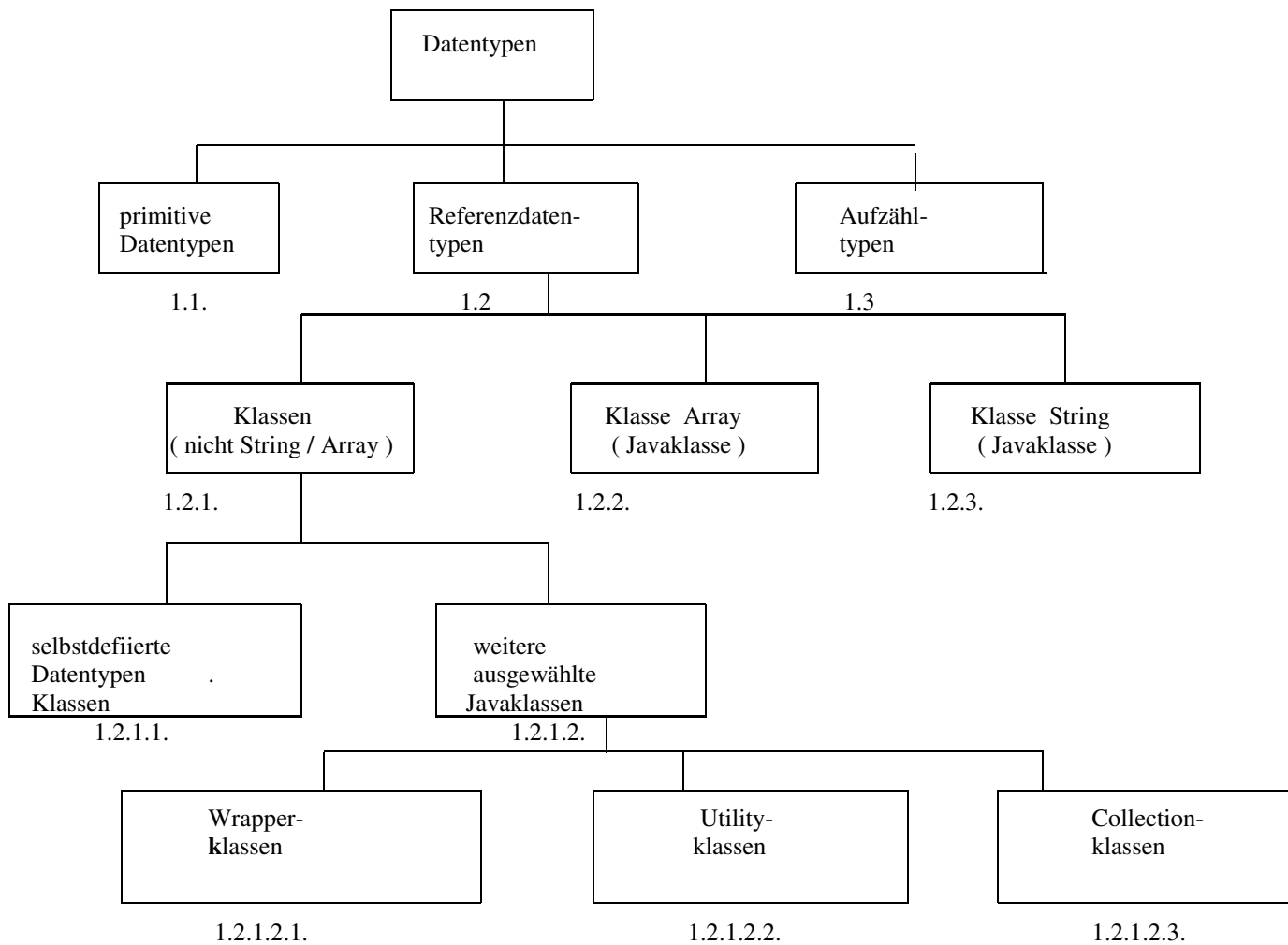


1. Datentypen	1
1.1. Primitive Datentypen	1
1.2. Referenztypen	3
1.2.1. Klassen	5
1.2.1.1. Selbstdefinierte Datentypen	5
1.2.1.2. Javaklassen	7
1.2.2. Die Klasse Array in Auszügen	12
1.2.3. Die Klasse String in Auszügen	14
1.3. Aufzähltypen	18
1.4. Instanzvariable, lokale Variable, Klassenvariable und Klassenmethoden	22
2. Ein- Ausgabe über Tastatur und Bildschirm	23
3. Beziehungen zwischen Klassen	26
3.1. Assoziationen	26
3.1.1. 1 :1 - Assoziation	26
3.1.2. 1 : n- Assoziation	29
3.2. Vererbung	32
3.2.1. einfache Vererbung	32
3.2.2.. Polymorphie	36
3.2.3.. Konstruktoren und Vererbung	37
3.2.4. Abstrakte Klassen	40
4. Typisierung beliebiger Klassen	44
5. Interface	47
5.1. Vererbung – Interface	47
5.2. Selbstcodiertes Interface und Übung	48
5.3. selbstcodiertes generisches Interface (für .Collection)	51
6. Exceptions	52
7. Programmieren unter einer graphischen Oberfläche	56
7.1. Fenster aufbauen	56
7.2. DV- technische Realisierung des Fensteraufbaues	57
7.3. Delegation- Event- Modell	57
7.4. DV- technische Realisierung der Funktionalität	58
7.4.1. Registrieren der ActionListener	58
7.4.2. Implementieren der benötigten Interfaces	59
7.5. Ausführen eines GUI- Programmes	60
7.6. Fenster mit mehreren Buttons	60
7.7. Innere anonyme Klassen	62
8. Entwurfsmuster	64
8.1. Singleton	64
8.2. Observer	66

1. Datentypen

Mit Hilfe der Datentypen legt man Variable an, die mit Befehlen bzw. Methoden angesprochen werden



1.1. Primitive Datentypen

1.

```
public static void main (...)  
{  
    int x    = 4711;  
    double y = 0.0;  
    y = x;  
    System.out.println ( y );  
}
```
2.

```
public static void main (...)  
{  
    int x    = 0;  
    final double y = 3.14;  
    x = y;  
    System.out.println ( x );  
}
```
3.

```
public static void main (...)  
{  
    int x    = 3;  
    int y    = 4;  
    double z = 0.0;  
    z = x / y;  
    System.out.println ( z );  
}
```

Übungen :

1. Was wird jeweils ausgegeben ?

```
public class Anwendung
{
    public static void main(String[] args)
    {
        int x0 = 25;
        int y0 = 4;
        System.out.println ( x0 / y0 );
        System.out.println ( x0 == y0 );

        int x1 = 25;
        float y1 = 4.0f;
        System.out.println ( x1 / y1 );

        x1 = y1;
        System.out.println ( x1 );

        int x2 = 25;
        double y2 = 4.0;
        System.out.println ( x2 / y2 );
        System.out.println ( y1 == y2 );

    } // end main
} // end class Anwendung
```

// a)
// ergibt
// ergibt

// b)
// ergibt
// float- Zahlen sind auf 7 signifikante
// Stellen genau

//

// c)
// ergibt
// ergibt
// double- Zahlen sind auf 15 signifikante
// Stellen genau

2 Runden Sie kaufmännisch (= auf 2 Nachkommastellen)

eine reelle Zahl (Datentyp float mit drei Nachkommastellen

Beispiel: 1.234	→	1.23
1.237	→	1.24

1.2. Referenzdatentypen

Unterschied zwischen Variablen mit

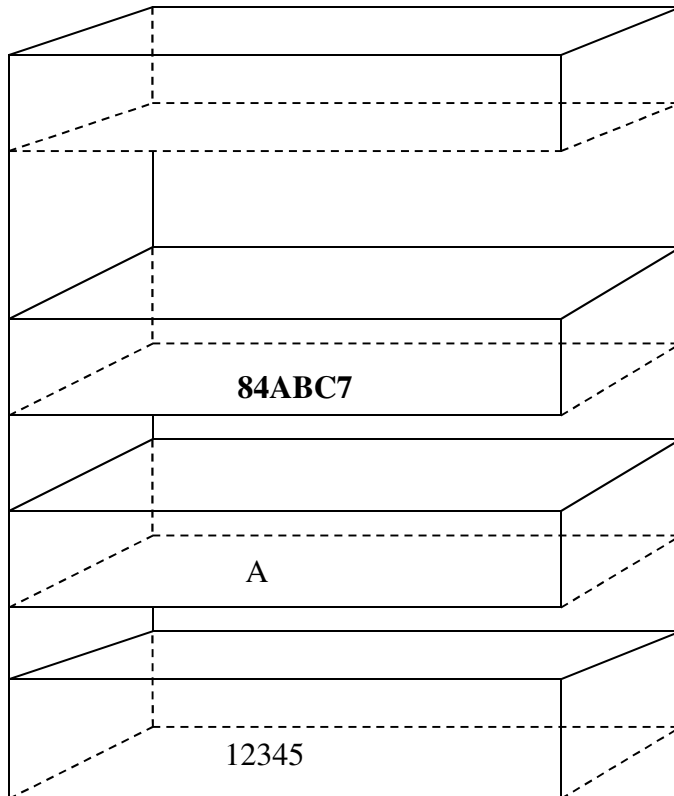
primitiven Datentypen

und

mit Referenzdatentypen

Stack

Heap



cubus
0x0F9D

zeichen
0x0FA1

zahl
0x0FA2

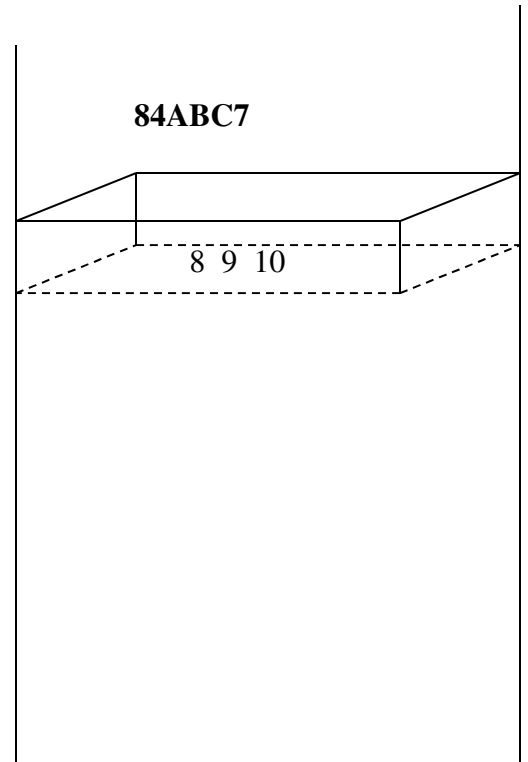


Tabelle			
Symbolische Adresse	Maschinenadresse	Wert	Länge
zahl	0x0FA2	12345	4
zeichen	0x0FA1	A	1
cubus	0x0F9D	84ABC7	4

Definition:

```
int zahl = 12345;
char zeichen = 'A';
Cubus cubus = new Cubus ( 8, 9, 10 );
```

Referenzdatentypen

Das sind die selbstdefinierten Datentypen, die Javaklassen, die Klasse String, die Klasse Array und die vordefinierte Konstante null, die eine leere Referenz bezeichnet.

Referenzdatentypen können im Prinzip genauso benutzt werden wie primitive Datentypen. Da sie jedoch lediglich einen Verweis darstellen, ist die Semantik einiger Operatoren anders als bei primitiven Datentypen.

1. Bei Zuweisung einer Variablen mit einem Referenzdatentyp zu einer anderen solchen Variablen wird lediglich der Wert des Verweises des Sendeobjektes im Verweis des Empfangsobjektes eingetragen. Das Empfangsobjekt selbst behält seinen Inhalt, ist aber nicht mehr adressierbar. Nach der Zuweisung zeigen beide Verweise auf das gleiche Objekt (Sendeobjekt). Das zu überschreibende wird irgendwann vom Garbage Collector entsorgt.

```
public class Anwendung
{
    public static void main ( String [ ] args )
    {
        Artikel a_1 = new Artikel ( 1, "Kette", 1000.0 );
        Artikel a_2 = new Artikel ( 2, "Uhr", 2000.0 );
        a_1 = a_2;
        System.out.println ( a_1 );
        System.out.println ( a_2 );
        a_1.aendereBezeichnung ( "Armband" );
        // Damit wurde das Attribut Bezeichnung im Objekt, das durch a_1 und a_2
        // angesprochen wird, geändert.
        System.out.println( a_1 );
        System.out.println( a_2 );
    } // end main
} // end class Anwendung
```

(besser : Kopieren : clone, per Hand)

2. Der Gleichheitstest testet, ob beide Verweise gleich sind, d. h. auf dasselbe Objekt zeigen. Es werden nicht die Inhalte eines Objektes untersucht.

```
public class Anwendung
{
    public static void main ( String [ ] args )
    {
        Artikel a_1 = new Artikel ( 1, "Kette", 1000.0 );
        Artikel a_2 = new Artikel ( 2, "Uhr", 2000.0 );
        if ( a_1.equals ( a_2 ) )
            System.out.println ( "gleich" );
        else
            System.out.println ( "ungleich" );

        a_1 = a_2;
        if ( a_1.equals ( a_2 ) )
            System.out.println ( "gleich" );
        else
            System.out.println ( "ungleich" );

    } // end main
} // end class Anwendung
```

3. a) Primitive Typen werden lediglich definiert.
int zahl = 0;
b) Referenztypen müssen mit Hilfe des new- Operators explizit erzeugt werden.
(Array- und Stringobjekte allerdings können auch ohne new durch Zuweisen von Literalen erzeugt werden.)
Artikel a_1 = new Artikel (2, "Uhr", 2000);
4. Die Rückgabe von Speicher, der von Referenzvariablen adressiert wird, erfolgt durch den Garbage Collector. Er sucht periodisch nach Objekten, die nicht mehr referenziert werden und gibt deren Platz dann frei. Um Bereiche mit primitiven Datentypen kümmert er sich nicht.

1.2.1. Klassen

1.2.1.1. Selbstdefinierte Datentypen

Der Platzbedarf (= alle Eigenschaften / Attribute) für eine selbstdefinierte Variable Objekt), wird dem Compiler vom Programmierer zur Verfügung gestellt in der Klasse. Ebenso muß er ihm sagen, wie man auf die einzelnen Attributen zugreifen kann, denn **alle** Attribute eines selbstdefinierten Datentyps sind von außen nicht sichtbar, d.h. **private**.

```
class Cubus
```

```
{
```

```
    // Attribute
```

```
    private int laenge;
```

```
    private int breite;
```

```
    private int hoehe;
```

```
    /*
```

```
        Der Zugriffsmechanismus auf die einzelnen Attribute sind die Methoden.
```

```
        Mit diesen Methoden kann man
```

- Objekte anlegen (Konstruktor)
- einzelne Werte eines Objektes abfragen
- einzelne Werte eines Objektes ändern

Format der Methoden:

Datentyp Der Rückgabe	Name der Methode	(ggf. Argumente)	;
--------------------------	---------------------	--------------------	---

Der Konstruktor heißt wie die Klasse und hat **keinen** Datentyp der Rückgabe, auch nicht void.

Die Implementierung der Methoden zeigt dem Compiler, was er zu tun hat, wenn er eine Methode (z.B. in der main- Funktion) ausführen soll.

```
*/
```

```
    // Konstruktor ( = wie wird die Variable angelegt ? )
```

```
    public Cubus ( int laenge, int breite, int hoehe )
```

```
    {
```

```
        this.laenge = laenge;
```

```
        this.breite = breite;
```

```
        this.hoehe = hoehe;
```

```
    }
```

```
    // weitere Methoden
```

```
    int getLaenge ( )
```

```
    {
```

```
        return this.laenge;
```

```
    }
```

```
    int getBreite ( )
```

```
    {
```

```
        return this.breite;
```

```
    }
```

```
    int getHoehe
```

```
    {
```

```
        return this.hoehe;
```

```
    }
```

```
    void setLaenge ( int neueLaenge )
```

```
    {
```

```
        this.laenge = neueLaenge;
```

```
    }
```

```
    void setHoehe ( int neueHoehe )
```

```
        // dto. mit Breite
```

```
    {
```

```
        this.hoehe = neueHoehe;
```

```
    } // end class Cubus
```

Symbol:

Name der Klasse
Attribute
Methoden

(vereinfachte) Syntax der Attribute:

Sichtbarkeit attributsName : Typ

(vereinfachte) Syntax der Methoden:

Sichtbarkeit methodName (ggf. parameterName : Datentyp ...) [: Datentyp der Rückgabe]

Einfache Übungen

Übung 1

Datum
- tag : int - monat : int - jahr : int
+ get/ set + getAnzahlTage ProMonat () : int + istSchaltjahr () : boolean + getNaechstenMonat () : int + getNaechstenTag () : Datum + toString () : String + datumUmwandeln (tag : int, monat : int)

/*Die Methode „datumUmwandeln“ bringt beispielhaft folgende Ergebnisse:

```
1. Januar      :      ttt = 1
2. Januar      :      ttt = 2
1. Februar     :      ttt = 32
1. März        :      im Schaltjahr: ttt = 61, im normalen Jahr ttt = 60
24. September  :      ttt = 267 ( im normalen Jahr )
```

*/

Zur Methode „istSchaltjahr“ :

Ein Jahr ist ein Schaltjahr, wenn es durch 4 aber nicht zugleich durch 100 teilbar ist.

Ist das Jahr jedoch durch 400 teilbar, ist es aber wieder ein Schaltjahr.

Ihre Aufgabe:

Codieren Sie die Klasse Datum, und testen Sie die Methoden alle Methoden

Übung 2

Die Gebühren für die Benutzung des Internetcafes sollen verwaltungsfreundlich Pripaid- Karte abgerechnet werden. Als Kosten werden sowohl die Kosten der online- Zeit als auch die Kosten für die Serviceleistung „Drucken“ für eine Sitzung erfasst. Dabei gelten folgende Bedingungen:

- 1) Die Benutzungsgebühren betragen 0,50 EUR pro angefangener Viertelstunde (= 1 Einheit).
- 2) Kosten für A4 Laserdruck in Farbe:
 - bis 19 Seiten 0,12 EUR/ Seite
 - 20 – 49 Seiten 0,10 EUR/ Seite
 - 50 – 100 Seiten 0.08 EUR/ Seite
 - über 100 Seiten 0.05 EUR/ Seite

Ihre Aufgaben:

a) Codieren Sie eine Klasse Gebühren mit folgenden Attributen:

-	einheiten	-	Ganzzahl
-	online_gebuehren	-	reelle Zahl
-	druckkosten	-	reelle Zahl
-	gesamtkosten	-	reelle Zahl

Sie hat einen Konstruktor, der alle Attribute mit dem Wert 0 versieht.

Sie hat zusätzlich die folgende Methode

ermittleGebuehren (onlinZeitInMinuten : int, anzahlSeiten : int)“

Die Methode gibt die Gesamtkosten zurück.

b). Erstellen Sie ein Struktogramm für die Methode

c). Testen Sie die Methode in einer main- Funktion

1.2.1.2. Javaklassen (Guido Krüger, Handbuch der Javaprogrammierung S. 223) (nicht Array und String)

Wrapper- Klasse (Klasse Integer in Auszügen)

Zu jedem primitiven Datentyp gibt es in Java eine korrespondierende Klasse, Wrapper- Klasse. Diese kapselt die primitive Variable in einer objektorientierten Hülle und stellt eine Reihe von Methoden zum Zugriff auf die Variable zur Verfügung. Zwar wird man bei der Programmierung meist die primitiven Datentypen verwenden, doch gibt es Situationen, in denen die Anwendung von Wrapper- Klassen sinnvoll ist. z.B. stellt das Paket java.util eine Reihe von Collection- Klassen zur Verfügung, die beliebige Objekttypen speichern können. Um darin auch elementare Typen ablegen zu können, ist es notwendig, anstelle der primitiven Datentypen ihre Wrapper- Klassen zu verwenden.

1. Konstruktor für Wrapper- Klassen- Objekte

Man übergibt den korrespondierenden primitiven Datentyp an den Konstruktor, um ein Objekt mit demselben Wert zu erzeugen:

```
Integer i0 = new Integer ( 0 );           ( Double d0 = new Double ( 0.0 ); )  
Integer i1 = 0;                          ( Double d1 = 3.14; ) usw.
```

2 Weitere Methoden

Alle Wrapper- Klassen stellen Methoden zur Verfügung

2.1. zur Umwandlung eines Wrapper- Datentyps in einen String

public static String toString (Integer integer [, int Zahlensystem]) ;

Wird kein Zahlensystem angegeben, ist das Zahlensystem 10.

```
System.out.println ( Integer.toString ( 10 ) );           // 10    String  
System.out.println ( Integer.toString ( 10, 2 ) );        // 1010  String
```

2.2. zur Umwandlung eines Strings in einen Wrapper- Datentyp

public static Integer parseInt (String string [, int Zahlensystem])

Wird kein Zahlensystem angegeben, ist das Zahlensystem 10.

```
System.out.println ( Integer.parseInt ( "123" ) );        //  
Angenommen, in args [ 0 ] steht 123.  
System.out.println ( args [ 0 ] );                       //  
System.out.println ( Integer.parseInt ( args [ 0 ] ) );   //  
System.out.println ( Integer.parseInt ( args [ 0 ], 2 ) ); //  
Angenommen in args [ 1 ] steht 111 :                     //  
System.out.println ( Integer.parseInt ( args [ 1 ], 2 ) ); //
```



```

public class Anwendung
{
    public static void main(String[] args)
    {
        //      Umwandlung Wrapper- Datentyp in einen String
        Integer i1 = 10;
        System.out.println ( Integer.toString ( i1 ) );    //
        System.out.println ( Integer.toString ( i1, 2 ) ); //
        System.out.println ( Integer.toString ( 255, 2 ) ); //

        //      Umwandlung eines String in einen Wrapper- Datentyp
        System.out.println ( Integer.parseInt( "10" ) );    //
        System.out.println ( Integer.parseInt ( "111", 2 ) ); //
    } // end main
} // end Anwendung

```

3. Abfragen des internen Wertes eines Wrapper- Klassen- Objektes

Die meisten Wrapper- Klassen besitzen zwei Methoden, um den internen Wert der Objekte abzufragen:

Eine liefert ihn passend zum korrespondierenden Grundtyp. Der Name dieser Methode setzt sich aus dem Namen des Basistyps und der Erweiterung **Value** zusammen.

Beispiel:

```

Integer zahl = 100;
System.out.println ( zahl. intValue ( ) );    //

```

Die andere liefert einen String zurück, Das ist die toString- Methode. (siehe 2.1 oben) :

Beispiel:

```

System.out.println ( Integer.toString ( 123 ) ); //

```

4. Methoden, die Konstanten spezieller Werte zur Verfügung stellen

```

public static final int MIN_VALUE;
/*
    rechenfeld wird mit dem niedrigsten Wert belegt, der in 4 Bytes Platz hat.
    ( - 2 147 483 648 ),
    analog Integer.MAX_VALUE ( mit dem höchsten Wert ( 2147483647 ) );
*/
int rechenfeld = Integer. MIN_VALUE;
System.out.println ( rechenfeld );

```

Übung

1. Entwickeln Sie eine Klasse Binärzahl,

die ein Attribut string für die als String dargestellte Binärzahl enthält.

Beispiel: Die Dezimalzahl 6 hat die Binärdarstellung 110 und wird im Attribut String mit dem Wert "110" abgelegt.

Binaerzahl
- string String;
+ Binaerzahl (s :String) + addiere (b : Binaerzahl) : Binaerzahl + multipliziere (b : Binaerzahl) : Binaerzahl + toString () : String + getZahl () : String

Der Konstruktor bekommt einen String mit dem Wert einer Binaerzahl als Argument

addiere addiert zum internen Wert eines Binaerzahl- Objektes den Wert der übergebenen Binärzahl und gibt ein Binaerzahl Objekt mit dem internen Wert der Summe zurück.

multipliziere multipliziert den internen Wert eines Binaerzahl.- Objektes mit dem internen Wert des übergebenen Binaerzahl- Objektes und gibt ein Binärzahl- Objekt mit dem Produkt zurück.

toString gibt das Attribut des Binaerzahl- Objektes aus. Beispielhafte Ausgabe: 110.

getZahl () gibt das Attribut des Binaerzahl- Objektes zurück.
Beispielhafte Rückgabe: 110.

Um Addition und Multiplikation durchführen zu können, ist die Umwandlung des Attributes im String- Format in int- Werte nötig.

Diese Umwandlung des Binaerzahl- Strings in die zugehörigen Zahlenwerte soll mit der

Klassenmethode: **parseInt** der Wrapperklasse **Integer** erfolgen:

Beispiel: `int i = Integer.parseInt ("110", 2);` // liefert 6

Die Rückwandlung vom Zahlenwert in die Binärdarstellung soll mit der

Klassenmethode **toString ()** der Wrapperklasse **Integer** erfolgen:

Beispiel: `String s = Integer.toString (6, 2);` // liefert 110

2. Schreiben Sie eine main- Funktion, die folgende Ausgabe erzeugt:

```
Zahl_1 binär :      110
Zahl_1 dezimal      6
Zahl_2 binär       111
Zahl_2 dezimal      7
Summe binä         1101
Summe dezimal      13
Produkt binär      101010
Produkt dezimal     42
```

Die Zahlen wählen Sie selbst

Utilityklasse

Klasse Random

wird mit **import java.util.Random;** ins package aufgenommen.

Jede Instanz der Klasse Random dient als eigenständiger, unabhängiger Pseudo-zufallszahlengenerator.

```
public class Random
{
    protected long seed;
    protected double nextNextGaussian;
    protected boolean haveNextNextGaussian = false;
    public Random ( );
    public Random ( long seed );
    public void setSeed ( longSeed );
    public int next ( int bits );
    public int nextInt ( int bits );
    public int nextInt ( );
    public long nextLong ( );
    public long nextFloat ( );
    public long nextDouble ( );
    public long nextGaussian ( );
}
```

Beispiele:

```
Random random = new Random ( );
```

1.

```
System.out.println ( random.nextInt ( 100 ) );
```


// gibt eine Zufallszahl aus 0 bis 99 aus.
2.

```
System.out.println ( random.nextInt ( );
```


// gibt eine Zufallszahl aus 2³² möglichen Zahlen aus.

Übung

Ermitteln Sie aus den Zahlen 0 bis 300 5 Zufallszahlen.

Collectionklasse ArrayList in Auszügen

(kleine Liste, Klasse für lesende > schreibende Zugriffe)

```
import java.util.ArrayList;
```

Die Klasse ArrayList gehört zu den Collection- Klassen.

Eine Collection vom Typ List ist eine geordnete Menge von Objekten, auf die entweder sequentiell oder über ihren Index zugegriffen werden kann. Das erste Element hat den Index 0.

Einige Methoden der Klasse ArrayList:

1. **public < Datentyp > ArrayList ()** // Datentyp gibt den Typ der Elemente in der ArrayList an
Das ist der Konstruktor, mit dem ein neues Objekt der Klasse ArrayList erzeugt werden kann.
Parameter werden nicht übergeben. Ein Beispiel-Aufruf sieht so aus:

```
ArrayList < Vertrag > vertraege = new ArrayList < Vertrag > ( );
```
2. **public boolean add (Objekt objekt)**
Diese Methode hängt das als Parameter übergebene Objekt an das Ende der bestehenden Liste. Ein korrekter Aufruf dieser Methode sieht so aus:

```
Vertrag vertrag = new Vertrag ( 123, "Mietvertrag" );
```

```
boolean b = vertraege.add ( vertrag );
```


Falls die Liste noch leer sein sollte, wird auf diese Weise das erste Objekt in die Liste aufgenommen. Es hat den Index 0.
3. **public boolean contains (Objekt objekt)** // sinnvoll bei String, Integer, Double usw.
Der Inhalt der ArrayList wird überprüft. Rückgabe true oder false
4. **public void add (int index, Objekt objekt)**
Diese zweite Variante von add () hängt das neue Objekt nicht an das Ende der Arrayliste, sondern fügt das Objekt in der angegebenen Position ein. Die weiter rechts stehenden Elemente werden dann jeweils eine Position weiter nach rechts verschoben.
5. **public void clear ()**
Diese Methode löscht die gesamte Liste .
6. **public Objekt get (int index)**
Liefert das Objekt mit dem angegebenen Index zurück. Ein korrekter Aufruf sieht so aus:

```
Vertrag vertrag = vertraege.get ( 2 );
```
7. **public boolean isEmpty ()**
Diese Methode sagt, ob die Arraylist leer ist.

Beispiele:

```
Random random = new Random ( );
```

1.

```
System.out.println ( random.nextInt ( 100 ) );
```


// gibt eine Zufallszahl aus 0 bis 99 aus.
2.

```
System.out.println ( random.nextInt ( );
```


// gibt eine Zufallszahl aus 2³² möglichen Zahlen aus.

Übung

Ermitteln Sie aus den Zahlen 0 bis 300 5 Zufallszahlen.

Collectionklasse ArrayList in Auszügen

(kleine Liste, Klasse für lesende > schreibende Zugriffe)

```
import java.util.ArrayList;
```

Die Klasse ArrayList gehört zu den Collection- Klassen.

Eine Collection vom Typ List ist eine geordnete Menge von Objekten, auf die entweder sequentiell oder über ihren Index zugegriffen werden kann. Das erste Element hat den Index 0.

Einige Methoden der Klasse ArrayList:

1. **public < Datentyp > ArrayList ()** // Datentyp gibt den Typ der Elemente in der ArrayList an
Das ist der Konstruktor, mit dem ein neues Objekt der Klasse ArrayList erzeugt werden kann.
Parameter werden nicht übergeben. Ein Beispiel-Aufruf sieht so aus:

```
ArrayList < Vertrag > vertraege = new ArrayList < Vertrag > ( );
```
2. **public boolean add (Objekt objekt)**
Diese Methode hängt das als Parameter übergebene Objekt an das Ende der bestehenden Liste. Ein korrekter Aufruf dieser Methode sieht so aus:

```
Vertrag vertrag = new Vertrag ( 123, "Mietvertrag" );
```



```
boolean b = vertraege.add ( vertrag );
```


Falls die Liste noch leer sein sollte, wird auf diese Weise das erste Objekt in die Liste aufgenommen. Es hat den Index 0.
3. **public boolean contains (Objekt objekt)** // sinnvoll bei String, Integer, Double usw.
Der Inhalt der ArrayList wird überprüft. Rückgabe true oder false
4. **public void add (int index, Objekt object)**
Diese zweite Variante von add () hängt das neue Objekt nicht an das Ende der Arrayliste, sondern fügt das Objekt in der angegebenen Position ein. Die weiter rechts stehenden Elemente werden dann jeweils eine Position weiter nach rechts verschoben.
5. **public void clear ()**
Diese Methode löscht die gesamte Liste .
6. **public Objekt get (int index)**
Liefert das Objekt mit dem angegebenen Index zurück. Ein korrekter Aufruf sieht so aus:

```
Vertrag vertrag = vertraege.get ( 2 );
```
7. **public boolean isEmpty ()**
Diese Methode sagt, ob die Arraylist leer ist.

8. **public Iterator iterator ()**

Beispiel:

```
import java.util.Iterator;
public double getGuthaben ( )
{
    double guthaben = 0.0;
    Konto konto = null;
    Iterator < Konto > iterator = konten.iterator ( );
    while ( iterator.hasNext ( ) )
    {
        konto = iterator.next ( );
        guthaben += konto.getSaldo ( );
    }
    return guthaben;
}
```

Mit der Zeile : `Iterator < Konto > iterator = konten.iterator ();`
wird ein sogenannter Iterator erzeugt, und zwar ein Objekt der Klasse Iterator.

Iterator ist eine Klasse,

`iterator ()` aber eine Methode der Klasse ArrayList.

`while (iterator.hasNext ())`

Hier wird die `hasNext ()`-Methode des Iterator-Objektes `iterator` aufgerufen. Diese Methode liefert den Wert `true` zurück, falls noch nicht alle Objekte der Arrayliste abgearbeitet sind.

`iterator.next ()` liefert das nächste Element der Arrayliste

9. **public Object remove (int index)**

Diese Methode macht das Gegenteil von `add ()`, sie löscht das Objekt mit dem angegebenen Index.
Rückgabotyp : Objekt

10. **public boolean remove (int index);**

Diese Methode löscht das Objekt mit dem angegebenen Index. Rückgabotyp: Boolean

11. **public void set (int index, Objekt objekt)**

Diese Methode arbeitet ähnlich wie die zweite `add ()`-Variante, allerdings wird hier das Element an der Position `index` einfach mit dem neuen Objekt überschrieben. Die weiter rechts stehenden Elemente bleiben an ihren Positionen.

12. **public int size ()**

Sehr nützliche Methode, liefert die Zahl der in der Liste enthaltenen Elemente zurück. Diese Methode muß man verwenden, wenn man die gesamte Liste wie einen Array durchlaufen will.

Übung:

Verwenden Sie die Klasse Artikel

und schreiben Sie eine `main`- Funktion:

1. ArrayList mit 5 Artikeln anlegen
2. Fragen, ob die Liste leer ist
3. alle Artikel anzeigen mit
 - einer einfachen `for`- Schleife
 - einer `for- each`- Schleife
 - einem Iterator
4. Uhr löschen (Index ist nicht bekannt), dann alle Artikel anzeigen
5. neue Uhr an die 2. Stelle mit `set` einfügen, dann alle Artikel anzeigen
6. überschriebenen Artikel mit der Nummer 2 wieder an die 2 Stelle einfügen

Format für for- each

```
for (
    Datentyp der Arrayelemente   irgendein Datename für die Arrayelemente
    :   Name des Array's
)
{
    /*
        Verarbeitung – Ausgabe , jedes Arrayelement wird mit dem gewählten
        Datennamen angesprochen.
    */
}
```

1.2.2. Die Klasse Array (in Auszügen)

Auch Arrays sind Objekte, aber einige Besonderheiten rechtfertigen die Unterscheidung von normalen Objekten:

- Der Compiler kennt Literale, die einen expliziten Aufruf des new- Operators überflüssig machen:
`int [] array = { 1, 2, 3, 4 };`
- Arrays sind klassenlose Objekte. Sie können ausschließlich vom Compiler erzeugt werden, besitzen aber keine explizite Klassendefinition. Sie haben dennoch eine öffentliche Instanzvariable `length` und werden vom Laufzeitsystem wie normale Objekte behandelt.
- Array's sind Objekte, d.h. Array- Variable sind Referenzen.
Die neuere Klasse Arrays (Utilityklasse) hat viele statische Methoden zur Verwaltung und zum Manipulieren von Array's, z.B. zum binären Suchen, Sortieren, Vergleichen.
- Die Klasse System (Utilityklasse) stellt eine Methode für das sichere Kopieren eines Array's zur Verfügung.

Anlegen von eindimensionalen Array's

1. `int [] a;`
und
`a = new int [5];`
2. **oder**
`int [] a = new int [5];`

Anlegen und Initialisieren von eindimensionalen Array's

1. `int [] a = { 1, 2, 3, 4, 5 };`
`String [] b = new String { "true", "false" };`
`boolean [] c = new boolean { true, false };`
2. **oder**
`int [] a = new int [5];`
und
`a [0] = 1;`
`a [1] = 2; usw.`

Anlegen eines zweidimensionalen Array's

```
char [ ] [ ] rechteck = new char [ 4 ] [ 6 ];
```

Zeile ↑ ↑ Spalte

Ansprechen eines Array- Elementes:

```
rechteck [ 0 ] [ 1 ] = 'X';
```

Belegen eines zweidimensionalen Array's mit Werten

```
public class Anwendung
{
    public static void main(String[] args)
    {
        int [ ] [ ] zahl = { { 1, 4 },
                             { 2, 5 },
                             { 3, 6 }
                           };
        int zeile = 0;
        int spalte = 0;
```

```

        for ( spalte = 0; spalte < 2; spalte ++ )           // äußere Schleife
        {
            for ( zeile = 0; zeile < 3; zeile ++ )         // innere Schleife
            {
                System.out.println ( zahl [ zeile ] [ spalte ] + " " );
            } // end innere Schleife
        } // end äußere Schleife
    } // end main
} // end class Anwendung

```

Sortieren von Array' s

Format: **Arrays.sort (arrayname, position_von, anzahl_elemente);**

Soll das ganze Array sortiert werden, können position_von und anzahl_elemente weggelassen werden.

```

import java.util.Arrays;
public class Anwendung
{
    public static void main ( String [ ] args )
    {
        int [ ] array = { 5555, 33, 1, 756, 890, 22222 };
        Arrays.sort ( array, 0, array.length );
        // length ermittelt die Länge des Array' s

        for ( int a : array )
            System.out.println ( a );

    } // end main
} // end class Anwendung

```

Kopieren von Array' s

Array' s werden richtig kopiert mit:

<pre> System.arraycopy (quellfeld. indexposition_start, zielfeld, indexposition_ziel, anzahl_elemente) </pre>
--

```

public class Anwendung
{
    public static void main(String[] args )
    {
        int [ ] array_1 = { 5555, 33, 1, 756, 890, 22222 };
        int [ ] array_2 = new int [ 6 ];
        System.arraycopy ( array_1, 0, array_2, 0, array_1.length );

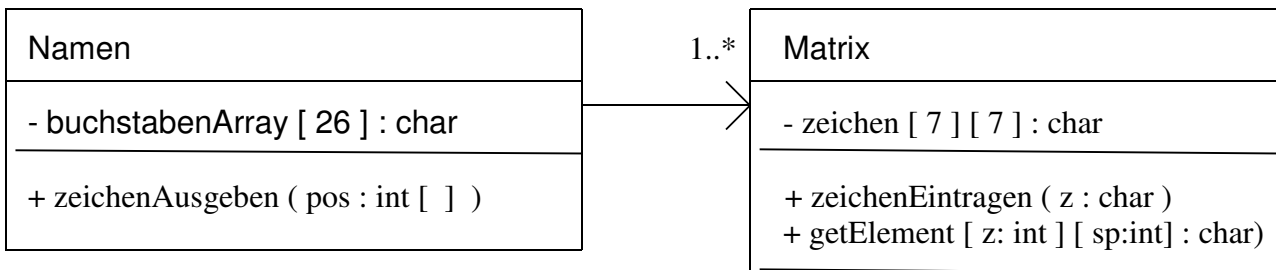
        for ( int a : array_2 )
            System.out.println ( a );

    } // end main
} // end Anwendung

```

Übungen zu Array's

1. Ermitteln Sie den größten und den kleinsten Wert des folgenden Array's in einer for- each- Schleife, und geben sie beide aus.
`int [] array = { 1, 2, 3, 4, 5 };`
2. a) Sortieren Sie folgendes Array:
`String [] wort = { "fort", "alle", "fliegen", "amseln" };`
 b) Geben Sie das sortierte Array aus
- 3.. Gegeben ist folgendes Array:
`int [] array = { 1, 2, 3, 4, 5 };`
 Dieses Array ist für die weitere Verarbeitung zu klein.
 Kopieren Sie seine Inhalte in ein Array, das Platz hat für 10 Elemente vom Typ int.
4. Legen Sie in einer main- Funktion einen Würfel an, würfeln Sie dreimal, und geben Sie die Summe von drei Würfeln aus.
- 5.



Namen sollen `A` stilisiert ausgegeben werden. Die Methode „zeichenEintragen“ `A A` in der Klasse Matrix steht zur Verfügung. Die Klasse Namen hat ein `A A` Array, in das alle Buchstaben eingetragen sind. Ihrer Methode `AAAAAA` „zeichenAusgeben“ wird ein Array übergeben, das die Positionen `A A` der Buchstaben im „buchstbenArray“ enthält,

und zwar in der Reihenfolge, wie sie zum Namen zusammengesetzt werden müssen.
 Codieren Sie die Klasse Namen und Matrix.
 Die Methode „zeichenEinfuegen“ steht in Teilen bei den Lösungsvorschlägen auf Seite 14.

6. Erstellen Sie eine Klasse Matrix. Sie hat als Attribute **die Matrix „entfernung“** und das **eindimensionale Array „orte“**, als Methode **„getEntfernung (String von, String bis)“**. Testen Sie die Methode in einer main- Funktion
 (Lösungsvorschlag S. 14)

1.2.3. Die Klasse String in Auszügen

Eigentlich sind auch Strings Objekte, aber einige Besonderheiten rechtfertigen die Unterscheidung von normalen Objekten: Der Compiler kennt Literale, die einen expliziten Aufruf des new- Operators überflüssig machen:

```
String string = "hallo";
```

Die Klasse String ist zwar wie eine gewöhnliche Klasse in der Laufzeitbibliothek von Java vorhanden. Der Compiler hat aber Kenntnisse über den inneren Aufbau von Strings und generiert bei Stringoperationen Code, der auf Methoden der Klasse String zugreift.

Die Klasse String verfügt über Methoden

- zur Untersuchung einzelner Zeichen der Folge
- zum Vergleich von Zeichenketten
- zum Kopieren von Zeichenketten usw.

Konstrukturen

- **public String s1 = new String ()**
Das erzeugte Objekt stellt eine leere Zeichenfolge dar.
- **public String s2 = new String ("hallo")**
Das erzeugte Objekt ist eine Kopie des Argumentes, d.h. es erhält eine Zeichenkette mit dem gleichen Inhalt wie das Argument.
- **public String s3 = "hallo"**
Die Schreibweise orientiert sich an primitiven Datentypen, das Ergebnis ist das gleiche wie bei s2.
- **public String (char [] data)**
Dieser Konstruktor erzeugt und initialisiert ein String- Objekt so, daß es die im Array enthaltene Zeichenfolge repräsentiert. Der Inhalt des Array's wird kopiert. Ist data = null, wird eine NullPointerException ausgelöst.

Weitere Methoden

- **public char [] toCharArray ()**

Eine neues Zeichenfeld (Array) wird erzeugt und zurückgegeben. Die Länge des Feldes ist gleich der Länge (public int length) des Stringobjektes. Das Feld wird mit der von diesem String repräsentierten Zeichenfolge initialisiert.

Beispiel:

1. String string = new String ("hallo");
char [] erstesFeld = string.toCharArray ();
// erstesFeld ist ein Feld oder Array und enthält jetzt "hallo".
2. Länge bestimmen mit length:
char [] zweitesFeld = new char [erstesFeld.length];

- **public char charAt (int i)**
Das Zeichen an der Stelle i des Stringobjektes wird zurückgegeben.

Beispiel: String string = "hallo";
char c = string.charAt (0);
in c steht h.

Vergleichen

- **public boolean equalsIgnoreCase (String anotherString)**
Die Methode kennt keine Unterschiede zwischen Groß- und Kleinschreibung.

- **public boolean equals (String anotherString)**
Die Methode kennt Unterschiede zwischen Groß- und Kleinschreibung.

Beide Methoden vergleichen (== nicht möglich) die Zeichenkette in einem Stringobjekt mit der Zeichenkette des übergebenen Stringobjektes.

```
String string = new String ( "hallo" );  
if ( string.equals ( "hallo" ) )  
{  
    System.out.println ( "gleich" );  
}  
else  
{  
    System.out.println ( "ungleich" );  
}  
// oder statt if : System.out.println ( string.equals ( "hallo" ) );
```

- **public int compareTo (String anotherString)**

“equalsIgnoreCase” und “equals”

vergleichen auf gleich/ ungleich und geben entsprechend true/ false zurück.

‘compareTo’ vergleicht lexikographisch (case- sensitive) und gibt einen ‘int’- Wert zurück:

- negativ : Das String- Objekt ist kleiner als der übergebene String
- positiv : Das String- Objekt ist größer als der übergebene String
- 0 : beide sind gleich

Beispiel:

String s1 = “otto”;

String s2 = “hans”;

int erg = s1.compareTo (s2);

Ergebnis: otto ist größer, also enthält erg eine positive Zahl > 0.

Ersetzen und Extrahieren

- **String replace (char oldchar, char newchar)**

Beispiel: String string = “Drei Chinesen mit dem Kontrabass”;
// Ersetzen Sie in string alle e durch o.

- **String substring (int beginindex)**

Beispiel: String string = “max moritz”;
// Legen Sie den Nachnamen (moritz) in “ String nachname“ ab.

- **String substring (int beginindex, int endindex)**

Beispiel: String string = “Sonnenblumenwiese”;
// Schneiden Sie aus “Sonnenblumenwiese“ “blume“ aus.

- **Länge eines String ermitteln**

- **int length ()**

Beispiel: System.out.println (“\nLänge des String: ” + string.length ());

Konvertieren und Zerlegen

Die Datentypen boolean, char, char [], double, float, int, long werden mit der valueOf- Methode der Klasse String in String’ s konvertiert.

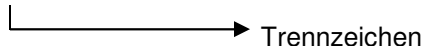
- **static String valueOf (boolean b)**

Beispiel: boolean b = true;
System.out.println (String.valueOf (b));
Ergebnis : true

System.out.println (String.valueOf (10));
Ergebnis: 10 (Datentyp : String)

String x = “hallo”;
System.out.println (String.valueOf (x.equals (“bello”)));
Ergebnis: false

- **split (zerlegt String’ s in mehrere Teile)**

Beispiel: split (“\ _ “);
 Trennzeichen
String string = “28.08.1749”;
String [] ergebnis =string.split (“. “);
for (int i = 0; i < ergebnis.length ; i ++)
{
 System.out.println (ergebnis [i]);
}

Übungen :

1. Vergleichen Sie bitte s2 mit s3.
 String s2 = „hallo“;
 String s3 = new String („hallo“);
 Bei gleich geben Sie „gleich“ aus, bei ungleich „ungleich“. Verwenden Sie zwei verschiedene Vergleichsmethoden.
2. String s4 = „maria ciriello“;
 2.1. Geben Sie die Anfangsbuchstaben des Vor- und des Nachnamens aus.
 2.2. Geben Sie den Nachnamen aus.
3. Geben Sie die Länge des Strings String s5 = „Sonnenblumenwiese“ aus.
4. Ein Datum liegt im String- Format vor: „28.8.1749“. Die Elemente Tag, Monat, Jahr sollen im int- Format verarbeitet werden. Ermitteln Sie für Tag, Monat und Jahr das int- Format, legen Sie ein Datumsobjekt an und geben Sie das Datum dieses Objektes aus in der Form 28.8.1749.
5. Transformieren Sie den String „Drei Chinesen mit dem Kontrabass“ in ein Array und Verwandeln Sie alle Vokale in den Vokal e.
6. Stellen Sie die Ziffern des Strings : **“2468097531“** in einem int- Array zur Verfügung
7. Geben Sie folgenden Text spaltenweise in das Array „rechteck“ ein:
 char rechteck [4] [6];
 Text: EAAEITG NZTMEE A DUKJIS
 ↑ ↑
 ein Blank zwei Blanks

Geben Sie den Inhalt von rechteck anschließend zeilenweise aus.

zu 2. (Übungen zu Strings)

Die substring- Methode nimmt die Veränderung nicht am Originalstring vor, sondern erzeugt eine Kopie, die mit dem gewünschten Inhalt gefüllt wird. Diese Kopie gibt sie dann an den Aufrufer zurück (hier s4), der das Ergebnis erneut an s4 zuweist und damit die Originalinstanz für den Garbage Collector frei gibt. Durch den Referenzcharakter von Objekten und das automatische Speichermanagement entsteht also der Eindruck, als wären Stringobjekte veränderlich.

Ein Objekt vom Typ String ist aber nach dem Erzeugen unveränderlich. Es stellt eine von der Lage her festgelegte Zeichenfolge dar.

Unterschied	
Array	String
char [] array = { ‚a‘, ‚b‘, ‚c‘ } ;	
array [0] = ‚x‘;	
char c = array [0];	
int laenge = array.length;	

1.3. Aufzähltypen **enum**

Werte von Datentypen sind oft ein kleiner konstanter Wertevorrat, z.B. stehen für

- Anrede HERR, FRAU, FIRMA
- Farbe BLAU, GRUEN, GELB, LILA, ROT usw.
- Wochentag MONTAG, DIENSTAG, MITTWOCH usw. zur Verfügung.

Aufzählungen sind als Klassen und ihre Werte als Objekte realisiert.

Mit dem Schlüsselwort **enum** kann ein Aufzählungstyp definiert werden.

In seiner einfachsten Form hat er folgende Syntax:

enum Typname (Wert_1, Wert_2, Wert_3, usw.)

Aufzähltypen haben noch weitere nützliche Eigenschaften:

- Sie haben eine `toString ()`-Methode, die den Namen des `enum`-Objektes im Klartext ausgibt. Ist sie codiert, wird der Wert des Objektes ausgegeben.
- Sie können mit `equals` auf Gleichheit geprüft werden
- Sie können in `switch`-Anweisungen verwendet werden.
- Mit der Methode „`values ()`“, die einen Iterator liefert, können alle Werte nacheinander durchlaufen werden.

...

Beispiel zur einfachen Form:

```
public enum Ampel { ROT, GRUEN, GELB }
```

```
/*  
    steht in einer eigenen enum- Klasse, die den Namen Ampel hat.  
    Diese einfache enum- Klasse besitzt nur die Methoden der Aufzähltypen.  
*/
```

```
public class Anwendung
```

```
{  
    public static void main ( String [ ] args )  
    {  
        Ampel ampel = Ampel.ROT;  
        switch (ampel )  
        {  
            case ROT : System.out.println ( "stehen bleiben" ); break;  
            case GRUEN : System.out.println ( "weiterfahren" );break;  
            case GELB : System.out.println ( "bremsen – anhalten " ); break;  
        }  
        System.out.println ( ampel );  
    } // end main  
} // end class Anwendung
```

Beispiel zur erweiterten Form einer Aufzählungsklasse

```
public enum Farbe  
{  
    // kann durch weitere Attribute und Methoden erweitert werden  
    ROT ( "rot" ),  
    GRUEN ( "gruen" ),  
    GELB ( "gelb" );  
    // Attribut  
    private String farbe;  
  
    // privater Konstruktor  
    private Farbe ( String farbe )  
    {  
        this.farbe = farbe;  
    }  
    // weitere Methoden  
    public static void vergleichen ( Farbe f1, Farbe f2 )  
    {  
        System.out.print ( f1 );  
        System.out.print ( f1.equals ( f2 ) ? " = " : " != " ) ;  
        System.out.println ( f2 + "\n");  
        // toString braucht nicht codiert zu werden.  
        // ( Der Name des Objektes wird ausgegeben. ) }  
}
```

```

public static String auswerten ( Farbe f )
{
    String zurueck = null;
    switch ( f )
    {
        case ROT      : zurueck = "rot ist die Liebe "; break;
        case GRUEN    : zurueck = "grün ist die Hoffnung "; break;
        case GELB     : zurueck = "gelb ist die Eifersucht "; break;
        default       : zurueck = "Fehler";
    }
    return zurueck;
}
public static void alleAnzeigen ( )
{
    for ( Farbe f : Farbe.values ( ) )
    {
        System.out.println ( f );
    }
}
public String gibFarbe ( )
{
    return farbe;
}
} // end enum Farbe

```

```

////////////////////////////////////
public class Farbe_Anw
{
    public static void main(String[] args)
    {
        Farbe f1 = Farbe.ROT;
        Farbe f2 = Farbe.GRUEN;
        Farbe f3 = Farbe.GELB;
        Farbe f4 = Farbe.ROT;
        Farbe f5 = Farbe.ROT;

        ArrayList <Farbe> farben = new ArrayList <Farbe> ( );

        farben.add ( f1 );
        farben.add ( f2 );
        farben.add ( f3 );
        farben.add ( f4 );
        farben.add ( f5 );

        for ( Farbe f : farben )
            System.out.println ( f );

        System.out.println ( "000000000000000000000000 " );
        f1.alleAnzeigen ( );
        System.out.println ( "1111111111111111111111 " );
        Farbe.alleAnzeigen ( );
        System.out.println ( "2222222222222222222222 " );

        System.out.println ( ( f1 == f4 ) + "\n" );

        f1 = f2;
        System.out.println( f1);
        System.out.println( f2);
        System.out.println ( "3333333333333333333333 " );
        System.out.println ( Farbe.auswerten ( f2 ) );
    }
}

```

Regeln:

1. enum- Werte können als Attribute in Klassen eingefügt werden. Sie werden mit dem enum- Namen angesprochen.
2. Werden in einer Anwendung oder in der enum- Klasse Methoden von enum gebraucht, die **statisch** sind, müssen die Methoden mit dem enum- Namen angesprochen werden, z.B.
`for (Farbe f : Farbe.values ()) System.out.println (f);`
3. Der Konstruktor ist private und darf nicht statisch sein.
4. Eine enum- Datei hat ein privates Attribut, in das per Konstruktor ein entsprechender Wert (aus allen enum- Werten) eingesetzt wird.

Übung_1

4 Farben können gewinnen:

Gelb	: 100 Euro
Rot	: 1000 Euro
Grün	: 5000 Euro
Blau	: 10000 Euro
Weiss	: kein Gewinn

1. Erstellen Sie eine enum- Klasse Gewinn, und codieren Sie
 - die erforderlichen Attribute
 - einen Konstruktor
 - eine Methode **public static void gewinn (Farbe f)**, die in Abhängigkeit von der übergebenen Farbe den entsprechenden Gewinn am Bildschirm anzeigt.
2. Nehmen Sie in einer main- Funktion alle oben angegebenen Farben in eine typsichere ArrayList auf. Mit Hilfe des Zufallsgenerators Random ermitteln Sie eine Position in der ArrayList. Die an dieser Position abgelegte Farbe hat gewonnen (oder nicht). Mit der Methode „gewinn “ können Sie dann den Gewinn (oder : "Leider haben Sie nicht gewonnen ") anzeigen.

Übung_2

In einer Warenhaussoftware sollen Produkte verwaltet werden.

1. Schreiben Sie eine Klasse Produkt. Ein Produktobjekt besteht aus zwei Attributen. Das erste Attribut ist die Produktbeschreibung, Datentyp : String, das zweite ist der Produktpreis, enum- Datentyp : **ProduktPreis**
Die Klasse Produkt soll einen Konstruktor besitzen, der die Werte für beide Attribute übergeben bekommt, ebenfalls eine
“ public String toString () “- und zwei get- Methoden.
2. Schreiben Sie eine Klasse Produktbeschreibung, die eine Produktbeschreibung repräsentiert. Die Klasse hat zwei Attribute, eine Produktgruppe und einen Beschreibungstext. Beide werden durch die Klasse String repräsentiert. Implementieren Sie einen Konstruktor und die Methode “ public String toString () “. (Der Ergebnis- String soll die Produktgruppe und den Beschreibungstext enthalten.)
3. Das Preissystem der Warenhaussoftware soll umgestellt werden. Produktpreise sollen nicht mehr beliebig vergeben werden können. Stattdessen gibt es vier Preisstufen, denen ein Produkt zugeordnet werden kann.

Die Stufen sind:

LOW	(9,99),	Datentyp double
MEDIUM	(19,99),	
HIGH	(49,99) und	
EXCLUSIVE	(99,99).	

Innerhalb der Warenhaussoftware sollen diese Preisstufen durch einen Enum- Typ (Produktpreis) repräsentiert werden.

Der Enum- Typ muß zusätzlich eine “public String toString () “ – Methode (**Datentyp der Rückgabe : String**), einen Konstruktor, der den Preis als Argument erhält, und eine get- Methode enthalten,

4. Legen Sie in einer main- Funktion zwei Produktobjekte an, und speichern Sie diese in einem Array.
Verwenden Sie eine for- each- Schleife, und geben die Produktobjekte aus.

1.4. Instanzvariable, lokale Variable , Klassenvariable, Klassenmethoden

(Eine Variable hat immer einen Datentyp)

Instanzvariable

- Sie dienen zur Speicherung des Zustandes oder der Eigenschaften eines Objektes.
- Jede Instanz besitzt ihre eigenen Instanzvariablen.

Lebensdauer

- Instanzvariable haben die gleiche Lebensdauer wie das zugehörige Objekt.

Sichtbarkeit innerhalb der Klasse

- Sie sind mit **private** angelegt, sichtbar innerhalb eines Objektes der Klasse selbst, aber nicht in Klassenmethoden

Lokale Variable

- Sie werden nur innerhalb einer Methode, bzw. eines Blocks angelegt.
- Sie werden bei jedem Methodenaufruf neu angelegt.
- Auch formale Parameter einer Methode stellen lokale Variable dar.

Lebensdauer

- Sie leben vom Zeitpunkt der Deklaration bis zum Ende des aktuellen Blocks, spätestens bis zum Methodenende.

Sichtbarkeit innerhalb der Klasse

- nur innerhalb des Blockes, in dem sie deklariert wurden, auf jeden Fall nur innerhalb einer Methode

Klassenvariable

- Sie dienen zur Speicherung von instanzübergreifenden Informationen
- Pro Klasse existiert eine Klassenvariable nur einmal und ist damit unabhängig von den Objekten
- Sie werden mit static deklariert
- Die Initialisierung ist unabhängig von den Objekten der Klasse
Beispiel: // Attribute
private String bezeichnung;
private double preis;
private static int anzahlArtikel = 0; // Klassenvariable
- Ein Zugriff über this. ist nicht möglich.

Lebensdauer

- Sie existieren ab dem ersten Zugriff auf die Klasse durch Instantiierung eines Objektes oder durch Aufruf einer Klassenmethode bis zum Ende des Programmes.

Sichtbarkeit innerhalb der Klasse

- Sie sind für alle Objekte und Methoden der Klasse sichtbar

/* Klassenmethoden

Sie beziehen sich nicht auf ein bestimmtes Objekt, sondern nur auf die Klasse. Der Aufruf von Klassenmethoden erfolgt über die Klasse oder über jedes Objekt der Klasse und ist unabhängig von der Existenz von Objekten dieser Klasse, Die Definition von Klassenmethoden erfolgt durch Voranstellen von static vor der Methodendefinition.

Format:

```
class Klasse
{
    .....
    static Datentyp der Rückgabe Methodenname ( ggf Parameter )
}
```

*/

2. Ein- Ausgabe über Tastatur und Bildschirm (Standard- E/ A)

Ein Javaprogramm kommuniziert generell über Streams mit der Außenwelt. Ein Stream ist ein Kommunikationskanal, ein abstraktes Konstrukt, dessen Fähigkeit darin besteht, Zeichen von einem imaginären Eingabegerät zu lesen und Zeichen

auf ein imaginäres Ausgabegerät zu schreiben. Erst konkrete Unterklassen von Streams bieten die Zugriffsroutinen für konkrete E/ A- Geräte an, wie z.B. Tastatur, Festplatte, Bildschirm.

Standard- E/ A

In der Klasse `java.lang.System` werden drei Referenzen auf Streamobjekte erzeugt:

```
public final class System
{
    public static InputStreamReader in;
    public static PrintStream * out;          * abgeleitet von OutputStream
    public static PrintStream err;
    ...
}
```

Diese Referenzen können ab Programmstart automatisch genutzt werden, z.B. wird das Objekt `in` bei Programmstart so vorbereitet, daß es dazu verwendet werden kann, Eingabedaten von der Konsole zu lesen. Das Objekt `out` wird beim Starten des Programmes so initialisiert, daß seine Ausgabe auf die Standardausgabe (Bildschirm) geleitet wird.

Lesen von der Standardeingabe (Tastatur)

1. Das Objekt `System.in` besteht seit Starten des Programmes (kann Eingabedaten von der Konsole lesen)
2. Es wird ein Objekt vom Typ `InputStreamReader` erstellt. Dem Konstruktor wird ein Verweis auf `System.in` mitgegeben. Das `InputStreamReader`- Objekt kann u.a. Bytes (1 Zeichen = 1 Byte) in Zeichen (1 Zeichen = 2 Bytes) konvertieren, d.h. die Daten werden in das Unicode- Format konvertiert.

```
InputStreamReader isr = new InputStreamReader ( System.in );
```

3. Es wird ein Objekt vom Typ `BufferedReader` erstellt. Dem Konstruktor wird ein Objekt vom Typ `InputStreamReader` übergeben. `BufferedReader` bietet u.a. gepuffertes Lesen an, kann Zeichen (1 Zeichen = 2 Bytes) lesen und die gelesenen Daten im **String- Format** an den Aufrufer zurückgeben. (Methode: **public String readLine ()**)

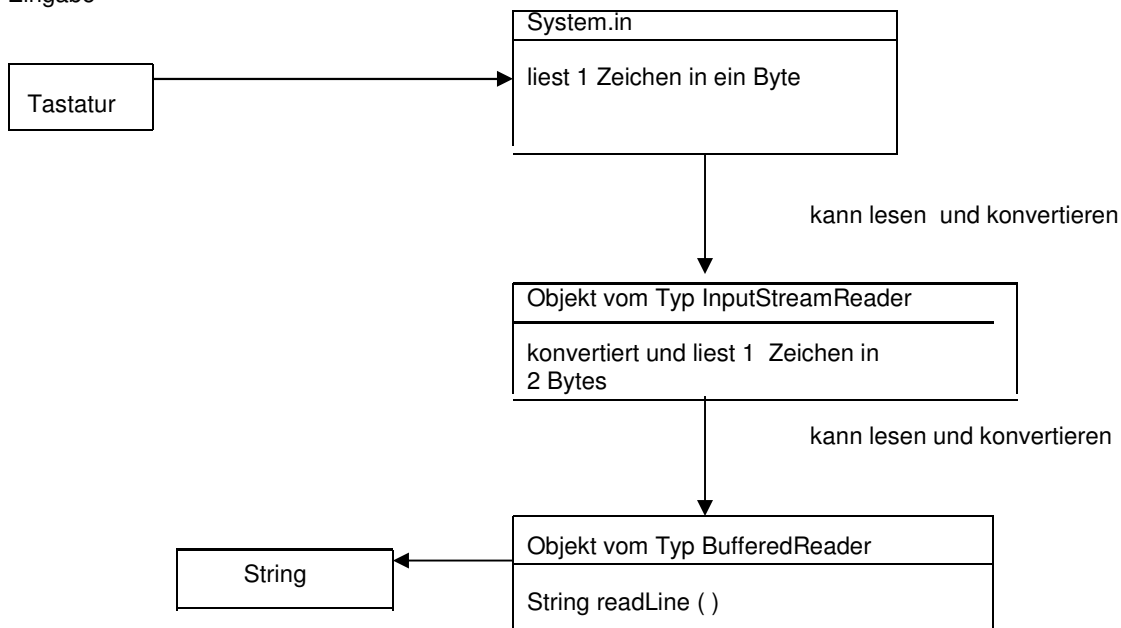
```
BufferedReader br = new BufferedReader ( isr );
```

Dieser Lesevorgang kann auch in einem Schritt codiert werden:

```
BufferedReader br = new BufferedReader ( new InputStreamReader ( System.in ) );
```

Über Tastatur eingelesene Daten müssen in einem Bereich mit String- Format abgelegt werden.

Eingabe



Beispiele:

Voraussetzungen:

1. Der Eingabebereich im Programm muß für Tastatureingaben den Datentyp String haben, oder die Eingabe muß, falls der Eingabebereich einen anderen Datentyp hat, in den Datentyp String umgewandelt werden. (Beispiel : `int zahl = Integer.parseInt (br.readLine ());`);
2. Einlesen über die Konsole verlangt `try` und `catch`.)

Beispiel 1:

```
public class Anwendung
{
    public static void main (String [ ] args )
    {
        BufferedReader br = new BufferedReader ( new InputStreamReader ( System.in ) );

        String eingabe = null;
        System.out.println ( "Bitte gebn Sie einen Text ein " );

        // Einlesen über die Konsole verlangt try und catch
        try
        {
            eingabe = br.readLine ( );
        }
        catch ( IOException e )
        {
            e.printStackTrace ( );
        }

        // Verarbeitung
    } // end main
} // end class Anwendung
```

Beispiel 2:

```
public class Anwendung
{
    public static void main ( String [ ] args )
    {
        BufferedReader br = new BufferedReader ( new InputStreamReader ( System.in ) );
        int zahl_1 = 0;
        int zahl_2 = 0;
        System.out.println ( "Bitte geben Sie zwei ganze Zahlen ein : " );
        try
        {
            zahl_1 = Integer.parseInt ( br.readLine ( ) );
            zahl_2 = Integer.parseInt ( br.readLine ( ) );
        }
        catch ( IOException e )
        {
            System.out.println ( "Fehler bei der Engabe " );
        }
        System.out.println ( zahl_1 * zahl_2 );
    } // end main
} // end class Anwendung
```

Schreiben über die Standardausgabe (Bildschirm)

Mit `println (...)` und `print (...)` lassen sich Zeichenketten und einfache Datentypen ausgeben. Es sind Methoden der Klasse `PrintStream`, die dem Objekt `System.out` zur Verfügung stehen.

Die Klasse `PrintStream` bietet auch viele überladenen Methoden an, die keine I/O-Exception auslösen. Für alle primitiven Datentypen existieren Ausgabemethoden. Für Referenztypen gibt es die Methode:

<code>void print (Object object)</code>

// auch `println`

Bei Ausführung dieser Methode wird vor der eigentlichen Ausgabe die Anweisung **`,String string = object.toString ();'`** abgearbeitet.

Die Methode **`,String toString ()'`** ist in der Klasse `java.lang.Object` definiert und erzeugt eine Stringdarstellung des Objektes `object`. Der String besteht aus:

- Klassename
- Dem Zeichen @
- Der vorzeichenlosen Hexadezimaldarstellung des Hashcodes dieses Objekts

Diese Darstellung ist für die wenigsten Objekte zufriedenstellend. Deshalb wird die Methode **“String toString ()”** sehr oft überschrieben.

Beispiel 3 (toString)

```
public class Datum
{
    private int tag;
    private int monat;
    private int jahr;

    public Datum ( int tag, int monat, int jahr )
    {
        this.tag = tag;
        this.monat = monat;
        this.jahr = jahr;
    }
    /*
        Ueberschreiben von toString ( ), um ein schoeneres Datumsformat zu erhalten
    */
    public String toString ( )
    {
        return this.tag + "." + this.monat + "." + this.jahr;
    }
}

public class Anwendung
{
    public static void main ( String [ ] args )
    {
        Datum datum = new Datum ( 10. 11. 1759 );
        System.out.println ( datum );
        /*
            Ausgabe, ohne daß die Methode String toString ( ) codiert ist:
            Datum @ Adresse des Datumsobjektes

            Ausgabe, wenn die Methode String toString ( ) codiert ist:
            10.11.1759
        */
    } // end main
} // end class Anwendung
```

Im main wird eine Instanz der Klasse Datum erzeugt und die Referenz datum auf diese Instanz der Methode **println (...)** übergeben. Das System stellt zur Laufzeit fest, daß datum eine Instanz von Datum referenziert und führt , wenn vorhanden , die überschriebene Methode toString () aus.

Übung:

Schreiben Sie eine Klasse Tauschen. Sie hat einen leeren Konstruktor.

Ihre Methode **“String lesenUndTauschen ()”**

fordert einen Text über die Konsole an, liest ihn und gibt ihn getauscht zurück.

Beispiel: hallo wird zu ollah.

3. Testen Sie die Methode in einer main- Funktion

3. Beziehungen zwischen den Klassen

3.1. Assoziationen

Assoziationen sind notwendig, damit Objekte miteinander kommunizieren können. Eine Assoziation beschreibt eine Verbindung zwischen Klassen.

Objektorientierte Assoziationen werden dadurch realisiert, dass eine beteiligte Klasse ein Referenzattribut auf eine andere Klasse erhält. Beziehungsklassen, in denen eine $n : m$ -Beziehung hergestellt wird, sind nicht im strengen Sinne objektorientiert.

Objektorientierte Assoziationen haben eine andere Semantik als relationale Assoziationen im ERM, wo grundsätzlich bidirektionale Assoziationen angenommen werden (Pkey, Fkey), Anfänger mit relationalem Erfahrungshintergrund betrachten Assoziationen fälschlicherweise weiterhin mit relationaler Semantik. (Österreich).

3.1.1. 1 : 1- Assoziation

Eine Rechnung hat eine Anschrift



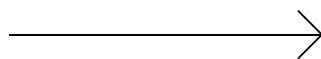
Die Rechnung kann auf die Anschrift zugreifen, die Anschrift weiß jedoch nicht, zu welcher Rechnung sie gehört. Die Beziehung wird dadurch realisiert, daß ein Anschrift- Objekt im Rechnung- Objekt abgelegt wird.

(
Im ERD würde hier eine 1 : 1- Beziehung modelliert:
- Eine Rechnung hat eine Anschrift
- Eine Anschrift gehört zu einer Rechnung.

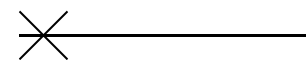
Die Beziehung wird durch PKEY – FKEY hergestellt.

)

Notation:



Navigationsrichtung (Rechnung kann auf Anschrift zugreifen)



Navigationsrichtung ist explizit ausgeschlossen



Navigationsrichtung ist nicht definiert, also ist noch alles möglich

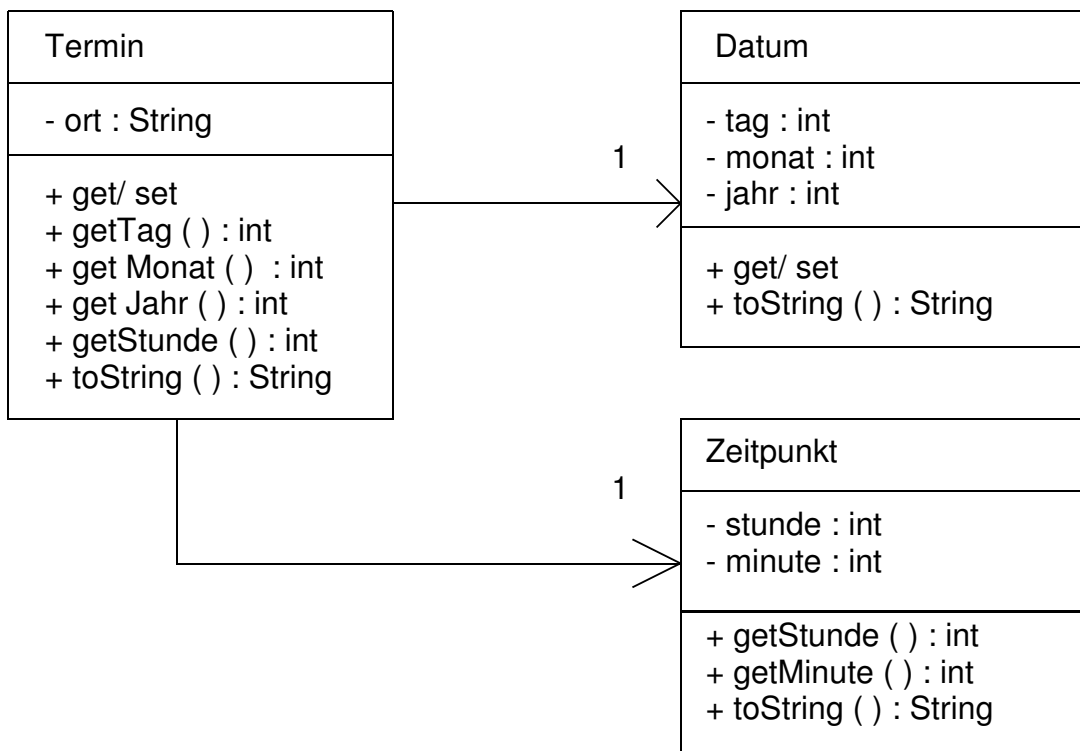


Leserichtung (Rechnung hat eine Anschrift, nicht: Anschrift hat eine Rechnung ,

Zu einem Objekt gehört / gehören

1	genau 1 Objekt
0..1	kein oder ein Objekt
*	kein, ein oder viele Objekte
0..*	kein, ein oder viele Objekte
1..*	ein oder mehrere Objekte
2..8	2 bis 8 Objekte

Beispiel



```
public class Termin
{
    // Attribute
    private String ort;
    private Datum datum;
    private Zeitpunkt zeitpunkt;

    // Konstruktor
    public Termin( String ort, Datum datum, Zeitpunkt zeitpunkt )
    {
        this.ort = ort;
        this.datum = datum;
        this.zeitpunkt = zeitpunkt;
    }

    // weitere Methoden
    public String getOrt ( )
    {
        return this.ort;
    }
    public int getTag ( )
    {
        // einfuegen
    }
    public int getMonat ( )
    {
        // einfuegen
    }
    public int getJahr ( )
    {
        // einfuegen
    }
}
```

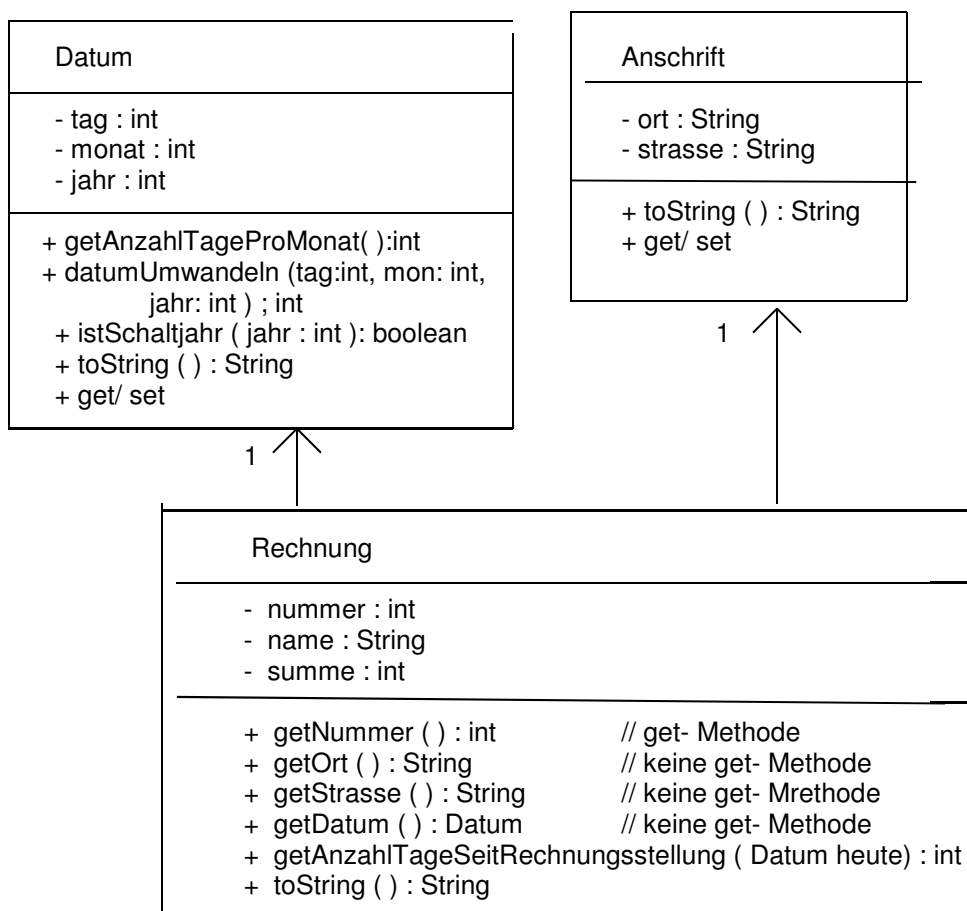
```

    public int getStunde ( )
    {
        return this.zeitpunkt.getStunde ( );
    }
    public int getMinute ( )
    {
        return this.zeitpunkt.getMinute ( );
    }
    public String toString ( )
    {
        return "\n" + this.ort + "\n" + // einfuegen
    }
}

```

Übung

UML_Klassendiagramm



Codieren Sie die oben angezeigten Klassen, und testen Sie Ihre Implementierung in einer main- Funktion.

3.1.2. 1 : n- Assoziation



Ein Automat hat **ein bis viele** Account's. Ein Account gehört zu **einem Automaten**.
Im Beispiel kennt die Bank ihre Konten, ein Konto weiß nicht, zu welcher Bank es gehört.

Der Automat hat sinnvollerweise ein **Objekt einer Collectionklasse** als Attribut in sich, das eine unbestimmte Zahl von Account's aufnehmen kann. Ein Array als Attribut der Klasse Automat kann nur eine bestimmte Anzahl von Account's aufnehmen.

Die Klasse `ArrayList` z.B. vertritt ein Array mit variabler Länge. Der Zugriff auf die Elemente erfolgt über Indizes. Da in einer `ArrayList` jedes Exemplar einer von `Object` abgeleiteten Klasse Platz findet, ist eine `ArrayList` nicht auf bestimmte Datentypen fixiert. Zur Sicherheit wird deshalb beim Anlegen einer `ArrayList` festgelegt, welchen Datentyp die zu speichernden Objekte haben müssen.

Beispiel

```
public class Account
{
    // Attribute
    private int ktoNummer;
    private String name;
    private int guthaben;
    // Konstruktoren
    public Account ( ) { }
    public Account ( int ktoNummer, String name )
    {
        this.ktoNummer = ktoNummer;
        this.name = name;
        this.guthaben = 1;
    }
    // weitere Methoden
    boolean kontoGleich ( int nr )
    {
        return this.ktoNummer == nr ;
    }
    public void setKtoNummer ( int ktoNummer )
    {
        this.ktoNummer = ktoNummer;
    }
    public void setName ( String name )
    {
        this.name = name;
    }
    public void setGuthaben ( int guthaben )
    {
        this.guthaben = guthaben;
    }
    public int getKtoNummer ( )
    {
        return this.ktoNummer;
    }
    public String getName ( )
    {
        return this.name;
    }
    public int getGuthaben ( )
    {
        return this.guthaben;
    }
}
```

```

public void abheben ( int betrag
{
    this.guthaben = this.guthaben - betrag;
}
public void einzahlen ( int betrag )
{
    this.guthaben = this.guthaben + betrag;
}
public String toString ( )
{
    return this.ktoNummer + " Name: " + this.name +
        " Guthaben: " + this.guthaben ;
}
}
import java.util.*;
import java.util.Iterator.*;
public class Automat
{
    //      Attribut
    //      einfuegen

    //      Konstruktor
    //      einfuegen

    // weitere Methoden
    public void kontoEinfuegen ( int ktoNummer, String name )
    {
        alleKonten.add ( new Account ( ktoNummer, name) );
    }
    public void kontoEinfuegen ( Account konto )
    {
        alleKonten.add ( konto );
    }
    public void kontoLoeschen ( int ktoNummer )
    {
        Account a = null;
        for ( int i = 0; i < alleKonten.size ( ); i ++ )
        {
            a = alleKonten.get ( i );
            if ( a.getKtoNummer ( ) == ktoNummer )
            {
                alleKonten.remove ( a );
            }
        }
    }
    // usw.
}

```

Übungen

Übung 1 Automat

1. Ergänzen Sie die Klasse Automat um folgende Methoden.

```

public class Automat
{
    .....
    //      public String kontoAbfragen ( int ktoNummer )
    //      public String toString ( )
    //      public void abheben ( int kontonummer, int betrag )
    //      public void einzahlen ( int kontonummer, int betrag )
}

```

2. Schreiben Sie eine main- Funktion, die die oben angezeigten Funktionen testet, z.B,

```

public class Anwendung

```



```

{
    public static void main(String[] args)
    {

        //      Im main kann die ArrayList nicht mehr direkt angesprochen werden,
        //      das geht nur über die Methoden der Klasse Bank

        //      dem Automat drei Konten hinzufuegen ( Nummer, Name )
        //      alle Account's anzeigen
        //      100 Euro auf Account 1 einzahlen
        //      20 Euro von Account 1 abheben
        //      Account Nr. 2 löschen
    } // end main
} // end class Anwendung

```

Übung 2 Kartenstapel

Die Klasse Karte liegt Ihnen vor : Lösungsvorschlag, Seite 31 : Karte

Legen Sie einen Kartenstapel an. Er hat 32 Karten.

Jede Karte hat einen Typ (Herz, Kreuz, Pic, Karo) und eine Art (7, 8, 9, 10, Bube, Dame, König, Ass)..

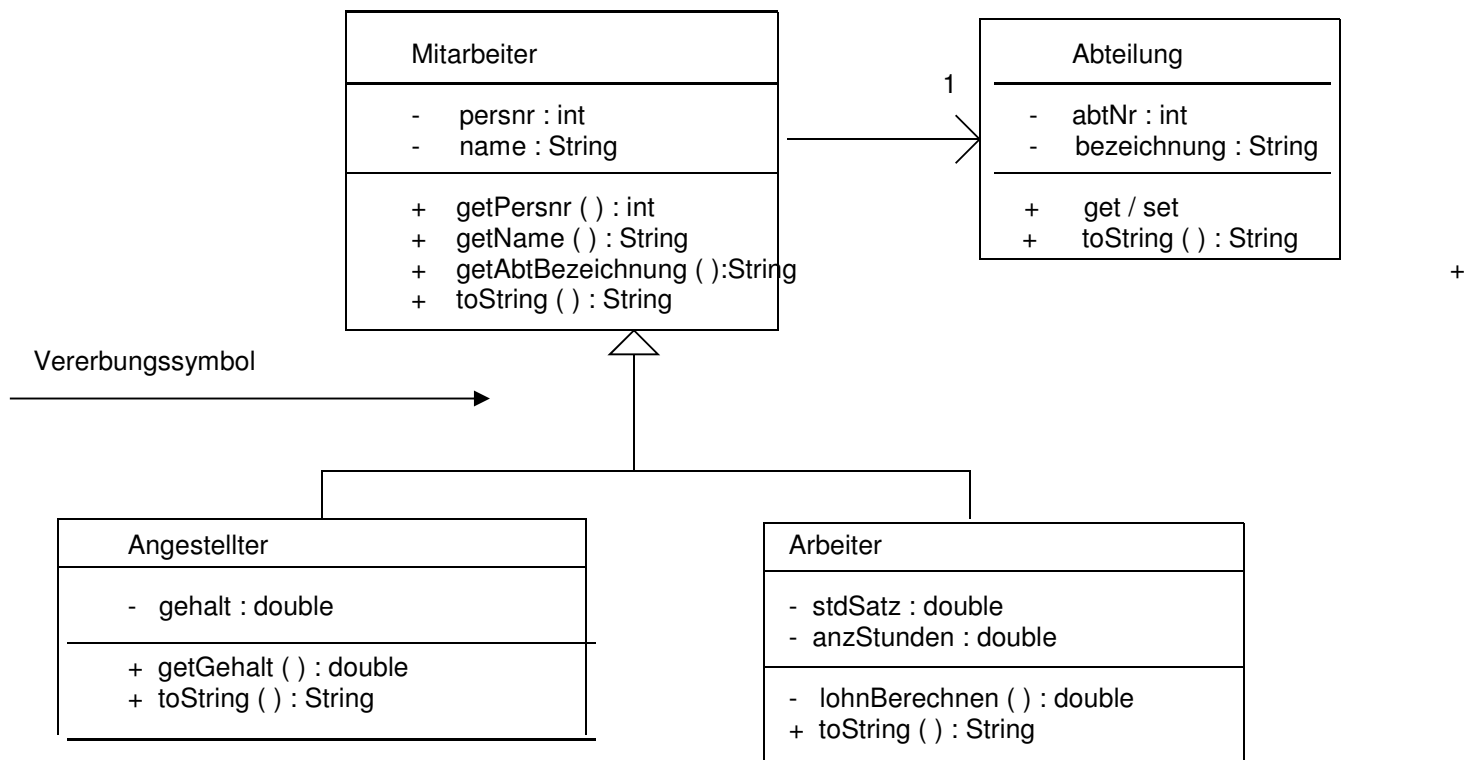
Der Konstruktor des Stapels initialisiert in einer (inneren und äußeren) Schleife
Jede Karte und fügt sie in den Stapel ein

Weitere Methoden :

- Ziehen einer Karte
- Mischen des Stapels
- alle Karten anzeigen

3.2. Vererbung

3.2.1. Einfache Vererbung



Regeln:

(UML- Klassendiagramm und Codierung)

1. Attribute und Assoziationen werden direkt vererbt und in den erbenden Klassen nicht aufgeführt bzw. codiert.
2. Methoden werden bei der Vererbung unterschiedlich behandelt:
In einer Unterklasse wird eine Methode
 - a) **übernommen**,
wenn sie in der erbenden Klasse ohne die geringste Änderung verwendet werden kann. Sie wird weder im UML- Diagramm aufgeführt noch im Programm codiert.
 - b) **redefiniert oder überschrieben/ überlagert**,
wenn eine neue Implementierung in der Unterklasse eine vorliegende Implementierung einer Oberklasse erweitert oder ersetzt.
 - c) **hinzugefügt**,
wenn von der Sache her eine neue Funktion in der Unterklasse erforderlich wird.
 - d) **implementiert**,
wenn die Oberklasse eine abstrakte Methode hat.

```

public class Mitarbeiter
{
    // Attribute
    private int persnr;
    private String name;
    private Abteilung abteilung;

    // Konstruktor
    public Mitarbeiter ( int persnr, String name, Abteilung abteilung )
    {
        this.persnr = persnr;
        this.name = name;
        this.abteilung = abteilung;
    }

    // weitere Methoden
    public int getPersNr ( )
    {
        return this.persnr;
    }
    public String getName ( )
    {
        return this.name;
    }
    public String getAbteilungsBezeichnung ( )
    {
        return this.abteilung.getAbteilungsBezeichnung ( );
    }
    public int getAbteilungsNummer ( )
    {
        return this.abteilung.getAbteilungsNummer( );
    }
    public String toString ( )
    {
        return "\nPersonalnummer : " + this.persnr + " Name : " +
            this.name + " " + this.abteilung + "\n" ;
    }
} // end class Mitarbeiter

```

```

public class Abteilung
{
    // Attribute
    private int abtnr;
    private String bezeichnung;

    // Konstruktor
    public Abteilung ( int abtnr, String bezeichnung )
    {
        this.abtnr = abtnr;
        this.bezeichnung = bezeichnung;
    }
}

```

```

// weitere Methoden

public int getAbteilungsNummer ( )
{
    return this.abtnr;
}
public String getAbteilungsBezeichnung ( )
{
    return this.bezeichnung;
}
public String toString ( )
{
    return "Abteilungsnummer : " + this.abtnr + " Bezeichnung : " +
        this.bezeichnung + " ";
}
} // end class Abteilung

```

```

public class Arbeiter // einfuegen
{
    // Attribute
    private double stdSatz;
    private double anzStunden;
    // Konstruktor
    Arbeiter ( // einfuegen )
    {

        // einfuegen

    }

    // weitere Methoden
    public void aendereStdSatz ( double stdSatz )
    {
        this.stdSatz = stdSatz;
    }
    public void aendereAnzStunden ( double anzStunden )
    {
        this.anzStunden = anzStunden;
    }
    public String toString ( ) // Elternmethode überschreiben
    {

        // einfuegen

    }
    private double lohnBerechnen ( )
    {
        return this.stdSatz * this.anzStunden;
    }
} // end class Arbeiter

```

```

public class Angestellter          // einfuegen
{
    // Attribut
    private double gehalt;
    // Konstruktor
    public Angestellter (          // einfuegen
    {

        // einfuegen

    }
    // weitere Methoden
    public double getGehalt ( )
    {
        return this.gehalt;
    }
    public String toString ( )
    {
        // einfuegen
    }

} // end class Angestellter

```

```

public class Anwendung
{
    public static void main ( String [ ] args )
    {
        Abteilung ab1 = new Abteilung ( 1, "edv");
        Abteilung ab2 = new Abteilung ( 2, "verkauf");
        Abteilung ab3 = new Abteilung ( 3, "werkstatt");
        Mitarbeiter a1 = new Mitarbeiter ( 1, "max", ab2 );
        Angestellter a2 = new Angestellter ( 2, "eva", 5000.0 ,ab1 );
        Arbeiter a3 = new Arbeiter ( 3, "otto", ab3, 25.0, 40.0 );

        System.out.println ( a1 + "\n " + a2 + "\n" + a3 );
        /*
            „toString“ wird überschrieben (= gleiche Signatur,
            aber unterschiedliche Implementierung) und mit
            dem entsprechenden Objekt unterschiedlich ausgeführt
            = einfache Vererbung
        */
    } // end main
} // end class Anwendung

```

```

Personalnummer : 1   Name : max   Abteilungsnummer : 2   Bezeichnung : verkauf

Personalnummer : 2   Name : eva   Abteilungsnummer : 1   Bezeichnung : edv
Gehalt : 5000.0

Personalnummer : 3   Name : otto   Abteilungsnummer : 3   Bezeichnung : werkstatt
Stundensatz: 25.0 Anzahl Stunden : 40.0 Lohn : 1000.0

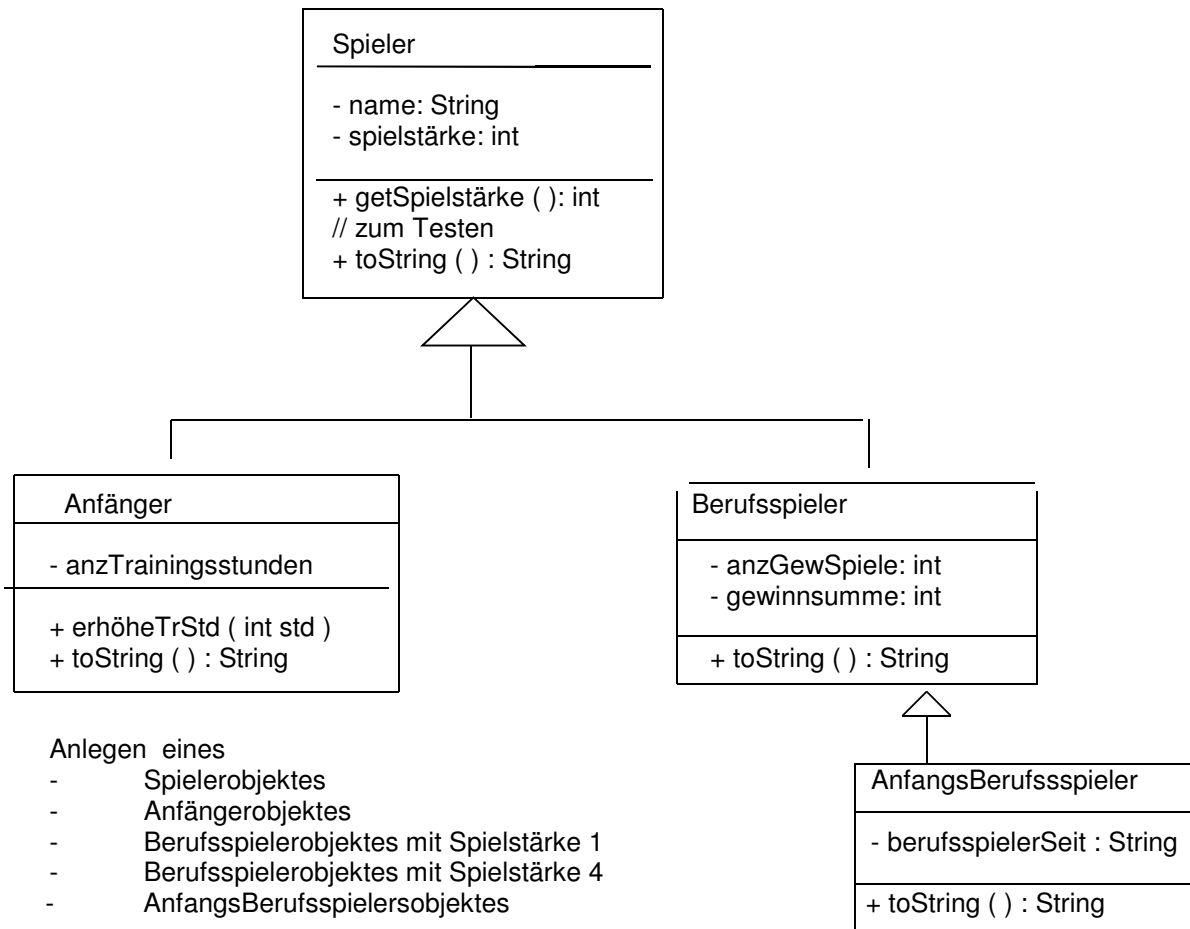
```

3.2.2. Polymorphie

Die wichtigste Voraussetzung zur Realisierung polymorpher Strukturen ist das dynamische Binden.

Dynamisches Binden einer Methode bedeutet, daß die Festlegung, welche Methode innerhalb einer Hierarchie überschriebener Methoden aufgerufen wird, nicht zur Compilierungszeit, sondern zur Laufzeit getroffen wird.

Das System erkennt, wenn in einem Programm einem Basisklassenobjekt ein Objekt einer Unterklasse zugewiesen wurde, daß es die ggf. überschriebene Methode der zugewiesenen Unterklasse ausführen muß.



1. Anlegen eines
 - Spielerobjektes
 - Anfängerobjektes
 - Berufsspielerobjektes mit Spielstärke 1
 - Berufsspielerobjektes mit Spielstärke 4
 - AnfangsBerufsspielersobjektes
2. Ausführen der objekteigenen Methoden "toString ()"
3. Anlegen einer ArrayList vom Typ **Spieler** und einfügen der 5 Spielerobjekte.
(
 das ist eine Zuweisung von unten nach oben: in der Hierarchie.
 spieler = anfaenger, spieler = berufsspieler
)
4. Ausführen der entsprechenden toString- Methoden in einer Schleife über die ArrayList.
5. Zuweisung von oben nach unten:
Anfaenger a = new Spieler ("Anfaenger Max" ,1);
System.out.println (a);

3.2.3. Konstruktoren und Vererbung

(1) Standardkonstruktor in der Elternklasse ohne Codierung

```
public class A
{
    //      Attribut
    private String a;

    //      Standardkonstruktor ohne Codierung
} // end class A
public class B      extends A
{
    //      Attribut
    private String b;

    //      Konstruktor
    public B ( String b )
    {
        this.b = b;
        System.out.println ( b );
    }
}
public class Anwendung
{
    public static void main ( String [ ] args )
    {
        B b = new B ( "hallo B" );
    }
} // end class Anwendung
```

1. Impliziter Aufruf des Konstruktors der Elternklasse
2. Ausführung des Konstruktors der Kindklasse
3. Ergebnis: hallo B

(2) Standardkonstruktor in der Elternklasse codiert

```
public class A
{
    //      Attribut
    private String a;

    //      codierter Standardkonstruktor
    public A ( )
    {
        this.a = "hallo A" ;
        System.out.println ( a );
    }
} // end class A
public class B      extends A
{
    //      Attribut
    private String b;
    //      Konstruktor
    public B ( String b )
    {
        this.b = b;
        System.out.println ( b );
    }
} // end class B
```

public class Anwendung

```
{
    public static void main ( String [ ] args )
    {
        B b = new B ( "hallo B");
    }
} // end class Anwendung
```

1. Impliziter Aufruf des Konstruktors der Elternklasse
2. Ausführung des Konstruktors der Kindklasse

Ergebnis: hallo A
 hallo B

**(3) Nichtstandardkonstruktor in der Elternklasse codiert.
 Kein expliziter Aufruf des Konstruktors der Elternklasse
 vom Konstruktor der Kindklasse.**

public class A

```
{
    //      Attribut
    private String a;

    //      Nichtstandardkonstruktor
    public A ( String a )
    {
        this.a = a ;
        System.out.println ( ( a );
    }
} // end class A
```

public class B extends A

```
{
    //      Attribut
    private String b;

    //      Konstruktor
    public B ( String b )
    {
        this.b = b;
        System.out.println ( b );
    }
} // end class B
```

public class Anwendung

```
{
    public static void main ( String [ ] args )
    {
        B b = new B ( "hallo B");
    }
} // end class Anwendung
```

1. Implizit wird versucht, den Standardkonstruktor der Elternklasse aufzurufen.
Den gibt es aber nicht. Ein expliziter Aufruf des Konstruktors der Elternklasse im Kindkonstruktor fehlt

2. Fehler (wird schon vom Compiler festgestellt)

**(4) Nichtstandardkonstruktor in der Elternklasse codiert.
Expliziter Aufruf des Konstruktors der Elternklasse
Vom Konstruktor der Kindklase.**

```
public class A
{
    //      Attribut
    private String a;

    //      Nichtstandardkonstruktor
    public A ( String a )
    {
        this.a = a ;
        System.out.println ( ( a );
    }
} // end class A
```

```
public class B      extends A
{
    //      Attribut
    private String b;

    //      Konstruktor
    public B ( String a, String b )
    {
        super ( a );
        this.b = b;
        System.out.println ( b );
    }
} // end class B
```

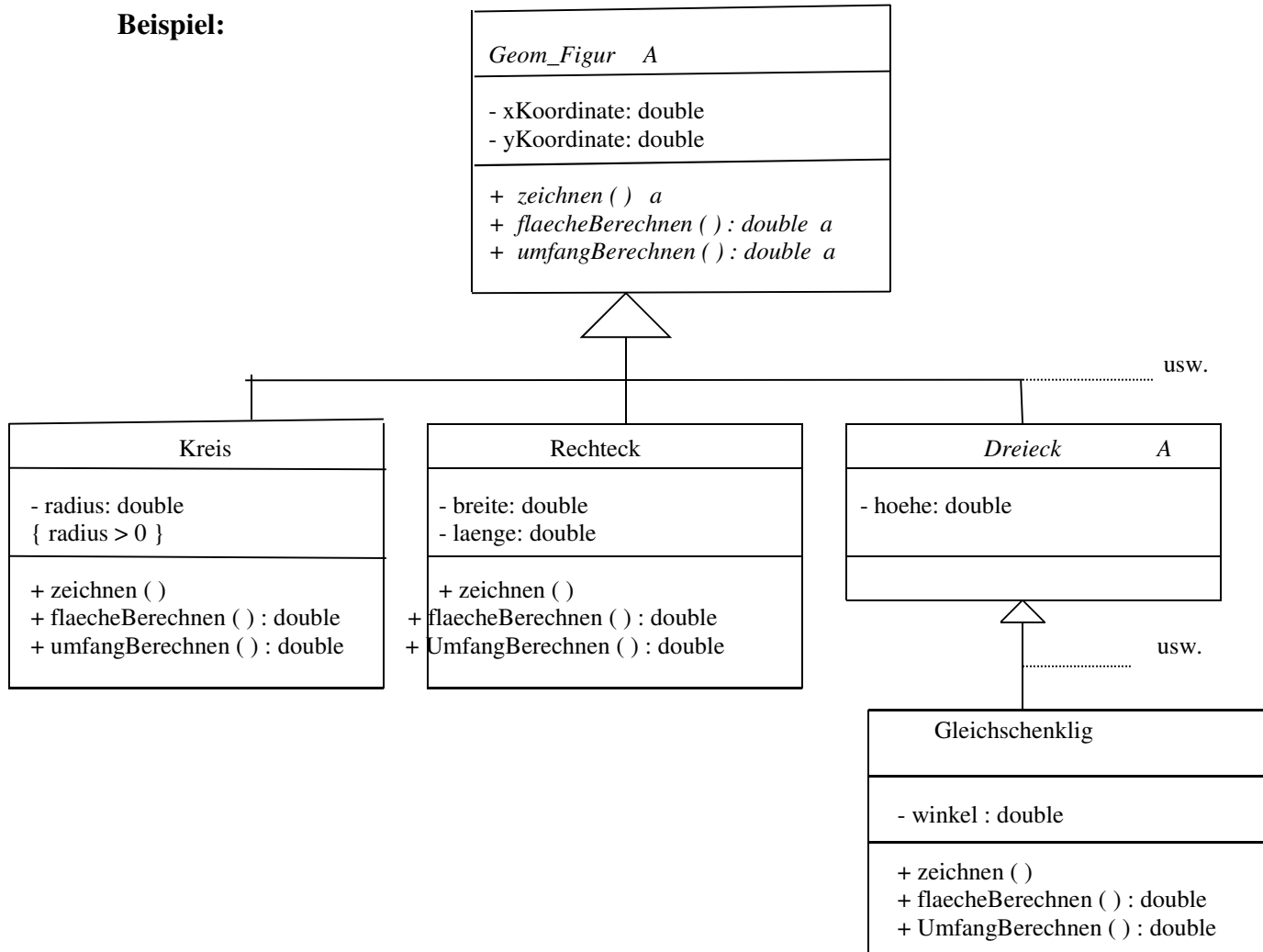
```
public class Anwendung
{
    public static void main ( String [ ] args )
    {
        B b = new B ( "hallo A", "hallo B");
    }
} // end class anwendung
```

1. super (...); ist im Konstruktor der Kindklasse immer der erste Befehl.
2. Der Konstruktor der Elternklasse wird ausgeführt.
3. Der Konstruktor der Kindklasse wird ausgeführt.

Ergebnis: hallo A
 hallo B

3.2.4 Abstrakte Klassen

Beispiel:



Allgemein:

1. Es kann sinnvoll sein, ein Objekt einer abgeleiteten Klasse als ein Objekt seiner Basisklasse zu behandeln. Soll z.B. von vielen geometrischen Figuren die Fläche oder der Umfang berechnet werden, ist es sinnvoll, alle geometrischen Figuren, z.B. Kreis, Rechteck, Linie, Quadrat in einem Array vom Typ `GeomFigur` abzulegen

Es lohnt sich also, alle Methoden, die den abgeleiteten Klassen gemeinsam sind, bereits in der Basisklasse zu deklarieren. Was sich nicht immer lohnt, ist, für alle diese Methoden Implementierungen vorzunehmen.

u.U. kann es zu Fehlern führen, wenn eine Methode in einer abgeleiteten Klasse nicht überschrieben wurde, obwohl die Implementierung der Basisklasse für die abgeleitete Klasse ungeeignet ist. (Vergessen von Überschreiben)

Mit den abstrakten Klassen sind die Vorteile der Container- Verwaltung gewährleistet, und das Vergessen des Überschreibens führt zu einem Compilerfehler, nicht zu einem Programmfehler.

2. Weitaus wichtiger ist das Prinzip der SW- Erstellung:
'Programmiere auf Schnittstellen hin'
 Werden in einer Basisklasse Methoden nur deklariert und nicht implementiert, besteht zwischen Eltern und Kindern eine geringere Bindung, als wenn die Kinder Teile der Implementierungen der Elternmethoden erben. Änderungen in den Eltern schlagen nicht auf die Kinder durch.

DV- Technik bei abstrakten Klassen

```
public abstract Klassenname  
{
```

```
.....
```

```
}
```

Attribute:

Sie können 'protected' sein. So können alle erbenenden Klassen direkt auf sie zugreifen.

Der Philosophie der Objektorientierung entspricht es aber, die Attribute **'private'** anzulegen.

(Information hiding)

Methoden:

1. Auch wenn von abstrakten Klassen keine Objekte angelegt werden können, haben sie einen normalen Konstruktor. Abgeleitete Klassen greifen auf diesen Konstruktor zu. Abstrakte Klassen können auch implementierte Methoden haben.

2. Abstrakte Methoden:

Format:

abstract	Rückgabotyp	name	(ggf.Eingabeparameter)	;
-----------------	-------------	------	--------------------------	---

In erbenenden Klassen werden die abstrakten Methoden ggf. überschrieben.

Ist das nicht der Fall, ist auch die erbenende Klasse eine abstrakte Klasse.

public abstract class GeomFigur

```
{  
    // Attribute  
    private int xKoordinate;  
    private int yKoordinate;  
  
    // Konstruktor  
    public GeomFigur(int xKoordinate, int yKoordinate)  
    {  
        this.xKoordinate = xKoordinate;  
        this.yKoordinate = yKoordinate;  
    }  
    // weitere Methoden  
    public abstract void zeichnen ();  
    public abstract double flaecheBerechnen ();  
    public abstract double umfangBerechnen ();  
} // end class GeomFigur  
  
public class Kreis extends GeomFigur  
{  
    // Attribute  
    private int radius;  
    // Konstruktor  
    public Kreis ( int xKoordinate, int yKoordinate, int radius )  
    {  
        super (xKoordinate, yKoordinate);  
        this.radius = radius;  
    }  
    // weitere Methoden  
    public void zeichnen ()  
    {  
        System.out.println ( "ich bin ein Kreis" );  
    }  
    public double flaecheBerechnen ()  
    {  
        return ( this.radius * this.radius * pi );  
    }  
    public double umfangBerechnen ()  
    {  
        return 2 * this.radius * p1 ;  
    }  
} // end class Kreis
```

```

public class Rechteck           extends GeomFigur
{
    // Attribute
    private int breite;
    private int laenge;
    // Konstruktor
    public Rechteck(int xKoordinate, int yKoordinate, int breite, int laenge)
    {
        super( xKoordinate, yKoordinate );
        this.laenge = laenge;
        this.breite = breite;
    }
    // weitere Methoden
    public void zeichnen ( )
    {
        System.out.println ( "ich bin ein Rechteck" );
    }
    public double flaecheBerechnen ( )
    {
        return  this.laenge * this.breite;
    }
    public double umfangBerechnen ( )
    {
        return  2 ( this.laenge + this.breite ) ;
    }
} // end class Rechteck

```

```

public class Anwendung
{
    public static void main(String[] args)
    {
        GeomFigur [ ] figuren = new GeomFigur [ 3 ];

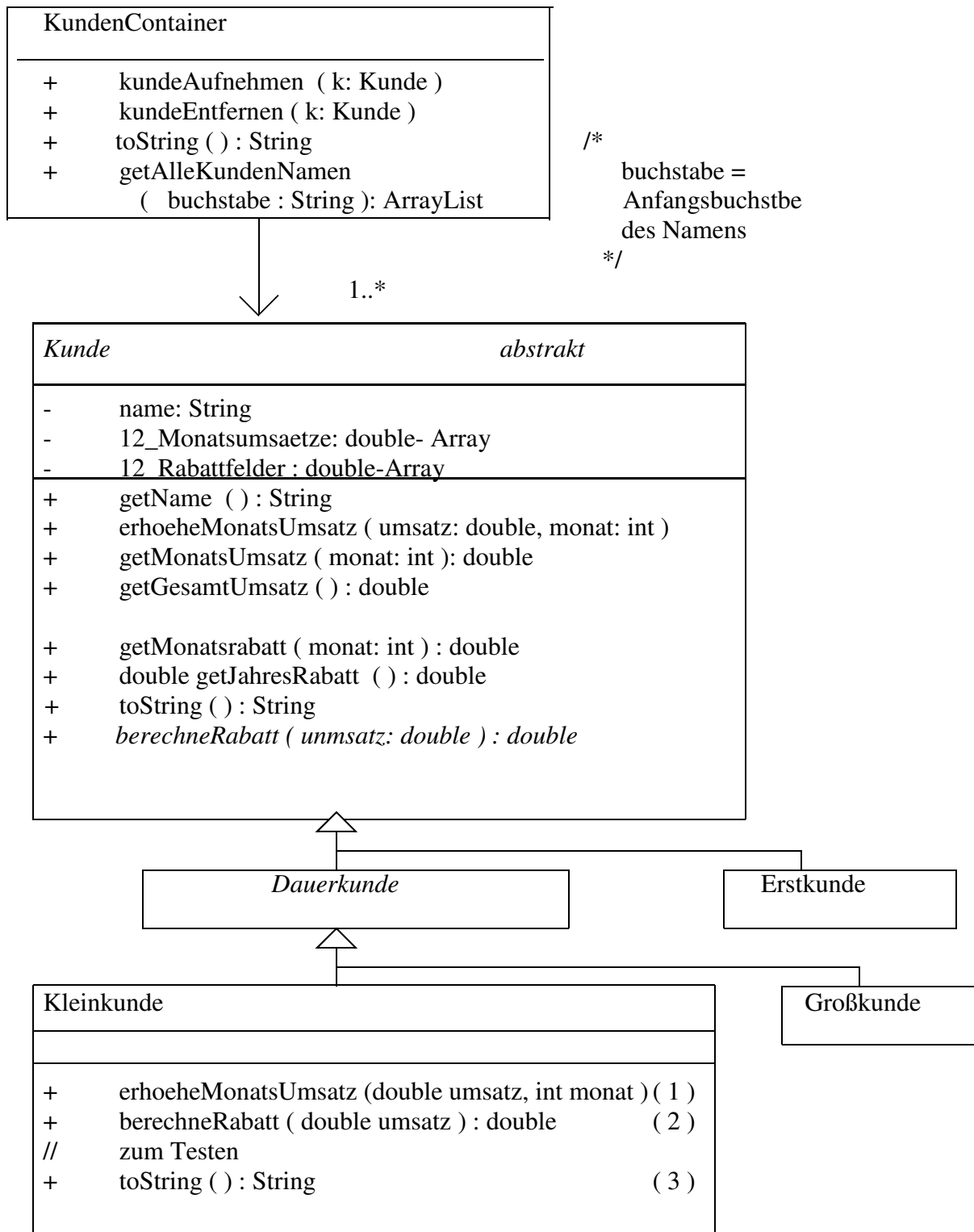
        figuren [ 0 ] = new Kreis (20, 30, 10 );
        figuren [ 1 ] = new Rechteck( 2, 78, 50,50 );
        figuren [ 2 ] = new Linie ( 201, 44, 201, 66 );

        for ( GeomFigur element : figuren )
        {
            element.zeichnen ( );
        }
    }
} // end class Anwendung

```

Übung

abstrakte Klasse



- (1) Die Methode 'erhoeheMonatsUmsatz' ermittelt auch den Rabatt für den neuen Umsatz und addiert ihn in das entsprechende Monats- Rabattfeld.
- (2) Kleinkunden bekommen 2,5 % Rabatt
- (3) Die Methode 'toString ()' zeigt alle Daten eines Kleinkunden

4. Typisierung beliebiger Javaklassen

(**Zunächst Beispiel in den Lösungsvorschlägen: Seite 44**)

Eine typisierte Javaklasse hat ein Attribut mit einem noch nicht festgelegten Datentyp. Im Beispiel hat die generische Klasse Box das Attribut value mit dem Datentyp T).

Im konkreten Anwendungsfall (beim Anlegen eines Objektes) wird für diesen Datentyp dann ein konkreter Datentyp eingesetzt. Im Beispiel werden für T die Datentypen, Artikel, String und Integer eingesetzt. T kann aber auch alle anderen Datentypen annehmen, die die Anwendung verlangt.

```
public class Artikel
```

```
{
    //      Attribute
    private int nummer;
    private String bezeichnung;
    //      Konstruktor
    public Artikel ( int nummer, String bezeichnung )
    {
        this.nummer = nummer;
        this.bezeichnung = bezeichnung;
    }
    // weitere Methoden
    String getBezeichnung ( )
    {
        return this.bezeichnung;
    }
    public String toString ( )
    {
        return "\nArtikelnummer : " + this.nummer + " Bezeichnung : " + this.bezeichnung ;
    }
} // end class Artikel
```

```
public class Box < T >
```

```
{
    //      Attribute
    private T value;
    private String verwendung;
    //      Konstruktor
    public Box ( T value, String verwendung )
    {
        this.value = value;
        this.verwendung = verwendung;
    }
    //      weitere Methoden
    public void setValue ( T value )
    {
        this.value = value;
    }
    public T getValue ( )
    {
        return this.value;
    }
    public String getVerwendung ( )
    {
        return this.verwendung;
    }
    public String toString ( )
    {
        return this.value + " wird verwendet in : " + this.verwendung;
    }
} // end class Box
```

```

public class Anwendung
{
    public static void main(String[] args)
    {
        Artikel artikel = new Artikel ( 1, "Armband" );
        Box < Artikel > artikelBox  = new Box < Artikel > ( artikel, "Katalog" );
        Box < String > stringBox  = new Box < String > ( "Hallo" , "Text" );
        Box < Integer > integerBox = new Box < Integer > ( 123, "Rechenoperation" );

        System.out.println ( artikelBox + "\n" + stringBox + "\n" + integerBox);
    }
} // end Anwendung

```

Würde die gleiche Anwendung nicht mit der typisierten Klasse Box < T > realisiert, bräuhete man so viele Klasse Box (im Beispiel drei), wie Datentypen für T eingesetzt werden, im Beispiel:

- class ArtikelBox
- class StringBox
- class IntegerBox.

// Übung: enum und typsichere (generische) Klasse

Eine Gärtnerei beabsichtigt, ihre Pflanzen in einem Katalog abzubilden und anzubieten.

1. Für jede Pflanze soll festgehalten werden:

- a) die Bezeichnung (String)
- b) die Blütezeit
Als Blütezeit sollen nur FRÜHLING; SOMMER; HERBST und WINTER angegeben werden.

DV- Technik::

Die Blütezeit soll durch einen enum- Typ repräsentiert werden. Der enum- Typ muß zusätzlich besitzen

- einen privaten Konstruktor, der als Argument die jeweilige Blütezeit erhält.
- eine String toString () - Methode, die die entsprechende Blütezeit zurückgibt.

- c) eine Beschreibung
Jede Pflanze soll beschrieben werden
 - durch die Pflanzenart.
(Beispiel: Die Kornblume gehört zur Pflanzenart Korbblütler)
 - und durch eine Pflegeanleitung (Beispiel: wenig gießen)

DV- Technik::

- Die tatsächliche Klasse Beschreibung soll erst bei der Verwendung, d.h. beim Erstellen eines Pflanzenobjektes festgelegt werden. Der Datentyp des Attributes beschreibung soll also eine Typvariable, z.B. T sein.

2. Die Klasse Beschreibung hat zwei Attribute vom Typ String
- pflanzenart
 - pflegeanleitung

Implementieren Sie für die Klasse Beschreibung einen Konstruktor und eine String toString () – Methode, die die Pflanzenart und die Pflegeanleitung zurückgibt..

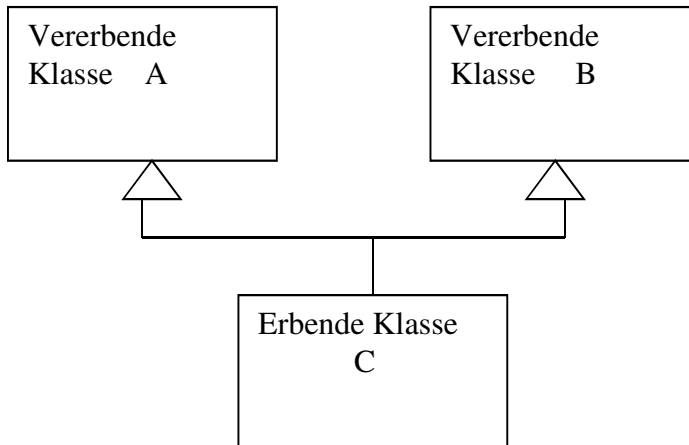
3. Schreiben Sie ein Testprogramm, das je zwei Pflanzenobjekte erzeugt. Verwenden Sie für **zwei** Pflanzenobjekte
- den Datentyp String für das Attribut beschreibung
 - und für die **zwei anderen** Pflanzenobjekte den Datentyp Beschreibung (class Beschreibung) für das Attribut beschreibung.

Legen Sie im main die zwei Pflanzenobjekte mit den entsprechenden Datentypen in ein typsicheres Array. und geben Sie die Attribute der beiden Array' s mit der for- each- Schleife aus.

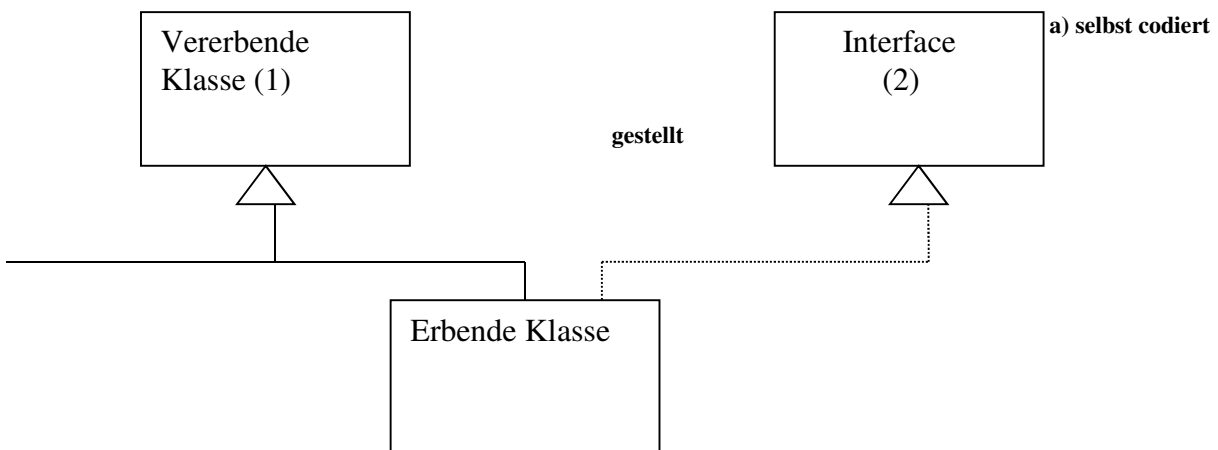
5. Interface

5.1. Vererbung - Interface

Mehrfachvererbung - in Java nicht möglich

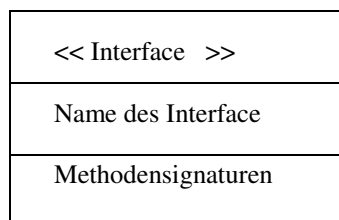


Lösung durch Interfaces



- (1) - Die erbende Klasse übernimmt alle Attribute
- Methoden können in den Kindern
 - übernommen
 - redefiniert
 - neu hinzugefügt
 - implementiertwerden
- (2) - Ein Interface hat höchstens Konstante
- In den Klassen, die ein Interface „implementieren“, müssen alle Methodensignaturen des Interface implementiert werden.

Notation



Ein Interface definiert die volle Funktionsfähigkeit, ohne eine Implementierung festzulegen. Die einzelnen Klassen stellen unterschiedliche Implementierungen des Interface dar.

5.2. Selbstcodiertes Interface

Beispiel: Interface Groesse (selbst codiert)

```
public interface Groesse
{
    // ermittelt die Laenge
    public int getLaenge ( );
    // ermittelt die Hoehe
    public int getHoehe ( );
    // ermittelt die Breite
    public int getBreite ( );
}
```

Die gewünschte Funktionalität der Schnittstelle wird einer Klasse noch nicht durch Definition zur Verfügung gestellt, sondern lediglich beschrieben. Soll die Funktionalität von einer Klasse tatsächlich realisiert werden, muß sie das Interface implementieren. Dazu erweitert sie die class- Anweisung um eine implements- Klausel. Der Compiler sorgt dafür, daß alle im Interface geforderten Methoden definitionsgemäß implementiert werden. Zusätzlich verleiht er der Klasse einen neuen Datentyp, der ähnliche Eigenschaften wie eine Klasse hat.

```
public class Auto                implements Groesse
{
    private String marke;
    private int leistung;
    private int laenge;
    private int hoehe;
    private int breite;
    public Auto ( String marke, int leistung, int laenge, int hoehe, int breite )
    {
        this.marke = marke;
        this.leistung = leistung;
        this.laenge = laenge;
        this.hoehe = hoehe;
        this.breite = breite;
    }
    public int getLaenge ( )
    {
        return this.laenge;
    }
    public int getHoehe ( )
    {
        return this.hoehe;
    }
    public int getBreite ( )
    {
        return this.breite;
    }
    public String getMarke ( )
    {
        return this.marke;
    }
    public String toString ( )
    {
        return " Marke " + this.marke + " Länge " + this.laenge +
            " Hoehe " + this.hoehe + " Breite " + this.breite;
    }
} // end class Auto

public class Sportplatz        implements Groesse
{
    public Sportplatz ( ) { }
    public int getLaenge ( )
    {
        return 10500;
    }
    public int getHoehe ( )
    {
        return 0;
    }
}
```

```

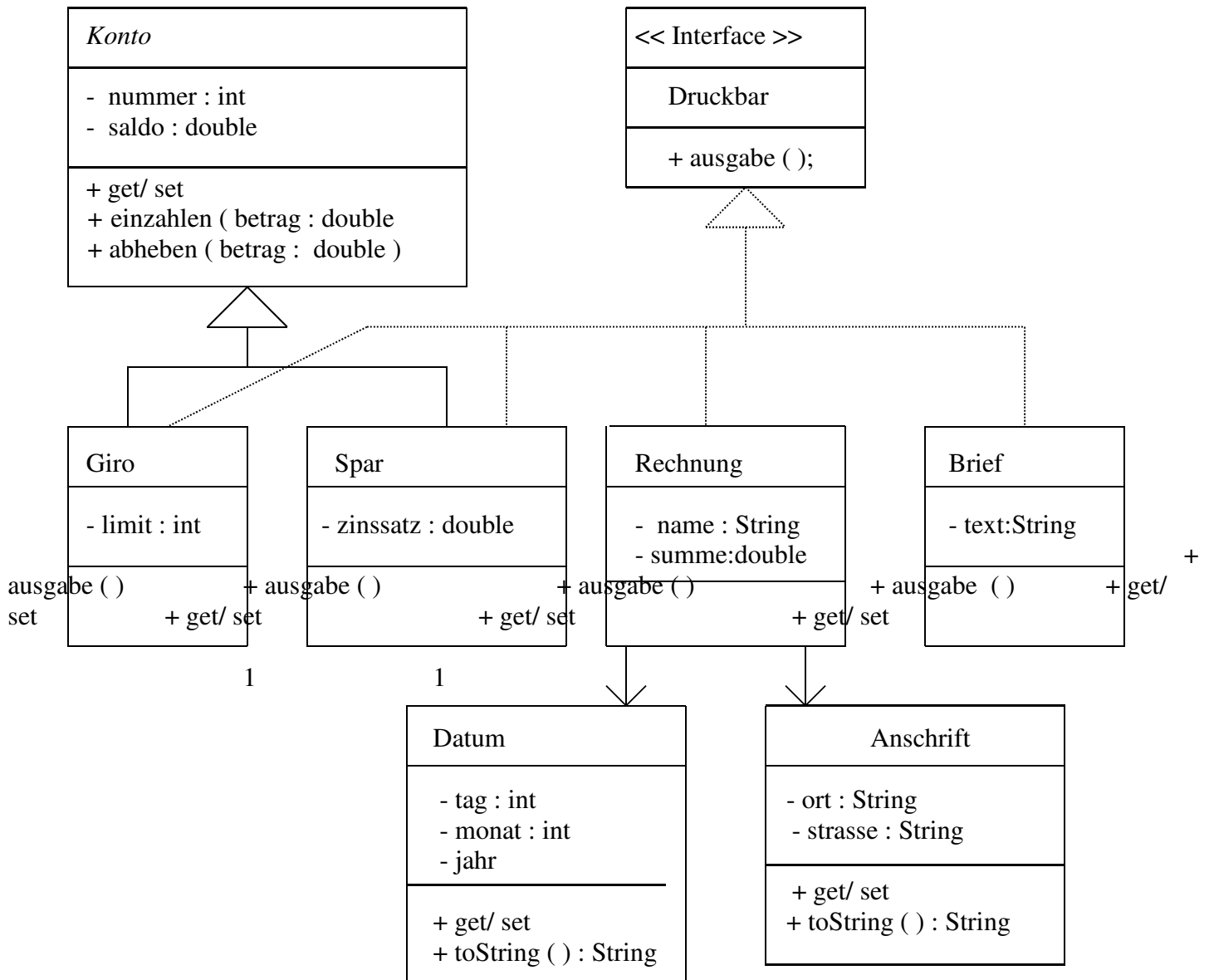
    public int getBreite ( )
    {
        return 70000;
    }
    public String toString ( )
    {
        return " Sportplatz: " + " Länge : " + getLaenge ( ) +
            " Breite : " + getBreite ( );
    }
} // end class Sportplatz
public class Papierformat           implements Groesse
{
    private int format;                // 0 = DIN A0 usw.
    public Papierformat ( int format )
    {
        this.format = format;
    }
    public int getLaenge ( )
    {
        int ret = 0;
        switch ( format)
        {
            case 0: ret = 1189; break;
            case 1: ret = 841; break;
            // usw:
            default : System.out.println ( "falsches Format" );
        }
        return ret;
    }
    // für breite entsprechend
    public int getBreite ( )
    {
        int ret = 0;
        switch ( format)
        {
            case 0: ret =    ...    ; break;
            case 1: ret =    ...    ; break;
            // usw:
            default : System.out.println ( "falsches Format" );
        }
        return ret;
    }
    public int getHoehe ( )
    {
        return 0;
    }
    public String toString ( )
    {
        return " Papierformat : " + this.format + " Länge " +
            getLaenge ( ) + " Breite : " + getBreite ( ) ;
    }
} // end class Papierformat
public class Anwendung
{
    public static void main ( String [ ] args )
    {
        Auto a          = new Auto ( "BMW", 300, 6, 2, 5 );
        Sportplatz s     = new Sportplatz ( );
        Papierformat p   = new Papierformat ( 0 );
        System.out.println ( "Auto : " + a );
        System.out.println ( "Sportplatz : " + s );
        System.out.println ( "Papierformat : " + p );
    } // end main} // end class Anwendung

```

Ein Interface kann viele Klassen um einen neuen Datentyp erweitern. Nützlich ist ein Interface immer dann, wenn Eigenschaften einer Klasse beschrieben werden sollen, die nicht direkt in seiner normalen Vererbungshierarchie abgebildet werden können. Hätte man Groesse als abstrakte Klasse definiert, ergäbe sich eine sehr ungewöhnliche Ableitung, wenn Autos, Sportplätze und Papierformate aus ihr abgeleitet würden. Durch Implementierung des Groesse- Interfaces können diese Klassen die Verfügbarkeit der drei Methoden getLaenge (), getBreite (), getHoehe () unabhängig von ihrer eigenen Vererbungslinie garantieren.

Würden die Methoden getLaenge (), getBreite (), getHoehe () in den Klassen Auto, Sportplatz und Papierformat direkt implementiert, gingen die Vorteile der Polymorphie verloren. Das Interface wirkt so etwa wie eine abstrakte Elternklasse für Auto, Sportplatz und Papierformat.

Übung Interface Druckbar



1. Codieren Sie
 - das Interface Druckbar
 - die Klasse Konto
 - die Klasse Giro
2. Schreiben Sie eine main- Funktion, in dem Sie eine ArrayList anlegen, in die ArrayList ein Objekt von jeder konkreten Klasse aus dem UML einfügen und in einer for- each- Schleife die Attribute der einzelnen Objekte ausgeben. Die Klassen Spar, Datum, Anschrift, Rechnung, Brief

5.3. Übung selbstcodiertes generisches Interface für eine Collection

```
public interface List < E >
{
    /*
     * fügt ein Element an der 1. Stelle ( = Position 0 ) der Liste ein,
     * überschreibt nicht.
     * Ist die Liste voll, Fehlermeldung: „Liste voll“
     */
    public void insertFirst ( E element ) ;

    /*
     * Hängt ein Element an das Ende der Liste an.
     * Ist die Liste voll, Fehlermeldung: „Liste voll“
     */
    public void insertLast ( E element ) ;

    /*
     * Entfernt das zuerst in die Liste eingefügte Element. Das ist das
     * Element an Position 0. Ist die Liste leer, kann kein Element entfernt werden.
     * Fehlermeldung
     */
    public void leaveFirst ( ) ;

    /*
     * Entfernt das letzte Element der Liste.
     * Ist die Liste leer, kann kein Element entfernt werden.
     * Fehlermeldung
     */
    public void leaveLast ( ) ;

    /*
     * Prüft, ob die Liste leer ist,
     */
    public boolean isEmpty ( ) ;

    /*
     * fügt an der Stelle index das übergebene Element ein. Die Elemente ab der Position
     * index werden nach rechts verschoben
     */
    public void insertAt ( int index, E element ) ;
}
```

Das oben angegebene Interface List ist gegeben.
Programmieren Sie eine Klasse Liste, deren Implementierung auf einem Array basiert.
Codieren Sie zusätzlich die Methode **public String toString ()**

Die Klasse Liste hat

- einen Standardkonstruktor, der ein Array mit 4 Elementen anlegt
- einen allgemeinen Konstruktor mit dem Argument : anfangskapazität.

6. Exceptions (= Ausnahmen)

Die Exceptionklassen ermöglichen eine strukturierte Behandlung von Fehlern, die während der Laufzeit auftreten.

Eine Exception ist ein Fehler, der durch ein Programm zur Laufzeit verursacht werden kann. DV- technisch sind Exceptions Objekte von Fehlerklassen. Solche Fehlerklassen sind für verschiedene Typen von Fehlern von Java erstellt oder können vom Anwender selbst erstellt werden. Bei fehlerhaftem Zugriff erzeugt eine solche Klasse ein Objekt und wirft es als Exception aus.

Die Klasse `IndexOutOfBoundsException` z.B ist von Java erstellt für fehlerhaften Zugriff auf Arrays.

Prinzip des Exception- Mechanismus

Laufzeitfehler oder eine vom Entwickler gewollte Bedingung lösen eine Exception aus.

a) Keine Fehlerbehandlung

Wird der Fehler von **niemandem behandelt**, führt er zum Abbruch des Programmes.
Eine Fehlermeldung wird ausgegeben.

Beispiel:

```
public class HP
{
    public static void main(String[] args)
    {
        double erg = 0.0;
        /*
            Wird die Ausnahme von keinem Programmteil behandelt,
            führt das zum Abbruch des Programmes und zu einer Fehlermeldung
        */
        erg = 20 / 0;
        System.out.println ( "Divisionsergebnis: " + erg );
    }
}
```

Ausgabe:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at HP.main(HP.java:13)
```

b) Behandlung des Fehlers im Programmteil, in dem er ausgelöst wurde

Soll der Fehler vom Programmteil, in dem er ausgelöst wurde, **behandelt** werden, muß ein try- und catch- Block angelegt werden..

Die fehleranfällige Codesequenz wird in einem try- Block abgelegt. Tritt der Fehler auf, verzweigt der Compiler in den direkt an den try- Block anschließenden catch- Block, in dem die Fehlerbehandlung erfolgt.

try- catch- Anweisung

```
try
{
    // fehleranfällige Anweisung
}
catch ( Typ der möglichen Exception    Name desFehlerobjektes )
{
    // Fehlerbehandlung
}
```

In der catch- Anweisung wird

- Der Typ des abzufangenden Fehlers definiert, z.B. `ArithmeticException`.

ein formaler Parameter angegeben (z.B. `ArithmeticException a`), der beim Auftreten des Fehlers ein Fehlerobjekt übernehmen soll. Dieses Fehlerobjekt wurde von der entsprechenden Fehlerklasse (`Throwable` oder eine ihrer Unterklassen), hier z.B. `ArithmeticException`, erzeugt und ausgeworfen.

Das Fehlerobjekt enthält Informationen über die Art des aufgetretenen Fehlers.

- Mit `toString ()` kann man den Fehlertext ausgeben.
- `getMessage ()` liefert auch den Fehlertext
- `printStackTrace ()` gibt einen Auszug aus dem Laufzeitstack

Man kann auch eine selbstgewählte Fehlermeldung ausgeben.

Beispiel:

public class Anwendung

```
{
    public static void main(String[] args)
    {
        double erg = 0.0;
        /*
           Wird die fehlerabhängige Codesequenz
           in einem try- und catch- Block behandelt .
           kann eine geeignete Fehlermeldung ausgegeben werden
        */
        try
        {
            erg = 20 / 0;
        }
        catch ( ArithmeticException a )
        {
            //      selbstgewählte Fehlermeldung
            System.out.println ( erg );
            System.out.println ( "Division durch 0 ist nicht moeglich" );

            //      Hier wird die toString- Methode von Throwable aufgerufen
            System.out.println ( "toString ( )- Methode von Throwable: " + a );

            //      gibt den Fehlertext des Fehlerobjektes aus
            System.out.println ( "Fehlertext: " + a.getMessage ( ) );

            //      gibt einen Auszug aus dem LaufzeitStack aus,
            System.out.println ( "vom Laufzeit- Stack: " );
            a.printStackTrace ( );
        }
        System.out.println ( "Divisionsergebnis: " + erg );
    } // end main
} // end class anwendung
```

Ausgabe:

```
0.0
Division durch 0 ist nicht moeglich

toString ( )- Methode von Throwable:  java.lang.ArithmeticException: / by zero

Fehlertext:  / by zero

vom Laufzeit- Stack:
java.lang.ArithmeticException: / by zero
    at HP.main(HP.java:23)
ivisionsergebnis:  0.0
```

Mehrere Fehlerarten in einer Codesequenz

In einem try- block können Codesequenzen abgelegt werden, die jeweils unterschiedliche Fehler zur Folge haben können. Für jede Fehlerart wird dann im Anschluß an den try- Block eine Fehlerbehandlung in einem unterschiedlichen catch- Block angeboten.

Beispiel:

Fordern Sie zwei ganze Zahlen über die Konsole an, teilen Sie diese, und geben Sie das Divisionsergebnis aus.

```
import java.io.*;
public class HP
{
    public static void main(String[] args)
    {
        BufferedReader br = new BufferedReader ( new InputStreamReader ( System.in ) );

        // Bereiche
        int dividend = 0;
        int divisor = 0;
        int ergebnis = 0;
        // Befehle

        System.out.println ( "Bitte Dividend eingeben - ganze Zahl " );
        System.out.println ( "Bitte Divisor eingeben - ganze Zahl " );
        try
        {
            dividend = Integer.parseInt ( br.readLine ( ) );
            divisor = Integer.parseInt ( br.readLine ( ) );
            ergebnis = dividend / divisor ;
        }
        catch ( IOException i )
        {
            System.out.println ( "Fehler beim Lesen " );
        }
        catch ( ArithmeticException a )
        {
            System.out.println ( "Fehler: Teilen durch 0 " );
        }
        System.out.println ( "Divisionsergebnis : " + dividend + " : " + divisor + " = " + ergebnis );
    }
}
```

Ausgabe;

```
Bitte Dividend eingeben - ganze Zahl
Bitte Divisor eingeben - ganze Zahl
20
0
Fehler: Teilen durch 0
Divisionsergebnis : 20 : 0 = 0
```

c) Weitergabe des Fehlers

Wird der Fehler weitergegeben, hat der Empfänger wieder die Möglichkeit, ihn zu behandeln oder ihn weiterzugeben.

Alle Fehler, die nicht innerhalb einer Methode behandelt werden, werden dem Compiler mit Hilfe der throws- Klausel bekannt gemacht. Er kennt also zu jeder Methode die potentiellen Fehler, die in ihr vorkommen können. Innerhalb der Methode werden die Bedingungen angegeben, die einen Fehler hervorrufen können.

Wenn eine solche Bedingung zutrifft, wird vom Programmierer mit **throw** und **new** ein Fehlerobjekt der entsprechenden Klasse erzeugt .

Dort, wo die fehleranfällige Methode aufgerufen wird, muß sie in einen try- Block gesetzt werden, und anschließend wird im catch- Block der Fehler behandelt.

Man leitet eine nach den eigenen Erfordernissen konzipierte Fehlerklasse von Exception ab.

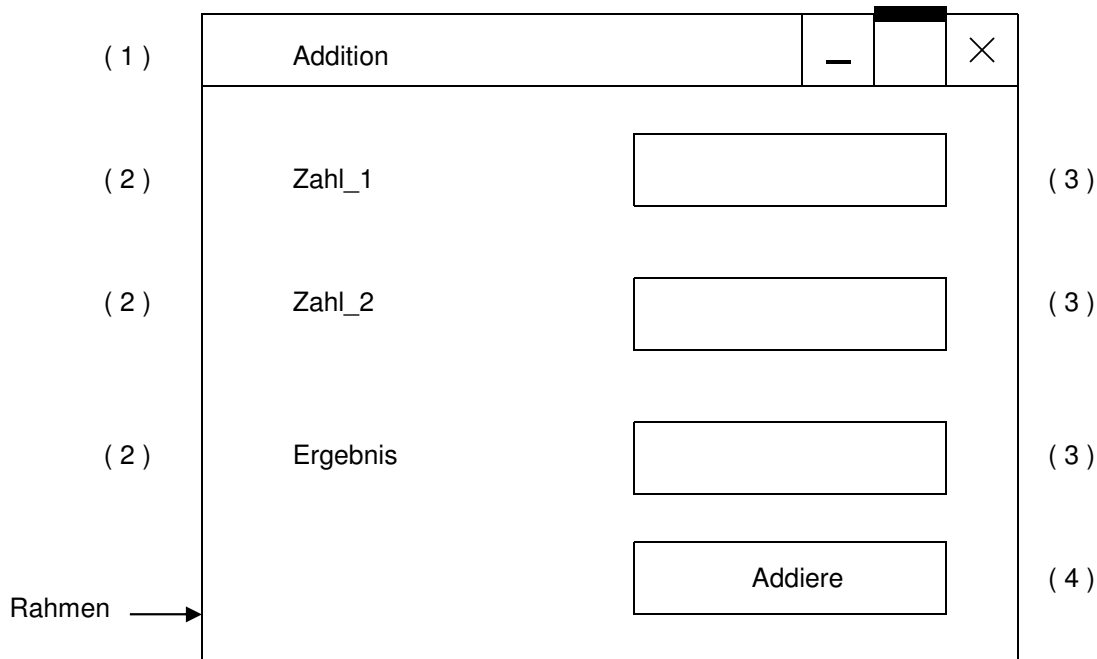
Man macht bei einer selbstcodierten Fehlerklasse den Compiler bei der Methode, in der der Fehler auftreten kann, darauf aufmerksam (**throws selbstcodierte_Fehlerklasse**), daß in der Methode ein Fehler auftreten kann.

Innerhalb der Methode listet man die Bedingungen auf, bei deren Eintreten ein Fehler entstehen kann, und man erstellt hinter jeder Bedingung mit **throw** und **new** ein Fehlerobjekt der selbstcodierten Fehlerklasse.

```
class MyException      extends Exception
{
    MyException ( ) { super( );}
    MyException ( String s )
    {
        super(s);
    }
} // end class MyException

    public void schiffPositionieren
        ( int xanfang, int yanfang, int xende, int yende ) throws MyException
    {
        char c;
        // Bedingung xe >= xa und ye >= ya
        if ( xanfang > xende || yanfang > yende )
            throw new MyException ( "verkehrt herum" );
    } // end schiffPositionieren
```

7.1. Fenster aufbauen



Die Fensterklasse wird von der Klasse JFrame abgeleitet und erbt

- einen Rahmen
 - eine Titelzeile ohne Titel (1)
- Der Titel kann mit „void setTitle (String titel);“ gesetzt werden.
Sie enthält folgende Elemente:



Mausklick drauf schließt das Fenster



Mausklick drauf vergrößert/ stellt den Ursprungszustand wieder her



Mausklick drauf macht das Fenster zum Icon

Eine Anwendung braucht verschiedene Elemente, z.B.

- (2) Label (= ,Beschriftungstext)
- (3) Ein- Ausgabefelder
- (4) Knöpfe (Button's, Das Drücken eines Button stößt eine Funktion im Programm an, die dann ausgeführt wird.)

Diese Elemente sind Objekte, die im Fensterobjekt angeordnet werden

Ein Layoutmanager ist für die Anordnung der Elemente verantwortlich. Er wird dem Fenster, für das er zuständig ist, zugeordnet. Für die Anordnung gibt es auch verschiedene Layout's.

Im Layout werden z.B. festgelegt:

- die Anzahl der Zeilen und Spalten
- der Text der Titelzeile

Alle Elemente werden dem Fenster zugeordnet.

7.2. DV- technische Realisierung des Fensteraufbaues

```
public class View                extends JFrame
{
    //      1.      Layoutmanager anlegen
    private GridLayout gridLayout = new GridLayout ( );
    //      2.      Elemente anlegen
    private JLabel lbl_zahl_1      = new JLabel ( "Zahl_1");
    private JTextField txt_zahl_1  = new JTextField ( );
    private JLabel lbl_zahl_2      = new JLabel ( "Zahl_2");
    private JTextField txt_zahl_2  = new JTextField ( );
    private JLabel lbl_ergebnis   = new JLabel ( "Ergebnis");
    private JTextField txt_ergebnis = new JTextField ( );
    private JButton btn_addieren   = new JButton ( "Addiere");
    //      3.      Konstruktor, er führt den Fensteraufbau aus
    public View ( )
    {
        viewAufbau ( );
    } // end Konstruktor
    /*      4.      In der Methode viewAufbau ( ) das Layout festlegen ( z.B. Zeilen und Spalten )
                und den Layoutmanager dem Fenster zuordnen.
                Alle Elemente dem Fenster zuordnen.
    */
    private void viewAufbau ( )
    {
        gridLayout.setRows ( 4 );
        gridLayout.setColumns ( 2 );
        this.setLayout ( gridLayout );
        this.setTitle ( "Addition");

        add ( lbl_zahl_1 );
        add ( txt_zahl_1 );
        add ( lbl_zahl_2 );
        add ( txt_zahl_2 );
        add ( lbl_ergebnis );
        add ( txt_ergebnis );
        add ( new JLabel ( ) );
        add ( btn_addieren );

        // Hier kommen noch weitere Informationen, die aber nichts mit dem Fensteraufbau
        // zu tun haben : z. B. WL beim Fenster und AL beim Button registrieren
    } // end viewAufbau
}
```

7.3. Delegation- Event- Modell

(Funktionalität eines Fensters realisieren)

Am Modell Beteiligte:

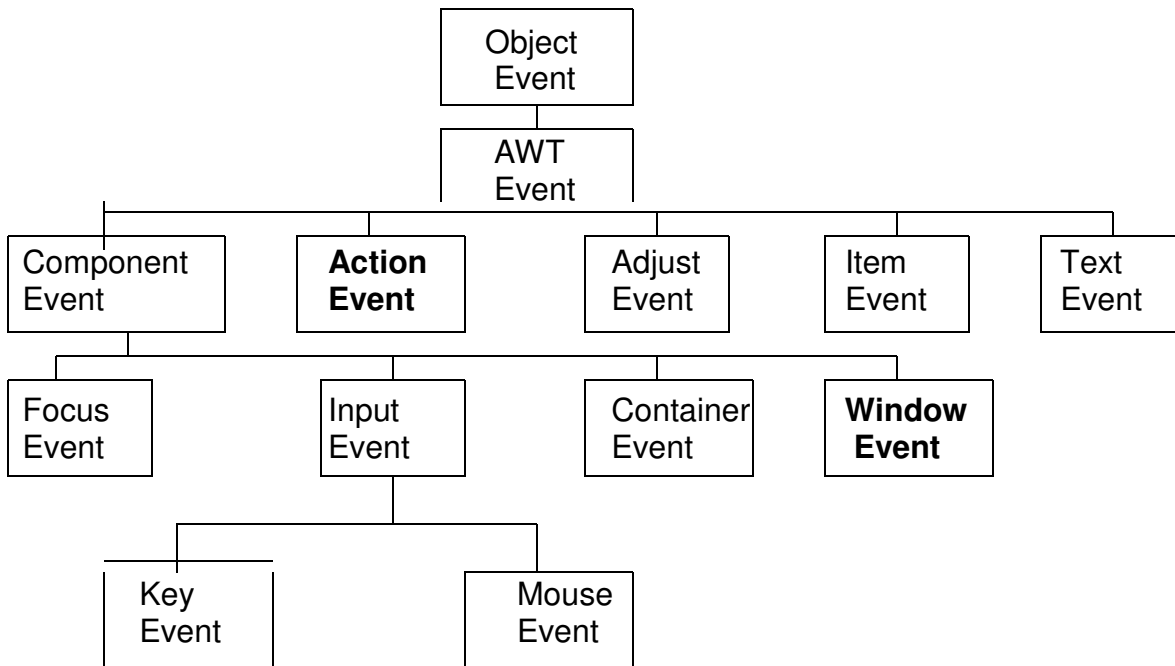
1. Ereignisquellen,
z.B. Button (= Auslöser einer Nachricht, = Auslöser eines Ereignisses)
2. Ereignisempfänger (= EventListener)

EventListener hören die Ereignisquellen ab und reagieren auf deren Nachrichten.

Damit ein EventListener die Nachricht einer bestimmten Ereignisquelle empfangen

kann, muß er sich bei dieser registrieren. Von den Ereignisquellen gehen z.B. durch Mausklick oder Tastendruck vielerlei Ereignisarten (Nachrichten) an die passenden EventListener. Für jede Ereignisart gibt es einen speziellen EventListener.

Ereignisarten



Beispiel_1

Das Element ☒ des Fensters wird gedrückt. Es wird ein WindowEvent (Ereignisart) erzeugt. Das Ereignis (die Nachricht: „Ich will geschlossen werden“) wird durch den Mausklick ausgelöst und vom zugehörigen EventListener, dem WindowListener empfangen, interpretiert und ausgeführt, aber nur, wenn er sich beim Fenster registriert hat. **Das Fenster ist Ereignisquelle und EventListener zugleich.**

Beispiel_2

Der Button wird gedrückt und erzeugt ein ActionEvent. Die Nachricht, daß addiert werden soll, wird durch das Drücken des Button vom EventListener, dem ActionListener empfangen, interpretiert und ausgeführt, aber nur, wenn er sich bei dem Button registriert hat. **Der Button „Addiere“ ist die Ereignisquelle, das Fenster ist der ActionListener.**

7.4. DV- technische Realisierung der Funktionalität (Delegation- Event- Modell)

Das Modell ermöglicht eine Vielzahl von Implementierungen.

7.4.1. Registrieren des WindowListener und des ActionListener

Das Registrieren erfolgt mit speziellen Methoden, denen ein Objekt übergeben wird, das das jeweilige EventListener- Interface implementiert. Beide Registrieremethoden werden in der Methode „viewAufbau ()“ codiert.

Beispiel_1 und Beispiel_2:

Die EventListener registrieren sich bei den Ereignisquellen.

```

public class View extends JFrame
{
    ...
    private void viewAufbau ( )
    { ...
        this.addWindowListener ( this );           // Methode der Klasse JFrame
        ↑
        // Ereignisquelle: Fenster           EventListener: Fenster

        this.btn_addieren.addActionListener ( this ); // Methode der Klasse Button
        ↑
        // Ereignisquelle: Button: addieren   EventListener : Fenster
    }
    // end viewAufbau
  
```

7.4.2. Implementieren der benötigten Interfaces

```
public class View                extends JFrame                implements WindowListener, ActionListener
{
    ...
```

Für ein WindowEvent gibt es sieben verschiedene Möglichkeiten. Das WindowListener-Interface bietet sieben Methodenrumpfe an, die in der Fensterklasse zu implementieren sind.

// In der Klasse View nach der Methode „viewAufbau ()“ zu codieren.

// Implementierte Methoden des Interface WindowListener

```
public void windowOpened ( WindowEvent e ) { }
public void windowClosing ( WindowEvent e )      // Fenster wird geschlossen
{
    this.setVisible ( false );
    dispose ( );
    System.exit ( 0 );
}
public void windowClosed ( WindowEvent e ) { }    // Fenster ist geschlossen
public void windowIconified ( WindowEvent e ) { }
public void windowDeiconified ( WindowEvent e ) { }
public void windowActivated ( WindowEvent e ) { }
public void windowDeactivated ( WindowEvent e ) { }
```

// Für ein ActionEvent bietet das ActionListener-Interface nur einen Methodenrumpf an,
// der in der Fensterklasse zu implementieren ist.

// Implementierte Methode des Interface ActionListener

```
public void actionPerformed ( ActionEvent e )
{
    addieren ( );
} // end actionPerformed

public void addieren ( )
{
    String zahl = this.txt_zahl_1.getText ( );
    int zahl_1 = Integer.parseInt( zahl );
    zahl = this.txt_zahl_2.getText ( );
    int zahl_2 = Integer.parseInt( zahl );
    int ergebnis = zahl_1 + zahl_2;
    txt_ergebnis.setText ( String.valueOf ( ergebnis ) );
} // end addieren
```

In der Regel werden bei einem ActionEvent Daten aus dem Fenster zur Verarbeitung gebraucht und Ergebnisse der Verarbeitung ans Fenster zurückgeliefert. Dazu gibt es Methoden, die Daten holen und die Daten zurückliefern.

```
public String getText ( );
public void setText ( String text );
```

Alle Daten werden im Fenster im Stringformat abgelegt. Das bedingt beim Lesen (vom Fenster ins Programm) z.B. für Rechenvorgänge eine Transformation:
public static Integer parseInt (String string)

Sollen z.B. Rechenergebnisse zurück ins Fenster transportiert werden, ist wiederum das Stringformat herzustellen:

z.B. public static String Integer.toString (Integer integer)

7.5. Ausführen eines GUI- Programmes

```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

public class Anwendung

```
{
    public static void main(String[] args)
    {
        // Fensterobjekt anlegen
        View v = new View ( );
        // Linke obere Ecke des Fensters an die Bildschirmposition x und y bewegen
        v.setLocation ( 500, 300 );
        // Größe des Fensters auf den zur Darstellung aller Komponenten anpassen
        v.pack ( );
        // Fenster sichtbar machen
        v.setVisible ( true );
    } // end main
} // end classAnwendung
```

7.6. Fenster mit mehreren Button' s

Rechnen		—	×
Zahl_1	<input type="text"/>		
Zahl_2	<input type="text"/>		
Ergebnis	<input type="text"/>		
<input type="button" value="Subtrahiere"/>	<input type="button" value="Addiere"/>		

```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

public class View extends JFrame implements WindowListener, ActionListener

```
{
    private GridLayout gridLayout = new GridLayout ( );

    private JLabel lbl_zahl_1      = new JLabel ( "Zahl_1" );
    private JTextField txt_zahl_1  = new JTextField ( );
    private JLabel lbl_zahl_2      = new JLabel ( "Zahl_2" );
    private JTextField txt_zahl_2  = new JTextField ( );
    private JLabel lbl_ergebnis   = new JLabel ( );
    private JTextField txt_ergebnis = new JTextField ( );

    private JButton btn_add = new JButton ( "Addiere" );
    private JButton btn_sub = new JButton ( "Subtrahiere" );

    public View ( )
    {
        viewAufbau ( );
    }
}
```

```

private void viewAufbau ( )
gridLayout setRows ( 4 );
gridLayout.setColumns ( 2 );
this.setLayout ( gridLayout );
this.setTitle ( "Rechner" );
add ( txt_zahl_1 );
add ( lbl_zahl_2 );
add ( txt_zahl_2 );
add ( lbl_ergebnis );
add ( txt_ergebnis );
add ( btn_sub );
add ( btn_add );

// ActionListener bei den Buttons registrieren
this.btn_add.addActionListener ( this );
this.btn_sub.addActionListener ( this );

// WindowListener registrieren
this.addWindowListener ( this );
} // end viewAufbau
// Implementierte Methoden des Interface WindowListener
public void windowOpened ( WindowEvent e ) { }
public void windowClosing ( WindowEvent e )
{
    this.setVisible ( false );
    dispose ( );
    System.exit ( 0 );
}
public void windowClosed ( WindowEvent e ) { }
public void windowIconified ( WindowEvent e ) { }
public void windowDeiconified ( WindowEvent e ) { }
public void windowActivated ( WindowEvent e ) { }
public void windowDeactivated ( WindowEvent e ) { }

// Implementierte Methode des Interface ActionListener
public void actionPerformed ( ActionEvent e )
{
    /*
    Die Klasse ActionEvent stellt folgende Methoden zur Verfügung:
    - getId ( )
    - getSource ( )
    - getActionCommand ( ) ( wenn mehrere Button' s gedrückt werden können. )
    */
    String string = e.getActionCommand ( );
    if ( string.equals( "Addiere" ) )
    {
        addieren ( );
    }
    if ( string.equals( "Subtrahiere" ) )
    {
        subtrahieren ( );
    }
} // end actionPerformed
public void addieren ( )
{
    String zahl = this.txt_zahl_1.getText ( );
    int zahl_1 = Integer.parseInt( zahl );
    zahl = this.txt_zahl_2.getText ( );
    int zahl_2 = Integer.parseInt( zahl );
    int ergebnis = zahl_1 + zahl_2;
    txt_ergebnis.setText ( String.valueOf ( ergebnis ) );
} // end addieren

```

```

        public void subtrahieren ( )
        {
            String zahl = this.txt_zahl_1.getText ( );
            int zahl_1 = Integer.parseInt( zahl );
            zahl = this.txt_zahl_2.getText ( );
            int zahl_2 = Integer.parseInt( zahl );
            int ergebnis = zahl_1 - zahl_2;
            txt_ergebnis.setText ( String.valueOf ( ergebnis ) );
        } // end subtrahieren
    } // end class View

```

```

public class Anwendung
{
    public static void main(String[] args)
    {
        View v = new View ( );
        v.setLocation ( 500, 300 );
        v.pack ( );
        v.setVisible ( true );
    } // end main
} // end class Anwendung

```

7.7. Innere Klassen

Implementierung mit anonymen Klassen

Eine anonyme Klasse hat keinen eigenen Namen. Definition und Initialisierung erfolgen in einer kombinierten Anweisung. Eine anonyme Klasse ist eine Einwegklasse, die nur einmal instanziiert werden kann.

Ihre wichtigste Anwendung finden die anonymen Klassen bei der Definition von Listnern.

EventListener

Damit ein Objekt Nachrichten empfangen kann, muß es eine Reihe von Methoden implementieren, die von der Nachrichtenquelle, bei der es sich angemeldet hat, aufgerufen werden können. Um sicherzustellen, daß diese Methoden vorhanden sind, müssen die EventListener bestimmte Interfaces implementieren.

ActionListener

Ein ActionListener bekommt ein semantisches Ereignis als Argument (= ein Objekt vom zugehörigen Ereignistyp). Alle Listener, die semantische Ereignisse bekommen, haben nur eine Methode, die immer vom Anwender individuell implementiert werden muß.

Beim Registrieren wird eine interne anonyme Klasse erzeugt:

```

button.addActionListener
(
    new ActionListener ( )
    {
        public void actionPerformed ( ActionEvent event )
        {
            addieren( );
        }
    }
);

```

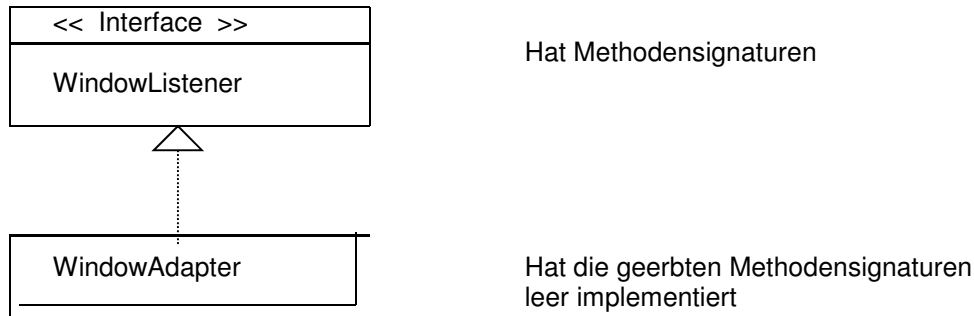

WindowListener

Ein WindowListener bekommt Low- Level- Ereignisse. Sie sind für den Transfer von elementaren Nachrichten da. Ein WindowListener (und alle Listener, die Low-Level- Ereignisse bekommen) haben mehrere Methoden.

Wenn nun eine Anwendung nicht alle Methoden implementieren will, ist es günstig, eine Adapterklasse einzufügen.

Eine Adapterklasse implementiert ein gegebenes Interface mit leeren Methodenrumpfen. Die Anwendung braucht dann nur noch die leer implementierten Methoden zu überschreiben, die sie wirklich braucht.

Zu jedem Low- Level- EventListener werden Adapterklassen zur Verfügung gestellt.



// WindowListener registrieren

```
this.addWindowListener
(
    new WindowAdapter ( )
    {
        public void WindowClosing ( WindowEvent event )
        {
            setVisible ( false );
            dispose ( );
        }
    }
);
```

Übung Speisekarte

1. Codieren Sie die Elemente einer erweiterten enum- Klasse, und stellen Sie für die einzelnen Wochentage je ein Gericht zur Verfügung.
2. Codieren Sie folgende Methoden:
public String welcheSpeise (Speisekarte speisekarte)
(gibt z.B. zurück: Am Montag gibt es Apfelstrudel, usw)
public String to String ()
public String gibGericht ()
gibt das Gericht am entsprechenden Wochentag zurück;
3. Erstellen Sie eine Klasse View

Speisekarte	
Wochentag	Montag ↓
Speise	
	Welche Speise

→ ComboBox

→ TextField

→ Button

Mit Hilfe der ComboBox wird der Wochentag eingestellt.
Nach Drücken des Button „Welche Speise“ wird im Textfeld angezeigt, welche Speise am ausgewählten Wochentag gegessen werden kann.

8..Entwurfsmuster

8.1.Singleton

Beschreibung einer Singletonklasse

Ein Singleton ist eine Klasse, von der nur ein einziges Objekt erstellt werden darf und kann,
z. B

- eine GUI für mehrere, die auf die gleiche GUI zugreifen wollen oder
- eine Datenbank für mehrere Benutzer

Die Klasse ist für die Verwaltung ihres einzigen Exemplares zuständig.

1. Sie fängt die Befehle zum Erzeugen neuer Objekte ab und stellt so sicher, daß nur ein einziges Objekt erstellt wird.
2. Sie bietet Zugriffsmöglichkeiten auf das Objekt.

Wichtige Designmerkmale:

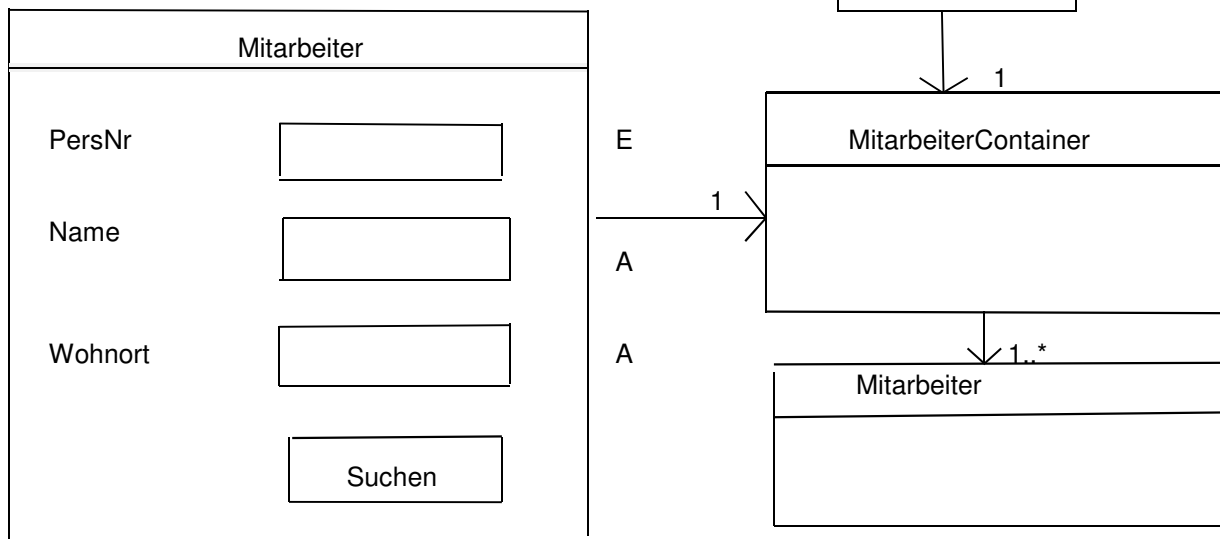
1. Sie besitzt eine statische Membervariable ihres eigenen Typs.
2. Sie besitzt einen privaten parameterlosen Konstruktor, um zu verhindern, daß andere Klassen durch Anwendung des new - Operators ein Objekt erzeugen.
3. Sie besitzt eine statische Methode: **getInstance ()**, mit der auf das Objekt zugegriffen werden kann. In ihr wird beim ersten Aufruf der Methode ein Objekt der Singleton- klasse erzeugt und bei weiteren Aufrufen nie wieder.
4. Sie kann weitere
 - Attribute und
 - Methoden haben.

Beispiel

Mit folgender View soll auf den MitarbeiterContainer zugegriffen werden.

Es sollen von einem Mitarbeiter mit einer bestimmten Personalnummer der

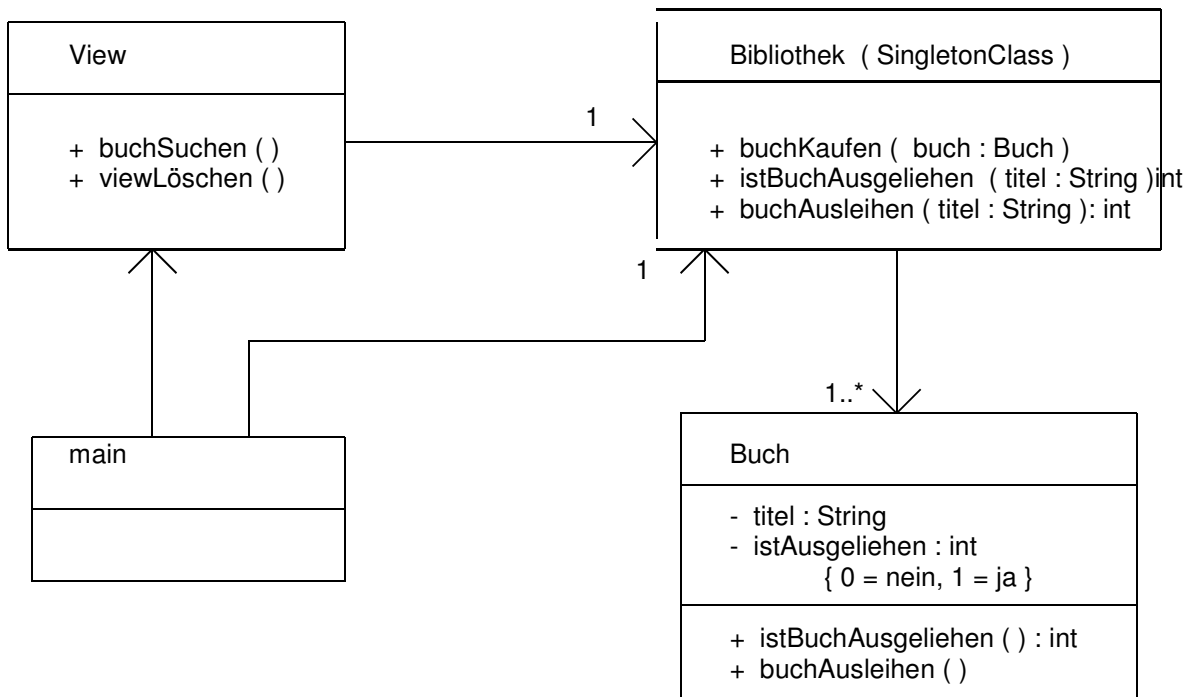
- Name und
 - Wohnort
- angezeigt werden



1. Die Mitarbeiterobjekte werden von der main- Funktion aus im Container gespeichert.
2. Wird die Personalnummer in die View eingegeben, sollen auf Anforderung (Suchen) Name und Wohnort dazu angezeigt werden.

Übung Singleton:

UML Klassendiagramm



- 1) Bücher kaufen (= buchKaufen (buch Buch)) wird über die main-Funktion realisiert
- 2) Bücher ausleihen (= buchAusleihen (titel : String)) wird über die Main-Funktion realisiert
- 3) Ein Buch suchen wird über die View realisiert
- 4) Die View kann gelöscht werden

8.2. Observer

Das Entwurfsmuster Observer baut eine Beziehung auf zwischen einem Subjekt (Observable) und seinem bzw. seinen Beobachtern (Observer). Eine Aktie (konkretes Subjekt) z.B. ändert häufig ihren Kurs. Die Zustandsänderungen werden von den Observern ‚protokolliert‘.

DV technisch muß sich jeder Observer beim Subjekt anmelden. Dann kann er nach jeder Zustandsänderung vom Subjekt informiert werden und darauf reagieren.

Teilnehmer am Observer- Muster:

1. Subjekt (Klasse Observable)
 - kennt seine Beobachter (beliebige Anzahl)
 - bietet eine Schnittstelle zum An- und Abmelden und zum Informieren von Beobachtern
2. Beobachter (Interface Observer)
 - definiert eine Aktualisierungsschnittstelle für Objekte, die von Änderungen eines Subjektes informiert werden sollen.
3. Konkretes Subjekt
 - erbt von Observable
 - speichert den jeweiligen Zustand nach der Änderung, benachrichtigt seine Observer von jeder Zustandsänderung
4. Konkreter Beobachter
 - verwaltet eine Referenz auf ein konkretes Objekt
 - speichert den Zustand, der mit dem des konkreten Subjektes in Einklang stehen muß.
 - implementiert die Aktualisierungsschnittstelle des Beobachter- Interfaces, um seinen Zustand mit dem des Subjektes konsistent zu halten.

Observer

Method Summary

void	update (Observable o, Object arg)
	This method is called whenever the observed object is changed.

Method Detail

update

```
void update (Observable o,  
             Object arg)
```

This method is called whenever the observed object is changed. An application calls an Observable object's `notifyObservers` method to have all the object's observers notified of the change.

Parameters: o - the observable object. arg - an argument passed to the `notifyObservers` method.

Observable

Method Summary

void	<u>addObserver</u> (<u>Observer</u> o) Adds an observer to the set of observers for this object, provided that it is not the same as some observer already in the set.
protected void	<u>clearChanged</u> () Indicates that this object has no longer changed, or that it has already notified all of its observers of its most recent change, so that the <code>hasChanged</code> method will now return <code>false</code> .
int	<u>countObservers</u> () Returns the number of observers of this <code>Observable</code> object.
void	<u>deleteObserver</u> (<u>Observer</u> o) Deletes an observer from the set of observers of this object.
void	<u>deleteObservers</u> () Clears the observer list so that this object no longer has any observers.
boolean	<u>hasChanged</u> () Tests if this object has changed.
void	<u>notifyObservers</u> () If this object has changed, as indicated by the <code>hasChanged</code> method, then notify all of its observers and then call the <code>clearChanged</code> method to indicate that this object has no longer changed.
void	<u>notifyObservers</u> (<u>Object</u> arg) If this object has changed, as indicated by the <code>hasChanged</code> method, then notify all of its observers and then call the <code>clearChanged</code> method to indicate that this object has no longer changed.
protected void	<u>setChanged</u> () Marks this <code>Observable</code> object as having been changed; the <code>hasChanged</code> method will now return <code>true</code> .

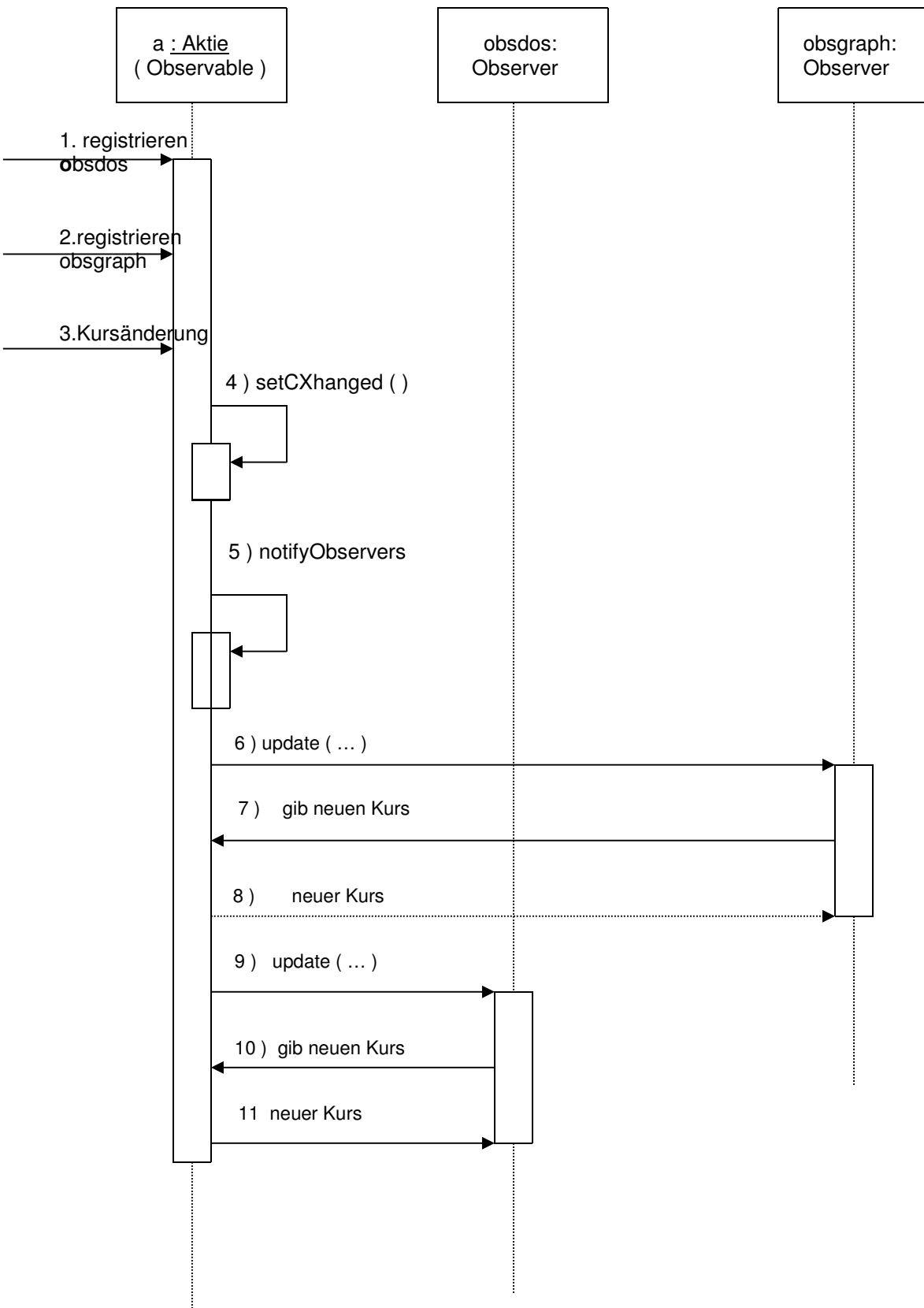
Constructor Detail

Observable

```
public Observable()  
    Construct an Observable with zero Observers.
```

Ablauf einer Datenänderung

Im main sind das Observableobjekt (Aktie) und die Observerobjekte (Dosbox und GUI)angelegt.



Beispiel:

Eine Aktie hat eine Bezeichnung und einen Kurs. Kursänderungen werden von zwei Observern beobachtet:

- Observer_Dosbox
Er sagt, daß er der Observer „Dosbox „ ist und gibt den Namen und den neuen Kurs der Aktie aus.
- Observer_Graphik
Er sagt, daß er der „Observer_Graphik „ ist und gibt auch Namen und den neuen Kurs aus. (Simuliert die Graphikdarstellung)

In der main- Funktion sollen alle Schritte ausgeführt werden, die das Arbeiten der Observer ermöglichen.

- Klassen:
- Aktie
 - Observer_Dos
 - Observer_Graphik
 - main- Funktion

Übung 1:

Ein Artikel hat:

- eine Nummer
- eine Bezeichnung
- einen Bestand
- einen Mindestbestand

Im main ist folgender Artikel angelegt: **Artikel artikel = new Artikel (1234, "Lampe", 8, 5);**
Die Mengenänderungen (mehrmals) erfolgen auch im main, z.B. **artikel.aendereBestand (1);**

Beispielhafte Ausgabe in der Dosbox (Mindestbestand ist 5) bei viermaliger Änderung:

Lampe Nr. 1234 Bestand : 7, Mindestbestand : 5, Bestand o.k.
Lampe Nr. 1234 Bestand : 6, Mindestbestand : 5, Bestand o.k.
Lampe Nr. 1234 Bestand : 5, Mindestbestand : 5, anfragen
Lampe Nr. 1234 Bestand : 4, Mindestbestand : 5, bestellen

View: (bei Änderungen erscheinen die gleichen Daten wie in der Dosbox)

Mindestbestand	
Artikel	<input type="text"/>
Bestand	<input type="text"/>
Mindestbestand	<input type="text"/>
Aktion	<input type="text"/>

-
Wird bei einer Bestandsänderung der Mindestbestand erreicht,
erscheint bei Aktion : der Text "anfragen";
Wird der Mindestbestand unterschritten, erscheint der Text "bestellen", sonst "Bestand o.k".
Erstellen Sie die Klassen Artikel, View, Dosbox und Anwendung

Übung 2

In einem Personalbestand werden Personenobjekte gespeichert. Eine Person hat als Attribut nur einen Namen. Gehen sie davon aus, daß ein Name nur einmal vorkommt. Die Namen können geändert werden.

Die Namen werden nach Änderung wie folgt angezeigt:

a) In einer DOS- Box : Name Meier wurde geändert in Müller

b) In einer View :

Personenregister	
Alter Name	<input type="text"/>
Neuer Name	<input type="text"/>

Eingabe

Ausgabe