `Runnable` verpackt und dem Thread übergeben. Wird der Thread gestartet, arbeitet er die Programmzeilen aus dem Befehlsobjekt parallel zum restlichen Programmcode ab. Die Schnittstelle `Runnable` ist schmal und schreibt nur eine `run()`-Methode vor.

```
interface java.lang.Runnable
```

- ✓ void run()

Implementierende Klassen realisieren die Operation und setzen dort den parallel auszuführenden Programmcode ein.



UML-Diagramm
der einfachen
Schnittstelle
`Runnable`

Wir wollen zwei Threads angeben, wobei einer zwanzig mal das aktuelle Datum und die Uhrzeit ausgibt und der andere einfach eine Zahl:

`com/tutego/insel/thread/DateCommand.java`

```
package com.tutego.insel.thread;

public class DateCommand implements Runnable
{
    @Override public void run()
    {
        for ( int i = 0; i < 20; i++ )
            System.out.println( new java.util.Date() );
    }
}
```

`com/tutego/insel/thread/CounterCommand.java`

```
package com.tutego.insel.thread;

class CounterCommand implements Runnable
{
    @Override public void run()
    {
        for ( int i = 0; i < 20; i++ )
            System.out.println( i );
    }
}
```

Unser parallel auszuführender Programmcode in `run()` besteht aus einer Schleife, die in einem Fall ein aktuelles `Date`-Objekt ausgibt und im anderen Fall einen Schleifenzähler.

Thread mit `Runnable` starten

Nun reicht es nicht aus, einfach die `run()`-Methode einer Klasse direkt aufzurufen, denn dann wäre nichts nebenläufig, sondern wir würden einfach eine Methode sequenziell ausführen. Damit der Programmcode parallel zur Applikation läuft, müssen wir ein Thread-Objekt mit dem `Runnable` verbinden und dann den Thread explizit starten. Dazu übergeben wir dem Konstruktor der Klasse `Thread` eine Referenz auf das `Runnable`-Objekt und rufen `start()` auf. Nachdem `start()` für den Thread eine Ablaufumgebung geschaffen hat, ruft es intern selbstständig die Methode `run()` genau einmal auf. Läuft der Thread schon, so löst ein zweiter Aufruf der `start()`-Methode eine `IllegalThreadStateException` aus:

```
com/tutego/insel/thread/FirstThread.java, main()

Thread t1 = new Thread( new DateCommand() );
t1.start();

Thread t2 = new Thread( new CounterCommand() );
t2.start();
```

Beim Starten des Programms erfolgt eine Ausgabe auf dem Bildschirm, die in etwa so aussehen kann:

```
Tue Aug 21 16:59:58 CEST 2007
```

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

```
Tue Aug 21 16:59:58 CEST 2007
```

```
10  
...
```

Deutlich ist die Verzahnung der beiden Threads zu erkennen. Was allerdings auf den ersten Blick etwas merkwürdig wirkt, ist die erste Zeile des Datum-Threads und viele weitere Zeilen des Zähl-Threads. Dies hat jedoch nichts zu bedeuten und zeigt deutlich den Nichtdeterminismus bei Threads. (Nicht vorhersehbar, bedeutet hier: Wann der Scheduler den Kontextwechsel vornimmt, ist unbekannt.) Interpretiert werden kann dies jedoch durch die unterschiedlichen Laufzeiten, die für die Datums- und Zeitausgabe nötig sind.

```
class java.lang.Thread
    implements Runnable
```

- ✓ `Thread(Runnable target)`
Erzeugt einen neuen Thread mit einem `Runnable`, das den parallel auszuführenden Programmcode vorgibt.
- ✓ `void start()`
Ein neuer Thread - neben dem die Methode aufrufenden Thread - wird gestartet. Der neue Thread führt die `run()`-Methode nebenläufig aus. Jeder Thread kann nur einmal gestartet werden.



Wenn ein Thread im Konstruktor einer `Runnable`-Implementierung gestartet wird, sollte die Arbeitsweise bei der Vererbung beachtet werden. Nehmen wir an, eine Klasse leitet von einer anderen Klasse ab, die im Konstruktor einen Thread startet. Bildet die Applikation ein Exemplar der Unterklasse, so werden bei der Bildung des Objekts immer erst die Konstruktoren der Oberklasse aufgerufen. Dies hat zur Folge, dass der Thread schon läuft, auch wenn das Objekt noch nicht ganz gebaut ist. Die Erzeugung ist erst abgeschlossen, wenn nach dem Aufruf der Konstruktoren der Oberklassen der eigene Konstruktor vollständig abgearbeitet wurde.

Die Klasse Thread erweitern

Da die Klasse `Thread` selbst die Schnittstelle `Runnable` implementiert und die `run()`-Methode mit leerem Programmcode bereitstellt, können wir auch `Thread` erweitern, wenn wir eigene parallele Aktivitäten programmieren wollen:

```
com/tutego/insel/thread/DateThread.java, DateThread
```

```
public class DateThread extends Thread
{
    @Override public void run()
    {
        for ( int i = 0; i < 20; i++ )
            System.out.println( new Date() );
    }
}
```

Dann müssen wir kein `Runnable`-Exemplar mehr in den Konstruktor einfügen, denn wenn unsere Klasse eine Unterklasse von `Thread` ist, reicht ein Aufruf der geerbten Methode `start()`. Danach arbeitet das Programm direkt weiter, führt also kurze Zeit später die nächste Anweisung hinter `start()` aus:

`com/tutego/insel/thread/DateThreadUser, main()`

```
Thread t = new DateThread();
t.start();
new DateThread().start(); // (*)
```

Die (*)-Zeile zeigt, dass das Starten sehr kompakt auch ohne Zwischenspeicherung der Objektreferenz möglich ist.

```
class java.lang.Thread
implements Runnable
```

✓ `void run()`
 Diese Methode in `Thread` hat einen leeren Rumpf. Unterklassen überschreiben `run()`, sodass sie den parallel auszuführenden Programmcode enthält.

Überschreiben von `start()` und Selbststarter

Die Methode `start()` kann von uns auch überschrieben werden, was aber nur selten sinnvoll beziehungsweise nötig ist. Wir müssen dann darauf achten, `super.start()` aufzurufen, damit der Thread wirklich startet. Damit wir als Thread-Benutzer nicht erst die `start()`-Methode aufrufen müssen, kann ein Thread sich auch selbst starten. Der Konstruktor ruft dazu einfach die eigene `start()`-Methode auf:

```
class DateThread extends Thread
{
    DateThread()
    {
        start();
    }
    // ... der Rest bleibt ...
}
```

run() wurde statt start() aufgerufen: Ja, wo laufen sie denn?

Ein Programmierfehler, der Anfängern schnell unterläuft, ist folgender: Statt `start()` rufen sie aus Versehen `run()` auf dem Thread auf. Was geschieht? Fast genau das Gleiche wie bei `start()`, nur mit dem Unterschied, dass die Objektmethode `run()` nicht parallel zum übrigen Programm abgearbeitet wird. Der aktuelle Thread bearbeitet die `run()`-Methode sequenziell, bis sie zu Ende ist und die Anweisungen nach dem Aufruf an die Reihe kommen. Der Fehler fällt nicht immer direkt auf, denn die Aktionen in `run()` finden ja statt - nur eben nicht nebenläufig.

Erweitern von Thread oder Implementieren von Runnable?

Die beste Idee wäre, `Runnable`-Objekte zu bauen, die dann dem Thread übergeben werden. Befehlsobjekte dieser Art sind recht flexibel, da die einfachen `Runnable`-Objekte leicht übergeben und sogar von Threads aus einem Thread-Pool ausgeführt werden können. Ein Nachteil der `Thread`-Erweiterung ist, dass die Einachvererbung störend sein kann; erbt eine Klasse von `Thread`, ist die Erweiterung schon "aufgebraucht". Doch, egal ob eine Klasse `Runnable` implementiert oder `Thread` erweitert, eines bleibt: eine neue Klasse.



UML-Diagramm der Klasse `Thread`, die `Runnable` implementiert

5.3 Thread-Eigenschaften und -Zustände

Ein Thread hat eine ganze Reihe von Zuständen, wie einen Namen und eine Priorität, die sich erfragen und setzen lassen. Nicht jede Eigenschaft ist nach dem Start änderbar, doch welche das sind, zeigen die folgenden Abschnitte.

Der Name eines Threads

Ein Thread hat eine ganze Menge Eigenschaften - wie einen Zustand, eine Priorität und auch einen Namen. Dieser kann mit `setName()` gesetzt und mit `getName()` erfragt werden.

```
class java.lang.Thread
    implements Runnable
```

- ✓ `Thread(String name)`
Erzeugt ein neues Thread-Objekt und setzt den Namen. Sinnvoll bei Unterklassen, die den Konstruktor über `super(name)` aufrufen.
- ✓ `Thread(Runnable target, String name)`
Erzeugt ein neues Thread-Objekt mit einem Runnable und setzt den Namen.
- ✓ `final String getName()`
Liefert den Namen des Threads. Der Name wird im Konstruktor angegeben oder mit `setName()` zugewiesen. Standardmäßig ist der Name "Thread-x", wobei x eine eindeutige Nummer ist.
- ✓ `final void setName(String name)`
Ändert den Namen des Threads.

Wer bin ich?

Eine Erweiterung der Klasse `Thread` hat den Vorteil, dass geerbte Methoden wie `getName()` sofort genutzt werden können. Wenn wir `Runnable` implementieren, genießen wir diesen Vorteil nicht.

Die Klasse `Thread` liefert mit der statischen Methode `currentThread()` die Objektreferenz für das Thread-Exemplar, das diese Anweisung gerade ausführt. Auf diese Weise lassen sich nicht-statische `Thread`-Methoden wie `getName()` verwenden.

Beispiel: Gib die aktuelle Priorität des laufenden Threads und den Namen aus:

```
System.out.println( Thread.currentThread().getPriority() ); // z. B. 5
System.out.println( Thread.currentThread().getName() ); // z. B. main
```

Falls es in einer Schleife wiederholten Zugriff auf `Thread.currentThread()` gibt, sollte das Ergebnis zwischengespeichert werden, denn der Aufruf ist nicht ganz billig.

```
class java.lang.Thread
    implements Runnable
```

- ✓ `static Thread currentThread()`
Liefert den Thread, der das laufende Programmstück ausführt.

Schläfer gesucht

Manchmal ist es notwendig, einen Thread eine bestimmte Zeit lang anzuhalten. Dazu lassen sich Methoden zweier Klassen nutzen:

- ✓ **die überladene statische Methode `Thread.sleep()`:** Etwas erstaunlich ist sicherlich, dass sie keine Objektmethode von einem `Thread`-Objekt ist, sondern eine statische Methode. Ein Grund wäre, dass dadurch verhindert wird, externe Threads zu beeinflussen. Es ist nicht möglich, einen fremden Thread, über dessen Referenz wir verfügen, einfach einige Sekunden lang schlafen zu legen und ihn so von der Ausführung abzuhalten.
- ✓ **die Objektmethode `sleep()` auf einem `TimeUnit`-Objekt:** Auch sie bezieht sich immer auf den ausführenden Thread. Der Vorteil gegenüber `sleep()` ist, dass hier die Zeiteinheiten besser sichtbar sind.

Beispiel: Der ausführende Thread soll zwei Sekunden lang schlafen. Einmal mit `Thread.sleep()`:

```
try {
    Thread.sleep( 2000 );
} catch ( InterruptedException e ) { }
Dann mit TimeUnit:
try {
    TimeUnit.SECONDS.sleep( 2 );
} catch ( InterruptedException e ) { }
```

Der Schlaf kann durch eine `InterruptedException` unterbrochen werden, etwa durch `interrupt()`. Die Ausnahme muss behandelt werden, da sie keine `RuntimeException` ist.

Praktisch wird das Erweitern der Klasse `Thread` bei inneren anonymen Klassen. Die folgende Anweisung gibt nach zwei Sekunden Schlafzeit eine Meldung auf dem Bildschirm aus:

`com/tutego/insel/thread/SleepInInnerClass.java, main()`

```
new Thread() {
    @Override public void run() {
        try {
            Thread.sleep( 2000 );
            System.out.println( "Zeit ist um." );
        } catch ( InterruptedException e ) { e.printStackTrace(); }
    }
}.start();
```

Da `new Thread(){...}` ein Exemplar der anonymen Unterklasse ergibt, lässt die auf dem Ausdruck aufgerufene Objektmethode `start()` den Thread gleich loslaufen. Aufgaben dieser Art lösen auch die Timer gut.

```
class java.lang.Thread
implements Runnable
```

- ✓ `static void sleep(long millis) throws InterruptedException`

Der aktuell ausgeführte Thread wird mindestens `millis` Millisekunden schlafen gelegt. Unterbricht ein anderer Thread den schlafenden, wird vorzeitig eine `InterruptedException` ausgelöst.

- ✓ `static void sleep(long millis, int nanos) throws InterruptedException`

Der aktuell ausgeführte Thread wird mindestens `millis` Millisekunden und zusätzlich `nanos` Nanosekunden schlafen gelegt. Im Gegensatz zu `sleep(long)` wird bei einer negativen Millisekundenanzahl eine `IllegalArgumentException` ausgelöst; auch wird diese Exception ausgelöst, wenn die Nanosekundenanzahl nicht zwischen 0 und 999.999 liegt.

```
enum java.util.concurrent.TimeUnit
extends Enum<TimeUnit>
implements Serializable, Comparable<TimeUnit>
```

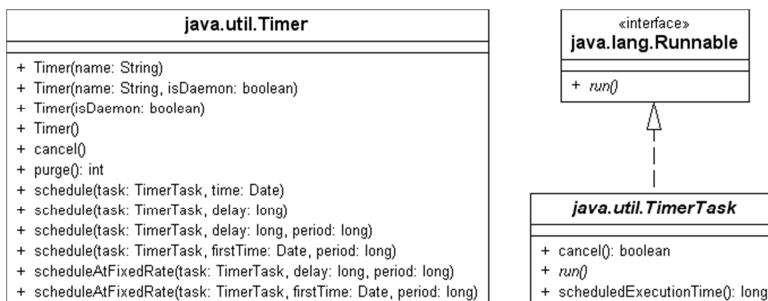
- ✓ `NANOSECONDS, MICROSECONDS, MILLISECONDS, SECONDS, MINUTES, HOURS, DAYS`

Aufzählungselemente von `TimeUnit`.

- ✓ `void sleep(long timeout) throws InterruptedException`

Führt ein `Thread.sleep()` für die Zeiteinheit aus.

Eine überladene Methode `Thread.sleep(long, TimeUnit)` wäre nett, gibt es aber nicht.



UML-Diagramme für `Timer` und `TimerTask`

Mit `yield()` auf Rechenzeit verzichten

Neben `sleep()` gibt es eine weitere Methode, um kooperative Threads zu programmieren: die Methode `yield()`. Sie funktioniert etwas anders als `sleep()`, da hier nicht nach Ablauf der genannten Millisekunden zum Thread zurückgekehrt wird, sondern `yield()` den Thread bezüglich seiner Priorität wieder in die Thread-Warteschlange des Systems einordnet. Einfach ausgedrückt, sagt `yield()` der Thread-Verwaltung: "Ich setze diese Runde aus und mache weiter, wenn ich das nächste Mal dran bin."

```
class java.lang.Thread
    implements Runnable
```

- ✓ static void yield()

Der laufende Thread gibt freiwillig seine Rechenzeit ab. Die Methode ist für Implementierungen der JVM nicht verbindlich.

Der Thread als Dämon

Ein Server reagiert oft in einer Endlosschleife auf eingehende Aufträge vom Netzwerk und führt die gewünschte Aufgabe aus. In unseren bisherigen Programmen haben wir oft Endlosschleifen eingesetzt, sodass ein gestarteter Thread nie beendet wird. Wenn also `run()` wie in den vorangehenden Beispielen nie abbricht (Informatiker sprechen hier von *terminiert*), läuft der Thread immer weiter, auch wenn die Hauptapplikation beendet ist. Dies ist nicht immer beabsichtigt, da vielleicht Server-Funktionalität nach dem Beenden der Applikation nicht mehr gefragt ist. Dann sollte auch der endlos laufende Thread beendet werden. Um dies auszudrücken, erhält ein im Hintergrund arbeitender Thread eine spezielle Kennung: Der Thread wird als **Dämon** gekennzeichnet. (Das griechische δαίμων (engl. daemon) bezeichnet allerlei Wesen zwischen Gott und Teufel. Eine gute Einleitung gibt <http://de.wikipedia.org/wiki/D%C3%A4mon>.) Standardmäßig ist ein aufgebauter Thread kein Dämon.

Ein Dämon ist wie ein Heinzelmännchen im Hintergrund mit einer Aufgabe beschäftigt. Wenn das Hauptprogramm beendet ist und die Laufzeitumgebung erkennt, dass kein normaler Thread mehr läuft, sondern nur Dämonen, dann ist das Ende der Dämonen eingeläutet, und die JVM kommt zum Ende. Denn Dämonen-Threads sind Zulieferer: Gibt es keine Klienten mehr, werden auch sie nicht mehr gebraucht. Das ist wie bei den Göttern der Scheibenwelt: Glaubt keiner an sie, hören sie auf zu existieren. Wir müssen uns also um das Ende des Dämons nicht kümmern. Gleichzeitig heißt das aber auch, dass ein Dämonen-Thread vorsichtig mit Ein-/Ausgabeoperationen sein muss, denn er kann jederzeit - auch während einer Schreiboperation auf die Festplatte - abgebrochen werden, was zu beschädigten Daten führen kann.



Der Garbage-Collector (GC) ist ein gutes Beispiel für einen Dämon. Nur, wenn es andere Threads gibt, muss der Speicher aufgeräumt werden. Gibt es keine anderen Threads mehr, kann auch die JVM mit beendet werden, was auch die Dämonen-Threads beendet.

Wie ein Thread in Java zum Dämon wird

Einen Thread in Java als Dämon zu kennzeichnen, heißt, die Methode `setDaemon()` mit dem Argument `true` aufzurufen. Die Methode ist nur vor dem Starten des Threads erlaubt. Danach kann der Status nicht wieder vom Dämon in den normalen Benutzer-Thread umgesetzt werden. Die Auswirkungen von `setDaemon(true)` können wir am folgenden Programm ablesen:

```
com/tutego/insel/thread/DaemonThread.java
package com.tutego.insel.thread;
class DaemonThread extends Thread
{
    DaemonThread()
    {
        setDaemon( true );
    }
    @Override
    public void run()
    {
        while ( true )
            System.out.println( "Lauf, Thread, lauf" );
    }
}
```

```

    }
    public static void main( String[] args )
    {
        new DaemonThread().start();
    }
}

```

In diesem Programm wird der Thread gestartet, und danach ist die Anwendung sofort beendet. Vor dem Ende kann der neue Thread aber schon einige Zeilen auf der Konsole ausgeben. Klammern wir die Anweisung mit `setDaemon(true)` aus, läuft das Programm ewig, da die Laufzeitumgebung auf das natürliche Ende der Thread-Aktivität wartet.

```

class java.lang.Thread
implements Runnable

```

✓ `final void setDaemon(boolean on)`

Markiert den Thread als Dämon oder normalen Thread. Die Methode muss aufgerufen werden, bevor der Thread gestartet wurde, andernfalls folgt eine `IllegalThreadStateException`. Mit anderen Worten: Nachträglich kann ein existierender Thread nicht mehr zu einem Dämon gemacht werden, und ihm kann auch nicht die Dämonenhaftigkeit genommen werden, so er sie hat.

✓ `final boolean isDaemon()`

Testet, ob der Thread ein Dämon-Thread ist.

Das Ende eines Threads

Es gibt Threads, die dauernd laufen, weil sie zum Beispiel Serverfunktionen implementieren. Andere Threads führen einmalig eine Operation aus und sind danach beendet. Allgemein ist ein Thread beendet, wenn eine der folgenden Bedingungen zutrifft:

- ✓ Die `run()`-Methode wurde ohne Fehler beendet. Wenn wir eine Endlosschleife programmieren, würde diese potenziell einen nie endenden Thread bilden.
- ✓ In der `run()`-Methode tritt eine `RuntimeException` auf, die die Methode beendet. Das beendet weder die anderen Threads noch die JVM als Ganzes.
- ✓ Der Thread wurde von außen abgebrochen. Dazu dient die prinzipbedingt problematische Methode `stop()`, von deren Verwendung abgeraten wird und die auch veraltet ist.
- ✓ Die virtuelle Maschine wird beendet und nimmt alle Threads mit ins Grab.

Wenn der Thread einen Fehler melden soll

Da ein Thread nebenläufig arbeitet, kann die `run()`-Methode synchron schlecht Exceptions melden oder einen Rückgabewert liefern. Wer sollte auch an welcher Stelle darauf hören? Eine Lösung für das Problem ist ein Listener, der sich beim Thread anmeldet und darüber informiert wird, ob der Thread seine Arbeit machen konnte oder nicht. Eine andere Lösung gibt `Callable`, mit dem ein spezieller Fehlercode zurückgegeben oder eine Exception angezeigt werden kann. Speziell für ungeprüfte Ausnahmen kann ein `UncaughtExceptionHandler` weiterhelfen.

Einen Thread höflich mit Interrupt beenden

Der Thread ist in der Regel zu Ende, wenn die `run()`-Methode ordentlich bis zum Ende ausgeführt wurde. Enthält eine `run()`-Methode jedoch eine Endlosschleife - wie etwa bei einem Server, der auf eingehende Anfragen wartet -, so muss der Thread von außen zur Kapitulation gezwungen werden.

Wenn wir den Thread schon nicht von außen beenden wollen, können wir ihn immerhin bitten, seine Arbeit aufzugeben. Periodisch müsste er dann nur überprüfen, ob jemand von außen den Abbruchswunsch geäußert hat.

Die Methoden `interrupt()` und `isInterrupted()`

Die Methode `interrupt()` setzt von außen in einem Thread-Objekt ein internes Flag, das dann in der `run()`-Methode durch `isInterrupted()` periodisch abgefragt werden kann.

Das folgende Programm soll jede halbe Sekunde eine Meldung auf dem Bildschirm ausgeben. Nach zwei Sekunden wird der Unterbrechungswunsch mit `interrupt()` gemeldet. Auf dieses Signal achtet die sonst unendlich laufende Schleife und bricht ab:

```
com/tutego/insel/thread/ThreadusInterruptus.java, main()

Thread t = new Thread()
{
    @Override
    public void run()
    {
        System.out.println( "Es gibt ein Leben vor dem Tod.  " );

        while ( ! isInterrupted() )
        {
            System.out.println( "Und er läuft und er läuft und er läuft" );

            try
            {
                Thread.sleep( 500 );
            }
            catch ( InterruptedException e )
            {
                interrupt();
                System.out.println( "Unterbrechung in sleep()" );
            }
        }

        System.out.println( "Das Ende" );
    }
};

t.start();
Thread.sleep( 2000 );
t.interrupt();
```

Die Ausgabe zeigt hübsch die Ablaufsequenz:

```
Es gibt ein Leben vor dem Tod.
Und er läuft und er läuft und er läuft
Und er läuft und er läuft und er läuft
Und er läuft und er läuft und er läuft
Und er läuft und er läuft und er läuft
Unterbrechung in sleep()
Das Ende
```

Die `run()`-Methode im Thread ist so implementiert, dass die Schleife genau dann verlassen wird, wenn `isInterrupted()` den Wert `true` ergibt, also von außen die `interrupt()`-Methode für dieses Thread-Exemplar aufgerufen wurde. Genau dies geschieht in der `main()`-Methode. Auf den ersten Blick ist das Programm leicht verständlich, doch vermutlich erzeugt das `interrupt()` im `catch`-Block die Aufmerksamkeit. Stünde diese Zeile dort nicht, würde das Programm aller Wahrscheinlichkeit nach nicht funktionieren. Das Geheimnis ist folgendes: Wenn die Ausgabe nur jede halbe Sekunde stattfindet, befindet sich der Thread fast die gesamte Zeit über in der Schlafmethode `sleep()`. Also wird vermutlich der `interrupt()` den Thread gerade beim Schlafen stören. Genau dann wird `sleep()` durch `InterruptedException` unterbrochen, und der `catch`-Behandler fängt die Ausnahme ein. Jetzt passiert aber etwas Unerwartetes: Durch die Unterbrechung wird das interne Flag zurückgesetzt, sodass `isInterrupted()` meint, die Unterbrechung habe gar nicht stattgefunden. Daher muss `interrupt()` erneut aufgerufen werden, da das Abbruch-Flag neu gesetzt werden muss und `isInterrupted()` das Ende bestimmen kann.

Wenn wir mit der Objektmethode `isInterrupted()` arbeiten, müssen wir beachten, dass neben `sleep()` auch die Methoden `join()` und `wait()` durch die `InterruptedException` das Flag löschen.

Die Methoden `sleep()`, `wait()` und `join()` lösen alle eine `InterruptedException` aus, wenn sie durch die Methode `interrupt()` unterbrochen werden. Das heißt, `interrupt()` beendet diese Methoden mit der Ausnahme.



Zusammenfassung: `interrupted()`, `isInterrupted()` und `interrupt()`

Die Methodennamen sind verwirrend gewählt, sodass wir die Aufgaben noch einmal zusammenfassen wollen: Die Objektmethode `interrupt()` setzt in einem (anderen) Thread-Objekt ein Flag, dass es einen Antrag gab, den Thread zu beenden. Sie beendet aber den Thread nicht, obwohl es der Methodenname nahelegt. Dieses Flag lässt sich mit der Objektmethode `isInterrupted()` abfragen. In der Regel wird dies innerhalb einer Schleife geschehen, die darüber bestimmt, ob die Aktivität des Threads fortgesetzt werden soll. Die statische Methode `interrupted()` ist zwar auch eine Anfragemethode und testet das entsprechende Flag des aktuell laufenden Threads, wie `Thread.currentThread().isInterrupted()`, aber zusätzlich löscht es den Interrupt-Status auch, was `isInterrupted()` nicht tut. Zwei aufeinanderfolgende Aufrufe von `interrupted()` führen daher zu einem `false`, es sei denn, in der Zwischenzeit erfolgt eine weitere Unterbrechung.

UncaughtExceptionHandler für unbehandelte Ausnahmen

Einer der Gründe für das Ende eines Threads ist eine unbehandelte Ausnahme, etwa von einer nicht aufgefangenen `RuntimeException`. Um in diesem Fall einen kontrollierten Abgang zu ermöglichen, lässt sich an den Thread ein `UncaughtExceptionHandler` hängen, der immer dann benachrichtigt wird, wenn der Thread wegen einer nicht behandelten Ausnahme endet.

`UncaughtExceptionHandler` ist eine in `Thread` deklarierte innere Schnittstelle, die eine Operation `void uncaughtException(Thread t, Throwable e)` vorschreibt. Eine Implementierung der Schnittstelle lässt sich entweder einem individuellen Thread oder allen Threads anhängen, sodass im Fall des Abbruchs durch unbehandelte Ausnahmen die JVM die Methode `uncaughtException()` aufruft. Auf diese Weise kann die Applikation im letzten Atemzug noch den Fehler loggen, den die JVM über das `Throwable e` übergibt.

```
class java.lang.Thread
    implements Runnable
```

- ✓ `void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)`
Setzt den `UncaughtExceptionHandler` für den Thread.
- ✓ `Thread.UncaughtExceptionHandler getUncaughtExceptionHandler()`
Liefert den aktuellen `UncaughtExceptionHandler`.
- ✓ `Static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)`
Setzt den `UncaughtExceptionHandler` für alle Threads.
- ✓ `static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler()`
Liefert den zugewiesenen `UncaughtExceptionHandler` aller Threads.

Ein mit `setUncaughtExceptionHandler()` lokal gesetzter `UncaughtExceptionHandler` überschreibt den Eintrag für den `setDefaultUncaughtExceptionHandler()`. Zwischen dem mit dem Thread assoziierten Handler und dem globalen gibt es noch einen Handler-Typ für Thread-Gruppen, der jedoch seltener verwendet wird.

5.4 Der Ausführer (Executor) kommt

Zur parallelen Ausführung eines `Runnable` ist immer ein Thread notwendig. Obwohl die nebenläufige Abarbeitung von Programmcode ohne Threads nicht möglich ist, sind doch beide sehr stark verbunden, und es wäre gut, wenn das `Runnable` von dem tatsächlich abarbeitenden Thread etwas getrennt wäre. Das hat mehrere Gründe:

- ✓ Schon beim Erzeugen eines Thread-Objekts muss das `Runnable`-Objekt im Thread-Konstruktor übergeben werden. Es ist nicht möglich, das Thread-Objekt aufzubauen, dann über eine JavaBean-Setter-Methode das `Runnable`-Objekt zuzuweisen und anschließend den Thread mit `start()` zu starten.

- ✓ Wird `start()` auf dem `Thread`-Objekt zweimal aufgerufen, so führt der zweite Aufruf zu einer Ausnahme. Ein erzeugter Thread kann also ein `Runnable` durch zweimaliges Aufrufen von `start()` nicht gleich zweimal abarbeiten. Für eine erneute Abarbeitung eines `Runnable` ist also mit unseren bisherigen Mitteln immer ein neues `Thread`-Objekt nötig.
- ✓ Der Thread beginnt mit der Abarbeitung des Programmcodes vom `Runnable` sofort nach dem Aufruf von `start()`. Die Implementierung vom `Runnable` selbst müsste geändert werden, wenn der Programmcode nicht sofort, sondern später (nächste Tagesschau) oder wiederholt (immer Weihnachten) ausgeführt werden soll.

Wünschenswert ist eine Abstraktion, die das Ausführen des `Runnable`-Programmcodes von der technischen Realisierung (etwa den Threads) trennt.

Die Schnittstelle Executor

Seit Java 5 gibt es eine Abstraktion für Klassen, die Befehle über `Runnable` ausführen. Die Schnittstelle `Executor` schreibt eine Methode vor:

```
interface java.util.concurrent.Executor
```

- ✓ `void execute(Runnable command)`

Wird später von Klassen implementiert, die ein `Runnable` abarbeiten können.

Jeder, der nun Befehle über `Runnable` abarbeitet, ist `Executor`.

Konkrete Executoren

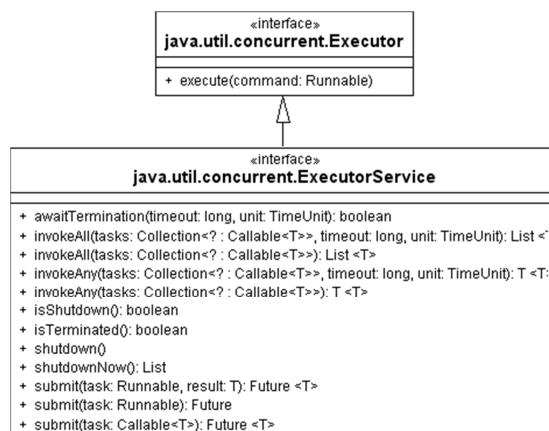
Von dieser Schnittstelle gibt es bisher zwei wichtige Implementierungen:

- ✓ `ThreadPoolExecutor`: Die Klasse baut eine Sammlung von Threads auf, den **Thread-Pool**. Ausführungsanfragen werden von den freien Threads übernommen.
- ✓ `ScheduledThreadPoolExecutor`: Eine Erweiterung von `ThreadPoolExecutor` um die Fähigkeit, zu bestimmten Zeiten oder mit bestimmten Wiederholungen Befehle abzuarbeiten.

Die beiden Klassen haben nicht ganz so triviale Konstruktoren, und eine Utility-Klasse vereinfacht den Aufbau dieser speziellen `Executor`-Objekte.

```
class java.util.concurrent.Executors
```

- ✓ `static ExecutorService newCachedThreadPool()`
Liefert einen Thread-Pool mit wachsender Größe.
- ✓ `static ExecutorService newFixedThreadPool(int nThreads)`
Liefert einen Thread-Pool mit maximal `nThreads`.
- ✓ `static ScheduledExecutorService newSingleThreadScheduledExecutor()`
- ✓ `static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)`
Gibt spezielle `Executor`-Objekte zurück, um Wiederholungen festzulegen.



Die Schnittstelle `ExecutorService`, die `Executor` erweitert

`ExecutorService` ist eine Schnittstelle, die `Executor` erweitert. Unter anderem sind hier Operationen zu finden, die die Ausführer herunterfahren. Im Falle von Thread-Pools ist das nützlich, da die Threads ja sonst nicht beendet würden, weil sie auf neue Aufgaben warten.

java.util.concurrent.Executors
+ newFixedThreadPool(nThreads: int): ExecutorService
+ newFixedThreadPool(nThreads: int, threadFactory: ThreadFactory): ExecutorService
+ newSingleThreadExecutor(): ExecutorService
+ newSingleThreadExecutor(threadFactory: ThreadFactory): ExecutorService
+ newCachedThreadPool(): ExecutorService
+ newCachedThreadPool(threadFactory: ThreadFactory): ExecutorService
+ newSingleThreadScheduledExecutor(): ScheduledExecutorService
+ newSingleThreadScheduledExecutor(threadFactory: ThreadFactory): ScheduledExecutorService
+ newScheduledThreadPool(corePoolSize: int): ScheduledExecutorService
+ newScheduledThreadPool(corePoolSize: int, threadFactory: ThreadFactory): ScheduledExecutorService
+ unconfigurableExecutorService(executor: ExecutorService): ExecutorService
+ unconfigurableScheduledExecutorService(executor: ScheduledExecutorService): ScheduledExecutorService
+ defaultThreadFactory(): ThreadFactory
+ privilegedThreadFactory(): ThreadFactory
+ callable(task: Runnable, result: T): Callable <T>
+ callable(task: Runnable): Callable
+ callable(action: PrivilegedAction<?>): Callable
+ callable(action: PrivilegedExceptionAction<?>): Callable
+ privilegedCallable(callable: Callable<T>): Callable <T>
+ privilegedCallableUsingCurrentClassLoader(callable: Callable<T>): Callable <T>

Executors-Klasse mit statischen Methoden

Die Thread-Pools

Eine wichtige statische Methode der Klasse `Executors` ist `newCachedThreadPool()`. Das Ergebnis ist ein `ExecutorService`-Objekt, eine Implementierung von `Executor` mit der Methode `execute(Runnable)`:

com/tutego/insel/thread/concurrent/ThreadPoolDemo.java, main() - 1

```
Runnable r1 = new Runnable() {
    @Override public void run() {
        System.out.println( "A1 " + Thread.currentThread() );
        System.out.println( "A2 " + Thread.currentThread() );
    }
};

Runnable r2 = new Runnable() {
    @Override public void run() {
        System.out.println( "B1 " + Thread.currentThread() );
        System.out.println( "B2 " + Thread.currentThread() );
    }
};
```

Jetzt lässt sich der Thread-Pool als `ExecutorService` beziehen und lassen sich die beiden Befehlsobjekte als `Runnable` über `execute()` ausführen:

com/tutego/insel/thread/concurrent/ThreadPoolDemo.java, main() - 2

```
ExecutorService executor = Executors.newCachedThreadPool();

executor.execute( r1 );
executor.execute( r2 );

Thread.sleep( 500 );

executor.execute( r1 );
executor.execute( r2 );

executor.shutdown();
```

Die Ausgabe zeigt sehr schön die Wiederverwendung der Threads:

```
A1 Thread[pool-1-thread-1,5,main]
A2 Thread[pool-1-thread-1,5,main]
B1 Thread[pool-1-thread-2,5,main]
B2 Thread[pool-1-thread-2,5,main]
B1 Thread[pool-1-thread-1,5,main]
B2 Thread[pool-1-thread-1,5,main]
A1 Thread[pool-1-thread-2,5,main]
A2 Thread[pool-1-thread-2,5,main]
```

Die `toString()`-Methode von `Thread` ist so implementiert, dass zunächst der Name der Threads auftaucht, den die Pool-Implementierung gesetzt hat, dann die Priorität und der Name des Threads, der den neuen Thread gestartet hat. Am neuen Namen ist abzulesen, dass hier zwei Threads von einem Thread-Pool 1 verwendet werden: `thread-1` und `thread-2`. Nach dem Ausführen der beiden Aufträge und der kleinen Wartezeit sind die Threads fertig und für neue Jobs bereit, sodass `A1` und `A2` beim zweiten Mal mit den wieder freien Threads abgearbeitet werden.

Interessant sind die folgenden drei Operationen zur Steuerung des Pool-Endes:

```
interface java.util.concurrent.ExecutorService
extends Executor
```

- ✓ `void shutdown()`
Fährt den Thread-Pool herunter. Laufende Threads werden nicht abgebrochen, aber neue Anfragen werden nicht angenommen.
- ✓ `boolean isShutdown()`
Wurde der Executor schon heruntergefahren?
- ✓ `List<Runnable> shutdownNow()`
Gerade ausführende Befehle werden zum Stoppen angeregt. Die Rückgabe ist eine Liste der zu beendenden Kommandos.

5.5 Synchronisation über kritische Abschnitte

Wenn Threads in Java ein eigenständiges Leben führen, ist dieser Lebensstil nicht immer unproblematisch für andere Threads, insbesondere beim Zugriff auf gemeinsam genutzte Ressourcen. In den folgenden Abschnitten erfahren wir mehr über gemeinsam genutzte Daten und Schutzmaßnahmen beim konkurrierenden Zugriff durch mehrere Threads.

Gemeinsam genutzte Daten

Ein Thread besitzt zum einen seine eigenen Variablen, etwa die Objektvariablen, kann aber auch statische Variablen nutzen, wie das folgende Beispiel zeigt:

```
class T extends Thread
{
    static int result;

    public void run() { ... }
}
```

In diesem Fall können verschiedene Exemplare der Klasse `T`, die jeweils einen Thread bilden, Daten austauschen, indem sie die Informationen in `result` ablegen oder daraus entnehmen. Threads können aber auch an einer zentralen Stelle eine Datenstruktur erfragen und dort Informationen entnehmen oder Zugriff auf gemeinsame Objekte über eine Referenz bekommen. Es gibt also viele Möglichkeiten, wie Threads - und damit potenziell parallel ablaufende Aktivitäten - Daten austauschen können.

Probleme beim gemeinsamen Zugriff und kritische Abschnitte

Da Threads ihre eigenen Daten verwalten - sie haben alle eigene lokale Variablen und einen Stack -, kommen sie sich gegenseitig nicht in die Quere. Auch wenn mehrere Threads gemeinsame Daten nur lesen, ist das unbedenklich; Schreiboperationen sind jedoch kritisch. Wenn sich zehn Nutzer einen Drucker teilen, der die Ausdrucke nicht als unteilbare Einheit bündelt, lässt sich leicht ausmalen, wie das Ergebnis aussieht. Seiten, Zeilen oder gar einzelne Zeichen aus verschiedenen Druckaufträgen werden bunt gemischt ausgedruckt. Die Probleme haben ihren Ursprung in der Art und Weise, wie die Threads umgeschaltet werden. Der Scheduler unterbricht zu einem uns unbekannten Zeitpunkt die Abarbeitung eines Threads und lässt den nächsten arbeiten. Wenn nun der erste Thread gerade Programmzeilen abarbeitet, die zusammengehören, und der zweite Thread beginnt, parallel auf diesen Daten zu arbeiten, so ist der Ärger vorprogrammiert. Wir müssen also Folgendes ausdrücken können: "Wenn ich den Job mache, dann möchte ich der Einzige sein, der die Ressource - etwa einen Drucker - nutzt." Erst nachdem der Drucker den Auftrag eines Benutzers fertiggestellt hat, darf er den nächsten in Angriff nehmen.

Kritische Abschnitte

Zusammenhängende Programmblöcke, denen während der Ausführung von einem Thread kein anderer Thread "reinwurschteln" sollte und die daher besonders geschützt werden müssen, nennen sich **kritische Abschnitte**. Wenn lediglich ein Thread den Programmteil abarbeitet, dann nennen wir dies **gegenseitigen Ausschluss** oder **atomar**. Wir könnten das etwas lockerer sehen, wenn wir wüssten, dass innerhalb der Programmblöcke nur von den Daten gelesen wird. Sobald aber nur ein Thread Änderungen vornehmen möchte, ist ein Schutz nötig. Denn arbeitet ein Programm bei nebenläufigen Threads falsch, ist es nicht **thread-sicher** (engl. **thread-safe**).

Wir werden uns nun Beispiele für kritische Abschnitte anschauen und dann sehen, wie wir diese in Java realisieren können.

Nicht kritische Abschnitte

Wenn mehrere Threads auf das gleiche Programmstück zugreifen, muss das nicht zwangsläufig zu einem Problem führen, und Thread-Sicherheit ist immer gegeben. Immutable Objekte - nehmen wir an, ein Konstruktor belegt einmalig die Zustände - sind automatisch thread-sicher, da es keine Schreibzugriffe gibt und bei Lesezugriffen nichts schiefgehen kann. Immutable-Klassen wie `String` oder Wrapper-Klassen kommen daher ohne Synchronisierung aus.

Das Gleiche gilt für Methoden, die keine Objekteigenschaften verändern. Da jeder Thread seine thread-eigenen Variablen besitzt - jeder Thread hat einen eigenen Stack -, können lokale Variablen, auch Parametervariablen, beliebig gelesen und geschrieben werden. Wenn zum Beispiel zwei Threads die folgende statische Utility-Methode aufrufen, ist das kein Problem:

```
public static String reverse( String s )
{
    return new StringBuilder( s ).reverse().toString();
}
```

Jeder Thread wird eine eigene Variablenbelegung für `s` haben und ein temporäres Objekt vom Typ `StringBuilder` referenzieren.

Thread-sichere und nicht thread-sichere Klassen der Java Bibliothek

Es gibt in Java viele Klassen, die nicht thread-sicher sind - das ist sogar der Standard. So sind etwa alle `Format`-Klassen, wie `MessageFormat`, `NumberFormat`, `DecimalFormat`, `ChoiceFormat`, `DateFormat` und `SimpleDateFormat` nicht für den nebenläufigen Zugriff gemacht. In der Regel steht das in der JavaDoc, etwa bei `DateFormat`:

`"Synchronization. Date formats are not synchronized. It is recommended to create separate format instances for each thread. If multiple threads access a format concurrently, it must be synchronized externally."`

Wer also Objekte nebenläufig verwendet, der sollte immer in der Java API-Dokumentation nachschlagen, ob es dort einen Hinweis gibt, ob die Objekte überhaupt thread-sicher sind.

In einigen wenigen Fällen haben Entwickler die Wahl zwischen thread-sicheren und nicht thread-sicheren Klassen im JDK:

Obwohl es die Auswahl bei den Datenstrukturen im Prinzip gibt, werden `Vector` und `Hashtable` dennoch nicht verwendet.

Nicht thread-sicher	Thread-sicher
<code>StringBuilder</code>	<code>StringBuffer</code>
<code>ArrayList</code>	<code>Vector</code>
<code>HashMap</code>	<code>Hashtable</code>

Punkte parallel initialisieren

Nehmen wir an, ein Thread T1 möchte ein `Point`-Objekt `p` mit den Werten (1,1) belegen und ein zweiter Thread T2 möchte eine Belegung mit den Werten (2,2) durchführen.

Thread T1	Thread T2
<code>p.x = 1;</code>	<code>p.x = 2;</code>
<code>p.y = 1;</code>	<code>p.y = 2;</code>

Zwei Threads belegen beide den Punkt `p`

Beide Threads können natürlich bei einem 2-Kern-Prozessor parallel arbeiten, aber da sie auf gemeinsame Variablen zugreifen, ist der Zugriff auf `x` bzw. `y` von `p` trotzdem sequenziell. Um es nicht allzu kompliziert zu machen, vereinfachen wir unser Ausführungsmodell so, dass wir zwar zwei Threads laufen haben, aber nur jeweils einer ausgeführt wird. Dann ist es möglich, dass T1 mit der Arbeit beginnt und `x = 1` setzt. Da der Thread-Scheduler einen Thread jederzeit unterbrechen kann, kann nun T2 an die Reihe kommen, der `x = 2` und `y = 2` setzt. Wird dann T1 wieder Rechenzeit zugeteilt, darf T1 an der Stelle weitermachen, wo er aufgehört hat, und `y = 1` folgt. In einer Tabelle ist das Ergebnis noch besser zu sehen:

Wir erkennen das nicht beabsichtigte Ergebnis (2,1), es könnte aber auch (1,2) sein, wenn wir das gleiche Szenario beginnend mit T2 durchführen. Je nach zuerst abgearbeitetem Thread wäre jedoch nur (1,1) oder (2,2) korrekt. Die Threads sollen ihre Arbeit aber atomar erledigen, denn die Zuweisung bildet einen kritischen Abschnitt, der geschützt werden muss. Standardmäßig sind die zwei Zuweisungen nicht-atomare Operationen und können unterbrochen werden.

Thread T1	Thread T2	x/y
<code>p.x = 1;</code>		1/0
	<code>p.x = 2;</code>	2/0
	<code>p.y = 2;</code>	2/2
<code>p.y = 1;</code>		2/1

Mögliche sequentielle Abarbeitung der Punktbelegung

Um dies an einem Beispiel zu zeigen, sollen zwei Threads ein `Point`-Objekt verändern. Die Threads belegen `x` und `y` immer gleich, und immer dann, wenn sich die Koordinaten unterscheiden, soll es eine Meldung geben:

```
com/tutego/insel/thread/concurrent/ParallelPointInit.java, main()
final Point p = new Point();

Runnable r = new Runnable()
{
    @Override public void run()
    {
        int x = (int)(Math.random() * 1000), y = x;

        while ( true )
        {
            p.x = x; p.y = y; // *

            int xc = p.x, yc = p.y; // *

            if ( xc != yc )
                System.out.println( "Aha: x=" + xc + ", y=" + yc );
        }
    }
};

new Thread( r ).start();
new Thread( r ).start();
```

Die interessanten Zeilen sind mit * markiert. `p.x = x; p.y = y;` belegt die Koordinaten neu, und `int xc = p.x, yc = p.y;` liest die Koordinaten erneut aus. Würden Belegung und Auslesen in einem Rutsch passieren, dürfte überhaupt keine unterschiedliche Belegung von `x` und `y` zu finden sein. Doch das Beispiel zeigt es anders:

```
Aha: x=58, y=116
Aha: x=116, y=58
Aha: x=58, y=116
Aha: x=58, y=116
...

```

Was wir mit den parallelen Punkten vor uns haben, sind Effekte, die von den Ausführungszeiten der einzelnen Operationen abhängen. In Abhängigkeit von dem Ort der Unterbrechung wird ein fehlerhaftes Verhalten produziert. Dieses Szenario nennt sich im Englischen **race condition** beziehungsweise **race hazard** (zu Deutsch auch **Wettlaufsituation**).

i++ sieht atomar aus, ist es aber nicht

Das Beispiel von vorhin ist plastisch und einleuchtend, weil zwischen Anweisungen unterbrochen werden kann. Das Problem liegt aber noch tiefer. Schon einfache Anweisungen wie `i++` müssen geschützt werden. Um dies zu verstehen, wollen wir einen Blick auf folgende Zeilen werfen:

com/tutego/insel/thread/IPlusPlus.java, IPlusPlus

```
public class IPlusPlus
{
    static int i;
    static void foo()
    {
        i++;
    }
}
```

Die Objektmethode `foo()` erhöht die statische Variable `i`. Um zu erkennen, dass `i++` ein kritischer Abschnitt ist, sehen wir uns den dazu generierten Bytecode für die Methode `foo()` an. (Machbar ist das zum Beispiel mit dem jeder Java-Distribution beiliegenden Dienstprogramm `javap` und der Option `-c`.)

```
0 getstatic #19 <Field int i>
3  iconst_1
4  iadd
5  putstatic #19 <Field int i>
8  return
```

Die einfach aussehende Operation `i++` ist also etwas komplizierter. Zuerst wird `i` gelesen und auf dem Stack abgelegt. Danach wird die Konstante 1 auf den Stack gelegt, und anschließend addiert `iadd` beide Werte. Das Ergebnis steht wiederum auf dem Stack und wird von `putstatic` zurück in `i` geschrieben.

Wenn jetzt auf die Variable `i` von zwei Threads A und B gleichzeitig zugegriffen wird, kann folgende Situation eintreten:

- ✓ Thread A holt sich den Wert von `i` in den internen Speicher, wird dann aber unterbrochen. Er kann das um 1 erhöhte Resultat nicht wieder `i` zuweisen.
- ✓ Nach der Unterbrechung von A kommt Thread B an die Reihe. Auch er besorgt sich `i`, kann aber `i + 1` berechnen und das Ergebnis in `i` ablegen. Dann ist B beendet, und der Scheduler beachtet Thread A.
- ✓ Jetzt steht in `i` das von Thread B um 1 erhöhte `i`. Thread A addiert nun 1 zu dem gespeicherten alten Wert von `i` und schreibt dann nochmals denselben Wert wie Thread B zuvor. Insgesamt wurde die Variable `i` nur um 1 erhöht, obwohl zweimal inkrementiert werden sollte. Jeder Thread hat für sich gesehen das korrekte Ergebnis berechnet.

Wenn wir unsere Methode `foo()` atomar ausführen, haben wir das Problem nicht mehr, weil das Lesen aus `i` und das Schreiben zusammen einen unteilbaren, kritischen Abschnitt bilden.

Kritische Abschnitte schützen

Beginnen wir mit einem anschaulichen Alltagsbeispiel. Gehen wir aufs Klo, schließen wir die Tür hinter uns. Möchte jemand anderes auf die Toilette, muss er warten. Vielleicht kommen noch mehrere dazu, die müssen dann auch warten, und eine Warteschlange bildet sich. Dass die Toilette besetzt ist, signalisiert die abgeschlossene Tür. Jeder Wartende muss so lange vor dem Klo ausharren, bis das Schloss geöffnet wird, selbst wenn der auf der Toilette Sitzende nach einer langen Nacht einnicken sollte.

Wie übertragen wir das auf Java? Wenn die Laufzeitumgebung nur einen Thread in einen Block lassen soll, ist ein **Monitor** nötig. (Der Begriff geht auf C. A. R. Hoare zurück, der in seinem Aufsatz "Communicating Sequential Processes" von 1978 erstmals dieses Konzept veröffentlichte.) Ein Monitor wird mithilfe eines **Locks** (zu Deutsch **Schloss**) realisiert, das ein Thread öffnet oder schließt. Tritt ein Thread in den kritischen Abschnitt ein, muss Programmcode wie eine Tür abgeschlossen werden (engl. **lock**). Erst wenn der Abschnitt durchlaufen wurde, darf die Tür wieder aufgeschlossen werden (engl. **unlock**), und ein anderer Thread kann den Abschnitt betreten.



Ein anderes Wort für Lock ist **Mutex** (engl. mutual exclusion, also "gegenseitiger Ausschluss"). Der Begriff Monitor wird oft mit Lock (Mutex) gleichgesetzt, doch kann ein Monitor mit Warten/Benachrichtigen mehr als ein klassischer Lock. In der Definition der Sprache Java tauchen die Begriffe Mutex und Lock allerdings nicht auf; die Autoren sprechen nur von den Monitor-Aktionen `lock` und `unlock`. Die Java Virtual Machine definiert dafür die Opcodes `monitorenter` und `monitorexit`.

Java-Konstrukte zum Schutz der kritischen Abschnitte

Wenn wir auf unser Punkte-Problem zurückkommen, so stellen wir fest, dass zwei Zeilen auf eine Variable zugreifen:

```
p.x = x; p.y = y;
int xc = p.x, yc = p.y;
```

Diese beiden Zeilen bilden also einen kritischen Abschnitt, den jeweils nur ein Thread betreten darf. Wenn also einer der Threads mit `p.x = x` beginnt, muss er so lange den exklusiven Zugriff bekommen, bis er mit `yc = p.y` endet.

Aber wie wird nun ein kritischer Abschnitt bekannt gegeben? Zum Markieren und Abschließen dieser Bereiche gibt es zwei Konzepte:

Konstrukt	Eingebautes Schlüsselwort	Java-Standardbibliothek
Schlüsselwort/Typen	<code>synchronized</code>	<code>java.util.concurrent.locks.Lock</code>
Nutzungsschema	<code>synchronized</code> { Tue1 Tue2 }	<code>lock.lock();</code> { Tue1 Tue2 } <code>lock.unlock();</code> *

Lock-Konzepte (Vereinfachte Darstellung, später mehr.)*

Beim `synchronized` entsteht Bytecode, der der JVM sagt, dass ein kritischer Block beginnt und endet. So überwacht die JVM, ob ein zweiter Thread warten muss, wenn er in einen synchronisierten Block eintritt, der schon von einem Thread ausgeführt wird. Bei `Lock` ist das Ein- und Austreten explizit vom Entwickler programmiert, und vergisst er das, ist das ein Problem. Und während bei der `Lock`-Implementierung das Objekt, an dem synchronisiert wird, offen hervortritt, ist das bei `synchronized` nicht so offensichtlich. Hier gilt es zu wissen, dass jedes Objekt in Java implizit mit einem Monitor verbunden ist. Da moderne Programme aber mittlerweile mit `Lock`-Objekten arbeiten, tritt die `synchronized`-Möglichkeit, die schon Java 1.0 zur Synchronisation bot, etwas in den Hintergrund.

Fassen wir zusammen: Nicht thread-sichere Abschnitte müssen geschützt werden. Sie können entweder mit `synchronized` geschützt werden, bei dem der Eintritt und Austritt implizit geregelt ist, oder durch `Lock`-Objekte. Befindet sich dann ein Thread in einem geschützten Block und möchte ein zweiter Thread in den Abschnitt, muss er so lange warten, bis der erste Thread den Block wieder freigibt.

So ist die Abarbeitung über mehrere Threads einfach synchronisiert, und das Konzept eines Monitors gewährleistet seriellen Zugriff auf kritische Ressourcen. Die kritischen Bereiche sind nicht per se mit einem Monitor verbunden, sondern werden eingerahmt, und dieser Rahmen ist mit einem Monitor (Lock) verbunden.

Mit dem Abschließen und Aufschließen werden wir uns noch intensiver in den folgenden Abschnitten beschäftigen.

Kritische Abschnitte mit ReentrantLock schützen

Seit Java 5 gibt es die Schnittstelle `Lock`, mit der sich ein kritischer Block markieren lässt. Ein Abschnitt beginnt mit `lock()` und endet mit `unlock()`:

```
com/tutego/insel/thread/concurrent/ParallelPointInitSync.java, main()

final Lock lock = new ReentrantLock();
final Point p = new Point();

Runnable r = new Runnable()
{
    @Override public void run()
    {
        int x = (int)(Math.random() * 1000), y = x;

        while ( true )
        {
            lock.lock();

            p.x = x; p.y = y;          // *
            int xc = p.x, yc = p.y;    // *
            lock.unlock();

            if ( xc != yc )
                System.out.println( "Aha: x=" + xc + ", y=" + yc );
        }
    }
};

new Thread( r ).start();
new Thread( r ).start();
```

Mit dieser Implementierung wird keine Ausgabe auf dem Bildschirm folgen.

Die Schnittstelle `java.util.concurrent.locks.Lock`

`Lock` ist eine Schnittstelle, von der `ReentrantLock` die wichtigste Implementierung ist. Mit ihr lässt sich der Block betreten und verlassen.

```
interface java.util.concurrent.locks.Lock
```

- ✓ `void lock()`

Wartet so lange, bis der ausführende Thread den kritischen Abschnitt betreten kann, und markiert ihn dann als betreten. Hat schon ein anderer Thread an diesem `Lock`-Objekt ein `lock()` aufgerufen, so muss der aktuelle Thread warten, bis der Lock wieder frei ist. Hat der aktuelle Thread schon den Lock, kann er bei der Implementierung `ReentrantLock` wiederum `lock()` aufrufen und sperrt sich nicht selbst.

- ✓ `boolean tryLock()`

Wenn der kritische Abschnitt sofort betreten werden kann, ist die Funktionalität wie bei `lock()`, und die Rückgabe ist `true`. Ist der Lock gesetzt, so wartet die Methode nicht wie `lock()`, sondern kehrt mit einem `false` zurück.

- ✓ `boolean tryLock(long time, TimeUnit unit) throws InterruptedException`
Versucht in der angegebenen Zeitspanne den Lock zu bekommen. Das Warten kann mit `interrupt()` auf dem Thread unterbrochen werden, was `tryLock()` mit einer Exception beendet.
- ✓ `void unlock()`
Verlässt den kritischen Block.
- ✓ `void lockInterruptibly() throws InterruptedException`
Wartet wie `lock()`, um den kritischen Abschnitt betreten zu dürfen, kann aber mit einem `interrupt()` von außen abgebrochen werden (der `lock()`-Methode ist ein Interrupt egal). Implementierende Klassen müssen diese Vorgabe nicht zwingend umsetzen, sondern können die Methode auch mit einem einfachen `lock()` realisieren. `ReentrantLock` implementiert `lockInterruptibly()` erwartungsgemäß.

Beispiel: Wenn wir sofort in den kritischen Abschnitt gehen können, tun wir das; sonst tun wir etwas anderes:

```
Lock lock = ...;
if ( lock.tryLock() )
{
    try {
        ...
    }
    finally { lock.unlock(); }
}
else
    ...
```

Die Implementierung `ReentrantLock` kann noch ein bisschen mehr als `lock()` und `unlock()`:

```
class java.util.concurrent.locks.ReentrantLock
implements Lock, Serializable
```

- ✓ `ReentrantLock()`
Erzeugt ein neues Lock-Objekt, das nicht dem am längsten Wartenden den ersten Zugriff gibt.
- ✓ `ReentrantLock(boolean fair)`
Erzeugt ein neues Lock-Objekt mit fairem Zugriff, gibt also dem am längsten Wartenden den ersten Zugriff.
- ✓ `boolean isLocked()`
Fragt an, ob der Lock gerade genutzt wird und im Moment kein Betreten möglich ist.
- ✓ `final int getQueueLength()`
Ermittelt, wie viele auf das Betreten des Blocks warten.
- ✓ `int getHoldCount()`
Gibt die Anzahl der erfolgreichen `lock()`-Aufrufe ohne passendes `unlock()` zurück. Sollte nach Beenden des Vorgangs 0 sein.

Beispiel: Das Warten auf den Lock kann unterbrochen werden:

```
Lock l = new ReentrantLock();
try
{
    l.lockInterruptibly();
    try
    {
        ...
    }
    finally { l.unlock(); }
}
catch ( InterruptedException e ) { ... }
```

Wenn wir den Lock nicht bekommen haben, dürfen wir ihn auch nicht freigeben!

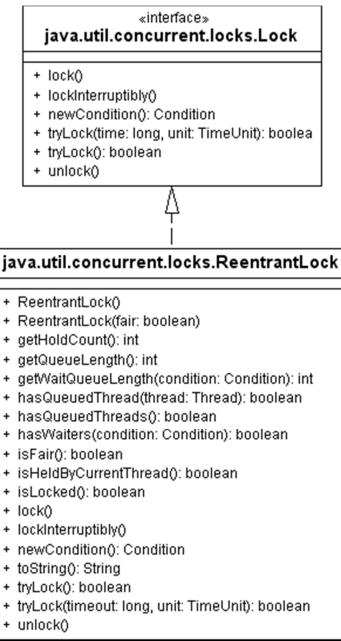


ReentrantReadWriteLock

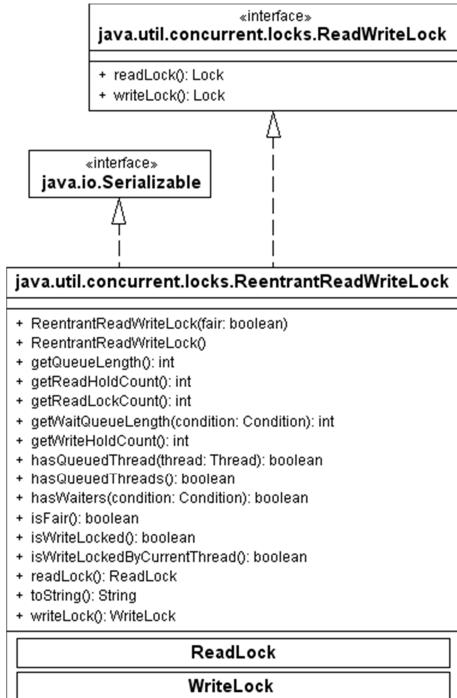
Unsere Klasse `ReentrantLock` blockt bei jedem `lock()` und lässt keinen Interessenten in den kritischen Abschnitt. Viele Szenarien sind jedoch nicht so streng, und so kommt es zu Situationen, in denen lesender Zugriff durchaus von mehreren Parteien möglich ist, schreibender Zugriff aber blockiert wird.

Für diese Lock-Situation gibt es die Schnittstelle `ReadWriteLock`, die nicht von `Lock` abgeleitet ist, sondern mit `readLock()` und `writeLock()` die Lock-Objekte liefert. Die bisher einzige Implementierung der Schnittstelle ist `java.util.concurrent.locks.ReentrantReadWriteLock`. Ein Programm Ausschnitt könnte so aussehen:

```
ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
try
{
    lock.readLock().lock();
    ...
}
finally
{
    lock.readLock().unlock();
}
```



Die Klasse `ReentrantLock` implementiert die Schnittstelle `Lock`



Die Klasse `ReentrantReadWriteLock` implementiert `ReadWriteLock`

Synchronisieren mit synchronized

Schon seit Java 1.0 können kritische Abschnitte mit `synchronized` geschützt werden. Im einfachsten Fall markiert der Modifizierer `synchronized` die gesamte Methode. Ein betretender Thread setzt bei Objektmethoden den Monitor des `this`-Objekts und bei statischen Methoden den Lock des dazugehörigen `Class`-Objekts.

Betrifft ein Thread A eine synchronisierte Methode eines Objekts O und versucht anschließend Thread B eine synchronisierte Methode des gleichen Objekts O aufzurufen, muss der nachfolgende Thread B so lange warten, bis A wieder aus dem synchronisierten Teil austritt. Das geschieht, wenn der erste Thread A die Methode verlässt, denn mit dem Verlassen einer Methode - oder auch einer Ausnahme - gibt die JVM automatisch den Lock frei. Die Dauer eines Locks hängt folglich mit der Dauer des Methodenaufrufs zusammen, was zur Konsequenz hat, dass längere kritische Abschnitte die Parallelität einschränken und zu längeren Wartezeiten führen. Eine Endlosschleife in der synchronisierten Methode gäbe den Lock niemals frei.

Das aus IPlusPlus.java bekannte Problem mit dem `i++` lässt sich mit `synchronized` einfach lösen:

```
synchronized void foo() { i++; }
```

Bei einem Konflikt (mehrere Threads rufen `foo()` auf) verhindert `synchronized`, dass sich mehr als ein Thread gleichzeitig im kritischen Abschnitt, dem Rumpf der Methode `foo()`, befinden kann. Dies bezieht sich nur auf mehrere Aufrufe von `foo()` für dasselbe Objekt. Zwei verschiedene Threads können durchaus parallel die Methode `foo()` für unterschiedliche Objekte ausführen.

Neben diesem speziellen Problem für atomares Verändern von Variablen lassen sich auch Klassen aus dem Paket `java.util.concurrent.atomic` verwenden.

Bei einem orthografisch anspruchsvollen Wort wie `synchronized` ist es praktisch, dass Eclipse auch Schlüsselwörter vervollständigt. Hier reicht ein Tippen von `sync` und für einen Dialog.

Synchronized-Methoden der Klasse StringBuffer

Wir wollen uns anhand einiger Beispiele noch ansehen, an welchen Objekten der Monitor beziehungsweise Lock gespeichert wird. Zunächst betrachten wir die Methode `charAt()` der Klasse `StringBuffer` und versuchen zu verstehen, warum die Methode `synchronized` ist.

```
public synchronized char charAt( int index )
{
    if ( (index < 0) || (index >= count) )
        throw new StringIndexOutOfBoundsException( index );

    return value[index];
}
```

Neben `charAt()` sind noch eine ganze Reihe anderer Methoden synchronisiert, etwa `getChars()`, `setCharAt()` und `append()`. Bei einer `synchronized`-Methode wird also der Lock bei einem konkreten `StringBuffer`-Objekt gespeichert. Wäre die Methode `charAt()` nicht atomar, dann könnte es passieren, dass durch Multithreading zwei Threads das gleiche `StringBuffer`-Objekt bearbeiten. Probleme könnten sich zum Beispiel dann ergeben, wenn ein Thread gerade den String verkleinert und gleichzeitig `charAt()` aufgerufen wird. Auch wenn zuerst `charAt()` einen gültigen Index feststellt, dann aber der `StringBuffer` verkleinert wird, gibt es ein Problem. Dann wäre nämlich der Index ungültig und `value[index]` fehlerhaft. Da aber `charAt()` synchronisiert ist, kann kein anderer Thread dasselbe `StringBuffer`-Objekt über synchronisierte Methoden modifizieren.

Beispiel

Das `StringBuffer`-Objekt `sb1` wird von zwei Threads T1 und T2 bearbeitet, indem synchronisierte Methoden genutzt werden. Bearbeitet Thread T1 den `StringBuffer` `sb1` mit einer synchronisierten Methode, dann kann T2 erst dann eine synchronisierte Methode für `sb1` aufrufen, wenn T1 die Methode abgearbeitet hat. Denn T1 setzt bei `sb1` die Sperre, die T2 warten lässt. Gleichzeitig kann aber T2 synchronisierte Methoden für ein anderes `StringBuffer`-Objekt `sb2` aufrufen, da `sb2` einen eigenen Monitor besitzt. Das macht noch einmal deutlich, dass die Locks zu einem Objekt gehören und nicht zur synchronisierten Methode.

Mit `synchronized` synchronisierte Blöcke

Wenn wir mit `Lock`-Objekten arbeiten, können wir den Block so fein wählen, wie es erforderlich ist. Mit `synchronized` haben wir bisher nur eine gesamte Methode sperren können, was in manchen Fällen etwas viel ist. Dann kann eine allgemeinere Variante in Java eingesetzt werden, die nur einen Block synchronisiert. Dazu schreiben wir in Java Folgendes:

```
synchronized ( objektMitDemMonitor )
{
    ...
}
```

Der Block wird in die geforderten geschweiften Klammern gesetzt, und hinter dem Schlüsselwort in Klammern muss ein Objekt stehen, das den zu verwendenden Monitor besitzt. Die Konsequenz ist die Möglichkeit, über einen beliebigen Monitor zu synchronisieren und nicht unbedingt über den Monitor des Objekts, für das die synchronisierte Methode aufgerufen wurde, wie es bei synchronisierten Objektmethoden üblich ist.

Eine synchronisierte Objektmethode ist nichts anderes als eine Variante von:



```
synchronized( this )
{
    // Code der Methode.
}
```

Statisch synchronisierte Blöcke

Nicht nur Objektmethoden, sondern auch Klassenmethoden können `synchronized` sein. Doch die Nachbildung in einem Block sieht etwas anders aus, da es keine `this`-Referenz gibt. Hier kann ein `Object`-Exemplar für einen Lock herhalten, der extra für die Klasse angelegt wird. Dies ist eines der seltenen Beispiele, in denen ein Exemplar der Klasse `Object` Sinn ergibt:

```
com/tutego/insel/thread/StaticSync.java
package com.tutego.insel.thread;

class StaticSync
{
    private static final Object o = new Object();

    static void staticFoo()
    {
        synchronized( o )
        {
            // ...
        }
    }
}
```

Alternativ könnten wir auch das zugehörige `Class`-Objekt einsetzen. Wir müssen das entsprechende Klassenobjekt dann nur mittels `StaticSync.class` erfragen. Würden wir gleich mit `Lock`-Objekten arbeiten, stellt sich die Frage erst gar nicht.

Bei `Lock`-Objekten oder `synchronized`-Blöcken kann der zwingend synchronisierbare Teil in einem kleinen Abschnitt bleiben. Die JVM kann die anderen Teile parallel abarbeiten, und andere Threads dürfen die anderen Teile betreten. Als Resultat ergibt sich eine verbesserte Geschwindigkeit.



Dann machen wir doch gleich alles synchronisiert!

In nebenläufigen Programmen kann es schnell zu unerwünschten Nebeneffekten kommen. Das ist auch der Grund, warum thread-lastige Programme schwer zu debuggen sind. Warum sollten wir also nicht alle Methoden synchronisieren? Wäre dann nicht das Problem aus der Welt geschafft? Prinzipiell würde das einige Probleme lösen, doch hätten wir uns damit andere Nachteile eingefangen:

- ✓ Methoden, die synchronisiert sind, müssen von der JVM besonders bedacht werden, damit keine zwei Threads die Methode für das gleiche Objekt ausführen. Wenn also ein zweiter Thread in die Methode eintreten möchte, kann er das nicht einfach machen, sondern muss vielleicht erst neben vielen anderen Threads warten. Es muss also eine Datenstruktur geben, in der wartende Threads eingetragen und ausgewählt werden. Das kostet zusätzlich Zeit und ist im Vergleich zu einem normalen Methodenaufruf teurer.

- ✓ Zusätzlich kommt ein Problem hinzu, wenn eine nicht notwendigerweise, also überflüssigerweise synchronisierte Methode eine Endlosschleife oder lange Operationen durchführt. Dann warten alle anderen Threads auf die Freigabe, und das kann im Fall der Endlosschleife ewig sein. Auch bei Multiprozessorsystemen profitieren wir nicht von dieser Programmiertechnik. Unnötig und falsch synchronisierte Blöcke machen die Vorteile von Mehrprozessormaschinen zunichte.
- ✓ Wenn alle Methoden synchronisiert sind, steigt auch die Gefahr eines Deadlocks. In den folgenden Abschnitten erfahren wir etwas mehr über den Fall, dass zwei Threads wechselseitig auf Ressourcen eines jeweils anderen Threads zugreifen wollen und sich dabei im Wege stehen.

Ist der gesamte Zugriff auf ein Objekt synchronisiert und kann kein zweiter Thread irgendwelche Eigenschaften parallel zu einem anderen Thread nutzen, nennt sich das Objekt voll synchronisiert im Gegensatz zu teil-synchronisiert. Sind einige Methoden der Klasse nicht synchronisiert, kann ein zweiter Thread parallel zu den synchronisierten Blöcken an die Eigenschaften gehen.

Lock-Freigabe im Fall von Exceptions

Kommt es innerhalb eines `synchronized`-Blocks beziehungsweise innerhalb einer synchronisierten Methode zu einer nicht überprüften `RuntimeException`, wird die JVM den Lock automatisch freigeben. Der Grund: Die Laufzeitumgebung gibt den Lock automatisch frei, wenn der Thread den synchronisierten Block verlässt, was bei einer Exception der Fall ist.

Werden die mit dem Schlüsselwort `synchronized` geschützten Blöcke durch Lock-Objekte umgesetzt, ist darauf zu achten, die Locks auch im Exception-Fall wieder freizugeben. Ein `finally` mit `unlock()` kommt da gerade recht, denn `finally` wird ja immer ausgeführt, egal, ob es einen Fehler gab oder nicht:

```
com/tutego/insel/thread/concurrent/UnlockInFinally.java, main()
ReentrantLock lock = new ReentrantLock();

try
{
    lock.lock();

    try
    {
        System.out.println( lock.getHoldCount() ); // 1
        System.out.println( 12 / 0 );
    }
    finally
    {
        lock.unlock();
    }
}
catch ( Exception e )
{
    System.out.println( e.getMessage() ); // by zero
}
System.out.println( lock.getHoldCount() ); // 0
```

Nach dem `lock()` liefert `getHoldCount()` 1, da ein Thread den Block betreten hat. Die Division durch null provoziert eine `RuntimeException`, und `finally` gibt den Lock frei. Die Ausnahme wird abgefangen, und `getHoldCount()` liefert wieder 0, da `finally` das `unlock()` ausgeführt hat. Würden wir die Zeile mit `unlock()` auskommentieren, so würde `getHoldCount()` weiterhin 1 liefern, was ein Fehler ist.

Deadlocks

Ein **Deadlock** (zu Deutsch etwa **tödliche Umarmung**) kommt beispielsweise dann vor, wenn ein Thread A eine Ressource belegt, die ein anderer Thread B haben möchte, und Thread B eine Ressource belegt, die A gerne nutzen würde. In dieser Situation können beide nicht vor und zurück und befinden sich in einem dauernden Wartezustand. Deadlocks können in Java-Programmen nicht erkannt und verhindert werden. Uns fällt also die Aufgabe zu, diesen ungünstigen Zustand gar nicht erst herbeizuführen.

Das nächste Beispiel soll über eine Verklemmung einen Deadlock provozieren. Zwei Threads wetteifern um die Lock-Objekte lock1 und lock2. Dabei kommt es zu einem Deadlock, da der eine genau den einen Lock besetzt, den der jeweils andere zum Weiterarbeiten benötigt:

```
com/tutego/insel/thread/Deadlock.java
package com.tutego.insel.thread;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.*;

class Deadlock
{
    static Lock lock1 = new ReentrantLock(),
              lock2 = new ReentrantLock();

    static class T1 extends Thread
    {
        @Override
        public void run()
        {
            lock1.lock();
            System.out.println( "T1: Lock auf lock1 bekommen" );

            try { TimeUnit.SECONDS.sleep( 1 ); } catch ( InterruptedException e ) { }

            lock2.lock();
            System.out.println( "T1: Lock auf lock2 bekommen" );

            lock2.unlock();
            lock1.unlock();
        }
    }

    static class T2 extends Thread
    {
        @Override
        public void run()
        {
            lock2.lock();
            System.out.println( "T2: Lock auf lock2 bekommen" );

            lock1.lock();
            System.out.println( "T2: Lock auf lock1 bekommen" );

            lock1.unlock();
            lock2.unlock();
        }
    }

    public static void main( String[] args )
    {
        new T1().start();
        new T2().start();
    }
}
```

In der Ausgabe sehen wir nur zwei Zeilen, und schon hängt das gesamte Programm:

```
T1: Lock auf lock1 bekommen
T2: Lock auf lock2 bekommen
```

Eine Lösung des Problems wäre, bei geschachteltem Synchronisieren auf mehrere Objekte diese immer in der gleichen Reihenfolge zu belegen, also etwa immer erst `lock1`, dann `lock2`. Bei unbekannten, dynamisch wechselnden Objekten muss dann unter Umständen eine willkürliche Ordnung festgelegt werden.



Die JVM von Oracle verfügt über eine eingebaute Deadlock-Erkennung, die auf der Konsole aktiviert werden kann. Dazu ist unter Windows die Tastenkombination `Strg` `Pause` zu drücken und unter Linux oder Solaris `Strg` `Esc`.

5.6 Synchronisation über Warten und Benachrichtigen

Die Synchronisation von Methoden oder Blöcken ist eine einfache Möglichkeit, konkurrierende Zugriffe von der virtuellen Maschine auflösen zu lassen. Obwohl die Umsetzung mit den Locks die Programmierung einfach macht, reicht dies für viele Aufgabenstellungen nicht aus. Wir können zwar Daten in einer synchronisierten Abfolge austauschen, doch gerne möchte ein Thread das Ankommen von Informationen signalisieren, und andere Threads wollen informiert werden, wenn Daten bereitstehen und abgeholt werden können. Bei der Realisierung der Benachrichtigungen gibt es eine Reihe von Möglichkeiten. Im Folgenden nennen wir die einfachsten:

- ✓ Jedes Objekt besitzt über die Klasse `java.lang.Object` die Methoden `wait()` und `notify()`. Ein Thread, der über den Monitor verfügt, kann die Methoden aufrufen und sich so in einen Wartezustand versetzen oder einen anderen Thread aufwecken. Diese Möglichkeit gibt es seit Java 1.0 (es ist schon ein wenig seltsam, dass Java für die Synchronisation ein eingebautes Schlüsselwort hat, aber die Benachrichtigung über Methoden realisiert).
- ✓ Von einem `ReentrantLock` - der den Monitor repräsentiert - liefert `newCondition()` ein `Condition`-Objekt, das sich über `await()` und `signal()` warten und benachrichtigen lässt. Diese Typen gibt es seit Java 5.

Szenarien mit Warten und Benachrichtigen sind oft Produzenten-Konsumenten-Beispiele. Ein Thread liefert Daten, die ein anderer Thread verwenden möchte. Da dieser in keiner kostspieligen Schleife auf die Information warten soll, synchronisieren sich die Partner über ein beiden bekanntes Objekt. Erst wenn der Produzent sein Okay gegeben hat, ergibt es für den Datennutzer Sinn, weiterzuarbeiten; jetzt hat er seine benötigten Daten. So wird keine unnötige Zeit in Warteschleifen vergeudet, und der Prozessor kann die übrige Zeit anderen Threads zuteilen.

Die Schnittstelle Condition

Mit einem `Lock`-Objekt wie `ReentrantLock` können zwecks Benachrichtigung `Condition`-Objekte abgeleitet werden. Dazu dient die Methode `newCondition()`:

```
interface java.util.concurrent.locks.Lock
```

- ✓ `Condition newCondition()`
Liefert ein `Condition`-Objekt, das mit dem `Lock` verbunden ist. Mit einem `Lock`-Objekt können beliebig viele `Condition`-Objekte gebildet werden.

Warten mit `await()` und Aufwecken mit `signal()`

Damit das Warten und Benachrichtigen funktioniert, kommunizieren die Parteien über ein gemeinsames `Condition`-Objekt, das vom `Lock` erfragt wird:

```
Condition condition = lock.newCondition();
```

In einem fiktiven Szenario soll ein Thread T1 auf ein Signal warten und ein Thread T2 dieses Signal geben. Da nun beide Threads Zugriff auf das gemeinsame `Condition`-Objekt haben, kann T1 sich mit folgender Anweisung in den Schlaf begeben:

```

try {
    condition.await();
} catch ( InterruptedException e ) {
    ...
}

```

Mit dem `await()` geht der Thread in den Zustand **nicht ausführend** über. Der Grund für den `try`-Block ist, dass ein `await()` durch eine `InterruptedException` vorzeitig abgebrochen werden kann. Das passiert zum Beispiel, wenn der wartende Thread per `interrupt()`-Methode einen Hinweis zum Abbruch bekommt.

Die Methode `await()` bestimmt den ersten Teil des Paars. Der zweite Thread T2 kann nun nach dem Eintreffen einer Bedingung das Signal geben:

```
condition.signal();
```

Um das `signal()` muss es keinen Block geben, der Exceptions auffängt.

Wenn ein Thread ein `signal()` auslöst und es keinen wartenden Thread gibt, dann verhält das `signal()` ungehört. Der Hinweis wird nicht gespeichert, und ein nachfolgendes `await()` muss mit einem neuen `signal()` aufgeweckt werden.

<code>interface java.util.concurrent.lock.Condition</code>	
«interface»	
<code>java.util.concurrent.locks.Condition</code>	
<ul style="list-style-type: none"> + <code>await()</code> + <code>await(time: long, unit: TimeUnit): boolean</code> + <code>awaitNanos(nanosTimeout: long): long</code> + <code>awaitUninterruptibly()</code> + <code>awaitUntil(deadline: Date): boolean</code> + <code>signal()</code> + <code>signalAll()</code> 	

- ✓ `void await() throws InterruptedException`
Wartet auf ein Signal, oder die Methode wird unterbrochen.
- ✓ `void signal()`
Weckt einen wartenden Thread auf.

Die Schnittstelle Condition

Vor der Condition kommt ein Lock

Auf den ersten Blick scheint es, als ob das `Lock`-Objekt nur die Aufgabe hat, ein `Condition`-Objekt herzugeben. Das ist aber noch nicht alles, denn die Methoden `await()` und auch `signal()` können nur dann aufgerufen werden, wenn vorher ein `lock()` den Signal-Block exklusiv sperrt:

```

lock.lock();
try {
    condition.await();
} catch ( InterruptedException e ) {
    ...
}
finally {
    lock.unlock();
}

```

Doch was passiert ohne Aufruf von `lock()`? Zwei Zeilen zeigen die Auswirkung:

```
com/tutego/insel/thread/concurrent/AwaitButNoLock.java, main()
Condition condition = new ReentrantLock().newCondition();
condition.await();      // java.lang.IllegalMonitorStateException
```

Das Ergebnis ist eine `java.lang.IllegalMonitorStateException`.

Temporäre Lock-Freigabe bei await()

Um auf den `Condition`-Objekten also `await()` und `signal()` aufrufen zu können, ist ein vorangehender Lock nötig. Augenblick mal: Wenn ein `await()` kommt, hält der Thread doch den Monitor, und kein anderer Thread könnte in einem kritischen Abschnitt, der über das gleiche `Lock`-Objekt gesperrt ist, `signal()` aufrufen. Wie ist das möglich?

Die Lösung besteht darin, dass `await()` den Monitor freigibt und den Thread so lange sperrt, bis zum Beispiel von einem anderen Thread das `signal()` kommt (wenn wir ein Programm mit nur einem Thread haben, dann ergibt natürlich so ein Pärchen keinen Sinn). Kommt das Signal, weckt das den wartenden Thread wieder auf, und er kann am Scheduling wieder teilnehmen. Da also ein anderer Thread prinzipiell in den gleichen Block wie der Wartende hineinlaufen kann, ist das nicht **logisch atomar** - was es wäre, wenn der Thread komplett einen synchronisierten Block durchlief, bevor ein anderer Thread den Block betritt.

Mehrere Wartende und `signalAll()`

Es kann durchaus vorkommen, dass mehrere Threads in einer Warteposition an demselben Objekt sind und aufgeweckt werden wollen. `signal()` wählt dann aus der Liste der Wartenden einen Thread aus und gibt ihm das Signal. Sollten alle Wartenden einen Hinweis bekommen, lässt sich `signalAll()` verwenden.

```
interface java.util.concurrent.lock.Condition
```

- ✓ void signalAll()
 Weckt alle wartenden Threads auf.

`await()` mit einer Zeitspanne

Ein `await()` wartet im schlechtesten Fall bis zum Sankt-Nimmerleins-Tag, wenn es kein `signal()` gibt. Es gibt jedoch Situationen, in denen wir eine bestimmte Zeit lang warten, aber bei Fehlen der Benachrichtigung weitermachen wollen. Dazu kann dem `await()` in unterschiedlichen Formen eine Zeit mitgegeben werden.

```
interface java.util.concurrent.lock.Condition
```

- ✓ long awaitNanos(long nanosTimeout) throws InterruptedException
 Wartet eine bestimmte Anzahl Nanosekunden auf ein Signal, oder die Methode wird unterbrochen. Die Rückgabe gibt die Wartezeit an.
- ✓ boolean await(long time, TimeUnit unit) throws InterruptedException
- ✓ boolean awaitUntil(Date deadline) throws InterruptedException
 Wartet eine bestimmte Zeit lang auf ein Signal. Kommt das Signal in der Zeit nicht, geht die Methode weiter und liefert `true`. Kam das Signal oder ein `interrupt()`, liefert die Methode `false`.
- ✓ void awaitUninterruptibly()
 Wartet ausschließlich auf ein Signal und lässt sich nicht durch ein `interrupt()` beenden.

An den Methoden ist schon zu erkennen, dass die Wartezeit einmal relativ (`await()`) und einmal absolut (`awaitUntil()`) sein kann (mit den eingebauten Methoden `wait()` und `notify()` ist immer nur eine relative Angabe möglich).

Eine `IllegalMonitorStateException` wird das Ergebnis sein, wenn beim Aufruf einer `Condition`-Methode das `lock()` des zugrunde liegenden `Lock`-Objekts gefehlt hat.

Beispiel

Warte maximal zwei Sekunden auf das Signal über `condition1`. Wenn es nicht ankommt, versuche `signal()`/`signalAll()` von `condition2` zu bekommen:

```
condition1.await( 2, TimeUnit.SECONDS );
condition2.await();
```

Die Ausnahmebehandlung muss bei einem lauffähigen Beispiel noch hinzugefügt werden.

Warten mit `wait()` und Aufwecken mit `notify()`

Nachdem im vorigen Abschnitt der Weg mit den Klassen und Schnittstellen aus Java 5 beschritten wurde, wollen wir uns abschließend mit den Möglichkeiten beschäftigen, die Java seit der Version 1.0 mitbringt.

Nehmen wir wieder zwei Threads an. Sie sollen sich am Objekt `o` synchronisieren - die Methoden `wait()` und `notify()` sind nur mit dem entsprechenden Monitor gültig, und den besitzt das Programmstück, wenn es sich in einem synchronisierten Block aufhält. Thread T1 soll auf Daten warten, die Thread T2 liefert. T1 führt dann etwa den folgenden Programmcode aus:

```
synchronized( o )
{
    try {
        o.wait();
        // Habe gewartet, kann jetzt loslegen.
    } catch ( InterruptedException e ) {
        ...
    }
}
```

Wenn der zweite Thread den Monitor des Objekts `o` bekommt, kann er den wartenden Thread aufwecken. Er bekommt den Monitor durch das Synchronisieren der Methode, was ja bei Objektmethoden `synchronized(this)` entspricht. T2 gibt das Signal mit `notify()`:

```
synchronized( o )
{
    // Habe etwas gemacht und informiere jetzt meinen Wartenden.
    o.notify();
}
```

Um `notify()` muss es keinen Block geben, der Exceptions auffängt. Wenn ein Thread ein `notify()` auslöst und es keinen wartenden Thread gibt, dann verpufft es.

```
class java.lang.Object
```

- ✓ `void wait() throws InterruptedException`
Der aktuelle Thread wartet an dem aufrufenden Objekt darauf, dass er nach einem `notify()` / `notifyAll()` weiterarbeiten kann. Der aktive Thread muss natürlich den Monitor des Objekts belegt haben. Andernfalls kommt es zu einer `IllegalMonitorStateException`.
- ✓ `void wait(long timeout) throws InterruptedException`
Wartet auf ein `notify()` / `notifyAll()` eine gegebene Anzahl von Millisekunden. Nach Ablauf dieser Zeit geht es ohne Fehler weiter.
- ✓ `void wait(long timeout, int nanos) throws InterruptedException`
Wartet auf ein `notify()` / `notifyAll()` - angenähert $1.000.000 * \text{timeout} + \text{nanos}$ Nanosekunden.
- ✓ `void notify()`
Weckt einen beliebigen Thread auf, der an diesem Objekt wartet.
- ✓ `void notifyAll()`
Benachrichtigt alle Threads, die auf dieses Objekt warten.

Für die Variante `wait(long)` / `wait(long, int)` mit der Zeiteinheit bietet `TimeUnit` eine Alternative.

```
enum java.util.concurrent.TimeUnit
extends Enum<TimeUnit>
```

- ✓ `void timedWait(Object obj, long timeout) throws InterruptedException`
Berechnet aus der gewählten `TimeUnit` und dem `timeout` die Millisekunden (`ms`) und Nanosekunden (`ns`) und führt `obj.wait(ms, ns)` aus.

Ein `wait()` kann mit einer `InterruptedException` vorzeitig abbrechen, wenn der wartende Thread per `interrupt()`-Methode unterbrochen wird. Die Tatsache, dass `wait()` temporär den Lock freigibt, was für uns mit `synchronized` aber nicht möglich ist, spricht dafür, dass etwas wie `wait()` nativ implementiert werden muss.

Falls der Lock fehlt: IllegalMonitorStateException

Wenn `wait()` oder `notify()` aufgerufen werden, uns aber der entsprechende Lock für das Objekt fehlt, kommt es zum Laufzeitfehler `IllegalMonitorStateException`, wie wir es schon bei `Condition` und dem fehlenden `lock()` vom Lock gesehen haben. Was wird bei folgendem Programm passieren?

com/tutego/insel/thread/NotOwner.java, main()

```
package com.tutego.insel.thread;

class NotOwner
{
    public static void main( String[] args ) throws InterruptedException
    {
        new NotOwner().wait();
    }
}
```

Der Compiler kann das Programm übersetzen, doch zur Laufzeit wird es zu einem Fehler kommen:

```
java.lang.IllegalMonitorStateException: current thread not owner
at java.lang.Object.wait(Native Method)
at java.lang.Object.wait(Object.java:426)
at NotOwner.main(NotOwner.java:5)
Exception in thread "main"
```

Der Fehler zeigt an, dass der aktuelle ausführende Thread (`current thread`) nicht den nötigen Lock besitzt, um `wait()` auszuführen. Das Problem ist hier mit einem synchronized-Block (oder einer synchronized-Methode) zu lösen. Um den Fehler zu beheben, setzen wir:

```
NotOwner o = new NotOwner();
synchronized( o )
{
    o.wait();
}
```

Dies zeigt, dass das Objekt `o`, das den Lock besitzt, für ein `wait()` "bereit" sein muss. In die richtige Stimmung wird es nur mit `synchronized` gebracht:

```
synchronized( NotOwner.class )
{
    new NotOwner().wait();
}
```

Doch natürlich könnten wir auch am Klassenobjekt synchronisieren:

```
synchronized( NotOwner.class )
{
    NotOwner.class.wait();
}
```



Die Ähnlichkeit zwischen Lock auf der einen Seite und einem synchronisierten Block bzw. einer synchronisierten Methode auf der anderen und den Methoden `wait()` und `notify()` bei `Object` und den analogen Methoden `await()` und `signal()` bei den `Condition`-Objekten ist nicht zu übersehen.

Auch der Fehler beim Fehlen des Monitors ist der gleiche: Ein Aufruf der Methoden `await()/wait()` und `notify()/signal()` führt zu einer `IllegalMonitorStateException`. Es muss also erst ein synchronisierter Block für den Monitor her oder ein Aufruf `lock()` auf dem Condition zugrunde liegenden Lock-Objekt.

5.7 Datensynchronisation durch besondere Concurrency-Klassen

Arbeiten mehrere Threads zusammen, so wollen sie in der Regel Daten austauschen und sich an bestimmten Bedingungen synchronisieren. Die Java API bietet für diese Zusammenarbeit eine Reihe von Klassen:

- ✓ **Semaphore**: Erlaubt eine maximale Anzahl Threads in einem Programmblöck.
- ✓ **CyclicBarrier**: Eine Menge von Threads wartet aufeinander, um zu einem gemeinsamen Punkt zu kommen.
- ✓ **CountDownLatch**: Ein oder mehrere Threads warten auf eine Bedingung. Ist sie erfüllt, können die Threads fortfahren.
- ✓ **Exchanger**: Zwei Threads treffen sich und tauschen Daten aus.

Semaphor

Ein Semaphor stellt sicher, dass nur eine bestimmte Anzahl von Threads auf ein Programmstück zugreift, und zählt damit zur Technik der Sperrmechanismen. Semaphoren wurden 1968 vom niederländischen Informatiker Edsger Wybe Dijkstra eingeführt, also zehn Jahre vor Hoares Monitoren.

Es lassen sich zwei Typen von Semaphoren unterscheiden:

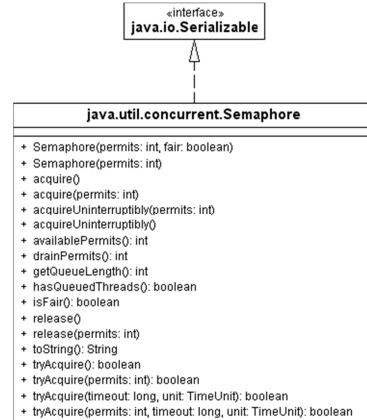
- ✓ **Binäre Semaphoren** lassen höchstens einen Thread auf ein Programmstück zu. Das bekannte Paar `await() / signal()` bei Condition beziehungsweise `wait() / notify()` von Object bietet sich für binäre Semaphoren an.
- ✓ **Allgemeine Semaphoren** erlauben eine begrenzte Anzahl an Threads in einem kritischen Abschnitt. Das Semaphor verwaltet intern eine Menge sogenannter Erlaubnisse (engl. **permits**).

Die Klasse Semaphore

Für allgemeine Semaphoren mit einer maximalen Anzahl Threads im Programmstück deklariert die Java-Bibliothek die Klasse `java.util.concurrent.Semaphore`.

Die wichtigen Eigenschaften der `Semaphore`-Klasse sind der Konstruktor und die Methoden zum Betreten und Verlassen des kritischen Abschnitts. Intern vermerkt das Semaphor jedes Betreten und lässt Threads warten, wenn das gesetzte Maximum erreicht ist, bis ein anderer Thread das Programmsegment verlässt.

```
class java.util.concurrent.Semaphore
    implements Serializable
```



UML-Diagramm für Semaphore

- ✓ `Semaphore(int permits)`
Das neue Semaphor, das bestimmt, wie viele Threads in einem Block sein dürfen.
- ✓ `void acquire()`
Versucht, in den kritischen Block einzutreten. Wenn der gerade belegt ist, wird gewartet. Vermindert die Menge der Erlaubnisse um eins.
- ✓ `void release()`
Verlässt den kritischen Abschnitt und legt eine Erlaubnis zurück.

Allgemeine Semaphoren vereinfachen das Konsumenten-Produzenten-Problem, da eine bestimmte Anzahl von Threads in einem Block erlaubt ist. Die verbleibende Größe des Puffers ist somit automatisch die maximale Anzahl von Produzenten, die sich parallel im Einfügeblock befinden können.

Unser Beispiel soll mit einem Semaphor arbeiten, das nur zwei Threads gleichzeitig in den kritischen Abschnitt lässt:

com/tutego/insel/thread/concurrent/SemaphoreDemo.java, Teil 1

```
package com.tutego.insel.thread.concurrent;

import java.util.concurrent.Semaphore;

public class SemaphoreDemo
{
    static Semaphore semaphore = new Semaphore( 2 );
```

Der kritische Abschnitt besteht aus zwei Operationen: einer Ausgabe auf dem Bildschirm und einer Wartezeit von zwei Sekunden. Er ist in einem `Runnable` eingebettet:

com/tutego/insel/thread/concurrent/SemaphoreDemo.java, Teil 2

```
static Runnable r = new Runnable() {
    @Override public void run() {
        while ( true ) {
            try
            {
                semaphore.acquire();
                try
                {
                    System.out.println( "Thread=" + Thread.currentThread().getName() +
                        ", Available Permits=" + semaphore.availablePermits() );
                };
                TimeUnit.SECONDS.sleep( 2 );
            }
            finally
            {
                semaphore.release();
            }
        }
        catch ( InterruptedException e )
        {
            e.printStackTrace();
            Thread.currentThread().interrupt();
            break;
        }
    }
};
```

Der kritische Abschnitt beginnt mit dem `acquire()` und endet mit `release()`. Wichtig ist die richtige Ausnahmebehandlung. Fünf Dinge müssen beachtet werden und formen den Quellcode:

- ✓ Das `acquire()` kann eine `InterruptedException` auslösen, was eine geprüfte Ausnahme ist. Wir müssen sie folglich behandeln.
- ✓ Da `acquire()` sowie `sleep()` eine `InterruptedException` auslösen können, ist die Frage, wie damit umzugehen ist. Im Prinzip signalisiert die Ausnahme die Bitte, das Warten zu beenden. Das wollen wir berücksichtigen, und wir brechen daher aus der Endlosschleife aus.
- ✓ Immer dann, wenn ein `acquire()` erfolgreich war, muss auch ein `release()` folgen. Das `release()` ist im `finally` sehr gut aufgehoben, denn wir wollen in jedem Fall die Semaphoren wieder freigeben, auch wenn irgendwie eine andere `RuntimeException` auftauchen sollte.
- ✓ Ein `release()` darf nur dann erfolgen, wenn es ein zugehöriges `acquire()` gibt. Eine Programmierung wie `try { semaphore.acquire(); ... } finally { semaphore.release(); }` ist unsicher, denn wenn ein `acquire()` wirklich eine Ausnahme erzeugt, wird fälschlicherweise ein `release()` ausgelöst.
- ✓ Ein `release()` erzeugt keine Ausnahme, daher muss auch nichts behandelt werden.

Drei Threads sollen sich koordinieren:

com/tutego/insel/thread/concurrent/SemaphoreDemo.java, Teil 3

```
public static void main( String[] args )
{
    new Thread( r ).start();
    new Thread( r ).start();
    new Thread( r ).start();
}
```

Nach dem Starten ist gut zu beobachten, wie jeweils zwei Threads im Abschnitt sind (eine Leerzeile symbolisiert die Wartezeit):

Thread=Thread-0, Available Permits=1
 Thread=Thread-1, Available Permits=0

Thread=Thread-2, Available Permits=0
 Thread=Thread-0, Available Permits=0

Thread=Thread-2, Available Permits=0
 Thread=Thread-0, Available Permits=0

Fair und unfair

In der Ausgabe ist zu sehen, dass Thread 0, 1 und 2 zwar ihre Aufgaben ausführen können, aber plötzlich eine Sequenz 0, 2, 0 entsteht. Unser Gerechtigkeitssinn sagt uns jedoch, dass Thread 1 wieder an die Reihe kommen müsste. Wie ist das möglich? Die Antwort lautet, dass das `acquire()` nicht berücksichtigt, wer am längsten wartet, sondern dass es sich aus der Liste der Wartenden einen beliebigen Thread auswählt (wir kennen das von `notify()` her und dem Betreten eines `synchronized-Blocks`). Um ein faires Verhalten zu realisieren, wird die Fairness einfach über den Konstruktor von `Semaphore` angegeben. Ändern wir im Programm folgende Zeile:

```
static Semaphore semaphore = new Semaphore( 2, true );
```

Nun bekommen wir eine Ausgabe wie die folgende:

Thread=Thread-0, Available Permits=1
 Thread=Thread-1, Available Permits=0
 Thread=Thread-2, Available Permits=0
 Thread=Thread-0, Available Permits=0
 Thread=Thread-1, Available Permits=0
 Thread=Thread-2, Available Permits=0
 Thread=Thread-0, Available Permits=0
 Thread=Thread-1, Available Permits=0

Barrier und Austausch

Der Punkt, an dem alle Threads zusammenkommen und sich die Ergebnisse zusammenlegen lassen, heißt auf Englisch **barrier**. Seit Java 5 gibt es die Klasse `CyclicBarrier` im Paket `java.util.concurrent`, die eine solche Barriere realisiert. Der Vorteil gegenüber `join()` besteht in der Tatsache, dass der für die Abarbeitung verwendete Thread nicht enden muss - und ein Thread im Thread-Pool endet eigentlich nicht -, sondern dass er mit `await()` sein "Bin-fertig"-Signal geben kann. Das folgende Beispiel zeigt anhand eines parallelen Summierers die Funktionsweise:

com/tutego/insel/thread/concurrent/ArraySummer.java, ArraySummer

```
public class ArraySummer
{
    public static void main( String[] args )
    {
        int[] array = new int[ 1000 ];
        Random r = new Random();
        for ( int i = 0; i < array.length; i++ )
            array[ i ] = Math.abs( r.nextInt() / 2 );

        parallSummer( array );
    }

    public static void parallSummer( int[] array )
    {
        int prozessors = 2; // Runtime.getRuntime().availableProcessors();

        final List<Long> longs = new ArrayList<Long>();

        Runnable merger = new Runnable() {
            @Override public void run()
            {
                long sum = 0;
                for ( long i : longs )
                    sum += i;
                System.out.println( sum );
            }
        };

        CyclicBarrier barrier = new CyclicBarrier( prozessors, merger );

        for ( int part = 0; part < prozessors; part++ )
            new Thread( new AtomarSummer( barrier, array, prozessors, part,
                                         longs ) ).start();
    }
}
```

com/tutego/insel/thread/concurrent/AtomarSummer.java, AtomarSummer

```
class AtomarSummer implements Runnable
{
    private final CyclicBarrier barrier;
    private final int[] array;
    private final List<Long> longs;
    private int start, end;

    public AtomarSummer( CyclicBarrier barrier, int[] array, int maxPart,
                        int currentPart, List<Long> longs )
    {
        this.barrier = barrier;
        this.array   = array;
        this.longs   = longs;

        start = (int) ((double) array.length / maxPart * currentPart);
        end   = (int) ((double) array.length / maxPart * (currentPart + 1) - 1);
    }

    @Override public void run()
    {
        long sum = 0;

        for ( int i = start; i < end; i++ )
            sum += array[ i ];

        longs.add( sum );
    }
}
```

```

try
{
    barrier.await();
}
catch ( InterruptedException e )
{
    e.printStackTrace();
}
catch ( BrokenBarrierException e )
{
    e.printStackTrace();
}
}
}
}

```

Stop and go mit Exchanger

Die generische Klasse `java.util.concurrent.Exchanger` dient ebenfalls dem Zusammenkommen von Threads, die jedoch bei ihrem **Rendezvous** Daten austauschen können. Ein üblicher Fall betrifft das Füllen von Puffern, etwa wenn ein Thread Daten vom Datensystem liest und ein anderer Thread die Daten über das Netzwerk weiterschickt. Ein Dateisystem-Thread füllt den Puffer, und wenn er komplett gefüllt ist, trifft er sich mit einem anderen Netzwerk-Thread, dem er den vollen Puffer gibt und von dem er wieder einen leeren Puffer empfängt. Der Netzwerk-Thread kann dann den Inhalt des Puffers wieder "verbrauchen" und der erste Thread den Puffer wieder mit Daten vom Dateisystem füllen.

5.8 Atomare Operationen und frische Werte mit volatile

Die JVM arbeitet bei den Ganzzahl-Datentypen kleiner gleich `int` intern nur mit einem `int`. Doch obwohl es auch für die 64-Bit-Datentypen `long` und `double` einzelne Bytecode-Befehle gibt, ist nicht gesichert, dass die Operationen auf diesen Datentypen unteilbar sind, da etwa eine 64-Bit-Zuweisung sich aus zwei 32-Bit-Zuweisungen zusammensetzen lässt. Es kann also passieren, dass ein Thread mitten in einer `long`- oder `double`-Operation von einem anderen Thread verdrängt wird. Greifen zwei Threads auf die gleiche 64-Bit-Variable zu, so könnte möglicherweise der eine Thread eine Hälfte schreiben und der andere Thread die andere.

Beispiel: Zwei Threads versuchen gleichzeitig, die Variable `l` vom Typ `long` zu ändern (aus Gründen der Übersichtlichkeit ist ein Leerzeichen nach dem ersten 32-Bit-Block in der hexadezimalen Notation eingefügt):

```
l = 0x00000000 00000000          l = 0xFFFFFFFF FFFFFFFF
```

Schreibt der erste Thread den ersten Teil der `long`-Variablen mit `00000000`, und findet dann ein Kontextwechsel statt, sodass der zweite Thread die Variable komplett mit `FFFFFFF FFFFFFFF` initialisiert, gibt es nach dem erneuten Umschalten ein Problem, weil dann der erste Thread seine zweite Hälfte schreiben wird und die Belegung der Variablen `FFFFFFF 00000000` ist. Insgesamt gibt es vier denkbare Belegungen durch das Scheduling beim Kontextwechsel:

```
l = 0x00000000 00000000
l = 0xFFFFFFFF FFFFFFFF
l = 0x00000000 FFFFFFFF
l = 0xFFFFFFFF 00000000
```

Der Modifizierer `volatile` bei Objekt-/Klassenvariablen

Um dies zu vermeiden, können die Objekt- und Klassenvariablen mit dem Modifizierer `volatile` deklariert werden. Die Zugriffsoperationen werden auf diesen Variablen dann atomar ausgeführt. Der Modifizierer ist bei lokalen Variablen nicht gestattet, da sie auf dem Stapel liegen und für andere Threads nicht zugänglich sind. Achtung: Auch mit `volatile` sind Operationen wie `i++` natürlich noch nicht atomar, da `i++` aus Grundoperationen besteht: Lesen von `i`, Erhöhen um eins und Schreiben von `i`.

Zwischenspeicherung untersagen

`volatile` beugt zusätzlich einem anderen Problem vor: Während der Berechnung könnte die Laufzeitumgebung Inhalte von Variablen im Prozessorspeicher (zum Beispiel Register) zwischengespeichert haben. Das passiert etwa, wenn in einer Schleife ein Wert immer hochgezählt wird, der in einer Objektvariablen gespeichert wird.

Eine Laufzeitumgebung könnte den (teureren) Zugriff auf Objektvariablen optimieren, indem sie zuerst den Variableninhalt in eine (schnelle) interne lokale Variable kopiert und anschließend, nach einer Berechnung, das Ergebnis zurückgespeichert. Nehmen wir eine Objektvariable `cnt` und folgende Schleife an:

```
for ( int cnt = 0; cnt < 1000000; cnt++ )
    cnt++;
```

Eine optimierende Laufzeitumgebung könnte nun auf die Idee kommen, nicht bei jedem Schleifendurchlauf den Speicher der Objektvariable `cnt` auszulesen und zu beschreiben, sondern nur am Ende. Etwa so:

```
int i = cnt;
for ( int i = 0; i < 1000000; i++ )
    i++;
cnt = i;
```

Wenn die Umgebung `cnt` zuerst in eine interne lokale Variable kopiert, dann die gesamte Schleife ausführt und erst anschließend den internen Wert nach `cnt` zurückkopiert, hätten wir viel Zeit gespart, denn der Zugriff auf eine lokale Variable, die sich im Register des Prozessors aufhalten kann, ist wesentlich schneller als der Zugriff auf eine Objektvariable. Dies ist im Übrigen eine beliebte Strategie, um die Performance eines Programms zu steigern. Mit dieser internen Optimierung kommt es jedoch zu schwer kontrollierbaren Nebeneffekten, und Änderungen am Wert von `cnt` sind nicht sofort für andere Threads sichtbar, denen damit die Information über die einzelnen Inkrementschritte fehlt.

Oder ein anderes Beispiel:

```
int i = cnt;
Thread.sleep( 10000 );
int j = cnt;
```

Verändert ein anderer Thread während des Wartens die Variable `cnt`, könnte `j = i` sein, wenn die Laufzeitumgebung die Variable `j` nicht frisch mit dem Wert aus `cnt` initialisiert. Nebenläufige Programmierung und Programmoptimierung passen schwer zusammen!

Auch hier hilft wieder das ungewöhnliche Schlüsselwort. Ist die Variable mit `volatile` gekennzeichnet, wird das Ergebnis nicht im Zwischenspeicher belassen, sondern ständig aktualisiert. Die parallelen Threads sehen somit immer den korrekten Variablenwert, da er vor jeder Benutzung aus dem Speicher gelesen und nach einer Änderung sofort wieder zurückgeschrieben wird. Sehr gut ist das für `boolean`-Werte, die Flags anzeigen, die andere Threads zur Synchronisation beobachten. Ein Array, das mit `volatile` markiert ist, hat alle Zugriffe auf die Elemente `volatile`, denn auch hier könnte die JVM die Elemente aus Performance-Gründen zwischenspeichern. Wohlgernekt bleiben nur die Werte aktuell; nicht-atomare Operationen wie `cnt++` sind damit immer noch nicht atomar.

Das Paket `java.util.concurrent.atomic`

Während `volatile` das Lesen und Schreiben auf den 64-Bit-Datentypen atomar macht, ist eine Operation wie `cnt++` oder `cnt > 12` nicht atomar. Kurz vor dem Erhöhen oder dem Vergleich kann ein Thread-Wechsel stattfinden und `cnt` geändert werden, sodass der Vergleich einen anderen Ausgang findet. Die Rettung ist ein synchronisierter Block, wobei jedoch eine Synchronisationsvariable nötig ist.

Seit Java 5 helfen hier die Klassen des Pakets `java.util.concurrent.atomic`. Ein Beispiel für einen veränderbaren Behälter ist `AtomicInteger`, der Operationen wie `getAndIncrement()` anbietet, die den Wert des Behälters atomar erhöhen.

Beispiel: Eine öffentliche Klasse bietet über eine statische Methode eine ID (im Bereich `long`) an.

`com/tutego/insel/thread/atomic/Id.java`

```
public class Id
{
    private static final AtomicLong id = new AtomicLong();

    private Id() { /* Empty */ }

    public long next()
    {
        return id.getAndIncrement();
    }
}
```

Insgesamt befinden sich im Paket `java.util.concurrent.atomic` folgende Klassen für elementare Datentypen: `AtomicBoolean`, `AtomicInteger`, `AtomicLong` - es gibt weder `AtomicShort` noch `AtomicByte`. Da diese Objekte alle veränderbar sind, eignen sie sich nicht unbedingt als Ersatz für immutable Wrapper-Objekte wie `Integer`, insbesondere nicht in allgemeinen Assoziativspeichern oder Mengen; sie deklarieren auch gar kein eigenes `equals()` und `hashCode()`.

Für Felder stehen bereit: `AtomicIntegerArray`, `AtomicLongArray`. Die Klassen `AtomicIntegerFieldUpdater<T>` und `AtomicLongFieldUpdater<T>` gehen über Reflection an Attribute heran, die `volatile` sind, und ermöglichen atomare Operationen. Dazu kommen noch einige Klassen zum atomaren Arbeiten mit Referenzen: `AtomicReference`, `AtomicReferenceArray<E>`, `AtomicReferenceFieldUpdater<T,V>`, `AtomicMarkableReference<V>` (assoziiert ein Objekt mit einem Flag, das atomar geändert werden kann) und `AtomicStampedReference<V>` (assoziiert ein Objekt mit einer Ganzzahl).

5.9 Threads in einer Thread-Gruppe

Wenn wir einen Thread erzeugen, gehört dieser automatisch zu einer Gruppe, die durch ein `ThreadGroup`-Objekt repräsentiert wird. Die Verwaltung ist Aufgabe der Laufzeitumgebung. Die Gruppenzugehörigkeit bekommt jeder Thread bei seiner Erzeugung zugeteilt; wir können sie aber beeinflussen: Entweder geben wir die Gruppe explizit an, wofür es einen passenden Konstruktor gibt, oder der Thread wird automatisch der Gruppe jenes Threads zugeordnet, der ihn erzeugt hat. Da Thread-Gruppen baumartig organisiert sind, kann jede Thread-Gruppe wiederum weitere Untergruppen besitzen. Die Wurzel dieses Baums bildet für Benutzer-Threads die Gruppe `main`. Über diese Gruppenzugehörigkeit lassen sich Threads leicht verwalten, da sich beispielsweise alle Threads einer Gruppe gleichzeitig stoppen lassen oder die Priorität geändert werden kann. Von Applets erzeugte Threads gehören automatisch zu speziellen Untergruppen, sodass sie die Laufzeitumgebung nicht negativ beeinflussen.

Aktive Threads in der Umgebung

Es gibt zwei Möglichkeiten, um herauszufinden, welche Threads gerade im System laufen. Der erste Weg führt über die Klasse `Thread`, der zweite über die Klasse `ThreadGroup`.

```
class java.lang.Thread
implements Runnable
```

- ✓ `static int activeCount()`
Liefert die Anzahl der noch nicht beendeten Threads in der Gruppe des aktiven Threads.
- ✓ `static int enumerate(Thread[] tarray)`
Jeder Thread in der gleichen Gruppe wie der aktive Thread und alle Threads in den Untergruppen dieser Gruppe werden in das Feld kopiert. Der `SecurityManager` überprüft, ob wir dies überhaupt dürfen, und kann eine `SecurityException` auslösen. Die Rückgabe ist die Anzahl kopierter Elemente im Feld.

Etwas über die aktuelle Thread-Gruppe herausfinden

Das folgende Programm nutzt die Methode `getThreadGroup()`, um das `ThreadGroup`-Objekt des aktuellen Threads zu bekommen. Anschließend verwendet es `getParent()`, um die Thread-Gruppen ebenenweise nach oben zu durchlaufen und somit das Vater-Objekt für alle Threads auszulesen:

```
com/tutego/insel/thread/group>ShowThreadsInMain.java
package com.tutego.insel.thread.group;

public class ShowThreadsInMain
{
    public static void main( String[] args )
    {
        ThreadGroup top = Thread.currentThread().getThreadGroup();

        while ( top.getParent() != null )
            top = top.getParent();

        showGroupInfo( "  ", top );
    }

    public static void showGroupInfo( String indent, ThreadGroup group )
    {
        Thread[] threads = new Thread[ group.activeCount() ];
        group.enumerate( threads, false );
        System.out.println( group );

        for ( Thread t : threads )
            if ( t != null )
                System.out.printf( "%s%s -> %s ist %sDaemon-Thread%n",
                    indent, group.getName(), t, t.isDaemon() ? "" : "kein " );

        ThreadGroup[] activeGroup = new ThreadGroup[ group.activeGroupCount() ];
        group.enumerate( activeGroup, false );

        for ( ThreadGroup g : activeGroup )
            showGroupInfo( indent + indent, g );
    }
}
```

Dann wird mit der schon bekannten `enumerate()`-Methode eine Liste erstellt und eine Auflistung aller Untergruppen auf den Bildschirm ausgegeben. Die Anzahl der Threads in der Gruppe ergibt sich mit `activeCount()`. Die Methoden `enumerate()` sowie `activeCount()` beziehen sich hier auf Objektmethoden einer Thread-Gruppe und nicht mehr auf Klassenmethoden von `Thread`.

```
java.lang.ThreadGroup[name=system,maxpri=10]
    system -> Thread[Reference Handler,10,system] is Daemon
    system -> Thread[Finalizer,8,system] is Daemon
    system -> Thread[Signal Dispatcher,9,system] is Daemon
    system -> Thread[Attach Listener,5,system] is Daemon
java.lang.ThreadGroup[name=main,maxpri=10]
    main -> Thread[main,5,main] is no Daemon
```

```
class java.lang.Thread
implements Runnable
```

- ✓ `static Thread currentThread()`
Liefert eine Referenz auf den aktuell laufenden Thread.
- ✓ `final ThreadGroup getThreadGroup()`
Liefert die Thread-Gruppe, zu der der Thread gehört. Wenn es den Thread schon nicht mehr gibt, liefert die Methode `null`.

```
class java.lang.ThreadGroup
    implements Thread.UncaughtExceptionHandler
```

- ✓ final ThreadGroup getParent()
 Liefert die Obergruppe der Thread-Gruppe.
- ✓ int activeCount()
 Liefert die Anzahl aktiver Threads in der Gruppe inklusive aller Untergruppen.
- ✓ int activeGroupCount()
 Liefert die Anzahl aktiver Untergruppen in der Gruppe.
- ✓ int enumerate(Thread[] threadList)
 Kopiert eine Referenz auf jeden aktiven Thread der Gruppe und auf alle Threads in ihren Untergruppen in das Array.
- ✓ int enumerate(Thread[] threadList, boolean recurse)
 Kopiert eine Referenz auf jeden aktiven Thread der Gruppe in das Array. Ist `recurse` gleich `true`, werden auch Referenzen auf die Threads der Untergruppen mit kopiert.
- ✓ int enumerate(ThreadGroup[] groupList)
 Kopiert Referenzen auf die Untergruppen, die mindestens einen aktiven Thread enthalten, inklusive aller Unteruntergruppen in das Array.
- ✓ int enumerate(ThreadGroup[] groupList, boolean recurse)
 Kopiert Referenzen auf jede aktive Untergruppe inklusive aller Unteruntergruppen, wenn `recurse` gleich `true` ist.

Ein Array der passenden Größe müssen wir zunächst anlegen. Dies geschieht am besten mit `activeCount()`. Ist das Array für die Threads zu klein, werden die überzähligen Threads nicht mehr in das Array kopiert.

Threads in einer Thread-Gruppe anlegen

Wollen wir Threads in einer Gruppe anlegen, dann müssen wir zunächst ein `ThreadGroup`-Objekt erzeugen. Dazu bietet die Klasse zwei Konstruktoren. Im ersten müssen wir lediglich den Namen der Gruppe angeben. Da eine Thread-Gruppe, die mehrere Threads zusammenfasst, wiederum Mitglied eines `ThreadGroup`-Objekts sein kann, existiert ein zweiter Konstruktor, der die übergeordnete Gruppe bestimmt. Ohne diese Angabe wird die Thread-Gruppe des aktuellen Threads zum Vater der neuen Gruppe.

```
class java.lang.ThreadGroup
    implements Thread.UncaughtExceptionHandler
```

- ✓ ThreadGroup(String name)
 Erzeugt eine Thread-Gruppe namens `name`.
- ✓ ThreadGroup(ThreadGroup parent, String name)
 Erzeugt eine neue Thread-Gruppe, deren Vater `parent` ist.

Die Gruppe haben wir angelegt, nun fehlen uns noch die Threads. Diese können nur bei ihrer Erzeugung in die Gruppe aufgenommen werden. Dafür bietet uns die `Thread`-Klasse spezielle Konstruktoren, um den neuen Thread einer bestimmten Gruppe hinzuzufügen. Folgende Zeilen reichen aus, um die drei Comic-Helden in eine Gruppe zu stecken:

```
ThreadGroup group = new ThreadGroup( "Disney Family" );
Thread thread1 = new Thread( group, "Donald" );
Thread thread2 = new Thread( group, "Daisy" );
Thread thread3 = new Thread( group, "Micky" );
```

Die Zugehörigkeit eines Threads zu einer Gruppe lässt sich nachträglich nicht mehr ändern.

java.lang.ThreadGroup
+ ThreadGroup(name: String) + ThreadGroup(parent: ThreadGroup, name: String) + activeCount(): int + activeGroupCount(): int + allowThreadSuspension(b: boolean): boolean + checkAccess() + destroy() + enumerate(list: ThreadGroup[], recurse: boolean): int + enumerate(list: ThreadGroup[]): int + enumerate(list: Thread[]): int + enumerate(list: Thread[], recurse: boolean): int + getMaxPriority(): int + getName(): String + getParent(): ThreadGroup + interrupt() + isDaemon(): boolean + isDestroyed(): boolean + list() + parentOf(g: ThreadGroup): boolean + resume() + setDaemon(daemon: boolean) + setMaxPriority(pri: int) + stop() + suspend() + toString(): String + uncaughtException(t: Thread, e: Throwable)

Klassendiagramm für `ThreadGroup`

```
class java.lang.Thread
implements Runnable
```

- ✓ Thread(ThreadGroup group, Runnable target, String name)
 Erzeugt ein neues Thread-Objekt mit dem Namen name in der Gruppe group und dem Runnable-Objekt target, das die auszuführende run()-Methode enthält. Ist die Gruppe null, so wird der Thread in der Gruppe erzeugt, in der auch der erzeugende Thread liegt. Ist target gleich null, hat das Thread-Objekt selbst das passende run().
- ✓ Thread(ThreadGroup group, Runnable target)
 Erzeugt ein neues Thread-Objekt. Der Konstruktor verweist auf Thread(parent, target, "Thread-" + n), wobei n eine Ganzzahl ist, die mit nextThreadNum() erfragt wird. Bei jedem erzeugten Thread wird n inkrementiert.
- ✓ Thread(ThreadGroup group, String name)
 Erzeugt ein neues Thread-Objekt in der Gruppe group. Entspricht dem Konstruktor Thread(group, null, name).

Das nachfolgende Beispiel zeigt die Erstellung und Nutzung einer eigenen Thread-Gruppe:

```
com/tutego/insel/thread/group/ThreadInThreadGroup.java
package com.tutego.insel.thread.group;

public class ThreadInThreadGroup
{
    static public void main( String[] args )
    {
        ThreadGroup group = new ThreadGroup( "Helden" );

        new OwnThread( group, "Darkwing Duck" ).start();
        new OwnThread( group, "Kikky" ).start();

        Thread[] threadArray = new Thread[ group.activeCount() ];
        // Fill array with all Threads of the group
        group.enumerate( threadArray );
        // List array
        for ( Thread t : threadArray )
            System.out.println( t );
        System.out.println( "--" );
        group.list();
    }

    class OwnThread extends Thread
    {
        public OwnThread( ThreadGroup group, String name ) {
            super( group, name );
        }

        @Override public void run()
        {
            for ( int i = 0; i < 3; i++ )
            {
                System.out.println( getName() + ": Ich bin der Schrecken, "+
                    "der die Nacht durchflattert" );
            }
        }
    }
}
```

Läuft das Programm, produziert es eine Ausgabe folgender Art:

```
Darkwing Duck: Ich bin der Schrecken, der die Nacht durchflattert
Darkwing Duck: Ich bin der Schrecken, der die Nacht durchflattert
Darkwing Duck: Ich bin der Schrecken, der die Nacht durchflattert
Kikky: Ich bin der Schrecken, der die Nacht durchflattert
Kikky: Ich bin der Schrecken, der die Nacht durchflattert
Kikky: Ich bin der Schrecken, der die Nacht durchflattert
Thread[Darkwing Duck,5,Helden]
Thread[Kikky,5,]
--
java.lang.ThreadGroup [name=Helden,maxpri=10]
```

Wesentlich einfacher ist die Testausgabe einer Thread-Gruppe mit der Objektmethode `list()`. Diese Methode ergibt aber nur zu Testzwecken Sinn.

```
class java.lang.ThreadGroup
implements Thread.UncaughtExceptionHandler
```

- ✓ `void list()`

Schreibt eine Liste mit Informationen über die Thread-Gruppe auf die Standardausgabe.

Methoden von Thread und ThreadGroup im Vergleich

Einige Methoden, die ein Thread besitzt, lassen sich auf Gruppen übertragen, wobei es zum Teil Bedeutungsunterschiede gibt. So suggeriert vielleicht `setMaxPriority()`, dass die Priorität aller Threads in der Gruppe verändert wird, aber das ist nicht so. `setMaxPriority()` und `getMaxPriority()` beziehen sich auf die interne Variable `maxPriority` aus der Thread-Gruppe - die Threads aus der Gruppe werden von den beiden Methoden überhaupt nicht beeinflusst.

```
class java.lang.ThreadGroup
implements Thread.UncaughtExceptionHandler
```

- ✓ `final void setMaxPriority(int priority)`

Setzt die höchstens einstellbare Priorität für Threads aus der `ThreadGroup`. Zum Beispiel: Hat ein Thread die Priorität 7 und wird danach die `maxPriority` seiner Gruppe auf 3 gesetzt, dann wird die Priorität des Threads bei einem folgenden `setPriority(i)` (wobei `i` zwischen 3 und 7 sein kann) auch nur auf 3 gesetzt. Existierende Threads ändern ihre Priorität nicht.

- ✓ `final int getMaxPriority()`

Gibt zurück, wie hoch die Priorität eines Threads aus einer `ThreadGroup` höchstens gesetzt werden kann. Sie gibt nicht zurück, wie hoch die höchste Priorität aller Prioritäten der Threads aus der Thread-Gruppe ist!

Wird von einem Thread ein neuer Thread erzeugt, erbt der neue Thread üblicherweise die Priorität. Ist die Priorität des erzeugenden Threads aber höher als die maximale Priorität der Gruppe, dann erhält der neue Thread trotzdem nur den maximalen Prioritätswert der Gruppe zuerkannt.



Veraltete Methoden und Ähnliches

Einige Methoden sind veraltet und sollten nicht mehr verwendet werden. Dazu zählen `allowThreadSuspension(boolean)`, `resume()`, `stop()` und `suspend()`. Außer `allowThreadSuspension()` sind die anderen Methoden auch in der Klasse `Thread` veraltet. Immer noch aktuell und nützlich sind die `ThreadGroup`-Methoden `destroy()` und `interrupt()`.

```
class java.lang.ThreadGroup
    implements Thread.UncaughtExceptionHandler
```

- ✓ `final void destroy()`
Entfernt die Thread-Gruppe und alle Untergruppen. Die darin enthaltenen Threads müssen beendet worden sein. Eine `IllegalThreadStateException` wird ausgelöst, wenn die Gruppe nicht leer ist oder wenn die Gruppe schon entfernt wurde.
- ✓ `final void interrupt()`
Unterbricht alle Threads in der Gruppe und den Untergruppen.
- ✓ `final void setDaemon(boolean daemon)`
Ändert den Status der Thread-Gruppe. Diese Gruppeneigenschaft ist orthogonal zur Dämon-Eigenschaft einzelner Threads. Die Dämon-Gruppe wird entfernt, falls der letzte Thread stoppt oder die letzte Thread-Gruppe entfernt wurde.
- ✓ `final boolean isDaemon()`
Testet, ob die Thread-Gruppe eine Dämon-Gruppe ist. Eine Dämon-Thread-Gruppe wird automatisch entfernt, wenn der letzte Thread gestoppt oder zerstört wurde.

Kein join in einer Thread-Gruppe

Eine Thread-Gruppe kennt keine `join()`-Methode wie ein Thread, sodass es über eine einzelne Methode nicht möglich ist, das Ende aller Threads in einer Gruppe zu erkennen. Doch die Thread-Gruppe besitzt die nützliche Methode `activeCount()`, über die etwa die Anzahl aktiver Threads - inklusive der Threads in Untergruppen - in Erfahrung gebracht werden kann. Damit ist Folgendes denkbar:

```
void join( ThreadGroup tg ) throws InterruptedException
{
    synchronized( tg )
    {
        while ( tg.activeCount() > 0 )
            tg.wait( 10 );
    }
}
```

Das alles kommt noch in einen `synchronized`-Block und in ein `wait()`, damit die Schleife nicht zu viel Zeit kostet.

5.10 Einen Abbruch der virtuellen Maschine erkennen

Läuft ein Java-Programm, so kann der Benutzer es jederzeit beenden, indem er die virtuelle Maschine stoppt. Das kann er zum Beispiel durch die Tastenkombination **Strg** **C** auf der Konsole oder auch durch Abbruchsignale vornehmen. Ein plötzlicher Programmabbruch kann aber für das Programm sehr unsicher sein, und zwar aus den gleichen Gründen wie bei einem unvermittelten Abbruch durch die `stop()`-Methode der Klasse `Thread`. Daher bietet Java durch einen **Shutdown-Hook** die Möglichkeit, das Abbruchsignal zu erkennen und sauber Ressourcen freizugeben. Das Teilwort **Hook** erinnert an einen Haken, der sich ins System einhängt, um Informationen abzugreifen. Der Hook ist ein initialisierter, aber noch nicht gestarteter Thread.

Beispiel

Eine Endlosschleife schmort im eigenen Saft. Der eingefügte Hook reagiert auf das Ende der virtuellen Maschine. Das Programm muss auf der Konsole beendet werden, da die meisten Entwicklungsumgebungen das **Strg** **C** nicht an die Java-Umgebung weiterleiten.

`com/tutego/insel/thread/ThatsMyEnd.java, main()`

```
Runtime.getRuntime().addShutdownHook( new Thread() {
    @Override public void run() {
        System.out.println( "Jedes Leben endet tödlich." );
    }
} );
 JOptionPane.showConfirmDialog( null, "Ende?" );
System.exit( 0 );
```

Zusätzlich zum bewussten Abbruch wird der Thread auch immer dann ausgeführt, wenn das Programm normal beendet wird. Es können mehrere Threads als Shutdown-Hooks installiert sein. Wenn sich dann die JVM beendet, werden die Threads in beliebiger Reihenfolge abgearbeitet.

Wird unter Windows der Prozess mit dem Task-Manager beendet, läuft der Shutdown-Hook nicht!



`class java.lang.Runtime`

✓ `void addShutdownHook(Thread hook)`

Startet den angegebenen Thread, wenn die JVM beendet wird. Der Thread kann keinen neuen Thread unter dem Hook registrieren. Die `run()`-Methode des Threads sollte schnell ablaufen, um das Beenden der JVM nicht länger als nötig aufzuhalten. Es ist nicht möglich (vorgesehen), das bevorstehende Ende der JVM zu verhindern.

✓ `boolean removeShutdownHook(Thread hook)`

Entfernt den angegebenen Hook. Der Rückgabewert ist `true`, falls der Hook registriert war und entfernt werden konnte.

6 Spezielle Streams und Serialisierung

In diesem Kapitel erfahren Sie

- ✓ wie Threads Daten austauschen
- ✓ wie Übertragungsfehler vermieden werden können
- ✓ wie Objektstrukturen gespeichert und gelesen werden
- ✓ wie bestimmte Merkmale in Datenströmen erkannt werden können

Voraussetzungen

- ✓ Streams und Threads

6.1 Kommunikation zwischen Threads mit Pipes

Die Kommunikation zwischen Programmteilen kann auf vielfältige Weise geschehen. Einige Möglichkeiten haben wir bei Threads kennengelernt. Bei getrennten Programmen lässt sich die Kommunikation über Dateien realisieren. Auch Datenströme können von einem Teil geschrieben und vom anderen gelesen werden. Wenn wir jedoch mit Threads arbeiten, wäre eine Kommunikation über Dateien zwar denkbar, aber zu aufwändig. Ein anderes Stromkonzept ist praktisch.

PipedOutputStream und PipedInputStream

Einfacher ist der Austausch der Daten über eine sogenannte **Pipe**. Sie wird über Paare spezieller Stromklassen gebildet:

- ✓ PipedOutputStream, PipedInputStream beziehungsweise
- ✓ PipedWriter, PipedReader

Die PipedXXX-Klassen sind übliche Unterklassen von OutputStream/InputStream und Writer/Reader (im nächsten Beispiel verfolgen wir die Byte-Variante). Wenn dann Threads Daten austauschen wollen, kann ein Produzent sie über `write()` in den Ausgabestrom schreiben, und der andere Thread wird sie dort über `read()` empfangen können.

Natürlich muss ein schreibender Pipe-Strom wissen, wer der Empfänger ist. Daher müssen die Schreib/Lese-Pipes miteinander verbunden werden. Eine Möglichkeit bietet `connect()`.

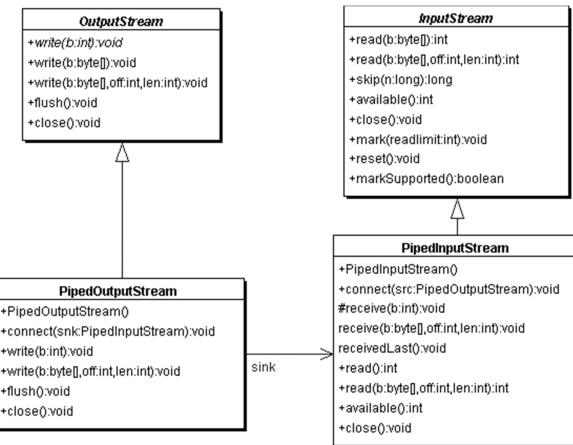
Beispiel: Ein PipedOutputStream soll mit einem PipedInputStream verbunden werden:

```
PipedOutputStream pos = new PipedOutputStream();
PipedInputStream pis = new PipedInputStream();
pos.connect( pis );
// oder pis.connect( pos );
```

Werden jetzt Daten produziert und in den `pos` geschrieben, kommen sie über den `pis` wieder an und können dort konsumiert werden.

Ob wir nun vom `PipedOutputStream` die Methode `connect(PipedInputStream)` nehmen oder vom `PipedInputStream` die Methode `connect(PipedOutputStream)`, ist dabei egal.

Anstatt nach dem Aufbau der Ströme über den Standard-Konstruktor beide mit `connect()` zu verbinden, gibt es eine alternative Lösung: Entweder lässt sich nach dem Erzeugen des `PipedOutputStream` über den Standard-Konstruktor das frische Strom-Objekt in den parametrisierten Konstruktor von `PipedInputStream` übergeben oder eben umgekehrt ein neues `PipedInputStream`-Objekt in den parametrisierten Konstruktor von `PipedOutputStream` legen.



UML-Diagramm mit den Beziehungen zwischen den PipedXXX-Klassen

Beispiel: Verbinde den Eingabe-Stream `pis` mit dem Ausgabe-Stream `pos`:

```
PipedInputStream pis = new PipedInputStream();
PipedOutputStream pos = new PipedOutputStream( pis );
```

Intern

Der Austausch der Daten geschieht über einen internen Puffer, den `PipedInputStream` anlegt. Die Daten, die `PipedOutputStream` über `write()` schreiben soll, gelangen direkt zum Puffer des Eingabestroms. Werfen wir einen kurzen Blick auf die relevanten Teile der Implementierung:

```

class PipedOutputStream extends OutputStream
{
    private PipedInputStream sink;

    public PipedOutputStream( PipedInputStream snk )
        throws IOException
    {
        /* Auskommentierte Fehlerbehandlung */
        sink = snk;
        snk.in = -1;
        snk.out = 0;
        snk.connected = true;
    }

    public void write( int b ) throws IOException
    {
        if ( sink == null )
            throw new IOException( "Pipe not connected" );

        sink.receive( b );
    }
}
  
```

Der `PipedInputStream` nutzt intern einen Puffer von standardmäßig 1.024 Elementen. Das bedeutet: Der Schreibende kann standardmäßig bis zu 1.024 Byte (oder Zeichen bei `PipedReader`) produzieren, bis die Kommunikation stoppen muss. Denn mit dieser Größe ist der Puffer voll und der Produzent blockiert; der Lesende muss den Puffer erst leeren, damit der Konsument weiterarbeiten darf. Umgekehrt bedeutet dies, dass der lesende Thread bei ungenügend vielen Zeichen warten muss, bis der Schreiber die nötige Anzahl hinterlegt hat. Dafür wird intern mittels Thread-Synchronisation gearbeitet. Lebt die andere Seite nicht mehr, gibt es eine `IOException`.

Seit Java 6 lässt sich die Größe über einen Konstruktor wie `PipedInputStream(int pipeSize)`, `PipedInputStream(PipedOutputStream src, int pipeSize)`, `PipedReader(int pipeSize)` oder `PipedReader(PipedWriter src, int pipeSize)` setzen.

PipedWriter und PipedReader

Die Klassen `PipedWriter` und `PipedReader` sind die char-Varianten für die sonst byte-orientierten Klassen `PipedOutputStream` und `PipedInputStream`. Diese sollen uns als Beispiel dienen. Zwei Threads arbeiten miteinander und tauschen Daten aus. Der eine Thread produziert Zufallszahlen, die ein anderer Thread auf dem Bildschirm darstellt:

com/tutego/insel/io/stream/PipeDemo.java, PipeRandomWriter

```
package com.tutego.insel.io.stream;

import java.io.*;

class PipeRandomWriter extends PipedWriter implements Runnable
{
    @Override public void run()
    {
        while ( true ) {
            try
            {
                write( String.format("%f%n", Math.random()) );
                Thread.sleep( 200 );
            }
            catch ( Exception e ) { e.printStackTrace(); }
        }
    }
}
```

Die Klasse ist eine Spezialisierung von `PipedWriter` und produziert in `run()` endlos Zufallszahlen, die in den Ausgabestrom vom `PipedWriter` geschoben werden. Der `PipeRandomReader` wiederum ist ein `PipedReader`, der über einen `BufferedReader` alle geschriebenen Zeilen ausliest:

com/tutego/insel/io/stream/PipeDemo.java, PipeRandomReader

```
class PipeRandomReader extends PipedReader implements Runnable
{
    @Override public void run()
    {
        BufferedReader br = new BufferedReader( this );

        while ( true )
            try
            {
                System.out.println( br.readLine() );
            }
            catch ( IOException e ) { e.printStackTrace(); }
    }
}
```

Das Hauptprogramm erzeugt die beiden spezialisierten Pipes und verbindet sie. Danach werden die Threads gestartet:

com/tutego/insel/io/stream/PipeDemo.java, PipeDemo

```
public class PipeDemo
{
    public static void main( String[] args ) throws Exception
    {
        PipeRandomWriter out = new PipeRandomWriter();
        PipeRandomReader in  = new PipeRandomReader();
        in.connect( out );

        new Thread( out ).start();
        new Thread( in ).start();
    }
}
```

6.2 Prüfsummen

Damit Fehler bei Dateien oder bei Übertragungen von Daten auffallen, werden vor der Übertragung **Prüfsummen** (engl. **checksums**) gebildet und mit dem Paket versendet. Der Empfänger berechnet diese Prüfsumme neu und vergleicht sie mit dem übertragenen Wert. Stimmt der berechnete Wert mit dem übertragenen überein, so war die Übertragung höchstwahrscheinlich in Ordnung. Es sollte ziemlich unwahrscheinlich sein, dass eine Änderung einzelner Bits nicht auffällt. Prüfsummen erkennen auch beschädigte Archive. Pro Datei wird eine Prüfsumme berechnet. Soll die Datei entpackt werden, so errechnen wir wieder die Summe. Ist diese fehlerhaft, muss auch die Datei fehlerhaft sein (wir wollen hier ausschließen, dass zufälligerweise die Prüfsumme fehlerhaft ist, was natürlich ebenfalls der Fall sein kann).

Die Schnittstelle Checksum

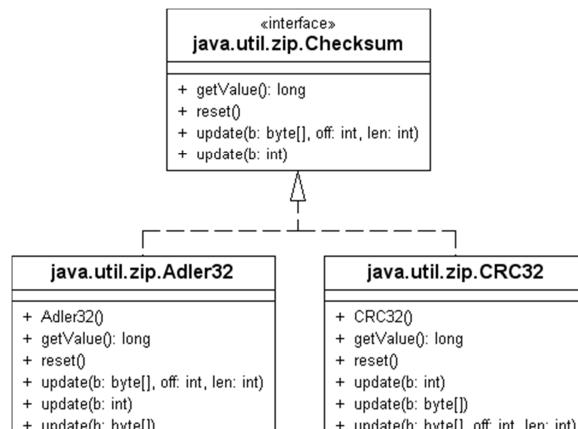
Wir finden Zugang zur Prüfsummenberechnung über die Schnittstelle `java.util.zip.Checksum`, die für ganz allgemeine Prüfsummen steht. Eine Prüfsumme wird entweder für ein Feld oder ein Byte berechnet. `Checksum` liefert die Schnittstelle zum Initialisieren und Auslesen von Prüfsummen, die die konkreten Prüfsummen-Klassen implementieren müssen.

```
interface java.util.zip.Checksum
```

- ✓ `long getValue()`
Liefert die aktuelle Prüfsumme.
- ✓ `void reset()`
Setzt die aktuelle Prüfsumme auf einen Anfangswert.
- ✓ `void update(int b)`
Aktualisiert die aktuelle Prüfsumme mit `b`.
- ✓ `void update(byte[] b, int off, int len)`
Aktualisiert die aktuelle Prüfsumme mit dem Feld.

Die Standardbibliothek bietet bisher zwei Klassen für die Prüfsummenberechnung als Implementierungen von `Checksum`:

- ✓ `java.util.zip.CRC32`: CRC-32 basiert auf einer zyklischen Redundanzprüfung und testet etwa Zip-Archive oder PNG-Grafiken.
- ✓ `java.util.zip.Adler32`: Die Berechnung von CRC-32-Prüfsummen kostet - obwohl sie in C programmiert ist - viel Zeit. Eine Adler-32-Prüfsumme kann wesentlich schneller berechnet werden und bietet eine ebenso geringe Wahrscheinlichkeit, dass Fehler unentdeckt bleiben.



UML-Diagramm der Prüfsummenklassen `Adler32` und `CRC32`

Die Klasse CRC32

Oft sind Polynome die Basis der Prüfsummenberechnung. Eine häufig für Dateien verwendete Prüfsumme ist CRC-32, und das bildende Polynom lautet:

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

Nun lässt sich zu einer 32-Bit-Zahl eine Prüfsumme berechnen, die genau für diese 4 Byte steht. Damit bekommen wir aber noch keinen ganzen Block kodiert. Um das zu erreichen, berechnen wir den Wert eines Zeichens und Xor-verknüpfen den alten CRC-Wert mit dem neuen. Jetzt lassen sich beliebig Blöcke sichern. Die Berechnung ist insgesamt sehr zeitaufwändig, und Adler-32 stellt eine schnellere Alternative dar.

Beispiel

Die Klasse `CRC32` berechnet eine Prüfsumme entweder für ein Byte oder für ein Feld. In aller Kürze sieht ein Programm zur Berechnung von Prüfsummen für Dateien dann folgendermaßen aus (in ist ein `InputStream`-Objekt):

```
CRC32 crc = new CRC32();
byte[] ba = new byte[ (int)in.available() ];
in.read( ba );
crc.update( ba );
in.close();
```

`CRC32` implementiert nicht nur alle Methoden, sondern fügt noch zwei Methoden und natürlich einen Konstruktor hinzu.

```
class java.util.zip.CRC32
implements Checksum
```

- ✓ `CRC32()`
Erzeugt ein neues `CRC32`-Objekt mit der Start-Prüfsumme 0.
- ✓ `long getValue()`
Liefert den `CRC32`-Wert.
- ✓ `void reset()`
Setzt die interne Prüfsumme auf 0.
- ✓ `void update(byte[] b)`
Aktualisiert die Prüfsumme mit dem Feld durch Aufruf von `update(b, 0, b.length)`.
- ✓ `void update(int b)`
Implementiert `update()` aus `Checksum` für ein Byte. Nativ implementiert.
- ✓ `void update(byte[] b, int off, int len)`
Implementiert `update()` aus `Checksum` für ein Feld. Nativ implementiert.

CRC eines Datenstroms berechnen

Eine Möglichkeit, die `CRC32` eines Datenstroms zu berechnen, bestünde darin, einen Datenstrom entgegenzunehmen und anschließend so lange Byte-Folgen auszulesen, bis `available()` null liefert. An diesem Punkt lässt sich mit `update()` jeweils die Prüfsumme korrigieren. Bei großen Dateien ist es sicherlich angebracht, Blöcke einzulesen, die `crc.update(byte[])` verarbeitet. Für diese Aufgabe verfügt die Java-Bibliothek über zwei Filter-Klassen: `CheckedInputStream` und `CheckedOutputStream`. Beide sind Filter, die existierende andere Streams ummanteln und gleichzeitig die Berechnung erledigen:

```
com/tutego/insel/io/CRC32Demo.java, main()
InputStream in = CRC32Demo.class.getResourceAsStream( "/lyrics.txt" );
try
{
    CRC32 crc = new CRC32();
    InputStream cis = new CheckedInputStream( in, crc );

    while ( cis.read() != -1 ) { /* Bis zum Ende */ }

    System.out.printf( "%X", crc.getValue() ); // F9A39CFC
}
catch ( IOException e ) {
    e.printStackTrace();
}
finally {
    try { if ( in != null ) in.close(); }
    catch ( IOException e ) { e.printStackTrace(); }
}
```

Die Adler32-Klasse

Der Algorithmus Adler-32 ist nach seinem Programmierer Mark Adler benannt und im RFC 1950 beschrieben. Die Adler-32-Prüfsumme gilt für 32-Bit-Zahlen und setzt sich aus zwei Summen für ein Byte zusammen. s_1 ist die Summe aller Bytes und s_2 die Summe aller s_1 . Beide Werte werden Modulo 65521 genommen. Am Anfang ist $s_1 = 1$ und $s_2 = 0$. Die Prüfsumme speichert den Wert als $s_2 * 65536 + s_1$ in der MSB-Reihenfolge (**most significant byte first**, Netzwerkreihenfolge).

```
class java.util.zip.Adler32
    implements Checksum
```

- ✓ `Adler32()`
Erzeugt ein neues Adler32-Objekt mit der Start-Prüfsumme 1.
- ✓ `long getValue()`
Liefert den Adler32-Wert.
- ✓ `void reset()`
Setzt die interne Prüfsumme auf 1.

Aus der Schnittstelle `Checksum` implementiert `Adler32` natürlich auch die `update()`-Methoden.

6.3 Persistente Objekte und Serialisierung

Objekte liegen zwar immer nur zur Laufzeit vor, doch auch nach dem Beenden der virtuellen Maschine soll ihre Struktur nicht verloren gehen. Gewünscht ist ein Mechanismus, der die Objektstruktur und Variablenbelegung zu einer bestimmten Zeit sicher (**persistent**) macht und an anderer Stelle wieder hervorholt und die Objektstruktur und Variablenbelegung restauriert. Im gespeicherten Datenformat müssen alle Informationen wie der Objekttyp und der Variablenotyp enthalten sein, um später das richtige Wiederherstellen zu ermöglichen. Da Objekte oftmals weitere Objekte einschließen, müssen auch diese Unterobjekte gesichert werden (schreibe ich eine Liste mit Bestellungen, so ist die Liste ohne die referenzierten Objekte sinnlos). Genau dieser Mechanismus wird auch dann angewendet, wenn Objekte über das Netzwerk schwirren (die Rede ist hier von RMI). Die persistenten Objekte sichern also neben ihren eigenen Informationen auch die Unterobjekte - also die von der betrachtenden Stelle aus erreichbaren. Beim Speichern wird rekursiv ein Objektbaum durchlaufen, um eine vollständige Datenstruktur zu erhalten. Der doppelte Zugriff auf ein Objekt wird hier ebenso beachtet wie der Fall, dass zyklische Abhängigkeiten auftreten. Jedes Objekt bekommt dabei ein Handle, sodass es im Datenstrom nur einmal kodiert wird.

Unter Java SE lassen sich Objekte über verschiedene Ansätze automatisch persistent abbilden und speichern:

- ✓ **Standardserialisierung:** Die Objektstruktur und Zustände werden in einem binären Format gesichert. Das Verfahren wird auch **Java Object Serialization** (JOS) genannt - der Punkt, mit dem wir uns im Folgenden beschäftigen wollen. Die Standardserialisierung ist bei entfernten Methodenaufrufen sehr wichtig und weniger, um Dinge über einen langen Zeitraum abzuspeichern und dann irgendwann einmal wieder aus dem Schrank zu holen.
- ✓ **XML-Serialisierung über JavaBeans Persistence:** JavaBeans - und nur solche - können wir in einem XML-Format sichern. Eine Lösung ist die JavaBeans Persistence (JBP), die ursprünglich für Swing gedacht war. Denn wenn der Zustand einer grafischen Oberfläche mit JOS binär persistiert wird, sind Änderungen an den Interna der Swing-API nicht so einfach möglich, da das Binärformat der JOS sehr eng mit dem Objektmodell verbunden ist. Das heißt, Objekte lassen sich mitunter nicht mehr aus dem Binärdokument rekonstruieren. JBP entkoppelt das, indem nur über Setter/Getter kommuniziert wird und nicht auf internen Referenzen, die ein Implementierungsdetail sind, das sich jederzeit ändern kann. Heutzutage spielt JBP in der Praxis kaum eine Rolle.
- ✓ **XML-Abbildung über JAXB:** Mit JAXB steht eine zweite API zum Abbilden der Objektstruktur auf XML-Dokumente bereit. JAXB ist Teil der Standardbibliothek ab Version 6. Sie ist eine sehr wichtige Technologie, insbesondere für Web-Service-Aufrufe.

Die drei Möglichkeiten JOS, JBP und JAXB sind in Java SE schon eingebaut. Die Standardserialisierung erzeugt ein binäres Format und ist sehr stark auf Java ausgerichtet, sodass andere Systeme nicht viel mit den Daten anfangen können. XML ist als Format praktisch, da es auch von anderen Systemen verarbeitet werden kann.

Ein anderes kompaktes Binärformat, das auch Interoperabilität erlaubt, ist **Protocol Buffers** (<http://code.google.com/p/protobuf/>) von Google; das Unternehmen setzt es intern ein, wenn unterschiedliche Anwendungen Daten austauschen sollen.

Etwas weiter gedacht lassen sich auch Objekte in relationalen Datenbanken speichern, was sich **objekt-relacionales Mapping (OR-Mapping)** nennt. Das ist sehr anspruchsvoll, da die Objektmodelle und Tabellen so ganz anders sind. Die Java SE bietet keine Unterstützung für das OR-Mapping an, doch mit zusätzlichen Frameworks, wie der JPA (**Java Persistence API**), ist das zu schaffen. Auch von Hand können die Objekte über JDBC in die Datenbank gebracht werden, was aber nicht zeitgemäß ist.

Objekte mit der Standard-Serialisierung speichern und lesen

Die Standard-Serialisierung bietet eine einfache Möglichkeit, um Objekte persistent zu machen und später wiederherzustellen. Dabei werden die Objektzustände (keine statischen!) in einen Byte-Strom geschrieben (Serialisierung), woraus sie später wieder zu einem Objekt rekonstruiert werden können (Deserialisierung). Im Zentrum stehen zwei Klassen und ihre (De-)Serialisierungsmethode:

- ✓ **Serialisierung:** Die Klasse `ObjectOutputStream` und die Methode `writeObject()`. Während der Serialisierung geht `ObjectOutputStream` die Zustände und Objektverweise rekursiv ab und schreibt die Zustände Schritt für Schritt in einen Ausgabestrom.
- ✓ **Deserialisierung:** Zum Lesen der serialisierten Objektzustände dient die Klasse `ObjectInputStream`. Ihre Methode `readObject()` findet den Typ des serialisierten Objekts und baut daraus zur Laufzeit das Zielobjekt auf.

`ObjectOutputStream`

An einem Beispiel lässt sich gut erkennen, wie ein `ObjectOutputStream` einen String und das aktuelle Tagesdatum in einen `OutputStream` speichert. Um die Daten in eine Datei zu holen, ist der `OutputStream` ein `FileOutputStream` für eine Datei `datum.ser`. Der Dateiname wird meist so gewählt, dass er mit `.ser` endet:

```
com/tutego/insel/io/ser/SerializeAndDeserializeDate.java, serialize()

OutputStream fos = null;

try
{
    fos = new FileOutputStream( filename );
    ObjectOutputStream o = new ObjectOutputStream( fos );
    o.writeObject( "Today" );
    o.writeObject( new Date() );
}
catch ( IOException e ) { System.err.println( e ); }
finally { try { fos.close(); } catch ( Exception e ) { e.printStackTrace(); } }
```

Allen Anfang bildet wie üblich ein `OutputStream`, der die Zustände der Objekte und Meta-Informationen aufnimmt. In unserem Fall ist das der `FileOutputStream`. Die Verbindung zwischen der Datei und dem Objektstrom durch die Klasse `ObjectOutputStream` geschieht über den Konstruktor, der einen `OutputStream` annimmt. `ObjectOutputStream` implementiert die Schnittstelle `ObjectOutput` und bietet so beispielsweise die Methode `writeObject()` zum Schreiben von Objekten. Damit wird das Serialisieren des String-Objekts (das "Today") und des anschließenden Datum-Objekts zum Kinderspiel.

```
class java.io.ObjectOutputStream
extends OutputStream
implements ObjectOutput, ObjectStreamConstants
```

- ✓ `ObjectOutputStream(OutputStream out) throws IOException`
Erzeugt einen `ObjectOutputStream`, der in den angegebenen `OutputStream` schreibt. Ein Fehler kann von den Methoden aus dem `OutputStream` kommen.
- ✓ `final void writeObject(Object obj) throws IOException`
Schreibt das Objekt.

- ✓ void flush() throws IOException
Schreibt noch gepufferte Daten.
- ✓ void close() throws IOException
Schließt den Datenstrom. Die Methode muss aufgerufen werden, bevor der Datenstrom zur Eingabe verwendet werden soll.

Die Methode `writeObject()` kann nicht nur bei Ein-/Ausgabefehlern eine `IOException` auslösen, sondern auch eine `NotSerializableException`, wenn das Objekt gar nicht serialisierbar ist, und eine `InvalidClassException`, wenn beim Serialisieren etwas falsch läuft.

Objekte über die Standard-Serialisierung lesen

Aus den Daten im Datenstrom stellt der `ObjectInputStream` ein neues Objekt her und initialisiert die Zustände, wie sie geschrieben wurden. Wenn nötig, restauriert der `ObjectInputStream` auch Objekte, auf die verwiesen wurde. Die Klasseninformationen müssen zur Laufzeit vorhanden sein, weil bei der Serialisierung nur die Zustände, aber keine `.class`-Dateien gesichert werden. Während des Lesens findet `readObject()` also bei unserem Beispiel den String und das Datum. Der `ObjectInputStream` erwartet die Rohdaten wie üblich über einen Eingabestrom. Kommen die Informationen aus einer Datei, verwenden wir den `FileInputStream`:

```
com/tutego/insel/io/ser/SerializeAndDeserializeDate.java, deserialize()

InputStream fis = null;

try
{
    fis = new FileInputStream( filename );
    ObjectInputStream o = new ObjectInputStream( fis );
    String string = (String) o.readObject();
    Date date      = (Date) o.readObject();

    System.out.println( string );
    System.out.println( date );
}
catch ( IOException e ) { System.err.println( e ); }
catch ( ClassNotFoundException e ) { System.err.println( e ); }
finally { try { fis.close(); } catch ( Exception e ) { } }
```

Die explizite Typumwandlung kann natürlich bei einer falschen Zuweisung zu einem Fehler führen. Bei generischen Typen ist diese Typanpassung immer etwas lästig.

```
class java.io.ObjectInputStream
extends InputStream
implements ObjectInput, ObjectStreamConstants
```

- ✓ `ObjectInputStream(InputStream out) throws IOException`
Erzeugt einen `ObjectInputStream`, der aus einem gegebenen `InputStream` liest.
- ✓ `final Object readObject() throws ClassNotFoundException, IOException`
Liest ein `Object` und gibt es zurück. Eine `ClassNotFoundException` wird ausgelöst, wenn das Objekt zu einer Klasse gehört, die nicht auffindbar ist.

Die Schnittstellen `ObjectOutput` und `ObjectInput`

Die Klasse `ObjectOutputStream` bekommt die Vorgabe für `writeObject()` aus einer Schnittstelle `ObjectOutput`, genauso wie `ObjectInputStream` die Operation `readObject()` aus `ObjectInput` implementiert. Bis auf die Standard-Serialisierung haben die Schnittstellen `ObjectOutput` und `ObjectInput` in Java keine weitere Verwendung.

Die Schnittstelle `ObjectOutput` erweitert selbst die Schnittstelle `DataOutput` um das Schreiben von Primitiven: `write(byte[])`, `write(byte[], int, int)`, `write(int)`, `writeBoolean(boolean)`, `writeByte(int)`, `writeBytes(String)`, `writeChar(int)`, `writeChars(String)`, `writeDouble(double)`, `writeFloat(float)`, `writeInt(int)`, `writeLong(long)`, `writeShort(int)` und `writeUTF(String)`. Das ist bei einer eigenen angepassten Serialisierung interessant, wenn wir selbst das Schreiben von Zuständen übernehmen. Umgekehrt erweitert die Schnittstelle `ObjectInput` die Schnittstelle `DataInput` und bietet diverse `readXXX()`-Methoden.

Zwei einfache Anwendungen der Serialisierung

Im Folgenden wollen wir uns zwei Beispiele für die Serialisierung anschauen:

- ✓ Objektzustände zu verpacken, ist bei der Kommunikation über ein Netzwerk sehr sinnvoll. Die Serialisierung kann einfach die Zustände von einem Rechner auf den anderen übertragen.
- ✓ Serialisierung ist aber auch eine Möglichkeit, die Zustände als Byte-Feld etwa in eine Datenbank zu schreiben. Dabei werden wir sehen, dass der `ByteArrayOutputStream` eine nützliche Stream-Klasse ist.

Objekte über das Netzwerk schicken

Es ist natürlich wieder feines objektorientiertes Design, dass es der Methode `writeObject()` egal ist, wohin das Objekt geschoben wird. Dazu wird ja einfach dem Konstruktor von `ObjectOutputStream` ein `OutputStream` übergeben, und `writeObject()` delegiert dann das Senden der entsprechenden Einträge an die passenden Methoden der `Output`-Klasse. Im Beispiel `SerializeAndDeserializeDate` haben wir ein `FileOutputStream` benutzt. Es gibt aber noch eine ganze Menge anderer Klassen, die vom Typ `OutputStream` sind. So können die Objekte auch in einer Datenbank abgelegt beziehungsweise über das Netzwerk verschickt werden. Wie dies funktioniert, zeigen die nächsten Zeilen:

```
Socket s = new Socket( host, port );
OutputStream os = s.getOutputStream();
ObjectOutputStream oos = new ObjectOutputStream( os );
oos.writeObject( object );
oos.flush();
```

Über `s.getOutputStream()` gelangen wir an den Datenstrom. Dann sieht alles wie gewohnt aus. Da wir allerdings auf der Empfängerseite noch ein Protokoll ausmachen müssen, verfolgen wir diesen Weg der Objektversendung nicht weiter und verlassen uns vielmehr auf eine Technik, die sich **RMI** nennt.

Objekte in ein Byte-Feld schreiben

Die Klassen `ObjectOutputStream` und `ByteArrayOutputStream` sind zusammen zwei gute Partner, wenn es darum geht, eine Repräsentation eines Objekts im Speicher zu erzeugen und die geschätzte Größe eines Objekts herauszufinden.

```
Object o = ...;
ByteArrayOutputStream baos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream( baos );
oos.writeObject( o );
oos.close();
byte[] array = baos.toByteArray();
```

Nun steht das Objekt im Byte-Feld. Wollten wir die Größe erfragen, müssten wir das Attribut `length` des Felds auslesen.

Die Schnittstelle Serializable

Bisher nahmen wir immer an, dass eine Klasse weiß, wie sie geschrieben wird. Das funktioniert wie selbstverständlich bei vielen vorhandenen Klassen, und so müssen wir uns bei `writeObject(new Date())` keine Gedanken darüber machen, wie die Bibliothek das Datum schreibt und auch wieder liest.

Damit Objekte serialisiert werden können, müssen die Klassen die Schnittstelle `Serializable` implementieren. Diese Schnittstelle enthält keine Methoden und ist nur eine **Markierungsschnittstelle** (engl. **marker interface**). Implementiert eine Klasse diese Schnittstelle nicht, folgt beim Serialisierungsversuch eine `NotSerializableException`. Eine Klasse wie `java.util.Date` implementiert somit `Serializable`, `Thread` jedoch nicht. Der Serialisierer lässt damit alle Klassen "durch", die `instanceof Serializable` sind. Daraus folgt, dass alle Unterklassen einer Klasse, die serialisierbar ist, auch ihrerseits serialisierbar sind. So implementiert `java.lang.Number` - die Basisklasse der Wrapper-Klassen - die Schnittstelle `Serializable`, und die konkreten Wrapper-Klassen wie `Integer`, `BigDecimal` sind somit ebenfalls serialisierbar.

Werden Exemplare einer nicht-statischen inneren Klasse serialisiert, ohne dass die äußere Klasse `Serializable` implementiert, gibt es einen Fehler, denn intern hält ein Objekt der inneren Klasse einen Verweis auf das Exemplar der äußeren Klasse. Statische innere Klassen machen das nicht, was das Problem mit der Serialisierung lösen kann. Das Datenvolumen kann natürlich groß werden, wenn schlanke, nicht-statische innere `Serializable`-Klassen in einer äußeren `Serializable`-Klasse liegen, die sehr viele Eigenschaften besitzt.



Die serialisierbare Klasse Person

Wir wollen im Folgenden eine Klasse `Person` serialisierbar machen. Dazu benötigen wir das folgende Gerüst:

```
com/tutego/insel/io/ser/Person.java
package com.tutego.insel.io.ser;

import java.io.Serializable;
import java.util.Date;

public class Person implements Serializable
{
    static int BMI_OVERWEIGHT = 25;

    String name;
    Date birthday;
    double bodyHeight;
}
```

Erzeugen wir ein `Person`-Objekt `p` und rufen `writeObject(p)` auf, so schiebt der `ObjectOutputStream` die Variablen-Belegungen (hier `name`, `birthday` und `bodyHeight`) in den Datenstrom.

Statische Variablen wie `BMI_OVERWEIGHT` werden nicht mit dem Standard-Serialisierungsmechanismus gesichert. Bevor durch Deserialisierung ein Objekt einer Klasse erzeugt wird, muss schon die Klasse geladen sein, was bedeutet, dass statische Variablen schon initialisiert sind. Wenn zwei Objekte wieder deserialisiert werden, könnte es andernfalls vorkommen, dass beide unterschiedliche Werte aufweisen. Was sollte dann passieren?

Feld-Objekte sind standardmäßig serialisierbar - sie implementieren versteckt die Schnittstelle `Serializable`.



Nicht serialisierbare Objekte

Nicht alle Objekte sind serialisierbar. Zu den nicht serialisierbaren Klassen gehören zum Beispiel `Thread` und `Socket` und viele weitere Klassen aus dem `java.io`-Paket. Das liegt daran, dass nicht klar ist, wie zum Beispiel ein Wiederaufbau aussehen sollte. Wenn ein Thread etwa eine Datei zum Lesen geöffnet hat, wie soll der Zustand serialisiert werden, sodass er beim Deserialisieren auf einem anderen Rechner sofort wieder laufen und dort weitermachen kann, wo er mit dem Lesen aufgehört hat?

Ob Objekte als Träger sensibler Daten serialisierbar sein sollen, ist gut zu überlegen. Denn bei der Serialisierung der Zustände - es werden auch private Attribute serialisiert, an die zunächst nicht so einfach heranzukommen ist - öffnet sich die Kapselung. Aus dem Datenstrom lassen sich die internen Belegungen ablesen und auch manipulieren.

Nicht serialisierbare Attribute aussparen

Es gibt eine Reihe von Objekttypen, die sich nicht serialisieren lassen - technisch gesprochen implementieren diese Klassen die Schnittstelle `Serializable` nicht. Der Grund, dass nicht alle Klassen diese Schnittstelle implementieren, kann zum Beispiel die Sicherheit sein: Ein Objekt, das Passwörter speichert, soll nicht einfach geschrieben werden. Da reicht es nicht, dass die Attribute privat sind, denn auch sie werden geschrieben. Der andere Punkt ist die Tatsache, dass sich nicht alle Zustände beim Deserialisieren wiederherstellen lassen. Was ist, wenn ein `FileInputStream` oder `Thread` serialisiert wird? Soll dann bei der Deserialisierung eine Datei geöffnet werden oder der Thread neu starten? Was ist, wenn die Datei nicht vorhanden ist? Da all diese Fragen ungeklärt sind, ist es am einfachsten, wenn die Klassen nicht serialisierbarer Objekte die Schnittstelle `Serializable` nicht implementieren.

Doch was soll geschehen, wenn ein Objekt geschrieben wird, das intern auf ein nicht serialisierbares Objekt - etwa auf einen Thread - verweist?

Die Serialisierung der folgenden Klasse führt zu einem Laufzeitfehler:

com/tutego/insel/io/ser/SerializeTransient.java, NotTransientNotSerializable

```
class NotTransientNotSerializable implements Serializable
{
    Thread t = new Thread();
    // transient Thread t = new Thread();
    String s = "Fremde sind Freunde, die man nur noch nicht kennengelernt hat.";
}
```

Der Fehler wird eine `NotSerializableException` sein:

```
Exception in thread "main" java.io.NotSerializableException: java.lang.Thread
  at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1151)
  at java.io.ObjectOutputStream.defaultWriteFields(ObjectOutputStream.java:1504)
  at java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:1469)
  at java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.java:1387)
  at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1145)
  at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:326)
  at com.tutego.insel.io.ser.SerializeTransient.main(SerializeTransient.java:19)
```

Die Begründung dafür ist einfach: Ein Thread lässt sich nicht serialisieren.

Wollten wir ein Objekt vom Typ `NotTransientNotSerializable` ohne Thread serialisieren, müssen wir dem Serialisierungsmechanismus mitteilen: "Nimm so weit alle Objekte, aber nicht den Thread!"

Um Elemente bei der Serialisierung auszusparen, bietet Java zwei Möglichkeiten:

- ✓ ein spezielles Schlüsselwort: `transient`
- ✓ das Feld `private final ObjectOutputStreamField[] serialPersistentFields = {...}, das alle serialisierbaren Eigenschaften aufzählt`

Statische Eigenschaften würden auch nicht serialisiert, aber das ist hier nicht unser Ziel.



Ausnahmen sind standardmäßig serialisierbar, da `Throwable` die Schnittstelle `Serializable` implementiert. Denn gibt es Serverfehler bei entfernten Methodenaufrufen, so werden die Fehler gerne mit über die Leitung übertragen. Natürlich darf in dem Fall die zu serialisierende Ausnahme auch nur serialisierbare Attribute referenzieren.

Das Schlüsselwort `transient`

Um beim Serialisieren Attribute auszusparen, bietet Java den Modifizierer `transient`, der alle Attribute markiert, die nicht persistent sein sollen. Damit lassen wir die nicht serialisierbaren Kandidaten außen vor und speichern alles ab, was sich speichern lässt.

Beispiel: Das Thread-Objekt hinter `t` soll nicht serialisiert werden:

```
transient Thread t;
```

Die Variable `serialPersistentFields`

Erkennt der Serialisierer in der Klasse eine private statische Feld-Variable `serialPersistentFields`, wird er die `ObjectStreamField`-Einträge des Feldes beachten und nur die dort aufgezählten Elemente serialisieren, egal, was `transient` markiert ist.

Beispiel: Von einer Klasse sollen nur der String `name` und das Datum `date` serialisiert werden:

```
private static final ObjectStreamField[] serialPersistentFields {
    new ObjectStreamField( "name", String.class ),
    new ObjectStreamField( "date", Date.class )
};
```

Das Abspeichern selbst in die Hand nehmen

Die Java-Bibliothek realisiert intern ein Serialisierungsprotokoll, das beschreibt, wie die Abbildung auf einen Bytestrom aussieht. Dieses **Object Serialization Stream Protocol** beschreibt Oracle unter <http://download.oracle.com/javase/7/docs/platform/serialization/spec/protocol.html> etwas genauer, aber Details sind nicht nötig. Es kann aber passieren, dass die Standard-Serialisierung nicht erwünscht ist, wenn zum Beispiel beim Zurücklesen weitere Objekte erzeugt werden sollen oder wenn beim Schreiben eine bessere Abbildung durch Kompression möglich ist.

Für diesen Fall müssen spezielle (private!) Methoden implementiert werden. Beide müssen die nachstehenden Signaturen aufweisen:

```
private synchronized void writeObject( java.io.ObjectOutputStream s )
    throws IOException
```

und

```
private synchronized void readObject( java.io.ObjectInputStream s )
    throws IOException, ClassNotFoundException
```

Die Methode `writeObject()` ist für das Schreiben verantwortlich. Ist der Rumpf leer, gelangen keine Informationen in den Strom, und das Objekt wird folglich nicht gesichert. `readObject()` wird während der Deserialisierung aufgerufen. Ist dieser Rumpf leer, werden keine Zustände rekonstruiert.

Mit diesen Methoden können wir also die Serialisierung selbst in die Hand nehmen und die Attribute so speichern, wie wir es für sinnvoll halten; eine Kompatibilität lässt sich erzwingen. Eine kleine Versionsnummer im Datenstrom könnte eine Verzweigung provozieren, in der die Daten der Version 1 oder andere Daten der Version 2 gelesen werden.

Beim Lesen können komplettete Objekte wieder aufgebaut werden, und es lassen sich zum Beispiel nicht-transiente Objekte wiederbeleben. Stellen wir uns einen Thread vor, dessen Zustände beim Schreiben persistent gemacht werden; beim Lesen wird ein Thread-Objekt wieder erzeugt und zum Leben erweckt.

Oberklassen serialisieren sich gleich mit

Wird eine Klasse serialisiert, so werden automatisch die Informationen der Oberklasse mitserialisiert. Hierbei gilt, dass wie beim Konstruktor erst die Attribute der Oberklasse in den Datenstrom geschrieben werden und anschließend die Attribute der Unterklassen. Insbesondere bedeutet dies, dass die Unterklasse nicht noch einmal die Attribute der Oberklasse speichern sollte. Das folgende Programm zeigt den Effekt:

```
com/tutego/insel/io/ser/WriteTop.java
import java.io.*;

class Base implements Serializable
{
    private void writeObject( ObjectOutputStream oos )
    {
        System.err.println( "Base" );
    }
}
```

```

public class WriteTop extends Base
{
    public static void main( String[] args ) throws IOException
    {
        ObjectOutputStream oos = new ObjectOutputStream( System.out );
        oos.writeObject( new WriteTop() );
    }

    private void writeObject( ObjectOutputStream oos )
    {
        System.err.println( "Top" );
    }
}

```

Doch noch den Standardserialisierer nutzen

Die Methoden `readObject()`/`writeObject()` arbeiten nach dem Alles-oder-nichts-Prinzip. Erkennt der Serialisierer, dass die Schnittstelle `Serializable` implementiert wird, fragt er die Klasse, ob sie die Methoden implementiert. Wenn nicht, beginnt bei der Serialisierung der Serialisierungsmechanismus eigenständig, die Attribute auszulesen und in den Datenstrom zu schreiben. Gibt es die `readObject()`/`writeObject()`-Methoden, so wird der Serialisierer diese aufrufen und nicht selbst die Objekte nach den Werten fragen oder die Objekte mit Werten füllen.

Doch die Arbeit des Serialisierers ist eine große Hilfe. Falls viele Attribute zu speichern sind, fällt viel lästige Arbeit beim Programmieren an, da für jedes zu speichernde Attribut der Aufruf einer `writeXXX()`-Methode und beim Lesen eine entsprechende `readXXX()`-Methode nötig ist. Aus diesem Dilemma gibt es einen Ausweg, weil der Serialisierer in den `readObject()`/`writeObject()`-Methoden auch nachträglich dazu verpflichtet werden kann, die nicht-transienten Attribute zu lesen oder zu schreiben. Die privaten Methoden `readObject()` und `writeObject()` bekommen als Argument ein `ObjectInputStream` und ein `ObjectOutputStream`, die über die entsprechenden Methoden verfügen.

Die Klasse `ObjectOutputStream` erweitert `java.io.OutputStream` unter anderem um die Methode `defaultWriteObject()`. Sie speichert die Attribute einer Klasse.

```

class java.io.ObjectOutputStream
extends OutputStream
implements ObjectOutput, ObjectStreamConstants

```

- ✓ `public final void defaultWriteObject() throws IOException`

Schreibt alle nicht-statischen und nicht-transienten Attribute in den Datenstrom. Die Methode kann nur innerhalb einer privaten `writeObject()`-Methode aufgerufen werden; andernfalls erhalten wir eine `NotActiveException`.

Das Gleiche gilt für die Methode `defaultReadObject()` in der Klasse `ObjectInputStream`.



Die Standard-Deserialisierung hat mit finalen Variablen kein Problem. Wenn wir allerdings selbst `readObject()` aufrufen, können wir nicht problemlos finale Variablen initialisieren. Hier bietet sich an, auf `defaultReadObject()` zurückzugreifen oder abartig zu tricksen, was etwa nötig ist, wenn eine Variable `final` und `transient` ist, da ja transiente Variablen erst gar nicht von der Standardserialisierung berücksichtigt werden. Das Problem ist unter der Fehlernummer 6379948 (http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6379948) bekannt, und dort werden auch einige Lösungen präsentiert.

Beispiel für `defaultReadObject()`/`defaultWriteObject()`

Unsere nächste Klasse `SpecialWomen` deklariert zwei Attribute: `name` und `alter`. Da manche Frauen nicht über ihr Alter sprechen wollen, soll `alter` nicht serialisiert werden; es ist transient. Wir implementieren eigene `readObject()`/`writeObject()`-Methoden, die den Standardserialisierer bemühen. Bei der Rekonstruktion über `readObject()` wird die Frau dann immer 30 bleiben:

`com/tutego/insell/io/ser/SpecialWomen.java`

```

package com.tutego.insel.io.ser;

import java.io.*;

public class SpecialWomen implements Serializable
{
    private static final long serialVersionUID = 2584203323009771108L;

    String name = "Madonna";
    transient int age = 30;

    private void writeObject( ObjectOutputStream oos ) throws IOException
    {
        oos.defaultWriteObject(); // Schreibe Name, aber kein Alter
    }

    private void readObject( ObjectInputStream ois ) throws IOException
    {
        try
        {
            ois.defaultReadObject(); // Lies Name, aber kein Alter
            age = 30;
        }
        catch ( ClassNotFoundException e )
        {
            throw new IOException( "No class found. HELP!!" );
        }
    }
}

```

Es ist gar nicht so abwegig, nur eine `readObject()`-, aber keine `writeObject()`-Methode zu implementieren. In `readObject()` lässt ein `defaultReadObject()` alle Eigenschaften initialisieren und danach noch Initialisierungsarbeit ähnlich einem Konstruktor durchführen. Dazu zählen etwa die Initialisierung von transienten Attributen, die Registrierung von Listenern und Weiteres.



Der andere macht's: `writeReplace()` und `readResolve()`

Eine Klasse muss die Serialisierung nicht selbst übernehmen, sondern kann die Arbeit abgeben. Dazu muss zum Schreiben eine Methode `writeReplace()` implementiert werden, die eine Referenz auf ein Objekt liefert, das das Schreiben übernimmt. Anregungen finden Leser unter <http://download.oracle.com/javase/7/docs/platform/serialization/spec/output.html#5324> sowie unter <http://www.jguru.com/faq/view.jsp?EID=44039>.

Tiefe Objektkopien

Implementieren Klassen die Markierungsschnittstelle `Serializable` und überschreiben sie die `clone()`-Methode von `Object`, so können sie eine Kopie der Werte liefern. Die üblichen Implementierungen liefern aber nur flache Kopien. Dies bedeutet, dass Referenzen auf Objekte, die von dem zu klonenden Objekt ausgehen, beibehalten und diese Objekte nicht extra kopiert werden. Als Beispiel kann die Datenstruktur `List` genügen, die `Map`-Objekte enthält. Ein Klon dieser Liste ist lediglich eine zweite Liste, deren Elemente auf die gleichen `Maps` zeigen.

Möchten wir das Verhalten ändern und eine tiefe Kopie anfertigen, so haben wir dank eines kleinen Tricks damit keine Mühe: Wir könnten das zu klonende Objekt einfach serialisieren und dann wieder auspacken. Die zu klonenden Objekte müssen dann neben `Cloneable` noch das `Serializable`-Interface implementieren:

com/tutego/insel/io/ser/Dolly.java, deepCopy()

```
@SuppressWarnings("unchecked")
public static <T> T deepCopy( T o ) throws Exception
{
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    new ObjectOutputStream( baos ).writeObject( o );

    ByteArrayInputStream bais = new ByteArrayInputStream( baos.toByteArray() );
    Object p = new ObjectInputStream( bais ).readObject();

    return (T) p;
}
```

Das Einzige, was wir zum Gelingen der Methode `deepCopy()` beitragen müssen, ist, das Objekt in einem `ByteArrayOutputStream` zu serialisieren, es wieder auszulesen und zu einem Objekt zu konvertieren. Den Einsatz eines `ByteArrayOutputStream` haben wir schon beobachtet, als wir die Länge eines Objekts herausfinden wollten. Nun fügen wir das Feld einfach wieder zu einem `ByteArrayInputStream` hinzu, aus dessen Daten dann `ObjectInputStream` das Objekt rekreieren kann.

Überzeugen wir uns anhand eines kleinen Programms, dass die tiefe Kopie tatsächlich etwas anderes als ein `clone()` ist:

Dolly.java, main()

```
Map<String, String> map = new HashMap<String, String>();
map.put( "Cul de Paris", "hinten unter dem Kleid getragenes Gestell oder Polster" );

LinkedList<Map<String, String>> ll = new LinkedList<Map<String, String>>();
ll.add( map );

@SuppressWarnings("unchecked")
List<Map<String, String>> l2 = (List<Map<String, String>>) ll.clone();

List<Map<String, String>> l3 = (List<Map<String, String>>) deepCopy( ll );

map.clear();

System.out.println( ll ); // []
System.out.println( l2 ); // []
System.out.println( l3 ); // [{Cul de Paris=hinten unter dem Kleid ...}]
```

Zunächst erstellen wir eine `Map`, die wir anschließend in eine Liste packen. Die `Map` enthält ein Pärchen. Kopiert `clone()` die Liste, so wird sie zwar selbst kopiert, aber nicht die referenzierten `Map`-Objekte - erst die tiefe Kopie kopiert die `Map` mit. Das sehen wir dann, wenn wir den Eintrag aus der `Map` löschen. Dann ergibt `ll` genauso wie `l2` eine leere Liste, da `l2` nur die Verweise auf die `Map` gespeichert hat, die dann aber geleert ist. Anders ist dies bei `l3`, der tiefen Kopie: Hier ist das Paar noch vorhanden.

Versionenverwaltung und die SUID

Die erste Version einer Klassenbibliothek ist in der Regel nicht vollständig und nicht beendet. Es kann gut sein, dass Attribute und Methoden nachträglich in die Klasse eingefügt, gelöscht oder modifiziert werden. Das bedeutet aber auch, dass die Serialisierung zu einem Problem werden kann. Denn ändert sich der VariablenTyp oder kommen Variablen hinzu, ist eine gespeicherte Objektserialisierung nicht mehr gültig. Bei der Serialisierung wird in Java nicht nur der Objektinhalt geschrieben, sondern zusätzlich eine eindeutige Kennung der Klasse, die `UID`. Die `UID` ist ein Hashcode aus Namen, Attributen, Parametern, Sichtbarkeit und so weiter. Sie wird als `long` wie ein Attribut gespeichert. Ändert sich der Aufbau einer Klasse, ändern sich der Hashcode und damit die `UID`. Klassen mit unterschiedlicher `UID` sind nicht kompatibel. Erkennt der Lese-mechanismus in einem Datenstrom eine `UID`, die nicht zur Klasse passt, wird eine `InvalidClassException` ausgelöst. Das bedeutet, dass schon ein einfaches Hinzufügen von Attributen zu einem Fehler führt.

Wir wollen uns dies einmal anhand einer einfachen Klasse ansehen. Wir entwickeln eine Klasse `Player` mit einem einfachen Ganzzahlattribut. Später fügen wir eine Fließkommazahl hinzu:

com/tutego/insel/io/ser/InvalidSer.java, Player

```
class Player implements Serializable
{
    String name;
    int age;
}
```

Dann benötigen wir noch das Hauptprogramm. Wir bilden ein Exemplar von `Player` und schreiben es in eine Datei:

com/tutego/insel/io/ser/InvalidSer.java, Ausschnitt main()

```
ObjectOutputStream oos = new ObjectOutputStream(
                    new FileOutputStream( "c:/test.ser" ) );
oos.writeObject( new Player() );
oos.close();
```

Ohne Änderungen können wir es direkt wieder deserialisieren:

com/tutego/insel/io/ser/InvalidSer.java, Ausschnitt main()

```
ObjectInputStream ois = new ObjectInputStream(
                    new FileInputStream( "c:/test.ser" ) );
Player player = (Player) ois.readObject();
System.out.println( player );
ois.close();
```

Ändern wir die Klassendeklaration `Player`, sodass wir etwa aus dem `int age` **ein double age machen**, führt dies beim Deserialisieren zu einem Fehler:

```
Exception in thread "main" java.io.InvalidClassException: com.tutego.insel.io.ser.Player;
local class incompatible: stream classdesc serialVersionUID = 44259824709362049,
local class serialVersionUID = 8962277452270582278
at java.io.ObjectStreamClass.initNonProxy(ObjectStreamClass.java:562)
at java.io.ObjectInputStream.readNonProxyDesc(ObjectInputStream.java:1583)
at java.io.ObjectInputStream.readClassDesc(ObjectInputStream.java:1496)
at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1732)
at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1329)
at java.io.ObjectInputStream.readObject(ObjectInputStream.java:351)
at com.tutego.insel.io.ser.InvalidSer.main(InvalidSer.java:22)
```

Die eigene SUID

Dem oberen Fehlerauszug entnehmen wir, dass der Serialisierungsmechanismus die SUID selbst berechnet. Das Attribut ist als statische, finale Variable mit dem Namen `serialVersionUID` in der Klasse abgelegt. Ändern sich die Klassenattribute, ist es günstig, eine eigene SUID einzutragen, denn der Mechanismus zum Deserialisieren kann dann etwas gutmütiger mit den Daten umgehen. Beim Einlesen gibt es nämlich Informationen, die nicht hinderlich sind. Wir sprechen in diesem Zusammenhang auch von **stream-kompatibel**. Dazu gehören zwei Bereiche:

- ✓ **Neue Felder:** Befinden sich in der neuen Klasse Attribute, die im Datenstrom nicht benannt sind, werden diese Attribute mit `0` oder `null` initialisiert.
- ✓ **Fehlende Felder:** Befinden sich im Datenstrom Attribute, die in der neuen Klasse nicht vorkommen, werden sie einfach ignoriert.

Die SUID kann eigentlich beliebig sein, doch die IDE bzw. das kleine Java-Dienstprogramm `serialver` berechnet einen Wert, der der gleiche Wert wie der ist, den der Serialisierungsmechanismus berechnet. Auf diese Weise erreichen wir eine stream-kompatible Serialisierung.

Beispiel: Dies wollen wir für unsere Klasse `Player` mit dem Dienstprogramm testen:

```
$ serialver com.tutego.insel.io.ser.Player
com.tutego.insel.io.ser.Player: static final long serialVersionUID =
8962277452270582278L;
```

Die Anweisung aus der letzten Zeile können wir in unsere Klasse `Player` kopieren. Wird danach ein weiteres Attribut in die Klasse gesetzt, gelöscht oder ändert sich der Typ eines Attributs, tritt die `InvalidClassException` nicht mehr auf, da die Stream-Kompatibilität über die `serialVersionUID` gewährleistet ist.



Da der Wert der Variablen `serialVersionUID` egal ist, kann sie bei 1 beginnen und immer dann, wenn es inkompatible Änderungen gibt, um eins erhöht werden.

Wie die `ArrayList` serialisiert

Am Beispiel einer `java.util.ArrayList` lässt sich sehr schön beobachten, wie sich die Methoden `writeObject()` und `readObject()` nutzen lassen. Eine `ArrayList` beinhaltet eine Reihe von Elementen. Zur Speicherung nutzt die Datenstruktur ein internes Feld. Das Feld kann größer als die Anzahl der Elemente sein, damit bei jedem `add()` das Feld nicht immer neu vergrößert werden muss. Nehmen wir an, die `ArrayList` würde eine Standardserialisierung nutzen. Was passiert nun? Es könnte das Problem entstehen, dass bei nur einem Objektverweis in der Liste und einer internen Feldgröße von 1.000 Elementen leider 999 null-Verweise gespeichert würden. Das wäre aber Verschwendungs! Besser ist es, eine angepasste Serialisierung zu verwenden:

`java.util.ArrayList.java, Ausschnitt`

```
private void writeObject(java.io.ObjectOutputStream s)
    throws java.io.IOException {
    // Write out element count, and any hidden stuff
    int expectedModCount = modCount;
    s.defaultWriteObject();

    // Write out array length
    s.writeInt(elementData.length);

    // Write out all elements in the proper order.
    for (int i=0; i<size; i++)
        s.writeObject(elementData[i]);

    if (modCount != expectedModCount) {
        throw new ConcurrentModificationException();
    }

}

private void readObject(java.io.ObjectInputStream s)
    throws java.io.IOException, ClassNotFoundException {
    // Read in size, and any hidden stuff
    s.defaultReadObject();

    // Read in array length and allocate array
    int arrayLength = s.readInt();
    Object[] a = elementData = new Object[arrayLength];

    // Read in all elements in the proper order.
    for (int i=0; i<size; i++)
        a[i] = s.readObject();
}
```

Probleme mit der Serialisierung

Der klassische Weg von einem Objekt zu einer persistenten Speicherung führt über den Serialisierungsmechanismus von Java über die Klassen `ObjectOutputStream` und `ObjectInputStream`. Die Serialisierung in Binärdaten ist aber nicht ohne Nachteile. Schwierig ist beispielsweise die Weiterverarbeitung von Nicht-Java-Programmen oder die nachträgliche Änderung ohne Einlesen und Wiederaufbauen der Objektverbunde. Wünschenswert ist daher eine Textrepräsentation. Diese hat nicht die oben genannten Nachteile. Ein weiteres Problem ist die Skalierbarkeit. Die Standard-Serialisierung arbeitet nach dem Prinzip: Alles, was vom Basisknoten aus erreichbar ist, gelangt serialisiert in den Datenstrom. Ist der Objektgraph sehr groß, steigen die Zeit für die Serialisierung und das Datenvolumen an. Anders als bei anderen Persistenz-Konzepten ist es nicht möglich, nur die Änderungen (die Differenz) zu schreiben. Wenn sich zum Beispiel in einer sehr großen Adressliste die Hausnummer einer Person ändert, muss die gesamte Adressliste neu geschrieben werden - das nagt an der Performance.

Auch parallele Änderungen können ein Problem sein, da die Serialisierung über kein transaktionales Konzept verfügt. Während der Serialisierung sind die Objekte und Datenstrukturen nicht gesperrt, und ein anderer Thread kann derweil alles Mögliche modifizieren. Der Entwickler muss sich selbst auferlegen, während des Schreibens keine Änderungen vorzunehmen, damit der Schreibzugriff isoliert ist. Auch wenn es während des Schreibens ein Problem (etwa eine Ausnahme) gibt, kommt ein halbfertiger Datenstrom beim Client an.

Heutzutage würden Bibliotheksdesigner keine Markierungsschnittstelle wie `Serializable` mehr einführen, sondern eine Annotation deklarieren. Die Serialisierungs-ID würde dann auch nicht mehr eine private statische Variable mit einem magischen Variablenamen sein, sondern ein Attribut der Annotation, sodass es zum Beispiel an einer Klasse heißen würde: `@Serializable(1234566778L)`. Markierungsschnittstellen haben noch ein anderes Problem, das mit der Endgültigkeit von Vererbung und Typen zu tun hat: Implementiert eine Klasse einmal `Serializable`, so gilt diese Eigenschaft auch für alle Unterklassen, auch wenn vielleicht die Unterklassen gar nicht serialisierbar sein sollen. Auch hier lösen Annotationen das Problem, denn es lässt sich einstellen, ob Annotationen vererbt werden sollen oder nicht. Zu guter Letzt: Für transiente, also nicht serialisierte Zustände hätte kein Schlüsselwort in der Sprache reserviert, sondern lediglich eine neue Annotation deklariert werden müssen. Auch statt der magischen privaten Methoden `readObject()`/`writeObject()` hätte gut eine Annotation deklariert werden können, die eben die Methoden markiert, die bei der (De-)Serialisierung aufgerufen werden sollen.



6.4 Alternative Datenaustauschformate

Die Standard-Serialisierung hat das Problem, dass sie nicht plattformunabhängig ist. Sollen aber über Rechnergrenzen Daten übertragen und ausgetauscht werden, so kommen andere Formate ins Spiel. Dieses Kapitel stellt einige Lösungen zur Serialisierung vor.

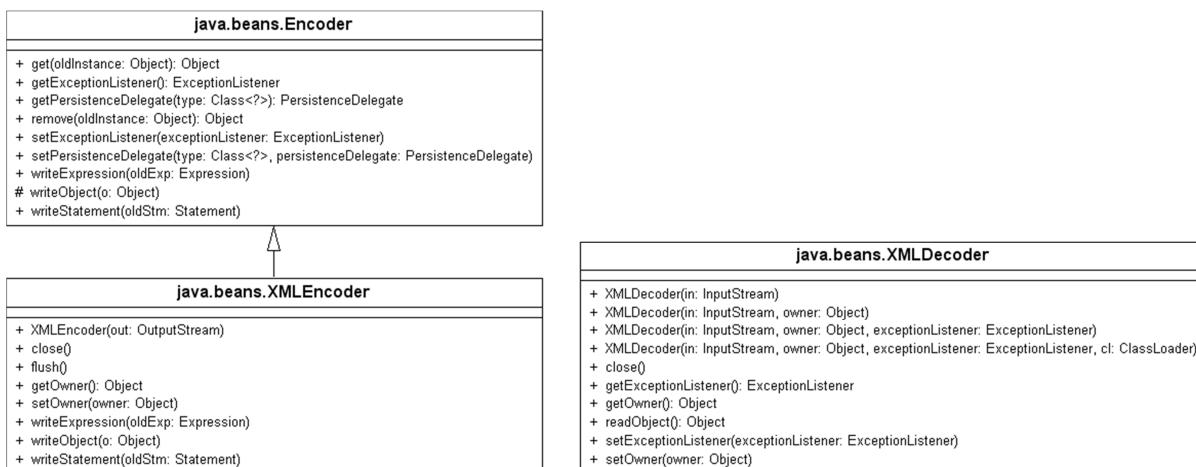
Serialisieren in XML-Dateien

Eine Abbildung in XML hat viele Vorteile, unter anderem den, dass auch andere Programmiersprachen leicht an die Daten kommen. Mittlerweile finden sich viele Bibliotheken, die Objektgraphen in XML abbilden:

- ✓ XStream (<http://xstream.codehaus.org/>)
- ✓ Java Architecture for XML Binding: JAXB (<https://jaxb.dev.java.net/>)
- ✓ Commons Betwixt (<http://jakarta.apache.org/commons/betwixt/>)
- ✓ XMLBeans (<http://xmlbeans.apache.org/>)
- ✓ Castor (<http://www.castor.org/>)
- ✓ Simple (<http://simple.sourceforge.net/>)

XML-Serialisierung von JavaBeans mit JavaBeans Persistence

Um mit der JavaBeans Persistence Objekte in XML zu schreiben und von dort zu laden, werden statt der Klassen `ObjectOutputStream` und `ObjectInputStream` die Klassen `XMLEncoder` und `XMLDecoder` eingesetzt.



Klassendiagramme von `XMLEncoder` und `XMLDecoder` (der keine besondere Oberklasse hat)

Die folgende Klasse ist unserem Programm `SerializeAndDeserialize` nachempfunden. Ersetzen müssen wir lediglich die `ObjectXXXStream`-Klassen. Die Klassen `XMLEncoder` und `XMLDecoder` liegen auch nicht in `java.io`, sondern unter dem Paket `java.beans`. Interessanterweise muss die Ausnahme `ClassNotFoundException` nicht mehr aufgefangen werden:

```
com/tutego/insel/io/ser/SerializeAndDeserializeXML.java
package com.tutego.insel.io.ser;

import java.io.*;
import java.util.Date;
import java.beans.*;

public class SerializeAndDeserializeXML
{
    public static void main( String[] args )
    {
        String filename = "datum.ser.xml";
        // Serialisieren

        XMLEncoder enc = null;

        try
        {
            enc = new XMLEncoder( new FileOutputStream(filename) );
            enc.writeObject( "Today" );
            enc.writeObject( new Date() );
        }
        catch ( IOException e ) {
            e.printStackTrace();
        }
        finally {
            if ( enc != null )
                enc.close();
        }

        // Deserialisieren

        XMLDecoder dec = null;

        try
        {
            dec = new XMLDecoder( new FileInputStream(filename) );

            String string = (String) dec.readObject();
            Date date     = (Date)   dec.readObject();

            System.out.println( string );
            System.out.println( date );
        }
        catch ( IOException e ) {
            e.printStackTrace();
        }
        finally {
            if ( dec != null )
                dec.close();
        }
    }
}
```

Und so sehen wir nach dem Ablauf des Programms in der Datei `datum.ser.xml` Folgendes:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0" class="java.beans.XMLDecoder">
<string>Today</string>
```

```
<object class="java.util.Date">
<long>1272904776250</long>
</object>
</java>
```

Bei eigenen Objekten ist immer zu bedenken, dass der eingebaute XML-Serialisierer nur JavaBeans schreibt. Eigene Klassen müssen daher immer `public` sein, einen Standard-Konstruktor besitzen und ihre serialisierbaren Eigenschaften über `getXXX()`/`setXXX()`-Methoden bereitstellen; sie müssen jedoch die Markierungs-schnittstelle `Serializable` nicht implementieren.

PersistenceDelegate

Dem `XMLEncoder` lässt sich über `setPersistenceDelegate(Class, PersistenceDelegate)` für einen speziellen Klassentyp ein `java.beans.PersistenceDelegate` mitgeben, der den Zustand eines Objekts speichert. Das ist immer dann praktisch, wenn der Standard-Mechanismus Eigenschaften nicht mitnimmt oder Klassen so nicht abbilden kann, weil sie zum Beispiel keinen Standard-Konstruktor deklarieren.

Für eigene Delegates ist die Unterklasse `DefaultPersistenceDelegate` recht praktisch. Sie ist auch hilfreich, um bestimmte Typen erst gar nicht zu schreiben:

```
XMLEncoder e = new java.beans.XMLEncoder( out );
e.setPersistenceDelegate( NonSer.class, new DefaultPersistenceDelegate() );
```

Die Open-Source-Bibliothek XStream

XStream ist eine quelloffene Software unter der BSD-Lizenz, mit der sich serialisierbare Objekte in XML umwandeln lassen. (Wer im Internet nach XStream sucht, der findet auch pinkfarbene Inhalte.) Damit ähnelt XStream eher der Standard-Serialisierung als der JavaBeans Persistence. Nachdem die unter <http://xstream.codehaus.org/download.html> geladene Bibliothek `xstream-x.y.jar` sowie der schnelle XML-Parser `xpp3_min-x.y.jar` auf der gleichen Seite eingebunden worden sind, ist ein Beispielprogramm schnell formuliert:

```
Point p = new Point( 120, 32 );
XStream xstream = new XStream();
String xml = xstream.toXML( p );
System.out.println( xml );
Point q = (Point) xstream.fromXML( xml );
```

Alle Ausnahmen von XStream sind Unterklassen von `RuntimeException` und müssen daher nicht explizit aufgefangen werden. Der String hinter `xml` enthält:

```
<java.awt.Point>
<x>120</x>
<y>32</y>
</java.awt.Point>
```

Ein XML-Prolog fehlt.

Binäre Serialisierung mit Google Protocol Buffers

Da Google unterschiedliche Programmiersprachen in seiner Softwarelandschaft nutzt, stand das Unternehmen vor der Frage, wie ein effektiver Datenaustausch aussehen kann, wenn etwa ein Server in C++ und ein Client in Java geschrieben ist. Für den Zweck hat Google das Datenformat **Protocol Buffers** entwickelt, das mittlerweile quelloffen unter der BSD-Lizenz für jeden unter <http://code.google.com/p/protobuf/> verfügbar ist.

Der Ausgangspunkt für die plattformunabhängige Übertragung von Strukturen, die Nachrichten (engl. **messages**) genannt werden, ist eine Strukturdefinition in einer **Proto-Definition-Datei** (Dateiendung `.proto`). Sie enthält die unterschiedlichen Nachrichten mit Feldern und Größenangaben und kann auch Aufzählungen enthalten. Der Protocol-Buffer-Compiler `protoc` generiert aus der Datei eine Klasse mit einer bestimmten Protocol-Buffer-API, die Java-Objekte entweder über den JavaBeans-Standard oder über das Builder-Pattern aufbaut und Methoden zum Serialisieren/Deserialisieren oder zum Zusammenfügen anbietet. Mehr zur Arbeitsweise zeigt das Tutorial unter <http://code.google.com/intl/de-DE/apis/protocolbuffers/docs/javatutorial.html>.

6.5 Tokenizer

Zu den schon im String-Kapitel vorgestellten Tokenizern Scanner, StringTokenizer und split() aus String gibt es im java.io-Paket eine weitere Klasse: StreamTokenizer.

StreamTokenizer

Die Klasse StreamTokenizer arbeitet noch spezialisierter als die StringTokenizer-Klasse aus dem util-Paket, und die Klasse Scanner kommt der Klasse StreamTokenizer schon sehr nahe. Im Gegensatz zum Scanner beachtet ein StreamTokenizer keine Unicode-Eingabe, sondern nur Zeichen aus dem Bereich von \u0000 bis \u00FF, kann aber mit Kommentaren umgehen.

Während des Parsens erkennt der Tokenizer bestimmte Merkmale, so unter anderem Bezeichner (etwa Schlüsselwörter), Zahlen, Strings in Anführungszeichen und verschiedene Kommentararten (C-Stil oder C++-Stil). Verschiedene Java-Tools von Oracle verwenden intern einen StreamTokenizer, um ihre Eingabedateien zu verarbeiten, etwa das Policy-Tool für die Rechteverwaltung. Der Erkennungsvorgang wird anhand einer Syntaxtabelle überprüft. Diese Tabelle enthält zum Beispiel die Zeichen, die ein Schlüsselwort identifizieren, oder die Zeichen, die Trennzeichen sind. Jedes gelesene Zeichen wird dann keinem, einem oder mehreren Attributen zugeordnet. Diese Attribute fallen in die Kategorie Trennzeichen, alphanumerische Zeichen, Zahlen, Hochkommata beziehungsweise Anführungszeichen oder Kommentarzeichen.

Zur Benutzung der Klasse wird zunächst ein StreamTokenizer-Objekt erzeugt, und dann werden die Syntaxtabellen initialisiert. Ob Kommentarzeilen überlesen werden sollen, wird durch

```
st.slashSlashComments( true );           // Kommentar
st.slashStarComments( true );           /* Kommentar */
```

gesteuert. Die erste Methode überliest im Eingabestrom alle Zeichen bis zum Return. Die zweite Methode überliest nur alles bis zum Stern/Slash. Geschachtelte Kommentare sind hier nicht möglich.

Beim Lesen des Datenstroms mit nextToken() kann über bestimmte Flags erfragt werden, ob im Stream ein Wort beziehungsweise Bezeichner (TT_WORD), eine Zahl (TT_NUMBER), das Ende der Datei (TT_EOF) oder das Ende der Zeile (TT_EOL) vorliegt. Wichtig ist, eolIsSignificant(true) zu setzen, da andernfalls der StreamTokenizer nie ein TT_EOL findet. Wurde ein Wort erkannt, dann werden alle Zeichen in Kleinbuchstaben konvertiert. Dies lässt sich über die Methode lowerCaseMode(boolean) einstellen. Nach der Initialisierung eines StreamTokenizer-Objekts wird normalerweise so lange nextToken() aufgerufen, bis die Eingabe keine neuen Zeichen mehr hergibt, also ein TT_EOF-Token erkannt wurde.

Ein Beispiel: Die folgende Klasse liest die Eingabe vom Netzwerk und gibt die erkannten Textteile aus:

```
com/tutego/insel/io/stream/StreamTokenizerDemo.java, main()

URL url = new URL( "http://www.tutego.com/index.html" );
Reader reader = new InputStreamReader( url.openStream() );
StreamTokenizer st = new StreamTokenizer( reader );

//      st.slashSlashComments( true ); /*
st.slashStarComments( true );
st.ordinaryChar( '/' );
st.parseNumbers();
st.eolIsSignificant( true );

for ( int tval; (tval = st.nextToken()) != StreamTokenizer.TT_EOF; )
{
    if ( tval == StreamTokenizer.TT_NUMBER )
        System.out.println( "Nummer: " + st.nval );
    else if ( tval == StreamTokenizer.TT_WORD )
        System.out.println( "Wort: " + st.sval );
    else if ( tval == StreamTokenizer.TT_EOL )
        System.out.println( "Ende der Zeile" );
    else
        System.out.println( "Zeichen: " + (char) st.ttype );
}
```