

Java Erweiterungen I

Effiziente Java-Programmierung

Gesamtinhaltsverzeichnis

1	Einführung	1-3
1.1	Inhalte des Seminars	1-3
1.2	Präsentation	1-3
1.3	Grundbegriffe.....	1-4
1.3.1	Laufzeitumgebung	1-5
1.3.2	Speicher der VM	1-5
1.3.3	Objektorientierte Programmierung	1-6
1.3.4	UML Klassendiagramm.....	1-8
1.4	Einige Klassen aus der Java Bibliothek.....	1-9
1.4.1	String und StringBuilder	1-10
1.4.2	Einige Methoden der Klasse String.....	1-11
1.4.3	Einige Methoden der Klasse StringBuilder.....	1-13
1.4.4	Formatierte Ausgaben	1-15
1.5	Zugriffsrechte.....	1-16
1.6	Pakete	1-17
1.6.1	Eigene Pakete erstellen	1-17
1.6.2	Import von Klassen	1-18
1.6.3	Statische Importe	1-19
1.7	Assertions.....	1-20
1.8	Fundamentale Datentypen	1-21
1.9	Typumwandlung durch Cast.....	1-22
1.10	Arrays	1-23
1.10.1	Mehrdimensionale Arrays	1-24
1.11	Variable Parameterliste	1-25
1.11.1	Das Problem	1-25
1.11.2	Die Lösung.....	1-26
1.12	Java Schlüsselwörter.....	1-28
1.13	Komponenten der Java Standard Edition	1-28
2	Programmierung	2-3
2.1	Die Klasse <code>java.lang.Object</code>	2-3
2.1.1	Die <code>equals()</code> -Methode	2-3
2.1.2	Die <code>toString()</code> -Methode.....	2-5
2.1.3	Die <code>clone()</code> -Methode	2-5
2.2	Garbage Collector	2-6

2.3	Ausnahmebehandlung.....	2-7
2.3.1	Die Klasse Throwable	2-7
2.3.2	Die Klasse Error.....	2-8
2.3.3	Die Klasse Exception	2-8
2.3.4	Ausnahmen mit fehlerbeschreibendem Text.....	2-10
2.3.5	Die Klasse RuntimeException.....	2-11
2.4	Typen von Klassen	2-12
2.4.1	Globale Klassen (Top-Level-Klassen).....	2-13
2.4.2	Statische Klassen	2-14
2.4.3	Innere Klassen (Memberklassen)	2-14
2.4.4	Lokale Klassen	2-16
2.4.5	Anonyme Klassen.....	2-17
2.5	Interfaces (Schnittstellen)	2-18
2.5.1	Start	2-19
2.5.2	Statische Methoden	2-20
2.5.3	Default-Methoden	2-21
2.5.4	Konflikte	2-23
2.6	Reflection	2-25
2.6.1	Die Klasse <code>Class</code>	2-25
2.6.2	Introspektion	2-25
2.6.3	Dynamische Erzeugung mit Reflection	2-26
2.7	Das Entwurfsmuster Factory	2-27
3	Das Collections Framework und Enums.....	3-3
3.1	Bestandteile des Collection-Frameworks	3-3
3.2	Einfache Implementierungen.....	3-6
3.2.1	List	3-6
3.2.2	Queue	3-7
3.2.3	Set	3-8
3.2.4	Map.....	3-9
3.3	Generische Collections.....	3-10
3.3.1	Notwendigkeit generischer Collections	3-10
3.3.2	Dokumentation.....	3-12
3.3.3	Technischer Hintergrund.....	3-13
3.3.4	Collections und einfache Datentypen	3-15
3.4	Iteration über Collections.....	3-16
3.4.1	Iteration mit Iterator.....	3-16
3.4.2	Iteration mit <code>forEach</code>	3-16
3.4.3	Iteration mit <code>forEach</code> -Methode.....	3-17
3.4.4	Iteration mit <code>forEach</code> -Methode als Lambda-Ausdruck	3-17

3.5	Die Klasse Collections	3-18
3.5.1	Sortieren von Collections	3-18
3.5.2	Unveränderbare Listen	3-20
3.5.3	Leere Listen	3-20
3.6	Enums	3-21
3.6.1	Ein Problem	3-21
3.6.2	Eine Lösung mit dem "alten" Java	3-22
3.6.3	Enums: enum.....	3-25
4	Ein-/Ausgabe und Properties	4-3
4.1	Einführung in Java I/O	4-3
4.1.1	Ein erstes Beispiel	4-3
4.2	Die Klasse File.....	4-5
4.3	Byte-basierte Ein- und Ausgabeströme	4-7
4.3.1	Schreiben einer Byte-Datei	4-8
4.3.2	Schreiben einer Byte-Datei mit Ressourcen	4-8
4.4	Reader und Writer	4-9
4.4.1	Schreiben einer Zeichen-Datei	4-9
4.4.2	Gepuffertes Schreiben	4-10
4.5	Konfiguration einer Anwendung mit Properties	4-11
4.5.1	System-Properties	4-11
4.5.2	Die Klasse java.util.Properties	4-11
4.5.3	Die Konfigurationsdatei.....	4-12
4.5.4	Benutzerabhängige Konfiguration mit Preferences.....	4-13
5	GUI-Programmierung.....	5-3
5.1	Das Abstract Windowing Toolkit (AWT).....	5-3
5.2	Das AWT Package	5-5
5.3	Swing.....	5-6
5.3.1	Die Swing-Bibliothek.....	5-6
5.3.2	Vor- und Nachteile von Swing.....	5-7
5.3.3	Beispiel: JFrame und JButton	5-7
5.4	Swing-Komponenten	5-9
5.4.1	Widgets.....	5-9
5.5	Layout Manager.....	5-10
5.5.1	FlowLayout	5-11
5.5.2	BorderLayout	5-12
5.5.3	GridLayout	5-15
5.5.4	Weitere Layouts	5-16
5.5.5	GridBagLayout.....	5-17
5.5.6	Schachtelung von Panels	5-17

5.6	Event Handling	5-19
5.7	Event-Objekte.....	5-20
5.8	Das Listener-Konzept.....	5-21
5.8.1	Event-Klassen, Listener und Methoden	5-21
5.8.2	Topologie der Listener	5-22
5.9	Designalternativen für Listener	5-23
5.9.1	Externe Klasse als Listener	5-23
5.9.2	Container als Listener	5-24
5.9.3	Innere Klasse als Listener.....	5-25
5.9.4	Lokale Klasse als Listener	5-25
5.9.5	Anonyme Klasse als Listener.....	5-26
5.9.6	Lambda-Ausdruck als Listener	5-27
5.9.7	Verwendung einer Adapter-Klasse	5-27
5.10	Zusammenfassende Wertung	5-28
6	Multithreading	6-3
6.1	Technische Einführung.....	6-3
6.2	Das Thread-API.....	6-5
6.2.1	Erzeugen eigener Threads	6-5
6.2.2	Start eines Threads	6-6
6.2.3	Subklasse von Thread	6-6
6.2.4	Implementierung von Runnable	6-7
6.2.6	Das Interface Runnable	6-8
6.2.7	Der Lebenszyklus eines Threads.....	6-10
6.2.8	Thread-Prioritäten.....	6-12
6.2.9	Daemon-Thread.....	6-13
6.2.10	Stoppen eines Threads.....	6-14
6.3	Timer und TimerTask	6-15
7	Java und Datenbanken	7-3
7.1	Überblick	7-3
7.2	Zwei- und Drei-Schicht-Modell	7-4
7.3	Das JDBC-API.....	7-5
7.4	Treibertypen	7-6
7.4.1	JDBC-ODBC-Brückentreiber	7-7
7.4.2	JDBC-DBMS API-Treiber.....	7-8
7.4.3	JDBC-Middleware-Treiber	7-8
7.4.4	JDBC-DB-Treiber.....	7-9
7.4.5	Treiber laden.....	7-9
7.5	Die Verbindung.....	7-11
7.5.1	Datenbank-URL	7-11

7.5.2	Benutzer und Kennwort	7-12
7.5.3	Verbindung öffnen.....	7-13
7.5.4	Verbindung schließen	7-14
7.5.5	Verwendung der Connection	7-15
7.6	SQL-Befehle senden	7-16
7.6.1	JDBC Statements	7-16
7.6.2	PreparedStatement und CallableStatement.....	7-18
7.7	Auswertung der Ergebnisse.....	7-20
7.7.1	ResultSet	7-20
7.7.2	Ergebnistabelle schließen.....	7-21
7.8	Zusammenfassung Datenbankzugriff	7-22
7.8.1	Komplettes Beispiel	7-22
8	Anhang.....	8-3
8.1	Eclipse IDE Setup.....	8-3
8.1.1	Download Eclipse	8-3
8.1.2	Erzeugen eines ersten Java Projektes.....	8-5
8.1.3	Eclipse Shortcuts	8-5
8.2	Operator-Rangfolge	8-7
8.3	Online-Dokumentation	8-9
8.4	Javadoc	8-10
8.5	Java – Glossar.....	8-11
	Index.....	IDX-1

1

Einführung

1.1	Inhalte des Seminars.....	1-3
1.2	Präsentation	1-3
1.3	Grundbegriffe.....	1-4
1.3.1	Laufzeitumgebung	1-5
1.3.2	Speicher der VM	1-5
1.3.3	Objektorientierte Programmierung	1-6
1.3.4	UML Klassendiagramm.....	1-8
1.4	Einige Klassen aus der Java Bibliothek.....	1-9
1.4.1	String und StringBuilder	1-10
1.4.2	Einige Methoden der Klasse String.....	1-11
1.4.3	Einige Methoden der Klasse StringBuilder.....	1-13
1.4.4	Formatierte Ausgaben	1-15
1.5	Zugriffsrechte.....	1-16
1.6	Pakete	1-17
1.6.1	Eigene Pakete erstellen	1-17
1.6.2	Import von Klassen	1-18
1.6.3	Statische Importe	1-19
1.7	Assertions.....	1-20
1.8	Fundamentale Datentypen	1-21
1.9	Typumwandlung durch Cast	1-22
1.10	Arrays	1-23

1.10.1	Mehrdimensionale Arrays	1-24
1.11	Variable Parameterliste	1-25
1.11.1	Das Problem	1-25
1.11.2	Die Lösung.....	1-26
1.12	Java Schlüsselwörter.....	1-28
1.13	Komponenten der Java Standard Edition	1-28

1 Einführung

Dieses Java-Aufbau-Seminar zeigt, wie skalierbare Anwendungen durch den Einsatz fortgeschrittener Programmierkonzepte erstellt werden können.

Als Vorkenntnisse für dieses Seminar werden erste Erfahrungen in der Java-Programmierung, der Einsatz der objektorientierten Mechanismen und der Umgang mit den JDK-Tools erwartet.

1.1 Inhalte des Seminars

- Einsatz fortgeschrittener Programmierkonzepte
- Das Collection Framework und generische Datentypen
- Dateibasierte Ein/Ausgabe
- Konfiguration einer Anwendung mit Properties-Dateien
- Einfache grafische Oberflächen entwerfen und implementieren
- Das Event Handling anwenden
- Mechanismen der parallelen Programmierung (Multithreading)
- SQL-Datenbanken und Java

1.2 Präsentation

Zu jedem Seminarschwerpunkt gibt es Übungsmodule, mit denen Sie das Gelernte praktisch umsetzen können. Die Übungen sind so gehalten, dass Sie mit Hilfe des Vorgetragenen und der Broschüre die Aufgaben lösen können. Experimentierfreudige Programmierer können die Aufgaben selbständig erweitern. Zu allen Aufgaben gibt es Musterlösungen, die mit dem Referenten besprochen werden können.

1.3 Grundbegriffe

Folgende kurz eingeführten Begriffe und Regeln sollten bekannt sein. Java ist von seiner Grundkonzeption her eine typisierte objektorientierte Programmiersprache.

Neben den einfachen Datentypen (Zahlen, Einzelzeichen und logische Werte) enthält die Laufzeitumgebung eine Bibliothek mit den Standard-Klassen.

Klassen dienen als Vorlage für Objekte:

Ein Objekt ist eine Instanz einer Klasse.

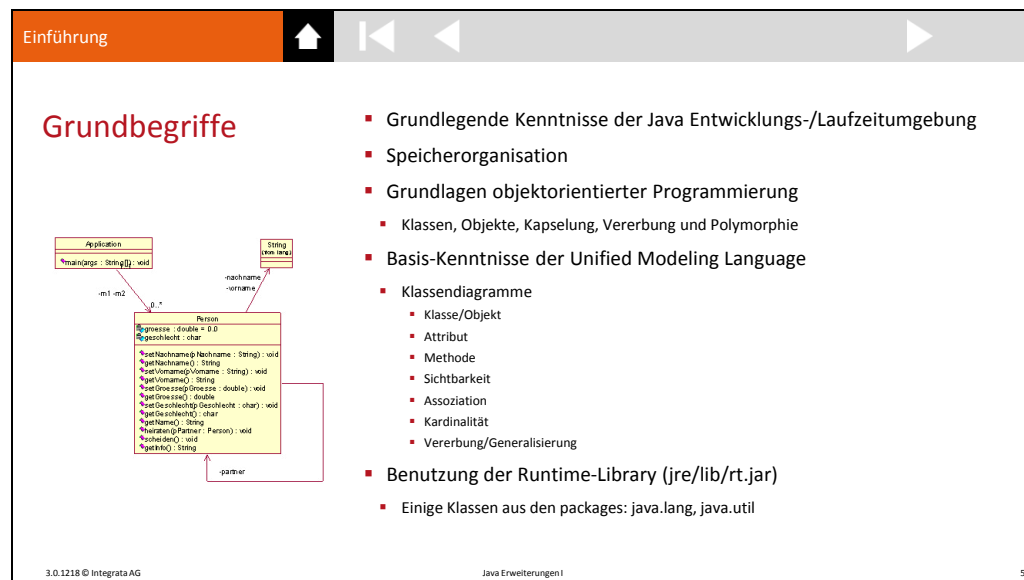


Abb. 1-1: Grundbegriffe

Die Organisation der Klassen erfolgt durch Gruppierung in die Pakete, die aufzufassen sind als Container für Klassen und andere Unterpakete. Der voll qualifizierte Klassenname schließt die Paketstruktur mit ein.

1.3.1 Laufzeitumgebung

Einführung
⬆
⏪ ⏩
⏴ ⏵

Laufzeitumgebung

- Java ist eine typisierte objektorientierte Sprache
- Die Klassenbibliothek ist organisiert in Paketen:
 - voll qualifizierter Klassenname
 - Umgebungsvariable CLASSPATH
 - Datei rt.jar
- Java-Programme laufen plattform-unabhängig in einer virtuellen Maschine
- Bytecode wird vom Java-Interpreter ausgeführt

3.0.1218 © Integrata AG
Java Erweiterungen I
6

Abb. 1-2: Laufzeitumgebung

Die Ausführung von Java-Programmen erfolgt plattformunabhängig durch die Einführung der virtuellen Maschine, einem Interpreter für Bytecode. Die virtuelle Maschine lädt Klassen zum einen aus der Standard-Bibliothek, den Klassen innerhalb der Datei `rt.jar`, andererseits wird die Umgebungsvariable `CLASSPATH` genutzt, um Anwender-Klassen zu finden. Bytecode (Dateiendung: `.class`) wird vom Java-Compiler aus einem Java-Quellcode (Endung: `.java`) generiert.

1.3.2 Speicher der VM

Einführung
⬆
⏪ ⏩
⏴ ⏵

Speicherorganisation

- Lokale Variable (Wert im Methoden-Frame)
- Variable auf dem Stack:
 - haben nur eine begrenzte Lebensdauer
 - werden nicht automatisch initialisiert
- Objekt-Attribute
 - sind gültig, solange die Instanz existiert
 - werden automatisch initialisiert
- Einfache Variablen
 - werden durch einen Wert des entsprechenden Datentyps repräsentiert
- Referenzen und Arrays
 - Wert wird interpretiert als Adresse eines Bereiches im gemeinsamen Heap-Speicher
- Reservieren/Belegen von Speicher durch das Schlüsselwort **new**

3.0.1218 © Integrata AG
Java Erweiterungen I
7

Abb. 1-3: Speicherorganisation

Der Speicher der virtuellen Maschine ist in zwei unterschiedlichen Bereichen organisiert:

Im Methoden-Frame sind alle Variablen, die innerhalb eines Methodenaufrufes lokal deklariert sind sowie die Übergabeparameter enthalten. Im Heap-Speicher der virtuellen Maschine werden alle Objekte und Arrays abgelegt.

Der Zugriff auf den Heap-Speicher erfolgt stets über eine Referenz. Im Bild sind die beiden Variablen `counter` und `test` einfache Datentypen, `name` ist ein `java.lang.String` und `zahlen` ein `integer-Array`.

Arrays dienen als typisierte Container für jeweils einen bestimmten Datentyp und werden über einen numerischen Index angesprochen.

1.3.3 Objektorientierte Programmierung

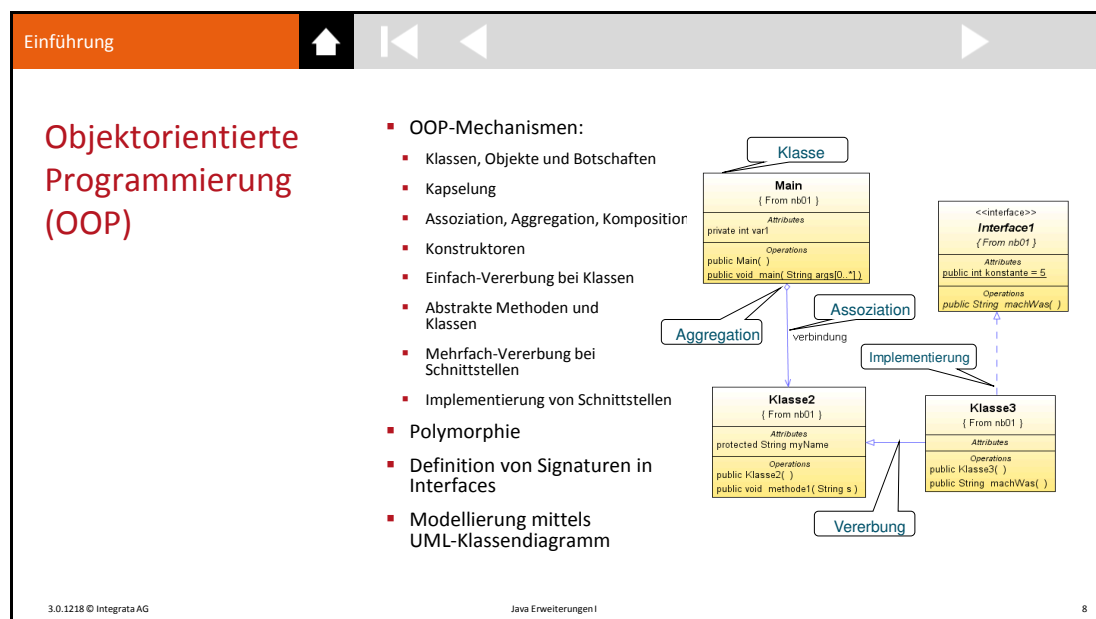


Abb. 1-4: OOP

Zur Erzeugung von Objekten und Arrays wird das Schlüsselwort **new** verwendet. Die Verwendung dieses Operators reserviert automatisch den benötigten Heap-Speicher, ruft einen Konstruktor der Klasse auf und gibt den Wert der Referenz zurück.

Auch eigene Typen können definiert werden. Java beschränkt sich auf das Erstellen eigener Klassen; einfache Datentypen oder Datenstrukturen können nicht neu definiert werden.

Eigene Klassen werden stets durch die Werkzeuge der objektorientierten Programmierung erstellt. Java kennt zwei unterschiedliche Gruppierungen: Klassen und Schnittstellen, sog. Interfaces.

Klassen enthalten die Deklaration von Attributen und Methoden inklusive Definitionen sowie eventuell von Konstruktoren. Interfaces enthalten statische konstante Attribute und abstrakte Methoden und definieren so eine Klassensignatur. Seit JDK 8 können Interfaces auch implementierte (Default-)Methoden, Schlüsselwort: **default**, haben.

Java unterstützt folgende objektorientierte Konzepte:

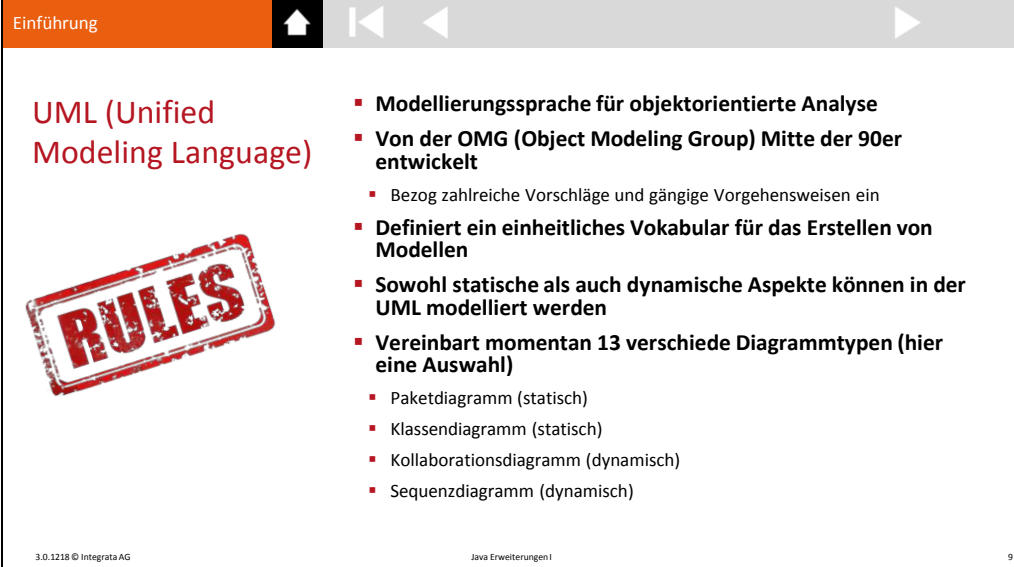
- Komposition durch Einführung von Attributen und dadurch Objekt-Verbindungen.
- Vererbung ermöglicht die Wiederverwendung von Definitionen einer Superklasse oder eines Superinterfaces in einer Subklasse oder einem Subinterface. Java unterstützt durch das Schlüsselwort **extends** für Klassen die Einfachvererbung.
- Die Implementierung von Schnittstellen erweitert die Klassensignatur um die Signatur der implementierten Interfaces. Dazu verwendet Java das Schlüsselwort **implements**. Es können mehrere Interfaces in einer Klasse implementiert werden.

Die Verwendung von Interface-Typen statt konkreter Klassenimplementierungen erhöht die Wiederverwendbarkeit.

Erweitern Sie Ihre Klassen nicht durch unkontrolliertes Hinzufügen von Methoden! Definieren Sie stattdessen ein Interface und implementieren dieses! Methoden können in Java überschrieben werden, Aufrufe sind stets polymorph.

1.3.4 UML Klassendiagramm

Klassenhierarchien können übersichtlich grafisch in einem UML-Klassendiagramm präsentiert werden.



UML (Unified Modeling Language)

RULES

- Modellierungssprache für objektorientierte Analyse
- Von der OMG (Object Modeling Group) Mitte der 90er entwickelt
 - Bezog zahlreiche Vorschläge und gängige Vorgehensweisen ein
- Definiert ein einheitliches Vokabular für das Erstellen von Modellen
- Sowohl statische als auch dynamische Aspekte können in der UML modelliert werden
- Vereinbart momentan 13 verschiedene Diagrammtypen (hier eine Auswahl)
 - Paketdiagramm (statisch)
 - Klassendiagramm (statisch)
 - Kollaborationsdiagramm (dynamisch)
 - Sequenzdiagramm (dynamisch)

3.0.1218 © Integrata AG Java Erweiterungen I 9

Abb. 1-5: UML-Klassendiagramm

Sowohl Klassen als auch Pakete unterstützen das Prinzip der Kapselung durch die Einführung der Schlüsselwörter **public**, **protected**, **private** sowie dem Standard, der Paketsichtbarkeit.

Gekapselte Klassen sind für sich selber verantwortlich.

1.4 Einige Klassen aus der Java Bibliothek

`java.lang.Object` ist die Elternklasse aller Klassen und definiert einige wichtige Methoden, die allen Objekten zur Verfügung stehen. Diese Klasse wird im nächsten Kapitel ausführlich erläutert.

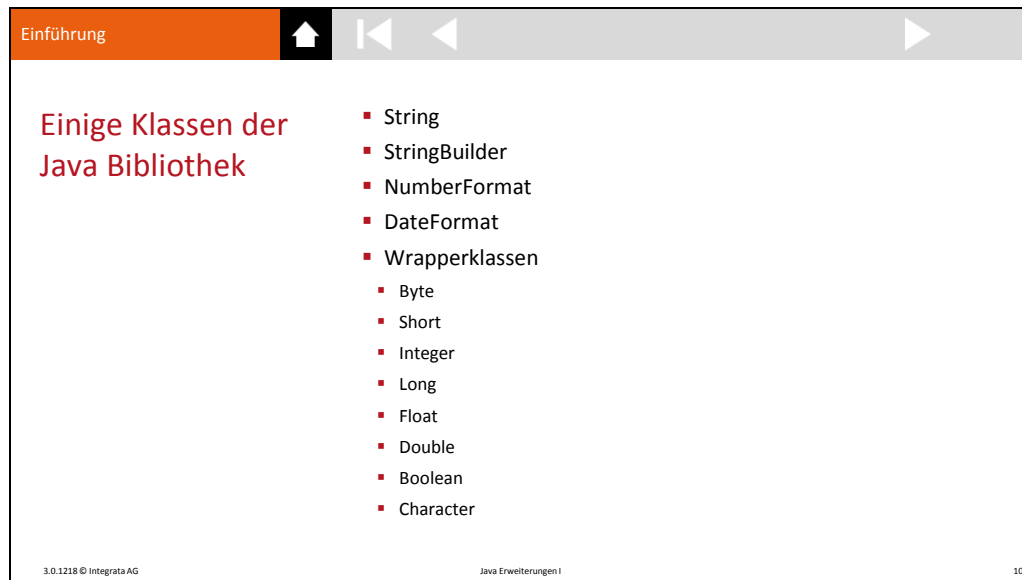


Abb. 1-6: Einige Klassen

Wrapper-Klassen ermöglichen es, einfache Datentypen als Objekte aufzufassen. Ein Wrapper ist eine Klasse, die einen anderen Datentyp als Attribut enthält und den Zugriff auf dieses regelt bzw. mit neuen Methoden erweitert. Die Wrapper-Klassen aus `java.lang` werden verwendet, wenn einfache Datentypen wie `short`, `boolean`, `int`, `double` usw. als Objekte behandelt werden sollen. Diese Klassen umfassen nützliche Werkzeuge und vor allem Konvertierungsmethoden.

Die Klassennamen der Wrapper-Klassen ergeben sich aus den elementaren Datentypen, indem in Konsistenz mit der Namenskonvention der erste Buchstabe groß geschrieben wird. So wird z. B. aus dem elementaren Datentyp `double` ein Objekt vom Typ der Wrapper-Klasse `java.lang.Double`.

Ausnahmen dieses Verfahrens sind die Datentypen `int` und `char`; `int` wird in `java.lang.Integer`, `char` in `java.lang.Character` abgebildet.

Die Werkzeuge der objektorientierten Programmierung sind die Einstiegspunkte für eine robuste Programmier-Strategie. Häufig wiederkehrende Problemstellungen können durch **Entwurfsmuster** (Designpattern) gelöst werden. Eine erste Auswahl aus dem Katalog standardisierter Entwurfsmuster enthält:

- das **Singleton**, mit dem sichergestellt werden kann, dass exakt eine Instanz einer Klasse zentral pro virtueller Maschine zur Verfügung gestellt wird,
- Den **Wrapper**, der ein vorhandenes Objekt mit zusätzlicher Funktionalität ausstattet, und schließlich
- die **Factory**, die die Instanziierung von Objekten kapselt.

Vermeiden Sie in Ihren Programmen die unkontrollierte Verwendung von `new`. Kapseln Sie stattdessen die Objekt-Erzeugung in speziellen Factory-Hilfsklassen.

1.4.1 String und StringBuilder

Unveränderbare Zeichenketten sind Instanzen der Klasse `java.lang.String`. Zur Zeichenkettenverarbeitung ist die Klasse `java.lang.StringBuffer` bzw. seit Version 5.0 `java.lang.StringBuilder` vorgesehen.

Mittels des Plusoperators können Strings verkettet werden. Strings können wie folgt erzeugt werden:

```
String txt = "Java und Coffee";
```

oder mit Verkettung:

```
String txt = "Java" + " und" + " Coffee";
```

Pro Zeichen werden 2 Bytes verbraucht, so dass Unicodezeichen im String verwendet werden können. Eine von Gänsefüßchen eingeschlossene Zeichenkette ist immer ein Objekt vom Typ `String`. Die Virtuelle Maschine implementiert intern einen Cache für `String`-Litereale und prüft bei jeder Zuweisung, ob der gewünschte Wert bereits vorhanden ist:

```
String s1 = "Hello";           //Hier wird "Hello" in die Liste ein-
                                getragen
String s2 = "Hello";           //Hier wird die Referenz auf das be-
                                reits vorhandene "Hello" zurückgegeben

boolean result = (s2 == s1); //true
```

Die Verkettung von Strings mit Hilfe des Plusoperators ist sehr unperformant. Für Zeichenkettenverarbeitung wird besser der `StringBuffer` oder der `StringBuilder` verwendet.

Ein Objekt vom Typ `StringBuilder` kann wie folgt erzeugt werden:

```
StringBuilder sb = new StringBuilder("Dies ist ein Text");  
StringBuilder sbig = new StringBuilder( 100 );
```

Eine Initialisierung der Form

```
StringBuilder s = "Dies ist ein Text";
```

führt natürlich zu einem Fehler (links `StringBuilder`, rechts `String`).

1.4.2 Einige Methoden der Klasse `String`

`int length()` liefert die Länge eines Strings.

`char charAt(int pos)` liefert das Zeichen an der Position `pos`.

`boolean equals(String s)`

reicht "true" zurück, wenn die beiden Strings denselben Inhalt haben, also wenn die Längen, die Zeichen und deren Reihenfolgen in den Strings identisch sind.

`boolean equalsIgnoreCase(String s)`

wie zuvor, aber unabhängig von Groß- oder Kleinschreibung.

`String trim()` entfernt führende und endende Leerzeichen.

`int compareTo(String s)`

Vergleich zweier Stringobjekte. Zurückgegeben wird ein Wert, der positiv ist, wenn das übergebene String-Argument in der Sortierreihenfolge vor dem aufrufenden String-Objekt kommt, null, wenn beide gleich sind, und negativ, wenn das Argument nach dem aufrufenden Objekt kommt.

`boolean regionMatches(int toffset, String other, int offset, int len)`

reicht "true" zurück, wenn der Bereich des einen Strings mit dem Bereich des anderen String übereinstimmt. Die Parameter sind:

<code>toffset</code>	Startposition im laufenden String
<code>other</code>	der andere String
<code>ooffset</code>	Startposition im anderen String (<code>other</code>)
<code>len</code>	Anzahl der zu vergleichenden Zeichen

`String substring(int start, int ende)`

reicht den String zwischen Start- und Ende-Position zurück, das Zeichen an der Position `ende` ist nicht mit enthalten.

`String concat(String s)` Stringverkettung, ähnlich `+-Operator`.

`String toLowerCase()` Umwandlung von Groß- nach Kleinbuchstaben.

`String valueOf(Datentyp x)` liefert einen String, welcher die Darstellung des Wertes vom Typ `x` hat. `Datentyp` kann sein: `int`, `float`, `double`, `long`, `char`, `boolean` oder `Object`.

Das folgende Beispiel verdeutlicht die Anwendung dieser Methoden:

Beispiel: String1.java

```
class String1 {

    public static void main( String [] args ){
        String s1 = new String("Java and Coffee");
        String s2 = new String("Java und Capuccino");

        System.out.println("\tLaenge s1: " + s1.length() );

        char c;
        c = s1.charAt(2);
        System.out.println("\t3. Zeichen : " + c );

        boolean b;
        b = s1.equals( s2 );
        System.out.println("\tVergleich 1: " + b );

        b = s1.equalsIgnoreCase( "java and coffee" );
        System.out.println("\tVergleich 2: " + b );

        int i;
        i = s1.compareTo("JavA and Coffee");
        System.out.println("\tVergleich 3: " + i );

        b = s1.regionMatches(0, s2, 0, 5 );
        System.out.println("\tVergleich 4: " + b );
        b = s1.regionMatches(0, s2, 0, 6 );
        System.out.println("\tVergleich 5: " + b );

        String s3 = s1.substring(3, 7);
        System.out.println("\tSubstring : " + s3 );

        s3 = s1.concat( s2 );
        System.out.println("\tVerkettung : " + s3 );

        s3 = s1.toLowerCase( );
        System.out.println("\ttoLower : " + s3 );

        String s4 = s3.toString();
        System.out.println("\tKopie : " + s4 );

        double x = -123.45;
        String s5 = s4.valueOf( x );
        System.out.println("\tValue of : " + s5 );

    }

}
```

Die Ausgabe lautet:

Laenge s1: 15

3. Zeichen: v

Vergleich 1: false

Vergleich 2: true

Vergleich 3: 32

Vergleich 4: true

Vergleich 5: false

Substring: a an

Verkettung: Java and CoffeeJava und Capuccino

toLowerCase: java and coffee

Kopie: java and coffee

Value of: -123.45

1.4.3 Einige Methoden der Klasse StringBuilder

`int length()` liefert die Länge.

`void setLength(int NeueLaenge)`

neue Länge des Buffers festlegen. So kann beispielsweise ein String durch Angabe einer geringeren Länge gekürzt werden.

`char charAt(int pos)`

liefert das Zeichen an der Position pos zurück.

`void setCharAt(int pos, char c)`

ändert das Zeichen an der Position pos in das Zeichen c.

`String toString()`

liefert ein Stringobjekt mit den Zeichen des Stringbuffers.

`StringBuffer append(String s)`

Ein String wird an das Ende des Buffers angefügt. Sinnvollerweise ist der Rückgabewert dieser Methode der eben verlängerte `StringBuilder` selber, so dass eine einfache Verkettung möglich ist:

```
StringBuilder lBuffer = new StringBuilder();
```

```
lBuffer.append("Hello").append(" ").
```

```
append("World").append('\n');
```

`StringBuilder insert(int pos, String s)`

an der Position pos wird ein String eingefügt.

Das folgende Beispiel verdeutlicht diese Methoden:

Beispiel: String2.java

```
class String2 {

    static public void main( String [] args )
    {
        StringBuilder b = new StringBuilder("Integrata");

        System.out.println("\tLaenge : " + b.length() );

        b.setLength( 3 );
        System.out.println("\tVerkleinern : " + b );

        b.setLength( 20 );
        System.out.println("\tVergroessern : " + b );

        b.setCharAt(10, 'X' );
        System.out.println("\tZeichen setzen : " + b );

        String s = b.toString();
        System.out.println("\tString erzeugen : " + s );

        b.append(" 1234567890");
        System.out.println("\tString anhaengen : " + b );

        b.insert(4, "_____");
        System.out.println("\tString einfuegen : " + b );

    }
}
```

Die Ausgabe lautet:

Laenge: 9

Verkleinern: Int

Vergroessern: Int

Zeichen setzen: Int X

String erzeugen: Int X

String anhaengen: Int X 1234567890

String einfuegen: Int _____ X 1234567890

1.4.4 Formatierte Ausgaben

Dazu dient das Paket `java.text` mit den Klassen

`NumberFormat`

`DateFormat`

Länderspezifische Einstellungen werden durch vordefinierte Konstanten der Klasse `Locale` vorgenommen.

Die Verwendung dieser Klassen ist sehr einfach:

```
public class DecimalFormat1 {  
  
    public static void main(String args[]) {  
  
        NumberFormat nf1 = NumberFormat.getInstance();  
  
        System.out.println(nf1.format(1234.56));  
  
        NumberFormat nf2 = NumberFormat.getInstance(Locale.GERMAN);  
  
        System.out.println(nf2.format(1234.56));  
    }  
}
```

Die Ausgabe lautet ausgeführt mit einem deutsch konfigurierten Betriebssystem:

1.234,56

1.234,56

Zu beachten ist der Tausender-Punkt und das Komma als Dezimaltrennzeichen.

In der amerikanischen Einstellung ergibt sich jedoch:

1,234.56

1.234,56


1.5 Zugriffsrechte

Nach der Einführung des Begriffs **Import** können wir nun die vollständige Bedeutung des Schlüsselwortes `protected` nachreichen bzw. eine allgemeine Übersicht über die Zugriffsrechte in Java liefern:

Einführung

⏮ ⏪ ⏩ ⏭

Datenkapselung



- **private**
Zugriff nur innerhalb der deklarierenden Klasse möglich
- **<default>**
Zugriff nur innerhalb des Paketes möglich
- **protected**
Zugriff innerhalb der deklarierenden Klasse und innerhalb aller Subklassen und innerhalb des Paketes möglich
- **public**
Zugriff aus allen Klassen heraus möglich

3.0.1218 © Integrata AG

Java Erweiterungen I

11

Abb. 1-7: Datenkapselung

Auch Klassen werden durch Pakete gekapselt: Klassen können mit Hilfe des Schlüsselwortes `public` als öffentlich deklariert werden. Nur dann ist eine Verwendung in anderen Paketen möglich. Wird dieses Schlüsselwort weggelassen, wird die Paketsichtbarkeit verwendet. Diese ist beschränkt auf die Klassen des eigenen Paketes. Dies ist natürlich erneut sehr praktisch, weil auf diese Art und Weise beispielsweise der Zugriff auf Hilfsklassen anderen Paketen verboten werden kann. Wichtiger ist hier jedoch die Möglichkeit, aus Modellierungsgründen den Zugriff zu verbieten. So ist es möglich, eine **Paketsignatur** zu definieren und Pakete als Ganzes über die `public`-Klassen zu koppeln.

1.6 Pakete

Die Festlegung der Zugehörigkeit einer Klasse zu einem Paket wird durch die Anweisung:

```
package paketname;
```

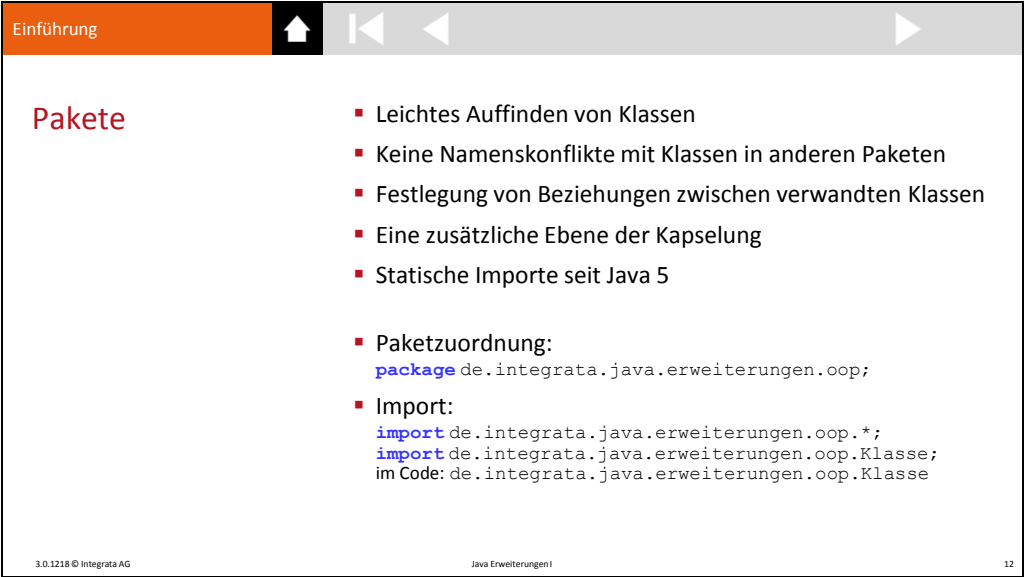
erreicht, welche als erste Anweisung im Quellcode stehen muss.

Beispiel:

```
package de.integrata.java.erweiterungen;
```

1.6.1 Eigene Pakete erstellen

Wir hatten bereits gesehen, dass in der ersten Anweisung einer Java-Sourcdatei die Zugehörigkeit einer Klasse zu einem Paket (package) definiert wird. In einem Paket werden verwandte Klassen und Schnittstellen zusammengefasst. Vorteile dieser Bündelung sind:



Pakete

- Leichtes Auffinden von Klassen
- Keine Namenskonflikte mit Klassen in anderen Paketen
- Festlegung von Beziehungen zwischen verwandten Klassen
- Eine zusätzliche Ebene der Kapselung
- Statische Importe seit Java 5

- Paketzurordnung:
`package de.integrata.java.erweiterungen.oop;`
- Import:
`import de.integrata.java.erweiterungen.oop.*;`
`import de.integrata.java.erweiterungen.oop.Klasse;`
im Code: `de.integrata.java.erweiterungen.oop.Klasse`

3.0.1218 © Integrata AG Java Erweiterungen I 12

Abb. 1-8: Pakete

Wird keine Paketfestlegung getroffen, werden die im Programm enthaltenen Informationen in einem anonymen Paket gesammelt und die Klassen können nur von Programmen im selben Verzeichnis benutzt werden. Dies ist für einfache Testprogramme zulässig, in der Praxis aber nicht häufig anzutreffen. Deshalb werden die verwendeten Klassen auch sofort den Paketen

```
de.integrata.java.erweiterungen.sprache bzw.  
de.integrata.java.erweiterungen.oop
```

zugeordnet. Eine feinere Unterteilung ist jedoch sicherlich möglich und auch sinnvoll.

1.6.2 Import von Klassen

Durch den **Import** von Klassen aus Paketen können diese auch verkürzt angesprochen werden.

Zur Importierung von Klassen steht das Schlüsselwort `import` zur Verfügung:

```
import de.integrata.java.erweiterungen.oop.Klasse;
```

Hier wird die Klasse `Klasse` aus dem Paket

```
de.integrata.java.erweiterungen.oop
```

verkürzt ansprechbar.

Import aller Klassen oder Interfaces eines Pakets ist durch die Angabe eines "`*`" möglich:

```
import de.integrata.java.erweiterungen.oop.*;
```

Hier ist anzumerken, dass die Typen eines Unterpaketes nicht mit importiert werden. Diese müssten in weiteren Anweisungen angegeben werden.

Importe der Form

```
import de.integrata.java.erweiterungen.oop.Ar*;
```

oder

```
import de.integrata.java.erweiterungen.*.*
```

sind nicht zulässig und liefern einen Compilerfehler.

```
import java.*
```

importiert keine einzige Klasse.

Das Java Laufzeitsystem importiert defaultmäßig zwei Pakete:

- das Paket ohne Namen
- das Paket `java.lang`

Pakete bündeln Gruppen von Klassen in eine Bibliothek, die unter dem Namen des Paketes angesprochen wird.

1.6.3 Statische Importe

Mit der Sprachversion 5.0 wurde die `import`-Anweisung erweitert: Neben der oben angesprochenen Möglichkeit, ganze Klassen zu importieren, können auch statische Attribute und Methoden angegeben werden. Dazu dient die Anweisung `import static`.

Mit der neuen Anweisung `import static` ist eine direkte Benutzung möglich, falls keine Namenskonflikte auftreten können:

```
import static java.lang.Math.*;
public class CircleUtil {

    public static void main(String[] args) {
        CircleUtil util = new CircleUtil();
        double radius = 2;
        System.out.println("Area: " + util.area(radius));
    }

    public double area(double radius) {
        return pow(radius, 2d) * PI;
    }
}
```

Hinweis: Wie beim Importieren von Klassen ist auch hier der "`*`" zulässig, alle statischen Attribute und Methoden in einer Anweisung anzugeben. Im obigen Beispiel hätten auch statische Imports der Form

```
import static java.lang.Math.PI;
import static java.lang.Math.pow;
genügt. Er ist nicht mit einem Platzhalter-Zeichen zu verwechseln:
import static java.lang.Math.P*;
ist syntaktisch falsch.
```

Überladenen Methoden werden komplett importiert. So würde ein Import der Form `import static java.lang.Math.abs` insgesamt 4 Methoden statisch importieren (mit `double`, `float`, `int` und `long` als Parameter).

Auch bei statischen Imports müssen die Klassennamen stets voll qualifiziert angegeben werden. `import static Math.PI` ist syntaktisch falsch.

1.7 Assertions

Neben den `Exception`-Klassen bietet Java auch ein Schlüsselwort an, das während des Programm-Tests sehr sinnvoll eingesetzt werden kann: `assert`. Eine Assertion, zu Deutsch: Behauptung, wird ähnlich wie eine `if`-Abfrage geschrieben:

```
assert <Bedingung>;
```

Ist die Bedingung erfüllt, so wird die nächste Programmzeile ausgeführt. Ist die Assertion jedoch falsch, bricht das Programm mit einem `AssertionError` ab.

Assertions wurden erst mit der Java-Version 1.4 eingeführt und es stellt sich natürlich die Frage, was diese Konstruktion denn eigentlich bringen soll.

Assertions sollen einen klaren Fehler der Anwendung signalisieren. Nach einer Assertion ist das Programm nicht mehr sinnvoll weiter zu führen. Damit würde sich das Werfen einer speziellen

`RuntimeException` anbieten, die dann jedoch in der normalen Fehlerbehandlung untergehen würde. Die Verletzung einer Assertion muss beim Test des Programms ein Error sein!

Daneben sind Assertions tief in die Laufzeitumgebung integriert worden. Assertions müssen nach erfolgreichem Test von der Virtuellen Maschine nicht mehr beachtet werden. Deshalb wurde der Aufruf der Virtuellen Maschine durch die Optionen `-ea` (enable assertions) und `-da` (disable assertions) erweitert. Beim Programmstart können Assertions bis hin zur Klassenebene aktiviert oder deaktiviert werden.

Die Ausdrücke in Assertions sollen keine Nebeneffekte enthalten, d.h. Code, der z.B. Variablen verändert. Der Grund dafür ist, dass diese bei abgeschalteten Assertions nicht ausgeführt werden. Das Programm würde sich also anders verhalten, als wenn die Assertions angeschaltet wären. Dadurch würden sehr schwer zu findende Fehler erzeugt.

1.8 Fundamentale Datentypen

Alle Variablen in Java müssen von einem bestimmten Datentyp sein. Der Typ bestimmt den Wert, den eine Variable annehmen kann, und die Operationen, welche auf dieser Variablen zulässig sind. Die Definition einer Variablen ist wie folgt:

```
typ bezeichner = Wert;
```

Mehrere Bezeichner können, durch Kommata getrennt, hinter einem Typ deklariert werden. Die Wertzuweisung an dieser Stelle ist optional, muss aber vor dem ersten Auslesen der Variable erfolgen, es ist nicht zulässig, Variable zu verwenden, bevor ihnen ein Wert zugewiesen wurde.

Variablen können im Programm an beliebiger Stelle definiert werden.

Per Konvention beginnen Variablennamen immer mit einem Kleinbuchstaben und sollen möglichst sprechende Namen bekommen.

1.9 Typumwandlung durch Cast

Java ist eine streng typisierte Programmiersprache, der Compiler prüft auf korrekten Typ bei Zuweisungen und Übergabeparametern. Eine implizite Typumwandlung, ein Cast, ist aus einem „*kleineren*“ in einen „*größeren*“ Datentyp implizit möglich, sollte jedoch aus Gründen der Übersichtlichkeit vermieden werden. Eine Umwandlung in den „*kleineren*“ oder einen nur kompatiblen Datentyp ist zwingend explizit. Die Syntax lautet:

```
wertNeuerTyp = (NeuerTyp) wertAlterTyp;
```

Das folgende Beispiel verdeutlicht die Typumwandlung durch den Cast:

```
class Cast {  
    public static void main(String[] args) {  
        int iZahl;  
        double dZahl;  
        float fZahl;  
        dZahl = 1;      //Konvertierung implizit möglich  
        dZahl = 1.0;  
        //fZahl = 1.0; //geht nicht: Abwärts cast nur explizit  
        fZahl = 1.0f; //f macht die Zahl zum float;  
        iZahl = (int)1.0; //Expliziter cast  
        dZahl = 1/2;  
        System.out.println(dZahl); //Ausgabe 0.0, ???  
        dZahl = 1.0/2.0;  
        System.out.println(dZahl); //Ausgabe 0.5,  
        dZahl = (double)1/2;  
        System.out.println(dZahl); //Ausgabe 0.5  
        dZahl = (double)(1/2);  
        System.out.println(dZahl); //Ausgabe 0.0  
    }  
}
```

1.10 Arrays

Ein Array ist die Aneinanderreihung von Daten desselben Typs. Über einen Index kann dann auf das gewünschte Element eines Arrays zugegriffen werden. Der Zugriff auf das erste Element erfolgt immer über den Index 0. Mit der Initialisierung eines Arrays wird eine Größe `length` erzeugt, welche als Wert die Anzahl der Elemente enthält.

Unter Java gibt es zwei Möglichkeiten, ein Array anzulegen:

1. Mit Initialisierung durch Aufzählung

```
i n t [ ] a r r a y 1 = { 2, 3, 4, 5, 6 } ;
```

2. Mit `new` und Nullinitialisierung

```
i n t [ ] a r r a y 2 = n e w i n t [ 5 ] ;
```

In diesem Fall werden alle Arrayelemente mit dem Wert 0 initialisiert. Eine Wertzuweisung kann komponentenweise erfolgen:

```
a r r a y 2 [ 0 ] = 2 ;  
a r r a y 2 [ 1 ] = 3 ;  
a r r a y 2 [ 2 ] = 4 ;  
a r r a y 2 [ 3 ] = 5 ;  
a r r a y 2 [ 4 ] = 6 ;
```

Der Wert `array1.length` liefert wie `array2.length` den Wert 5.

Im nächsten Fall wird nochmals eine Referenz auf ein Array angelegt:

```
i n t [ ] a r r a y 3 ;
```

Die Initialisierung kann dann später erfolgen:

```
a r r a y 3 = n e w i n t [ 100 ] ;
```

Zu beachten ist, dass in dem eckigen Klammern Paar links der Zuweisung niemals eine Konstante oder Variable stehen darf.

1.10.1 Mehrdimensionale Arrays

Es gibt in Java keine mehrdimensionalen Arrays, sondern Arrays von Arrays. So kann ein Array definiert werden, dessen Elemente Integer-Arrays sind:

```
int[][] multiInt = new int[2][3];
```

Hier wird ein Array aus 2 Komponenten angelegt. Jedes Element ist seinerseits eine Referenz auf ein Array von drei `int`-Werte. Die Initialisierung kann dann komponentenweise erfolgen:

```
multiInt[0][0] = 1;      multiInt[0][1] = 2;  
multiInt[0][2] = 3;      multiInt[1][0] = 4;  
multiInt[1][1] = 5;      multiInt[1][2] = 6;
```

Definition und Initialisierung können auch in einem Schritt erfolgen:

```
int[][] multiInt = { {1, 2, 3 }, {4, 5, 6 } };
```

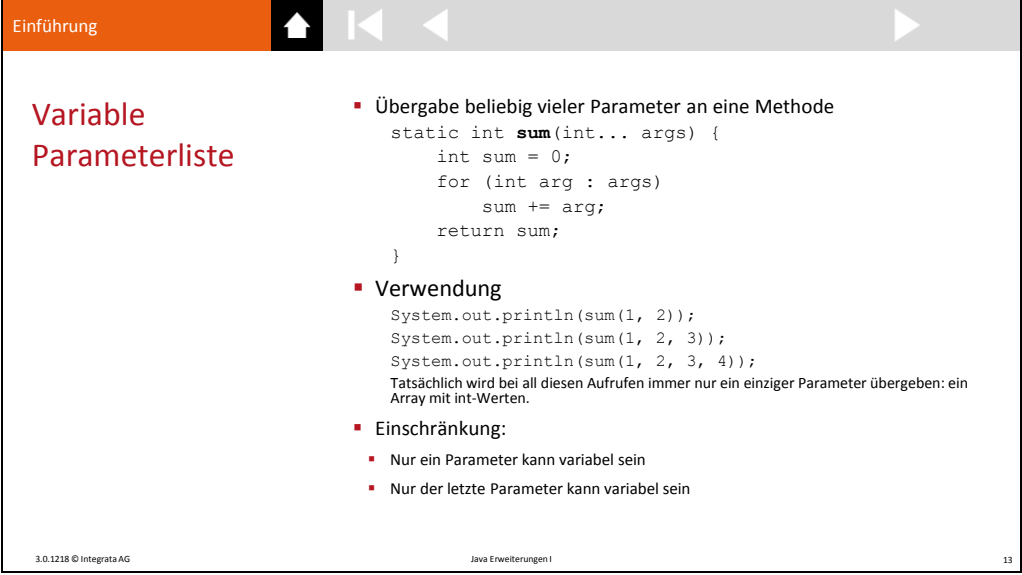
Zweidimensionale Arrays brauchen nicht "rechteckig" zu sein, daher sind auch folgende Gebilde möglich.

```
double[][] multiDouble = { {1.0}, {1.0, 2.0}, {1.0, 2.0, 3.0}};
```

Dasselbe gilt für Arrays mit mehr als zwei Dimensionen, welche jedoch in der Praxis selten auftreten.

1.11 Variable Parameterliste

Seit Java 5 gibt es bei Methoden die Möglichkeit, beliebig viele Parameter eines Typs zu übergeben.



The screenshot shows a presentation slide with a title bar 'Einführung' and navigation icons. The main content is titled 'Variable Parameterliste' in red. It contains three bullet points: 'Übergabe beliebig vieler Parameter an eine Methode' with a code snippet for a `sum` method; 'Verwendung' with code snippets showing calls to `sum` with 2, 3, and 4 arguments; and 'Einschränkung:' with two sub-bullets stating that only one parameter can be variable and it must be the last one. The footer includes '3.0.1218 © Integrata AG', 'Java Erweiterungen I', and the page number '13'.

Variable Parameterliste

- Übergabe beliebig vieler Parameter an eine Methode

```
static int sum(int... args) {
    int sum = 0;
    for (int arg : args)
        sum += arg;
    return sum;
}
```
- Verwendung

```
System.out.println(sum(1, 2));
System.out.println(sum(1, 2, 3));
System.out.println(sum(1, 2, 3, 4));
```

Tatsächlich wird bei all diesen Aufrufen immer nur ein einziger Parameter übergeben: ein Array mit int-Werten.
- Einschränkung:
 - Nur ein Parameter kann variabel sein
 - Nur der letzte Parameter kann variabel sein

3.0.1218 © Integrata AG Java Erweiterungen I 13

Abb. 1-9: VarArgs

1.11.1 Das Problem

Man möchte die Summe beliebig vieler Werte bestimmen können. Man könnte eine `sum`-Methode überladen:

```
static int sum(int arg0, int arg1) {
    return arg0 + arg1;
}
static int sum(int arg0, int arg1, int arg2) {
    return arg0 + arg1 + arg2;
}
static int sum(int arg0, int arg1, int arg2, int arg3) {
    return arg0 + arg1 + arg2 + arg3;
}
static int sum(int[] args) {
    int sum = 0;
    for (int arg : args)
        sum += arg;
    return sum;
}
```

Um die Summe von bis zu vier Werten könnte dann einer der ersten drei `sum`-Methoden aufgerufen werden. Für mehr als vier Werte müsste die vierte Methode aufgerufen, welche verlangt, dass die zu summierenden Werte in Form eines Arrays übergeben werden. (Diese vierte Methode könnte man natürlich auch dann verwenden, wenn weniger als vier Werte zu summieren wären.)

Hier eine mögliche Anwendung dieser Methoden:

```
System.out.println(sum(1, 2));
System.out.println(sum(1, 2, 3));
System.out.println(sum(1, 2, 3, 4));
// System.out.println(sum(1, 2, 3, 4, 5)); // illegal
System.out.println(sum(new int[] { 1, 2, 3, 4, 5 }));
```

Hier steckt offensichtlich ein gewisser Bruch in der Logik: bis zu vier Werten können summiert werden, ohne einen expliziten Array dieser Werte zu übergeben; für mehr als vier Werte muss aber genau ein solcher Array übergeben werden. (Man kann die Reihe der ersten drei `sum`-Methoden natürlich nicht "beliebig" fortsetzen.)

1.11.2 Die Lösung

Seit Java 5 kann eine einzige `sum`-Methode definiert werden, welcher scheinbar(!) beliebig viele Werte übergeben werden können:

```
static int sum(int... args) {
    int sum = 0;
    for (int arg : args)
        sum += arg;
    return sum;
}
```

Diese Methode kann wie folgt aufgerufen werden:

```
System.out.println(sum(1, 2));
System.out.println(sum(1, 2, 3));
System.out.println(sum(1, 2, 3, 4));
System.out.println(sum(1, 2, 3, 4, 5));
```

Aber der Schein trügt. Tatsächlich wird bei all diesen Aufrufen immer nur ein einziger Parameter übergeben: ein Array mit `int`-Werten. Da der Parameter von `sum` vom Typ `int...` ist, wird der Compiler diese Zeilen zunächst einmal neu schreiben (rewriting):

```
System.out.println(sum(new int[] {1, 2}));
System.out.println(sum(new int[] {1, 2, 3}));
System.out.println(sum(new int[] {1, 2, 3, 4}));
System.out.println(sum(new int[] {1, 2, 3, 4, 5}));
```

Und diese Code-Ergänzungen, die der Compiler automatisch vornimmt, hätte man auch im Programmtext selbst explizit notieren können. Auch die letzten vier Aufrufe sind also legale Aufrufe der oben definierten `sum`-Methode.

Und bei der `sum`-Methode selbst betrachtet der Compiler den Typ `int...` als äquivalent zu `int[]`: er betrachtet `args` einfach als einen Array von `int`, über dessen Elemente man dann z.B. mit der neuen `for`-Schleife iterieren kann.

Natürlich könnte die obige `sum`-Methode auf wie folgt aufgerufen werden:

```
System.out.println(sum(1));  
System.out.println(sum());
```

Im letzten Fall würde der Compiler dafür sorgen, dass an `sum` ein leeres Array-Objekt übergeben wird (nicht `null`!).

Man sollte die `sum`-Methode daher besser wie folgt definieren:

```
static int sum(int arg0, int arg1, int... args) {  
    int sum = arg0 + arg1;  
    for (int arg : args)  
        sum += arg;  
    return sum;  
}
```

Dann müssten beim Aufruf mindestens zwei Werte notiert werden – was natürlich eher im Sinne einer Summierung ist.

Man beachte, dass nur der letzte Parameter einer Methode als VarArg-Parameter (als `T...`) definiert werden kann. Eine Methode kann also immer nur einen einzigen VarArg-Parameter haben – und dieser muss der letzte sein.

1.12 Java Schlüsselwörter

Einführung	↑	◀	◀	▶	
Schlüsselwörter					
abstract	continue	for	new	switch	
assert	default	(goto)	package	synchronized	
boolean	do	if	private	this	
break	double	implements	protected	throw	
byte	else	import	public	throws	
case	enum	instanceof	return	transient	
catch	extends	int	short	try	
char	final	interface	static	void	
class	finally	long	strictfp	volatile	
(const)	float	native	super	while	
3.0.1218 © Integrata AG	Java Erweiterungen I				14

Abb. 1-10: Schlüsselwörter

1.13 Komponenten der Java Standard Edition

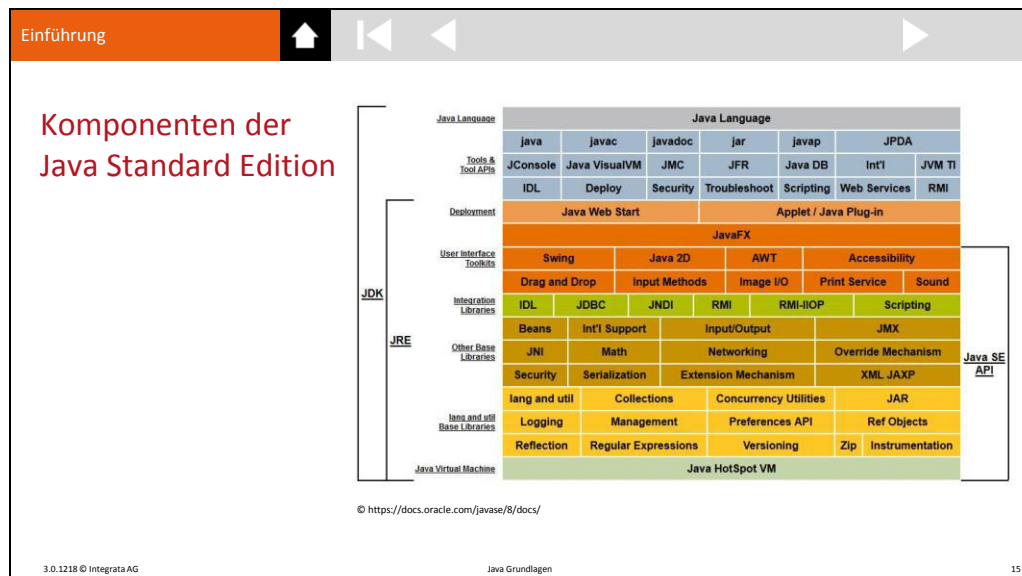


Abb. 1-11: Komponenten der JSE

2

Programmierung

2.1	Die Klasse <code>java.lang.Object</code>	2-3
2.1.1	Die <code>equals()</code> -Methode	2-3
2.1.2	Die <code>toString()</code> -Methode	2-5
2.1.3	Die <code>clone()</code> -Methode	2-5
2.2	Garbage Collector	2-6
2.3	Ausnahmebehandlung	2-7
2.3.1	Die Klasse <code>Throwable</code>	2-7
2.3.2	Die Klasse <code>Error</code>	2-8
2.3.3	Die Klasse <code>Exception</code>	2-8
2.3.4	Ausnahmen mit fehlerbeschreibendem Text	2-10
2.3.5	Die Klasse <code>RuntimeException</code>	2-11
2.4	Typen von Klassen	2-12
2.4.1	Globale Klassen (Top-Level-Klassen)	2-13
2.4.2	Statische Klassen	2-14
2.4.3	Innere Klassen (Memberklassen)	2-14
2.4.4	Lokale Klassen	2-16
2.4.5	Anonyme Klassen	2-17
2.5	Interfaces (Schnittstellen)	2-18
2.5.1	Start	2-19
2.5.2	Statische Methoden	2-20
2.5.3	Default-Methoden	2-21

2.5.4	Konflikte	2-23
2.6	Reflection	2-25
2.6.1	Die Klasse <code>Class</code>	2-25
2.6.2	Introspektion	2-25
2.6.3	Dynamische Erzeugung mit Reflection	2-26
2.7	Das Entwurfsmuster Factory	2-27

2 Programmierung

2.1 Die Klasse `java.lang.Object`

Die Klasse `java.lang.Object` ist die (implizite) Basisklasse aller anderen Klassen und bildet die Spitze der Klassenhierarchie, die in `java.lang.Object` definierten Methoden werden somit von allen Klassen geerbt.

Programmierung

Klasse Object

Java.lang.Object
+ Object()
clone(): Object
+ equals(obj: Object): boolean
finalize()
+ getClass(): Class
+ hashCode(): int
+ notify()
+ notifyAll()
+ toString(): String
+ wait()
+ wait(timeout: long)
+ wait(timeout: long, nanos: int)

- Wichtige Methoden
- String toString()**
 - Zeichenkettendarstellung, Standard: Klassenname + @ + Hashcode, Modifikation durch Überschreiben.
- boolean equals(Object)**
 - Vergleich von Objekten, Standard: Identität, Modifikation durch Überschreiben
 - Zu einer Implementierung von `equals()` sollte immer eine Implementierung von `hashCode()` gehören, denn wenn zwei Objekte gleich sind, müssen auch die Hashwerte gleich sein.
- int hashCode()**
 - Die Methode `hashCode()` berechnet zu einem Objekt einen Hash-Code. Der Hash-Code Wert wird verwendet, um Objekte in einem hash-basierten Container zu finden oder dort abzulegen.
- Object clone()**
 - Kopie eines Objektes, Standard-Implementierung wirft die : `CloneNotSupportedException`
 - Aktivieren mit `implements Cloneable`
 - flache Kopie mit `super.clone()`
 - tiefe Kopie mit Überschreiben.

3.0.1218 © Integrata AG

Java Erweiterungen I

17

Abb. 2-1: Klasse Object

2.1.1 Die `equals()`-Methode

Der Java-Programmierer arbeitet, wie bereits mehrfach erwähnt, mit Objektreferenzen, nicht mit den Objekten selber. Dies muss auch beim Vergleich zweier Objekte berücksichtigt werden, was folgendes Codefragment zeigen soll:

```

Person h1 = new Person("Meier");
Person h2 = new Person("Meier");
if (h1 == h2) {
    System.out.println("Personen sind identisch");
}
else {
    System.out.println("Personen sind nicht gleich");
}
    
```

Die Ausgabe des Programms ist "Personen sind nicht gleich", da die Objektreferenzen natürlich auf verschiedene Speicherbereiche zeigen.

Ein Vergleich auf gleichen Inhalt zweier Objekte ermöglicht die Methode

```
public boolean equals(Object o) .
```

Diese Methode ist in der Klasse `Object` deklariert und liefert in der Standard-Implementierung nichts anderes zurück als

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

also nichts anderes, als den Referenz-Vergleich. Was ist damit gewonnen? So natürlich nichts, aber eigene Klassen können selbstverständlich die `equals()`-Methode auf geeignete Art überschreiben, um z. B. den Vergleich auf gleichen Inhalt abzufragen:

```
public boolean equals(Object obj) {  
    Person lCompare = null;  
    if (obj instanceof Person) {  
        lCompare = (Person)obj;  
    }  
    else {  
        return false;  
    } //if  
  
    if (super.equals(lCompare)) {  
        return true;  
    }  
    else {  
        if (this.vorname.equals(lCompare.vorname) &&  
            (this.nachname.equals(lCompare.nachname))) {  
            return true;  
        }  
        else {  
            return false;  
        } //if  
    } //if  
} //equals(Object)
```

Damit lässt sich das obige Beispiel umformulieren zu:

```
Person h1 = new Person("Meier")  
Person h2 = new Person("Meier")  
if (h1.equals(h2)) {  
    System.out.println("Personen sind identisch");  
}  
else {  
    System.out.println("Personen sind nicht gleich");  
}
```

mit der Ausgabe "Personen sind identisch". Das hier definierte Kriterium für die Objektgleichheit, nämlich die Gleichheit von Vorname und Nachname ist natürlich rein willkürlich und in der realen Anwendung in dieser Form kaum anwendbar. Hier ist der Vergleich auf Identität sicherlich zu bevorzugen.

2.1.2 Die toString() -Methode

Die `toString()`-Methode erzeugt eine Zeichenkettendarstellung des Zustands des aktuellen Objekts (desjenigen Objekts, für welches diese Methode aufgerufen wird). Sie sollte in eigenen Klassen auf geeignete Art überschrieben werden.

```
public String toString(){
    return "Die Person " + this.getName();
}
```

2.1.3 Die clone() -Methode

Die `clone()`-Methode der Klasse `Object` kann eine Kopie eines Objektes erzeugen. Zu beachten ist dabei, dass zur korrekten Verwendung der `clone()`-Methode jede Klasse, deren Instanzen kopiert werden sollen, die Schnittstelle `java.lang.Cloneable` implementieren und den Rückgabewert korrekt casten muss. Ist das `Cloneable`-Interface nicht implementiert, wird die `clone()`-Methode aus `Object` die `java.lang.CloneNotSupportedException` werfen.

Eine Überschattung der `clone()`-Methode ist in der Regel immer notwendig, da die `clone()`-Methode in `java.lang.Object` als `protected` deklariert wurde und folglich ein Aufruf nur in Subklassen und dem Paket `java.lang` heraus möglich ist. Eine Alternative hierzu ist die Definition einer unabhängigen Methode in der Subklasse, die intern `clone()` verwendet.

Beispiel:

```
public Object clone() throws CloneNotSupportedException {
    return super.clone();
}
```

Hier ist zur Verdeutlichung die `clone()`-Methode überschrieben, es wird nur die Standard-Funktionalität aufgerufen.

Die `clone()`-Methode der Klasse `Object` macht nun aber nichts anderes als eine **flache** Kopie, d. h. die Attribute werden als Wert in den Klon kopiert. Wollen wir stattdessen eine **tiefe** Kopie erzeugen, müssen wir die `clone()`-Methode mit Funktionalität hinterlegen, beim Beispiel der Person mit Partner hätten wir:

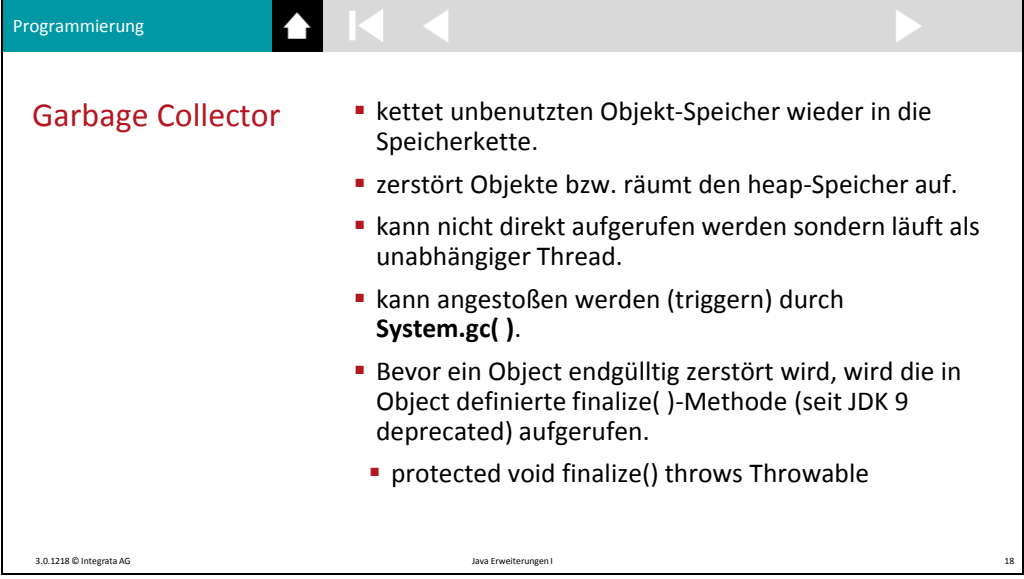
```
public Object clone() throws CloneNotSupportedException {
    Person lKlon = (Person)super.clone();
    lKlon.partner = (Person)lKlon.partner.clone();
    lKlon.partner.nachname = this.partner.nachname;
    lKlon.partner.vorname = this.partner.vorname;
    lKlon.partner.partner = lKlon;
    return lKlon;
} //Ende clone }
```

Hinweis: Diese Methode führt so kodiert zu einer Endlos-Rekursion. Eine konkrete Implementierung würde beispielsweise eine "boolean `clsCloning`"-Flagge benötigen.

Zu bevorzugen ist die Einführung einer eigenen Methode, z. B.:

```
public Person deepCopy() throws CloneNotSupportedException {  
    Person lKlon = (Person)super.clone();  
    lKlon.partner = (Person)lKlon.partner.clone();  
    lKlon.partner.nachname = this.partner.nachname;  
    lKlon.partner.vorname = this.partner.vorname;  
    lKlon.partner.partner = lKlon;  
    return lKlon;  
} //Ende clone
```

2.2 Garbage Collector



The screenshot shows a presentation slide with a teal header bar containing the word 'Programmierung' and navigation icons. The slide title 'Garbage Collector' is in red. It lists five bullet points about the garbage collector's functionality and usage. At the bottom, there is a footer with version information '3.0.1218 © Integrata AG', the course name 'Java Erweiterungen I', and a page number '18'.

Garbage Collector

- kettet unbenutzten Objekt-Speicher wieder in die Speicherkette.
- zerstört Objekte bzw. räumt den heap-Speicher auf.
- kann nicht direkt aufgerufen werden sondern läuft als unabhängiger Thread.
- kann angestoßen werden (triggern) durch **System.gc()**.
- Bevor ein Object endgültig zerstört wird, wird die in Object definierte `finalize()`-Methode (seit JDK 9 deprecated) aufgerufen.
- `protected void finalize() throws Throwable`

3.0.1218 © Integrata AG Java Erweiterungen I 18

Abb. 2-2: Garbage Collector


Der Garbage Collector ist ein JVM-Thread, der im Hintergrund parallel zur Methode `main()` läuft. Der GarbageCollector kettet unbenutzten Objekt/Array-Speicher wieder in die Speicherkette und zerstört Objekte bzw. räumt diese Objekte aus dem heap-Speicher. Er gibt den Speicher wieder frei.

Der Garbage Collector kann nicht direkt aufgerufen werden, sondern läuft als unabhängiger Thread. Bevor ein Object endgültig zerstört wird, wird die in Object definierte `finalize()`-Methode aufgerufen.

2.3 Ausnahmebehandlung

Programmierung

Ausnahmebehandlung



- Tritt während der Programmausführung ein Fehler auf, wird die normale Programmausführung abgebrochen und ein Fehlerobjekt erzeugt (geworfen).
- Die Klasse Throwable fasst alle Arten von Fehlern zusammen.
- Ein Fehlerobjekt kann gefangen und geeignet behandelt werden.
- Trennung von normalem Ablauf und Behandlung von Sonderfällen (wie fehlerhaften Eingaben, ...)

3.0.1218 © Integrata AG
Java Erweiterungen I
19


Abb. 2-3: Ausnahmebehandlung

2.3.1 Die Klasse Throwable

Throwable ist die oberste Klasse in der Hierarchie der Klassen zur Ausnahmebehandlung. Sowohl die Klasse Error als auch die Klasse Exception ist eine Subklasse von Throwable. Nur Subklassen von Throwable können mittels "throw(s)" geworfen werden.

Programmierung

Klassen zur Ausnahmebehandlung



- **Throwable** (Superklasse von Error und Exception)
- **Error**
 - für fatale Fehler, die zur Beendigung des gesamten Programms führen.
- **Exception**
 - für behandelbare Fehler oder Ausnahmen.
- **RuntimeException** (Subklasse von Exception)
 - fasst die bei normaler Programmausführung evt. auftretenden Ausnahmen zusammen. Sie kann jederzeit auftreten und kann, muss aber nicht abgefangen werden.
- unchecked exception
 - Ausnahmen der Klasse Error und RuntimeException müssen nicht im Methodenkopf deklariert werden.
- checked exception
 - Die anderen Ausnahmen, die in einer Methode auftreten können und dort nicht selbst abgefangen werden, müssen explizit im Kopf der Methode deklariert werden.

3.0.1218 © Integrata AG
Java Erweiterungen I
20

Abb. 2-4: Klassen zur Ausnahmebehandlung

2.3.2 Die Klasse Error

Error nutzt man dann, wenn es keinen Sinn macht, die Anwendung weiter fortzuführen. Error ist eine Klasse für fatale Fehler, die zur Beendigung des gesamten Programms führen. Ausnahmen der Klasse Error müssen nicht im Methodenkopf deklariert werden.

2.3.3 Die Klasse Exception

Mit Hilfe der Klasse Exception wird in Java das Auftreten und Behandeln von Fehlern wesentlich vereinfacht. Jede Methode hat die Möglichkeit, beim Auftreten eines Fehlers ein Objekt vom Typ `java.lang.Exception` zu erzeugen und der aufrufenden Methode zu übermitteln, ohne die `return`-Anweisung zu verwenden: Die Ausnahme „wird geworfen“.

Erzeugen der Ausnahme:

```
Exception e = new Exception( );
```

Werfen der eben erzeugten Ausnahme:

```
throw e;
```

Oder in der üblichen verkürzten Form als anonyme Exception:

```
throw new Exception();
```

Es ist zu beachten, dass dieses geworfene Objekt vom Typ `Exception` bisher vollkommen unspezifisch ist und nur signalisieren kann „Es ist eine Ausnahme aufgetreten“.

`Exception`-Objekte können jedoch bei der Instanziierung durch die Übergabe einer Zeichenkette näher spezifiziert werden.

Eine Methode, hier z. B. eine namens „methode()“, die eine Ausnahme werfen kann, muss bei der Deklaration durch `throws Exception` ergänzt werden.

```
public void methode( ) throws Exception {  
    // Anweisungen ...  
    throw new Exception( );  
}
```

Der große Vorteil des Werfens von Ausnahmen ist nun, dass die aufrufende Methode notwendig auf die Ausnahme reagieren muss, ein einfacher Aufruf via

```
referenz.methode( );
```

führt nun zu einem Compilerfehler.

Um die Methode aufrufen zu können, muss der Aufruf in einem try-Block erfolgen. Ein try-Block wird stets von mindestens einem catch-Block begleitet, der in einer Fehlersituation die geworfene Ausnahme auffängt und die Fehlerbehandlungsroutine enthält. Nach dem catch-Zweig kann noch ein finally-Block stehen, der in jedem Falle durchlaufen wird.

Allgemeiner Aufbau:

```
try {
    referenz.methode();
    // weitere Anweisungen
}
catch(Exception e) {
    System.out.println("allgemeiner Fehler");
}
finally { // wird in jedem Fall durchlaufen
    //Aufräumarbeiten
}
```

Besonders sei anhand dieses Beispiels auf die Vorteile der Verwendung von Exception hingewiesen:

1. Saubere Trennung zwischen Programmlogik und Fehlerbehandlungsteil im Quellcode.
2. Es werden keine willkürlichen Rückgabewerte oder globale Fehlervariablen benötigt.
3. Die aufrufende Ebene muss auf das mögliche Auftreten eines Fehlers reagieren können.

Für die flexiblere Fehlerbehandlung seien hier noch darauf hingewiesen, dass beim Erzeugen einer Exception ein beschreibender Text mitgegeben werden kann.

Schlüsselwörter

- **try**
 - Definiert einen Block, innerhalb dessen Ausnahmen (Exceptions) auftreten können (geworfen werden können).
- **catch**
 - Definiert einen Block, der die Fehlerbehandlung für die durch den im catch-Befehl angegebenen Ausnahmetyp durchführt.
- **finally**
 - Definiert einen Block, der stets ausgeführt wird, egal ob ein Fehler auftrat oder nicht. Auch wenn
 - innerhalb des catch-Zweiges eine weitere Exception geworfen wird.
 - im catch-Zweig ein return steht.
 - kein catch-Zweig durchlaufen wird.
- **throw**
 - Erzeugt (wirft) eine Ausnahme. Hierfür muss dem Befehl throw ein Objekt übergeben werden, das eine Unterklasse von Throwable ist (dies sind die Klassen Exception und Error).
- **throws Exception**
 - Die Methode muss mit throws eine Liste aller Ausnahmen definieren, die geworfen werden können.

3.0.1218 © Integrata AG Java Erweiterungen I 21

Abb. 2-5: Schlüsselwörter

2.3.4 Ausnahmen mit fehlerbeschreibendem Text

Die bisherige Verwendung der Exception ist recht unkomfortabel, da außer der Aussagen "Fehler aufgetreten" keine weiteren Informationen in die aufrufende Ebene gegeben wurden. Analog der Klasse String kann auch eine Exception mit einem Text erzeugt werden, der dann über die Methode `getMessage()` ausgelesen werden kann.

Im diesem Beispiel wird in der Klasse Rechnen durch 0 geteilt:

```
public class Rechnen {
    public static void main( String[ ] args ) {
        int erg = 1 / 0;
    }
}
```

Das Ergebnis:

Exception in thread "main" [java.lang.ArithmeticException: / by zero](#)
 at Rechnen.main([Rechnen.java:3](#))

Wenn das Teilen durch 0 nicht abgefangen wird, erscheint eine `java.lang.ArithmeticException` und das Programm wird beendet. Eine `ArithmeticException` ist eine `RuntimeException` und muss nicht abgefangen werden.

Sie kann aber abgefangen werden:

```
public class Rechnen {
    public static void main(String[] args) {
        try {
            int erg = 1 / 0;
        }
        catch( ArithmeticException ex ) {
            System.out.println( ex.getMessage( ) );
        }
    }
}
```

Das Ergebnis:

/ by zero

2.3.5 Die Klasse RuntimeException

Eine Subklasse von Exception ist die Klasse RuntimeException. Sie fasst die bei normaler Programmausführung evtl. auftretenden Ausnahmen zusammen. Eine RuntimeException kann jederzeit auftreten, muss aber nicht abgefangen werden (unchecked Exception).

2.4 Typen von Klassen

Bis jetzt wurde immer eine neue Klasse in einer neuen Datei angelegt und anschließend kompiliert. Aber Java bietet auch die Möglichkeit, innere Klassen, d.h. Klassen in Klassen zu verwenden. Die Vorteile solcher Klassen sind:

- Bessere Kapselung
- Gruppieren von Klassen, die nur an einem Ort benötigt werden
- Erhöhung der Wartbarkeit des Codes
- Verbesserte Lesbarkeit des Codes

The screenshot shows a presentation slide with a teal header bar containing the word 'Programmierung' and navigation icons. The slide title is 'Klassentypen' in red. The content lists five types of classes with their characteristics and bytecodes:

- Globale Klasse
 - Organisation in Paketen
 - Bytecode: Klasse.class
- Innere Klasse (Memberklasse)
 - Klasse in Klasse
 - Bytecode: KlasseAussen\$KlasseInnen.class
- Statische innere Klasse
 - Static Klasse in Klasse kann ohne Instanz der äußeren Klasse Instanziiert werden
 - Bytecode: KlasseAussen\$KlasseStaticInnen
- Lokale Klasse
 - Klasse in Methode
 - Bytecode: KlasseAussen\$1KlasseLokal.class
- Anonyme Klasse
 - Namenlose Klasse in Methode
 - Bytecode: KlasseAussen\$1.class

At the bottom, there is a footer with '3.0.1218 © Integrata AG' on the left, 'Java Erweiterungen I' in the center, and '22' on the right.

Abb. 2-6: Typen von Klassen

2.4.1 Globale Klassen (Top-Level-Klassen)

Globale Klassen sind in Paketen organisiert. Der Bytecode einer Globalen Klasse wird in einer eigenen Bytecode-Datei abgelegt, die im unteren Beispiel die Bezeichnung `Klasse.class` erhält.

Programmierung

Beispiel

Globale Klasse

Ausgabe:

Klasse erzeugt

```

public class Klasse {
    // Konstruktor Klasse
    public Klasse ( ) {
        System.out.println( "Klasse erzeugt" );
    }
} // class KlasseAussen

```

Instanzerzeugung

```

Klasse k = new Klasse ( );

```

3.0.1218 © Integrata AG

Java Erweiterungen I

23

Abb. 2-7: Globale Klasse

2.4.2 Statische Klassen

Statische Klassen werden direkt über die äußere Klasse angesprochen, sofern die Kapselung dies zulässt. Das Verhalten entspricht den gewöhnlichen Klassen. Es können Instanzen erzeugt werden. Attribute, Methoden und statische Inhalte sind ohne Probleme im Zugriff. Ein zusätzlicher Zugriff auf die statischen Attribute und Methoden der äußeren Klasse ist möglich. Der Zugriff auf Instanzattribute oder Instanzmethoden der äußeren Klasse ist hingegen tabu.

Statische innere Klassen sind im Prinzip normale Klassen mit zusätzlichen Zugriffsrechten.

Beispiel
Statische Innere Klasse

Ausgabe:

KlasseStaticInnen erzeugt

```
public class KlasseAussen {
    // Konstruktor Äußere Klasse
    public KlasseAussen( ) {
        System.out.println( "KlasseAussen erzeugt" );
    }

    public static class KlasseStaticInnen {
        // Konstruktor Innere static Klasse
        public KlasseStaticInnen( ) {
            System.out.println( "KlasseStaticInnen erzeugt" );
        }
    } // class KlasseStaticInnen
} // class KlasseAussen
```

■ **Instanzerzeugung**

```
KlasseAussen.KlasseStaticInnen ki = new KlasseAussen.KlasseStaticInnen( );
```

3.0.1218 © Integrata AG Java Erweiterungen I 25

Abb. 2-8: Statische Klasse

2.4.3 Innere Klassen (Memberklassen)

Innere Klassen sind nicht-statische Klassen in Klassen. Um von außen ein Objekt der inneren Klasse zu erzeugen, muss man erst ein Objekt der äußeren Klasse generieren. Nichtstatische innere Klassen sind immer an eine konkrete Instanz der äußeren Klasse gebunden.

Innere Klassen können auf Instanzattribute, Instanzmethoden und auch auf statische Elemente der äußeren Klasse zugreifen. Sie dürfen aber keine eigenen Klassenattribute oder Klassenmethoden definieren.

Innere Klassen können im Rahmen der Klassenmodellierung dazu eingesetzt werden, um starke Abhängigkeiten zwischen Klassen zu definieren.

Innere Klassen können gekapselt werden, die Schlüsselwörter `public`, `protected` und `private` sowie die Paketsichtbarkeit bleiben in ihrer Bedeutung ohne Beschränkung verwendbar.

Der Bytecode einer Inneren Klasse wird aber weiterhin in einer eigenen Bytecode-Datei abgelegt, die in diesem Beispiel die Bezeichnung `KlasseAussen$KlasseInnen.class` erhält.

Programmierung

Beispiel

Innere Klasse

Ausgabe:

KlasseAussen erzeugt

KlasseInnen erzeugt

```

public class KlasseAussen {
    // Konstruktor Äußere Klasse
    public KlasseAussen( ) {
        System.out.println( "KlasseAussen erzeugt" );
    }

    public class KlasseInnen {
        // Konstruktor Innere Klasse
        public KlasseInnen( ) {
            System.out.println( "KlasseInnen erzeugt" );
        }
    } // class KlasseInnen
} // class KlasseAussen
        
```

■ Instanzerzeugung

```

KlasseAussen ka = new KlasseAussen( );
KlasseAussen.KlasseInnen ki = ka.new KlasseInnen( );
        
```

3.0.1218 © Integrata AG

Java Erweiterungen I

24

Abb. 2-9: Innere Klasse

Normalerweise erzeugt man keine Objekte einer inneren Klasse von außen. Stattdessen bietet häufig die äußere Klasse eine Funktion, die ein Objekt der inneren Klasse zur Verfügung stellt. Innere Klassen sind häufig als *private* gekennzeichnet.

Innere Klassen haben einen direkten Bezug zu der Instanz der äußeren Klasse. Um auf (namensgleiche) Elemente der äußeren Klasse zugreifen zu können, wird nun die Verwendung des `this`-Schlüsselwortes mit `KlasseAussen.this` erweitert.

2.4.4 Lokale Klassen

Die Lokalen Klassen werden in einer Methode einer Klasse definiert. Da es sich um lokale Elemente handelt, dürfen natürlich keine Anweisungen zur Sichtbarkeit verwendet werden. Lokale Klassen können zusätzlich auf alle Parameter und Variablen der Methode zugreifen.

Der Bytecode einer Lokalen Klasse wird aber weiterhin in einer eigenen Bytecode-Datei abgelegt, die im unteren Beispiel die Bezeichnung `KlasseAussen$1KlasseLokal.class` erhält.

The screenshot shows a presentation slide with a teal header bar labeled 'Programmierung'. The slide content is divided into three main sections: a title, a code block, and an output section.

Beispiel Lokale Klasse

```
public class KlasseAussen {  
    // Konstruktor Äußere Klasse  
    public KlasseAussen( ) {  
        System.out.println( "KlasseAussen erzeugt" );  
    }  
  
    public void methodeAussen( ) {  
        class KlasseLokal{  
            // Konstruktor Lokale Klasse  
            KlasseLokal( ){  
                System.out.println( "KlasseLokal erzeugt" );  
            }  
        } // KlasseLokal  
        new KlasseLokal( );  
    } // MethodeAussen  
}  
// class KlasseAussen
```

Ausgabe:

KlasseAussen erzeugt
KlasseLokal erzeugt

■ **Instanzerzeugung**

```
KlasseAussen ka = new KlasseAussen( );  
ka.methodeAussen( );
```

At the bottom of the slide, there is a footer with the text '3.0.1218 © Integrata AG' on the left, 'Java Erweiterungen I' in the center, and '26' on the right.

Abb. 2-10: Lokale Klasse

2.4.5 Anonyme Klassen

Innere Klassen können in einer Klassendefinition, aber auch in einer Methodendefinition stehen. Letzteres wird häufig im Zusammenhang mit so genannten anonymen Klassen gebraucht. Zur Einführung einer anonymen Klasse stellen wir uns folgende Situation vor: Wir möchten in einer bestimmten Programmsituation eine Instanz einer Klasse verwenden, die exakt einmal für diese spezielle Aufgabe definiert sein wird.

Der Bytecode einer anonymen Klasse wird auch in einer eigenen Bytecode-Datei abgelegt, die im unteren Beispiel die Bezeichnung `KlasseAussen$1.class` erhält. Falls eine Klasse mehrere anonyme Klassen enthält, werden diese durchnummeriert.

Programmierung

↑

◀

◀

▶

Beispiel

Anonyme Klasse

Ausgabe:

KlasseAussen erzeugt

Implementierung Anonym

```

public class KlasseAussen {
    // Konstruktor Äußere Klasse
    public KlasseAussen( ) {
        System.out.println( "KlasseAussen erzeugt" );

        // Anonyme Klasse
        doSomething( new Object( ) {
            public String toString( ) {
                return "Implementierung Anonym";
            }
        } );
    }

    private void doSomething( Object pObject ) {
        System.out.println( pObject.toString( ) );
    } // doSomething
} // class KlasseAussen

▪ Instanzerzeugung

KlasseAussen ka = new KlasseAussen( );

```

3.0.1218 © Integrata AG

Java Erweiterungen I

27

Abb. 2-11: Anonyme Klasse

Grafische Entwicklungsumgebungen erzeugen häufig solche anonymen Klassen beim Erstellen von Event-Listnern. Hierzu später mehr. Ansonsten sollte die Verwendung von anonymen Klassen wohl überlegt werden: Diese können weder wieder verwendet noch sonderlich einfach gewartet werden.


2.5 Interfaces (Schnittstellen)

In Interfaces können seit Java 8 nicht nur abstrakte Methoden und statische Konstanten definiert werden, sondern zusätzlich auch statische Methoden und sog. `default`-Methoden. Java unterstützt damit das, was man als "mixin inheritance" bezeichnet.

Programmierung

⬅
⬅
➡

Ein Interface (Schnittstelle) kann enthalten



- Abstrakte Methoden
- Implementierte (default) Methoden seit Java 8
- Statische Konstanten
- Statische Methoden
- Private Methoden seit Java 9
- Private statische Methoden seit Java 9

```

public interface Foo {
    int x = 42; //implizit public static final
    final int y = 43; //implizit public static
    public static final int z = 44;

    void f(); //implizit public abstract
    public void g(); //implizit abstract
    public abstract void h();

    class C { }; //implizit public static
    public static class D { }
}
                    
```

3.0.1218 © Integrata AG
Java Erweiterungen I
28

Abb. 2-12: Interface

Der Vorteil ist klar. Einem Interface eine weitere abstrakte Methode hinzuzufügen, würde bedeuten, dass alle dieses Interface nutzende Klienten ihrerseits erweitert werden müssten: sie müssten die neue abstrakte Methode implementieren. Ein Interface kann aber problemlos um bereits implementierte Funktionalität erweitert werden, ohne dass die bisherigen Klienten des Interfaces sich darum kümmern müssen. Und zukünftige Klienten können von diesen Erweiterungen profitieren.

Die Interfaces von APIs können also unter Beibehaltung der Rückwärtskompatibilität erweitert werden – ohne den Vertrag mit den diese Interfaces nutzenden Klienten zu brechen. Die in Interfaces implementierten Methoden können dabei natürlich die bereits vorhandenen abstrakten Methoden nutzen. Und das von `default`-Methoden beschriebene Standardverhalten schließlich kann von konkreten Klassen jederzeit überschrieben werden.

Der Nachteil ist aber ebenso klar: Bislang dienten Interfaces nur der Spezifikation (von der Möglichkeit der Definition statischer Konstanten einmal abgesehen). Dieser "saubere" Interface-Begriff wird nun verwässert. Die Schönheit von Interfaces, die auf ihrer vollständigen Abstraktheit beruhte, geht verloren.

Resultat: Man sollte die neuen Möglichkeiten nur mit Bedacht nutzen.

2.5.1 Start

Bislang konnten in einem Interface nur öffentliche abstrakte Methoden, öffentliche statische Konstanten und öffentliche statische innere Klassen definiert werden:

```
public interface Foo {

    int x = 42;
    final int y = 43;
    public static final int z = 44;

    void f();
    public void g();
    public abstract void h();

    class C { }
    public static class D { }
}
```

Die Variable `i` sieht zwar aus wie eine nicht öffentliche Instanzvariable, ist aber `static`, `public` und `final` (implizit). Dasselbe gilt auch für `y`. Die Definition von `z` ist die ausführlichste Definition – `x` und `y` sind implizit aber genauso definiert.

Auch bei der `f`-Definition fügt der Compiler die Modifizierer `public` und `abstract` automatisch hinzu; bei `g` wird `abstract` hinzugefügt. `f` und `g` haben also (implizit) exakt dieselben Modifizierer, die bei `h` explizit notiert sind.

Auch die hier definierte Klasse `C` ist `public` und `static` – genauso wie `D`.

Hier eine mögliche Implementierung des obigen Interfaces:

```
public class FooImpl implements Foo {
    public void f() {
        System.out.println("f()");
    }
    public void g() {
        System.out.println("g()");
    }
    public void h() {
        System.out.println("h()");
    }
}
```

Ein `FooImpl`-Objekt kann nun über eine `Foo`-Referenz genutzt werden:

```
static void demo() {  
    Foo foo = new FooImpl();  
    System.out.println(Foo.x);  
    System.out.println(Foo.y);  
    System.out.println(Foo.z);  
    foo.f();  
    foo.g();  
    foo.h();  
}
```

Soweit zum bisherigen Stand der Dinge.

2.5.2 Statische Methoden

Ein Interface kann auch statische Methoden enthalten:

```
public interface Foo {  
    static final int x = 42;  
    static void printX() {  
        System.out.println(x);  
    }  
    void f();  
}
```

Statische Methoden eines Interfaces können natürlich auf statische Attribute dieses Interfaces zugreifen (oder eine andere statische Methode des Interfaces aufrufen).

Hier eine Implementierung des `Foo`-Interfaces:

```
public class FooImpl implements Foo {  
    public void f() {  
        System.out.println("f()");  
    }  
}
```

Um die `f`-Methode aufzurufen, benötigt man natürlich eine Instanz einer konkreten, das Interfaces implementierenden Klasse; `x` und `printX` können aber über den Namen des Interfaces angesprochen resp. aufgerufen werden:

```
static void demo() {  
    Foo foo = new FooImpl();  
    System.out.println(Foo.x);  
    Foo.printX();  
    foo.f();  
}
```

2.5.3 Default-Methoden

Ein Interface kann auch Instanz-Methoden implementieren – vorausgesetzt, sie sind als `default` definiert und nicht `final` (default und final schließen sich natürlich aus gutem Grunde aus...):

```
public interface Foo {
    void f();
    default void g() {
        System.out.print("g");
        System.out.println("g()");
    }
    // default final void h() { // illegal
    //     System.out.println("h()");
    // }
}
```

Semantisch gesehen sind default-Implementierungen ausdrücklich zum Überschreiben vorgesehen (eine konkrete Klasse könnte z.B. eine wesentliche performantere Implementierung anbieten – weil in ihr die konkrete Struktur der Daten bekannt ist, auf denen diese Methode operiert). Und dieser Semantik sollte man sich auch bewusst sein, wenn man eigene Interfaces mit solchen default-Methoden ausstattet.

Hier eine konkrete Implementierungsklasse:

```
public class FooImpl implements Foo {
    public void f() {
        System.out.println("f()");
    }
    @Override
    public void g() {
        System.out.println("gg()");
    }
}
```

Die `g`-Methode von `FooImpl` ist wahrscheinlich performanter als die `g`-Methode des Interfaces...

Eine Anwendung:

```
static void demo() {
    Foo foo = new FooImpl();
    foo.f();
    foo.g();
}
```

Hier wird natürlich die überschriebene `g`-Methode aufgerufen.

Ein weiteres, diesmal tatsächlich einigermaßen "sinnvolles" Interface – das Interface `YAC` ("Yet another Comparator"):

```
public interface YAC<T> {

    public abstract boolean eq(T v0, T v1);

    public abstract boolean gt(T v0, T v1);

    public default boolean ge(T v0, T v1) {
        return this.gt(v0, v1) || this.eq(v0, v1);
    }

    public default boolean lt(T v0, T v1) {
        return ! this.ge(v0, v1);
    }

    public default boolean le(T v0, T v1) {
        return this.eq(v0, v1) || this.lt(v0, v1);
    }
}
```

Eine von `YAC` abgeleitete instantiierbare Klasse muss nur zwei Methoden implementieren: `eq` und `gt`. Die drei weiteren Methoden des Interfaces werden alle auf diese beiden Methoden zurückgeführt.

Ein konkreter `YAC` zum Vergleich von `Integer`-Objekten:

```
YAC<Integer> yac = new YAC<Integer>() {
    public boolean eq(Integer v0, Integer v1) {
        return v0.equals(v1);
    }
    public boolean gt(Integer v0, Integer v1) {
        return v0.compareTo(v1) > 0;
    }
};
```

Alle folgenden Aufrufe der `yac`-Methoden liefern `true`:

```
static void demoYAC() {
    System.out.println(yac.eq(1, 1));
    System.out.println(yac.gt(2, 1));
    System.out.println(yac.ge(1, 1));
    System.out.println(yac.ge(2, 1));
    System.out.println(yac.lt(1, 2));
    System.out.println(yac.le(1, 2));
    System.out.println(yac.le(1, 1));
}
```

2.5.4 Konflikte

Was passiert, wenn zwei Interfaces Methode definieren, welche dieselbe Signatur haben – und eine Klasse dennoch beide Interfaces implementieren möchte? Solche Probleme gab's auch bereits im "alten" Java:

```
public interface Foo {
    public abstract void f();
}
```

```
public interface Bar {
    public abstract void f();
}
```

`Foo` und `Bar` spezifizieren beide eine parameterlose `f`-Methode vom Typ `void`. Eine Klasse, die beide Interfaces implementiert, kann natürlich nur eine einzige `f`-Methode enthalten:

```
public class FooBar implements Foo, Bar {
    public void f() { ... }
}
```

Die Lösung war nie so richtig zufriedenstellend (in C# z.B. kann es dagegen für jede der beiden `f`-Methoden eine eigene Implementierung geben). Dieses Manko des "alten" Java konnte natürlich nicht beseitigt werden. Man könnte das Problem mit dem Hinweis abtun, solche Schwierigkeiten seien rein akademischer Natur und würden in der Praxis nicht auftreten (wer definiert schon eine Methode namens `f`???)

Bei den neuen `default`-Methoden hat man nun aber solche möglichen Konflikte sauber gelöst:

Sei sowohl in `Foo` und `Bar` die `default`-Methode `f` definiert:

```
public interface Foo {
    public default void f() {
        System.out.println("Foo.f");
    }
}
```

```
public interface Bar {
    public default void f() {
        System.out.println("Bar.f");
    }
}
```

Man möchte nun eine Klasse `FooBar` bauen, die beide Interfaces implementiert.

Folgende "Lösung" weist der Compiler zurück:

```
public class FooBar implements Foo, Bar {
}
```

Man kann aber in der Klasse eine eigene `f`-Methode implementieren:

```
public class FooBar1 implements Foo, Bar {  
    public void f() {  
        System.out.println("FooBar1.f()");  
    }  
}
```

Egal, ob ein `FooBar1`-Objekt nun über eine `Foo`- oder über ein `Bar`-Referenz angesprochen wird – es wird immer die eine `f`-Methode von `FooBar1` aufgerufen werden. Die `default`-Methoden der Interfaces werden also überhaupt nicht aufgerufen.

In einer Methode der Implementierungsklasse können dann aber wieder die `default`-Implementierungen der Interfaces aufgerufen werden – über die Notation `<Interface>.super.<Methode>`:

```
public class FooBar2 implements Foo, Bar {  
    public void f() {  
        Foo.super.f();  
        Bar.super.f();  
    }  
}
```

Eine Anwendung:

```
static void demo() {  
    Foo foo = new FooBar1();  
    Bar bar = new FooBar2();  
    foo.f();  
    bar.f();  
}
```

Die Ausgaben:

`FooBar1.f()`

`Foo.f`

`Bar.f`

2.6 Reflection

2.6.1 Die Klasse `Class`

Zentraler Einstiegspunkt ist die Klasse `java.lang.Class`. Diese Klasse war bereits Bestandteil des JDK 1.0 und bietet eine einfache Möglichkeit zum dynamischen Laden einer Klasse, nämlich die statische Methode

```
static Class.forName( String className )
```

Eine Instanz dieser Klasse ist zugreifbar sowohl über den Klassennamen (im oberen Beispiel "`String.class`") als auch für jede Instanz über die Methode `getClass()` aus `java.lang.Object`. Steht ein Klassenobjekt zur Verfügung, kann dieses nun weiter untersucht oder besser: "introspeziert" werden.

2.6.2 Introspektion

Das für die `Reflection` relevante Ergebnis dieser Introspektion ist:

- Dynamisches Laden einer Klasse:

```
static Class.forName(String)
```

- Eine Liste aller Konstruktoren:

```
java.lang.reflect.Constructor[]
getConstructors()
getDeclaredConstructors()
```

- Eine Liste aller Attribute:

```
java.lang.reflect.Field[]
getFields()
getDeclaredFields( )
```

- Eine Liste aller Methoden:

```
java.lang.reflect.Method[]
getMethods()
getDeclaredMethods( )
```

- Aufruf des parameterlosen Konstruktors:

```
Object newInstance( )
```

2.6.3 Dynamische Erzeugung mit Reflection

Wir wollen eine Anwendung möglichst einfach konfigurieren können. Auf diese Art und Weise werden wir in die Lage versetzt, eine Anwendung für viele verschiedene Problemstellungen einsetzen zu können, ohne die Klassen neu kompilieren zu müssen. Das führt nun zu einer etwas absurden Folgerung: Wenn wir nun nicht das Schlüsselwort `new` verwenden sollen, wo denn dann? Die Antwort lautet: Wenn Flexibilität zur Laufzeit benötigt wird: Nirgends!

Wollen wir aus den verschiedensten Gründen die Erzeugung von Objekten flexibel und konfigurierbar machen, kann der Java-Programmierer dies mit den Typen des dem Paket `java.lang.reflect` realisieren.

Ein Objekt vom Typ `Klasse` kann dynamisch wie folgt erzeugt werden:

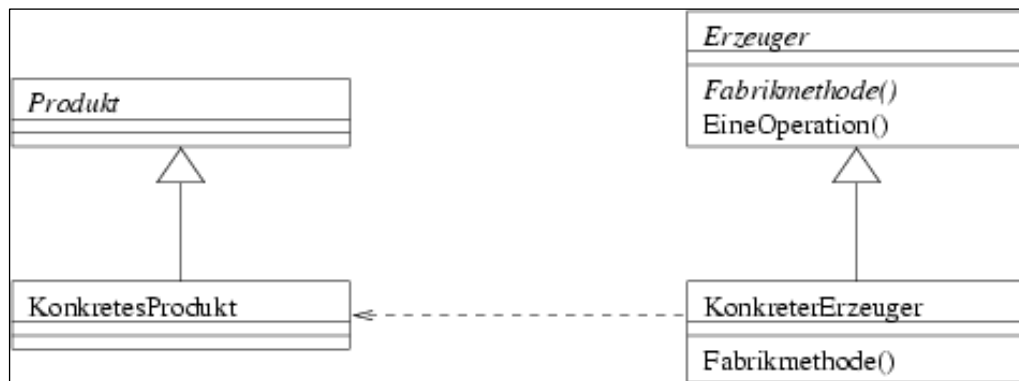
Variante 1: Klasse definiert einen parameterlosen Konstruktor

```
String name = "package.Klasse";  
Class co = Class.forName(name);    //Klasse laden  
Object obj = co.newInstance();     //Objekt erzeugen
```

Variante 2: Klasse definiert einen Konstruktor mit dem Parametertyp `String`

```
String name = "package.Klasse";  
Class co = Class.forName(name);    //Klasse laden  
Class[] types = { String.class }; //Parametertypen  
Object[] values = { "Irgendetwas" };  
           //Parameterwerte  
//Den Konstruktor holen  
Constructor con = co.getConstructor(types);  
Object obj = con.newInstance(values); //Obj erzeugen
```

2.7 Das Entwurfsmuster Factory



- Erzeuger
 - definiert die Fabrikmethode, welche ein Produkt erzeugt
 - eventuelle Default-Implementation der Fabrikmethode, welche ein konkretes Produkt erzeugt
- KonkreterErzeuger
 - überschreibt die Fabrikmethode zur Erzeugung konkreter Produkte
- Produkt
 - definiert die Schnittstelle für das Produkt, welches die Fabrikmethode erzeugt
- KonkretesProdukt
 - definiert ein konkretes Produkt durch Implementation der Schnittstelle
 - wird durch den korrespondierenden konkreten Erzeuger erzeugt

3

Das Collections Framework und Enums

3.1	Bestandteile des Collection-Frameworks	3-3
3.2	Einfache Implementierungen	3-6
3.2.1	List	3-6
3.2.2	Queue	3-7
3.2.3	Set	3-8
3.2.4	Map.....	3-9
3.3	Generische Collections.....	3-10
3.3.1	Notwendigkeit generischer Collections	3-10
3.3.2	Dokumentation.....	3-12
3.3.3	Technischer Hintergrund.....	3-13
3.3.4	Collections und einfache Datentypen.....	3-15
3.4	Iteration über Collections	3-16
3.4.1	Iteration mit Iterator.....	3-16
3.4.2	Iteration mit forEach.....	3-16
3.4.3	Iteration mit forEach-Methode.....	3-17
3.4.4	Iteration mit forEach-Methode als Lambda-Ausdruck	3-17
3.5	Die Klasse Collections.....	3-18
3.5.1	Sortieren von Collections	3-18
3.5.2	Unveränderbare Listen	3-20
3.5.3	Leere Listen	3-20
3.6	Enums	3-21

3.6.1	Ein Problem	3-21
3.6.2	Eine Lösung mit dem "alten" Java	3-22
3.6.3	Enums: enum.....	3-25

3 Das Collections Framework und Enums

3.1 Bestandteile des Collection-Frameworks

Das Collection-Framework aus vielen Klassen und Schnittstellen, die das Arbeiten mit unterschiedlichen Typen von Kollektionen (Listen, Mengen) definieren.

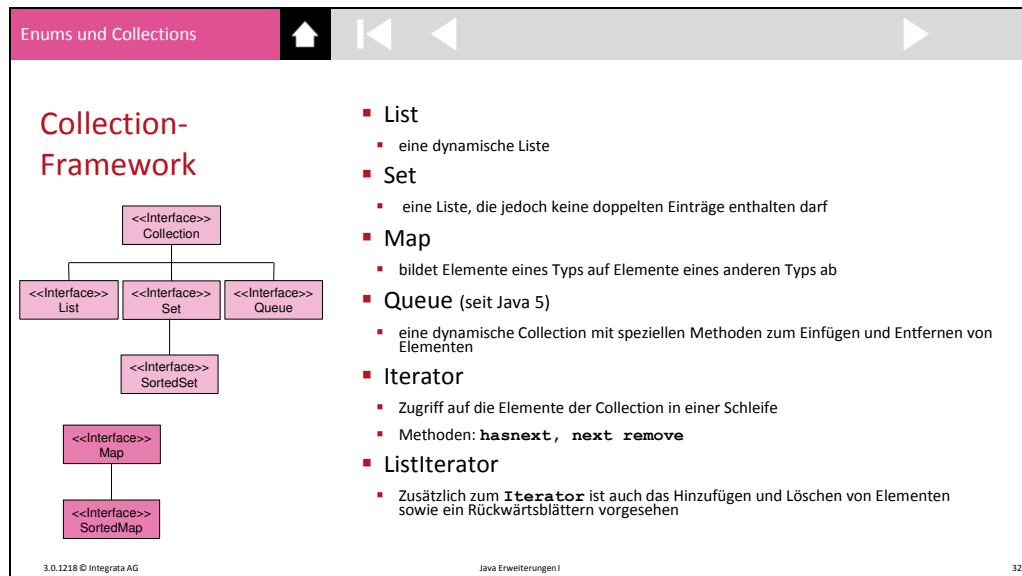


Abb. 3-1: Collection-Framework

Zur Übersichtlichkeit beschränken wir uns hier zunächst auf eine grundsätzliche Behandlung der vier Grundtypen:

- Eine `List` ist eine dynamische Liste, die eine beliebige Anzahl beliebiger Elemente aufnehmen kann. Der Zugriff erfolgt entweder wahlfrei oder sequentiell.

Eine Auswahl wichtiger Methoden der `List`-Schnittstelle:

```
add(Object element)
add(int index, element)
clear()
remove(Object element)
remove(int index)
size()
```

- Ein `Set` ist eine Menge, die keine doppelten Einträge enthalten darf und auch keine Reihenfolge kennt.

Eine Auswahl wichtiger Methoden der `Set`-Schnittstelle:

```
add(Object element)
clear()
remove(Object element)
size()
```

- Eine `Queue` ist spezielle `Collection`, die zusätzliche Methoden zum Hinzufügen und entfernen von Elementen aufweist:

```
offer(Object element)
remove()
element()
poll()
peek()
```

- Eine `Map` bildet Elemente eines Typs (Schlüssel, `key`) auf Elemente eines anderen Typs (Wert, `value`).

Eine Auswahl wichtiger Methoden der `Map`-Schnittstelle:

```
get(Object key);
put(Object key, Object value)
clear()
remove(Object key)
size()
```

Die Schnittstellen `Set` und `List` sind von der allgemeinen Schnittstelle `Collection` abgeleitet, die die für beide Fälle gemeinsamen Methoden enthält. Das `Collection`-Framework stellt eine ganze Reihe von Klassen zur Verfügung, die diese Schnittstellen bereits implementieren, wie z.B. `ArrayList` und `LinkedList` etc., siehe den folgenden Abschnitt. Weiterhin enthalten ist die Klasse `Collections`, die eine ganze Reihe statischer Hilfsmethoden enthält, insbesondere zum Sortieren von Listen bzw. Suchen von Elementen. Die Klasse `Arrays` definiert diese Methoden für Arrays.

Eine Aufzählung der Elemente innerhalb einer `Collection` oder `Map` erfolgt über einen `Iterator`. Dieser bietet die Methoden

```
boolean hasNext()
Object next()
void remove()
```

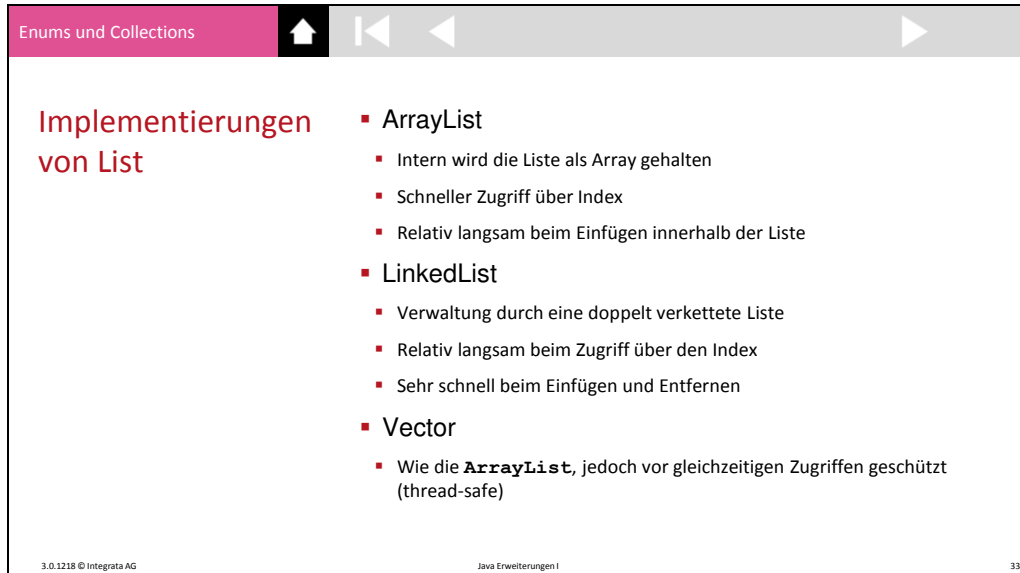
Eine Erweiterung des `Iterator` ist der `ListIterator`, der zusätzlich ermöglicht, Elemente an einer bestimmten Stelle hinzuzufügen, zu entfernen und zurück zu blättern.

Hier fassen wir die Hierarchie der Interfaces und Klassen noch kurz zusammen:

- Interface **Collection**
 - Interface **List**
 1. Klassen ArrayList, LinkedList, Vector, ...
 - Interface **Set**
 2. Klassen HashSet, TreeSet, ...
- Interface **Map**
 - Klassen HashMap, Hashtable, TreeMap, ...
- Interface **Iterator**
- Interface Comparator
- Interface Comparable
- Klasse Collections
- Klasse Arrays

3.2 Einfache Implementierungen

3.2.1 List



The screenshot shows a presentation slide with a pink header bar containing the text 'Enums und Collections' and navigation icons. The main content area has the title 'Implementierungen von List' in red. Below the title, there is a bulleted list of three implementations: ArrayList, LinkedList, and Vector, each with its own set of characteristics.

- **ArrayList**
 - Intern wird die Liste als Array gehalten
 - Schneller Zugriff über Index
 - Relativ langsam beim Einfügen innerhalb der Liste
- **LinkedList**
 - Verwaltung durch eine doppelt verkettete Liste
 - Relativ langsam beim Zugriff über den Index
 - Sehr schnell beim Einfügen und Entfernen
- **Vector**
 - Wie die **ArrayList**, jedoch vor gleichzeitigen Zugriffen geschützt (thread-safe)

At the bottom of the slide, there is a small footer with the text '3.0.1218 © Integrata AG' on the left, 'Java Erweiterungen I' in the center, and '33' on the right.

Abb. 3-2: List

Die Implementierungen der `List`-Schnittstelle sind:

`ArrayList`

`LinkedList`

`Vector`

und mit Einschränkung:

`Stack`

Jede dieser Implementierungen hat ihre ganz speziellen Einsatzbereiche. Ein `Vector` ist darauf ausgelegt, von mehreren Unterprozessen, so genannten `Threads`, gleichzeitig manipuliert zu werden, ohne dass es zu Inkonsistenzen kommt. Kann dieser gleichzeitige Zugriff vom Programmierer ausgeschlossen werden, sollte die schnellere `ArrayList` verwendet werden.

`ArrayList` und `Vector` verwenden intern ein `Array` und sind deshalb sehr gut geeignet, wenn der Zugriff über einen Index erfolgen kann. Das dynamische Hinzufügen von Elementen hingegen kann kritisch werden, da das `Array` dann vergrößert werden muss, was nur durch Umkopieren (`System.arraycopy()`) erfolgen kann. Besonders schlimm ist dieser Effekt dann, wenn am Kopf der Liste Elemente eingefügt oder gelöscht werden. Müssen häufiger Elemente hinzugefügt und gelöscht werden, ist die `LinkedList` besser geeignet, da hier die Elemente durch gegenseitige Referenzen eine verkettete Liste aufbauen. Dafür ist bei der `LinkedList` der Zugriff über einen Index langsamer.

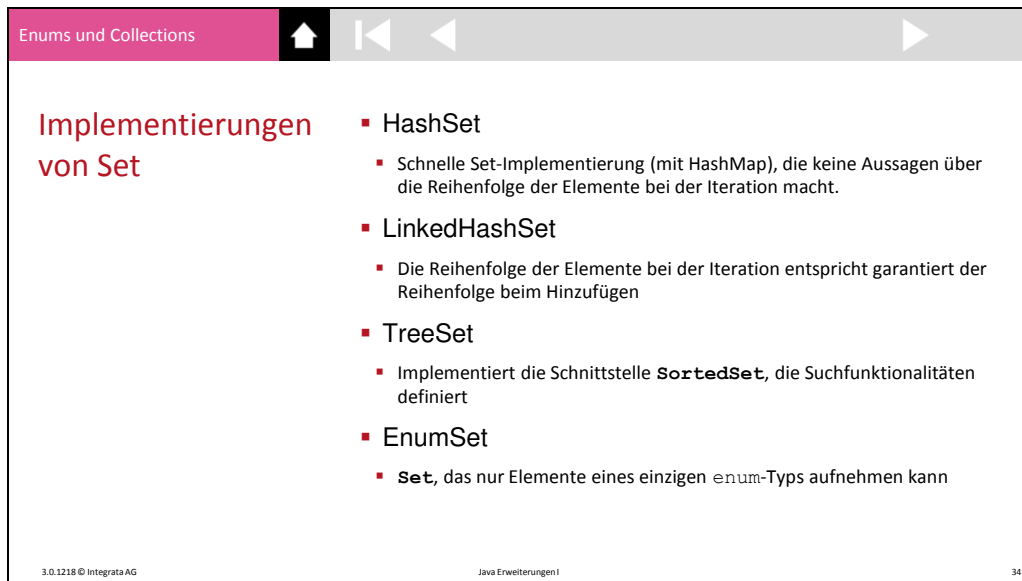
Der `Stack` ist ein LIFO (Last in, first out) Stack für Objekte und von `Vector` abgeleitet. Die zur Verfügung gestellten Methoden sind

```
empty()  
peek()  
push()  
search()
```

3.2.2 Queue

Die `LinkedList` implementiert ebenfalls die `Queue`-Schnittstelle. Andere Implementierungen befinden sich insbesondere im Paket `java.util.concurrent` und bieten spezielles Verhalten für den gleichzeitigen Zugriff nebenläufiger Anweisungen, so genannten Threads.

3.2.3 Set



Enums und Collections

Implementierungen von Set

- **HashSet**
 - Schnelle Set-Implementierung (mit HashMap), die keine Aussagen über die Reihenfolge der Elemente bei der Iteration macht.
- **LinkedHashSet**
 - Die Reihenfolge der Elemente bei der Iteration entspricht garantiert der Reihenfolge beim Hinzufügen
- **TreeSet**
 - Implementiert die Schnittstelle **SortedSet**, die Suchfunktionalitäten definiert
- **EnumSet**
 - **Set**, das nur Elemente eines einzigen `enum`-Typs aufnehmen kann

3.0.1218 © Integrata AG Java Erweiterungen I 34

Abb. 3-3: Set

Die Implementierungen der `Set`-Schnittstelle sind:

`HashSet`

`LinkedHashSet`

`TreeSet`

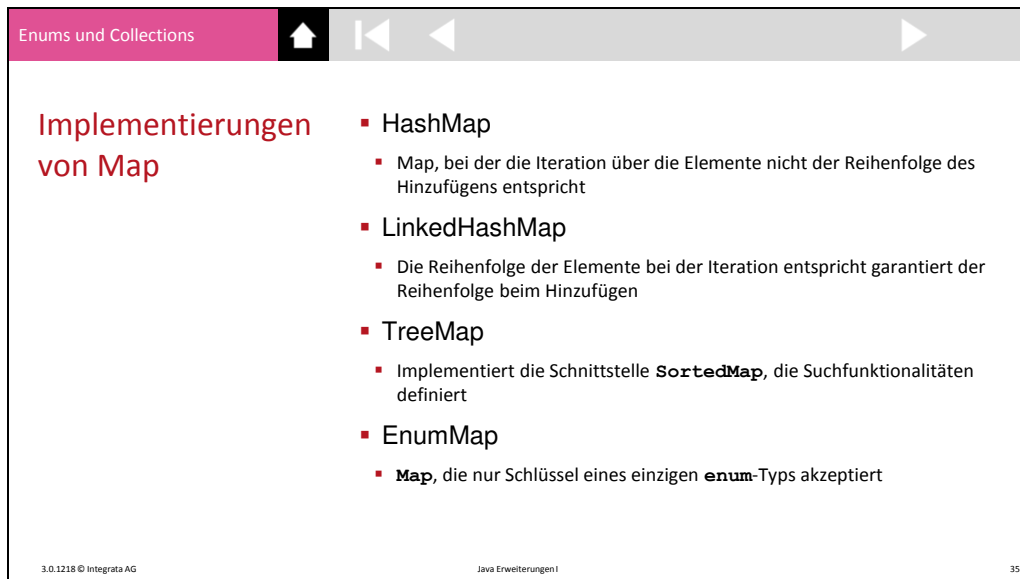
`EnumSet`

`HashSet` wird intern mit einer `HashMap` realisiert. `LinkedHashSet` garantiert bei der Iteration die Ordnung der Elemente, indem intern eine doppelt verkettete Liste geführt wird.

Das `TreeSet` implementiert die Schnittstelle `SortedSet`, eine Erweiterung der `Set`-Schnittstelle um Suchfunktionalitäten.

`EnumSet` verlangt, dass alle darin enthaltenen Elemente aus einem `enum`-Typ bestehen.

3.2.4 Map



Enums und Collections

Implementierungen von Map

- **HashMap**
 - Map, bei der die Iteration über die Elemente nicht der Reihenfolge des Hinzufügens entspricht
- **LinkedHashMap**
 - Die Reihenfolge der Elemente bei der Iteration entspricht garantiert der Reihenfolge beim Hinzufügen
- **TreeMap**
 - Implementiert die Schnittstelle **SortedMap**, die Suchfunktionalitäten definiert
- **EnumMap**
 - **Map**, die nur Schlüssel eines einzigen **enum**-Typs akzeptiert

3.0.1218 © Integrata AG Java Erweiterungen I 35

Abb. 3-4: Map

Die Implementierungen der `Map`-Schnittstelle sind:

`HashMap`

`Hashtable`

`IdentityHashMap`

`TreeMap`

`EnumMap`

`HashMap` und `Hashtable` unterscheiden sich wie `ArrayList` und `Vector` durch die Thread-Sicherheit. Während diese beiden Implementierungen bei der Suche nach einem Eintrag über einen Schlüssel dessen `hashCode()` und `equals()`-Methoden benutzen, verwendet die `IdentityHashMap` die Objektidentität (`==`).

Die `TreeMap` implementiert die Schnittstelle `SortedMap`, eine Erweiterung der `Map`-Schnittstelle um Suchfunktionalitäten.

Die `EnumMap` erwartet, dass die Schlüssel ausschließlich aus Werten eines einzigen Enumerations-Typs bestehen.

3.3 Generische Collections

3.3.1 Notwendigkeit generischer Collections

Seit der Sprachversion 5.0 ist mit den so genannten generischen Datentypen ein weiteres Sprachelement aufgenommen worden. Eine Fragestellung, die zu generischen Datentypen hinführt, scheint auf den ersten Blick fast trivial:

„Wie kann eine HashMap definiert werden, die als Schlüssel Integer-Objekte, als Werte String-Objekte enthält?“

Diese Aufgabe kann mit den bisher vorgestellten Sprachelementen nicht typsicher gelöst werden! Die folgenden Anweisungen führen zur Laufzeit zwangsläufig zu einem Fehler beim Cast bzw. sind logisch falsch (letzte Zeile), obwohl der Compiler die Klasse anstandslos¹ übersetzt:

```
Map table = new HashMap();
Integer tableKey = new Integer(5);
String tableValue = "Test";
table.put(tableKey, tableValue);

//Cast-Fehler beim Lesen des Objektes:
Integer wrong = (Integer)table.get(tableKey);

//Fehler beim Schreiben: Die Schlüssel sollten
//Integer-Objekte sein
table.put(tableValue, tableKey);
```

Generische Datentypen erlauben es nun, genau dieses Problem zu lösen.

¹ Der Compiler liefert aber zumindest Warnungen

Enums und Collections

Generische Collections

- Generische Datentypen erlauben eine typsichere Verwendung der Collection-Klassen
 - Bei der Deklaration der Collection wird bereits der Typ der verwendbaren Elemente bestimmt.
- Verwendung spezieller Platzhalter-Klassen bei der Deklaration und Erzeugung der Collection
- Beispiel ArrayList

```
List<String> liste = new ArrayList<String>( );
liste.add("Test");
```
- Beispiel Map

```
Map<Integer, String> table = new HashMap<Integer, String>();
Integer tableKey = new Integer(5);
String tableValue = "Test";
table.put(tableKey, tableValue);
String right = table.get(tableKey);
```

3.0.1218 © Integrata AG

Java Erweiterungen I

36

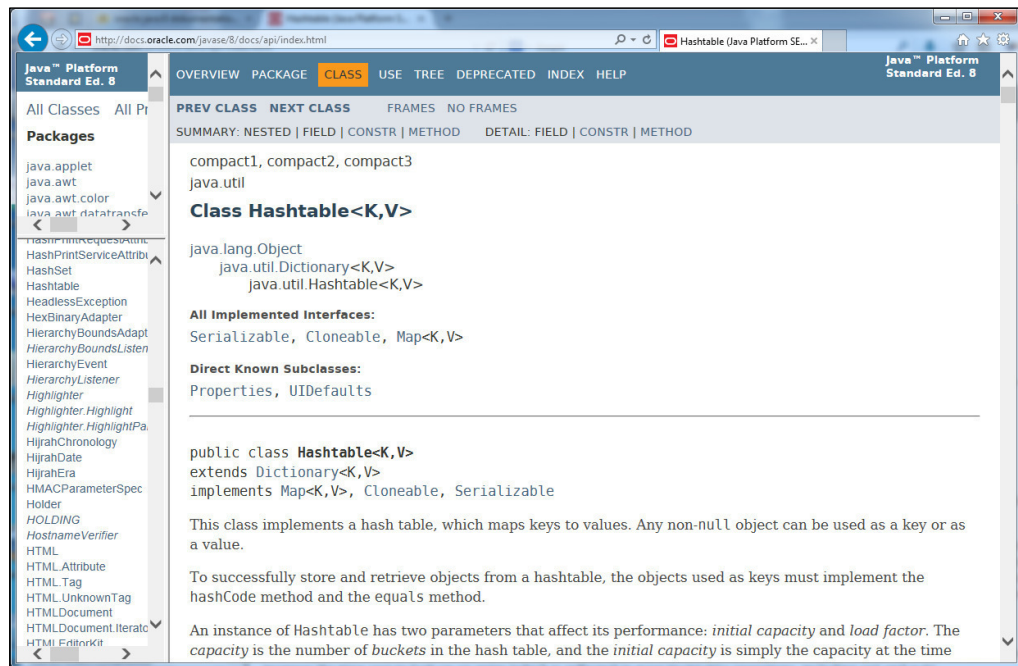
Abb. 3-5: Generische Collections

Bei der Definition der Klasse können in einem spitzen Klammern Paar Platzhalter-Typen geschrieben werden, die der Compiler dann prüfen kann:

```
Map<Integer, String> table = new HashMap<Integer, String>();
Integer tableKey = new Integer(5);
String tableValue = "Test";
table.put(tableKey, tableValue);
//Diese Zeile liefert nun einen Compiler-Fehler
Integer wrong = (Integer)table.get(tableKey);
//Diese Zeile ist richtig, kein Cast mehr notwendig
String right = table.get(tableKey);
//Auch diese Zeile liefert nun einen Compiler-Fehler
table.put(tableValue, tableKey);
```

3.3.2 Dokumentation

Ein Blick auf die Dokumentation der Klasse `HashMap` zeigt, wie generische Datentypen dort dargestellt werden. Um die „Platzhalter-Klassen“ von normalen Klassen zu unterscheiden empfiehlt Sun, erstere nur mit einem einzigen Großbuchstaben zu benennen.



keys

public [Enumeration](#)<[K](#)> **keys()**

Returns an enumeration of the keys in this hashtable.

Specified by:

[keys](#) in class [Dictionary](#)<[K](#), [V](#)>

Returns:

an enumeration of the keys in this hashtable.

See Also:

[Enumeration](#), [elements\(\)](#), [keySet\(\)](#), [Map](#)

3.3.3 Technischer Hintergrund

Mit der oben erläuterten Syntax ist es also möglich, die einzelnen Collection-Klassen typsicher zu verwenden; Fehler erkennt bereits der Java Compiler.

Es stellt sich nun aber die durchaus interessante Frage, wie dieser generische Ansatz zur Laufzeit umgesetzt werden wird. Dafür sind sofort mehrere Möglichkeiten denkbar:

- Die Java-Klassenbibliothek enthält für jede generische Collection eine spezielle Klasse.
- Generische Collections werden während des Übersetzens des Programms vom Compiler erkannt. Pro generische Collection wird eine eigene Klasse erzeugt. Dies entspricht im Wesentlichen einem Template-Ansatz, wie es beispielsweise C++ unterstützt.
- Aus den generischen Collections werden zur Laufzeit spezielle Klassen erzeugt.
- Zur Laufzeit unterscheiden sich generische Collections nicht voneinander. Die neu eingeführte Syntax wird nur vom Compiler benutzt, der bei der Erzeugung des Bytecodes beim Aufruf der get-Methoden einer Collection automatisch zusätzliche Cast-Anweisungen einführt.

Die erste Möglichkeit scheidet nach kurzem Nachdenken sofort aus: Die Platzhalter-Klassen könnten ja auch eben erst geschriebene eigenen Klassen sein, so dass die Klassenbibliothek unmöglich die spezielle Implementierung vorrätig haben kann.

Die zweite Hypothese kann auch sofort geprüft werden, indem im Verzeichnis der erzeugten Klassen nach Dateien gesucht wird, die nicht selber geschrieben wurden. Dies ist offensichtlich nicht der Fall, so dass auch statische Templates ausscheiden.

Der dritte Ansatz erfordert etwas mehr Aufwand. So können wir beispielsweise die Klassen zweier unterschiedlicher generischer Collections miteinander vergleichen:

```
public void test(){
    Map<String, String> stringMap = new HashMap<String, String>();
    Map<Integer, Integer> integerMap = new HashMap<Integer, Integer>();
    System.out.println(stringMap.getClass().getName());
    System.out.println(integerMap.getClass().getName());
    System.out.println(stringMap.getClass() == integerMap.getClass());
}
```

Die Ausgabe lautet:

```
java.util.HashMap
java.util.HashMap
true
```

und damit ist auch der dritte Ansatz nicht richtig.

Der vierte und letzte Ansatz scheint auf den ersten Blick etwas „wacklig“ zu sein, verdient aber dennoch eine nähere Betrachtung (natürlich auch deshalb, weil es der letzte verbliebene Ansatz ist...). Der klare Vorteil ist natürlich der, dass wie bisher mit einer einzigen Klasse alle generischen Collections behandelt werden können. Die Art der Umsetzung erfordert nur, dass der Compiler zusätzliche Prüfungen vornehmen kann und in der aufrufenden Klasse zusätzliche Cast-Anweisungen ergänzen muss. Und das ist doch genau das, was notwendig ist, nicht mehr, aber auch nicht weniger!

Zur Bestätigung der vierten Hypothese können wir beispielsweise versuchen, mit Hilfe des Reflection APIs für eine `HashMap` des Typs `<String, String>` die Methode `put(String, String)` zu suchen:

```
public void test2() {
    try {
        Map<String, String> stringMap = new HashMap<String, String>();
        StringMap.put("Key", "Value");
        System.out.println(stringMap.get("Key"));
        Class stringMapClass = stringMap.getClass();
        Class[] putParamTypes = {String.class, String.class};
        Method putMethod = stringMapClass.getMethod("put", putParamTypes);
        System.out.println(putMethod);
    }
    catch(Exception e) {
        e.printStackTrace();
    }
}
```

Die Ausgabe des Programmes lautet nun:

```
Value
java.lang.NoSuchMethodException:
java.util.HashMap.put(java.lang.String, java.lang.String)
    at java.lang.Class.getMethod(Class.java:1386)
    at com.rus.collections.CollectionDemo.test2(CollectionDemo.java:35)
    at com.rus.collections.CollectionDemo.<init>(CollectionDemo.java:15)
    at com.rus.collections.CollectionDemo.main(CollectionDemo.java:11)
```

Diese Ausgabe beweist nun die vierte Hypothese! Es gibt die in Zeile 4 aufgerufene Methode nur in der Variante `put(Object, Object)`. Das heißt, der Compiler meldet einen Fehler, obwohl die Methode „eigentlich“ da ist. Dieses so verwirrend oder widersprüchlich scheinende Verhalten ist bei näherer Betrachtung jedoch vollkommen richtig und ermöglicht eine äußerst effiziente Abbildung generischer Collections²!

² Neben den generischen Collections gibt es natürlich auch andere Klassen, die generische Datentypen einsetzen. Die angeführte Argumentation ist auch dort gültig.

3.3.4 Collections und einfache Datentypen

The screenshot shows a presentation slide with a pink header bar containing the text 'Enums und Collections' and navigation icons. The main content area has the title 'Collections und einfache Datentypen' in red. It lists three bullet points: 'Autoboxing/Unboxing', 'automatisch Wandlung von einfachen Datentypen (int, long, ...) in die entsprechende Wrapper-Klasse und zurück.', and 'Dadurch ist die Benutzung von Collections mit einfachen Datentypen möglich'. Below the third bullet point, there is a code snippet: 'Map<Integer, Double> intDoubleMap = new HashMap<Integer, Double>();' and 'intDoubleMap.put(42, 18.60);'. The footer of the slide contains '3.0.1218 © Integrata AG', 'Java Erweiterungen I', and the page number '37'.

Enums und Collections

Collections und einfache Datentypen

- Autoboxing/Unboxing
- automatisch Wandlung von einfachen Datentypen (`int`, `long`, ...) in die entsprechende Wrapper-Klasse und zurück.
- Dadurch ist die Benutzung von Collections mit einfachen Datentypen möglich
 - `Map<Integer, Double> intDoubleMap = new HashMap<Integer, Double>();`
 - `intDoubleMap.put(42, 18.60);`

3.0.1218 © Integrata AG Java Erweiterungen I 37

Abb. 3-6: Collections und einfache Datentypen

Bisher waren die Schlüssel und Werte einer Map stets Objektreferenzen. Es können aber auch einfache Datentypen verwendet werden. Die dazu verwendete Syntax ist aber nicht `HashMap<int, double>`, dies liefert einen Compiler-Fehler.

Da der Compiler jedoch keinen Unterschied zwischen einfachen Datentypen und den Wrapper-Klassen mehr macht, ist dies aber auch nicht notwendig.

```
private void mapMethod3(){
    Map<Integer, Double> intDoubleMap = new HashMap<Integer, Double>();
    intDoubleMap.put(42, 18.60);
}
```

Dieses automatische Konvertieren ist übrigens allgemein möglich und wird als „Autoboxing/Unboxing“ bezeichnet („Der einfache Datentyp wird automatisch in die Hülle des Wrappers gepackt bzw. daraus entpackt“).

3.4 Iteration über Collections

Jede `Collection` ist in der Lage, über die in ihr enthaltenen Elemente zu iterieren. Eine `Map` kann dies sowohl über ihre Schlüssel als auch über ihre Werte.

3.4.1 Iteration mit Iterator

Der `Iterator` ist natürlich auch typisiert:

```
private void listMethod(){
    List<String> stringList = new ArrayList<String>();
    stringList.add("Hello");
    stringList.add("World");
    Iterator<String> iter = stringList.iterator();
    while(iter.hasNext()){
        String element = iter.next();
        System.out.println( element );
    }
}
```

Dieses Verfahren ist jedoch nicht sonderlich elegant (die Deklaration des `Iterators`) und in verschachtelten Iterationen fehleranfällig.

3.4.2 Iteration mit `forEach`

Die bevorzugte Iteration erfolgt besser durch eine spezielle `for`-Schleife, die so genannte `foreach`-Schleife:

```
private void listMethod2(){
    List<String> stringList = new ArrayList<String>();
    stringList.add("Hello");
    stringList.add("World");
    for(String element:stringList){
        System.out.println( element );
    }
}
```

Wenn wir aber sehr große Listen haben, wäre es schön, wenn wir die Verarbeitung parallelisieren könnten. Wir müssten die Liste selber manuell aufteilen, um sie in mehreren Threads verarbeiten zu können. Doch zum Glück gibt es seit Java 8 eine bessere Möglichkeit.

3.4.3 Iteration mit `forEach`-Methode

Das von `Collection` implementierte Interface `Iterable` ist um eine Methode erweitert worden:

```
default void forEach(Consumer<? super T> action)
```

Damit wird uns diese aufwändige Arbeit abgenommen:

```
private void listMethod3(){
    List<String> stringList = new ArrayList<String>();
    stringList.add("Hello");
    stringList.add("World");
    stringList.forEach( new Consumer<String>( ) {
        public void accept( String element ){
            System.out.println( element );
        }
    });
}
```

Der Code ist zwar länger, aber wir haben den Vorteil einer Trennung von Iteration und Verarbeitung. Die `forEach`-Methode kann jetzt die Arbeit auf verschiedene Threads verteilen.

Der Code erscheint aber wesentlich unübersichtlicher.

3.4.4 Iteration mit `forEach`-Methode als Lambda-Ausdruck

Jetzt kommt ein Lambda-Ausdruck ins Spiel:

```
private void listMethod3(){
    List<String> stringList = new ArrayList<String>();
    stringList.add("Hello");
    stringList.add("World");
    stringList.forEach(
        (String element) -> { System.out.println(element); } );
}
```

Oder noch einfacher:

```
private void listMethod3(){
    List<String> stringList = new ArrayList<String>();
    stringList.add("Hello");
    stringList.add("World");
    stringList.forEach((element) -> System.out.println(element));
}
```


3.5 Die Klasse Collections

Enums und Collections

Hilfsklassen

- Collections
 - **Achtung:** nicht verwechseln mit dem Interface `Collection`)
 - Hilfsklasse für `List`, `Set` und `Map`
 - enthält allgemein verwendbare static Methoden
 - `copy`, `fill`, `list`, `max`, `min`, `reverse`, `shuffle`, `sort`, `swap`, ...
- Arrays
 - Hilfsklasse für Arrays
 - enthält allgemein verwendbare static Methoden zur Verwendung bei Arrays
 - `binarySearch`, `equals`, `fill`, `hashCode`, `sort`, `toString`, ...

3.0.1218 © Integrata AG Java Erweiterungen I 38

Abb. 3-7: Hilfsklassen

3.5.1 Sortieren von Collections

Enums und Collections

Sortieren von Elementen

- Interface `java.lang.Comparable`
 - Ermöglicht eine natürliche Folge der implementierenden Objekte.
 - Lists (und arrays), die dieses Interface implementieren, können `Collections.sort` (and `Arrays.sort`) sortiert werden.
 - Objekte können als Schlüssel/Elemente in sortierten Maps/Sets verwendet werden.
 - `public int compareTo(Object pToCompare);`
 - Rückgabewert negativ, 0, positiv
 - für natürliche Reihenfolge der Objekte, kompatibel mit `equals()`
- Interface `java.util.Comparator`
 - Kann alternativ benutzt zum Sortieren werden (Übergabe an `Collections.sort`).
 - `public int compare(Object pObj1, Object pObj2);`
 - Rückgabewert negativ, 0, positiv
 - für andere Sortierreihenfolge in einer Anwendung

3.0.1218 © Integrata AG Java Erweiterungen I 39

Abb. 3-8: Sortieren

Zum Sortieren der Elemente werden einige Interfaces bereitgestellt:

- `java.lang.Comparable` deklariert die Methode `int compareTo(java.lang.Object)`, eine implementierende Klasse definiert die Methode so, dass eine Einordnung in eine sortierte Liste durch Verwendung des Rückgabewertes erfolgen kann.

- Das Interface `java.util.Comparator` deklariert die Methode `int compare(Object pObj1, Object pObj2)` und kann dadurch beliebige Objekte vergleichen.

Die Sortier-Funktionalitäten des Collection-Frameworks beruhen maßgeblich auf diesen Typen. Wird beispielsweise eine Liste mit der Methode

```
static void sort(List list)
```

sortiert, so müssen die enthaltenen Referenzen allesamt vom Typ `Comparable` sein. Ist dies nicht der Fall, wird eine `ClassCastException` geworfen.

Um dies zu vermeiden, muss die Methode

```
static void sort(List list, Comparator comparator)
```

der `Collections`-Hilfsklasse benutzt werden. Diese zweite Methode wird auch dann gebraucht, wenn die Elemente nicht nach der in ihrer eigenen `compareTo`-Methode festgelegten "natürlichen Reihenfolge", sondern in einer für diese Anwendung benötigten anderen Reihenfolge sortiert werden sollen.

The screenshot shows a presentation slide with a pink header bar containing the text 'Enums und Collections' and navigation icons. The main content area has a title 'Sortieren von Elementen (Beispiel)' in red. It lists three examples of sorting a list of strings by length using `Collections.sort()`.

- Beispiel: Erzeugung einer List mit 3 Einträgen vom Typ String

```
List<String> tage = Arrays.asList( "Montag", "Dienstag", "Sonntag" );
```
- Sortierung der Elemente aufsteigend nach Länge des Strings mit einer anonymen Klasse

```
Collections.sort( tage,
    new Comparator<String>( ) {
        public int compare(String s1, String s2) {
            return s1.length() - s2.length();
        }
    } );
```
- Sortierung der Elemente mit Lambda-Ausdruck

```
Collections.sort( tage,
    ( String s1, String s2 ) -> { return s1.length() - s2.length(); } );
```
- Einfacher

```
Collections.sort( tage,
    (String s1, String s2) -> s1.length() - s2.length() );
```
- Noch einfacher

```
Collections.sort( tage, (s1, s2) -> s1.length() - s2.length() );
```

At the bottom of the slide, there is a footer with '3.0.1218 © Integrata AG', 'Java Erweiterungen I', and a page number '40'.

Abb. 3-9: Beispiel Sortieren

Auch Hilfsklassen des `java.util`-Paketes wie z. B. `java.util.TreeSet` und `java.util.TreeMap` nutzen diese Interfaces.

3.5.2 Unveränderbare Listen

Soll eine Liste nicht mehr verändert werden, so können die `unmodifiable<Type>`-Methoden der Klasse

`java.util.Collections` verwendet werden. Diese Methoden erwarten als Argument eine jeweils passende `Collection` und haben als Rückgabetyt ebenfalls eine `Collection`. Deren Implementierung wirft jedoch bei jedem Versuch einer Änderung (Hinzufügen, Löschen) eine Ausnahme.

```
Collection unmodifiableCollection(Collection)
```

```
List unmodifiableList(List)
```

```
Set unmodifiableSet(Set)
```

```
Map unmodifiableMap(Map)
```

```
SortedSet unmodifiableSortedSet(SortedSet)
```

```
SortedMap unmodifiableSortedMap(SortedMap)
```

3.5.3 Leere Listen

Werden beispielsweise als Rückgabewerte von Methoden leere Listen oder Maps benötigt, definiert `Collections` statische Methoden, die jeweils unveränderbare leere Ergebnisse liefern:

```
List emptyList()
```

```
Set emptySet()
```

```
Map emptyMap()
```


Die Vorteile dieses Ansatzes im Vergleich zur Erzeugung einer eigenen leeren Instanz sind:

- Überflüssige Instanziierungen werden vermieden
- Leere Collections sind stets identisch (`==` oder `equals`)

3.6 Enums

3.6.1 Ein Problem

Angenommen, man möchte die vier Jahreszeiten eindeutig identifizieren. Man könnte z.B. vier `int`-Konstanten definieren (z.B. in einem Interface):



The screenshot shows a presentation slide with a pink header bar containing the text 'Enums und Collections' and navigation icons. The slide content is as follows:

Enums:
Ein Problem

- Verwaltung von vier Jahreszeiten

```
public interface Season {
    public static final int SPRING = 0;
    public static final int SUMMER = 1;
    public static final int AUTUMN = 2;
    public static final int WINTER = 3;
}
```
- Aufruf

```
printSeason(Season.SPRING);
printSeason(Season.SUMMER * 2);
printSeason(42);
```

At the bottom of the slide, there is a footer with '3.0.1218 © Integrata AG', 'Java Erweiterungen I', and the page number '28'.

Abb. 3-10: Enum

Und dann z.B. die folgende Methode schreiben

```
static void printSeason(int season) {
    switch(season) {
        case Season.SPRING:
            System.out.println("Fruehling");
            break;
        case Season.SUMMER:
            System.out.println("Sommer");
            break;
        case Season.AUTUMN:
            System.out.println("Herbst");
            break;
        case Season.WINTER:
            System.out.println("Winter");
            break;
        default:
            throw new IllegalArgumentException();
    }
}
```

Diese Methode könnte dann wie folgt aufgerufen werden:

```
printSeason(Season.SPRING);
```

Leider könnte diese Methode aber auch mit völlig unsinnigen Parametern aufgerufen werden:

```
printSeason(Season.SUMMER * 2);
printSeason(42);
```

Denn eine Jahreszeit wird über einen `int` identifiziert – davon gibt's leider mehr, als es Jahreszeiten gibt. Und die `printSeason`-Methode ist natürlich mit jedem `int` zufrieden (zumindest sieht das der Compiler so – zur Laufzeit können "unsinnige" Aufrufe natürlich abgefangen werden).

3.6.2 Eine Lösung mit dem "alten" Java

Eine Lösung des oben diskutierten Problems besteht darin, die vier Jahreszeiten als Objekte zu repräsentieren – als Objekte einer Klasse `Season`. Dabei muss dann nur sichergestellt werden, dass es global exakt vier solcher Objekte gibt (nicht mehr und nicht weniger).

Die `Season`-Klasse bekommt einen privaten Konstruktor; und es existieren vier statische, konstante Referenzen, für welche innerhalb der Klasse selbst jeweils ein `Season`-Objekt erzeugt wird.

Hier eine erste, sehr einfache Variante:

Enums:
Lösung mit „altem“ Java

- Verwaltung von vier Jahreszeiten


```
public class Season {
    public static final Season SPRING = new Season();
    public static final Season SUMMER = new Season();
    public static final Season AUTUMN = new Season();
    public static final Season WINTER = new Season();

    private Season() {
    }
}
```
- Aufruf


```
static void printSeason(Season season) {
    if (season == Season.SPRING)
        System.out.println("Fruehling");
    else if (season == Season.SUMMER)
        System.out.println("Sommer");
    else if (season == Season.AUTUMN)
        System.out.println("Herbst");
    else if (season == Season.WINTER)
        System.out.println("Winter");
}
```

3.0.1218 © Integrata AG 29

Abb. 3-11: Enum

Der Parameter der Methode `printSeason` ist nun vom Typ `Season` – es kann also jeweils nur die Referenz auf eines der vier `Season`-Objekte übergeben werden. Z.B.: `printSeason(Season.SUMMER);`

Man könnte die `Season`-Klasse etwas intelligenter machen:

The screenshot shows a presentation slide with a pink header bar containing the text 'Enums und Collections' and navigation icons. The slide content is divided into two columns. The left column has red text: 'Enums:', 'Lösung mit „altem“', 'Java etwas', and 'intelligenter'. The right column has a red bullet point '▪ Verwaltung von vier Jahreszeiten' followed by a Java code block for the `Season` class. The code defines four static final constants (SPRING, SUMMER, AUTUMN, WINTER) as instances of the `Season` class, each with an ordinal and a name. It also includes private fields for `ordinal` and `name`, a constructor, and methods to return these values. An `@Override` annotation is present on the `toString` method. At the bottom of the slide, there is a footer with '3.0.1218 © Integrata AG' on the left, 'Java Erweiterungen I' in the center, and '43' on the right.

Enums:

Lösung mit „altem“

Java etwas intelligenter

▪ Verwaltung von vier Jahreszeiten

```
public class Season {
    public static final Season SPRING = new Season(0, "SPRING");
    public static final Season SUMMER = new Season(1, "SUMMER");
    public static final Season AUTUMN = new Season(2, "AUTUMN");
    public static final Season WINTER = new Season(3, "WINTER");

    private final int ordinal;
    private final String name;

    private Season(int ordinal, String name) {
        this.ordinal = ordinal;
        this.name = name;
    }

    public final int ordinal() {
        return this.ordinal;
    }

    public final String name() {
        return this.name;
    }

    @Override
    public String toString() { return this.name; }
}
```

3.0.1218 © Integrata AG

Java Erweiterungen I

43

Abb. 3-12: Enum

Nun kann z.B. die `name`-Methode auf `Season`-Objekte aufgerufen werden – sie liefert einen Namen zurück, der identisch ist mit dem Namen der entsprechenden statischen Referenzkonstanten. (Warum die Methoden die Namen `name` und `ordinal` heißen und nicht – wie gewohnt – `getName` resp. `getOrdinal`, wird später deutlich werden...)

Hier eine beispielhafte Verwendung:

```
Season s = Season.SPRING;
System.out.println(s.ordinal());
System.out.println(s.name());
System.out.println(s);
```

Die Ausgaben:

```
0
SPRING
SPRING
```

Und man könnte zur `Season`-Klasse schließlich noch weitere Intelligenz hinzufügen:

The screenshot shows a presentation slide with a pink header bar containing the text 'Enums und Collections' and navigation icons. The slide content is divided into two main sections. On the left, there is a red text block. On the right, there is a code block for a Java class named 'Season' and a small red square icon followed by a title. The code block contains the following Java code:

```
public class Season {  
    private static Map<String, Season> seasons =  
        new HashMap<String, Season>();  
    public static final ...  
    private final ...  
    private Season(int ordinal, String name) {  
        this.ordinal = ordinal;  
        this.name = name;  
        seasons.put(name, this);  
    }  
    public static Season valueOf(String name) {  
        Season s = seasons.get(name);  
        if (s == null) throw new IllegalArgumentException( );  
        return s;  
    }  
    public static Season[] values() {  
        return new Season[] { SPRING, SUMMER, AUTUMN, WINTER };  
    }  
    public final int ordinal() { return this.ordinal; }  
    public final String name() { return this.name; }  
    public String toString() { return this.name; }  
}
```

At the bottom of the slide, there is a small footer with the text '3.0.1218 © Integrata AG' on the left and 'Java Erweiterungen I' on the right, followed by the number '44'.

Abb. 3-13: Enum

Dann könnten z.B. die Seasons wie ausgegeben werden:

```
for (Season s : Season.values())  
    System.out.println(s);
```

Die Ausgaben:

SPRING

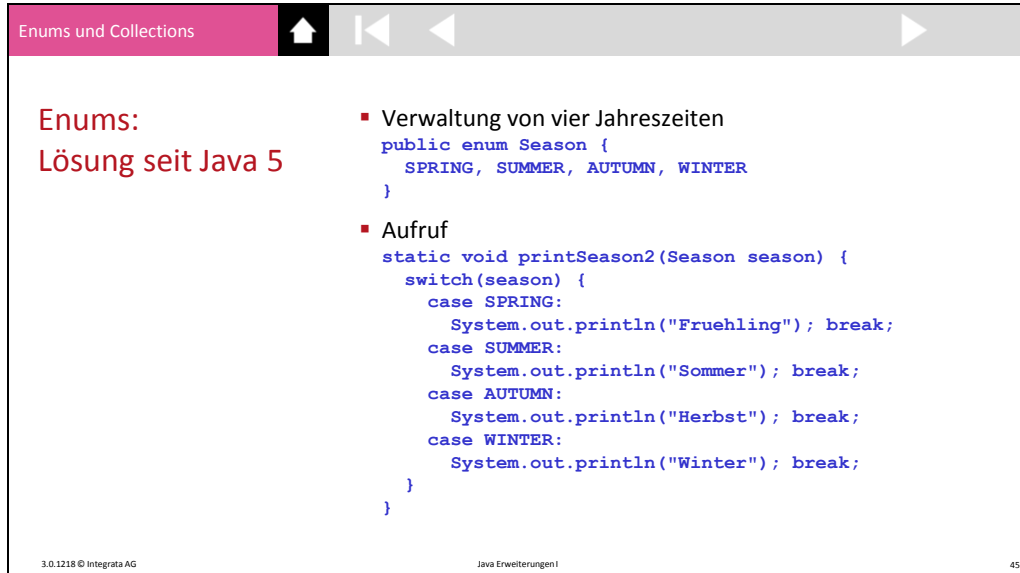
SUMMER

AUTUMN

WINTER

3.6.3 Enums: enum

Exakt die im letzten Abschnitt entwickelte `Season`-Klasse kann seit Java 5 einfacher formuliert werden:



Enums:
Lösung seit Java 5

- Verwaltung von vier Jahreszeiten

```
public enum Season {  
    SPRING, SUMMER, AUTUMN, WINTER  
}
```

- Aufruf

```
static void printSeason2(Season season) {  
    switch(season) {  
        case SPRING:  
            System.out.println("Fruehling"); break;  
        case SUMMER:  
            System.out.println("Sommer"); break;  
        case AUTUMN:  
            System.out.println("Herbst"); break;  
        case WINTER:  
            System.out.println("Winter"); break;  
    }  
}
```

3.0.1218 © Integrata AG Java Erweiterungen I 45

Abb. 3-14: Enum seit Java 5

Auch hier betreibt der Compiler wieder "Rewriting": Aus der obigen Definition wird eine Klasse, die im Prinzip identisch ist mit derjenigen, welche im letzten Abschnitt entwickelt wurde. Das neue `enum`-Konstrukt ist also nichts Anderes als "syntaktischer Zucker", welche eine recht einfache Formulierung erlaubt.

Auch diese `enum`-Klasse kann wie folgt verwendet werden:

```
Season s = Season.SPRING;  
System.out.println(s.ordinal());  
System.out.println(s.name());  
System.out.println(s);
```

(Nun wird deutlich, warum die getter-Methoden auch im letzten Abschnitt `name` und `ordinal` genannt wurden...)

```
for (Season s : Season.values())  
    System.out.println(s);  
  
Season s = Season.valueOf("SUMMER");  
System.out.println(s);  
System.out.println(s == Season.SUMMER);    // true
```

In Unterschied zur `Season`-Klasse, die im letzten Abschnitt entwickelt wurde, können "richtige" `enums` allerdings auch in einem `switch` verwendet werden.

4

Ein-/Ausgabe und Properties

4.1	Einführung in Java I/O	4-3
4.1.1	Ein erstes Beispiel	4-3
4.2	Die Klasse File.....	4-5
4.3	Byte-basierte Ein- und Ausgabeströme	4-7
4.3.1	Schreiben einer Byte-Datei	4-8
4.3.2	Schreiben einer Byte-Datei mit Ressourcen	4-8
4.4	Reader und Writer	4-9
4.4.1	Schreiben einer Zeichen-Datei	4-9
4.4.2	Gepuffertes Schreiben	4-10
4.5	Konfiguration einer Anwendung mit Properties	4-11
4.5.1	System-Properties	4-11
4.5.2	Die Klasse java.util.Properties	4-11
4.5.3	Die Konfigurationsdatei	4-12
4.5.4	Benutzerabhängige Konfiguration mit Preferences.....	4-13

4 Ein-/Ausgabe und Properties

4.1 Einführung in Java I/O

Mit Hilfe des Paketes `java.io` können einfache Werte und Java-Objekte in einen Datenstrom geschrieben und auch wieder gelesen werden. Auf diese Art und Weise können Objekte im gemeinsamen Speicher ausgetauscht und Objekte zwischen unterschiedlichen virtuellen Maschinen ausgetauscht werden.

The screenshot shows a presentation slide with a green header bar containing the text 'Ein-/Ausgabe und Properties' and navigation icons. The slide content is as follows:

- Einführung in Java I/O**
- Paket `java.io`
- Funktionalität:
 - Verwaltung von Verzeichnis-/Datei-Pfaden
 - `File`
 - Lesen und Schreiben von Dateien
 - `Reader`, `Writer`,
 - für 16-bit Text-I/O
 - `InputStream`, `OutputStream`
 - für 8-bit Binär-I/O
 - Austausch von Objekten im gemeinsamen Speicher
 - Austausch von Objekten zwischen verschiedenen virtuellen Maschinen
- Hier: Nur Lesen und Schreiben von Dateien

At the bottom of the slide, there is a footer with '3.0.1218 © Integrata AG' on the left, 'Java Erweiterungen I' in the center, and '47' on the right.

Abb. 4-1: Einführung

Im Rahmen dieser Präsentation werden wir uns auf den ersten Punkt beschränken. (Die anderen beiden werden im Bereich der verteilten Anwendungen eine wichtige Rolle spielen.)

4.1.1 Ein erstes Beispiel

Ein erster Blick auf das package `java.io` kann für den Java-Neuling durchaus erschreckend wirken. Das Paket enthält über 10 Interfaces und über 60 Klassen. Dies erweckt den Eindruck, dass das Lesen und Schreiben von Dateien mittels Java eine hochkomplizierte Angelegenheit ist. In vielen gängigen Programmiersprachen reichen für diese Aufgabe meist fünf Funktionen (`open`, `close`, `read`, `write` und `flush`). Wie das folgende Beispiel `WriteReadDemo.java` zeigt ist diese Aufgabe auch in Java recht einfach zu lösen:

```
import java.io.*;

public class WriteReadDemo {

    public static void main(String[] args) {
        FileWriter fw;
        FileReader fr;
        int c;

        try {
            fw = new FileWriter("daten.txt");
            fw.write("Das ist ein Text");
            fw.close();

            fr = new FileReader("daten.txt");
            while ((c = fr.read()) != -1) {
                System.out.print((char) c);
            }
            fr.close();
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}
```

Wenn es nun doch so einfach ist, wozu dann die über 60 Klassen in `java.io`. Spezielle Aufgaben werden in einzelnen Klassen implementiert. Anstelle weniger Klassen mit sehr vielen Methoden wurden viele Klassen mit einem einfachen überschaubaren Satz von Methoden implementiert. Als Beispiel soll das zeilenweise Einlesen einer Textdatei mit Nummerierung der Zeilen dienen (**LineNumberReader.java**):

```
import java.io.*;

public class LineReader {

    public static void main(String[] args) {
        LineNumberReader f;
        String line;

        try {
            f = new LineNumberReader(new
                FileReader("c:\\_training\\bin\\setup.bat"));
            while ((line = f.readLine()) != null) {
                System.out.print(f.getLineNumber() + ": ");
                System.out.println(line);
            }
            f.close();
        } catch (IOException e) {
            System.out.println("Fehler beim Lesen der Datei");
        }
    }
}
```

4.2 Die Klasse File

The screenshot shows a presentation slide with a title bar at the top containing the text 'Ein-/Ausgabe und Properties' and navigation icons. The main content area has the title 'Klasse File' in red. Below the title is a bulleted list of points regarding the File class. At the bottom of the slide, there is a footer with the text '3.0.1218 © Integrata AG' on the left, 'Java Erweiterungen I' in the center, and '48' on the right.

Klasse File

- Instanziierung unter Angabe eines Pfadnamens
 - Relative und absolute Angaben sind möglich
- Trennzeichen variieren zwischen den verschiedenen Betriebssystemen
 - '\\\' unter Windows
 - '/' unter Unix/Linux
- Java Laufzeitumgebung definiert Trennzeichen
 - Systemvariable `file.separator`
 - Auslesen mit: `System.getProperty("file.separator")`
 - Klassenattribut `File.separator`

3.0.1218 © Integrata AG Java Erweiterungen I 48

Abb. 4-2: File

Dateien und Verzeichnisse werden als Instanz der Klasse `java.io.File` zur Verfügung gestellt. Der Konstruktor erwartet, dass der Name einer Datei z. B. in der Form:

`C:\integrata\seminar_3304\test.java`

angegeben wird. Relative und absolute Angaben sind möglich. Das Trennzeichen variiert von Betriebssystem zu Betriebssystem, wird aber durch die Systemvariable "file.separator" durch den Aufruf

```
System.getProperty("file.separator");
```

oder noch einfacher durch das Klassenattribut

```
File.separator
```

ausgegeben. Eine Instanz der Klasse `java.io.File` kann unter anderem Dateien anlegen, auflisten, aber auch zum Lesen und Schreiben verfügbar machen.

Ein `File` kann zum einen Informationen über eine Datei oder die Verzeichnisstruktur anzeigen, zum anderen kann eine Instanz dieser Klasse den Ein- und Ausgabeströmen zur Verfügung gestellt werden.

```
import java.io.*;

public class IODemo {

    public IODemo() {
        String lFoldername =
            "_training";
        File lFile = new File(lFoldername);
        String[] lFiles = lFile.list();
        for (int i = 0; i < lFiles.length; i++) {
            System.out.println(lFiles[i]);
        }
    }
    catch(IOException pException) {
        System.out.println("Ausnahme Write/Read: "
            + pException.getMessage());
    }
}

public static void main(String[] args) {
    new IODemo();
}
```

4.3 Byte-basierte Ein- und Ausgabeströme

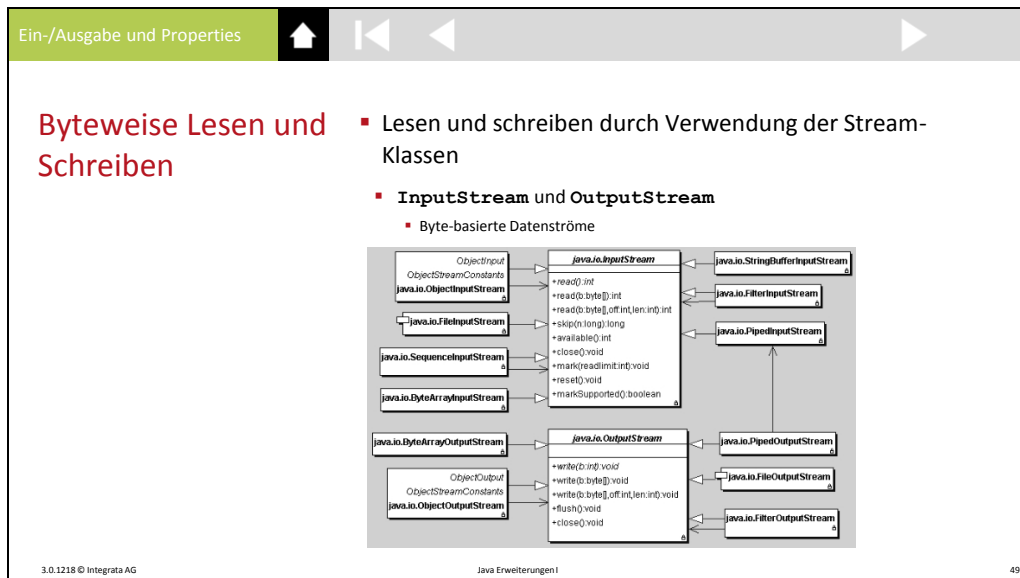


Abb. 4-3: Byteweise Lesen und Schreiben

Allein mit der Instanz der Klasse `File` kann eine Datei weder gelesen, noch geschrieben werden. Die abstrakten Klassen `java.io.OutputStream` und `java.io.InputStream` und deren konkrete Subklassen unterstützen das Speichern und Lesen von Byteströmen.

Die Byteströme werden immer dann benötigt, wenn Binärdaten verarbeitet werden sollen. Oft ist auch die Übertragung von Daten über ein Netzwerk (z.B. `HTTP`) byte-basiert. Zur Umwandlung zwischen Byte- und Characterströmen dienen die Klassen

`java.io.InputStreamReader`

und

`java.io.OutputStreamWriter`.

Beide Klassen besitzen einen Konstruktor, der jeweils einen Bytestrom als Argument erwartet.

Zur Speicherung und Auslesen von Byteströmen gibt es die Klassen `java.io.FileOutputStream` und `java.io.FileInputStream`.

4.3.1 Schreiben einer Byte-Datei

Ein-/Ausgabe und Properties

Beispiel:
try
Schreiben einer
Byte-Datei

```
OutputStream out = null;  
try {  
    out = new FileOutputStream("demo.bin");  
    out.write( 65 );  
} catch (IOException e) {  
    System.out.println(e);  
}  
finally {  
    try {  
        if (out != null)  
            out.close();  
    } catch (IOException e) {  
        System.out.println(e);  
    }  
}
```

3.0.1218 © Integrata AG

Java Erweiterungen I

50

Abb. 4-4: Byte-Datei schreiben

4.3.2 Schreiben einer Byte-Datei mit Ressourcen

Ein-/Ausgabe und Properties

Beispiel:
try mit Ressourcen
Schreiben einer Datei

```
try ( final OutputStream out = new  
        FileOutputStream("demo.bin")  
    ){  
    out.write( 65 );  
} catch (IOException e) {  
    System.out.println(e);  
}
```

3.0.1218 © Integrata AG

Java Erweiterungen I

51

Abb. 4-5: Schreiben einer Byte-Datei mit Ressourcen

4.4 Reader und Writer

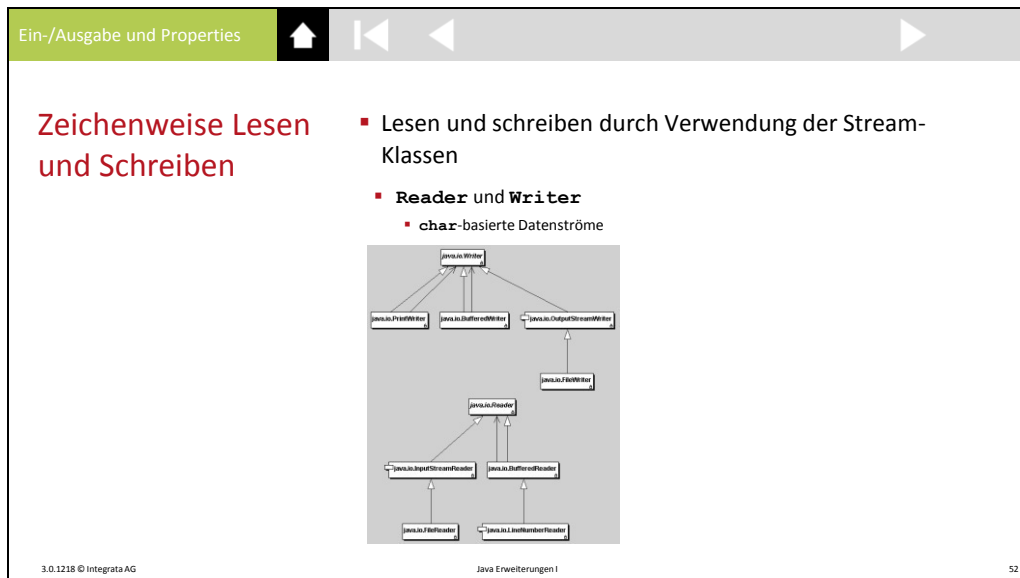


Abb. 4-6: Zeichen-Datei

Zum zeichenweisen (16 bit Unicode `char`) Lesen bzw. Schreiben wird ein `Writer` oder `Reader`-Objekt benötigt:

Diese Klassen bieten dann komfortable Methoden zum Lesen und Schreiben einer Datei an, wie z. B. zeilenweise lesen und schreiben, anhängen etc.

4.4.1 Schreiben einer Zeichen-Datei

Ein-/Ausgabe und Properties

**Beispiel:
Schreiben einer
Zeichen-Datei**

```
try (final Writer writer = new
    FileWriter("demo.txt") ) {
    writer.write( "Eins" );
}
catch (IOException e) {
    System.out.println(e);
}
```

3.0.1218 © Integrata AG

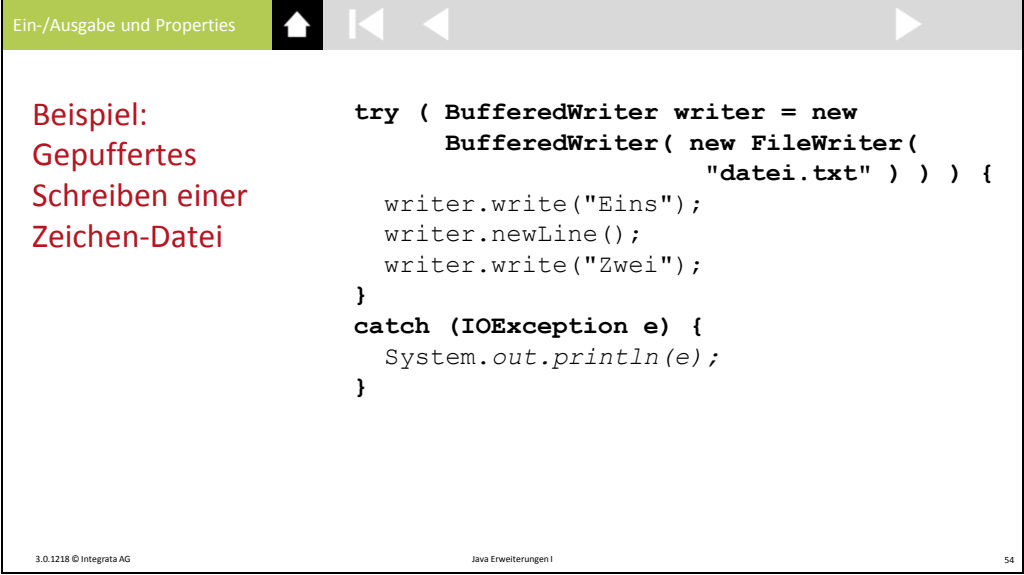
Java Erweiterungen I

53

Abb. 4-7: Schreiben einer Zeichen-Datei

Natürlich kann beim Lesen oder Schreiben in eine Datei auch eine Ausnahme geworfen werden. Dies ist die `java.io.IOException` oder eine ihrer Subklassen.

4.4.2 Gepuffertes Schreiben



The screenshot shows a presentation slide with a title bar 'Ein-/Ausgabe und Properties' and navigation icons. The slide content is as follows:

**Beispiel:
Gepuffertes
Schreiben einer
Zeichen-Datei**

```
try ( BufferedWriter writer = new
    BufferedWriter( new FileWriter(
        "datei.txt" ) ) ) {
    writer.write("Eins");
    writer.newLine();
    writer.write("Zwei");
}
catch (IOException e) {
    System.out.println(e);
}
```

At the bottom of the slide, there is a footer with '3.0.1218 © Integrata AG' on the left, 'Java Erweiterungen I' in the center, and '54' on the right.

Abb. 4-8: Gepuffertes Schreiben

Mit Hilfe der Klasse `java.io.BufferedWriter` können ganze Zeilen übertragen werden. Das Lesen erfolgt dann analog mit Hilfe der Klasse `java.io.BufferedReader`.

4.5 Konfiguration einer Anwendung mit Properties

4.5.1 System-Properties

Sehr einfach erfolgt die Konfiguration einer Anwendung mit Hilfe von System-Properties. Dazu bietet die Klasse `System` die Methode

```
String getProperty(String)
```

an. System-Properties werden entweder durch

```
void setProperty(String key, String value)
```

oder, was sicherlich häufiger der Fall sein wird, beim Aufruf der Virtuellen Maschine durch die Option

```
java -Dkey=value -Dkey2=value2 Application
```

gesetzt.

Hinweis: Beliebige Betriebssystemvariablen wie `PATH` können so nicht gelesen werden. Die Virtuelle Maschine stellt nur einen beschränkten Satz (z. B. `user.home`) an diesen Variablen zur Verfügung. Eine Auflistung liefert `System.getProperties()`;

4.5.2 Die Klasse `java.util.Properties`

The screenshot shows a presentation slide with a title bar 'Ein-/Ausgabe und Properties' and navigation icons. The main content is titled 'Klasse Properties' in red. On the left, a code window displays the class hierarchy and methods of `java.util.Properties`, which extends `java.util.Hashtable`. The methods listed are: `+Properties()`, `+Properties(java.util.Properties)`, `+getProperty(java.lang.String, java.lang.Object)`, `+load(java.io.InputStream)void`, `+save(java.io.OutputStream, java.lang.String)void`, `+store(java.io.OutputStream, java.lang.String)void`, `+getProperty(java.lang.String) java.lang.String`, `+getProperty(java.lang.String, java.lang.String) java.lang.String`, `+propertyNames() java.util.Enumeration`, `+list(java.io.PrintStream)void`, and `+list(java.io.PrintWriter)void`. On the right, a bulleted list summarizes the class's features: it is an extension of `Hashtable`, it allows reading and writing properties, and it provides methods for loading and storing properties from streams. The bottom of the slide contains the text '3.0.1218 © Integrata AG' on the left, 'Java Erweiterungen I' in the center, and '55' on the right.

Klasse Properties

- Erweiterung der `Hashtable`
- Properties lesen und schreiben
 - `String getProperty(String key)`
 - `void setProperty(String key, String value)`
- Lesen und speichern der Properties durch die Methoden
 - `load(InputStream pIn)`
 - `store (OutputStream pOut, String pHeader)`

3.0.1218 © Integrata AG Java Erweiterungen I 55

Abb. 4-9: Properties

Das `java.util`-Paket enthält eine sehr nützliche Hilfsklasse, um Anwendungen einfach durch eine Datei konfigurieren zu können. Dies ist die Klasse `java.util.Properties`.

`Properties` ist eine Erweiterung der `java.util.Hashtable` und folglich in der Lage, wie diese, Schlüssel-Werte-Paare aufzunehmen, die in diesem Fall nicht beliebige Objekttypen, sondern Strings sein müssen. Das Besondere ist nun aber, dass diese Paare einfach aus einem `InputStream` lesen (Methode `load(InputStream in)`) bzw. in einen `OutputStream` schreiben (Methode `store(OutputStream out, String header)`) zu können.

Eine einzelne Property kann mit der Methode `setProperty(String key, String value)` gesetzt bzw. mit einer der `getProperty()`-Methoden gelesen werden. Zu beachten ist, dass als Schlüssel und Werte nur Strings empfohlen sind.

Die Verwendung der Methode `put(Object key, Object value)` aus `java.util.Hashtable` wird von Sun selbst nicht empfohlen. Nachträglich stellt sich hier natürlich die Frage, ob das Erben von `Hashtable` eine gute Idee war. Aber Sun ändert das öffentliche Java API nicht mehr, und das ist auch gut so.

4.5.3 Die Konfigurationsdatei

Das Format eines Properties-Files ist einfach les- und editierbar:

```
#Properties für die Applikation
#Thu Aug 23 20:35:00 GMT+02:00 2019
eins=15
zwei=30
drei=100
```

Der Ort einer Properties-Datei kann in den Klassenpfad der Anwendung gelegt werden. Dann ist ein Zugriff möglich über die Methode

```
java.io.InputStream getResourceAsStream(String pResourceName)
```

der Klasse `java.lang.Class`. Die Ressource `pResourceName` wird relativ zu dem Pfad (Paketstruktur inklusive!) gesucht, aus dem die Klasse geladen wurde. So wird mit dem Aufruf

```
lObject.getClass().getResourceAsStream("config.properties");
```

die Datei `config.properties` in dem Pfad gesucht, in dem auch die Bytecode-Datei zur `lObject`-Klassendefinition gefunden wurde. Das Auslesen der Properties-Datei ist dann sofort möglich mit:

```
Properties lProps = new Properties();

lProps.load(lObject.getClass().getResourceAsStream
            ("config.properties"));
```

Soll die Ressource in der Wurzel des Klassenpfades gefunden werden, muss der angegebene Pfad mit einem Schrägstrich `"/` beginnen:

```
lObject.getClass().getResourceAsStream("/config.properties");
```

4.5.4 Benutzerabhängige Konfiguration mit Preferences

Mit Hilfe der oben angeführten Properties-Dateien ist es noch nicht automatisch möglich, in Abhängigkeit vom angemeldeten Benutzer Konfigurationseinstellungen in einem Profil abzulegen. Es ist zwar prinzipiell möglich, durch die Verwendung der System-Property `user.home` verschiedene Properties-Dateien anzusprechen. Dies erfordert aber zusätzlichen Programmieraufwand.

The screenshot shows a presentation slide with a title bar 'Ein-/Ausgabe und Properties' and navigation icons. The main content is titled 'Klasse Preferences' in red. It lists several bullet points: 'Benutzerabhängige Konfigurationseinstellungen', 'Properties-Dateien ablegen im User-Verzeichnis (System-Property: `user.home`)', 'oder Benutzung von `java.util.prefs.Preferences`', 'Ablage der Preferences', 'bei UNIX' (with sub-bullet 'im User-Home Verzeichnis'), 'unter Windows' (with sub-bullet 'in der Registry'), 'Hierarchische Organisation mit den Einstiegspunkten', '`public static Preferences userRoot()`', '`public static Preferences systemRoot()`', and 'Setzen von Informationen durch `put`-Methoden'. The footer contains '3.0.1218 © Integrata AG', 'Java Erweiterungen I', and the page number '56'.

Abb. 4-10: Preferences

Die Klasse `java.util.prefs.Preferences` übernimmt nun genau diese Aufgabe, und zwar natürlich Betriebssystem-unabhängig. So werden unter Unix-basierten Systemen wie oben aufgeführt spezielle Properties-Dateien im User-Home Verzeichnis abgelegt, unter Windows wird die Registry verwendet.

Einstiegspunkt für die Verwendung von Preferences sind die beiden statischen Methoden

```
public static Preferences userRoot()  
public static Preferences systemRoot()
```

Das dadurch erzeugte Preferences-Objekt kann nun benutzt werden, Einstellungen durch die Angabe von Strings als Schlüssel Informationen abzulegen (`put<Type>`-Methoden) bzw. wieder auszulesen (`get<Type>`-Methoden). Zulässige Typen sind:

- `boolean`
- `byte[]`
- `double`
- `float`

- int
- long
- String

wobei die Methoden für letztere nur `put` bzw. `get` heißen.

Die Preferences sind nicht flach organisiert, sondern hierarchisch. Eine Navigation durch den Baum erfolgt durch die Methoden

```
public Preferences node(String pathName)
public Preferences parent()
```

Beispiel:

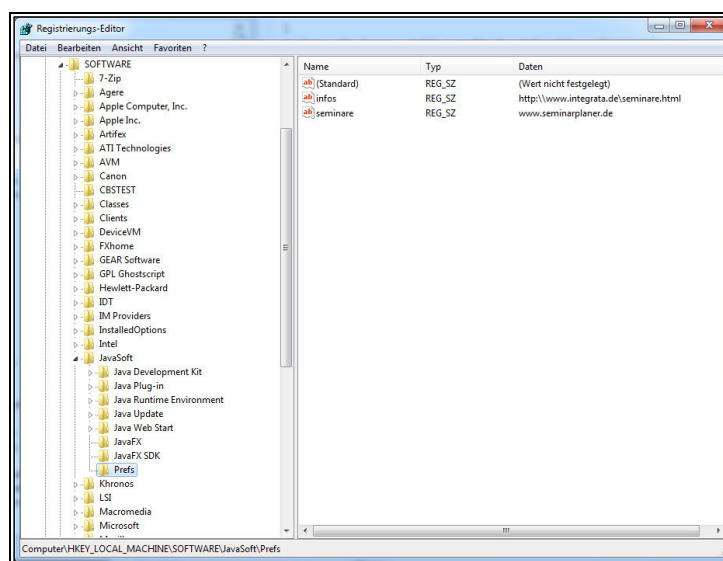
```
import java.util.prefs.BackingStoreException;
import java.util.prefs.Preferences;

public class PreferencesDemo {

    public static void main(String[] args) {
        Preferences prefs = Preferences.systemRoot();
        prefs.put("infos", "http://www.integrata.de/seminare.html");
        prefs.put("seminare", "www.seminarplaner.de");
        try {
            prefs.flush();
        }
        catch (BackingStoreException e) {
            e.printStackTrace();
        }
    }
}
```

Die mit dem Beispielcode gesetzten Werte findet man bei Windows in der Registry unter dem Eintrag:

HKEY_LOCAL_MACHINE\SOFTWARE\JavaSoft\Prefs



5

GUI-Programmierung

5.1	Das Abstract Windowing Toolkit (AWT).....	5-3
5.2	Das AWT Package	5-5
5.3	Swing.....	5-6
5.3.1	Die Swing-Bibliothek.....	5-6
5.3.2	Vor- und Nachteile von Swing.....	5-7
5.3.3	Beispiel: JFrame und JButton	5-7
5.4	Swing-Komponenten	5-9
5.4.1	Widgets.....	5-9
5.5	Layout Manager.....	5-10
5.5.1	FlowLayout	5-11
5.5.2	BorderLayout	5-12
5.5.3	GridLayout	5-15
5.5.4	Weitere Layouts	5-16
5.5.5	GridBagLayout.....	5-17
5.5.6	Schachtelung von Panels	5-17
5.6	Event Handling	5-19
5.7	Event-Objekte.....	5-20
5.8	Das Listener-Konzept.....	5-21
5.8.1	Event-Klassen, Listener und Methoden	5-21
5.8.2	Topologie der Listener	5-22
5.9	Designalternativen für Listener	5-23

5.9.1	Externe Klasse als Listener	5-23
5.9.2	Container als Listener	5-24
5.9.3	Innere Klasse als Listener.....	5-25
5.9.4	Lokale Klasse als Listener	5-25
5.9.5	Anonyme Klasse als Listener.....	5-26
5.9.6	Lambda-Ausdruck als Listener	5-27
5.9.7	Verwendung einer Adapter-Klasse	5-27
5.10	Zusammenfassende Wertung	5-28

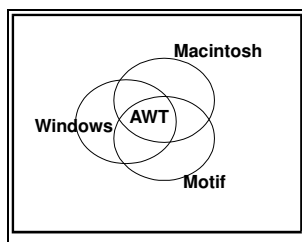
5 GUI-Programmierung

Kaum eine Client-Anwendung kommt heute ohne grafische Benutzeroberfläche (GUI = Graphical User Interface) aus. In diesem Kapitel lernen Sie die Struktur der GUI-Programmierung mit Java, die Ihnen auch im Umgang mit „Visuellen Editoren“ hilfreich sein wird. Sie werden die Prinzipien der Oberflächen-Programmierung kennen lernen. Schwerpunkt ist hier weniger die Programmierung komplexer Dialogfolgen, sondern die Darstellung allgemein gültiger Strategien.

Die Bibliotheken, die für die GUI-Programmierung benötigt werden, befinden sich im package `java.awt`, bzw. `javax.swing`.

5.1 Das Abstract Windowing Toolkit (AWT)

Die AWT-Klassen ermöglichen es, grafische Oberflächen zu erstellen, die unter allen Plattformen benutzt werden können. Die eigentliche Darstellung von AWT-Komponenten erfolgt mit plattformspezifischen Peer-Klassen.



Die schwergewichtigen (heavyweight) Komponenten sind nicht in Java geschrieben, sondern verwenden Betriebssystem-Funktionsaufrufe.

GUI-Programmierung

Grafische Benutzeroberfläche

- GUIs auf allen Plattformen
 - AWT-Klassen sind die plattformunabhängige Schnittstelle für den Programmierer
 - Peer-Klassen sind plattformabhängige, „schwergewichtige“ Klassen, die Betriebssystem-Aufrufe durchführen
- Java Foundation Classes – Swing
 - Eigener Fenster-Manager mit plattformunabhängigen Komponenten
 - sog. „lightweight“-Klassen, einstellbares look&feel
- Standard Widget Toolkit
 - Eingeführt durch Eclipse
 - Peer-Klassen wie AWT aber wesentlich umfangreicher
 - <http://www.eclipse.org/swt/>
- Java FX
 - JavaFX-UI-Toolkit: FXML ([XML](#)-basierte Sprache) zum Erstellen von Oberflächen
 - Ein *scene graph* verwaltet die einzelnen Bestandteile einer GUI

3.0.1218 © Integrata AG Java Erweiterungen I 58

Abb. 5-1: Grafische Benutzeroberfläche

5.2 Das AWT Package

Die meisten Klassen in der `java.awt`-Bibliothek sind sog. Widgets wie z. B. Knöpfe oder Listen. Weitere Komponenten sind Text-Komponenten, Container und eine Leinwand (`Canvas`), auf der gezeichnet werden kann. Grafische Komponenten werden in einem Container angeordnet. Dabei bestimmt der verwendete Layout-Manager die Platzierung der Komponenten. Außerdem gibt es weitere Werkzeuge-Klassen für Farben, Grafiken, Fonts, Menüs und das Drucken. Da diese Klassen den kleinsten gemeinsamen Nenner der Plattformen darstellen, sind diese nicht sehr mächtig. Benutzerfreundliche GUIs können mit Swing erstellt werden.

Die Java AWT-Klassen können in drei Gruppen eingeteilt werden:

- Komponenten (Button, Menu, List u.s.w.)
- LayoutManager (BorderLayout, FlowLayout, GridLayout u.s.w.)
- Utility-Klassen (Graphics, Font, Color, Polygon u.s.w.)

Folgendes Bild zeigt eine unvollständige Übersicht der AWT-Klassen:

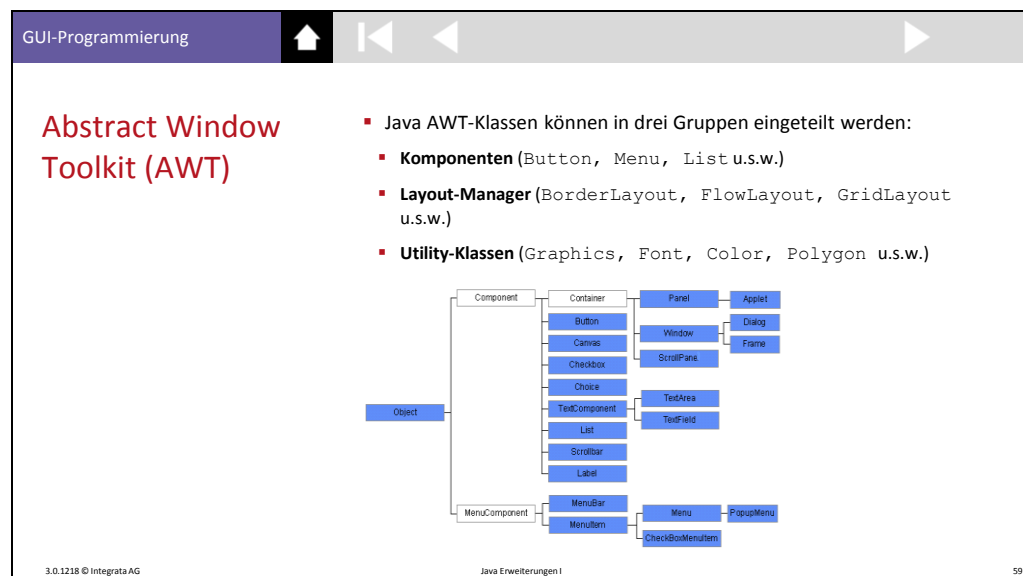


Abb. 5-2: AWT

5.3 Swing

5.3.1 Die Swing-Bibliothek

Die Java Foundation Classes (JFC, Codename "Swing") sind eine Gemeinschaftsentwicklung von Sun Microsystems und Netscape Corporation. Die JFC-Klassen erweitern die Funktionalität von AWT um nützliche Elemente für die GUI-Programmierung, z. B. ein Spreadsheet um Datenbankinhalte darzustellen (JTable). Swing ist seit Version 1.2 Teil der Java SE (J2SE) Klassenbibliothek.

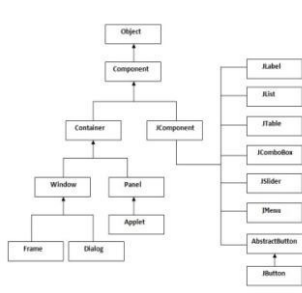
Die JFC bestehen aus:

- SWING
- Accessibility
- Java 2D
- Drag and Drop

Swing besteht aus einigen neuen Widgets und einem Pluggable Look and Feel. Im Gegensatz zum AWT-Button ist der Swing JButton komplett in Java geschrieben und kann unter Windows auf Wunsch auch so aussehen als wäre er ein Motif-Knopf. Swing-Klassen sind in 100 % Pure Java geschrieben.

GUI-Programmierung
⬆ ⬅ ➡ ⬆

Swing



```

graph TD
    Object --> Component
    Component --> Container
    Component --> JComponent
    Container --> Window
    Container --> Panel
    Window --> Frame
    Window --> Dialog
    Panel --> Applet
    JComponent --> JLabel
    JComponent --> JList
    JComponent --> JTable
    JComponent --> JScrollbar
    JComponent --> JSlider
    JComponent --> JMenu
    JComponent --> AbstractButton
    AbstractButton --> JButton
          
```

- Pakete **javax.swing** und Unterpakete (Klassen beginnen mit **J** z.B. **JButton**)
- 100 % in Java geschrieben ("leichtgewichtig")
- Sehr umfangreiche Möglichkeiten (trees, tables, tabbed pane)
- JavaBeans
- Widgets: Subklassen von **JComponent**
- Pluggable Look and Feel
- Model View Controller (MVC) Design Pattern
 - Trennung von View und Model
- Nachteile:
 - eventuell längere Ladezeiten
 - eventuell geringere Ausführungsgeschwindigkeit als bei Peer-Klasse

3.0.1218 © Integrata AG
Java Erweiterungen I
60

Abb. 5-3: Swing

5.3.2 Vor- und Nachteile von Swing

Da Swing-Klassen als JavaBeans realisiert sind, können Sie einfach mit visuellen Builder-Tools zu mächtigen Oberflächen zusammengebaut und angepasst werden.

Vorteile:

- Komplette in Java geschrieben
- Sehr umfangreiche Möglichkeiten (trees, tables, tabbed pane)
- Komponenten sind auch Container (Knöpfe mit Bild und Text)
- Trennung von View und Modell (plugable look and feel)

Nachteile:

- mehr Klassen, dadurch eventuell etwas längere Ladezeiten
- Die Ausführungsgeschwindigkeit ist etwas geringer als bei der Verwendung der ins System integrierten AWT-Peer-Klassen

Die Swing-Klassen befinden sich im Package: `javax.swing` und in weiteren Unter-Package.

Beispiele für Swing-Komponenten sind: `JFrame`, `JApplet`, `JButton`, `JProgressBar`, `JTable` uvm.

5.3.3 Beispiel: `JFrame` und `JButton`

Das Vorgehen ist ähnlich wie in AWT, bis auf die Tatsache, dass Komponenten nicht direkt in den Swing-Container gelegt werden, sondern in die `ContentPane`.

Die Klasse `JFrame` ist direkt abgeleitet von `java.awt.Frame`

```
public class JFrame extends java.awt.Frame
    implements WindowConstants, Accessible, RootPaneContainer
```

Ein `JFrame` besteht aus einer `RootPane` einer `LayeredPane` (`MenuBar` und `ContentPane`) und einer `GlassPane`. Die `ContentPane` ist der innere Teil des Fensters, in Gegensatz zu AWT, wo die Koordinaten relativ zum äußeren Eckpunkt des Fensterrahmens angegeben werden mussten und dadurch je nach Rahmenbreite und Menüleiste oft falsch waren, werden die Koordinaten hier direkt innerhalb des tatsächlich verfügbaren Platzes angegeben.

```
import javax.swing.*;
import java.io.*;
import java.util.*;

class SwingTest {
    public static void main(String[] args) {
        new SwingTest();
    }

    SwingTest() {
        JFrame frame = new JFrame("Titel");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.add("Center", new JButton("Click"));
        frame.pack();
        frame.setVisible(true);
    }
}
```

Die Mächtigkeit der Swing-Klassenbibliothek ist so umfangreich, dass hier nicht näher auf dieses Thema eingegangen werden kann. Swing wird im Seminar 3308 "GUI-Entwicklung mit Java Swing" ausführlich behandelt.

5.4 Swing-Komponenten

Die Klassen der grafischen Komponenten sind im Paket `javax.swing` gruppiert.

Die Vererbungshierarchie in diesem Paket lässt sich einfach zusammenfassen:

Object

- Component
 - einfache Widgets: JLabel, JButton, JCheckbox, JComboBox, JList, ...
 - TextComponent
 - JTextField, JTextArea
 - Container
 - JWindow, JFrame, JDialog, JPanel, JApplet, ...
 - Canvas

5.4.1 Widgets

Die Instanzen der Widget-Klassen sind voll funktionsfähig, d. h. die Standardfunktionalität ist bereits implementiert: Schaltflächen reagieren auf einen Mausklick durch eine entsprechende grafische Animation, in Textfeldern werden Tastaturanschläge übernommen, Listen verwalten ihre Einträge und stellen bei Bedarf Bildlaufleisten zur Verfügung.

Benötigt man Pixel-Grafiken, so erstellt man eine Subklasse der eigens hierfür vorgesehene Klasse `JPanel`. Sie ist auch eine Art Leinwand zum Zeichnen von Pixelgrafiken. Als Komponente lässt sie sich in einem Container wie dem `JApplet` anordnen. In der Subklasse der Klasse `JPanel` soll die `paintComponent()`-Methode überschrieben werden, die den grafischen Inhalt angibt, sowie einige weitere Methoden, die dem Layout-Manager die Größe angeben.

5.5 Layout Manager

Ein Layout Manager ist nichts anderes als eine Vorschrift, nach der zur Laufzeit die in einem Container enthaltenen Komponenten dimensioniert und positioniert werden.

The screenshot shows a presentation slide with a purple header bar containing the text 'GUI-Programmierung' and navigation icons. The slide title is 'Layoutmanager' in red. The content is as follows:

- Ein LayoutManager sorgt für eine plattformunabhängige und dynamische Platzierung der Komponenten im Container
 - FlowLayout
 - BorderLayout
 - GridLayout
 - GridBagLayout
 - ...
- Ohne LayoutManager:
 - `setLayout (null)` : *Nachteil: keine dynamische Anpassung an Fensteränderungen*
 - Lokation und Größe aller Komponenten setzen:
 - `setBounds (int x, int y, int width, int height)`

At the bottom left of the slide, it says '3.0.1218 © Integrata AG'. At the bottom right, it says 'Java Erweiterungen I' and '61'.

Abb. 5-4: Layoutmanager

Die wichtigsten Klassen des Paketes `java.awt`, die das Interface `LayoutManager` implementieren, sind:

<code>FlowLayout</code>	Standard für Panels und Applets
<code>BorderLayout</code>	Standard für Dialoge und Frames
<code>GridLayout</code>	Einteilung der Bedienelemente in Zeilen und Spalten
<code>CardLayout</code>	Dies wird dann benutzt, wenn verschiedene Bedienelemente zu unterschiedlichen Zeiten benötigt werden.
<code>GridBagLayout</code>	Bedienelemente können in verschiedener Größe vertikal und horizontal angeordnet werden.

Auf drei Layouts soll näher eingegangen werden.

5.5.1 FlowLayout

Die in diesem Layout verwendeten Elemente erscheinen nebeneinander in horizontaler Anordnung. Die Distanz zwischen diesen Elementen kann angegeben werden. Ist das Fenster zu klein, können die Elemente auch untereinander ausgegeben werden.

Die Komponenten werden in ihrer bevorzugten Größe dargestellt. Der Konstruktor von `FlowLayout` ist unten angegeben.

Mit dem ersten Parameter wird die Ausrichtung der Komponenten und mit den beiden anderen Parametern der horizontale und vertikale Abstand der Komponenten in Pixel angegeben.

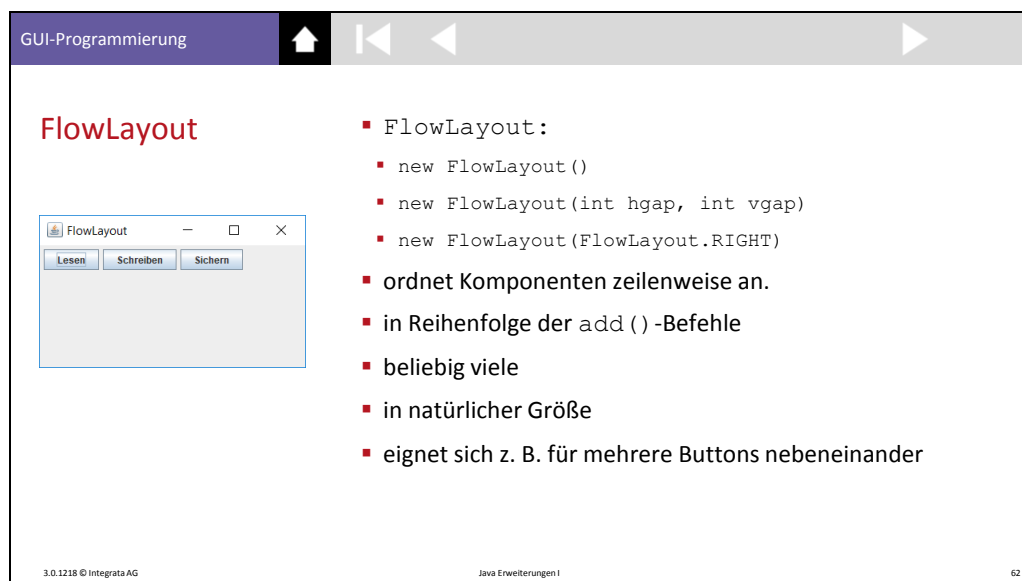


Abb. 5-5: FlowLayout

Beispiel:

```
import javax.swing.*;

import java.awt.FlowLayout;
import java.io.*;
import java.util.*;

class SwingTest {
    public static void main(String[] args) {
        new SwingTest();
    }

    SwingTest() {
        JFrame frame = new JFrame("FlowLayout");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.add( new FlowDemo( ) );
        frame.setSize(350, 200 );
        frame.setVisible(true);
    }
}
```

```
private class FlowDemo extends JPanel {
    private JButton knopf1, knopf2, knopf3;
    public FlowDemo () {
        setLayout(new FlowLayout(FlowLayout.LEFT));
        knopf1 = new JButton("Lesen");
        knopf2 = new JButton("Schreiben");
        knopf3 = new JButton("Sichern");
        add( knopf1 ); add( knopf2 ); add( knopf3 );
    }
}
```

Es werden Referenzen für drei Objekte vom Typ Button angelegt.

Es werden drei Objekte mit `new` erzeugt und anschließend mit `add()` dargestellt.

Die einzelnen Schaltflächen sind noch nicht mit Funktionalität hinterlegt! Die so genannten Callbacks müssen noch definiert und registriert werden.

5.5.2 BorderLayout

Mittels dieses Layouts können 5 Bedienelemente an 5 verschiedenen Stellen positioniert werden. Diese werden folgendermaßen angegeben: **oben** (`BorderLayout.NORTH`), **unten** (`BorderLayout.SOUTH`), **links** (`BorderLayout.WEST`), **rechts** (`BorderLayout.EAST`) und **das Zentrum** (`BorderLayout.CENTER`).

Neben dem parameterlosen Konstruktor

```
BorderLayout()
```

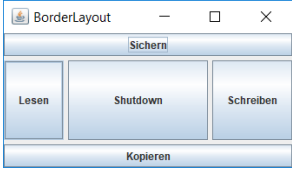
existiert noch ein weiterer Konstruktor

```
BorderLayout(int hgap, int vgap)
```

mit dem die horizontalen und vertikalen Distanzen benachbarter Elemente angegeben werden können.

GUI-Programmierung

BorderLayout



NORTH		
WEST	CENTER	EAST
SOUTH		

- BorderLayout:
- `new BorderLayout()`
- `new BorderLayout(int hgap, int vgap)`
- maximal eine Komponente an 4 Rändern und in Mitte
- Zusatzparameter im `add()`-Befehl
- `add(component, BorderLayout.NORTH)`
- und analog mit `EAST`, `WEST`, `SOUTH`, `CENTER`
- füllt Container komplett aus
- Komponenten auf ganze Breite bzw. Höhe verzerrt
- eignet sich für die Aufteilung des Fensters in größere Bereiche über oder nebeneinander (Panels, siehe geschachtelte Layouts)

3.0.1218 © Integrata AG

Java Erweiterungen I

63

Abb. 5-6: BorderLayout

Die Größe der Komponenten wird entsprechend der Container-Größe angepasst. Elemente im Norden und Süden wachsen horizontal, Elemente im Osten und Westen vertikal und das Element in der Mitte wächst in beide Richtungen beim Vergrößern des Containers.

Beispiel:

```
import java.awt.BorderLayout;
import java.awt.Font;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

class SwingTest {
    public static void main(String[] args) {
        new SwingTest();
    }

    SwingTest() {
        JFrame frame = new JFrame("BorderLayout");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.add(new BorderDemo());
        frame.setSize(350, 200);
        frame.setVisible(true);
    }

    private class BorderDemo extends JPanel {
        private JButton links, rechts, oben, unten, mitte;

        public BorderDemo() {
            setFont(new Font("Courier", Font.BOLD, 18));
            setLayout(new BorderLayout(5, 5));
            links = new JButton("Lesen");
            rechts = new JButton("Schreiben");
            oben = new JButton("Sichern");
            unten = new JButton("Kopieren");
            mitte = new JButton("Shutdown");

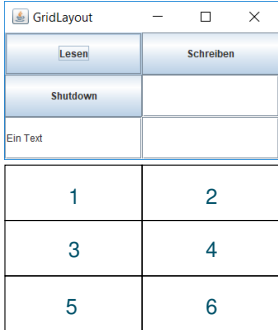
            add(links, BorderLayout.WEST);
            add(rechts, BorderLayout.EAST);
            add(oben, BorderLayout.NORTH);
            add(unten, BorderLayout.SOUTH);
            add(mitte, BorderLayout.CENTER);
        }
    }
}
```

5.5.3 GridLayout

Die Panel-Fläche wird in eine Tabelle unterteilt, die Komponenten werden nacheinander in die einzelnen Zellen eingefügt. In einer Zelle kann genau eine Komponente enthalten sein, es können keine Zellen übersprungen werden.

GUI-Programmierung
⏮ ⏪ ⏩ ⏭

GridLayout



- GridLayout:
- `new GridLayout(int rows, int cols);`
- `new GridLayout(int rows, int cols, int hgap, int vgap);`
- ordnet Komponenten in Gitter an
- in Reihenfolge der add()-Befehle
- es wird die genau richtige Anzahl rows*cols erwartet
- alle Komponenten gleich groß
- eignet sich z.B. für mehrere Eingabefelder übereinander

3.0.1218 © Integrata AG
Java Erweiterungen I
64

Abb. 5-7: GridLayout

Die Komponenten werden in der jeweiligen Zellgröße dargestellt.

Konstruktoren:

```
GridLayout(int rows, int cols)
```

```
GridLayout(int rows, int cols, int hgap, int vgap)
```

Beispiel: GridDemo.java

```
import java.awt.Font;
import java.awt.GridLayout;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;

class SwingTest {
    public static void main(String[] args) {
        new SwingTest();
    }

    SwingTest() {
        JFrame frame = new JFrame("GridLayout");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.add(new GridDemo());
        frame.setSize(350, 200);
        frame.setVisible(true);
    }
}
```

```

    }

    public class GridDemo extends JPanel {
        private JButton links, links1, rechts;
        private JTextField links2, rechts2, rechts1;

        public GridDemo () {
            setFont( new Font("Courier", Font.BOLD, 18 ) );
            setLayout( new GridLayout(3, 2));
            links = new JButton("Lesen");
            rechts = new JButton("Schreiben");
            links1 = new JButton("Shutdown");
            rechts1 = new JTextField(10);
            links2 = new JTextField("Ein Text");
            rechts2 = new JTextField(10);

            add( links );
            add( rechts );
            add( links1 );
            add( rechts1 );
            add( links2 );
            add( rechts2 );
        }
    }
}

```

Es ist zu beachten, dass die Angabe der Größe der Komponenten (z. B. 20 Zeichen bei den Eingabefeldern) durch die Verwendung des Layouts überschrieben wird.

5.5.4 Weitere Layouts

Es kann auch ohne Verwendung eines Layout-Managers gearbeitet werden:

```
setLayout( null );
```

schaltet den Layout-Manager aus. Dann muss der Programmierer jedoch in jedem Falle durch Aufruf der `setBounds()` - Methode die Dimensionierung und Positionierung der Komponente selbst übernehmen. Die damit erzielte pixel-genaue Positionierung kann allerdings zu unschönen Ergebnissen führen, z.B. wenn zur Laufzeit eine andere Bildschirmauflösung und damit eine andere Schriftgröße verwendet wird.

Auch eigene Layout-Manager können erzeugt werden, indem eine Klasse das Interface `LayoutManager` oder `LayoutManager2` implementiert.

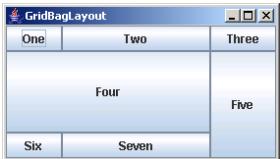
Professionelle Entwicklungstools (IDE) zur Erstellung von Oberflächen arbeiten oftmals mit dem `GridBagLayout`.

Dieses ist zwar am flexibelsten, führt aber oft dazu, dass die Oberfläche nicht sauber entworfen wird und somit nicht wieder verwendbar ist.

5.5.5 GridBagLayout

GUI-Programmierung

GridBagLayout



- GridBagLayout:
- `new GridBagLayout();`
- Ausrichtung der Komponenten vertikal und horizontal mit unterschiedlicher Größe
- Jede Komponenten kann eine oder mehrere Zellen des Gitters einnehmen
- Jede Komponenten wird von einer Instanz von GridBagConstraints gesteuert
- Das Constraints-Objekt spezifiziert die genaue Position
- Es wird die natürliche Größe einer Komponente verwendet

3.0.1218 © Integrata AG
Java Erweiterungen I
65

Abb. 5-8: GridBagLayout

5.5.6 Schachtelung von Panels

Oftmals benötigt man Verbundkomponenten, d. h. eine Komponente, die selber aus mehreren Komponenten besteht, also einen Container. Jedes Panel kann mit einem anderen Layout-Manager ausgestattet werden. Die so erstellten Panels können in einem Container oder Panel angeordnet werden. Dieses Vorgehen erspart oft den Einsatz von GridBagLayout und ermöglicht eine einfache Wiederverwertung der Panels.

Beispiel:

```
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class NpMain extends JFrame {
    public static void main(String[] args) {
        new NpMain();
    }

    public NpMain() {
        JPanel masterPanel = new JPanel();
        masterPanel.setLayout(new BorderLayout());
```



```

JPanel subPanel1 = new JPanel();
JPanel subPanel2 = new JPanel();
JPanel subPanel3 = new JPanel();

subPanel1.setLayout(new GridLayout(3, 2));
subPanel1.add(new JLabel("A"));
subPanel1.add(new JTextField(10));
subPanel1.add(new JLabel("B"));
subPanel1.add(new JTextField(10));
subPanel1.add(new JLabel("C"));
subPanel1.add(new JTextField(10));

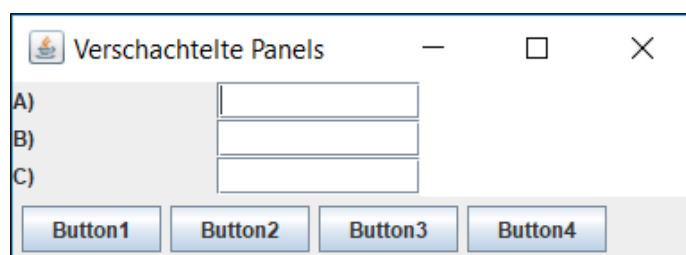
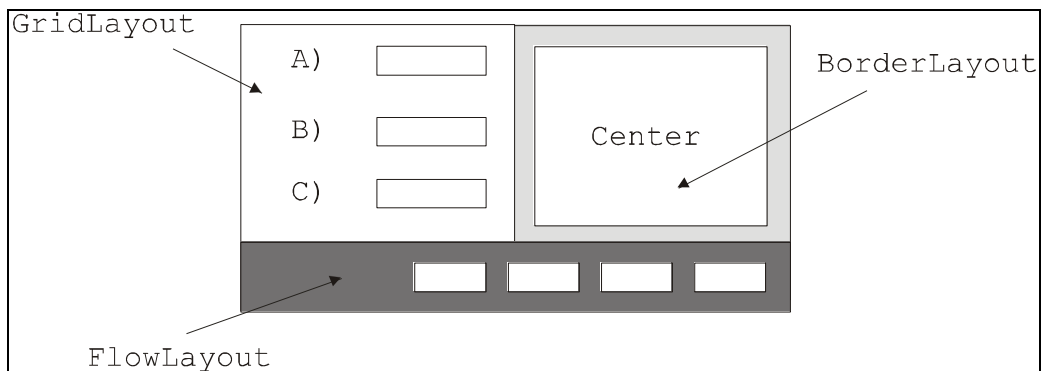
subPanel2.setLayout(new BorderLayout(20, 20));
subPanel2.add(new JTextArea(4, 14));

subPanel3.setLayout(new FlowLayout(FlowLayout.LEFT));
subPanel3.add(new JButton("Button1"));
subPanel3.add(new JButton("Button2"));
subPanel3.add(new JButton("Button3"));
subPanel3.add(new JButton("Button4"));

masterPanel.add(subPanel1, BorderLayout.WEST);
masterPanel.add(subPanel2, BorderLayout.CENTER);
masterPanel.add(subPanel3, BorderLayout.SOUTH);

this.add(masterPanel);
this.setTitle("Verschachtelte Panels");
this.pack();
this.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
});
this.setVisible(true);
}

```



5.6 Event Handling

Das Event Handling der Java AWT-Bibliothek beruht auf einem Delegationsmodell. Dadurch kann eine Trennung des Modells (Listener) und der Oberfläche (Quelle) vorgenommen werden.

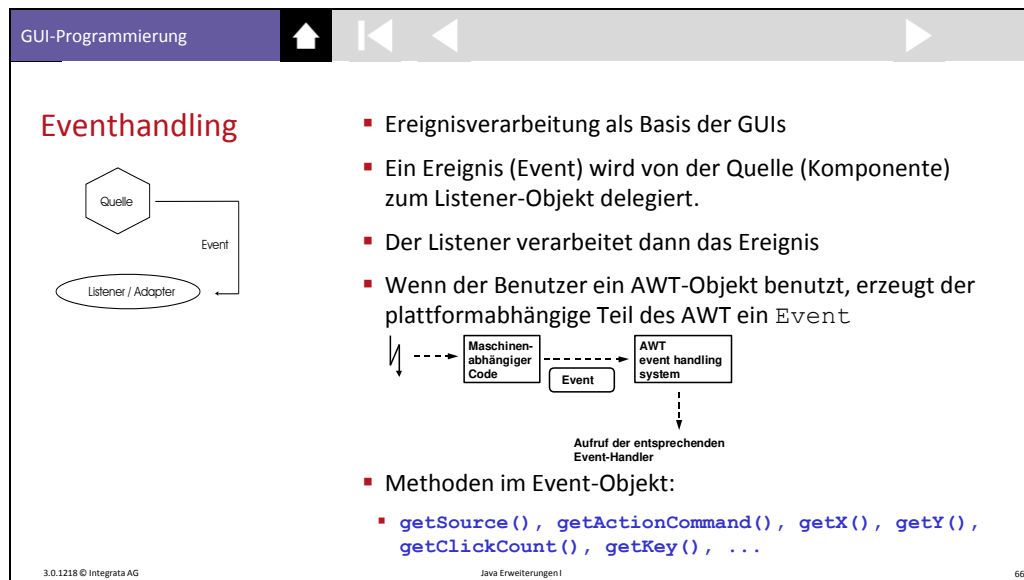


Abb. 5-9: Eventhandling

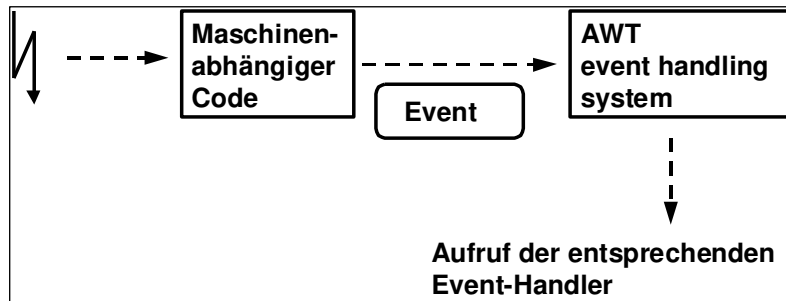
Ein Ereignis wird von der Quelle (Komponente) zum Listener-Objekt delegiert.

Der Listener verarbeitet dann das Ereignis. Listener sind selber Objekte, die sich bei der Quelle registrieren lassen können, um auf ein bestimmtes Ereignis zu reagieren.

Das AWT kennt zwei unterschiedliche Ereignistypen: Zum einen Low-Level-Ereignisse wie Tasten- oder Mausbewegungen. Zum anderen gibt es semantische Ereignisse wie z. B. das Erlangen eines Fokus oder aber die Eingabe durch "Return" oder "Doppelklick" mit der Maus. Alle Events wiederum sind Objekte, die das Ereignis selbst beschreiben.

5.7 Event-Objekte

Der eigentliche Low-Level-Event wird vom maschinenabhängigen Betriebssystem auf Grund eines Hardware-Ereignisses generiert:



Das Betriebssystem signalisiert den Event dem aktiven Prozess. In unserem Fall ist dies die virtuelle Maschine. In Java werden Events als Instanzen von Event-Klassen behandelt, die Subklassen der Klasse `java.awt.AWTEvent` sind, d. h., es wird eine Instanz dieser Klasse erzeugt. Die Eventklassen enthalten Methoden, mit denen Eigenschaften des Events abgefragt werden können. Ein paar wichtige Beispiele sind:

- In der Superklasse `AWTEvent` und damit in allen Events:
 - `getSource()` liefert das Objekt (die Komponente), in der der Event aufgetreten ist, also z.B. den Button, der angeklickt wurde, oder das TextField, in dem eine Eingabe erfolgte.
- In der Klasse `ActionEvent`:
 - `getCommandString()` liefert den Text des Buttons bzw. des Menüeintrags
- In der Klasse `MouseEvent`:
 - `getX()`, `getY()` liefert die Koordinaten, wo sich die Maus beim Ereignis befunden hat
 - `getClickCount()` liefert die Art eines Mausklicks (Einfach-, Doppel- oder Dreifachklick)
- In der Klasse `KeyEvent`:
 - `getKey()` liefert die Taste, die gedrückt wurde

5.8 Das Listener-Konzept

Das Event-Modell basiert auf dem Konzept der Delegation an einen oder mehrere Listener.

5.8.1 Event-Klassen, Listener und Methoden

Eine Java-Komponente wird als Event-Verursacher betrachtet. Ein beliebiges Objekt kann als Listener für den erzeugten Event registriert werden. Damit ein Objekt ein Listener sein kann, muss die zugrunde liegende Klasse in jedem Falle ein Listener-Interface implementieren. Die verschiedenen Events sind in verschiedenen Listenertypen und Event-Klassen gruppiert. Diese befinden sich im package `java.awt.event`.

The screenshot shows a presentation slide with a title bar 'GUI-Programmierung' and navigation icons. The slide content is as follows:

Event-Klassen, Listener-Interfaces und Methoden

- `ActionEvent`
- `ActionListener`
 - `actionPerformed()`
- `WindowEvent`
- `WindowListener`
 - `windowOpened()`, `windowClosing()`, `windowClosed()`,
`windowActivated()`, `windowDeactivated()`,
`windowIconified()`, `windowDeiconified()`
- `MouseEvent`
- `MouseListener` und `MouseMotionListener`
 - `mouseEntered()`, `mouseExited()`,
`mousePressed()`, `mouseReleased()`, `mouseClicked()`,
`mouseMoved()`, `mouseDragged()`
- `KeyEvent`
- `KeyListener`
 - `keyPressed()`, `keyReleased()`, `keyTyped()`

At the bottom left, it says '3.0.1218 © Integrata AG'. At the bottom right, it says 'Java Erweiterungen I' and '67'.

Abb. 5-10: Event-Klassen

Ein `Listener` registriert sich bei der Event-Quelle mit Hilfe einer `addXXXListener()`-Methode, von der es für jedes Interface eine passende gibt, also z. B.:

```
public void addActionListener(ActionListener al)
public void addWindowListener(WindowListener wl)
```

Wir unterscheiden Low-Level-Events wie Maus-Klick, Tastatur-Klick und semantische Events wie Action oder Focus. Ein `ActionEvent` wird z. B. durch die Low-Level-Events Maus-Klick oder Leertaste ausgelöst.

5.8.2 Topologie der Listener

Die Listener können sich an verschiedenen Stellen des Programms befinden. Oft werden hierfür innere Klassen eingesetzt. Besonders Entwicklungswerkzeuge benutzen oft inneren Klassen als Listener, die sogar automatisch generiert und compiliert werden. Dieser Ansatz ist jedoch nicht bedenkenlos anwendbar.

Das dynamische Zusammenspiel zwischen Event-Quelle und Listener und die verwendeten Methoden ist in folgender Grafik anhand eines `ActionEvents` veranschaulicht:

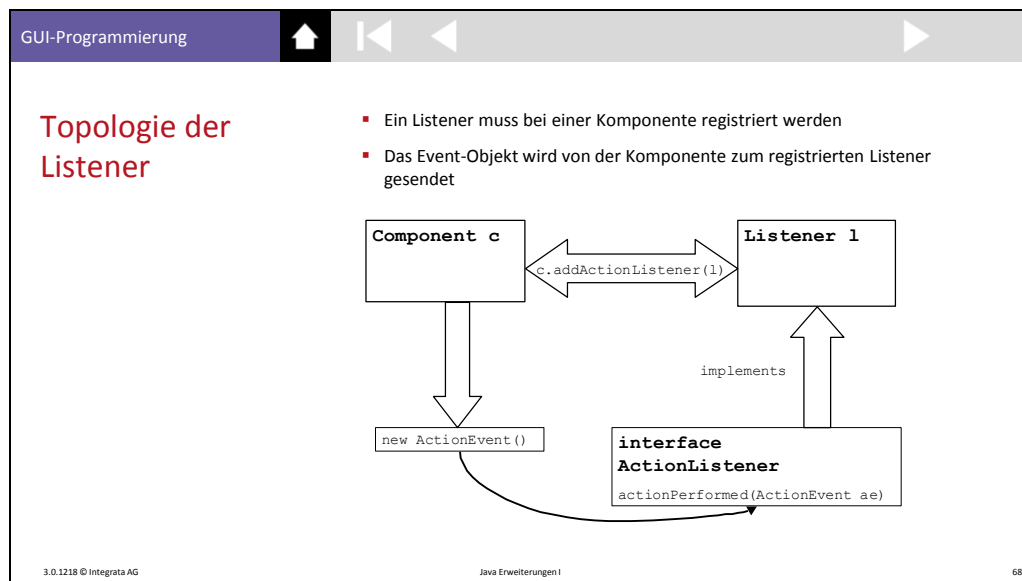


Abb. 5-11: Topologie der Listener

Registriert die Komponente `c` einen `ActionEvent` vom Betriebssystem (z. B. Mausklick), wird ein neues `ActionEvent`-Objekt erzeugt. Die im `ActionListener` implementierte Methode `actionPerformed(ActionEvent ae)` wird aufgerufen und erhält als Parameter den eben erzeugten Event.

Wichtig: Die Listener-Methoden (z.B. `actionPerformed()`) müssen nicht vom Programmierer aufgerufen werden. Die Komponente hält eine Liste aller Listener und ruft selbständig deren Methoden auf.

5.9 Designalternativen für Listener

GUI-Programmierung

Designalternativen für Listener

- Fenster-Komponente (View) und Listener (Controller):
 - zwei getrennte Klassen (View und Controller)
 - eine gemeinsame Klasse (Containerklasse)
 - Listener als innere Klasse
 - Listener als lokale Klasse
 - Listener als anonyme innere Klasse
 - Listener als Lambda-Ausdruck
 - Verwendung einer Adapter-Klasse

3.0.1218 © Integrata AG Java Erweiterungen I 69

Abb. 5-12: Designalternativen

5.9.1 Externe Klasse als Listener

Die Komponente wird mit einem getrennten Listener-Objekt registriert.

Beispiel:

```
import javax.swing.JButton;
import javax.swing.JFrame;

public class QuitButtonFrame extends JFrame {
    private JButton b = new JButton("Quit");
    public QuitButtonFrame() {
        add(b);
        b.addActionListener(new QuitButtonListener( ) );
        setSize(250, 250);
        setVisible(true);
    }
}
```

Das Listener-Objekt muss ein Listener-Interface implementieren.

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class QuitButtonListener implements ActionListener {
    public void actionPerformed (ActionEvent ae){
        System.exit(0);
    }
}
```

5.9.2 Container als Listener

Die Klasse selber implementiert ein Listener-Interface:

Beispiel:

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;

public class QuitButtonFrame extends JFrame
    implements ActionListener {

    private JButton b = new JButton("Button");

    public QuitButtonFrame() {
        add(b);
        b.addActionListener(this);
        setSize(250, 250);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent ae) {
        System.out.println( "Button gedrückt" );
    }
}
```

Der Behälter (die Klasse selber) muss eine Listener-Schnittstelle implementieren.

Falls der Behälter gleichzeitig für mehrere Komponenten Listener ist (z.B. für mehrer Buttons des Behälters), dann kann in der actionPerformed-Methode durch if-Anweisungen mit der Auswertung von `ae.getSource()` oder `ae.getActionCommand()` unterschieden werden, welche Komponente den Event ausgelöst hat, also z.B. welcher Button angeklickt wurde.

5.9.3 Innere Klasse als Listener

Die Komponente wird mit einem Listener-Objekt registriert. Das Listener-Objekt ist eine Instanz einer inneren Klasse.

Beispiel:

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;

public class QuitButtonFrame extends JFrame {
    private JButton b = new JButton("Button");

    public QuitButtonFrame() {
        add(b);
        b.addActionListener(new Listener());
        setSize(250, 250); setVisible(true);
    }

    private class Listener implements ActionListener {
        public void actionPerformed(ActionEvent ae) {
            System.out.println("Button gedrückt");
        }
    }
}
```

Für die Innere Klasse wird eine class-Datei erzeugt:

QuitButtonFrame\$Listener.class

5.9.4 Lokale Klasse als Listener

Die Komponente wird mit einem Listener-Objekt registriert. Das Listener-Objekt ist eine Instanz einer lokalen inneren Klasse.

Beispiel:

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;

public class QuitButtonFrame extends JFrame {
    private JButton b = new JButton("Button");

    public QuitButtonFrame() {
        add(b);
        class Listener implements ActionListener {
            public void actionPerformed(ActionEvent ae) {
                System.out.println("Button gedrückt");
            }
        }
        b.addActionListener(new Listener());
        setSize(250, 250); setVisible(true);
    }
}
```


Für die Lokale Klasse wird eine `class`-Datei erzeugt:

```
QuitButtonFrame$1Listener.class
```

5.9.5 Anonyme Klasse als Listener

Die Komponente wird mit einem Listener-Objekt registriert. Das Listener-Objekt ist eine Instanz einer anonymen inneren Klasse.

Anonyme innere Klassen haben keinen Namen und können keinen speziellen Konstruktor haben.

Beispiel:

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;

public class QuitButtonFrame extends JFrame {

    private JButton b = new JButton("Button");

    public QuitButtonFrame() {
        add(b);

        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                System.out.println("Button gedrückt");
            }
        });
        setSize(250, 250);
        setVisible(true);
    }
}
```

Für die Anonyme Klasse wird eine `class`-Datei erzeugt:

```
QuitButtonFrame$1.class
```

5.9.6 Lambda-Ausdruck als Listener

Beispiel:

```
import javax.swing.JButton;
import javax.swing.JFrame;

public class QuitButtonFrame extends JFrame {
    private JButton b = new JButton("Button");

    public QuitButtonFrame() {
        add(b);

        b.addActionListener(
            ( ae ) -> System.out.println("Button gedrückt") );

        setSize(250, 250);
        setVisible(true);
    }
}
```

Eine eigene `class`-Datei wird für den Lambda-Ausdruck **nicht** erzeugt.

5.9.7 Verwendung einer Adapter-Klasse

Statt der Implementierung eines Interfaces (alle Methoden müssen implementiert werden), wird die Subklasse eines Adapters erzeugt. Der Adapter ist eine Klasse, die alle Methoden des Interfaces als leere Methoden implementiert. In einer Subklasse des Adapters müssen dann nur die Methoden überschrieben werden, die nicht leer sein sollen.

```
class WindowKill extends WindowAdapter {
    public void windowClosing (WindowEvent e) {
        System.out.println("WindowKill");
        System.exit(0);
    }
}
```

Verwendung des Adapters:

```
frame.addWindowListener(new WindowKill());
```

5.10 Zusammenfassende Wertung

GUI-Programmierung

Zusammenfassende Wertung

- Externe Klasse
 - Vorteil: Die Listener können wieder verwendet werden. Mehrere Komponenten können sich am Listener registrieren
 - Nachteil: Der Zugriff auf die Komponenten, die im selben Container liegen, ist schwierig
- Container
 - Vorteil: Einfach zu realisieren, keine weitere Klasse nötig, Zugriff auf alle Attribute und Methoden des Containers
 - Nachteil: schlechtes Design, Delegation für mehrere Komponenten nur mit if-Abfragen
- Innere Klasse
 - Vorteil: klares Design, Zugriff auf Attribute und Methoden des Containers
 - Nachteil: aufwändig
- Anonyme Klasse
 - Vorteil: Listener ist direkt zugeordnet, Delegation, Objektmodell bleibt übersichtlich
 - Nachteil: Nicht wieder verwendbar (ist manchmal sogar gewollt)
- Lambda-Ausdruck
 - Vorteil: Listener ist direkt zugeordnet, nur Inhalt der Methode muss programmiert werden.
 - Nachteil: Nicht wieder verwendbar
- Adapter
 - Vorteil: Nur die benötigten Methoden des Adapters werden überschrieben
 - Nachteil: Geht nicht bei Einfachvererbung

3.0.1218 © Integrata AG

Java Erweiterungen I

70

Abb. 5-13: Zusammenfassende Wertung

6

Multithreading

6.1	Technische Einführung	6-3
6.2	Das Thread-API.....	6-5
6.2.1	Erzeugen eigener Threads	6-5
6.2.2	Start eines Threads.....	6-6
6.2.3	Subklasse von Thread	6-6
6.2.4	Implementierung von Runnable	6-7
6.2.6	Das Interface Runnable	6-8
6.2.7	Der Lebenszyklus eines Threads.....	6-10
6.2.8	Thread-Prioritäten	6-12
6.2.9	Daemon-Thread.....	6-13
6.2.10	Stoppen eines Threads.....	6-14
6.3	Timer und TimerTask	6-15

6 Multithreading

6.1 Technische Einführung

Die bisher vorgestellten Programme waren im Programmlauf recht einfach zu verstehen: Nach dem zentralen Einstieg durch den Aufruf der `main`-Methode ist der weitere Ablauf eindeutig dadurch nachzuvollziehen, dass eine Befehlszeile nach der anderen ausgeführt wird. Diese Art der Programmausführung stößt aber recht schnell an ihre Grenzen:

- In einer grafischen Benutzeroberfläche müssen einerseits Eingaben des Benutzers verarbeitet werden, andererseits sollen Animationen wie ein blinkender Cursor quasi gleichzeitig nebeneinander ablaufen.
- Auf Serverseite kann in größeren Rechnern die Abarbeitung eines Programms dadurch wesentlich beschleunigt werden, dass mehrerer Prozessoren gleichzeitig arbeiten.

Eine wesentliche Nebenbedingung beider Szenarien ist, dass ein gemeinsamer Speicherbereich zur Verfügung steht. Somit kann die Aufgabe nicht dadurch abgebildet werden, dass einfach weitere Prozesse gestartet werden. Genauso unpraktisch ist es, dieses Problem den Anwendungsprogrammierer lösen zu lassen, indem während der Abarbeitung zwischen den einzelnen Aufgaben programmgesteuert hin und her gewechselt wird.

Moderne Betriebssysteme wie Unix, Solaris oder Windows bieten deshalb einen Mechanismus an, innerhalb eines Prozesses eine nebenläufige Programmausführung mit Zugriff auf einen gemeinsam genutzten Speicherbereich zu ermöglichen. Das Betriebssystem kontrolliert, welche Abläufe wann ablaufen, und schaltet zwischen ihnen um. Diese nebenläufige Abarbeitung wird als "Thread" bezeichnet.

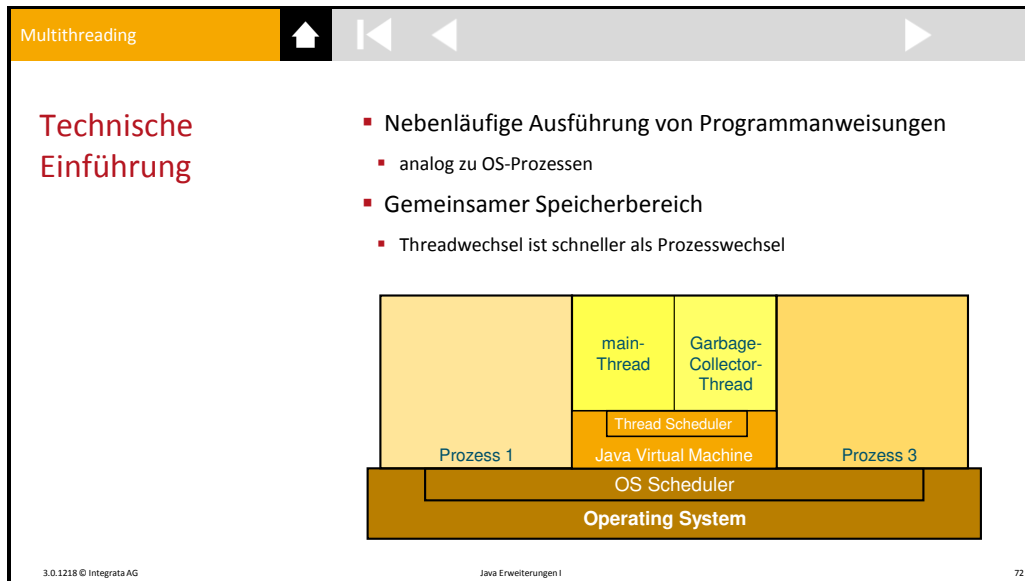


Abb. 6-1: Technische Einführung

Die Einführung von Threads als Hilfsmittel für den Programmierer ist zwar äußerst praktisch, führt aber in ihrer Anwendung zu einem ganzen Paket neuer Besonderheiten, die berücksichtigt werden müssen:

- Der nebenläufige Programmlauf kann nicht mehr so einfach linear verfolgt werden. Fehlersuche und -analyse werden dadurch komplizierter.
- Es muss ein Mechanismus zur Synchronisation von Threads eingeführt werden. Threads müssen eventuell warten, bis andere Threads eine Aufgabe erfüllt haben oder eine Bedingung eintritt wie z. B. "neue Daten eingetroffen".
- Threads haben Zugriff auf einen gemeinsamen Speicherbereich. Dies kann dann zu Problemen führen, wenn mehrere Threads eine Referenz auf das gleiche Objekt haben und sich gegenseitig bei der Methodenausführung unterbrechen.

Die Erstellung von Anwendungen mit mehreren Threads, so genannten multithreaded Programmen, erfordert folglich eine ganze Reihe von neuen Befehlen und Konstruktionen.

6.2 Das Thread-API

Java führt Threads in die Programmiersprache konsequent und elegant ein: Die Klasse `java.lang.Thread` enthält sämtliche Methoden, die zum Starten von nebenläufigen Anweisungen erforderlich sind.

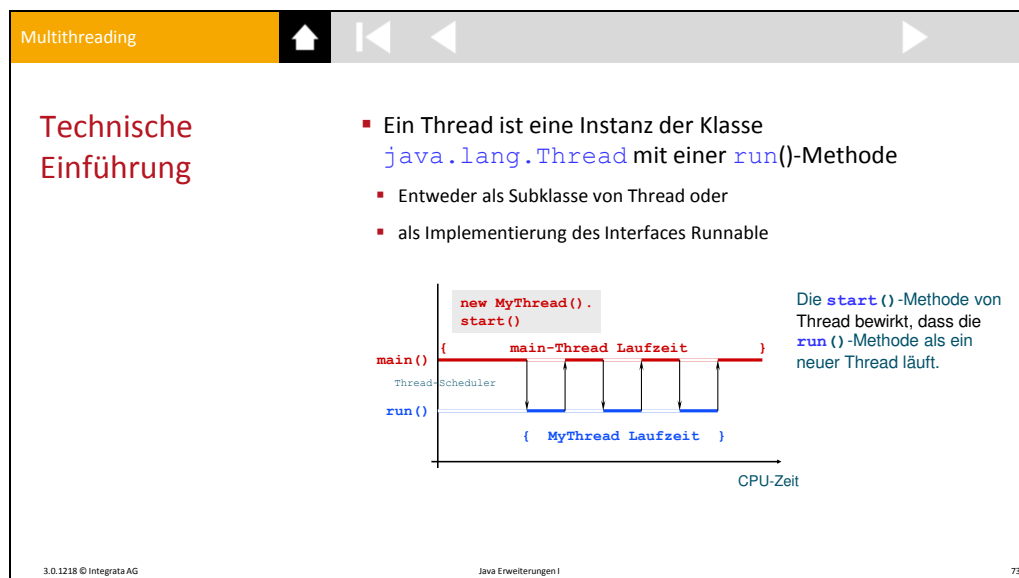


Abb. 6-2: Technische Einführung

6.2.1 Erzeugen eigener Threads

Die zur Laufzeit nebenläufig auszuführende Anweisungen werden definiert in der Methode `public void run()` innerhalb des Interfaces `java.lang.Runnable`.

Die Klasse `Thread` implementiert einerseits dieses Interface selbst und bietet andererseits einen Konstruktor an, der eine beliebige Referenz auf ein Objekt vom Typ `Runnable` entgegen nimmt.

Somit sind zwei Möglichkeiten vorhanden, einen eigenen Thread zu generieren:

- Überschreiben der `run()`-Methode der Klasse `Thread` in einer Subklasse
- Implementieren des Interfaces `java.lang.Runnable` in einer beliebigen Klasse und darin Definition der `run()`-Methode. Eine Instanz dieser Klasse kann dann als Parameter dem Konstruktor der Klasse `java.lang.Thread` übergeben werden.

6.2.2 Start eines Threads

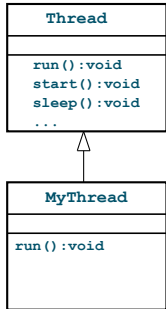
Zum Starten des Threads genügt es, die `start()`-Methode der Klasse `java.lang.Thread` zu verwenden. Die intern in der virtuellen Maschine ablaufenden Vorgänge zur Vorbereitung der nebenläufigen Abarbeitung der `run()`-Methoden werden durch `start()` automatisch durchgeführt.

6.2.3 Subklasse von Thread

Multithreading

⏮
⏪
⏩
⏭

**Eigenen Thread definieren:
Thread vererben**



```

classDiagram
    class Thread {
        run():void
        start():void
        sleep():void
        ...
    }
    class MyThread {
        run():void
    }
    Thread <|-- MyThread
        
```

- Programm ist Subklasse von Thread


```

public class MyThread extends Thread {
    public void run() {
        // do something
    }
}
                    
```
- Starten:


```

MyThread prog = new MyThread();
prog.start();
                    
```

3.0.1218 © Integrata AG
Java Erweiterungen I
74

Abb. 6-3: Thread

Beispiel:

Einfache `run()`-Methode, die von 0 bis 9 hoch zählt. Eine Erweiterung der Klasse `Thread`.

```

public class MyThread extends Thread {
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(Thread.currentThread( ) + ": " + i);
        }
    }
}
    
```

Verwendung in der Application-Klasse: Erzeugen der Threads durch parameterlosen Konstruktor der abgeleiteten Klasse `MyThread`:

```

public class ThreadApp {
    public static void main(String[] args) {
        MyThread lCounterThread = new MyThread();
        lCounterThread.start( );
        new MyThread( ).start( );
        System.out.println("Ende erreicht");
    }
}
    
```

6.2.4 Implementierung von Runnable

Multithreading
⏮ ⏪ ⏩ ⏭

**Eigenen Thread definieren:
Runnable implementieren**

```

<interface>
Runnable
run():void

MyThread
run():void
          
```

- Programm implementiert Runnable


```

public class MyThread implements Runnable {
    public void run() {
        // do something
    }
}
          
```
- Starten:


```

Thread t = new Thread( new MyThread( ) );
t.start();
          
```

3.0.1218 © Integrata AG
Java Erweiterungen I
75

Abb. 6-4: Runnable

Beispiel:

Implementierung des Runnable-Interfaces in einer eigenen Klasse.

```

public class Counter implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(Thread.currentThread() + ": " + i);
        }
    }
}
    
```

Verwendung in der Application-Klasse: Erzeugen der Threads durch Instanzierung der Counter-Klasse und Übergabe an den Konstruktor der Klasse Thread.

```

public class ThreadApp {
    public static void main(String[] args) {
        Thread lCounterThread = new Thread( new Counter( ) );
        lCounterThread.start( );
        new Thread( new Counter( ) ).start( );
        System.out.println("Ende erreicht");
    }
}
    
```

6.2.6 Das Interface Runnable

In `java.lang` ist neben der Klasse `Thread` das Interface `Runnable` definiert:

```
package java.lang;
public interface Runnable {
    public abstract void run();
}
```

Ein `Thread` kann einerseits gesteuert werden von der `run`-Methode einer von `Thread` abgeleiteten Klasse. Alternativ hierzu kann er aber auch gesteuert werden von der `run`-Methode irgendeiner Klasse, welche das Interface `Runnable` implementiert.

Man könnte etwa eine von `A` abgeleitete Klasse `B` schreiben, die eben dieses Interface implementiert:

```
class A { ... }
class B extends A implements Runnable {
    public void run() {
        ...
    }
}
```

Anstatt die Klasse `Thread` als Basisklasse für eine abgeleitete `Thread`-Klasse zu benutzen, könnte man dann die Klasse `Thread` direkt instanziiieren – indem dem Konstruktor von `Thread` ein `Runnable`-kompatibles Objekt übergeben wird:

```
B b = new B();
Thread t = new Thread(b);
t.start();
```

Der Vorteil dieses Verfahrens liegt darin, die eigentliche Funktionalität nicht in einer von `Thread` abgeleiteten Klasse definiert sein muss – sie kann in jeder beliebigen, das Interface `Runnable` implementierenden Klasse existieren.

LeftRunnable

```
public class LeftRunnable implements Runnable {
    public void run() {
        try { Thread.sleep(1000); }
        catch (InterruptedException ignored) { }
        System.out.println("A");
        try { Thread.sleep(1000); }
        catch (InterruptedException ignored) { }
        System.out.println("B");
        try { Thread.sleep(1000); }
        catch (InterruptedException ignored) { }
        System.out.println("C");
        System.out.println("end of LeftRunnable");
    }
}
```

RightRunnable

```
public class RightRunnable implements Runnable {  
    public void run() {  
        try { Thread.sleep(1800); } catch (...) { }  
        System.out.println("\t\tA");  
        try { Thread.sleep(1800); } catch (...) { }  
        System.out.println("\t\tB");  
        try { Thread.sleep(1800); } catch (...) { }  
        System.out.println("\t\tC");  
        System.out.println("\t\tend of RightRunnable");  
    }  
}
```

Application

```
public class Application {  
    public static void main(String [] args) {  
        Thread left = new Thread(new LeftRunnable());  
        Thread right = new Thread(new RightRunnable());  
  
        left.start ();  
        right.start ();  
  
        try { Thread.sleep(400); } catch (...) { }  
        System.out.println("\tA");  
        try { Thread.sleep(400); } catch (...) { }  
        System.out.println("\tB");  
        try { Thread.sleep(400); } catch (...) { }  
        System.out.println("\tC");  
  
        System.out.println("\tend of main");  
    }  
}
```

6.2.7 Der Lebenszyklus eines Threads

Die in den bisherigen Kapiteln dargestellten Abläufe lassen sich übersichtlich im Lebenszyklus eines Threads darstellen. Jeder Thread hat insgesamt vier mögliche Zustände:

Der Thread **lebt**. Dieser Zustand ist direkt nach der Instanzierung erreicht.

Der Thread **läuft**. Diese bedeutet, dass der Thread gestartet wurde und gerade aktiv Befehle innerhalb der `run()`-Methode abarbeitet. Ein Thread kann nur dann laufen, wenn vorher die `start()`-Methode aufgerufen wurde.

Der Thread **wartet**. Dieser Zustand kann ausschließlich von der virtuellen Maschine oder dem Betriebssystem erzeugt werden: Der Methodenaufrufstack des Threads sowie alle lokalen Variablen werden gesichert und die Ausführung unterbrochen.

Multithreading

⬆
⏮
⏪
⏩
⏭

Wechsel zwischen Threads

- Ein Thread kann sich selbst unterbrechen
 - Damit auch die anderen Threads "drankommen" (co-operative multithreading, non-preemptive)
- statische Methoden
 - `Thread.yield();` // Pause, damit andere Threads laufen können
 - `Thread.sleep(int milliseconds);` // Pause für eine definierte Zeit
 - `Thread.sleep(int milliseconds, int nanoSeconds);`
- Unterbrechung einer definierten Schlaf-Zeit ist möglich
 - `void interrupt()`
 - `static boolean interrupted()`
 - `boolean isInterrupted()`
- Überprüfung, ob der Thread noch lebt
 - `boolean isAlive()`

3.0.1218 © Integrata AG
Java Erweiterungen I
76

Abb. 6-5: Threadwechsel

Um den wartenden Zustand programmtechnisch zu erzwingen, bietet das Thread-API die Methoden

```
public static void yield()
public static void sleep(int) throws
java.lang.InterruptedException
public void join() throws
java.lang.InterruptedException
```

an.

`yield()` bedeutet, dass auf die Abarbeitung durch den Prozessor diesmal kurz verzichtet wird, `sleep(int)` lässt den Thread die übergebene Anzahl von Millisekunden schlafen. Ein Aufruf der Methode `join()` lässt den gerade aktiven Thread warten, bis der Thread, dessen `join()`-Methode aufgerufen wurde, sich beendet hat. Ein wartender Thread kann vom Betriebssystem automatisch **geweckt** werden. Der Programmierer kann dies durch den Aufruf der Methode

```
public void interrupt()
```

erreichen. Die Methoden `sleep()` und `join()` werden dann durch das Werfen der `InterruptedException` unterbrochen.

Der Thread ist **tot**. Nach der Abarbeitung der `run()`-Methode kann ein Thread nicht mehr gestartet werden.

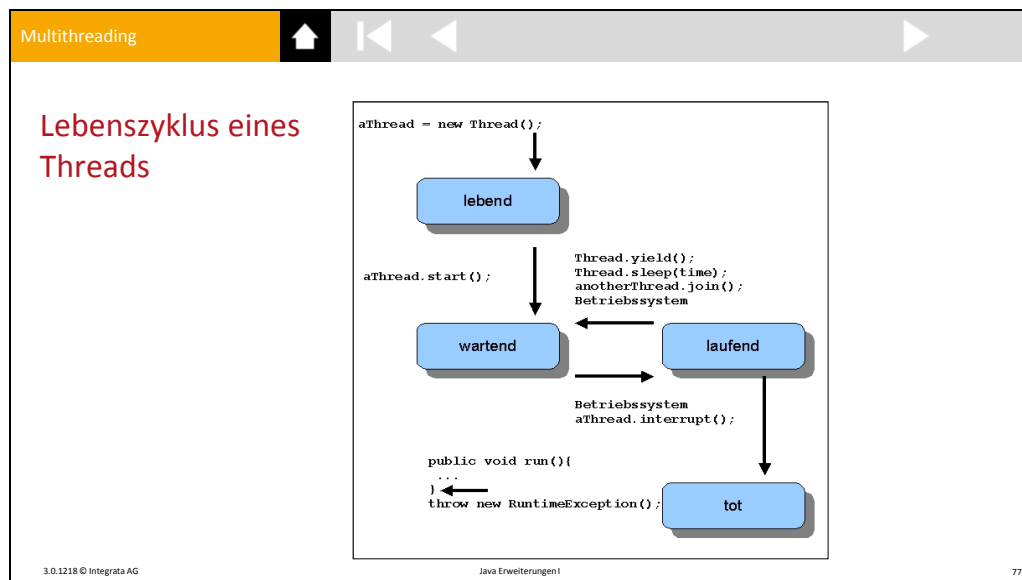


Abb. 6-6: Lebenszyklus

Ein `Thread` wird instanziiert und lebt damit. Durch den Aufruf der Methode `start()` wird der Thread in den wartenden Zustand versetzt. Das Betriebssystem steuert dann das Wechselspiel laufend-wartend. Der Programmierer kann dieses Wechselspiel durch die statischen Methoden `sleep(int)` und `yield()` sowie mit den Objektmethoden `join()` und `interrupt()` beeinflussen. Mit dem Erreichen der schließenden geschweiften Klammer der `run()`-Methodendefinition oder durch das Werfen einer `RuntimeException` wird der Thread sterben.

6.2.8 Thread-Prioritäten

Das Wechselspiel zwischen wartendem und laufendem Zustand kann noch global für Threads durch die Einstellung der Thread-Priorität eingestellt werden. Dies ist eine Property der Thread-Klasse und wird durch `get`- und `set`-Methoden zugreifbar gemacht.

The screenshot shows a presentation slide with a yellow header bar containing the text 'Multithreading' and navigation icons. The slide title is 'Threadprioritäten' in red. It contains a bulleted list of points about thread priorities in Java. The footer of the slide includes the version '3.0.1218 © Integrata AG', the text 'Java Erweiterungen I', and the page number '78'.

Threadprioritäten

- Die Häufigkeit und Länge, mit der ein Thread im laufenden Zustand verweilt, ist abhängig von seiner Priorität
- `setPriority(int pPriority)`
- `int getPriority`
- `Thread.MIN_PRIORITY`, `Thread.NORM_PRIORITY`, `Thread.MAX_PRIORITY`
- Ein Thread wird erst dann ausgeführt, wenn alle Threads mit höherer Priorität momentan nicht rechenwillig sind.
- Ablauf-Synchronisation durch Prioritäten ist nicht sinnvoll.
- Bevorzugung bestimmter Operationen kann man durch Variieren der Prioritäten der beteiligten Threads schaffen.
- Ausschlaggebend ist hierbei nicht die absolute Höhe, sondern die Abstufung zwischen den beteiligten Threads.

3.0.1218 © Integrata AG Java Erweiterungen I 78

Abb. 6-7: Prioritäten

```
public void setPriority(int pPriority)
```

```
public int getPriority();
```

`pPriority` muss sich im zulässigen Wertebereich zwischen `Thread.MIN_PRIORITY` und `Thread.MAX_PRIORITY` befinden. Die normale Priorität eines Threads ist `Thread.NORM_PRIORITY`.

6.2.9 Daemon-Thread

The screenshot shows a presentation slide with a yellow header bar containing the word 'Multithreading' and navigation icons. The slide title is 'User- und Daemon-Threads'. The content is organized into two main sections: 'Es gibt User- und Deamon-Threads' and 'Thread-Dämonen sind'. The first section lists that the `main`-Thread is a User-Thread, threads started from `main` are User-Threads by default, and the Garbage-Collector is a Daemon-Thread. The second section explains that Daemon-Threads have low priority and the VM won't keep them alive. It lists conditions for VM termination: only Daemons exist, or no User-Threads exist (checked with `isDaemon == false`), or no User-Threads are running. It also shows the methods `setDaemon(boolean)` and `boolean isDaemon()`, and notes they must be set before the thread starts.

Multithreading

User- und Daemon-Threads

- Es gibt User- und Deamon-Threads
 - Der `main`-Thread ist ein User-Thread
 - Vom `main`-Thread gestarteten Threads laufen defaultmäßig als User-Thread
 - Der Garbage-Collector läuft als Daemon-Thread (Hintergrund-Thread)
- Thread-Dämonen sind
 - Threads mit geringer Priorität, die die virtuelle Maschine nicht am „Leben“ erhalten
 - Falls beim Ende des main-Threads nur noch 'Daemons' existieren, wird die VM (mit allen Threads) beendet.
 - Falls noch andere sog. Anwendungs- bzw. User-Threads (`isDaemon == false`) existieren, werden diese nicht beendet.
 - (Die JVM wird beendet, wenn kein User-Thread mehr läuft.)
 - `setDaemon(boolean)` und `boolean isDaemon()`
 - Setzen vor dem Start des Threads

3.0.1218 © Integrata AG Java Erweiterungen I 79

Abb. 6-8: Daemon

Neben den bisher vorgestellten so genannten User-Threads existieren noch die Daemon-Threads. Daemon-Threads halten die virtuelle Maschine nicht am Leben: Das Kriterium für die virtuelle Maschine, sich selbst zu beenden lautet: Es laufen nur noch Thread-Daemonen.

```
public void setDaemon(boolean)
public boolean isDaemon();
```

Das Einstellen des Daemon-Charakters muss vor dem Start des Threads erfolgen und kann nachträglich nicht mehr geändert werden.

6.2.10 Stoppen eines Threads

Das Beenden eines Threads ist in Java exakt definiert: Die `run()`-Methode muss terminiert werden, entweder durch das Erreichen der endenden geschweiften Klammer bzw. eines `return`, oder durch Werfen einer `java.lang.RuntimeException`. Der Thread-Entwickler ist nun aber dafür verantwortlich, bei Bedarf ein Beenden des Threads kontrolliert von außen anstoßen zu lassen. Dazu müssen geeignete Methodendefinitionen zur Verfügung gestellt werden. Das Thread-API enthält zwar eine Methode namens `stop()`. Diese darf jedoch nicht verwendet werden, weil sie nie richtig funktioniert hat. Dies wurde erkannt, und seitdem ist diese Methode als "deprecated" gekennzeichnet.

The screenshot shows a presentation slide with a yellow header bar containing the text 'Multithreading'. The slide title is 'Stoppen eines Threads' in red. The content consists of a bulleted list of instructions on how to stop a thread. At the bottom, there is a footer with '3.0.1218 © Integrata AG', 'Java Erweiterungen I', and the page number '80'.

- Nur möglich durch
 - Normale Terminierung der `run()`-Methode
 - werfen einer `java.lang.RuntimeException`
- Stoppen durch Kontroll-Variable innerhalb der `run()`-Methode
 - Dies muss selber programmiert werden!
 - Setzen von „Außen“ durch setter-Methoden vorsehen!
 - Die sofortige Terminierung mit Hilfe der `stop()`-Methode darf **nicht** verwendet werden!

Abb. 6-9: Stoppen eines Threads

Um einen laufenden Thread zu unterbrechen, kann aber natürlich in der `run()`-Methode z. B. eine Variable geprüft werden:

```
public class MyThread extends Thread {
    public boolean laeuft = true;
    public void run() {
        if ( laeuft ) {
            for (int i = 0; i < 10; i++) {
                System.out.println(Thread.currentThread( ) + ": " + i);
            }
        }
    }
}
```

Somit besteht keine Möglichkeit, einen `Thread` zu sofort zu unterbrechen, sondern die `run()`-Methode läuft bis zur nächsten Prüfung der Bedingung. Damit wird sichergestellt, dass der in der while-Schleife festgelegte Arbeitsschritt noch komplett durchgeführt und nicht in einem undefinierten Zwischenstand abgebrochen wird.

6.3 Timer und TimerTask

Seit JDK 1.3 sind diese beiden nützliche Hilfsklassen eingeführt worden, mit denen das Arbeiten mit periodischen oder einmalig auszuführenden Aufgaben in einem nebenläufigen Thread sehr einfach wird.

`java.util.TimerTask` ist eine abstrakte Klasse, die eine in regelmäßigen Abständen zu verrichtende Aufgabe abbildet. Sie implementiert das `java.lang.Runnable`-Interface, die `run()`-Methode wird in konkreten Subklassen implementiert. Zur Verwaltung von `TimerTask`-Objekten wird der `java.util.Timer` verwendet. Dieser verwaltet `TimerTask`-Threads komfortabel. Die Konzeption der `java.util.Timer`-Klasse ist darauf ausgerichtet, eine große Menge an Tasks kontrollieren zu können.

Multithreading

⬆
⬅
➡

Die Klassen `Timer` und `TimerTask`

- Automatisiertes periodisches Starten eines Prozesses durch `java.util.Timer`
- Definition des Prozesses in einer Subklasse von `java.util.TimerTask`
 - Erfordert Implementierung der `run()`-Methode
- `Timer` ist dazu ausgelegt, sehr viele `TimerTask`-Objekte robust verwalten zu können

java.util.Timer
 -queue TaskQueue=new TaskQueue()
 -thread TimerThread=new TimerThread(queue)
 -threadReaper Object=new Object() (protected void finalize() throw
 +Timer()
 +Timer(boolean daemon)
 +scheduleTask TimerTask, delay long void
 +scheduleTask TimerTask, time Date void
 +scheduleTask TimerTask, delay long, period long void
 +scheduleTask TimerTask, firstTime Date, period long void
 +scheduleAtFixedRate Task TimerTask, delay long, period long void
 +scheduleAtFixedRate Task TimerTask, firstTime Date, period long void
 +scheduleTask TimerTask, time long, period long void
 +cancel() void

java.lang.Object
 java.lang.Runnable
java.util.TimerTask
 lock java.lang.Object
 state int
 VORGIN int
 SCHEDULED int
 EXECUTED int
 CANCELLED int
 nextExecutionTime long
 period long
 +TimerTask()
 +run() void
 +cancel() boolean
 +scheduleExecutionTime() long

3.0.1218 © Integrata AG
Java Erweiterungen I
81

Abb. 6-10: Timer und TimerTask

Der `Timer` verwaltet `TimerTasks`. Ein konkreter `TimerTask` muss die `run()`-Methode implementieren:

```
import java.util.*;

/**
 * Starterklasse für Timer/TimerTask
 */
public class TimerDemo {
    public static void main(String[] args) {

        Timer lTimer = new Timer();

        TimerTask lTask = new TimerTask() {
            public void run() {
                System.out.println("Message from Timer at "
                    + (new Date()).toString());
            }
        };
        lTimer.schedule(lTask, 0, 5000);
    }
}
```

3.0.1218 © Integrata AG

Java Erweiterungen I

82

Abb. 6-11: Beispiel Timer

Ein `Timer` nimmt eine anonyme Subklasse von `TimerTask` auf und führt dessen `run()`-Methode ohne erstmalige Verzögerung alle 5 Sekunden aus.

Mit `Timer` und `Timertasks` können Threads gestartet werden, die im Wesentlichen unabhängig voneinander laufen. Wenn eine Interaktion zwischen den Threads oder eine genauere Ablaufsteuerung für das Starten und Stoppen von Threads benötigt wird, muss man dies selber programmieren.

7

Java und Datenbanken

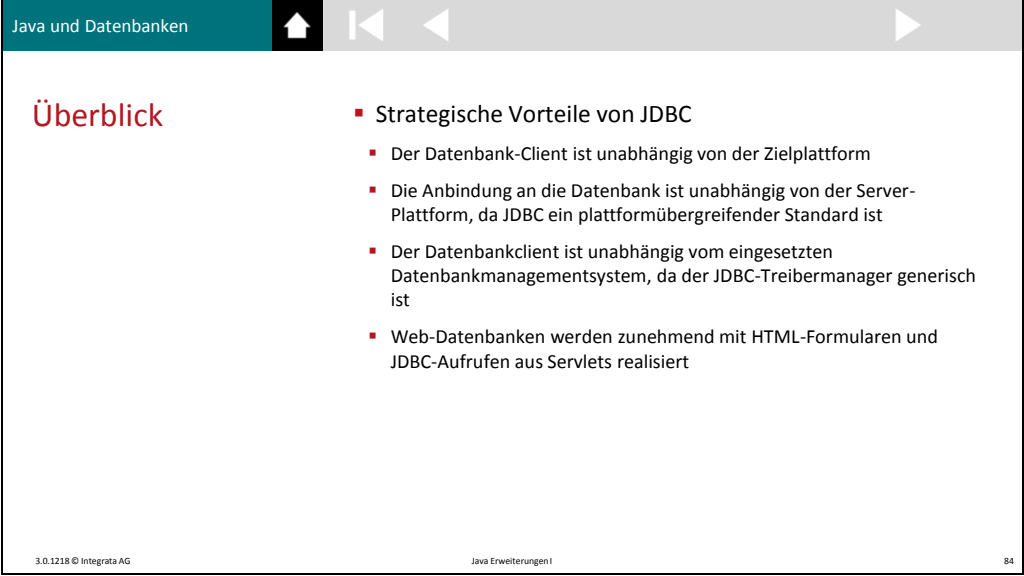
7.1	Überblick	7-3
7.2	Zwei- und Drei-Schicht-Modell.....	7-4
7.3	Das JDBC-API.....	7-5
7.4	Treibertypen	7-6
7.4.1	JDBC-ODBC-Brückentreiber	7-7
7.4.2	JDBC-DBMS API-Treiber.....	7-8
7.4.3	JDBC-Middleware-Treiber	7-8
7.4.4	JDBC-DB-Treiber.....	7-9
7.4.5	Treiber laden.....	7-9
7.5	Die Verbindung.....	7-11
7.5.1	Datenbank-URL	7-11
7.5.2	Benutzer und Kennwort	7-12
7.5.3	Verbindung öffnen.....	7-13
7.5.4	Verbindung schließen	7-14
7.5.5	Verwendung der Connection	7-15
7.6	SQL-Befehle senden	7-16
7.6.1	JDBC Statements	7-16
7.6.2	PreparedStatement und CallableStatement.....	7-18
7.7	Auswertung der Ergebnisse.....	7-20
7.7.1	ResultSet	7-20
7.7.2	Ergebnistabelle schließen.....	7-21

7.8	Zusammenfassung Datenbankzugriff	7-22
7.8.1	Komplettes Beispiel	7-22

7 Java und Datenbanken

7.1 Überblick

JDBC kann in beliebigen Java-Programmen eingesetzt werden, die einen Datenbankzugriff benötigen. Voraussetzung ist natürlich, dass die Datenbank JDBC-fähig ist.



The screenshot shows a presentation slide with a title bar 'Java und Datenbanken' and navigation icons. The main content is titled 'Überblick' in red. It lists four strategic advantages of JDBC:

- Strategische Vorteile von JDBC
 - Der Datenbank-Client ist unabhängig von der Zielplattform
 - Die Anbindung an die Datenbank ist unabhängig von der Server-Plattform, da JDBC ein plattformübergreifender Standard ist
 - Der Datenbankclient ist unabhängig vom eingesetzten Datenbankmanagementsystem, da der JDBC-Treibermanager generisch ist
 - Web-Datenbanken werden zunehmend mit HTML-Formularen und JDBC-Aufrufen aus Servlets realisiert

At the bottom, there is a footer with '3.0.1218 © Integrata AG', 'Java Erweiterungen I', and the page number '84'.

Abb. 7-1: Überblick

Programme, die auf eine Datenbank zugreifen, werden häufig in zwei Kategorien unterteilt:

- **Zwei-Schicht-Modell**, besteht aus einer Präsentationsschicht, in der meist auch die Geschäftslogik eingebettet ist und einer Datenhaltungsschicht, die der Datenbank entspricht. Das Zwei-Schicht-Modell wird auch als Two-Tier-Model bezeichnet.
- **Drei-Schicht-Modell**, besteht aus einer Präsentationsschicht, die üblicherweise auf dem Client läuft, einer Business-Schicht, die auf einem Server läuft und die Geschäftslogik definiert, und einer Datenhaltungsschicht, die der Datenbank entspricht. Das Drei-Schicht-Modell wird auch als Three-Tier-Model bezeichnet.

Im Zwei-Schicht-Modell kommuniziert der Java-Client direkt mit dem DBMS. Beim Drei-Schicht-Modell läuft die Kommunikation indirekt ab. Der Client kommuniziert mit der Business-Schicht, die erst mit dem DBMS in Verbindung tritt.

7.2 Zwei- und Drei-Schicht-Modell

Die heute übliche Vorgehensweise für geschäftskritische Anwendungen ist das Drei-Schicht-Modell.

In der folgenden Grafik ist der Aufbau des Zwei-Schicht- und des Drei-Schicht-Modells noch einmal gut zu sehen:

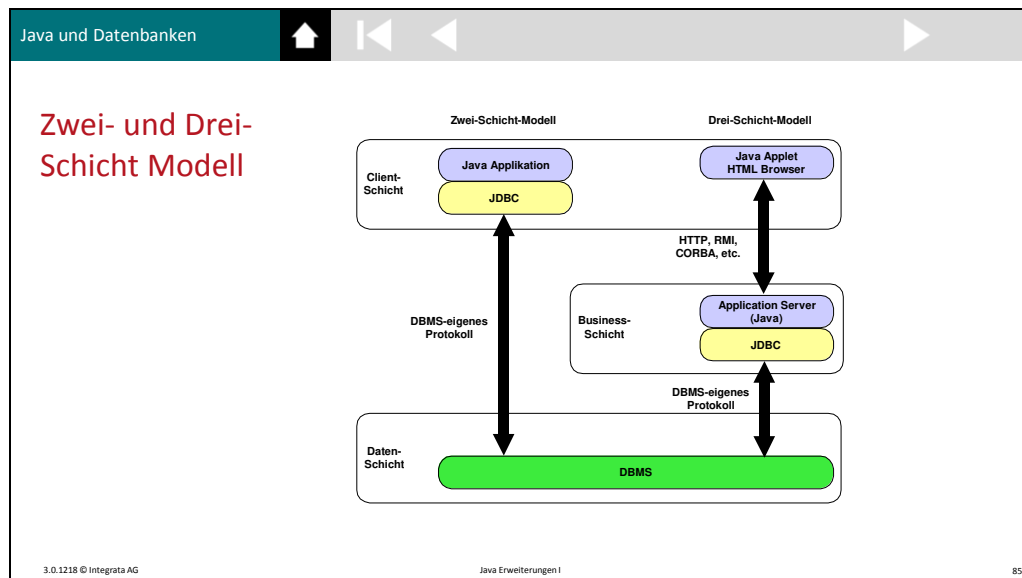


Abb. 7-2: Zwei- und Drei-Schicht Modell

7.3 Das JDBC-API

The screenshot shows a presentation slide with a teal header bar containing the text 'Java und Datenbanken' and navigation icons. The slide title 'JDBC-API' is in red. It contains a bulleted list of facts about the JDBC API. At the bottom, there is a footer with the version '3.0.1218 © Integrata AG', the text 'Java Erweiterungen I', and the page number '86'.

JDBC-API

- Das JDBC API ist eine Sammlung von Klassen und Interfaces
- Sie definiert also die Art und Weise, wie der Datenbankzugriff vom Java-Programm aus auszusehen hat
- Die eigentliche Implementierung muss jemand anderer liefern und zwar jemand, der Wissen über das DBMS hat
- Die Implementierung der JDBC-Schnittstellen nennen wir Treiber
 - Um mit Java auf eine Datenbank zugreifen zu können, brauchen wir einen JDBC-Treiber
 - Ein JDBC-Treiber ist natürlich nichts anderes als eine Klasse
 - Der JDBC-Treiber implementiert die im API definierten Interfaces
 - Der JDBC-Treiber wird meist vom Hersteller des DBMS geliefert.

3.0.1218 © Integrata AG Java Erweiterungen I 86

Abb. 7-3: JDBC-API

Wie schon erwähnt, wird es nicht möglich sein, mit JDBC auf eine x-beliebige Datenbank zuzugreifen. Die JDBC-API ist eine Sammlung von Klassen und in viel größerem Umfang von Interfaces im Paket `java.sql`. Sie definiert also nur die Art und Weise, wie der Datenbankzugriff vom Java-Programm aus auszusehen hat. Die eigentliche Implementierung muss jemand liefern, der Wissen über das DBMS hat.

Daher erfolgen die meisten Implementierungen entweder durch den DB-Hersteller selber oder sie wird von Drittanbietern angeboten, die das DBMS gut kennen.

Die Implementierung der JDBC-Schnittstellen geschieht über Treiber. Um mit Java auf eine Datenbank zugreifen zu können, brauchen wir einen JDBC-Treiber. Ein JDBC-Treiber ist natürlich nichts anderes als eine Klasse.

7.4 Treibertypen

Es gibt eine ganze Reihe von Treibern. Oracle hat die Treiber in vier Gruppen eingeteilt:

In der folgenden Grafik sind die vier Treibertypen schematisch dargestellt:

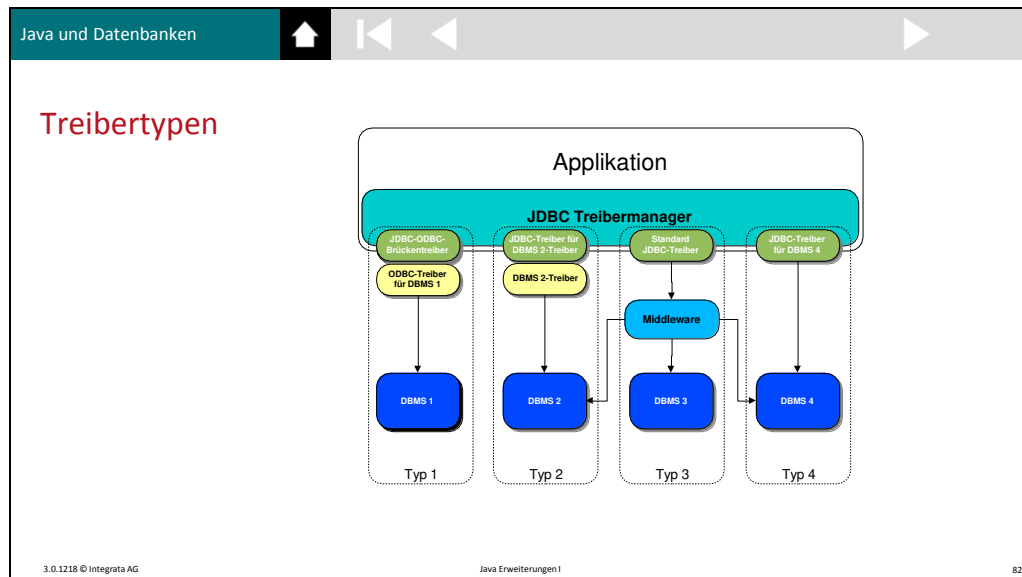


Abb. 7-4: Treibertypen

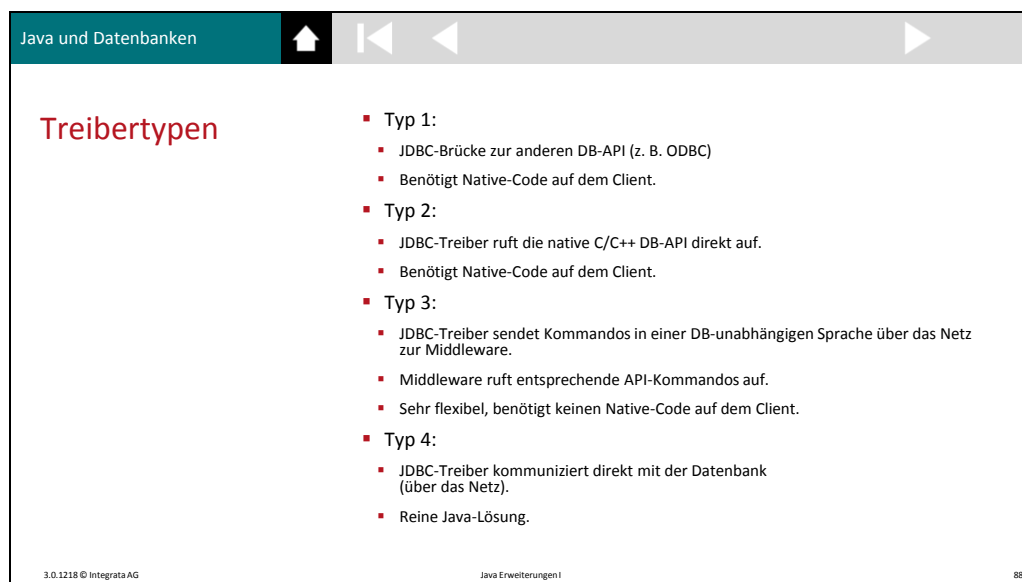


Abb. 7-5: Treibertypen

7.4.1 JDBC-ODBC-Brückentreiber

Um die Attraktivität von JDBC zu steigern und den Umstieg von ODBC auf JDBC zu erleichtern, enthält die Klassenbibliothek bis JDK 7 den JDBC-ODBC-Brückentreiber. Dieser Treiber übersetzt alle JDBC-Befehle in ODBC-Befehle, die dann über einen installierten und konfigurierten ODBC-Treiber auf die ODBC-fähige Datenbank zugreifen können. Der Nachteil dieses Treibers besteht darin, dass er relativ langsam, nicht netzwerkfähig und im Wesentlichen beschränkt auf die Windows-Plattform ist. Weiterhin sind auf Clientseite umfangreiche Konfigurationseinstellungen mit Hilfe des ODBC-Administrators durchzuführen.

The screenshot shows a presentation slide with a teal header bar containing the text 'Java und Datenbanken' and navigation icons. The slide content is as follows:

Typ 1:
JDBC-ODBC-
Brückentreiber

- Alle JDBC-Befehle werden in ODBC-Befehle übersetzt. Diese werden dann an eine ODBC-fähige Datenbank geschickt
- Vorteile
 - Alle ODBC-Datenquellen sind damit verwendbar
 - Wird kostenlos mit Java SE (bis JDK 7) mitgeliefert
- Nachteile
 - Nicht Internetfähig
 - Langsam (JDBC - ODBC - DBMS API - DB)
 - Installation auf dem Client notwendig
 - Keine reine Java-Lösung
- Beispiel
 - `sun.jdbc.odbc.JdbcOdbcDriver`

3.0.1218 © Integrata AG Java Erweiterungen I 89

Abb. 7-6: Typ 1

7.4.2 JDBC-DBMS API-Treiber

The slide is titled 'Typ 2: JDBC-DBMS API-Treiber' in red text. It contains a bulleted list of characteristics and examples. The slide has a header bar with 'Java und Datenbanken' and navigation icons. The footer contains '3.0.1218 © Integrata AG', 'Java Erweiterungen I', and the page number '90'.

**Typ 2:
JDBC-DBMS API-
Treiber**

- Es gibt eine DBMS API in einer anderen Programmiersprache, z.B. C/C++. Alle JDBC-Befehle werden in die Befehle der DBMS API übersetzt, die dann an die Datenbank geschickt werden
- Vorteile
 - Direkter Zugriff auf DBMS API ohne Umweg über ODBC.
 - Schnell
- Nachteile
 - Nicht Internetfähig
 - Installation auf dem Client notwendig
 - Keine reine Java-Lösung
- Beispiel
 - Oracle Thick Driver

3.0.1218 © Integrata AG Java Erweiterungen I 90

Abb. 7-7: Typ 2

7.4.3 JDBC-Middleware-Treiber

The slide is titled 'Typ 3: JDBC-Middleware-Treiber' in red text. It contains a bulleted list of characteristics and examples. The slide has a header bar with 'Java und Datenbanken' and navigation icons. The footer contains '3.0.1218 © Integrata AG', 'Java Erweiterungen I', and the page number '91'.

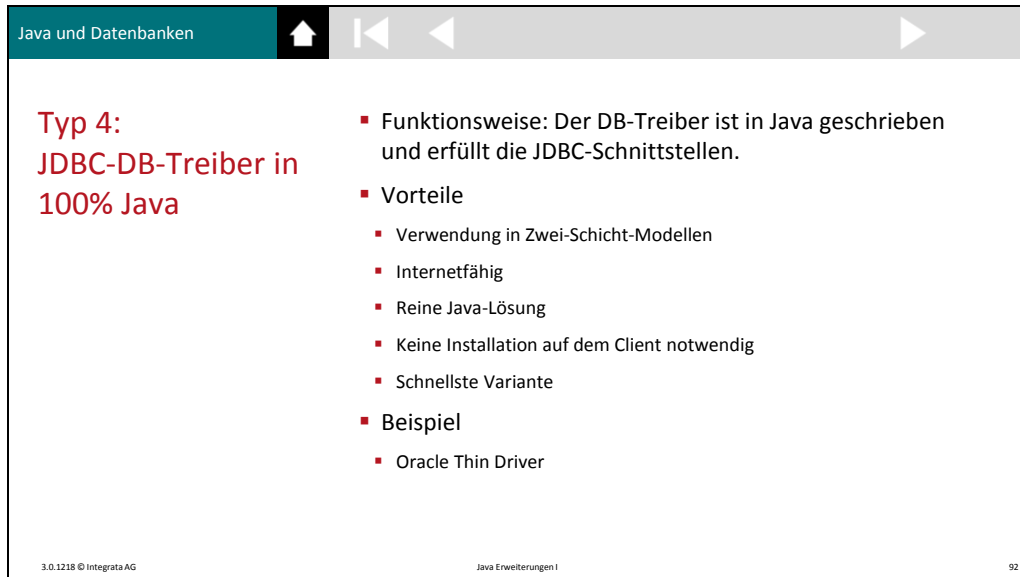
**Typ 3:
JDBC-Middleware-
Treiber**

- Der Treiber schickt die JDBC-Befehle nicht direkt an die Datenbank, sondern an eine Middleware. Diese verarbeitet die Befehle und leitet sie an die Datenbank weiter.
- Vorteile
 - Flexibel, da nicht auf ein DBMS spezialisiert. DB-Zugriff wird von der Middleware übernommen.
 - Verwendung in Drei-Schicht-Modellen
 - Internetfähig
 - Reine Java-Lösung
 - Keine Installation auf dem Client notwendig
- Nachteile
 - Langsamer als direkter DB-Zugriff
- Beispiel
 - COM.cloudscape.core.RmiJdbcDriver

3.0.1218 © Integrata AG Java Erweiterungen I 91

Abb. 7-8: Typ 3

7.4.4 JDBC-DB-Treiber



Java und Datenbanken

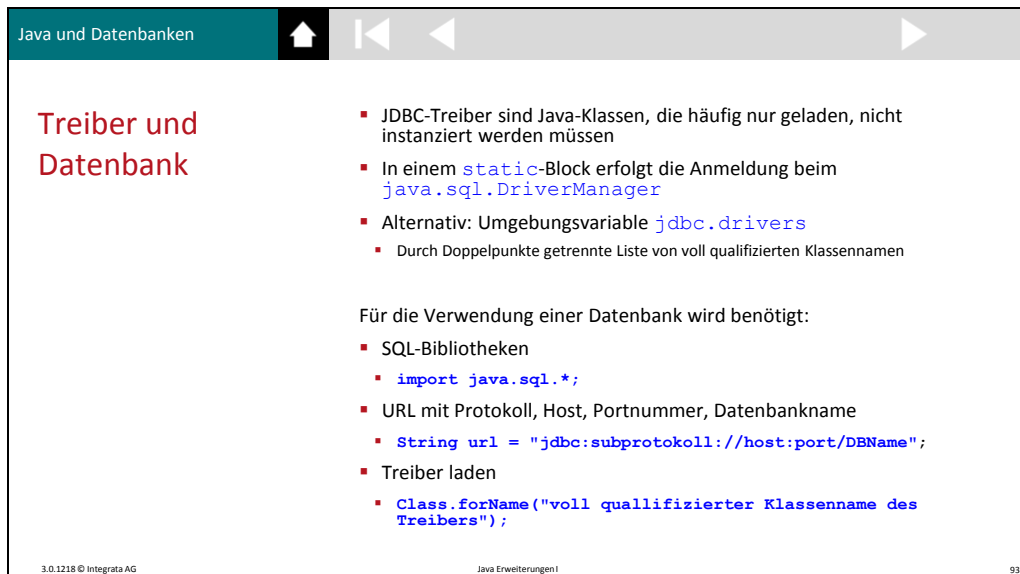
**Typ 4:
JDBC-DB-Treiber in
100% Java**

- Funktionsweise: Der DB-Treiber ist in Java geschrieben und erfüllt die JDBC-Schnittstellen.
- Vorteile
 - Verwendung in Zwei-Schicht-Modellen
 - Internetfähig
 - Reine Java-Lösung
 - Keine Installation auf dem Client notwendig
 - Schnellste Variante
- Beispiel
 - Oracle Thin Driver

3.0.1218 © Integrata AG Java Erweiterungen I 92

Abb. 7-9: Typ 4

7.4.5 Treiber laden



Java und Datenbanken

**Treiber und
Datenbank**

- JDBC-Treiber sind Java-Klassen, die häufig nur geladen, nicht instanziiert werden müssen
- In einem `static`-Block erfolgt die Anmeldung beim `java.sql.DriverManager`
- Alternativ: Umgebungsvariable `jdbc.drivers`
 - Durch Doppelpunkte getrennte Liste von voll qualifizierten Klassennamen

Für die Verwendung einer Datenbank wird benötigt:

- SQL-Bibliotheken
 - `import java.sql.*;`
- URL mit Protokoll, Host, Portnummer, Datenbankname
 - `String url = "jdbc:subprotokoll://host:port/DBName";`
- Treiber laden
 - `Class.forName("voll qualifizierter Klassenname des Treibers");`

3.0.1218 © Integrata AG Java Erweiterungen I 93

Abb. 7-10: Treiber

Der Treiber ist eine Java-Klasse, die das Interface `java.sql.Driver` implementieren muss. Theoretisch ist es möglich, direkt eine Instanz der Treiberklasse anzulegen. Das ist allerdings nicht der übliche Weg, um einen JDBC-Treiber zu laden.

Vielmehr bedienen wir uns einer Hilfsklasse, mit der wir einen oder mehrere Treiber verwalten können. Diese Klasse sucht dann aus den geladenen Treibern den geeigneten Treiber für die jeweilige Datenbank heraus. Die Klasse heißt `java.sql.DriverManager`.

Die Registrierung bei diesem `DriverManager` erfolgt in der Regel beim Einladen der Klasse in einem `static`-Block.

Die Treiber-Klasse wird durch den Befehl

```
Class.forName("treiberName");
```

geladen. Für "treiberName" muss der voll qualifizierende Klassenname des Treibers eingegeben werden, z. B.

```
COM.cloudscape.core.RmiJdbcDriver
```

Wenn wir eine `ClassNotFoundException` beim Laden der Klasse erhalten, dann wurde die Klasse nicht im `CLASSPATH` gefunden. Wir müssen also sicherstellen, dass die Bytecode-Dateien zur Laufzeit gefunden werden.

Beispiel:

```
try {
    Class.forName("COM.cloudscape.core.RmiJdbcDriver");
}

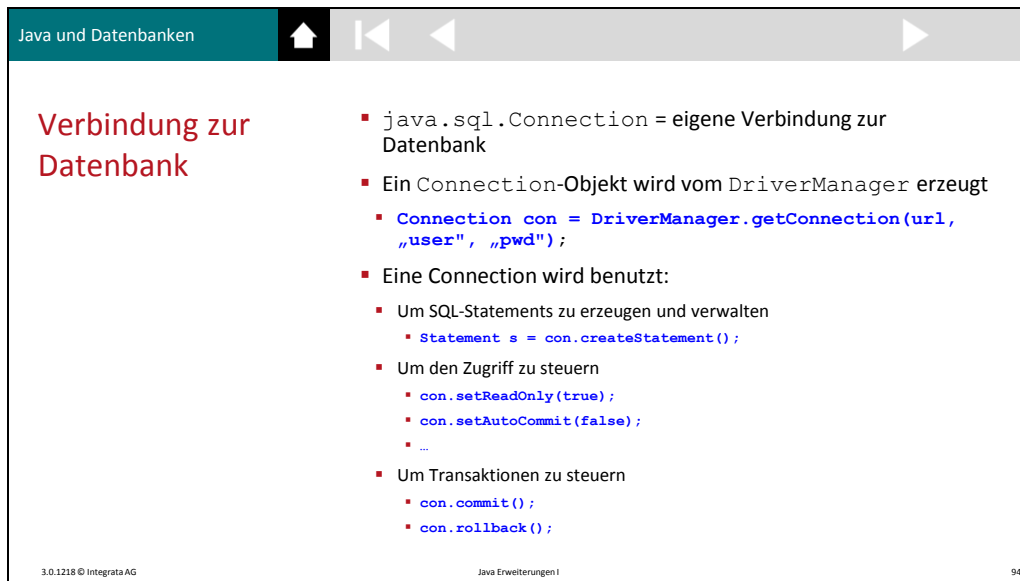
catch (ClassNotFoundException ex) {
    // Bytecodes nicht im CLASSPATH oder Klassenname falsch geschrieben
}
```

Eine Alternative zum expliziten Laden des Treibers ist die Verwendung der Umgebungsvariable `jdbc.drivers`. Beim Start der virtuellen Maschine kann eine durch Doppelpunkt getrennte Liste von Treibern übergeben werden, die beim ersten Zugriff auf den `java.sql.DriverManager` gelesen wird, z. B.

```
java -Djdbc.drivers = COM.cloudscape.core.RmiJdbcDriver:
                    sun.jdbc.odbc.JdbcOdbcDriver
```

Das explizite Laden eines Treibers ist seit Version Java 6 nicht mehr nötig. Der entsprechende Treiber wird von der JRE automatisch bei jedem Programmstart geladen.

7.5 Die Verbindung



The screenshot shows a presentation slide with a teal header bar containing the text 'Java und Datenbanken' and navigation icons. The slide content is as follows:

Verbindung zur Datenbank

- `java.sql.Connection` = eigene Verbindung zur Datenbank
- Ein `Connection`-Objekt wird vom `DriverManager` erzeugt
- `Connection con = DriverManager.getConnection(url, „user“, „pwd“);`
- Eine `Connection` wird benutzt:
 - Um SQL-Statements zu erzeugen und verwalten
 - `Statement s = con.createStatement();`
 - Um den Zugriff zu steuern
 - `con.setReadOnly(true);`
 - `con.setAutoCommit(false);`
 - ...
 - Um Transaktionen zu steuern
 - `con.commit();`
 - `con.rollback();`

At the bottom of the slide, there is a footer with '3.0.1218 © Integrata AG' on the left, 'Java Erweiterungen I' in the center, and '94' on the right.

Abb. 7-11: Verbindung

Wenn wir den richtigen Treiber geladen haben, können wir versuchen, diesen zu nutzen, um eine Verbindung zur Datenbank aufzubauen. Dazu brauchen wir folgende Informationen:

- **Datenbank-URL**, eine URL, die uns den Ort der Datenbank liefert.
- **User**, einen Benutzer, falls die Datenbank mit Benutzern und Kennwörtern arbeitet.
- **Kennwort**, das Kennwort des Benutzers, falls Benutzer eingerichtet worden sind.

7.5.1 Datenbank-URL

Wie in vielen Bereichen von Java, werden Lokationen durch URLs definiert. Eine Datenbank-URL setzt sich aus folgenden Daten zusammen:

- **Protokoll**, definiert das Protokoll, mit dem auf die Datenbank zugegriffen wird.
- **Server**, Host-Adresse.
- **Port**, Portnummer.
- **Datei-/Datenbankname**, Datei, in der sich die Datenbank befindet oder symbolischer Name der Datenbank.
- **Parameter**

Nachfolgend werden die einzelnen Bestandteile genauer betrachtet.

7.5.1.1 Protokoll

Das Protokoll besteht aus zwei Teilen,

- dem Hauptprotokoll, das immer `jdbc:` heißt und
- einem Subprotokoll, das sich nach der verwendeten Datenbank richtet. Es trägt daher häufig den Namen der Datenbank, des Datenbank-Herstellers oder des Treibers. Beispiele sind `odbc:`, `cloudscape:rmi:`, `borland:dsremote:` etc.

Das Protokoll sieht also immer so aus:

```
jdbc:subprotokoll:
```

7.5.1.2 Server und Port

An das Protokoll kann sich eine Server- und eine Portbezeichnung anschließen. Sie ist nicht notwendig, wenn sich die Datenbank auf demselben Rechner befindet, wie das Java-Programm oder wenn sie mit einem Alias über einen Dienst angesprochen wird.

Der Server wird durch eine IP-Adresse spezifiziert, entweder in der Punktnotation, z. B. 67.8.78.24 oder durch einen Domännennamen, z. B. www.integrata.de.

Der Port folgt auf die IP-Adresse durch einen ':' getrennt. Server und Port haben folgende vollständige Notation

```
//host:port
```

7.5.1.3 Datei- oder Datenbankname

Unbedingt erforderlich ist natürlich der Datenbankname. Wie dieser zu spezifizieren ist, hängt von dem jeweiligen Hersteller ab. Es gibt daher keine allgemein gültige Regel für den Datenbanknamen.

7.5.1.4 Vollständige URL

Eine vollständige Datenbank-URL setzt sich also folgendermaßen zusammen

```
jdbc:subprotokoll:[//host]:[port]/dbName[;parameter]
```

7.5.2 Benutzer und Kennwort

Wenn auf einer Datenbank Benutzer mit Kennwörtern eingerichtet sind, so können diese beim Öffnen der Verbindung spezifiziert werden. Sowohl der Benutzer als auch das Kennwort werden als String übergeben.

7.5.3 Verbindung öffnen

Wenn der Treiber geladen worden ist und wir die URL bestimmt haben, so können wir mit Benutzer und Kennwort eine Verbindung zur Datenbank öffnen. Dazu verwenden wir den `java.sql.DriverManager`. Er besitzt die Methoden:

- **`getConnection(String url)`**, um eine Verbindung zu einer Datenbank ohne Benutzer und Kennwörter aufzubauen, und
- **`getConnection(String url, String usr, String pwd)`**, um eine Verbindung zu einer Datenbank aufzubauen für einen bestimmten Benutzer mit dessen Kennwort.
- **`getConnection(String url, Properties props)`**, um eine Verbindung aufzubauen, bei der Benutzer, Kennwort und Parameter über ein Property-Set angegeben werden können.

Im Quellcode sieht das dann folgendermaßen aus:

```
String url =
    "jdbc:cloudscape:rmi:Customer;create=true";

Connection conn = null;

try {
    conn = DriverManager.getConnection(url);
} // try
...
catch (SQLException ex) {
    System.out.println( ex );
} // catch
```

Die Methode `getConnection()` ist der erste Berührungspunkt mit der JDBC- API in unserem Programm. Wir müssen bei ihrer Verwendung mit einer `java.sql.SQLException` rechnen. Im weiteren Verlauf werden wir sehen, dass wie bei fast allen Methoden der JDBC-API mit diesem Exception-Typ gerechnet werden muss.

Besser wird es mit try mit Ressourcen, da bei einer `SQLException` die `Connection` automatisch geschlossen wird:

```
try (Connection conn =
    DriverManager.getConnection(url)) {
    System.out.println(con);
}
catch (SQLException ex) {
    System.out.println( ex );
}
```


7.5.4 Verbindung schließen

Eine Datenbankverbindung ist eine aufwändige Sache. Wir sollten sie daher nicht länger beanspruchen als notwendig. So erlauben viele Datenbank-Managementsysteme nur eine beschränkte Anzahl von Verbindungen die, wenn sie nicht mehr gebraucht werden, geschlossen werden sollten, um anderen Anwendungen den Zugriff auf die Datenbank zu erlauben.

Das geschieht durch folgenden Befehl:

```
//...
try {
    ...
    conn = DriverManager.getConnection(url, usr, pwd);
} // try
//...                               // catch-Blöcke
finally {
    try {
        conn.close();
    } // try
    catch (SQLException ex) {
        System.out.println("Close connection failed!");
    } // catch
} // finally
```

Wir haben das Schließen in einen `finally`-Block gelegt, um sicherzustellen, dass die Datenbank auch wirklich geschlossen wird, egal ob der Verlauf richtig oder falsch war. Selbst das Schließen der Verbindung kann eine `SQLException` erzeugen. Daher müssen wir auch den `close()`-Befehl in einen `try-catch`-Block legen.

Sollen wir eine Verbindung nach jeder Aktion schließen und bei jeder neuen aufbauen? Die Antwort darauf ist sicher nein. Die Verbindung sollte nur dann geschlossen werden, wenn längere Zeit keine Zugriffe auf die Datenbank gemacht werden, anderenfalls sollten wir sie offen halten. Um diese Funktionalität abzubilden, werden Datenbank-Verbindungen häufig in Connection-Pools gehalten, die dann die Verwaltung zentral übernehmen.

7.5.5 Verwendung der Connection

Eine Connection wird benutzt:

1. Um SQL-Statements zu erzeugen und verwalten:

```
Statement s = con.createStatement();  
PreparedStatement s =  
    con.prepareStatement(sqlString);  
CallableStatement s =  
    con.createStatement(sqlString);
```

2. Um den Zugriff zu steuern:

```
con.setReadOnly(true);  
con.setAutoCommit(false);
```

3. Um Atomic-Transaktionen zu gruppieren:

```
con.commit();  
con.rollback();
```

7.6 SQL-Befehle senden

Sobald wir eine Verbindung zur Datenbank besitzen, können wir anfangen, SQL-Befehle an das DBMS zu schicken.

7.6.1 JDBC Statements

The screenshot shows a presentation slide with a teal header bar containing the text 'Java und Datenbanken' and navigation icons. The slide content is as follows:

JDBC Statements

- Statements werden innerhalb einer Connection erzeugt.
- `java.sql.Statement`
 - SQL wird an die DB gesandt und dort wird für jedes execute ein Bearbeitungsplan erstellt
 - Relativ langsam, aber flexibel
- `java.sql. PreparedStatement`
 - SQL wird an die DB gesandt und dort wird nur einmal ein Bearbeitungsplan erstellt
 - Schneller, besonders für oft wiederholte Abfragen
- `java.sql.CallableStatement`
 - Vorhandene SQL-Programme (stored procedures) werden aufgerufen
 - Am schnellsten, SQL-Code muss auf dem Server verfügbar sein

3.0.1218 © Integrata AG Java Erweiterungen I 95

Abb. 7-12: Statements

Dafür benötigen wir eine Implementierung des `java.sql.Statement`-Interfaces. Das erhalten wir aus der Datenbankverbindung.

```
Statement stmt = null;

try {
    stmt = conn.createStatement();
    ...
} // try
catch (SQLException ex) {
    ...
} // catch
finally {
    if (stmt != null) {
        try {
            stmt.close();
        } // try
        catch (SQLException ex) {
            ...
        } // catch
    } // if
} // finally
```

Ähnlich wie bei der Datenbankverbindung gilt für ein `Statement`-Objekt, dass es geschlossen werden sollte, wenn es nicht mehr benötigt wird. Auch das `Statement`-Interface deklariert die Methode `close()`.

Mit diesem `Statement` können wir nun beliebige SQL-Befehle an das DBMS schicken.

The screenshot shows a presentation slide with a title bar 'Java und Datenbanken' and navigation icons. The main content is titled 'SQL-Befehle' and lists methods for `Statement`, `PreparedStatement`, and `CallableStatement`. The footer contains '3.0.1218 © Integrata AG', 'Java Erweiterungen I', and the page number '96'.

SQL-Befehle

- verschiedene Methoden je nach Rückgabewert
- in `Statement`:
 - `ResultSet executeQuery (String sql)`
 - `int executeUpdate (String sql)`
 - `boolean execute (String sql) + getResultSe() / getUpdateCount()`
- in `PreparedStatement` und `CallableStatement`:
 - SQL-String beim Anlegen des Statements + set-Methoden
 - `ResultSet executeQuery ()`
 - `int executeUpdate ()`
 - `boolean execute () + getResultSe() / getUpdateCount()`

3.0.1218 © Integrata AG Java Erweiterungen I 96

Abb. 7-13: SQL-Befehle

Die Methoden dafür heißen:

- `executeUpdate(String sql)`, für beliebige Update-Anweisungen, aber keine SELECT-Befehle
- `executeQuery(String sql)`, nur für SELECT-Befehle und
- `execute(String sql)`, für beliebige SQL-Befehle.

Es verwundert vielleicht, warum es drei verschiedene Methoden gibt, um SQL-Anweisungen abzuschicken, wenn doch die `execute()`-Methode jeden SQL-Befehl verarbeiten kann. Die anderen beiden Methoden sind in ihrer Verwendung komfortabler. Wenn wir eine Abfrage ausführen, interessiert uns die Ergebnismenge. Bei einer Aktualisierung bekommen wir die Anzahl der veränderten Datensätze zurück.

Die Methoden unterscheiden sich im Rückgabewert. `executeQuery()` liefert eine Ergebnistabelle, die wir auswerten können. `executeUpdate()` liefert einen `int`-Wert. Die `execute()`-Methode liefert hingegen nur einen `boolean`-Wert. Wenn er `true` ist, handelte es sich um ein Abfrage und die Ergebnistabelle muss mit einer eigenen Methode abgefragt werden. Wenn er `false` ist, war es ein Update. `execute()` verwenden wir nur dann, wenn wir es mit generischen SQL-Anweisungen zu tun haben, ansonsten verwenden wir `executeQuery()` oder `executeUpdate()`.

Das `Statement` ist sehr flexibel, da es uns erlaubt beliebige SQL-Befehle zur Laufzeit zu erzeugen und an das DBMS zu schicken. Allerdings ist es auch das langsamste in der Ausführung. Jedes SQL-Statement muss aufs Neue vom DBMS geparkt werden und kann erst dann ausgeführt werden. Wenn wir eine Reihe kongruenter SQL-Befehle haben, in denen sich nur die Werte ändern, so ist das eine Zeitverschwendung.

Daher gibt es in der JDBC-API noch zwei weitere Statements:

`java.sql.PreparedStatement` und
`java.sql.CallableStatement`

Sie erlauben eine wesentlich effizientere Ausführung ähnlicher SQL-Befehle.

7.6.2 PreparedStatement und CallableStatement

Das `java.sql.PreparedStatement` ist direkt von `Statement` abgeleitet. Beim `PreparedStatement` definieren wir eine Schablone mit Platzhaltern. Die Platzhalter stehen für Werte, die wir füllen, bevor wir das Statement an das DBMS schicken. Dadurch muss das DBMS den SQL-Befehl nicht immer von neuem analysieren, sondern muss diese Arbeit nur einmal machen und später nur noch die entsprechenden Werte einsetzen.

Wir erzeugen ein `PreparedStatement` ebenfalls aus der `Connection`-Implementierung:

```
PreparedStatement pstmt = conn.prepareStatement(  
    "INSERT INTO CUSTOMER VALUES (?, ?, ?)");
```

Statement	PreparedStatement
<code>boolean execute(String sql);</code>	<code>boolean execute();</code>
<code>int executeUpdate(String sql);</code>	<code>int executeUpdate();</code>
<code>ResultSet executeQuery(String sql);</code>	<code>ResultSet executeQuery();</code>

Im Gegensatz zur Anweisung `createStatement()` erwartet die Methode `prepareStatement(String)` eine Zeichenkette, die das Muster für die zu erzeugenden SQL-Anweisungen enthält.

Jedes Fragezeichen steht für einen Platzhalter. Die Fragezeichen werden nun mit 1 beginnen durchnummeriert. Jeder Wert kann durch ein

`setXXX(int index, XXX value)`

des `PreparedStatement`s gesetzt werden. `index` spezifiziert dabei die Nummer des Platzhalters und `XXX` steht für den Datentyp, der dem Platzhalter zugeordnet ist.

Beispiel:

```
pstmt.setString(1, "Metzger");  
pstmt.setString(2, "Georg");  
pstmt.setInt(3, 56);  
pstmt.executeUpdate();
```

Am Ende wird das `PreparedStatement` mit einer der `execute`-Anweisungen abgeschickt. Hier ist nun jedoch keine weitere Information mehr von Nöten, sodass die Methode ohne Parameter auskommt.

Das `PreparedStatement` ist wesentlich schneller als das einfache `Statement`, wenn mehrere gleichartige Befehle weggeschickt werden müssen. Dafür ist es weniger flexibel als das einfache `Statement`.

`java.sql.CallableStatement` erweitert das `PreparedStatement` und ist die Schnittstelle, die den Durchgriff auf Methoden, die innerhalb der Datenbank definiert sind, ermöglicht. `CallableStatements` erfordern in der Regel eine detaillierte Kenntnis des verwendeten DBMS und können hier nicht ausführlicher erörtert werden.

7.7 Auswertung der Ergebnisse

7.7.1 ResultSet

Das Ergebnis einer einfachen Abfrage ist ein statisches `java.sql.ResultSet`. In einer Schleife können die einzelnen Datensätze der Ergebnistabelle durchlaufen werden. Bei jedem Schleifendurchlauf können die einzelnen Felder eines Datensatzes über den Spaltennamen oder über den Spaltenindex ausgelesen werden.

Java und Datenbanken

⬅
⬅
➡

Auswertung der Ergebnisse

beforeFirst

id	Pers_NR	Nachname	Vorname
1	K23408	Müll	Gerd
2	L77634	Abfall	Peter

- ResultSet = Ergebnis einer DB-Abfrage (Query)
- abhängig vom der JDBC-Version wird ein Cursor zur Navigation in der Ergebnismenge verwaltet
- Initial steht der Cursor vor der ersten Zeile der Ergebnismenge
- Methoden für Zugriff auf Daten

```

public boolean next()           //Existiert noch eine gültige
                                //Reihe?
Typ getTyp(String spalte)      //Liefert einen Typ zurück.
Typ getTyp(int spalte)         //Liefert einen Typ zurück.
first()
last()

Beispiel:
ResultSet rs = stmt.executeQuery("SELECT * FROM Tabelle");
while(rs.next()) {
    String sNachname = rs.getString("Nachname");
    String sVorname = rs.getString("Vorname");
}
```

3.0.1218 © Integrata AG
Java Erweiterungen I
97

Abb. 7-14: Auswertung der Ergebnisse

Eine generische Methode zum Auslesen des `ResultSet` ist `getObject()`, sie kann alle Daten auslesen. Mit `toString()` kann dann zumindest eine textliche Darstellung des Wertes erreicht werden. In der Regel ist das aber nicht das, was wir haben wollen, da wir die Werte ja weiterverarbeiten möchten und nicht nur anzeigen möchten.

Das `ResultSet` steht nach der Abfrage vor der ersten Zeile der Ergebnismenge. Um über diese Ergebnismenge zu laufen, bietet `ResultSet` die Methode

```
public boolean next()
```

deren Rückgabewert anzeigt, ob noch weitere Zeilen folgen oder nicht.

```
while(rs.next()) {
    //ResultSet auslesen
}
```

ist eine Schleife über die Ergebnismenge.

Zum Auslesen der Daten bietet uns das `ResultSet` deshalb eine Reihe von `getXYZ()`-Methoden an. XYZ steht dabei für den Datentyp, der ausgelesen werden soll. Wollen wir z. B. einen `int`-Wert auslesen, so verwenden wir die Methode `getInt(int pIndex)` oder `getInt(String pSpaltenName)`.

In der folgenden Methode wird eine `SELECT`-Anweisung an das DBMS geschickt und in einer `while`-Schleife ausgewertet.

7.7.2 Ergebnistabelle schließen

Nachdem das `ResultSet` ausgelesen worden ist, sollte es wieder geschlossen werden. Das geschieht mit dem Befehl: `close()`

Beispiel:

```
Statement stmt = conn.createStatement();
ResultSet rs = null;

try {
    rs = stmt.executeQuery("SELECT * FROM CUSTOMER");

    while (rs.next()) {
        System.out.println(rs.getObject(1));
    } // while
} // try
finally {
    rs.close();
    stmt.close();
} // finally
con.close();
```


7.8 Zusammenfassung Datenbankzugriff

Java und Datenbanken
🏠 ⏪ ⏩ ➡

Datenbankzugriff Schritt für Schritt

1. Treiber laden
`Class.forName(...)`
2. URL definieren
z.B. "jdbc:odbc:Tabelle" oder
3. Verbindung zur Datenbank herstellen
`con = DriverManager.getConnection(url, user, password);`
4. Statement besorgen
`stmt = con.createStatement();`
5. Anfrage in SQL
`rs = stmt.executeQuery("SELECT * FROM Tabelle");`
6. Ergebnisse ausgeben

```
while(rs.next()) {
    int i = rs.getInt(1);
    String s = rs.getString("Vorname");
}
```

3.0.1218 © Integrata AG
Java Erweiterungen I
98

Abb. 7-15: Zusammenfassung

7.8.1 Komplettes Beispiel

```
import java.math.BigDecimal;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class Application {

    private final static String
        url = "jdbc:derby:../dependencies/derby/data";
    private final static String user = "user";
    private final static String password = "password";

    public static void main(String[] args) {
        try (final Connection con = DriverManager.getConnection(
            url, user, password)) {

            dropBookTable(con);
            createBookTable(con);
            insertBooks(con);
            selectBooks(con);
            selectBook(con);
        }
        catch (SQLException e) {
            System.out.println(e);
        }
    }
}
```

```
static void dropBookTable(Connection con)
    throws SQLException {

    final String sql = "drop table book";
    try (final Statement stmt = con.createStatement()) {
        stmt.execute(sql);
    }
    catch (Exception e) {
        System.out.println(e);
    }
}

static void createBookTable(Connection con)
    throws SQLException {

    final String sql = "create table book (" +
        "isbn char(13), " +
        "title varchar(64), " +
        "price decimal, " +
        "primary key(isbn)" +
        ")";
    try (final Statement stmt = con.createStatement()) {
        stmt.execute(sql);
    }
}

static void insertBooks(Connection con) {

    final String sql = "insert into book values (?, ?, ?)";
    try (final PreparedStatement stmt =
        con.prepareStatement(sql)) {
        insert(stmt, "1111", "Pascal", BigDecimal.valueOf(10.10));
        insert(stmt, "2222", "Modula", BigDecimal.valueOf(20.20));
        insert(stmt, "3333", "Oberon", BigDecimal.valueOf(30.30));
    }
    catch (SQLException e) {
        System.out.println(e);
    }
}

static void selectBooks(Connection con) {
    Log.startMethod();
    final String sql = "select isbn, title, price from book";
    try (final Statement stmt = con.createStatement()) {
        try (final ResultSet rs = stmt.executeQuery(sql)) {
            while(rs.next())
                print(rs);
        }
    }
    catch (SQLException e) {
        System.out.println(e);
    }
}
```

```
static void selectBook(Connection con) {
    final String sql =
        "select isbn, title, price from book where isbn = ?";
    try (final PreparedStatement stmt =
        con.prepareStatement(sql)) {
        final String isbn = "2222";
        stmt.setString(1, isbn);
        try (final ResultSet rs = stmt.executeQuery()) {
            if(rs.next())
                print(rs);
            else
                System.out.println("book " + isbn + " not found");
        }
    }
    catch (SQLException e) {
        System.out.println(e);
    }
}

static void insert(PreparedStatement stmt, String isbn,
    String title, BigDecimal price) throws SQLException {
    stmt.setString(1, isbn);
    stmt.setString(2, title);
    stmt.setBigDecimal(3, price);
    int count = stmt.executeUpdate();
    // System.out.println(count);
}

static void print(ResultSet rs) throws SQLException {
    final String isbn = rs.getString("isbn");
    final String title = rs.getString("title");
    final BigDecimal price = rs.getBigDecimal("price");
    System.out.println(isbn + " " + title + " " + price);
}
}
```

Ergebnis:

```
1111    Pascal 10
2222    Modula 20
3333    Oberon 30
2222    Modula 20
```

8

Anhang

8.1	Eclipse IDE Setup.....	8-3
8.1.1	Download Eclipse	8-3
8.1.2	Erzeugen eines ersten Java Projektes.....	8-5
8.1.3	Eclipse Shortcuts	8-5
8.2	Operator-Rangfolge	8-7
8.3	Online-Dokumentation	8-9
8.4	Javadoc	8-10
8.5	Java – Glossar.....	8-11

8 Anhang

8.1 Eclipse IDE Setup

8.1.1 Download Eclipse

Auf der Webseite: www.eclipse.org finden Sie das Tool Eclipse. Diese kostenlose Entwicklungsumgebung für Java bietet dem Programmierer viele Erleichterungen an. U.a. können Getter- u. Setter-Methoden generiert oder Konstruktoren auf Grundlage von Attributen erzeugt werden.

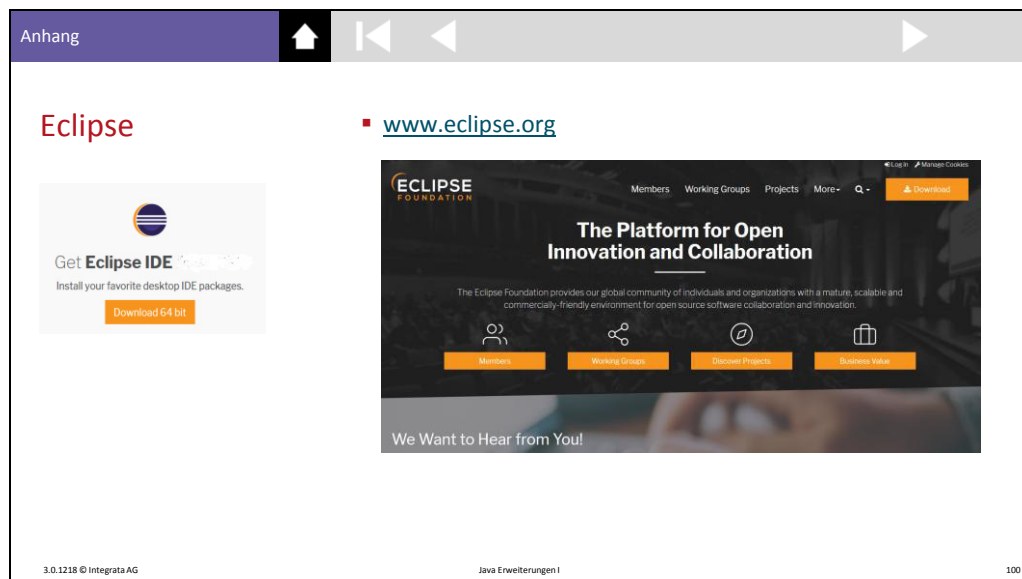
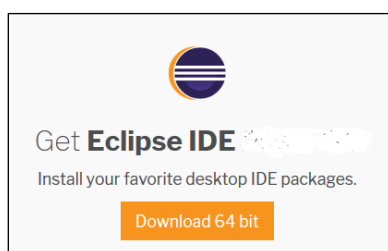


Abb. 8-1: Eclipse

- Download



- Installation

- Entzippen der Dateien auf ein beliebiges Verzeichnis der Festplatte, z.B. c:
- Eclipse befindet sich dann im Unterverzeichnis c:/eclipse
- Ausführen des Programms (Windows) c:/eclipse/eclipse.exe

- Start
 - Auswahl des Arbeitsverzeichnisses (Workspace)
- Anfangsbild: Help -> Welcome

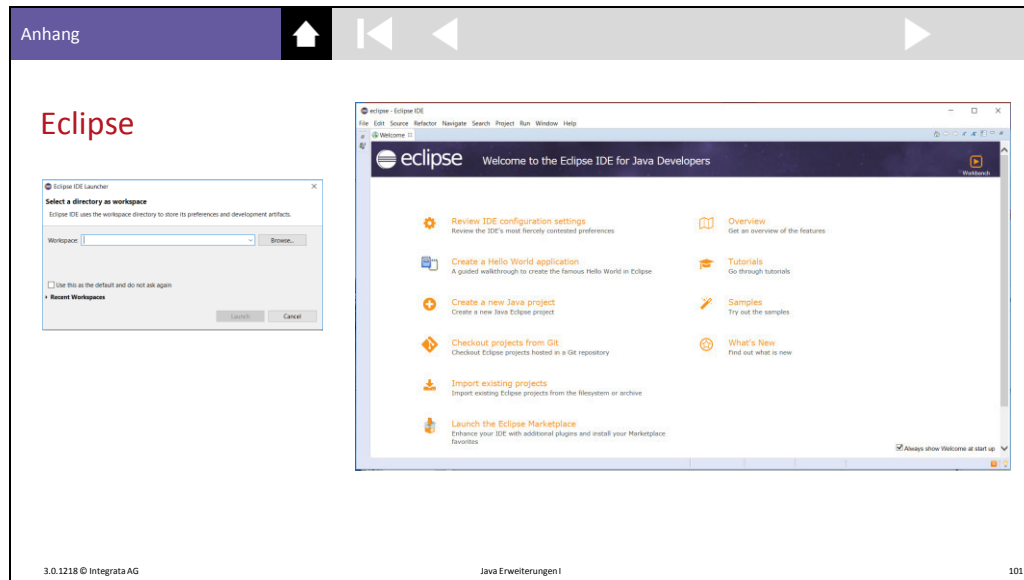


Abb. 8-2: Einstiegsbild

Nach dem Entpacken ist es sinnvoll, dass sie sich ein Icon für den Start auf dem Desktop oder in einem der Menüs erzeugen. Durch rechten Maus Click auf einem freien Bereich des Desktops, starten sie den Dialog ,**Verknüpfung erstellen**'.

Eclipse bietet gleich auf dem Startbild eine Dokumentation und Beispiele an, so dass die Bedienung relativ leichtfällt.

8.1.2 Erzeugen eines ersten Java Projektes

Mit dem Projekt Wizard in der linken oberen Ecke des Programmfensters oder mit dem Menu File.New.Java Project öffnen sie den ‚new Java project‘ Dialog. Hier können sie den Projektnamen und ein bestehendes Projekt Directory auswählen.

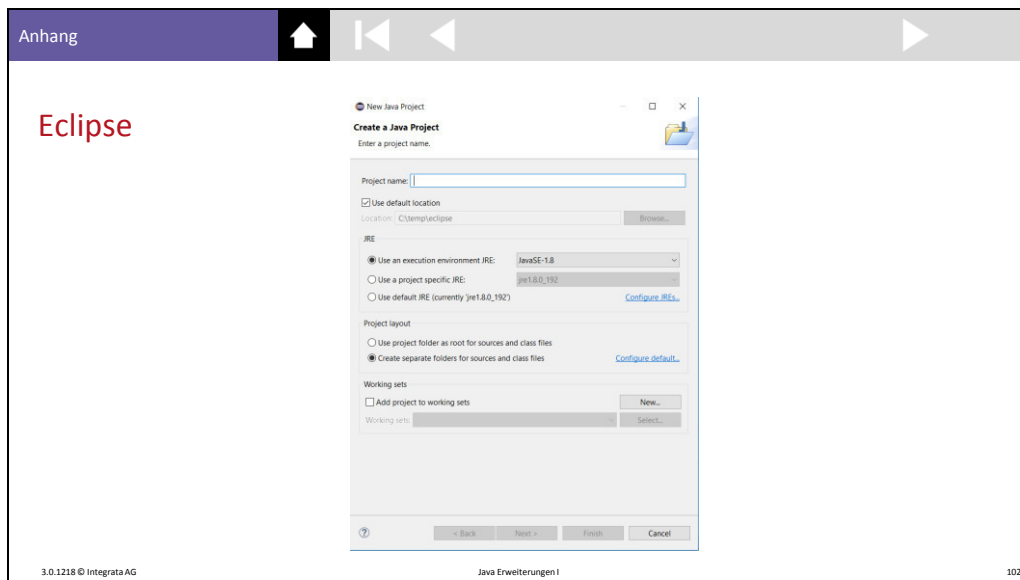


Abb. 8-3: Projekterzeugung

Wählen sie hier ‚Finish‘ und schließen Sie den Dialog ab.

8.1.3 Eclipse Shortcuts

[F1]	Hilfe
[Strg] + [Space]	Content Assist
[Strg] + [1]	Quick Fix
[Strg] + [Shift] + [T]	Open Type
[Strg] + [Shift] + [R]	Open Resource
[F3]	Open Declaration
[Strg] + [Shift] + [P]	Zur passenden Klammer springen
[Strg] + [O]	Quick Outline (mit Vererbungs-Hierarchie)
[Strg] + [.]	Next Annotation
[Strg] + [,.]	Previous Annotation
[F2]	Fokus on Info/Error
[Strg] + [I]	Einrückungen korrigieren (Zeile/markierter Ber.)
[Strg] + [Shift] + [Up/Down]	Von Methode zu Methode springen

[Alt] + [Shift] + [Up/Down]	Blockweise Markierung erweitern
[Alt] + [Left/Right]	Previous / Next Editor (nach Historie)
[Alt] + [Up/Down]	Zeile oder Auswahl im Editor verschieben
[Strg] + [F6]	wechselt zwischen offenen Editoren
[Strg] + [F7]	wechselt zwischen offenen Views
[Strg] + [F8]	wechselt zwischen aktuellen Perspektiven
[Strg] + [F]	Find / Replace
[Strg] + [T]	Quick Type Hierarchy
[Strg] + [Alt] + [H]	Open Call Hierarchy
[Shift] + [Left/Right]	Markierung Zeichenweise
[Strg] + [Shift] + [Left/Right]	Markierung Wortweise
[Strg] + [Alt] + [Up/Down]	Zeile oder Auswahl Duplizieren
[Strg] + [Z]	Rückgängig
[Strg] + [Shift] + [U]	Schnelle Suche auf dem ausgew. Element
[Strg] + [Shift] + [O]	Organize Imports
[Strg] + [/]	Zeile oder Auswahl aus-/einkommentieren
[Strg] + [Shift] + [Space]	Parameter Hinweise
[Strg] + [E]	Auswahl eines Editors
[Strg] + [Shift] + [L]	Zeigt alle aktuell verfügbaren Shortcuts

8.2 Operator-Rangfolge

Die Tabelle listet alle Operatoren in der Reihenfolge ihrer Vorrangregeln auf. Weiter obenstehende Operatoren haben dabei Vorrang vor weiter untenstehenden Operatoren. Innerhalb derselben Gruppe stehende Operatoren werden entsprechend ihrer Assoziativität ausgewertet.

Die Spalte Typisierung gibt die möglichen Operanden Typen an. Dabei steht »N« für numerische, »I« für integrale (also ganzzahlig numerische), »L« für logische, »S« für String-, »R« für Referenz- und »P« für primitive Typen. Ein »A« wird verwendet, wenn alle Typen in Frage kommen, und mit einem »V« wird angezeigt, dass eine Variable erforderlich ist.

Gruppe	Operator	Typisierung	Assoziativität	Bezeichnung
1	++	N	R	Inkrement
	--	N	R	Dekrement
	+	N	R	Unäres Plus
	-	N	R	Unäres Minus
	~	I	R	Einerkomplement
	!	L	R	Logisches NICHT
	(type)	A	R	Type-Cast
2	*	N,N	L	Multiplication
	/	N,N	L	Division
	%	N,N	L	Modulo
3	+	N,N	L	Addition
	-	N,N	L	Subtraction
	+	S,A	L	String-Verkettung
4	<<	I,I	L	Linksschieben
	>>	I,I	L	Rechtsschieben
	>>>	I,I	L	Rechtsschieben mit Nullexpansion
5	<	N,N	L	Kleiner
	<=	N,N	L	Kleiner gleich
	>	N,N	L	Größer
	>=	N,N	L	Größer gleich
	instanceof	R,R	L	Klassenzugehörigkeit
6	==	P,P	L	Gleich
Gruppe	Operator	Typisierung	Assoziativität	Bezeichnung
6	!=	P,P	L	Ungleich
	==	R,R	L	Referenzgleichheit

	!=	R,R	L	Referenzungleichheit
7	&	I,I	L	Bitweises UND
	&	L,L	L	Logisches UND mit vollständiger Auswertung
8	^	I,I	L	Bitweises Exklusiv-ODER
	^	L,L	L	Logisches Exklusiv-ODER
9		I,I	L	Bitweises ODER
		L,L	L	Logisches ODER mit vollständiger Auswertung
10	&&	L,L	L	Logisches UND mit Short-Circuit-Evaluation
11		L,L	L	Logisches ODER mit Short-Circuit-Evaluation
12	?:	L,A,A	R	Bedingte Auswertung
13	=	V,A	R	Zuweisung
	+=	V,N	R	Additionszuweisung
	-=	V,N	R	Subtraktionszuweisung
	*=	V,N	R	Multiplikationszuweisung
	/=	V,N	R	Divisionszuweisung
	%=	V,N	R	Restwertzuweisung
	&=	N,N u. L,L	R	Bitweises-UND-Zuweisung und Logisches-UND-Zuweisung
	=	N,N u. L,L	R	Bitweises-ODER-Zuweisung und Logisches-ODER-Zuweisung
	^=	N,N u. L,L	R	Bitweises-Exklusiv-ODER-Zuweisung und Logisches-Exklusiv-ODER-Zuweisung
	<<=	V,I	R	Linksschiebezuweisung
	>>=	V,I	R	Rechtsschiebezuweisung
	>>>=	V,I	R	Rechtsschiebezuweisung mit Nullexpansion

8.3 Online-Dokumentation

Eine übersichtliche Darstellung und vollständige Dokumentation der Klassenbibliothek ist wie das JDK frei erhältlich. Sie kann auf dem Web-Server der Firma Sun gelesen werden, z.B. für die Version 8 an der Adresse

<http://docs.oracle.com/javase/8/docs/api/index.html>

<http://docs.oracle.com/javase/9/docs/api/index.html>

<http://docs.oracle.com/javase/10/docs/api/index.html>

oder auch auf den eigenen Rechner heruntergeladen werden. Die API-Dokumentation liegt bei einer Standard-Installation im Verzeichnis

%JAVA_HOME%\doc\api\index.html

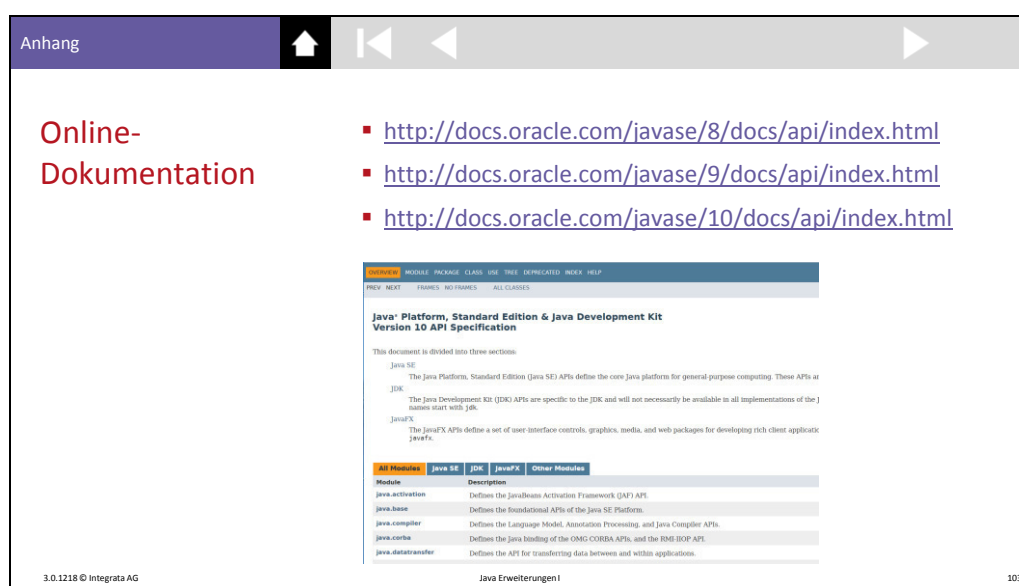


Abb. 8-4: Dokumentation

8.4 Javadoc

The screenshot shows a presentation slide with a purple header bar containing the word 'Anhang' and navigation icons. The slide title 'Javadoc' is in red. It contains a bulleted list of Javadoc features and tags. At the bottom, there is a footer with version information '3.0.1218 © Integrata AG', the text 'Java Erweiterungen I', and the page number '104'.

Javadoc

- Eigene Dokumentationen können automatisch generiert werden durch das im JDK enthaltene Tool javadoc.
- javadoc durchsucht Quelldateien nach Klassen, Methoden und Kommentaren.
- javadoc-Kommentare werden mit dem speziellen Kommentar `/** ... */` eingegrenzt.
- Ein Kommentar besteht aus beliebigem Text im HTML-Format.
- Zusätzlich erkennt javadoc noch spezielle Tags:
 - `@see Klasse`
 - `@see Klasse#Methode`
 - `@version Versionsname oder -nummer`
 - `@author Name des Autors`
 - `@return Beschreibung des Rückgabewertes`
 - `@throws Exceptionbeschreibung`
 - `@param Parameterbeschreibung`

3.0.1218 © Integrata AG Java Erweiterungen I 104

Abb. 8-5: Javadoc

Beispiele für den Aufruf von javadoc:

```
javadoc *.java
```

Alle Quellcodes werden im aktuellen Verzeichnis dokumentiert (nur die öffentlichen Schnittstellen).

```
javadoc -private -d c:\_training\doc *.java
```

Alle Quellcodes werden inklusive der privaten Elemente im Unterverzeichnis `c:_training\doc` dokumentiert.

Auch ganze Pakete können kommentiert werden:

```
javadoc -private -sourcepath ..\src -d ..\doc
```

8.5 Java – Glossar

Anweisung

Eine Anweisung (engl. Statement) ist ein einzelner Befehl einer Programmiersprache, der dafür sorgt, dass etwas passiert. Anweisungen stehen für eine einzelne Aktion, die in einem Java-Programm ausgeführt wird. Jede Anweisung wird mit einem Strichpunkt (;) abgeschlossen.

API

Sogenannte APIs (Application Programming Interfaces) stellen innerhalb von Funktionsbibliotheken sämtliche zur Programmierung benötigten Routinen mehr oder weniger komfortabel zur Verfügung.

Applet

Ein Applet wird innerhalb eines WWW-Browsers ausgeführt, der Java unterstützt. Applets können zusätzlich mit dem Applet-Viewer, der im JDK enthalten ist, angezeigt werden. Um ein Applet auszuführen, muss es in eine Webseite eingefügt werden.

Applikation

Eine Applikation (Anwendung) ist ein Java-Programm, das eigenständig lauffähig ist. Applikationen werden mit dem Java-Interpreter ausgeführt, der die Haupt-.class-Datei der Applikation lädt. Dazu wird meist das java-Tool des JDK von der Kommandozeile aus aufgerufen.

Arrays

Arrays stellen eine Methode zur Speicherung einer Reihe von Elementen dar, die alle denselben primitiven Datentyp oder dieselbe Klasse aufweisen. Jedem Element wird innerhalb des Arrays ein eigener Speicherplatz zugewiesen. Diese Speicherplätze sind nummeriert, so dass Sie auf die Informationen leicht zugreifen können. Java implementiert Arrays anders als andere Programmiersprachen, nämlich als Objekte, die wie andere Objekte auch behandelt werden können.

Assoziativität

Die Assoziativität regelt die Abarbeitung von Operatoren gleicher Präzedenz. In Java gilt die Regel, dass alle zweistelligen Operatoren links-assoziativ sind. Dies bedeutet, dass eine Formel mit gleichen Präzedenzen immer von links nach rechts abgearbeitet wird. Einzige Ausnahme sind die Zuweisungsoperatoren, welche rechts-assoziativ sind.

Attribut

Attribute sind die einzelnen Dinge, die die Klassen voneinander unterscheiden. Sie legen auch die Erscheinung, den Zustand und andere Qualitäten der Klasse fest. In einer Klasse werden Attribute über Variablen definiert.

Ausdruck

Ein Ausdruck (engl. Expression) ist eine Anweisung, die als Ergebnis einen Wert produziert. Dieser Wert kann zur späteren Verwendung im Programm gespeichert, direkt in einer anderen Anweisung verwendet oder überhaupt nicht beachtet werden. Der Wert, den eine Anweisung erzeugt, wird Rückgabewert genannt.

AWT

Das Abstract Windowing Toolkit, auch AWT genannt, ist ein Satz von Klassen, der es Ihnen ermöglicht, eine grafische Benutzeroberfläche zu erstellen und Eingaben des Benutzers über die Maus und die Tastatur entgegenzunehmen. Da Java eine plattformunabhängige Sprache ist, haben die Benutzerschnittstellen, die mit dem AWT entworfen werden, auf allen Systemen die gleiche Funktionalität und abgesehen von den Plattformeigenheiten die gleiche Erscheinung.

Bedingung

Eine Bedingung ist eine Programmanweisung, die nur dann ausgeführt wird, wenn eine bestimmte Situation eintritt. Die elementarste Bedingung wird mit dem Schlüsselwort `if` erzeugt. Eine `if`-Bedingung verwendet einen booleschen Ausdruck, um zu entscheiden, ob eine Anweisung ausgeführt werden soll. Wenn der Ausdruck `true` zurückliefert, wird die Anweisung ausgeführt.

Blockanweisung

Eine Gruppe von Anweisungen, die sich in einem Paar geschweifter Klammern (`{ }`) befindet, wird als Block oder Blockanweisung bezeichnet.

boolesche Werte

Boolesche Werte sind spezielle Variablentypen, die nur die Werte `true` oder `false` aufnehmen können. Im Gegensatz zu anderen Sprachen haben boolesche Werte keine numerischen Werte. Der Ausdruck `boolesch` geht auf George Boole, einen irischen Mathematiker zurück, der von 1815 bis 1864 lebte.

Casting

Casting ist ein Mechanismus, um einen neuen Wert zu erstellen, der einen anderen Typ aufweist als dessen Quelle. Casting ergibt ein neues Objekt oder einen neuen Wert. Casting wirkt sich nicht auf das ursprüngliche Objekt bzw. den ursprünglichen Wert aus.

Deadlock

Immer wenn zwei Threads eine Sperre auf zwei Objekte durchführen und dann anschließend versuchen, kreuzweise auf das andere, jeweils dann gesperrte Objekt zuzugreifen, besteht die Gefahr einer Verklemmung (eines Deadlocks). Beide Threads warten darauf, dass das jeweils andere Objekt seine Sperre verliert, was natürlich niemals geschieht. Ihr Programm bleibt stehen und arbeitet nicht mehr weiter.

Java hat keine Möglichkeiten, solche Deadlocks zu erkennen. Es ist Sache des Programmierers, Deadlocks zu vermeiden und dafür Sorge zu tragen, dass diese nicht auftreten.

dekrementieren

Eine Variable zu dekrementieren bedeutet, von deren Wert eins abziehen. Der Dekrement-Operator ist `--`. Dieser Operator wird direkt nach oder direkt vor einen Variablennamen platziert.

doppelte Pufferung

Die doppelte Pufferung ist ein Vorgang, bei dem alle Zeichenaktivitäten in einem Puffer abseits des Bildschirms vorgenommen werden. Anschließend wird der gesamte Inhalt dieses Puffers in einem Schritt am Bildschirm angezeigt. Diese Technik wird doppelte Pufferung genannt, weil es zwei Puffer für Grafikausgaben gibt, zwischen denen Sie wechseln.

Exception

Java stellt mit dem Sprachkonstrukt der Exception eine elegante Methode zur Verfügung, die Fehlerbehandlung in Programmen durchzuführen, ohne dass Klarheit und Einfachheit des Codes darunter leiden.

Expression

siehe Ausdruck

Felder

siehe Arrays

Finalizer

Finalizer-Methoden sind in gewissem Sinn das Gegenstück zu Konstruktor-Methoden. Während eine Konstruktor-Methode benutzt wird, um ein Objekt zu initialisieren, werden Finalizer-Methoden aufgerufen, kurz bevor das Objekt im Papierkorb landet und sein Speicher zurückgefordert wird. Finalizer-Methoden eignen sich am besten zur Optimierung des Entfernens von Objekten, z.B. durch Löschen aller Referenzen auf andere Objekte. In den meisten Fällen benötigen Sie `finalize()` überhaupt nicht.

Garbage Collection

Garbage Collection ist ein Mechanismus, der nicht mehr benutzten, dynamisch angeforderten Speicherplatz auf dem Heap freigibt.

Gosling

James Gosling erfand 1991 die Programmiersprache Java bei Sun Microsystems.

Gültigkeitsbereich

Gültigkeitsbereich ist in der Programmierung der Begriff für den Teil eines Programms, in dem eine Variable existiert und verwendet werden kann. Wenn das Programm den Gültigkeitsbereich einer Variablen verlässt, dann existiert diese nicht, und es treten Fehler auf bei dem Versuch, auf diese zuzugreifen. Der Gültigkeitsbereich einer Variablen ist der Block, in dem sie erzeugt wurde. Wenn Sie in einem Block lokale Variablen deklarieren und verwenden, dann hören diese Variablen auf zu existieren, sobald der Block ausgeführt ist.

GUI

Das sogenannte GUI (Graphical User Interface) ist für alles verantwortlich, was der Benutzer von seinem Computer primär wahrnimmt: die grafische Benutzeroberfläche mit all ihren Fensterchen, Menüs, Icons, Dialogboxen und vielem anderen mehr.

Heap

Heap (oder Halde) ist ein Speicherbereich, aus dem dynamisch (zur Laufzeit des Programms) Speicherblöcke angefordert werden können.

Information Hiding

Objekte kapseln ihre Information (u.a. private Membervariablen), indem nur klasseneigene Funktionen auf diese Variablen zugreifen oder sie verändern können.

Siehe auch Kapselung.

Initialisierung

Datenvariable in Instanzen werden beim Erzeugen der Instanz mit Default-Werten versehen. Numerische Daten werden mit 0, char-Variable mit 'u\0000', boolesche Größen mit false und Referenzvariable mit NULL vorbelegt. Eine Ausnahme bilden lokale Methodenvariablen, die innerhalb eines Methodenrumpfes deklariert werden. Diese werden nicht initialisiert.

inkrementieren

Eine Variable zu inkrementieren bedeutet, zu deren Wert eins hinzuzuzählen. Der Inkrement-Operator ist ++. Dieser Operator wird direkt nach oder direkt vor einen Variablennamen platziert.

Inline-Code

Der Compiler erzeugt die Anweisungen bei Inline-Code direkt an der Stelle, an der die Funktion aufgerufen wird, d.h. es erfolgt kein Sprungbefehl. Das kostet i.d.R. mehr Speicher, bringt aber bessere Performance.

Instanz

Dasselbe wie ein Objekt. Jedes Objekt ist eine Instanz einer Klasse.

Instanzmethoden

Befindet sich das Schlüsselwort `static` nicht vor dem Namen einer Methode, so wird diese zur Instanzmethode. Instanzmethoden beziehen sich immer auf ein konkretes Objekt anstatt auf die Klasse selbst. Da Instanzmethoden wesentlich häufiger verwendet werden als Klassenmethoden, werden Sie auch einfach nur als Methoden bezeichnet.

Instanzvariable

Eine Instanzvariable ist ein Stück Information, das ein Attribut eines Objekts definiert. Die Klasse des Objekts definiert die Art des Attributs, und jede Instanz speichert ihren eigenen Wert für dieses Attribut. Instanzvariablen werden auch als Objektvariablen bezeichnet.

Instanzvariablen gelten als solche, wenn sie außerhalb einer Methodendefinition deklariert werden. Üblicherweise werden die meisten Instanzvariablen direkt nach der ersten Zeile der Klassendefinition definiert.

Interface

siehe Schnittstelle

Introspection

siehe Reflection

iterativ

mehrere Male sich selbst wiederholend

JAR-Datei

Ein Java-Archiv, d.h. eine JAR-Datei, ist eine Sammlung von Java-Klassen oder anderen Dateien, die in eine einzige Archivdatei gepackt werden. JAR-Dateien eignen sich für die Verteilung und Installation von Java-Anwendungen und Klassenbibliotheken oder auch für einen schnelleren Download von Applets mit deren zugehörigen Dateien.

Kapselung

Wichtig für Objekte ist die Tatsache, dass diese Ihre Zustandsvariablen nach außen vor dem Benutzer verstecken. Ein Zugriff ist nur über die eigenen Objektmethoden zulässig. Damit ist das Objekt in der Lage, seine tatsächliche Realisierung vor dem Benutzer zu verbergen und

kann diese damit jederzeit abändern. Solange sich die Methodenschnittstelle nicht ändert, hat dies keine Konsequenzen.

Klasse

Eine Klasse ist eine Vorlage, die zur Erzeugung vieler Objekte mit ähnlichen Eigenschaften verwendet wird. Sie beinhaltet Variablen, um das Objekt zu beschreiben, und Methoden, um zu beschreiben, wie sich das Objekt verhält. Klassen können Variablen und Methoden von anderen Klassen erben.

Klassenbibliothek

Eine Klassenbibliothek ist eine Gruppe von Klassen, die zur Verwendung mit anderen Programmen entworfen wurden. Die Standard-Java-Klassenbibliothek beinhaltet Dutzende von Klassen.

Klassenmethoden

Eine Methode, die auf eine Klasse selbst angewendet wird und nicht auf eine bestimmte Instanz einer Klasse.

Klassenattribute

Ein Klassenattribut ist ein Stück Information, das ein Attribut einer Klasse definiert. Die Variable bezieht sich auf die Klasse selbst und all ihre Instanzen, so dass nur ein Wert gespeichert wird unabhängig davon, wie viele Objekte dieser Klasse erzeugt wurden. Sie definieren Klassenattribute, indem Sie das Schlüsselwort `static` vor die Variable setzen.

Kommentare

Eine der wichtigsten Methoden, die Lesbarkeit eines Programms zu verbessern, sind Kommentare. Kommentare sind Informationen, die in einem Programm einzig für den Nutzen eines menschlichen Betrachters eingefügt wurden, der versucht, herauszufinden, was das Programm tut. Der Java-Compiler ignoriert die Kommentare komplett, wenn er eine ausführbare Version der Java-Quelldatei erstellt.

Die erste Methode, einen Kommentar in einem Programm einzufügen, ist, dem Kommentartext zwei Schrägstriche (`//`) voranzustellen. Dadurch wird alles nach den Schrägstrichen bis zum Ende der Zeile zu einem Kommentar. Wenn Sie einen Kommentar einfügen wollen, der länger als eine Zeile ist, dann starten Sie den Kommentar mit der Zeichenfolge `/*` und beenden ihn mit `*/`. Alles zwischen diesen beiden Begrenzern wird als Kommentar gesehen.

Komponententechnologie

Hierbei handelt es sich um das Konzept der modularen Programmierung: Vorgefertigte Komponenten werden zu einer Anwendung zusammengesetzt, das Rad muss so nicht für jedes Programm neu erfunden werden. In Java steht hierfür das Konzept der Java-Beans.

Konstanten

Eine Konstante ist eine Variable, deren Wert sich nie ändert (was im Zusammenhang mit dem Wort »Variable« seltsam erscheinen mag). Konstanten sind sehr nützlich für die Definition von Werten, die allen Methoden eines Objekts zur Verfügung stehen sollen. Mit anderen Worten kann man mit Konstanten unveränderlichen, objektweit genutzten Werten einen aussagekräftigen Namen geben. Um eine Konstante zu definieren benutzen Sie das Schlüsselwort `final` vor der Variablendeklaration und geben für diese Variable einen Anfangswert an.

Konstruktor

Ein Konstruktor ist eine spezielle Methode zum Erstellen und Initialisieren neuer Instanzen von Klassen. Konstruktoren initialisieren das neue Objekt und seine Variablen, erzeugen andere Objekte, die dieses Objekt braucht, und führen im Allgemeinen andere Operationen aus, die für die Ausführung des Objekts nötig sind. Konstruktoren werden statt in Zusammenhang mit dem Schlüsselwort `new` aufgerufen.

Literale

Neben Variablen werden Sie in Java-Anweisungen auch Literale verwenden. Literale sind Zahlen, Text oder andere Informationen, die direkt einen Wert darstellen.

Locking

Sperren von Objekten, so dass nur ein Prozess, statt mehrerer gleichzeitig, auf bestimmte Daten zugreifen kann; wichtig für Datenkonsistenz.

Methoden

Methoden sind Gruppen von miteinander in Beziehung stehenden Anweisungen in einer Klasse. Diese Anweisungen beziehen sich auf die eigene Klasse und auf andere Klassen und Objekte. Sie werden verwendet, um bestimmte Aufgaben zu erledigen, wie das in anderen Programmiersprachen Funktionen tun.

Objekte kommunizieren miteinander über Methoden. Der Aufruf von Methoden besteht aus dem Namen der Methode und Klammern. Die Klammern dürfen auf keinen Fall weggelassen werden. Die Klammern können leer bleiben oder einen oder mehrere Parameter enthalten.

new

Um ein neues Objekt einer Klasse zu erstellen, benutzen Sie den Operator `new` und einen Konstruktor, der aus dem Namen der Klasse, von der Sie eine Instanz anlegen wollen, und Klammern besteht. Die Klammern dürfen auf keinen Fall weggelassen werden. Die Klammern können leer bleiben oder Argumente enthalten, die die Anfangswerte von Instanzvariablen oder andere Anfangsqualitäten des Objekts bestimmen.

Objekt

Ein Objekt ist ein abgeschlossenes Element eines Computerprogramms, das eine Gruppe miteinander verwandter Features darstellt und dafür ausgelegt ist, bestimmte Aufgaben zu erfüllen. Objekte werden auch als Instanzen einer Klasse bezeichnet. Mehrere Objekte, die Instanzen derselben Klasse sind, haben Zugriff auf dieselben Methoden, aber oft unterschiedliche Werte für deren Instanzvariablen.

objektorientiert

Als objektorientierte Programmierung – auch OOP genannt – wird eine Vorgangsweise bezeichnet, bei der Computerprogramme als eine Reihe von Objekten aufgebaut werden, die miteinander interagieren. Im Wesentlichen ist es eine bestimmte Methode, Computerprogramme zu organisieren.

Objektvariable

siehe Instanzvariable

Operatoren

Operatoren sind spezielle Symbole, die für mathematische Funktionen, bestimmte Zuweisungsarten und logische Vergleiche stehen.

Operatorpräzedenz

Unter Operatorpräzedenz versteht man die Priorität der einzelnen Operatoren untereinander. Operatoren mit hoher Präzedenz werden vor solchen Operatoren mit niedriger Präzedenz abgearbeitet, d.h. dass bei einem klammerfreien Ausdruck sich die Abarbeitungsreihenfolge nach dieser Präzedenz richtet.

siehe auch Assoziativität

overloading

siehe Überladen

Pakete

Pakete (engl. Packages) sind eine Möglichkeit, um verwandte Klassen und Schnittstellen zu gruppieren. Pakete ermöglichen es, dass Gruppen von Klassen nur bei Bedarf verfügbar sind. Klassen, die sich nicht in dem Paket `java.lang` befinden, müssen explizit importiert oder direkt über den vollen Paket- und Klassennamen angesprochen werden.

parallel

In den meisten Programmiersprachen werden die Programmteile nacheinander, also sequentiell abgearbeitet. Java erlaubt nun mit Hilfe von Threads die Realisierung von Programmeinheiten, die parallel bzw. quasiparallel ablaufen.

siehe auch Threads

Pointer

siehe Referenz

Postfix-Operatoren

Inkrement- und Dekrement-Operatoren werden als Postfix-Operatoren bezeichnet, wenn sie sich hinter dem Variablennamen befinden.

Präfix-Operatoren

Inkrement- und Dekrement-Operatoren werden als Präfix-Operatoren bezeichnet, wenn sie vor dem Namen der Variablen aufgeführt werden.

Punkt-Notation

Die Punkt-Notation ist eine Methode, um auf Instanzvariablen und Methoden innerhalb eines Objekts mit einem Punkt-Operator (».«) zuzugreifen.

Referenz

Eine Referenz ist eine Art Zeiger (Pointer), der auf ein Objekt verweist. Wenn Sie Objekte Variablen zuweisen oder Objekte als Argumente an Methoden weiterreichen, legen Sie Referenzen auf diese Objekte fest. Die Objekte selbst oder Kopien davon werden dabei nicht weitergereicht.

Wenn Sie eine Methode mit Objektparametern aufrufen, werden die Variablen, die Sie an den Körper der Methode übergeben, als Referenz übergeben. Das bedeutet, dass sich alles, was Sie mit diesen Objekten in der Methode anstellen, gleichermaßen auf die Originalobjekte auswirkt.

Reflection

Reflection bzw. Introspection ermöglicht es einer Java-Klasse – beispielsweise einem von Ihnen geschriebenen Programm – Details über eine beliebige andere Klasse zu ermitteln.

Mit Reflection kann ein Java-Programm zur Laufzeit eine Klasse laden, von der es zur Übersetzungszeit noch nichts weiß, die Variablen, Methoden und Konstruktoren dieser Klasse ermitteln und damit arbeiten.

Round-Robin-Strategie

Hierbei handelt es sich um ein Prinzip der Zuteilung von Prozessorzeit an die Threads. First come, first serve oder Ringelreihenprinzip oder FIFO (first in – first out) genannt. Ausnahme: Threads mit höherer Priorität verdrängen die mit niederer.

Scheduler

Der Java-Scheduler ist der Teil des Java-Interpreters, der bestimmt, welcher Prozess wann Prozessorzeit zugewiesen bekommt.

Schleife

Eine Schleife wiederholt eine Anweisung oder einen Anweisungsblock mehrmals, bis oder solange eine Bedingung zutrifft. for-Schleifen werden häufig für einfache Wiederholungen verwendet, um Blockanweisungen mehrmals auszuführen und dann zu stoppen. Die while-Schleife wird zum Wiederholen einer Anweisung oder von Blockanweisungen benutzt, solange eine bestimmte Bedingung zutrifft. Die do-Schleife entspricht der while-Schleife, außer dass sie eine bestimmte Anweisung oder einen Block so oft ausführt, bis eine Bedingung false ergibt.

Schnittstelle

Eine Schnittstelle (engl. Interface) ist eine Sammlung von Methoden, die benannt aber nicht unbedingt implementiert (default-Implementierung) sind. Dadurch wird angezeigt, dass eine Klasse neben dem aus der Superklasse geerbten Verhalten noch zusätzliche Verhaltensweisen hat.

Semantik

Die Semantik oder Wortbedeutungslehre beeinflusst, wie Programme konzipiert, von anderen verstanden und vom Rechner ausgeführt werden: wie verhält sich ein Programm, wenn es auf einem Rechner ausgeführt wird.

Signatur

Die Definition von Methoden besteht aus folgenden vier Teilen: (1) Name der Methode, (2) Objekttyp oder der primitive Typ, den die Methode zurückgibt, (3) Liste der Parameter und (4) Methodenrumpf. Die ersten drei Teile bilden die sogenannte Signatur der Methode (polymorphe Methoden können allerdings in den Parametern variieren, aber nicht in den Rückgabetypen).

Statement

siehe Anweisung

statisch

Die Bezeichnung statisch (über das Schlüsselwort static) für eine Variablen bezieht sich auf eine Bedeutung des Wortes: ortsfest. Wenn eine Klasse eine statische Variable besitzt, dann hat diese Variable in jedem Objekt dieser Klasse denselben Wert.

Strings

siehe Zeichenketten

Subklasse

Eine Klasse, die sich in der Klassenhierarchie weiter unten befindet als eine andere Klasse, ihre Superklasse.

Superklasse

Eine Klasse, von der andere Klassen abgeleitet werden (sprich, die ihre Funktionalität an andere Klassen vererbt), wird Superklasse genannt. Die Klasse, die die Funktionalität erbt, wird als Subklasse bezeichnet. Eine Klasse kann lediglich eine Superklasse besitzen. Jede Klasse kann allerdings eine uneingeschränkte Anzahl von Subklassen haben. Subklassen erben alle Attribute und sämtliche Verhaltensweisen ihrer Superklasse.

Syntax

Die Syntax legt fest, wie Programme geschrieben werden müssen, damit sie vom Rechner korrekt erkannt werden: wie werden Ausdrücke, Befehle, Vereinbarungen usw. zusammensetzt, um ein korrektes Programm zu bilden.

this

Um auf das aktuelle Objekt einer Klasse Bezug zu nehmen, können Sie das Schlüsselwort `this` verwenden. Da `this` eine Referenz auf die aktuelle Instanz einer Klasse ist, ist es sinnvoll, das Schlüsselwort nur innerhalb der Definition einer Instanzmethode zu verwenden. Klassenmethoden, d.h. Methoden, die mit dem Schlüsselwort `static` deklariert sind, können `this` nicht verwenden.

Thread

Ein Thread ist ein Teil eines Programms, der eingerichtet wird, um eigenständig zu laufen, während der Rest des Programms etwas anderes tut. Dies wird auch als Multitasking bezeichnet, da das Programm mehr als eine Aufgabe zur selben Zeit ausführen kann. Threads sind ideal für alles, was viel Rechenzeit in Anspruch nimmt und kontinuierlich ausgeführt wird, wie z.B. die wiederholten Zeichenoperationen, die eine Animation ausmachen. Indem Sie die Arbeitslast der Animation in einen Thread packen, machen Sie den Weg dafür frei, dass sich der Rest des Programms mit anderen Dingen beschäftigen kann.

Überladen

Methoden mit demselben Namen werden durch zwei Dinge voneinander unterschieden: (1) Die Anzahl der Argumente, die ihnen übergeben wird und (2) den Datentyp oder Objekttyp der einzelnen Argumente. Diese beiden Dinge bilden die Signatur einer Methode. Mehrere Methoden zu verwenden, die alle denselben Namen, aber unterschiedliche Signaturen haben, wird als Überladen (engl. *overloading*) bezeichnet.

Überschreiben

Zuweilen soll ein Objekt auf die gleichen Methoden reagieren, jedoch beim Aufrufen der jeweiligen Methode ein anderes Verhalten aufweisen. In diesem Fall können Sie die Methode überschreiben.

Durch Überschreiben von Methoden definieren Sie eine Methode in einer Subklasse, die die gleiche Signatur hat wie eine Methode in einer

Superklasse. Dann wird zum Zeitpunkt des Aufrufs nicht die Methode in der Superklasse, sondern die in der Subklasse ermittelt und ausgeführt.

Normalerweise gibt es zwei Gründe dafür, warum man eine Methode, die in einer Superklasse bereits implementiert ist, überschreiben will: (1) Um die Definition der Originalmethode völlig zu ersetzen oder (2) um die Originalmethode zu erweitern.

Unicode-Zeichensatz

Java unterstützt auch den Unicode-Zeichensatz, der den Standardzeichensatz plus Tausende anderer Zeichen beinhaltet, um internationale Alphabete zu repräsentieren. Zeichen mit Akzenten und andere Symbole können in Variablennamen verwendet werden, solange diese über eine Unicode-Nummer oberhalb von 00C0 verfügen.

Siehe auch Alan Wood's Unicode Resources
oder Unicode Organisation (www.unicode.org)

Variablen

Variablen sind Orte, an denen, während ein Programm läuft, Informationen gespeichert werden können. Der Wert der Variablen kann unter deren Namen von jedem Punkt im Programm aus geändert werden. Um eine Variable zu erstellen, müssen Sie dieser einen Namen geben und festlegen, welchen Typ von Informationen sie speichern soll. Sie können einer Variablen auch gleich bei der Erzeugung einen Wert zuweisen.

Siehe auch Instanzvariable und Klassenvariable

Vererbung

Vererbung ist ein Mechanismus, der es einer Klasse ermöglicht, all Ihre Verhaltensweisen und Attribute von einer anderen Klasse zu erben. Über die Vererbung verfügt eine Klasse sofort über die gesamte Funktionalität einer vorhandenen Klasse.

Siehe auch Subklasse und Superklasse

Zeichenketten

Zeichenketten sind wie Felder Objekte und damit ein Referenzdatentyp. Diese Objekte erlauben den effizienten Umgang mit Zeichenketten. Zu beachten ist, dass es zwei unterschiedliche Arten von Zeichenketten in Java gibt. Zum einen stellt Java Objekte der Klasse String – Zeichenketten mit fester Länge – und zum anderen Objekte der Klassen StringBuffer und StringBuilder – Zeichenketten mit variabler Länge während der Laufzeit des Programms – zur Verfügung.

Gesamtindex

A

Abfrage 7-18, 7-20
ActionEvent 5-21, 5-22
addActionListener 5-21
addWindowListener 5-21, 5-27
Applet 5-9
Array 1-23
Arrays 1-6, 1-23, 1-24
Attribute 2-5, 2-25
Ausgabe 1-13, 1-14, 2-3, 2-4
Ausnahmen 1-9
awt 5-3, 5-5, 5-7, 5-9, 5-10, 5-20, 5-21

B

boolean 1-9, 1-10, 1-11, 1-12, 2-4, 2-5
BorderLayout 5-5, 5-10, 5-12
Button 5-5, 5-6, 5-12
Bytecode 1-5, 2-13, 2-15, 2-16, 2-17, 4-12, 7-10

C

CallableStatement 7-15, 7-18, 7-19
cast 1-22
catch 2-9, 7-10, 7-13, 7-14, 7-17
char 1-9, 1-11, 1-12, 1-13
Class 2-25, 4-12, 7-10
CLASSPATH 1-5, 7-10
clone 2-5, 2-6
Collection-Framework 3-3, 3-4, 3-19
Collections 3-4, 3-19
Collections-Framework 3-3
Color 5-5
Comparable 3-18, 3-19
Connection 7-13, 7-15, 7-19
Constructor 2-25
Container 1-4, 1-6, 5-5, 5-7, 5-9, 5-10, 5-13, 5-17, 5-24

D

Dämon 6-13

Datenbank 7-3, 7-5, 7-7, 7-10, 7-11, 7-12, 7-13, 7-14, 7-16, 7-19
Datenbank-URL 7-11, 7-12
Datentypen 1-9
double 1-9, 1-11, 1-12
Drei-Schicht-Modell 7-3
Driver 7-9
DriverManager 7-10, 7-13, 7-14

E

else 2-3, 2-4
Entwurfsmuster 1-10
equals 1-11, 1-12, 2-3, 2-4
Event 2-17, 5-20, 5-21, 5-22
Exception 2-5, 2-8, 2-10, 4-10, 7-13

F

Factory 1-10
false 1-13, 2-4
Farben 5-5
Field 2-25
finally 2-9, 7-14, 7-17, 7-21
flache Kopie 2-5
float 1-11
FlowLayout 5-5, 5-10, 5-11
Font 5-5
forName 2-25, 7-10
Frame 5-7
Framework 3-3, 3-4

G

Graphics 5-5
GridLayout 5-5, 5-10, 5-15

H

Heap-Speicher 1-6

I

if 2-3, 2-4
import 1-18
instanceof 2-4

Interface 1-7, 3-19, 5-3, 5-16, 5-21, 6-5,
6-15, 7-9, 7-17

interrupt 6-11

Introspektion 2-25

J

javadoc 8-9, 8-10

JDBC 7-3, 7-5, 7-7, 7-8, 7-9, 7-13, 7-18
Treiber 7-5

JDK 1-3, 2-25, 6-15, 8-9

join 6-10, 6-11

K

Kapselung 1-8

Klasse 1-9, 1-10, 1-11, 1-13, 1-17, 1-18,
2-3, 2-4, 2-5, 2-13, 2-15, 2-16, 2-17,
2-25, 3-4, 3-18

Konstruktoren 2-25

L

LayoutManager 5-5, 5-16

List 3-3, 3-19, 5-5

Listener 5-19, 5-21, 5-22, 5-23, 5-24, 5-25,
5-26

long 1-11

M

main 1-12, 1-14

Map 3-4

Method 2-25

Methoden 1-9, 1-11, 1-12, 1-13, 1-14, 2-3,
2-25

Methoden-Frame 1-6

N

Netscape 5-6

new 1-11, 1-12, 1-13, 1-14, 2-3, 2-4, 2-26

newInstance 2-25

null 2-4

O

Object 1-11, 2-3, 2-4, 2-5, 2-25, 3-18, 3-19

Objekt 1-10, 1-11, 2-25, 2-26

öffentlich 1-16

Online-Dokumentation 8-9

P

package 1-17

Paket 1-17, 1-18, 2-5, 2-26

Panel 5-15

PreparedStatement 7-15, 7-18, 7-19

private 1-8, 2-14, 8-10

Property 4-12, 6-12, 7-13

protected 1-8, 1-16, 2-5, 2-14

public 1-8, 1-12, 1-14, 1-16, 2-4, 2-5, 2-6,
2-14, 5-7, 5-21, 5-27, 6-5, 6-10, 6-12,
6-13

Q

Quellcode 1-17

R

Referenz 1-10, 2-4

Referenzen 3-19, 5-12

Reflection 2-25

ResultSet 7-20, 7-21

return 2-4, 2-5, 2-6

rt.jar 1-5

run 6-5, 6-6, 6-10, 6-11, 6-14, 6-15, 6-16

Runnable 6-5, 6-7, 6-15

RuntimeException 6-11, 6-14

S

Schlüsselwörter 2-14

Set 3-3, 7-13

short 1-9

Singleton 1-10

sleep 6-10, 6-11

SQL 1-3, 7-15, 7-16, 7-17, 7-18, 7-19

SQLException 7-13

start 6-6, 6-10, 6-11

Statement 7-15, 7-16, 7-17, 7-18, 7-19,
7-21

static 1-12, 1-14, 2-25, 3-19

stop 6-14

String 1-10, 1-11, 1-12, 1-13, 1-14, 2-5,
2-25

StringBuffer 1-10, 1-13

StringBuilder 1-10, 1-11, 1-13, 1-14

Subklasse 2-5

Swing 5-3, 5-6, 5-7, 5-8

System.out.println 1-12, 1-14, 2-3, 2-4

T

this 2-4, 2-5, 2-6, 2-15

Thread 6-3, 6-5, 6-6, 6-10, 6-11, 6-12,
6-13, 6-14

Three-Tier-Model 7-3
throw 2-8
throws 2-5, 2-6, 2-8, 6-10
tiefe Kopie 2-5
Timer 6-15, 6-16
TimerTask 6-15, 6-16
toString 1-12, 1-13, 1-14, 2-5
TreeMap 3-19
TreeSet 3-19
Treiber 7-5, 7-7, 7-9, 7-10, 7-11, 7-12, 7-13
true 1-11, 1-13, 2-4
try 2-9, 7-10, 7-13, 7-14, 7-17, 7-21
Two-Tier-Model 7-3
Typumwandlung 1-22

U

Update 7-18
URL 7-11, 7-12, 7-13

V

Vererbung 1-7
Vergleich 1-11, 1-12, 1-13, 2-3, 2-4
Virtuelle Maschine 1-10
void 1-12, 1-13, 1-14, 3-19

W

WindowEvent 5-27
Wrapper 1-9, 1-10
Wrapper-Klassen 1-9

Y

yield 6-10, 6-11

Z

Zeichenketten 1-10
Zugriffsrechte 1-16
Zuweisung 1-10
Zwei-Schicht-Modell 7-3

