Inhalt

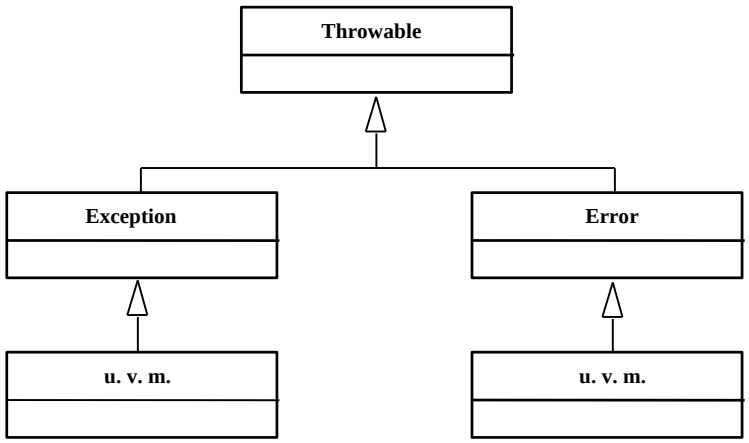
11Dynamische Verwaltung großer Datenmengen

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*. Um vorzeitige Abbrüche zu vermeiden, sollten deren Behandlung im Programm festgelegt werden. Tritt eine solche Ausnahmesituation ein, so wird ein *Objekt* dieser Klassen erzeugt. Die Ausführung des Programms wird unterbrochen und die für das Objekt 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 dieser *nicht* direkt zum Abbruch des Programms.
- Innerhalb des **`catch`**-Blockes stehen die Anweisungen, die bei einer Ausnahme innerhalb des **`try`**-Blockes ausgeführt werden.
- Schließlich kann optional noch ein **`finally`**-Block mit Anweisungen aufgenommen werden, welche auf jeden Fall ausgeführt werden. Zum Beispiel für Aufräumarbeiten, wie das Schließen geöffnete Dateien.

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

Div1.java

```
// Div1.java                                                MM 2014
import Tools.IO.*;                                       // Eingaben

/**
 * 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");
  }
```

```
}
```

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                                              MM 2014
import Tools.IO.*;                                    // Eingaben

/**
 * 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");
  }
```

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 eine neue *Unterklasse*.

Unterklassen der Klasse `Exception`

- class java.lang.**Exception**
  - class java.lang.**ClassNotFoundException**
  - class java.lang.**CloneNotSupportedException**
  - class java.lang.**IllegalAccessException**
  - class java.lang.**InstantiationException**
  - class java.lang.**InterruptedException**
  - class java.lang.**NoSuchFieldException**
  - class java.lang.**NoSuchMethodException**
  - class java.lang.**RuntimeException**
    - class java.lang.**ArithmeticException**
    - class java.lang.**ArrayStoreException**
    - class java.lang.**ClassCastException**
    - class java.lang.**IllegalArgumentException**
      - class java.lang.**IllegalThreadStateException**
      - class java.lang.**NumberFormatException**
    - class java.lang.**IllegalMonitorStateException**

- class java.lang.**IllegalStateException**
- class java.lang.**IndexOutOfBoundsException**
  - class java.lang.**ArrayIndexOutOfBoundsException**
  - class java.lang.**StringIndexOutOfBoundsException**
- class java.lang.**NegativeArraySizeException**
- class java.lang.**NullPointerException**
- class java.lang.**SecurityException**
- class java.lang.**UnsupportedOperationException**

Definition einer neuen `Exception`

Beliebige *eigene* Exception lassen sich durch Vererbung aus diesen Exception-Klassen erzeugen.

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 2014

/**
 * NennerException: Division durch Null.
 */
public class NennerException extends Exception
{
/**
 * Kommentar: Division durch Null.
 * @return Kommentar
 */
  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. Dazu wird in einer Methode `div` im Ausnahmefall ein Objekt der neuen Klasse `NennerException` geworfen und dieses dann im Hauptprogramm abgefangen.

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 *// Ausnahmen werden erkannt*
}

Div3.java

```
// Div3.java                                              MM 2014
import Tools.IO.*;                                    // Eingaben

/**
 * 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");
  }

/**
 * Division a / b, Ausnahme b == 0 wird weitergeleitet.
 * @param a Dividend
```

```
   * @param b Divisor
   * @throws NennerException Division durch Null
   * @return a / b, falls b != 0
   */
    public static double div( double a, double b)
       throws NennerException
    {
       if( b == 0) throw new NennerException();     // Ausnahme
       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

Die Methoden **`toString`** und **`getClass`** haben wir in den Beispielen schon besprochen. Interessant ist die Methode **`equals`** zum Vergleich von Objekten. Da jede Klasse eine Spezifikation der Klasse **`Object`** sind, ist oft eine Aktualisierung dieser Methode sinnvoll. Ein *Überschreiben* der Methode **`equals`** zieht in der Regel ein *Überschreiben* der Methode **`hashCode`** nach sich. 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 zwischen *Elemantardatentypen* 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 Interfaces

11.2.1 Abstrakte Klassen

Abstrakte Klassen **`abstact class`** werden, wie schon erwähnt, im Sinn der *Generalisierung* als Basisklassen eingeführt. Von ihr werden *keine* Instanzen gebildet. Diese Klassen 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. die Signatur (Schnittstelle) wird festgelegt, der Methodenrumpf bleibt leer. Abgeleitete Klassen konkretisieren diesen oder werden ebenfalls als abstrakt deklariert.

Deklaration einer abstrakten Klasse

public abstract class *Klassenname*
{
Deklarationen
Deklarationen_abstrakter_Methode
Implementieren_eigener_Methoden
}

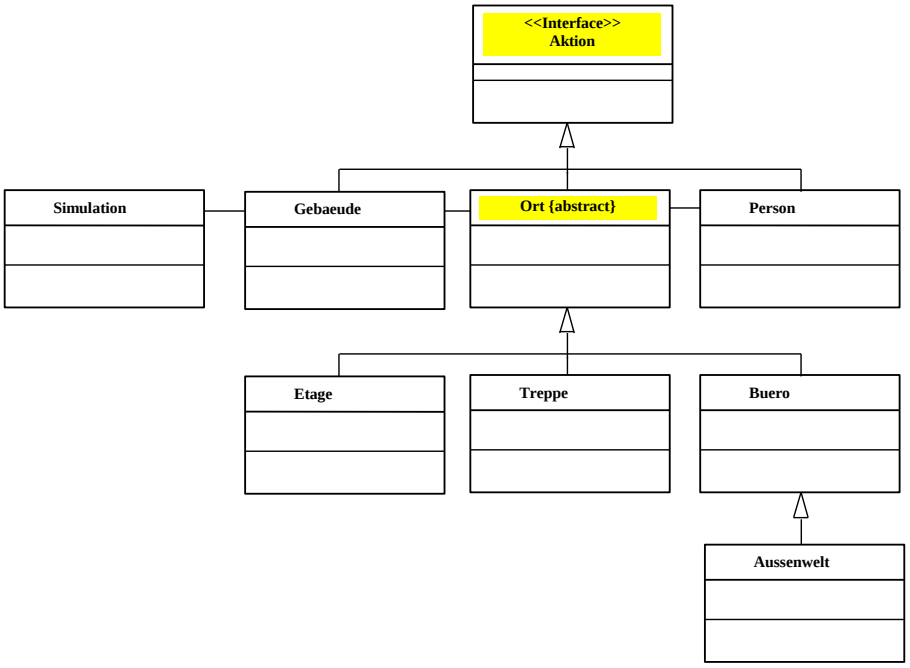
Deklaration abstrakter Methoden

public abstract *Ergebnistyp Methodenname* **(** *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 bzw. gesondert gekennzeichnet.



In einer Gebäudesimulation wird die abstrakte Klasse **`Ort`** als Generalisierung aller im Gebäude betrachteten Räumlichkeiten vereinbart. Diese Klasse verwaltet die sich an einem Ort befindenden Personen, ihr Kommen und Gehen. Dazu besitzt sie Instanzvariablen, um alle Personen an einem Ort zu speichern, und Instanzmethoden, um Personen an einem Ort an- bzw. abzumelden. Verschiedene Subklassen spezifizieren die Räumlichkeiten.

11.2.2 Interfaces

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 Methodenrümpfe 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 Methodenname* **(** *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 *Interfacename*
{
Deklarationen
Implementieren_aller_Interfacemethoden
Implementieren_eigener_Methoden
}

Die Klasse **`Aktion`** dient der Taktung der Gebäudesimulation und ist als **`<<Interface>>`** gekennzeichnet. Sie besitzt nur zwei abstrakte Methoden: **`zeitVergeht`** und **`aktion`**.

***Aktion.java* als Interface**

```
// Aktion.java                                          MM 2014

/**
 * Interface zur Taktung aller bei der Simulation
 * beteiligten Objekte.
 */
public interface Aktion
```

```
{
  public void zeitVergeht();
  public void aktion();
}
```

Alle **`Aktion`** implementierenden Klassen definieren die Methoden **`zeitVergeht`** und **`aktion`**.

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 vermieden, da diese *keine* Attribute und *keine* implementierten Methoden besitzen.

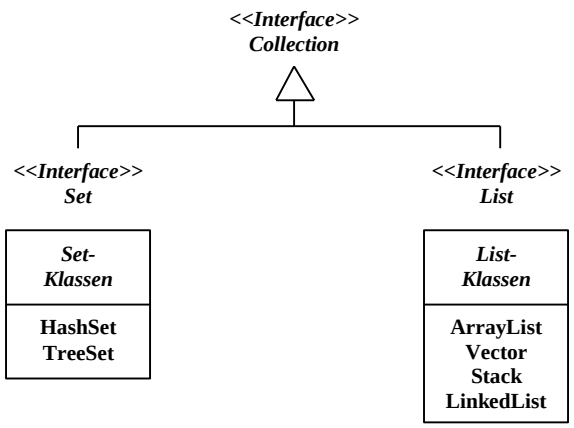
Diese Klasse **`Ort`** ist abstrakt, erfüllt aber nicht die Interface-Bedingungen.

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 Komponenten angepasst werden.

Die *Komponenten* der Collection-Klassen sind *Instanzen* der Klasse **`java.lang.Object`**. Da nun jede Klasse direkt oder indirekt von der Klasse **`java.lang.Object`** abgeleitet wurde, können alle Instanzen einer Klasse wegen des *Polymorphismus* als Komponenten in einer Collection-Klasse 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 automatisch[*)] in ein Objekt der entsprechenden *Wrapper-Klassen* umgewandelt.
Das Paket **`java.util.*`** umfasst ca. 20 Interfaces und Klassen zur dynamischen 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 Interfaces enthalten. Alle das Interface **`Collection`** implementierende 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.

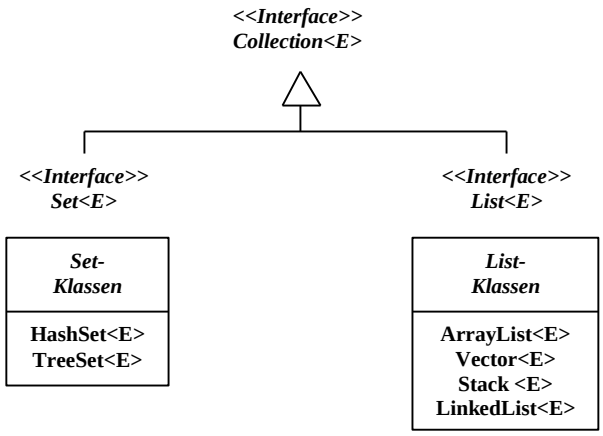
[*)] seit Java 1.5

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 spitzen 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
`add`	1	`E`	`boolean`	fügt Objekt ein
`addAll`	1	`Collection<E>`	`boolean`	fügt alle Collection-Objekte ein
`clear`	0		`void`	löscht alle Komponenten
`contains`	1	`Object`	`boolean`	fragt, ob Objekt vorhanden
`containsAll`	1	`Collection`	`boolean`	fragt, ob alle Collection-Objekte vorhanden

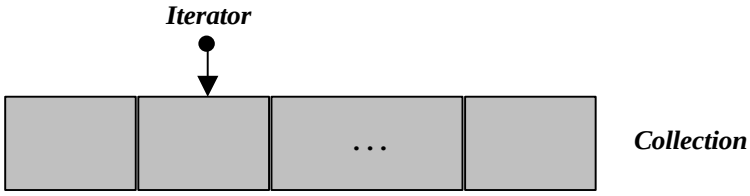
isEmpty	0		`boolean`	fragt, ob Collection leer
iterator	0		**`Iterator<E>`**	liefert Iterator für die Collection
remove	1	`Object`	`boolean`	löscht Objekt aus Collection
removeAll	1	`Collection`	`boolean`	löscht alle Collection-Objekte
retainAll	1	`Collection`	`boolean`	löscht alle Objekte, die *nicht* 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 Komponenten einer Collection-Klasse kontrolliert zu durchlaufen.

Methoden (Auswahl)

Name	Parameter-anzahl	Parameter-typ	Ergebnis-Typ	Beschreibung
hasNext	0		`boolean`	fragt, ob noch eine Komponente existiert
next	0		`E`	liefert die nächste Komponente
remove	0		`void`	entfernt die zuletzt angesprochene Komponente




```
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, ein Iterator wird überflüssig.

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

11.3.4 Interface `Set<E>`

Unter einer Menge versteht man eine Collection, in der jede Komponente *höchstens einmal* auftritt. Das Interface **`Set<E>`** legt für die von **`Collection<E>`** ererbten Methoden fest, dass diese *keine Dublikate* 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:

M ∪ N	≙	**`m.addAll( n)`**
M ∩ N	≙	**`m.retainAll( n)`**
M \ N	≙	**`m.removeAll( n)`**

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 2014

import java.util.*;                          // Collection-Klassen

/**
 * Lotto mit HashSet<Integer>
 */
public class Lotto1
{
  public static void main( String[] args)
  {
    Set<Integer> set = new HashSet<Integer>();   // HashSet
    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 Komponenten werden *geordnet* abgespeichert.

Lotto2.java

```
// Lotto2.java                                              MM 2014

import java.util.*;                          // Collection-Klassen

/**
 * Lotto mit HashSet<Integer>
 */
public class Lotto2
{
  public static void main( String[] args)
  {
    Set<Integer> set = new TreeSet<Integer>();   // TreeSet
    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 Klassennamens, notwendig.

11.3.5 Interface `List<E>`

Unter einer Liste versteht man eine Collection, in der Komponenten auch *mehrfach* vorkommen können. Die Reihenfolge ihres Auftretens wird beim Einfügen festgelegt. Analog den *gewöhnlichen Feldern* sind die Komponenten 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 Komponenten um eine Position *nach vorn verschoben* werden. Das Interface deklariert noch weitere Methoden:

Methoden (Auswahl)

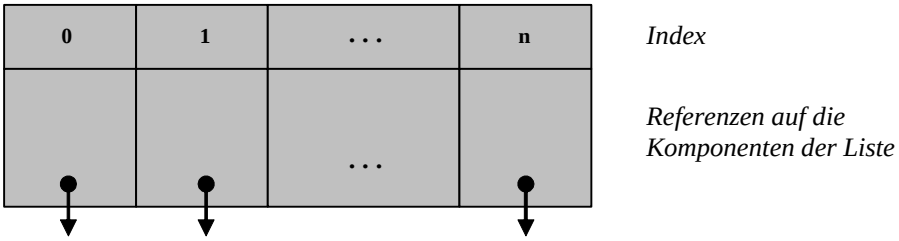
Name	Parameter-anzahl	Parameter-typ	Ergebnis-Typ	Beschreibung
`add`	2	`int, E`	`void`	fügt Objekt an Position ein
`get`	1	`int`	`E`	liefert Objekt an Position

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 für **lineare Listen** und werden als Felder von Komponenten des Typs **E** realisiert. Sie erlauben das Einfügen von Komponenten an beliebiger Stelle und bietet sowohl *sequentiellen* als auch *wahlfreien Zugriff* auf die Komponenten. Zugriffe auf Objekte der Klasse **Vector<E>** sind *synchronisiert*, d.h. zeitgleiche Zugriffe werden gesteuert behandelt.



Die *Zugriffe* auf vorhandene Komponenten 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 2014

import Tools.IO.*;
import java.util.*;                          // Collection-Klassen

/**
 * 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 Komponenten werden am vorderen Ende der Liste eingefügt und von dort auch wieder entnommen. Das heißt, die zuletzt eingefügte Komponente wird zuerst entnommen und die 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 weiter 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

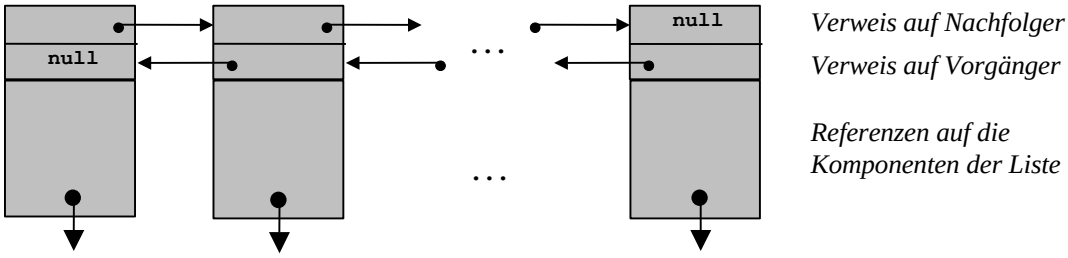
```
// Keller.java                                          MM 2014

import java.util.*;                    // Collection-Klassen

/**
 * 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 Kellerkomponenten
    {
      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 Löschoperationen* sind im Prinzip (viele Komponenten 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 **`ListIterator`** können sequentielle Zugriffe gesteuert werden:

DoubleLinkedListe.java

```
// DoubleLinkedListe.java                                MM 2014

import Tools.IO.*;                                  // Eingaben
import java.util.*;                       // Collection-Klassen

/**
 * DoubleLinkedList, Ausgabe mit Iterator.
 */
public class DoubleLinkedListe extends DoubleListe
{
/**
 * Konstruktor, legt Typ und Art der Liste fest.
 */
  public DoubleLinkedListe( List<Double> liste)
  {
    super( liste);
  }

/**
 * Ausgabe einer LinkedList, vorwaerts.
 */
  public void ausgabeListeVorwaerts()
  {
    ListIterator<Double> it = liste.listIterator( 0);
    while( it.hasNext())
      System.out.print( " " + it.next());
    System.out.println();
  }

/**
 * Ausgabe einer LinkedList, rueckwaerts.
 */
  public void ausgabeListeRueckwaerts()
  {
    ListIterator<Double> it
    = liste.listIterator( liste.size());
    while( it.hasPrevious())
      System.out.print( " " + it.previous());
    System.out.println();
  }

/**
 * Test der Klasse DoubleLinkedList
 */
  public static void main( String[] args)
  {
    System.out.println( "LinkedList");
```

```
      DoubleLinkedListe doubleListe
      = new DoubleLinkedListe( new LinkedList<Double>());

      System.out.println( "Eingabe:");
      doubleListe.eingabeListe();

      System.out.print( "Ausgabe vorwaerts:   ");
      doubleListe.ausgabeListeVorwaerts();

      System.out.print( "Ausgabe rueckwaerts: ");
      doubleListe.ausgabeListeRueckwaerts();
    }
  }
```

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.