4.3.1 Typ einer Methode

Als Rückgabetyp einer Methode sind alle einfachen Datentypen und Referenzdatentypen erlaubt. Die letzte Anweisung des Methodenkörpers ist dann zwingend die **return**-Anweisung.

```
return ausdruck;
```

ausdruck muss zuweisungskompatibel zum Rückgabetyp sein.

Erlaubt ist also auch die Rückgabe von Arrays:

```
static int[] xxx() { return new int[]{ 4, 3 }; }
```

Der Ausdruck **xxx()[1]** würde dann 3 liefern.

Es gibt ferner den Rückgabetyp **void**, der keine **return**-Anweisung erwartet.

4.3.2 Parameter einer Methode

Parameter werden **call-by-value** übergeben. Für Parameter mit Referenzdatentypen haben wir dann die Semantik einer **call-by-reference**-Übergabe: Es wird eine Kopie des Verweises übergeben; referenziert wird jedoch das Originalobjekt.

Objektorientierung/Autos/05-Parameter/Auto.java

```
27  public void abschleppen(Auto auto, double distanz)
28  {
29    this.fahren(distanz); // this ist optional, kürzer: fahren(distanz);
30    auto.fahren(distanz); // Test call-by-reference
31    distanz = 0;          // Test call-by-value
32    auto = null;          // Test call-by-value
33  }
```

```
45    bmw.abschleppen(meinPKW, d);
```

Bei der Instantiierung eines Objekts wird automatisch eine Referenz namens **this** erzeugt. Sie zeigt auf das Objekt selbst. Man kann **this** in der Punktnotation verwenden, um deutlich zu machen, dass es sich um ein Merkmal des Objekts selbst handelt. Intern ergänzt der Compiler **javac** alle Memberattribute und -methoden durch Voranstellen von **this**.

In dem Beispiel soll durch die Anweisung **this.fahren(distanz);** die Methode auf das Objekt selbst angewendet werden. Bei **auto.fahren(distanz);** ist dagegen das übergebene Objekt mit der Referenz **auto** gemeint.

4.3.3 Überladen von Methoden

Der Compiler **javac** unterscheidet Methoden anhand der vollständigen Methodensignatur. Man kann also innerhalb einer Klasse mehrere Methoden gleichen Namens definieren. Insbesondere kann man Methoden **überladen**, d. h. gleichnamige Methoden mit unterschiedlichen Parameterlisten deklarieren.

Objektorientierung/Autos/06-Überladen/Auto.java

```
22  public void ausgeben()
23  {
24    System.out.println("Tachostand = " + tachoStand + " km");
25  }
27  public void ausgeben(String name)
28  {
29    System.out.println("Tachostand von " + name + " = " + tachoStand + " km");
30  }
```

Welche Methode gewählt wird, wird anhand der Signatur entschieden.

```
37    bmw.ausgeben();
38    bmw.ausgeben("BMW Z3");
```


4.4 Konstruktoren und Garbage Collector

Fragen:

- Sie möchten direkt den Tachostand festlegen. Was geschieht beim Instantiieren von Objekten?

→ **Konstruktoren** arbeiten mit dem **new**-Operator zusammen.

- Was geschieht mit Objekten, die nicht mehr gebraucht werden?

→ Der **Garbage Collector** räumt den Speicher auf.

Wir betrachten wieder die zwei Ebenen bei der Entwicklung mit Java:

1. Implementierung der Klasse
2. Instantiierung von Objekten nach Vorlage der Klasse

4.4.1 Konstruktoren

Wie kann man das Instantiieren von Objekten beeinflussen?

- Ein **Konstruktor** dient zur Generierung von Objekten.
- Er wird innerhalb des Klassenkörpers implementiert.
- Er heißt genauso wie die Klasse.
- Er wird bei der Instantiierung eines Objektes durch den Operator **new** aufgerufen.
- Mehrere Konstruktoren für Objekte einer Klasse sind zulässig. Java unterscheidet die Konstruktoren anhand der Signatur, also anhand der Datentypen aus der Parameterliste.
- Es besteht die Möglichkeit, Konstruktoren zu verketten (siehe 5.2.4).
- Wird vom Entwickler kein Konstruktor implementiert, so verwendet Java einen Standardkonstruktor.

  Der setzt alle Attribute auf ihre Standardwerte (siehe Tabelle in 3.3, für Referenztypen **null**). Sobald ein eigener Konstruktor definiert wird, muss bei Bedarf auch der parameterlose Standardkonstruktor selbst definiert werden.

Wir erweitern unsere Klasse **Auto** um zwei Konstruktoren:

Objektorientierung/Autos/07-Konstruktoren/Auto.java

```
11  //Konstruktoren
12  /** erzeugt ein Auto mit Standard-Initialisierungen.
13   */
14  public Auto()
15  {
16    this.tachoStand = 100;
17  }
18
19  /** erzeugt ein Auto mit spezifizierten Initialisierungen.
20   *  @param tachoStand der spezifizierte Tachostand. Sollte ≥ 0 sein.
21   */
22  public Auto(double tachoStand)
23  {
24    if (tachoStand >= 0)
25      this.tachoStand = tachoStand;
26    else
27      this.tachoStand = 100;
28  }
```


4.4.2 Garbage Collector

Was geschieht mit Objekten, die nicht mehr gebraucht werden?

- Zum Entfernen nicht mehr benötigter Objekte stellt Java den Mechanismus der **Garbage Collection** bereit.
- Der **Garbage Collector** sucht in gewissen Abständen nach Objekten, die nicht mehr referenziert werden.
- Werden solche Objekte gefunden, so werden sie aus dem Speicher entfernt.
- Wir können aber kaum Einfluss darauf nehmen, **wann** das geschieht.
- Um ein nicht mehr benötigtes Objekt für den Garbage Collector freizugeben, müssen wir dafür sorgen, dass keine Referenz mehr darauf verweist. Wir können dies beispielsweise so erreichen:

  ```
  bmw = null;
  ```

- Der Aufruf

  ```
  System.gc();
  ```

  ist eine Empfehlung an die Virtual Machine, den Garbage Collector jetzt aufzurufen.

4.5 Klassenattribute und -methoden

Frage:

- Sie möchten wissen, wie viele Auto-Objekte Ihre Anwendung bislang erzeugt hat?

  Gibt es in Java globale Funktionen und Variablen?

→ nein, aber **statische** Merkmale

- Wir haben bislang **Objekt**attribute und -methoden kennen gelernt. Solche Merkmale sind an das Objekt gebunden.
- Ein **Klassenattribut** ist an die Klasse und nicht an eine bestimmte Instanz der Klasse gebunden.
- Es gibt nur eine Kopie eines Klassenattributes, unabhängig davon, wie viele Instanzen erzeugt werden.
- Klassenattribute werden in der Klassendeklaration durch das Schlüsselwort **static** gekennzeichnet.

Objektorientierung/Autos/08-static/Auto.java

```
8   // Klassenttribute
9   static int anzahl = 0;
```

- Zugriff erfolgt mit Hilfe der Punkt-Notation **Klassenname.Attributname**:

```
Auto.anzahl;
Color.black;
```

Objektname.Attributname ist auch möglich, aber schlechter Stil.

- **Klassenmethoden** werden in der Klassendeklaration ebenfalls durch das Schlüsselwort **static** gekennzeichnet.

```
30  // Klassenmethoden
31  public static int anzahlAutos()
32  {
33    return anzahl;
34  }
59  public static void main(String[] argv)
```

- Zugriff erfolgt mit Hilfe der Punkt-Notation **Klassenname.Methodenname()**:

```
63    System.out.println(Auto.anzahlAutos()); // druckt 1

      Math.sin(3.0);
```

- Klassenmethoden können nur auf Klassenvariablen verweisen und nur andere Klassenmethoden aus der aktuellen Klasse aufrufen.


```
14  //Konstruktoren
15  public Auto()
16  {
17    this.tachoStand = 100;
18    anzahl = anzahl + 1;
19  }
20
21  public Auto(double tachoStand)
22  {
23    if (tachoStand >= 0)
24      this.tachoStand = tachoStand;
25    else
26      this.tachoStand = 100;
27    anzahl = anzahl + 1;
28  }
```

```
61    Auto bmw = new Auto(500);
62    Auto zweiCV;
63    System.out.println(Auto.anzahlAutos()); // druckt 1
64    zweiCV = new Auto(-44444);
65    System.out.println(Auto.anzahlAutos()); // druckt 2
66    bmw = null;  // Objekt nicht mehr referenziert, könnte aufgeräumt werden
67    System.gc(); // evtl. jetzt (vielleicht auch später)
```


4.6 Gemischtes

Frage: Wie kann man **Konstanten** definieren?

Deklariert man ein Klassenattribut oder Objektattribut mit dem Modifizierer **final**, so kann das Attribut nach der erstmaligen Initialisierung nicht mehr geändert werden.

```
static final double ERDBESCHLEUNIGUNG = 9.81;
final double DICHTE_DES_OBJEKTS = 7.9;
```


Frage: Wie kann man abfragen, von welcher Klasse ein Objekt erzeugt wurde?

Antwort: mit Hilfe der Methode **getClass()** oder mit dem Operator **instanceof**

Objektorientierung/Autos/09-instanceof/Auto.java

```
32    System.out.println(bmw.getClass());   // Ausgabe: class Auto
33    if (bmw instanceof Auto)
34      System.out.println("ist ein Auto");
35    if ("FIAT" instanceof String)
36      System.out.println("ist ein String");
```

Frage: Java ist plattformunabhängig. Wie komme ich trotzdem an Informationen über das Betriebssystem?

Antwort: mit Hilfe der Klasse **java.lang.System**

4.7 Zusammenfassende Fragen

- Was sind Klassen?
- Was sind Objekte?
- Was sind Merkmale?
- Was ist der Unterschied zwischen Identität und Äquivalenz?
- Wie deklariert man Konstruktoren?
- Wie erzeugt man Objekte?
- Wie deklariert man Objektattribute/Objektmethoden?
- Wie greift man auf Objektattribute/Objektmethoden zu?
- Wie deklariert man Klassenattribute/Klassenmethoden?
- Wie greift man auf Klassenattribute/Klassenmethoden zu?
- Wie arbeitet der Garbage Collector?
- Was heißt Datenkapselung?
- Was sind Dokumentationskommentare?
- Was bedeuten die Schlüsselwörter `class`, `new`, `static`, `final`?

5 Objektorientierung in Java, Teil 2

Inhalt

Inhalt/Ziele: Wir lernen weitere Konzepte des objektorientierten Softwareentwurfs kennen. Wir konzentrieren uns dabei auf die Frage, wie in Java **Vererbung**, **Interfaces**, **Zugriffsrechte**, **Packages** usw. realisiert werden.

5.1 Neues Konzept: Vererbung

Frage:

- Wie lassen sich spezielle Autos verwalten, z. B. Geländewagen, Sportwagen, Limousinen, ...?

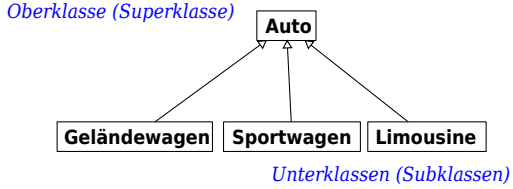
→ **Vererbung**

Wir haben gelernt, dass objektorientierte Sprachen wie Java Attribute und Methoden in Klassen vereinigen und dass Datenkapselung für die Einhaltung der Zusicherungen sorgt.

Drittes wichtiges Konzept

Objektorientierte Sprachen bieten die Möglichkeit der **Vererbung**. Wir haben die Möglichkeit, Merkmale vorhandener Klassen auf neue Klassen zu übertragen („zu vererben“).

Bei der Abbildung der realen Welt nach Software lassen sich oft Hierarchien ausnutzen:



- Das Bild gibt eine **Klassenhierarchie** wieder. Die **Subklassen** **`Geländewagen`**, **`Sportwagen`** und **`Limousine`** **erben** von der **Superklasse** **`Auto`**. (Subklasse = Unterklasse, Superklasse = Oberklasse)
- Eine Subklasse **erbt** Attribute und Methoden von ihrer Superklasse (z. B. **`tachoStand`**, **`fahren(double distanz)`**). Sie kann diese verwenden, als wären sie in der Subklasse selbst deklariert.

Spezialisierung: In den Subklassen können die Methoden und Attribute der Superklasse entweder **wiederverwendet** oder **überlagert** werden. Zusätzlich dürfen die Subklassen um neue Methoden und Attribute **ergänzt** werden.

Weitervererbung: Subklassen können selber **weitervererbt** werden. Soll eine Klasse nicht mehr weiter vererbt werden können, kennzeichnet man dies durch das Schlüsselwort **`final`**. Genauso kann man das Überlagern von Methoden durch eine Subklasse verbieten.

Mehrfachvererbung: Java unterstützt keine **Mehrfachvererbung**, also das Erben von mehreren Superklassen. Als Alternative gibt es die so genannten **Interfaces**.

abstrakte Klassen: Es besteht die Möglichkeit, Klassen so zu definieren, dass erst die Subklassen die Definitionen mit Leben füllen müssen. Man kann diese Klassen erkennen an dem Schlüsselwort **`abstract`**. Solche Klassen können nicht instantiiert werden.

5.2 Vererbung in Java

In Java wird das Schlüsselwort **`extends`** verwendet, um von anderen Klassen zu erben.

Objektorientierung/Autos/10-Vererbung/Limousine.java

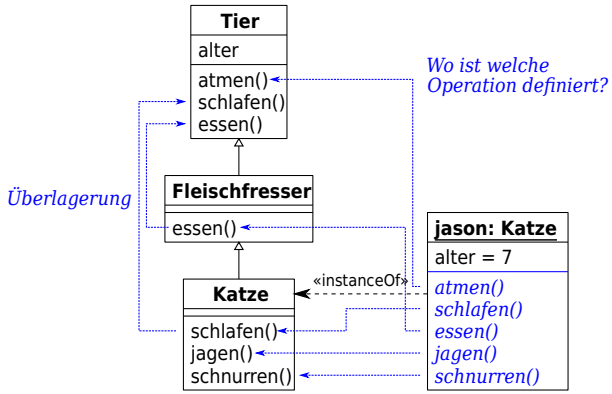
```
6  public class Limousine extends Auto
7  {
9    int sicherheitsKategorie;
12   public Limousine()
13   {
14     this.sicherheitsKategorie = 3;
15   }
29   public void setzeSicherheitsKategorie(int sicherheitsKategorie)
30   {
31     if ((sicherheitsKategorie > 0) && (sicherheitsKategorie < 4))
32       this.sicherheitsKategorie = sicherheitsKategorie;
33   }
35   public int leseSicherheitsKategorie()
36   { return this.sicherheitsKategorie; }
39   @Override public void ausgeben()
40   {
43   }
```


Quelltextanalyse

- Die Klasse **`Limousine`** erweitert die Klasse **`Auto`**.
- Ergänzung: Objektattribut **`sicherheitsKategorie`**
- Ergänzung: Objektmethoden **`setzeSicherheitsKategorie()`** (mit Zusicherung) und **`leseSicherheitsKategorie()`**
- Überlagerung: Die Methode **`ausgeben()`** der Superklasse **`Auto`** wird neu definiert und erhält eine andere Implementierung.
- Die optionale Annotation **`@Override`** zeigt dem Compiler an, dass eine geerbte Methode überschrieben werden soll. Sie verhindert, dass man durch Tippfehler versehentlich eine neue Methode mit anderem Namen oder anderen Parametern ergänzt statt eine vorhandene Methode zu überschreiben.
- anderes Beispiel für **`extends`** siehe 2.3:

```
public class HelloSwing extends JFrame
```


Ein Beispiel zur Überlagerung



5.2.1 Typkonvertierungen

Fragen: Objekte sind Referenzdatentypen.

- Kann man eine Referenz vom Typ **Auto** auf eine Instanz vom Typ **Limousine** verweisen lassen?
- Und umgekehrt?
- Hat eine Typkonvertierung irgendwelche Auswirkungen?

- Wir unterscheiden prinzipiell zwischen **erweiternden** Typkonvertierungen und **einschränkenden** Typkonvertierungen.
- Die Umwandlung eines Referenzdatentyps in eine seiner Superklassen nennen wir erweiternde Konvertierung.

Objektorientierung/Autos/10-Vererbung/Limousine.java

```
48      Limousine jaguar = new Limousine();
52      Auto meinPKW = new Auto();
56      meinPKW = jaguar;
```


- Erweiternde Konvertierungen werden in Java automatisch vorgenommen. Dies kennen wir schon bei einfachen Datentypen:

```
int m;
byte j = 120;
m = j;
```

- Folgende Anweisungen sind also erlaubt:

```
59      Auto zweiCV = new Auto();
60      zweiCV.abschleppen(jaguar, 200);
```

- Die Umwandlung eines Referenzdatentyps in eine seiner Subklassen nennen wir einschränkende Konvertierung. Sie muss mit dem Type-Cast-Operator vorgenommen werden und erfordert sorgfältige Behandlung:

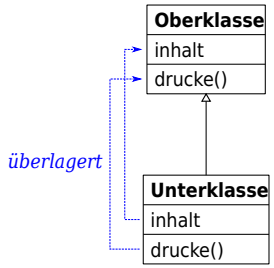
```
64      Limousine daimler = new Limousine(5000, 2);
68          daimler = (Limousine)zweiCV;  // Fehler
77          daimler = (Limousine)meinPKW; // OK, meinPKW ist Limousine
```

```
j = (byte)m;
```


5.2.2 Überlagerung und Binden

Frage: Welche Methoden ruft ein konvertiertes Objekt auf? Die der Subklasse oder die der Superklasse?

Wir haben gesehen, dass eine Variable der Klasse **Auto** auf Objekte verschiedener Subklassen verweisen kann (z.B. **Limousine**, **Sportwagen**). Jedes Objekt einer Subklasse ist gleichzeitig ein Objekt der Superklasse. (Eine **Limousine** ist ein **Auto**. Ein **byte** ist ein **int**.)
Beim Übersetzen mit **javac** kann der Compiler dann noch nicht wissen, welche der überlagerten Methoden aufgerufen werden soll.
Wir nehmen jetzt an, dass in einer Subklasse **Unterklasse** Merkmale (= Attribute und Methoden) der Superklasse **Oberklasse** überlagert werden.



Objektorientierung/BindTest/1/BindTest.java

```
28  class Oberklasse
29  {
30    String inhalt = "Attribut Oberklasse";
31
32    public void drucke() { System.out.println("Methode  Oberklasse" + "\n"); }
33  }
34
35  class Unterklasse extends Oberklasse
36  {
37    // Überlagerung eines Attributs
38    String inhalt = "Attribut Unterklasse";
39
40    // Überlagerung einer Methode
41    @Override public void drucke()
42    { System.out.println("Methode  Unterklasse" + "\n"); }
43  }
```

- Welche Merkmale sind gemeint, wenn ein Objekt **o** der Superklasse ein Objekt **u** der Subklasse zugewiesen bekommt?
- Welche Merkmale sind gemeint, wenn ein Objekt **u** der Subklasse als ein Objekt der Superklasse verwendet wird?
- Diese Fragen beantwortet das **Binden**:
  Dynamisches Binden liefert weiterhin das Merkmal der Subklasse = Typ des Objektes selbst.
  Statisches Binden liefert dagegen das Merkmal der Superklasse = Typ der Referenzvariablen.
- Java verwendet dynamisches Binden für Methoden und statisches Binden für Attribute.

```
11    Unterklasse u = new Unterklasse();
12    System.out.println(u.inhalt);                 // druckt Attribut Unterklasse
13    u.drucke();                                   // druckt Methode  Unterklasse
14
15    Oberklasse o = new Oberklasse();
16    System.out.println(o.inhalt);                 // druckt Attribut Oberklasse
17    o.drucke();                                   // druckt Methode  Oberklasse
18
19    o = u;
20    System.out.println(o.inhalt);                 // druckt Attribut Oberklasse
21    o.drucke();                                   // druckt Methode  Unterklasse
22
23    System.out.println(((Oberklasse)u).inhalt); // druckt Attribut Oberklasse
24    ((Oberklasse)u).drucke();                     // druckt Methode  Unterklasse
```

5.2.3 Die Referenz super

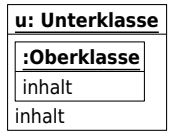
- Die Referenz **super** können wir benutzen, um auf Merkmale einer Superklasse zu verweisen, die überlagert wurden.
- **super** veranlasst, dass die Suche nach der Methode, dem Attribut oder dem Konstruktor nicht in der aktuellen Klasse, sondern erst in der unmittelbaren Superklasse beginnt.

Objektorientierung/Autos/10-Vererbung/Limousine.java

```
39  @Override public void ausgeben()
40  {
41    super.ausgeben();
42    System.out.println("Kategorie = " + sicherheitsKategorie);
43  }
```

Objektorientierung/BindTest/2-super/BindTest.java

```
28  class Unterklasse extends Oberklasse
29  {
38    public void zeigeInhalte()
39    {
40      System.out.println("Eigener Inhalt:      " + inhalt);
41      System.out.println("Überlagerter Inhalt: " + super.inhalt);
42    }
45    public void druckeAlles()
46    {
47      drucke();
48      super.drucke();
49    }
11      Unterklasse u = new Unterklasse();
12      u.zeigeInhalte();
13      // Eigener Inhalt:      Attribut Unterklasse
14      // Überlagerter Inhalt: Attribut Oberklasse
15      u.druckeAlles();
16      // Methode  Unterklasse
17      // Methode  Oberklasse
```



Tatsächlich enthält **u** in unserem Beispiel **zwei** Attribute names **inhalt**:

5.2.4 Verkettung von Konstruktoren

- Ist die erste Anweisung eines Konstruktors eine normale Anweisung – also kein Aufruf von **this()** oder **super()** – so ergänzt Java automatisch einen impliziten Aufruf von **super()** zum Aufruf des Standardkonstruktors der Superklasse.

  Nach Rückkehr von diesem Aufruf initialisiert Java die Instanzattribute der aktuellen Klasse und fährt mit der Ausführung der Anweisungen des aktuellen Konstruktors fort.
- Ist die erste Anweisung eines Konstruktors der Aufruf eines Superklassenkonstruktors über **super()**, ruft Java den gewünschten Superklassenkonstruktor auf.

  Nach Rückkehr von diesem Aufruf initialisiert Java die Instanzattribute der aktuellen Klasse und fährt mit der Ausführung der Anweisungen des aktuellen Konstruktors fort.
- Ist die erste Anweisung eines Konstruktors der Aufruf eines überladenen Konstruktors über **this()**, ruft Java den gewünschten Konstruktor auf und führt danach einfach die Anweisungen im aktuellen Konstruktor aus.

  Der Aufruf des Superklassenkonstruktors vollzieht sich explizit oder implizit innerhalb des überladenen Konstruktors, so dass die Initialisierung der Instanzvariablen bereits dort stattgefunden hat.

Instantiierung „von oben nach unten“

Objektorientierung/Autos/10-Vererbung/Limousine.java

```
17  public Limousine(double tachoStand, int sicherheitsKategorie)
18  {
19    super(tachoStand);
20    if ((sicherheitsKategorie > 0) && (sicherheitsKategorie < 4))
21      this.sicherheitsKategorie = sicherheitsKategorie;
22    else
23      this.sicherheitsKategorie = 3;
24  }
```

Objektorientierung/Autos/10-Vererbung/Auto.java

```
15  public Auto()
16  {
17    this(100);
18  }
```

5.3 Datenkapselung

Designmerkmale einer objektorientierten Sprache wie Java:

- Klassen vereinigen Attribute, Methoden und Zusicherungen.
- Vererbung erlaubt das Übertragen von Merkmalen vorhandener Klassen auf neue Klassen.
- Datenkapselung durch Zugriffsrechte

Klassen, ihre Attribute und Methoden können durch Zugriffsbezeichner modifiziert werden. Durch diese Bezeichner kann man die Zugriffsrechte und -möglichkeiten genauer definieren.

In Java unterscheidet man vier Zugriffsrechte:

Zugriffsrecht	Schlüsselwort	Beispiel
private	`private`	`private int i;`
protected	`protected`	`protected int i;`
package		`int i;`
public	`public`	`public int i;`

Man benutzt Zugriffsrechte, um Datenkapselung zu erzielen. Um z. B. zu vermeiden, dass Objektattribute unkontrolliert von außerhalb geändert werden können, kann man das Schlüsselwort **`private`** verwenden.

Objektorientierung/Geld/1/Sparstrumpf.java

```
10  // Auf k kann von außen nicht zugegriffen werden.
11  private float k;
27  // Eine Methode, die von außen aufgerufen werden kann.
28  public float auszahlen(float r)
41  // Eine Methode, die nicht von außen aufgerufen werden kann.
42  private void umrechnenYen()
```

Objektorientierung/Geld/1/Test.java

```
13    bar = s.auszahlen(1000);
14    // System.out.println("Kapital = " + s.k); // Fehler beim Compilieren
15    // s.umrechnenYen();                       // Fehler beim Compilieren
```


5.4 Packages

- Ein **Package** ist eine Sammlung verwandter Klassen, die Zugriffsrechte und Namensgebung verwaltet.
- Es gibt im JDK bereits eine Reihe nützlicher Packages, z. B. **`java.awt`**, **`java.io`**, **`java.net`**, **`javax.swing`** usw.
- Das Package **`java.lang`** wird automatisch eingebunden. Es enthält neben den elementaren Sprachkonstrukten auch die Klassen **`Math`** (3.7), **`System`** (4.6), **`Thread`** (13), **`Object`** (5.9), **`Class`** (5.9) sowie diverse Wrapperklassen (5.10) für einfache Datentypen.
- Man kann natürlich auch eigene Packages erstellen.
  - Man kann Packages einbinden durch die Anweisung **`import`**.

Objektorientierung/Packages/Test.java

```
1  import autos.*;
```

  - Man kann Packages erzeugen durch die Anweisung **`package`**.

Objektorientierung/Packages/autos/Auto.java

```
1  package autos;
```

Objektorientierung/Packages/autos/Limousine.java

```
1  package autos;
```


  - Die Quelltextdateien sollten in einem gleichnamigen Unterverzeichnis liegen.
  - Man kann Packages auch ohne **`import`** verwenden, wenn man den Klassennamen vollständig qualifiziert.

Objektorientierung/Packages/Test2.java

```
10    autos.Limousine jaguar = new autos.Limousine();
```

  - Bei einer statischen Funktion kann auf die Qualifikation durch den Klassennamen verzichtet werden, wenn diese statisch importiert wird (siehe 3.7).

Grundlagen/Math/MathBsp3.java

```
7  import static java.lang.Math.*;
```

- Trifft der Compiler **`javac`** auf eine **`import`**-Anweisung, so durchsucht er den **`CLASSPATH`**, um die Verzeichnisse mit den Packages zu finden.
- Der Suchpfad kann beim Aufruf des Compilers mit der Option **`-classpath`** (kurz **`-cp`**) oder über die Umgebungsvariable **`$CLASSPATH`** angegeben werden.
- Gibt man beim Aufruf keinen speziellen Pfad an, so wird zuerst das aktuelle Verzeichnis und dann das Standardverzeichnis der mitgelieferten JDK-Packages durchsucht.
- Benötigt der Compiler Klassen, die noch nicht compiliert sind, so werden diese **automatisch** mit übersetzt.

- Hier genügt es also, dem Compiler den Quelltext mit dem Hauptprogramm zu übergeben.

```
…/Quellen/Objektorientierung/Packages > rm -rf *.class autos/*.class
…/Quellen/Objektorientierung/Packages > javac Test.java
…/Quellen/Objektorientierung/Packages > ls -l *.class autos/*.class
-rw-r--r-- 1 holger users  320 27. Apr 13:03 Test.class
-rw-r--r-- 1 holger users 1407 27. Apr 13:03 autos/Auto.class
-rw-r--r-- 1 holger users 1227 27. Apr 13:03 autos/Limousine.class
```

- Bestandteile des Package sollten von außerhalb compiliert werden, sonst sind überraschende Fehlermeldungen möglich:

```
…/Quellen/Objektorientierung/Packages > rm -rf *.class autos/*.class
…/Quellen/Objektorientierung/Packages > cd autos
…/Quellen/Objektorientierung/Packages/autos > javac Limousine.java
Limousine.java:8: Fehler: Symbol nicht gefunden
public class Limousine extends Auto
                               ^
  Symbol: Klasse Auto
…
```

  Der Compiler sieht zwar die Datei **`Auto.java`**, wegen der Package-Deklaration müsste diese aber in einem Unterverzeichnis **`autos`** stehen, wir sind aber schon in diesem Verzeichnis.

- So klappt es:

```
…/Quellen/Objektorientierung/Packages/autos > cd ..
…/Quellen/Objektorientierung/Packages > javac autos/Limousine.java
```