## 5.5 Zugriffsrechte bei Vererbung und Packages

Siehe **Objektorientierung/Zugriff/Test.java**, **Objektorientierung/Zugriff/paket/A.java**, **Objektorientierung/Zugriff/paket/B.java**, **Objektorientierung/Zugriff/C.java**, **Objektorientierung/Zugriff/paket/ASub1.java**, **Objektorientierung/Zugriff/ASub2.java**.

Die Ergebnistabelle fasst die Zugriffsrechte zusammen:

| | Zugriff von | | | |
|---|---|---|---|---|
| auf A/M mit Zugriffsrecht | Klasse | Unterklasse (aber nicht Package) | Package | fremde Klasse |
| private | ✓ | | | |
| protected | ✓ | ✓ | ✓ | |
| package | ✓ | | ✓ | |
| public | ✓ | ✓ | ✓ | ✓ |

- Zugriffsrechte wirken auf Attribute und Methoden gleich.
- Das Attribut **public** ist zusätzlich auch bei der Klassendeklaration von Bedeutung.
- Nur Klassen, die als **public** deklariert wurden, sind außerhalb ihres Packages sichtbar.
- Pro Java-Quelltextdatei darf nur eine **public**-Klasse angelegt werden.
- Es ist guter Stil, die Zugriffsrechte auf Attribute soweit wie möglich einzuschränken (Datenkapselung!).
- (Package/protected bieten keinen vollständigen Schutz; vorhandene Packages können jederzeit durch weitere gleichnamige Verzeichnisse erweitert werden.)



## 5.6 Der Modifizierer final

- Modifiziert man ein **Attribut** mit dem Schlüsselwort **final**, so darf sein Wert nicht mehr geändert werden. Das Attribut kann man dann als Konstante verwenden.
- Modifiziert man eine **Methode** mit dem Schlüsselwort **final**, so darf sie nicht mehr überlagert werden. Das erspart dem Compiler das dynamische Binden und kann zur Geschwindigkeitsoptimierung benutzt werden.
- Modifiziert man eine **Klasse** mit dem Schlüsselwort **final**, so darf sie keine Subklassen mehr bilden.

## 5.7 Abstrakte Klassen

**Problem:**

- Es soll ein Wertpapierdepot verwaltet werden. Alle Wertpapiere sollen über eine Methode **berechneWert()** verfügen. Der Wert eines Sparbuchs berechnet sich aber ganz anders als der von Aktien.

→ Wir sollten eine abstrakte Klasse **Wertpapier** mit einer abstrakten Methode **berechneWert()** deklarieren. Jede Klasse, die davon erben will, **muss** dann eine konkrete Implementierung vornehmen.



**Objektorientierung/Geld/2/Wertpapier.java**

```
6  public abstract class Wertpapier
7  {
8    public abstract float berechneWert();
9  }
```

**Quelltextanalyse**

- Das Schlüsselwort **abstract** kennzeichnet eine Klasse oder Methode als abstrakt.
- Bei einer **abstrakten Methode** ersetzt ein Semikolon den Methodenkörper.
- Eine **abstrakte Klasse** kann neben abstrakten Methoden durchaus normale (nicht abstrakte) Attribute und Methoden enthalten.
- Von einer abstrakten Klasse können keine Instanzen gebildet werden.



- Um Objekte zu erzeugen, müssen wir zunächst eine Subklasse bilden:

**Objektorientierung/Geld/2/Sparbuch.java**

```
6  public class Sparbuch extends Wertpapier
25   @Override public float berechneWert()
26   {
27     float wert = k;
28     for (int i = 1; i <= l; i++)
29       wert = wert * (1 + p);
30     return wert;
31   }
```

**Objektorientierung/Geld/2/Aktie.java**

```
17   @Override public float berechneWert()
18   {
19     return anzahl * kurswert;
20   }
```

**Quelltextanalyse**

- Wir **müssen** dann die abstrakte Methode **public float berechneWert()** implementieren.
- Für eine abgeleitete Klasse **Aktie** kann eine ganz andere Implementierung der Methode vorgenommen werden.
- Von den Subklassen dürfen wir Objekte instantiieren.

**Objektorientierung/Geld/2/Depot.java**

```
10     Sparbuch sb = new Sparbuch(5112.92f, 0.0125f, 5);
11     System.out.println(sb.berechneWert());
```

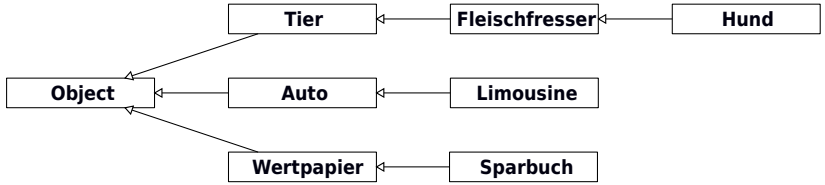
## 5.8 Interfaces

**Problem:** Die Steuererklärung steht an. Es sollte für Autos, Sparbücher und Hunde eine einheitliche Methode **`berechneSteuer()`** geben. Leider gehören die Klassen zu **verschiedenen** Vererbungshierarchien.

- Eine grundsätzlich Designentscheidung der Java-Erfinder war, keine **Mehrfachvererbung** zu unterstützen.

  Das heißt, dass jede Klasse nur von einer einzigen Oberklasse erben kann.

- Nehmen wir einmal an, dass wir die folgenden Vererbungshierarchien aufgebaut haben:




Siehe **`Objektorientierung/Interfaces`**.





- Nun möchten wir unsere Steuererklärung in Java programmieren. Zu versteuern sind alle Instanzen der Klassen **`Hund`**, **`Auto`** und **`Sparbuch`**.
- Es wäre nützlich, wenn für jede dieser Klassen eine Methode **`berechneSteuer()`** zur Verfügung stünde.
- Mit der Hilfe von **Interfaces** bietet Java eine Konstruktion an, um solche Anforderungen zu lösen:
- Wir erstellen zunächst ein Interface **`Steuer`**.

**`Objektorientierung/Interfaces/Steuer.java`**

```
6  public interface Steuer
7  {
8    double berechneSteuer();
9  }
```

- Dort listen wir die Methoden auf, die implementiert werden müssen, wenn wir das Interface benutzen wollen.
- Alle Methoden sind automatisch **`public`**.
- Interfaces dürfen keine Objektattribute und bis Java 7 auch keine nicht abstrakten Methoden enthalten.
- Eine Klasse, die das Interface benutzen will, zeigt dies durch das Schlüsselwort **`implements`** an. Wir müssen dann alle Methoden des Interfaces implementieren, es sei denn, die implementierende Klasse ist abstrakt.



**`Objektorientierung/Interfaces/Hund.java`**

```
6  public class Hund extends Fleischfresser implements Steuer
7  {
18   @Override public double berechneSteuer()
19   {
20     double steuerBetrag = 120;
21     return steuerBetrag;
22   }
23 }
```

- Siehe **`Objektorientierung/Interfaces/Sparbuch.java`**.
- Siehe **`Objektorientierung/Interfaces/Auto.java`**.
- Siehe **`Objektorientierung/Interfaces/Limousine.java`**.

- Nun verfügen alle Objekte dieser Klassen über eine Methode **`berechneSteuer()`**, die man so verwenden kann:

**`Objektorientierung/Interfaces/Test.java`**

```
10   Hund bello = new Hund();
11   Sparbuch sb = new Sparbuch(1200, 0.015f, 1);
12   Auto meinPKW = new Auto();
13   Limousine jaguar = new Limousine();
14   Steuer[] steuerObjekte = { bello, sb, meinPKW, jaguar };
15
16   double steuern = 0;
17   for (Steuer so : steuerObjekte)
18     steuern += so.berechneSteuer();
19   System.out.printf("zu zahlen: %7.2f\n", steuern);
```



- Eine Klasse kann mehrere Interfaces implementieren.
- Die Annotation **`@Override`** ist auch bei Interfaces erlaubt.
- Wann benutzen wir Interfaces?
  - Wenn wir ereignisgesteuerte Benutzeroberflächen programmieren wollen (z. B. **`ActionListener`**, siehe 10.4.3).
  - Wenn wir **Threads** programmieren wollen (**`Runnable`**, siehe 13.1).
- Wenn eine Anwendung sehr viele Konstanten definiert, kann man diese in einem Interface zusammenfassen. Jede Klasse, die diese Konstanten verwenden will, kann das Interface dann benutzen.

**`Objektorientierung/Interfaces/Defs/Defs.java`**

```
6  public interface Defs
7  {
14   static final int BUFFERLENGTH = 8;
15   static final int DEFAULTTAG = 0;
16   static final int DEFAULTERR = -1;
17 }
```

Diese Konstanten haben immer den Status **`static final`**, egal ob dies explizit angegeben wird oder nicht.

Eine nicht-instantiierbare Klasse (mit privatem Konstruktor) wäre hierfür aber sinnvoller – und die Verwendung ist dank **`import static`** genauso einfach.

Seit Java 8 dürfen Interfaces **Default-Methoden** beinhalten:

- Schlüsselwort **default** und normale Implementierung der Methode innerhalb des Interface
- Methode wird bei Implementierung des Interface geerbt
- statische Methoden in Interfaces nun ebenfalls erlaubt
- Im Unterschied zu abstrakten Klassen bleiben Objektattribute und Konstruktoren verboten.

**Objektorientierung/Interfaces/Default/Person.java**

```
6  public interface Person
7  {
8    default void sagHallo() { System.out.println("Hallo"); }
9  }
```

**Objektorientierung/Interfaces/Default/Student.java**

```
6  public class Student implements Person
7  {
8    private String name;
9    public Student(String n) { name = n; }
10 }
```

**Objektorientierung/Interfaces/Default/Test.java**

```
6  public class Test
7  {
8    public static void main(String[] argv)
9    {
10     Student hans = new Student("Hans");
11     hans.sagHallo();
12   }
13 }
```

Seit Java 9 dürfen Interfaces private (nicht abstrakte) Methoden beinhalten.





## 5.9 Object und Class

- Die Klasse **Object** aus dem Package **java.lang** ist die Mutter aller Klassen.
- Eine Klassendeklaration ohne die Klausel **extends** erweitert der Compiler **javac** automatisch zu **extends Object**.
- Die Klasse **Object** verfügt über einige nützliche Methoden, die wir bei der Definition eigener Klassen überlagern können (siehe auch API-Referenz).
  - **toString()**: gibt eine Darstellung des Objekts als String zurück
  - **equals()**: testet auf Äquivalenz zweier Objekte (siehe 4.2)
  - **clone()**: erzeugt ein neues Objekt derselben Klasse. Klassen, die eine **clone**-Methode implementieren, sollten dies durch **implements Cloneable** anzeigen.

    **Cloneable** ist ein sog. Markerinterface, d. h. es sind keine Methoden zu implementieren.
  - **hashCode()**: gibt einen Hashwert zurück; benötigt, falls Objekte in **HashMap**s oder **Hashtable**s gespeichert werden sollen (siehe 8 und 11.7)
  - **getClass()**: gibt eine Referenz auf die Klasse zurück, von der das Objekt abgeleitet ist



- Klassen im Java-Quellcode werden zur Laufzeit durch Instanzen der Klasse **Class** aus dem Package **java.lang** verwaltet.
- Zu jeder Klasse, die wir verwenden, gibt es ein **Class**-Objekt.
- Dieses ist für die Erstellung von Instanzen dieser Klasse verantwortlich.
- Die Klasse **Class** verfügt ebenfalls über einige nützliche Methoden (siehe API-Referenz). (Die Bedeutung von **<?>** wird in 7.2 erklärt werden.)

**Objektorientierung/Autos/11-Ahnen/Ahnen.java**

```
10   static void zeigeAhnenGalerie(Object obj)
11   {
12     System.out.print("Der␣Stammbaum␣von␣");
13     Class<?> cl = obj.getClass();
14     while (cl != null)
15       {
16         System.out.println("␣␣" + cl.getName());
17         cl = cl.getSuperclass();
18       }
19   }
```



- Die Methode **public boolean equals(Object obj)** muss zunächst prüfen, ob **obj** nicht **null** ist und zur selben Klasse gehört; danach wird der Inhalt verglichen.

  Das Ergebnis der **getClass()**-Aufrufe kann mit **!=** verglichen werden, denn es gibt zu allen Instanzen einer Klasse immer nur ein einziges **Class**-Objekt für deren Verwaltung in der virtuellen Maschine.
- Beim Überschreiben der Methode **public Object clone()** ist es erlaubt, den Rückgabetyp durch eine Unterklasse (z. B. hier **Auto**) zu ersetzen.
- Wird **equals()** überschrieben, sollte auch **public int hashCode()** überschrieben werden. Java verlangt, dass gleiche (äquivalente) Objekte den gleichen Hashcode haben.

**Objektorientierung/Autos/12-equals-clone-toString/Auto.java**

```
6  public class Auto implements Cloneable
7  {
47   // überlagert die Methode equals() von Object
48   @Override public boolean equals(Object obj)
49   {
50     if (obj == null || obj.getClass() != getClass())
51       return false;
52     return this.tachoStand == ((Auto)obj).tachoStand;
53   }
55   // überlagert die Methode clone() von Object
56   @Override public Auto clone()
57   {
58     return new Auto(this.tachoStand);
59   }
61   // überlagert die Methode toString() von Object
62   @Override public String toString()
63   {
64     return "Instanz von Auto, tachoStand = " + tachoStand;
65   }
67   // überlagert die Methode hashCode() von Object
68   @Override public int hashCode()
69   {
70     return (int)tachoStand;
71   }
```





```
74   public static void main(String[] argv)
75   {
76     Auto meinPKW = new Auto();
77     Auto bmw = new Auto();
78     Auto zweiCV;
79
80     // Identität und Äquivalenz mit == und equals()
81     meinPKW.fahren(100);
82     bmw.fahren(100);
83     System.out.println(bmw == meinPKW);      // false
84     System.out.println(bmw.equals(meinPKW)); // true
85     bmw.fahren(100);
86     System.out.println(bmw.equals(meinPKW)); // false
87     System.out.println(bmw.equals("Hallo")); // false
88     zweiCV = bmw.clone();
89     System.out.println(bmw == zweiCV);       // false
90     System.out.println(bmw.equals(zweiCV));  // true
91     zweiCV = null;
92     System.out.println(bmw.equals(zweiCV));  // false
93
94     // Test toString()
95     System.out.println(bmw);             // Instanz von Auto, tachoStand = 300.0
96     System.out.println(new Object());    // gibt z.B. java.lang.Object@58644d46
```

## 5.10 Wrapperklassen

**Frage:** Wie sind einfache Datentypen wie **int** oder **double** in die Klassenhierarchie einzuordnen?

- Eine Designentscheidung von Java ist es, einfache Datentypen **nicht** als Objekte zu betrachten.



- Es gibt jedoch Möglichkeiten, Variablen von einfachen Datentypen in eine objektorientierte Hülle einzuwickeln:

| einfacher Datentyp | Wrapperklasse |
|---|---|
| **byte** | **Byte** |
| **short** | **Short** |
| **int** | **Integer** |
| **long** | **Long** |
| **float** | **Float** |
| **double** | **Double** |
| **boolean** | **Boolean** |
| **char** | **Character** |
| **void** | **Void** |

- Mit Hilfe dieser Klassen aus dem Package **java.lang** lassen sich einfache Datentypen in Referenzdatentypen umwandeln.
- Insbesondere ist die Konvertierung von und nach **String** durch entsprechende Methoden der Klassen erleichtert.

**Objektorientierung/Wrapperklassen/WrapBsp.java**

```
10     double dx = 10;
11     Double dxw = Double.valueOf(dx);
12     String sx = "3.1415927";
13     Double sxw = Double.valueOf(sx);
14     double pi = sxw.doubleValue();
```



- Die Umwandlungen von einem Wert in sein entsprechendes Wrapper-Objekt und umgekehrt werden bei Bedarf automatisch vom Compiler durchgeführt **(Autoboxing/Unboxing)**.

**Objektorientierung/Wrapperklassen/WrapBsp2.java**

```
11     double d = 17;
12     Double dw;
13
14     dw = d;      // Autoboxing
15     d = dw;      // Unboxing
```

- Vorsicht! Bei Vergleichen mit **==** gilt weiterhin die Referenz-Semantik.

**Objektorientierung/Wrapperklassen/WrapBsp3.java**

```
13     int i;
14     Integer i1;
15     Integer i2;
16     i = 6;
17     i1 = i;
18     i2 = i;
19     System.out.println(i1 == i2);        // true (?!)
20     System.out.println(i1.equals(i2));   // true
21     i = 128;
22     i1 = i;
23     i2 = i;
24     System.out.println(i1 == i2);        // false
25     System.out.println(i1.equals(i2));   // true
```

- Bei mehreren gleichen „kleinen“ Zahlen legt Java offenbar nur ein Wrapper-Objekt an, um Speicher zu sparen. (Wrapper-Klassen sind sog. wertbasierte Klassen.)

## 5.11 Records

Records (seit Java 16) sind eingeschränkte Klassen, die im einfachsten Fall nur aus einer Liste von Attributen bestehen.

- Die Objekte sind unveränderlich.
- Der Compiler erzeugt automatisch
  - einen Konstruktor,
  - Zugriffsmethoden (gleicher Name wie Attribut),
  - Methoden **`equals()`**, **`hashCode()`**, **`toString()`**.
- Records sind automatisch abgeleitet von **`java.lang.Record`**.
- Zusätzliche Konstruktoren und weitere Methoden sind erlaubt.

**Objektorientierung/Autos/Record1/Auto.java**

```
6  public record Auto (String typ, double tachoStand)
7  {
8    public Auto(String typ) { this(typ, 0); }
9    public static void main(String[] argv)
10   {
11     Auto a1 = new Auto("PKW", 95423);
12     System.out.println(a1);                    // Auto[typ=PKW, tachoStand=95423.0]
13     System.out.println("Typ: " + a1.typ() + ", Tachstand: " + a1.tachoStand());
14     Auto a2 = new Auto("PKW");
15     System.out.println(a2);
16   }
17 }
```





## 5.12 Sealed Classes

Vesiegelte (sealed) Klassen können mit **`permits`** bestimmen, welche anderen Klassen von ihr erben dürfen.

Erbende Klassen müssen im gleichen Package oder Modul wie die Oberklasse sein und müssen entweder **`final`** oder **`sealed`** oder **`non-sealed`** sein.

**Objektorientierung/sealed/SealedBsp.java**

```
6  abstract sealed class Shape
7    permits Circle, Rectangle, Square, WeirdShape {}
8
9  final class Circle extends Shape {}
10
11 sealed class Rectangle extends Shape
12                           permits TransparentRectangle, FilledRectangle {}
13 final class TransparentRectangle extends Rectangle {}
14 final class FilledRectangle extends Rectangle {}
15
16 final class Square extends Shape {}
17
18 non-sealed class WeirdShape extends Shape {}
```



## 5.13 Zusammenfassende Fragen

- Welche Schlüsselwörter aus der Liste auf Seite 3-5 kennen wir schon und welche Bedeutung haben sie?
- Was sind die drei wichtigsten Konzepte des objektorientierten Softwareentwurfs?
- Was bedeutet Weitervererbung?
- Was bedeutet Mehrfachvererbung?
- Was ist der Unterschied zwischen Ergänzung und Überlagerung?
- Was ist der Unterschied zwischen erweiternder und einschränkender Konvertierung?
- Was bedeutet Binden?
- Was ist der Unterschied zwischen statischem und dynamischem Binden?
- Was sind Zugriffsrechte und wozu werden sie eingesetzt?
- Wie sind in Java die Zugriffsrechte geregelt?
- Was sind Packages? Wie viele Packages gehören zur aktuellen Java-Version?
- Was sind abstrakte Klassen?
- Was sind Interfaces?
- Welche Bedeutung haben die Klassen **`Object`** und **`Class`**?
- Was sind Wrapper-Klassen?



# 6 Ausnahmen

## Inhalt



**Inhalt/Ziele:** Das Konzept der Ausnahmebehandlung in Java stellt ein elegantes und mächtiges Instrument zum Umgang mit Laufzeitfehlern zur Verfügung. Wir werden in den weiteren Abschnitten diverse Packages des Java-API vorstellen. Da eine Vielzahl von Klassen die Behandlung von Ausnahmen erfordert, ist es wichtig, sich mit dem Konzept vertraut zu machen.

## 6.1 Was sind Ausnahmen?

- Eine **Ausnahme (exception)** ist ein Ereignis, das zur Laufzeit eines Programmes auftritt und den normalen Ablauf der Anweisungen unterbricht.
- Ausnahmen können beispielsweise auftreten bei
  - Rechenoperationen (**`ArithmeticException`**),
  - Zugriffen auf nicht vorhandene Feldelemente (**`ArrayIndexOutOfBoundsException`**),
  - Lese- und Schreiboperationen (**`IOException`**).
- Tritt in einer Java-Methode ein Fehler auf, so kann diese Methode ein Objekt der Klasse **`Exception`** (oder einer Subklasse) erzeugen und an das Laufzeitsystem (Interpreter) weitergeben. In Java wird dies als „werfen einer Ausnahme“ **(„throw an exception“)** bezeichnet.
- Wirft eine Methode eine Ausnahme, so tritt das Laufzeitsystem in Aktion, um diese Ausnahme zu behandeln. Es wird nach einem **exception handler** gesucht. Ein Exception Handler „fängt eine Ausnahme“ **(„catch an exception“)**.



- Die Sprache Java unterscheidet zwischen
  - **allgemeinen Ausnahmen** (z. B. fehlende Dateien, nicht verfügbare Rechner) und
  - **Laufzeitausnahmen** (das sind alle Nachfahren von **`RuntimeException`**, z. B. Division durch Null bei ganzzahligen Datentypen, Zugriff auf nicht vorhandene Feldelemente).
- Allgemeine Ausnahmen **müssen**, Laufzeitausnahmen **können** abgefangen werden.
- Wenn wir bei Laufzeitausnahmen keinen Exception Handler implementieren, bricht das Programm mit einer entsprechenden Fehlermeldung ab.

**Wozu braucht man diesen zweigeteilten Mechanismus eigentlich?**

Situation: Autor und Anwender einer Klasse sind nicht identisch.

- Der Autor einer Klasse kann Fehlersituationen feststellen, weiß aber nicht, wie der Benutzer seiner Klasse darauf reagieren möchte – er wirft eine Exception.
- Der Anwender weiß zwar, wie er auf Fehlersituationen reagieren möchte, kann aber keine Fehler feststellen, falls er nur class-Dateien und nicht den Java-Quellcode hat – aber er kann Exceptions fangen.



## 6.2 Behandlung einer Laufzeitausnahme

**`Exceptions/ExcBsp1.java`**

```
10    int[] a = new int[2];
11    a[0] = 1;
12    a[1] = 2;
13
14    System.out.println(a[2]);
15
16    System.out.println("Mich sieht man nur, wenn keine Ausnahmen auftraten.");
```

**Quelltextanalyse**

- Das obige Programm erzeugt eine Laufzeitausnahme:

```
wmai20 …/Java/Vorlesung/Quellen/Exceptions > java ExcBsp1
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 2 out of bounds for length 2
        at ExcBsp1.main(ExcBsp1.java:14)
```

Tritt in einem Java-Programm eine Ausnahme auf, so wird ein Objekt der Klasse **`Exception`** (oder einer erbenden Klasse) erzeugt und an das Laufzeitsystem (Interpreter) weitergeben. Wir bezeichnen dies als „werfen einer Ausnahme“ („throw an exception“).

In unserem Fall können wir die Ausnahme vom Laufzeitsystem behandeln lassen oder sie selbst abfangen („catch an exception“).



**`Exceptions/ExcBsp2.java`**

```
14    try
15      {
16        System.out.println(a[2]);
17        System.out.println("Mich sieht man nur, wenn keine Ausnahmen auftraten.");
18      }
19    catch (ArrayIndexOutOfBoundsException aioobe)
20      {
21        System.out.println("Habe gefangen: " + aioobe);
22        System.out.println("Die Message lautet: " + aioobe.getMessage());
23        aioobe.printStackTrace();
24      }
25
26    System.out.println("Sieht man mich?");
```

**Quelltextanalyse**

- Wir verwenden hier die **`try`**-**`catch`**-Kontrollstruktur.
- Die Anweisungen, die eine Ausnahme auslösen können, befinden sich innerhalb eines **`try`**-Blockes.
- Tritt zur Laufzeit eine Ausnahme auf, so springt der Interpreter an das Ende dieses **`try`**-Blockes.
- Im Anschluss daran gibt es einen oder mehrere **`catch`**-Blöcke. Jeder dieser Blöcke behandelt eine bestimmte Ausnahme.
- In unserem Fall erhalten wir eine Instanz der Klasse **`ArrayIndexOutOfBoundsException`**. Diese enthält diverse Zusatzinformationen.

- Sind mehrere **catch**-Blöcke vorhanden, wird in den ersten Block gesprungen, der Exceptions der geworfenen Klasse **oder einer Oberklasse** fängt.
- Deswegen sollten immer zuerst spezielle und danach allgemeinere Exceptions gefangen werden.

**Exceptions/ExcBsp3.java**

```
25    finally
26      {
27        System.out.println("Mich sieht man. Immer.");
28      }
```

**Quelltextanalyse**

- An die **catch**-Blöcke kann man noch einen **finally**-Block anschließen.
- Dort kann man Anweisungen eintragen, die immer (egal, welche Ausnahme auftrat, auch wenn gar keine auftrat) ausgeführt werden sollen. Die Anweisungen werden sogar dann ausgeführt, wenn im **try**- oder **catch**-Block eine **return**-Anweisung steht!
- Sinnvoll ist das für „Aufräumarbeiten“ wie das Schließen von Dateien oder Netzwerkverbindungen, die auf jeden Fall zu erledigen sind.



Allgemeine Syntax der **try-catch-finally**-Struktur:

```
try
  {
    Anweisungen
  }
catch (AException a)
  {
    Anweisungen zum Behandeln von AException a
  }
catch (BException b)
  {
    Anweisungen zum Behandeln von BException b
  }
finally
  {
    von Ausnahmen unabhängige Anweisungen
  }
```

Seit Java 7 kann ein **catch**-Block verschiedene Exceptions fangen (Multi-Catch):

```
try
  {
    Anweisungen
  }
catch (AException | BException e)
  {
    Anweisungen zum Behandeln von AException oder BException
  }
```



## 6.3 Mehr über den try-Block

- Tritt innerhalb eines **try**-Blockes eine Ausnahme auf, so werden die nachfolgenden Anweisungen übergangen.
- Dies hat eventuell Auswirkungen auf das Initialisieren von Variablen.
- Welche Möglichkeiten gibt es, dieses Problem zu umgehen?

**Exceptions/ExcBsp4.java**

```
13    int b, c;
15    try
16      {
17        b = 5;
20      }
32    //c = b;   // FEHLER: variable b might not have been initialized
```

**einfache Lösung:** **b** vor dem **try**-Block (z. B. mit 0) initialisieren

**besser:** Man sollte überlegen, was logisch mit der Fehlersituation zusammenhängt und die Größe des **try**-Blocks passend wählen.

Vielleicht ist die Zuweisung **c = b** im Fehlerfall ja unnötig und könnte in den **try**-Block geschoben werden?



## 6.4 Eine Ausnahme werfen

Konstruktoren und Methoden können Ausnahmen werfen.

Sie zeigen dies durch eine Klausel wie **throws Exception** an.

Diese Klausel wird auch vom Dokumentationsgenerator **javadoc** registriert, um dem Benutzer einer Klasse anzuzeigen, dass er ggf. eine **try-catch**-Umgebung aufbauen muss.

Wir demonstrieren dies am Beispiel **Sparbuch**. Es wird jedes mal, wenn die Zusicherung $k \geq 0$ in Gefahr ist, eine Ausnahme geworfen.

**Exceptions/Geld/3/Sparbuch.java**

```
18    public Sparbuch(float k, float p, int l) throws Exception
19    {
20      if (k < 0 || p < 0 || l < 0)
21        throw new Exception("Fehler beim Erzeugen des Sparbuchs.");
28    public float abheben(float s) throws Exception
29    {
30      if (k < s)
31        throw new Exception("Fehler beim Abheben.");
39    try
40      {
41        Sparbuch sb = new Sparbuch(100, 0.01f, 3);
42        sb.abheben(200);
43      }
44    catch (Exception e)
45      {
46        System.out.println(e.getMessage());
47      }
```