

Inhalt

11	Dynamische Verwaltung großer Datenmengen	11-2
11.1	Einige spezielle Klassen des Paketes <code>java.lang.*</code>	11-2
11.1.1	Klasse <code>Throwable</code> , <code>Exception</code> und <code>Error</code>	11-2
11.1.2	Klasse <code>Object</code>	11-7
11.1.3	Wrapper-Klassen	11-7
11.2	Abstrakte Klassen und Interface	11-9
11.2.1	Abstrakte Klassen	11-9
11.2.2	Interface	11-10
11.3	Collection-Klassen <code>java.util.*</code>	11-12
11.3.1	Generische Datentypen - Template	11-13
11.3.2	Interface <code>Collection<E></code>	11-13
11.3.3	Interface <code>Iterator<E></code>	11-14
11.3.4	Interface <code>Set<E></code>	11-15
11.3.5	Interface <code>List<E></code>	11-16
11.4	Klassen <code>Arrays</code> und <code>Collections</code>	11-22
11.5	Zusammenfassung	11-22

11 Dynamische Verwaltung großer Datenmengen

11.1 Einige spezielle Klassen des Paketes `java.lang.*`

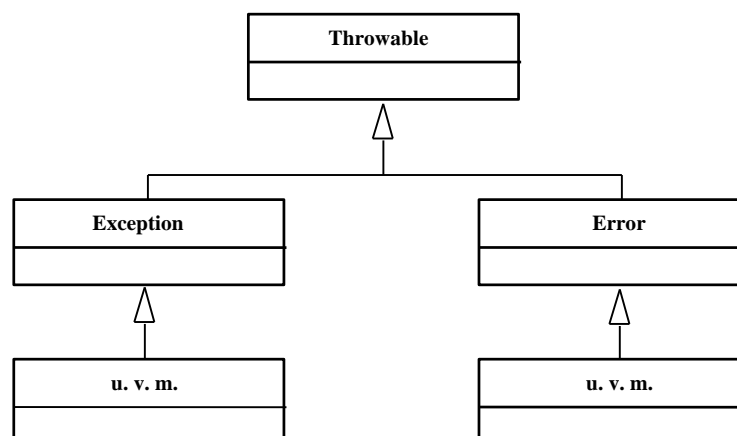
11.1.1 Klasse `Throwable`, `Exception` und `Error`

Kritische Ereignisse, die zu einem Programmabbruch führen können, werden als *Ausnahmen* (**Exceptions**) oder als *Fehler* (**Errors**) behandelt.

Mit **Exceptions** werden sogenannte „*leichte Fehler*“ bezeichnet. Das sind Fehler, die eine Rückkehr in einen definierten Programmablauf zulassen, wie zum Beispiel eine *Division durch Null*. Der Programmierer hat es dabei in der Hand zu entscheiden, was bei ihrem Auftreten zu geschehen hat.

Als **Errors** hingegen werden „*schwerwiegende Probleme*“ eingestuft, die eine weitere Ausführung des Programms unmöglich oder zumindest nicht mehr sinnvoll machen. Das sind zum Beispiel interne Fehler des Interpreters, mangelnder Speicherplatz bei der dynamischen Verwaltung der Objekte, fehlende Klassendefinitionen u. a. m.

Exceptions und *Errors* sind Objekte, die in Unterklassen der Klasse **Throwable** definiert sind.



Auf die Klassen **Error** soll hier *nicht* weiter eingegangen werden, da deren Behandlung meistens nicht sinnvoll ist. Man sollte der virtuellen Maschine das Beenden der Anwendung überlassen.

Die Klasse **Exception** und von ihr abgeleitete Klassen verwalten *Ausnahmesituationen*, deren Behandlung im Programm festgelegt werden sollte, um vorzeitige Abbrüche zu vermeiden. Tritt eine solche Ausnahmesituation ein, so wird ein Objekt einer dieser Klassen erzeugt. Die Ausführung des Programms wird unterbrochen und die im Programm festgelegte Fehlerbehandlung ausgelöst. Dieser Vorgang wird als *Werfen einer Ausnahme* bezeichnet (*throw Exception*). Die ausgelöste Behandlung wird im Gegenzug als *Abfangen einer Ausnahme* (*catch Exception*) bezeichnet.

try-catch-finally-Block

Ausnahmesituationen werden unmittelbar in der Methode abgefangen, in der sie auftreten.

```

try
{
    Anweisungen // zu schützende Anweisungen
}
catch ( Exception e) // e ist abgefangene Ausnahme
{
    Anweisungen // Reaktion auf die Ausnahme
}
finally // auf jeden Fall auszuführende Anweisungen, optional
{
    Anweisungen // unabhängig von Ausnahmen
}

```

- Innerhalb des **try**-Blockes stehen die Anweisungen, die geschützt werden sollen. Während der Laufzeit führen Ausnahmen und Fehler dieser *nicht* direkt zum Abbruch des Programms.
- Innerhalb des **catch**-Blockes stehen die Anweisungen, die bei einem Fehler innerhalb der **try**-Blockes ausgeführt werden.
- Schließlich kann noch ein **finally**-Block mit Anweisungen, die auf jeden Fall ausgeführt werden, existieren.

Durch diese Vorgehensweise kann zum Beispiel eine *Division durch Null* für ganze Zahlen wie folgt abgefangen werden:

Div1.java

```

// Div1.java
import Tools.IO.*;
/**
 * ArithmeticException, Division durch Null,
 * wird von der werfenden Methode abgefangen.
 */
public class Div1
{
    public static void main( String[] args)
    {
        int a = IOTools.readInteger( "Zaehler ");
        int b = IOTools.readInteger( "Nenner ");
        try
        {
            // zu schuetzende Anweisung
            System.out.println( "a/b = " + ( a / b ));
            System.out.println( "div: OK");
        }
        catch( Exception e)
        {
            // Reaktion auf Ausnahmen allgemein
            System.err.println( "FEHLER! " + e);
        }
        System.out.println( "main: OK");
    }
}

```

MM 2007

Eine *Division durch Null* ist für Gleitpunktzahlen möglich und muss deshalb *nicht unbedingt* abgefangen werden. Je nach Wert des Zählers erhält man **-Infinity**, **NaN** oder **Infinity**:

Div2.java

```
// Div2.java
import Tools.IO.*;

/**
 * Double: Division durch Null liefert
 * - Infinity, falls Zaehler < 0,
 *      NaN, falls Zaehler = 0,
 * + Infinity, falls Zaehler > 0.
 */
public class Div2
{
    public static void main( String[] args)
    {
        double a = IOTools.readDouble( "Zaehler ");
        double b = IOTools.readDouble( "Nenner ");
        System.out.println( "a/b = " + ( a / b));
        System.out.println( "main: OK");
    }
}
```

MM 2007

Um eine *Division durch Null* auszuschließen, muss man diesen Fall selber abfangen. Dazu verwendet man eine bereits definierte *Unterklasse der Klasse Exception* oder man definiert man eine neue *Unterklasse*.

Unterklassen der Klasse Exception

```
class java.lang.Exception
  o class java.lang.ClassNotFoundException
  o class java.lang.CloneNotSupportedException
  o class java.lang.IllegalAccessException
  o class java.lang.InstantiationException
  o class java.lang.InterruptedException
  o class java.lang.NoSuchFieldException
  o class java.lang.NoSuchMethodException
  o class java.lang.RuntimeException
    o class java.lang.ArithmeticException
    o class java.lang.ArrayStoreException
    o class java.lang.ClassCastException
    o class java.lang.IllegalArgumentException
      o class java.lang.IllegalThreadStateException
      o class java.lang.NumberFormatException
    o class java.lang.IllegalMonitorStateException
    o class java.lang.IllegalStateException
    o class java.lang.IndexOutOfBoundsException
      o class java.lang.ArrayIndexOutOfBoundsException
      o class java.lang.StringIndexOutOfBoundsException
    o class java.lang.NegativeArraySizeException
```

- `class java.lang.NullPointerException`
- `class java.lang.SecurityException`
- `class java.lang.UnsupportedOperationException`

Definition einer neuen *Exception*

Zuletzt sei noch erwähnt, dass sich beliebige *eigene* *Exception* durch Vererbung aus diesen *Exception*-Klassen erzeugen lassen.

EigeneException.java

```
public class EigeneException extends Exception
{
    public String toString()
    {
        return "Kommentar";
    }
}
```

Um eine Division durch Null auch für Gleitpunktzahlen auszuschließen, definieren wir eine neue Ausnahme **NennerException**:

NennerException.java

```
// NennerException.java                                MM 2006

/**
 * Definition der Ausnahme NennerException.
 */
public class NennerException extends Exception
{
    public String toString()
    {
        return "NennerException, Division durch Null: /0";
    }
}
```

Jetzt besteht mittels einer ***throws-Klausel*** die Möglichkeit, eine Division durch Null auch für Gleitpunktzahlen zu unterbinden:

throws-Klausel

Ausnahmesituationen werden von der aufrufenden Methode abgefangen. Eine eingetretene *Exception* wird an die aufrufende Methode weitergeleitet.

```
Methodenkopf throws Exception                        // Weiterleiten der Ausnahme
{
    Anweisungen                                     // zu schützenden Anweisungen
}
```

Div3.java

MM 2007

```
// Div3.java
import Tools.IO.*;

/**
 * Double: Division durch Null
 * erzeugt ein Objekt der Klasse NennerException,
 * welches von der werfenden Methode weitergeleitet
 * und von der aufrufenden Methode abgefangen wird.
 */

public class Div3
{
    /**
     * Testprogramm.
     */
    public static void main( String[] args)
    {
        double a = IOTools.readDouble( "Zaehler ");
        double b = IOTools.readDouble( "Nenner ");
        try
        {
            // zu schuetzende Anweisung
            System.out.println( "a/b = " + div( a, b));
            System.out.println( "div: OK");
        }
        catch( NennerException ne)
        {
            // Reaktion auf Nennerexception
            System.err.println( "FEHLER! " + ne);
        }
        catch( Exception e)
        {
            // Reaktion auf Ausnahmen allgemein
            System.err.println( "FEHLER! " + e);
        }
        System.out.println( "main: OK");
    }

    /**
     * Methode mit Ausnahme, diese wird weitergeleitet.
     */
    public static double div( double a, double b)
        throws NennerException
    {
        // Ausnahmeobjekt wird erzeugt
        if( b == 0) throw new NennerException();
        return a / b;
    }
}
```

Als weiteres Beispiel sei auf die Klasse Euklid hingewiesen, wie im Kapitel über Methoden bereits erläutert. Dort wird ein Fehler `NPlusException` geworfen, falls die Klassenmethoden `Euklid.ggt` bzw. `Euklid.kgV` mit nicht natürlichen Zahlen aufgerufen werden.

11.1.2 Klasse Object

Jede Klasse, welche *nicht* mit dem Zusatz **extends** von einer anderen Klasse abgeleitet wird, erhält automatisch die Klasse **Object** als *Superklasse* zugeordnet.

⇒ *Jede Klasse ist direkt oder indirekt von der Klasse Object abgeleitet.*

Methoden (Auswahl) der Klasse **Object**:

Name	Parameter-anzahl	Parameter-typ	Ergebnis-Typ	Beschreibung
toString	0		String	beschreibt Objekt
equals	1	Object	boolean	vergleicht auf Gleichheit
hashCode	0		int	berechnet Hash-Code eines Objekts
getClass	0		Class	ermittelt Klasse eines Objekt

Ein Überschreiben der Methode **equals** erfordert in der Regel ein *Überschreiben* der Methode **hashCode**. Der **Hash-Code** eines Objekts wird in speziellen Tabellen abgespeichert und verwendet, um das Objekt im Speicher abzulegen und schnell wieder aufzufinden. *Gleiche* Objekte werden nur einmal abgespeichert, folglich sollte ihr Hash-Code *derselbe* sein.

11.1.3 Wrapper-Klassen

Einfache Datentypen (**byte**, **boolean**, **int**, **double**, ...) sind *keine* Objekte einer Klasse. Um auch diese als Objekte behandeln zu können, stellt Java sogenannte **Wrapper-Klassen** (Hüllklassen) zur Verfügung. Es sind Klassen, die dem entsprechenden Datentyp ein Objekt zuweisen.

Datentyp	Wrapper-Klasse
boolean	Boolean
byte	Byte
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short
void	Void

Hüllklassen verfügen über viele nützliche *Klassenkonstanten* und *Klassenmethoden* zum Beispiel zur Datentypkonvertierung. Die Klassenmethode **Integer.parseInt(String)**; wandelt eine Zeichenkette vom Typ **String** in eine ganze Zahl vom Typ **int** um.

Seit Java 1.5 geschieht die Umwandlung (Wrappen) zwischen *Elementardatentypen* und *Objekten der zugehörigen Hüllklassen* in beiden Richtungen automatisch (*Autoboxing/Auto-unboxing*).

```
int a = Integer.parseInt( "10");      // String -> int
Integer b = a;                        // int -> Integer
int c = b;                            // Integer-> int

System.out.println( a + " " + b + " " + c); // 10 10 10
```

11.2 Abstrakte Klassen und Interface

11.2.1 Abstrakte Klassen

Abstrakte Klassen **abstract class** werden, wie schon erwähnt, im Sinn der *Generalisierung* als Basisklassen eingeführt. Von ihr werden *keine* Instanzen gebildet. Diese Klasse der obersten Ebene (Wurzel des Hierarchiebaumes) deklarieren *Attribute* oder/und *Methoden*, die von allen Subklassen ererbt werden. Methoden abstrakter Klassen können implementiert werden oder sind selbst **abstract**, d. h. ihre Schnittstelle wird mitgeteilt, aber der Methodenrumpf ist leer. Abgeleitete Klassen konkretisieren diesen.

Deklaration einer abstrakten Klasse

```
public abstract class Klassenname
{
    Deklarationen
    Deklarationen_abstrakter_Methode
    Implementieren_eigener_Methoden
}
```

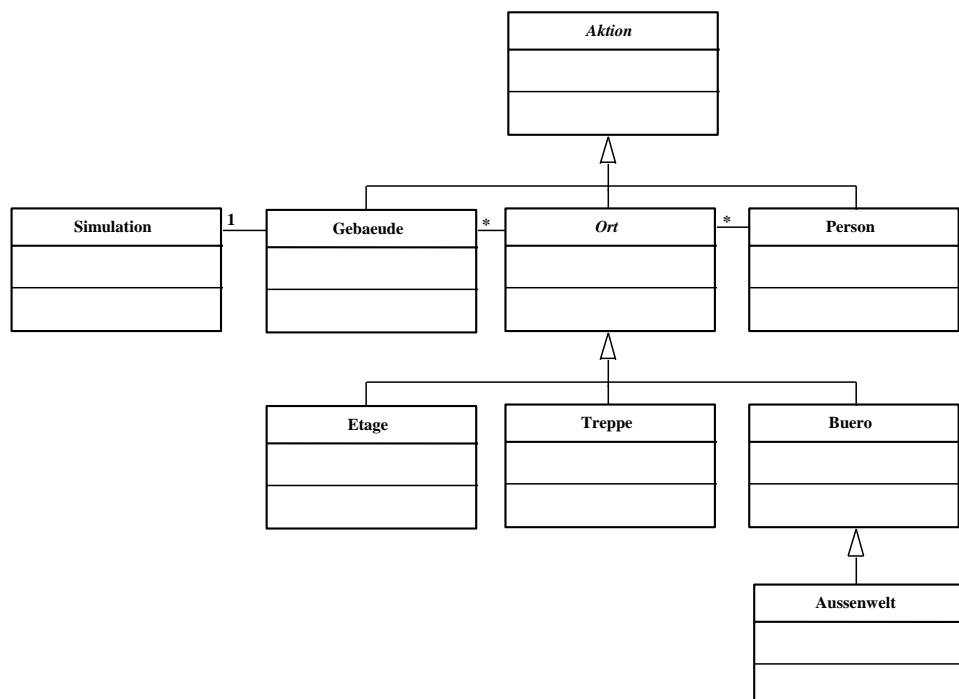
Deklaration abstrakter Methoden

```
public abstract Ergebnistyp Methodename ( Parameterliste );
```

Deklaration einer von einer abstrakten Klasse abgeleiteten Klasse

```
public class Subklassenname extends Superklassenname
{
    Deklarationen
    Implementieren_aller_abstrakten_Methoden
    Implementieren_eigener_Methoden
}
```

Im UML-Klassendiagramm werden abstrakte Klassen und abstrakte Methoden kursiv dargestellt.



In einer Gebäudesimulation werden zwei Klassen als **abstract** festgelegt:
Eine Klasse **Aktion** mit abstrakten Methoden **zeitVergeht** und **aktion** dient der Taktung der Gebäudesimulation. Die Klasse **Ort** verwaltet Personen, ihr Kommen und Gehen.

11.2.2 Interface

Ein **Interface** ist eine *spezielle abstrakte Klasse*, die die Beschreibung einer Klasse generell von ihrer Implementierung trennt. Es besitzt nur *Klassenkonstanten* und *abstrakte Methoden*, d.h. alle Methodenrumpfe sind leer. Dabei können die Schlüsselwörter **static final** für die Klassenkonstanten bzw. **abstract** für die abstrakten Methoden weggelassen werden. Ein Interface dient also ausschließlich der *Schnittstellenvorgabe*.

Deklaration eines Interface

```
public interface Klassenname
{
    Deklarationen_Klassenkonstanten
    Deklarationen_Interfacemethoden
}
```

Deklaration von Interfacemethoden

```
public Ergebnistyp Methodennamen ( Parameterliste );
```

Wird eine Klassen von einem Interface abgeleitet, verwendet man statt **extends** das Schlüsselwort **implements**. Klassen, die das Interface nutzen, *müssen alle* Methoden des Interface überschreiben.

Deklaration einer von einem Interface abgeleiteten Klasse

```
public class Subklassenname implements Interfacenamen
{
    Deklarationen
    Implementieren_aller_Interfacemethoden
    Implementieren_eigener_Methoden
}
```

Die Klasse **Aktion** erfüllt die Bedingung eines Interface. In ihr sind nur zwei Methoden deklariert und diese sind abstrakt. Deshalb ließe sich die Klasse auch als Interface festlegen:

Aktion.java als Interface

```
// Aktion.java MM 2003

/**
 * Interface zur Taktung aller beteiligten Objekte
 */
public interface Aktion
{
    public void zeitVergeht();
    public void aktion();
}
```

Alle von **Aktion** abgeleiteten Klassen implementieren die Methoden **zeitVergeht** und **aktion**.

Die Klasse **Ort** verwaltet Personen, die sich an- und abmelden. Dazu besitzt er Instanzvariablen, um alle Personen an einem Ort zu speichern, und Instanzmethoden, um Personen an einem Ort ab- bzw. anzumelden. Diese Klasse ist zwar abstrakt, erfüllt aber nicht die Interface-Bedingungen.

Im Gegensatz zu normalen Klassen haben Interfaces jedoch einen wesentlichen Vorteil. Sie erlauben **Mehrfachvererbung**: Eine Klasse darf zwar nur eine Superklasse besitzen, sie darf aber zusätzlich *beliebig viele Interfaces* implementieren.

Deklaration einer Subklasse mit Superklasse und Interfaces

```
public class Subklassenname
  extends Superklassenname
  implements Interfacename ...
{
  Deklarationen
  Implementieren_aller_abstrakten_Methoden
  Implementieren_aller_Interfacemethoden
  Implementieren_eigener_Methoden
}
```

Bei der *Mehrfachvererbung* können normalerweise *Konflikte* auftreten, in der Art, dass Methoden oder Attribute mit dem gleichen Namen in beiden Superklassen unterschiedlich implementiert wurden. Diese Konflikte werden bei Interfaces als Superklassen vermieden, da diese *keine* Attribute und *keine* implementierten Methoden besitzen.

11.3 Collection-Klassen `java.util.*`

Statische Felder (Kapitel 5)

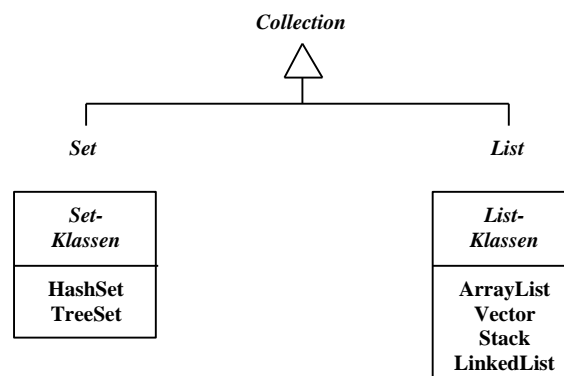
Gewöhnliche Felder sind Referenzdatentypen. Diese werden *statisch* angelegt und immer dann verwendet, wenn man *im voraus* bereits die Anzahl der Feldkomponenten kennt. Die Feldkomponenten können sowohl Elementar- als auch Referenzdatentypen aufnehmen. Überschreitet man die *Feldgrenzen*, kommt es zu einem *Laufzeitfehler*. Falls man diesen nicht abfängt, bricht das Programm ab.

Collection-Klassen

Collection-Klassen sind *Felder* und bieten bei der Verwaltung großer Datenmengen mehr Möglichkeiten als gewöhnliche Felder. Die Größe einer *Collection* kann jeder Zeit *dynamisch* an die gewünschte Anzahl der Elemente angepasst werden.

Die *Elemente* der Collection-Klassen sind immer *Instanzen* der Klasse `java.lang.Object`. Da nun jede Klasse direkt oder indirekt von der Klasse `java.lang.Object` abgeleitet wurde, können deren Instanzen wegen des *Polymorphismus*, als Elemente in einer Collection verwendet werden. Das gilt nur für Referenzdatentypen, Elementardatentypen bilden *keine* Objekte. Sollen elementare Werte in einem Objekt einer Collection-Klasse abgelegt werden, so werden diese zuvor automatisch^{*)} in deren *Wrapper-Klassen* umgewandelt.

Das Paket `java.util.*` umfasst ca. 20 Interfaces und Klassen zur dynamische Verwaltung von großen Datenmengen. Wurzel dieser ist das Interface **Collection**. Von ihm abgeleitet sind u.a. das Interface **Set** zur Verwaltung von *Mengen* und das Interface **List** zur Verwaltung von *Listen*.



Im Paket sind bereits verschiedene Implementierungen der dort definierten Interface enthalten. Alle vom Interface **Collection** abgeleitete Klassen nennt man *Collection-Klassen*. Die Collection-Klassen **ArrayList**, **LinkedList**, **Vector** und **Stack** implementieren das Interface **List** und werden deshalb auch als *List-Klassen* bezeichnet. Das Interface **Set** wird u.a. von den *Set-Klassen* **HashSet** und **TreeSet** implementiert.

^{*)}seit Java 1.5

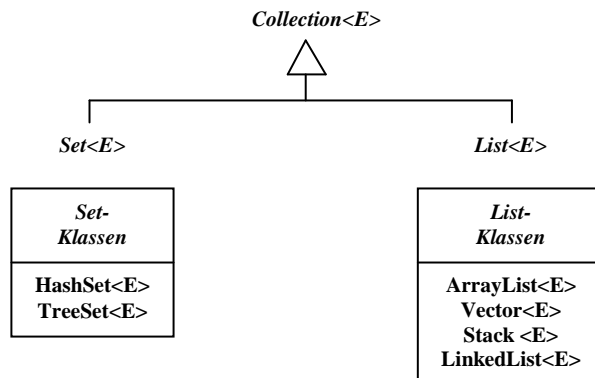
11.3.1 Generische Datentypen - Template

Eine **generische Klasse** ist eine Schablone (**Template**), mit deren Hilfe „echte Klassen“ erzeugt werden können. Sie hat formale Parameter (in spritzen Klammern notiert), die als Platzhalter für *Datentypen* stehen. Durch einen Generierungsprozess wird aus der Schablone eine Klasse erzeugt. dabei werden die formalen Parameter durch die konkreten Datentypen ersetzt. Gleiches gilt für **generische Interfaces**.

Das Interface `Collection<E>` definiert eine Schablone für Collections verschiedenen Typs. E ist der Typparameter. Beispielsweise kann `Collection` mit dem Typ `String` versehen werden:

```
Collection<String> strings;
```

Abgeleitete Klassen dieses Interfaces können nur `String` – Objekte verwalten. Der Versuch einer Aufnahme anderer Objekte führt zu Compilerfehlern. Durch die Parametrisierung, d.h. der Festlegung der Datentypen zur Compilerzeit, erfolgt eine statische Typüberprüfung. Diese ist erheblich sicherer als die dynamische Typüberprüfung zur Laufzeit.



11.3.2 Interface Collection<E>

In allen `Collection`-Klassen stehen folgende Methoden zu Verfügung:

Methoden (Auswahl)

Name	Parameter-anzahl	Parameter-typ	Ergebnis-Typ	Beschreibung
<code>add</code>	1	E	boolean	fügt Objekt ein
<code>addAll</code>	1	Collection<E>	boolean	fügt alle Collection-Objekte ein
<code>clear</code>	0		void	löscht alle Komponenten
<code>contains</code>	1	Object	boolean	fragt, ob Objekt vorhanden
<code>containsAll</code>	1	Collection	boolean	fragt, ob alle Collection-Objekte vorhanden

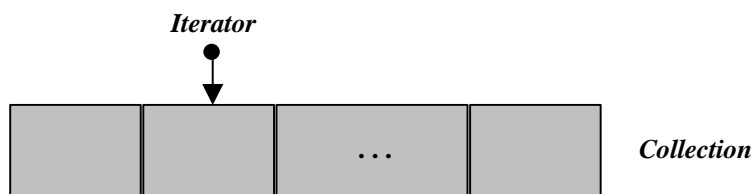
isEmpty	0		boolean	fragt, ob Collection leer
iterator	0		Iterator<E>	liefert Iterator für die Collection
remove	1	Object	boolean	löscht Objekt
removeAll	1	Collection	boolean	löscht alle Collection-Objekte
retainAll	1	Collection	boolean	löscht alle Objekte, die <i>nicht</i> Collection-Objekte
size	0		int	liefert Anzahl der Objekte
toArray	0		E[]	wandelt Collection in Feld um

11.3.3 Interface Iterator<E>

Die Methode **iterator** des Collection-Interface liefert einen **Iterator<E>** (genauer: ein Objekt einer *Iterator-Klasse*). Dieser ermöglicht es, alle Elemente einer Collection-Klasse kontrolliert zu durchlaufen.

Methoden (Auswahl)

Name	Parameteranzahl	Parameter-typ	Ergebnis-Typ	Beschreibung
hasNext	0		boolean	fragt, ob noch ein Element existiert
next	0		E	liefert das nächste Element
remove	0		void	entfernt zuletzt angesprochenes Element



```
Iterator<Double> it = liste.iterator();
while( it.hasNext()) System.out.print( " " + it.next());
```

Neu ist seit Java 1.5 eine erweiterte Form der for-Schleife zum Durchlaufen aller Komponenten eines Feldes (statisch oder dynamisch). Diese erleichtert an einigen Stellen die Programmierung, indem ein Iterator überflüssig wird.

```
for( Double n: liste ) System.out.print( " " + n);
```

11.3.4 Interface Set<E>

Unter einer Menge versteht man eine Collection, in der jedes Element *höchstens einmal* auftritt. Das Interface **Set<E>** legt für die von **Collection<E>** ererbten Methoden fest, dass diese *keine Duplikate* in *Set-Klassen* aufnehmen dürfen.

Damit lassen sich aus der Mathematik bekannte Mengenoperationen mit denen in **Collection<E>** und in den *Set-Klassen* spezifizierten Methoden realisieren, **n** und **m** seien Objekte einer Set-Klasse:

$$\begin{aligned} M \cup N &\triangleq m.\text{addAll}(n) \\ M \cap N &\triangleq m.\text{retainAll}(n) \\ M \setminus N &\triangleq m.\text{removeAll}(n) \end{aligned}$$

Klasse HashSet<E>

verwalten Mengen, ungeordnet

Im ersten Beispiel werden Lottozahlen zufällig ermittelt und in ein Objekt der Klasse **HashSet<Integer>** abgespeichert. Zusätzlich soll die Anzahl der mehrmals gezogenen Zahlen festgestellt werden.

Lotto1.java

```
// Lotto1.java MM 2007

import java.util.*;

/**
 * Lotto mit HashSet<Integer>
 */
public class Lotto1
{
    public static void main( String[] args)
    {
// HashSet
        Set<Integer> set = new HashSet<Integer>();
        int doppelt = 0;

        while( set.size() < 6) // Erzeugen der Lottozahlen
        {
            int num = ( int)( Math.random() * 49) + 1;
            if( !set.add( num)) doppelt++;
        }

// Ausgabe der Lottozahlen
        for( Integer n: set) System.out.println( n);
        System.out.println( "Duplikate: " + doppelt);
    }
}
```

Klasse TreeSet<E>

verwalten Mengen, geordnet

Die Elemente werden *geordnet* abgespeichert.

Lotto2.java

```
// Lotto2.java
MM 2007

import java.util.*;

/**
 * Lotto mit HashSet<Integer>
 */
public class Lotto2
{
    public static void main( String[] args)
    {
        // TreeSet
        Set<Integer> set = new TreeSet<Integer>();
        int doppelt = 0;

        while( set.size() < 6) // Erzeugen der Lottozahlen
        {
            int num = ( int)( Math.random() * 49) + 1;
            if( !set.add( num)) doppelt++;
        }

        // Ausgabe der Lottozahlen
        for( Integer n: set) System.out.println( n);
        System.out.println( "Duplikate: " + doppelt);
    }
}
```

Während im ersten Fall Lottozahlen ungeordnet ausgegeben werden, liegen im zweiten Fall die Zahlen bereits geordnet vor. Im Programm ist nur *eine* Änderung, die Änderung des Klassennamen, notwendig.

11.3.5 Interface List<E>

Unter einer Liste versteht man eine Collection, in der Elemente auch *mehrfach* vorkommen können. Die Reihenfolge ihres Auftretens wird beim Einfügen festgelegt. Analog den *gewöhnlichen Feldern* sind die Elemente einer Liste von 0 beginnend durchnummeriert (indiziert). Das Interface **List<E>** legt für die von **Collection<E>** ererbten Methoden **add**, **addAll**, **remove**, **removeAll**, **retainAll** fest, dass Objekte der *List-Klassen* an deren *Ende* eingefügt werden und nach dem Löschen eines Objekts stets die nachfolgenden Elemente um eine Position *nach vorn verschoben* werden. Das Interface deklariert noch weitere Methoden:

Methoden (Auswahl)

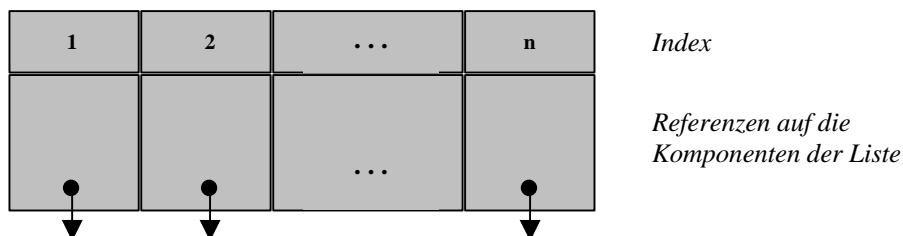
Name	Parameteranzahl	Parameter-typ	Ergebnis-Typ	Beschreibung
add	2	int, E	void	fügt Objekt an Position ein

get	1	int	E	liefert Objekt an Position
indexOf	1	Object	int	liefert kleinsten Index des Objekts
lastIndexOf	1	Object	int	liefert größten Index des Objekts
remove	1	int	E	löscht Objekt an Position und liefert das gelöschte zurück
set	2	int, E	E	ersetzt Objekt an Position und liefert es zurück
listIterator	0		ListIterator<E>	liefert ListIterator für die Collection

Mittels einem von Interface **Iterator<E>** abgeleiteten Interface **ListIterator<E>** kann man auf die Komponenten einer *List-Klasse* sowohl von *vorn nach hinten* als auch *umgekehrt* zugreifen (Beispiel „Spiegelzahlen“).

Klasse ArrayList<E>, Vector<E> *verwalten lineare Listen als Feld*

Diese Klassen sind Java-Repräsentationen einer **linearen Liste** und werden als Felder von Elementen des Typs **E** realisiert. Sie erlauben das Einfügen von Elementen an beliebiger Stelle und bietet sowohl *sequentiellen* als auch *wahlfreien Zugriff* auf die Elemente. Zugriffe auf Objekte der Klasse **Vector<E>** sind *synchronisiert*, d.h. zeitgleiche Zugriffe werden gesteuert behandelt.



Die *Zugriffe* auf vorhandene Elemente und das Durchlaufen der Liste sind *schnelle* Operationen. *Löschungen* und *Einfügungen* sind dagegen relativ *langsam*, da Teile des Feldes umkopiert werden müssen.

DoubleListe.java

```
// DoubleListe.java
```

MM 2007

```
import Tools.IO.*;
import java.util.*;

/**
 * Doubleliste mit Ein- und Ausgabe.
 */
public class DoubleListe
{
/**
```

```
* Liste von Objekten eines Typs Double.
*/
private List<Double> liste;

/**
 * Konstruktor, legt Typ und Art der Liste fest.
 */
public DoubleListe( List<Double> liste)
{
    this.liste = liste;
}

/**
 * Fuellen einer Liste.
 */
public void eingabeListe()
{
    while( true)
    {
        String eingabe = IOTools.readLine
            ("Naechste Komponente(<ENTER> fuer Abbruch) ");

        if( eingabe.equals( "")) break;

        try
        {
            liste.add( Double.valueOf( eingabe));
        }
        catch( Exception e)
        {
            System.out.println
                ( "Nur Zahlen eingeben!\n" + e);
        }
    }
}

/**
 * Ausgabe einer Liste.
 */
public void ausgabeListe()
{
    for( Double d: liste) System.out.print( " " + d);
    System.out.println();
}

/**
 * Test der List-Klassen
 * ArrayList, Vector, Stack, LinkedList.
 */
public static void main( String[] args)
{
    DoubleListe doubleListe;
```

```

// ArrayList
System.out.println( "ArrayList");
doubleListe
= new DoubleListe( new ArrayList<Double>());
doubleListe.eingabeListe();
doubleListe.ausgabeListe();

// Vector
System.out.println( "Vector");
doubleListe
= new DoubleListe( new Vector<Double>());
doubleListe.eingabeListe();
doubleListe.ausgabeListe();
}
}

```

Klasse Stack<E>

verwaltet Keller

Ein Keller ist eine Datenstruktur, die nach dem *LIFO-Prinzip* (last-in-first-out) arbeitet. Die Elemente werden am vorderen Ende der Liste eingefügt und von dort auch wieder entnommen. Das heißt, das zuletzt eingefügte Element wird zuerst entnommen und das zuerst eingefügte zuletzt (Beispiel „Türme von Hanoi“).

Die Klasse **Stack<E>** ist eine Ableitung der Klasse **Vector<E>**. Sie wurde um neue Zugriffsmethoden erweitert, um das typische Verhalten eines Kellers zu implementieren. Dies ist eine ökonomische Vorgehensweise und bedeutet, **Stack<E>** erbt alle Methoden von **Vector<E>** und kann damit auch wie **Vector<E>** verwendet werden.

Im Beispiel *DoubleListe.java* ließe sich deshalb ohne weitere Änderungen ergänzen:

```

// Stack
System.out.println( "Stack");
doubleListe
= new DoubleListe( new Stack<Double>());
doubleListe.eingabeListe();
doubleListe.ausgabeListe();

```

Methoden (Auswahl)

Name	Parameter-anzahl	Parameter-typ	Ergebnis-Typ	Beschreibung
empty	0		boolean	fragt, ob Keller leer
peek	0		E	liefert oberstes Objekt, ohne es zu löschen
pop			E	liefert oberstes Objekt und löscht es
push	1	E	E	speichert Objekt als oberstes Objekt ab
search	1	Object	int	liefert oberste Position des Objekts

Keller.java

// Keller.java

MM 2004

```
import java.util.*;

/**
 * Kelleroperationen
 */
public class Keller
{
    public static void main( String[] args)
    {
        Stack<String> keller = new Stack<String>();
        keller.push( "eins");      // Fuellen eines Kellers
        keller.push( "zwei");
        keller.push( "drei");

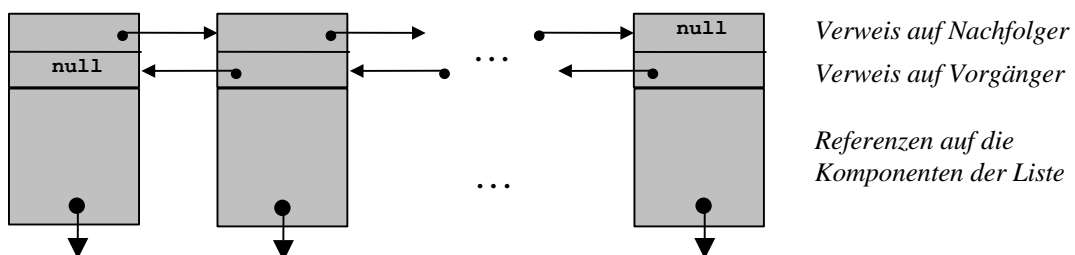
        while( true)              // Auslesen der Kellerelemente
        {
            try
            {
                System.out.println( keller.pop());
            }
            catch( EmptyStackException e) // Keller ist leer
            {
                break;
            }
        }
    }
}
```

Keller.out

```
drei
zwei
eins
```

Klasse LinkedList<E> *verwaltet doppelt verkettete lineare Listen*

Die Klasse **LinkedList<E>** realisiert **doppelt verkettete lineare Listen**. Ihre *Einfüge- und Löschooperationen* sind im Prinzip (viele Elemente vorausgesetzt) *effizienter* als die der einfachen linearen Listen. Der *sequentielle* und *wahlfreie Zugriff* ist dagegen normalerweise *langsamer*, da die Liste immer wieder vom Anfang oder vom Ende durchlaufen werden muss.



Durch den Einsatz eines Iterators können sequentielle Zugriffe beschleunigt werden:

```
/**
 * Ausgabe einer LinkedList
 */
public void ausgabeListe()
{
    Iterator<Double> it = liste.iterator();
    while( it.hasNext())
        System.out.print( " " + it.next());
    System.out.println();
}
```

11.4 Klassen Arrays und Collections

Die Klassen **Arrays** und **Collections** des Pakets `java.util.*` stellen Klassenmethoden zum *Suchen* und *Sortieren* in *gewöhnlichen Feldern* bzw. *Collection-Klassen* zur Verfügung, wobei die Klasse **Arrays** für *einfache Datentypen* und die Klasse **Collections** für *Referenzdatentypen* zuständig sind.

11.5 Zusammenfassung

Welche Klasse bei welchem Problem am günstigsten zur Anwendung kommen sollte, hängt von deren Funktionalität ab. Soll im eigenen Programm eine Liste verwendet werden, stellt sich die Frage, welche der genannten Implementierungen dafür am besten geeignet ist. Die Entscheidung für eine der Klassen ist von den Anforderungen der jeweiligen Anwendung abhängig.

- Bleibt die Liste *klein*, wird hauptsächlich *wahlfrei* darauf zugegriffen, überwiegen die *lesenden* gegenüber den schreibenden Zugriffen deutlich, so liefert die **ArrayList<E>** die besten Ergebnisse.
- Ist die Liste dagegen sehr *groß* und werden *häufig* Einfügungen und Löschungen vorgenommen, ist die **LinkedList<E>** die bessere Wahl.
- Wird von *mehreren* Anwendungen *gleichzeitig* auf die Liste zugegriffen, sollte die Klasse **Vector<E>** verwendet werden, denn ihre Methoden sind bereits weitgehend synchronisiert.
- Sollen *Duplikate* nicht abgespeichert werden, so verwendet man keine *List-Klasse*, sondern eine der *Set-Klassen*.
- Durch Parametrisierung werden dynamische Typüberprüfungen zur Laufzeit durch statische Typüberprüfung zur Compilerzeit ersetzt, was die Programmierung erheblich sicherer macht.