Vererbung

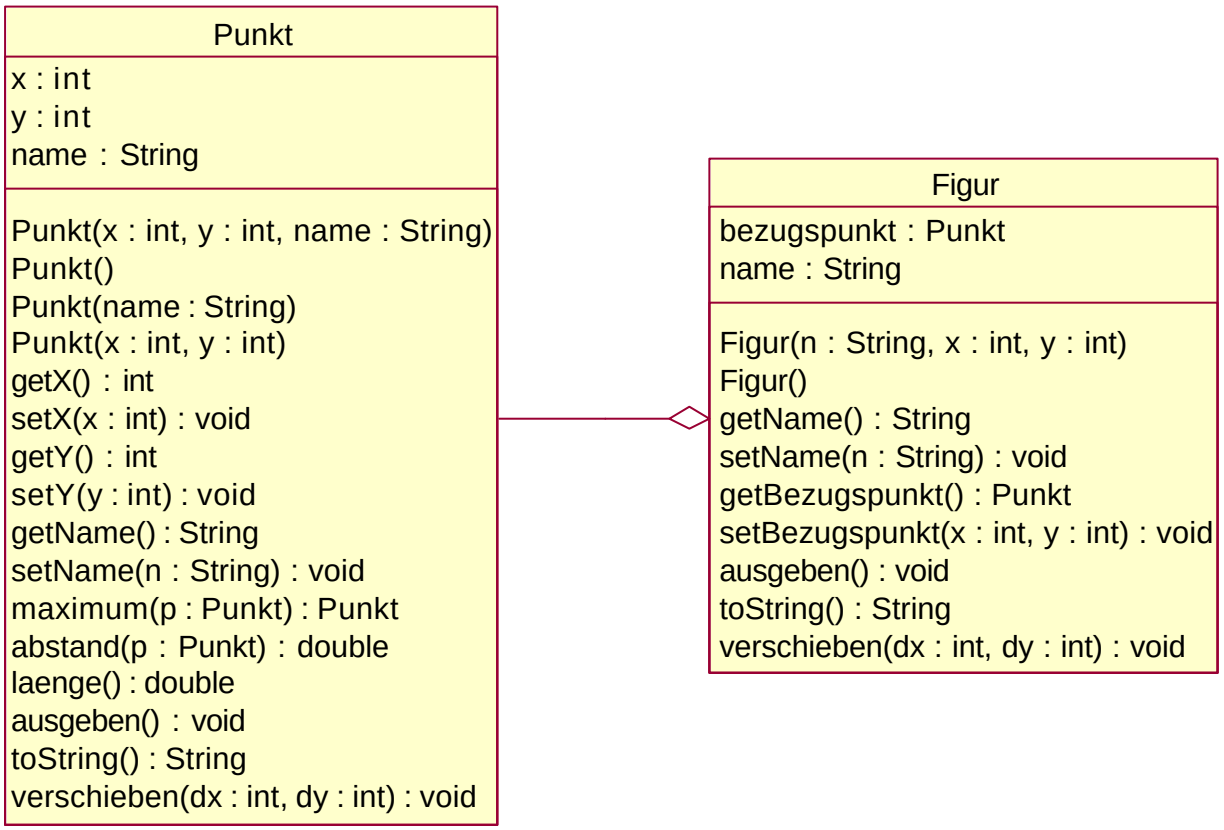
- Komposition von Klassen
- Erweitern von Klassen
- Das Schlüsselwort `super`
- Zuweisungskompatibilität
- Abstrakte Klassen
- Zugriffsrechte
- Der Modifizierer `final`
- Schnittstellen
- Die Klasse `Object`
- Wrapper-Klassen

Vererbung

Komposition von Klassen ("hat ein")

Beispiel: Die Klasse `Figur`

Die Klasse `Figur` *hat einen* `Punkt`, der die Rolle des Bezugspunktes der Figur spielt.



Vererbung

Komposition von Klassen ("hat ein")

Beispiel: Die Klasse `Figur`

```
class Figur {
   Punkt bezugspunkt;
   String name;

   Figur(String n, int x, int y) {
      bezugspunkt = new Punkt(x, y);
      name = n;
   }

   Figur()
   {  this("nichts", 0, 0);  }

   String getName()
   {  return name; }

   void setName(String n)
   {  name = n; }

   Punkt getBezugspunkt()
   {  return bezugspunkt; }

   void setBezugspunkt(int x, int y) {
      bezugspunkt.x = x;
      bezugspunkt.y = y;
   }
```

Vererbung

Komposition von Klassen ("hat ein")

Beispiel: Die Klasse `Figur` (Fortsetzung)

```
    void ausgeben()
    {
       System.out.print(
          "Figur: " + name + ", Bezugspunkt: ");
       bezugspunkt.ausgeben();
    }

    public String toString()
    {
       return "Figur: " + name +
          ", Bezugspunkt: " + bezugspunkt;
    }

    void verschieben(int dx, int dy)
    {  bezugspunkt.verschieben(dx, dy);  }

}
```

Vererbung

Erweitern von Klassen ("ist ein")

Wir definieren zwei konkrete geometrische Figuren: Kreis und Rechteck

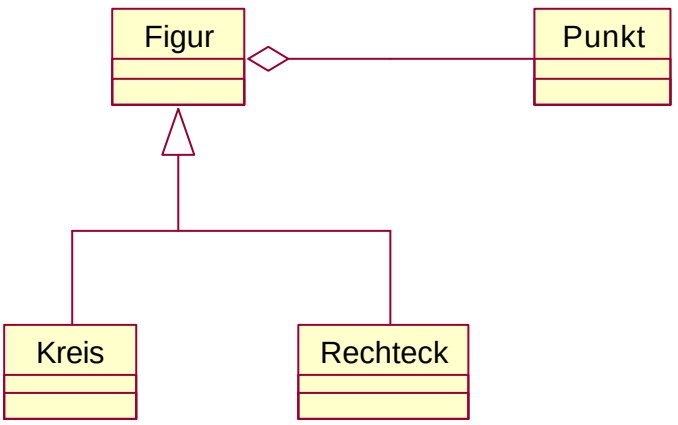
Ein Kreis *ist eine* Figur, und ein Rechteck *ist eine* Figur.

Die Klasse `Figur` wird **erweitert** zur Klasse `Kreis` bzw. `Rechteck`.

Diese Erweiterung wird **Vererbung** genannt.

Die ursprüngliche Klasse (`Figur`) heißt **Superklasse** oder Basisklasse, die abgeleiteten Klassen (`Kreis` und `Rechteck`) nennt man **Subklassen**.

Beispiel: Die Klassen `Kreis` und `Rechteck`



Vererbung

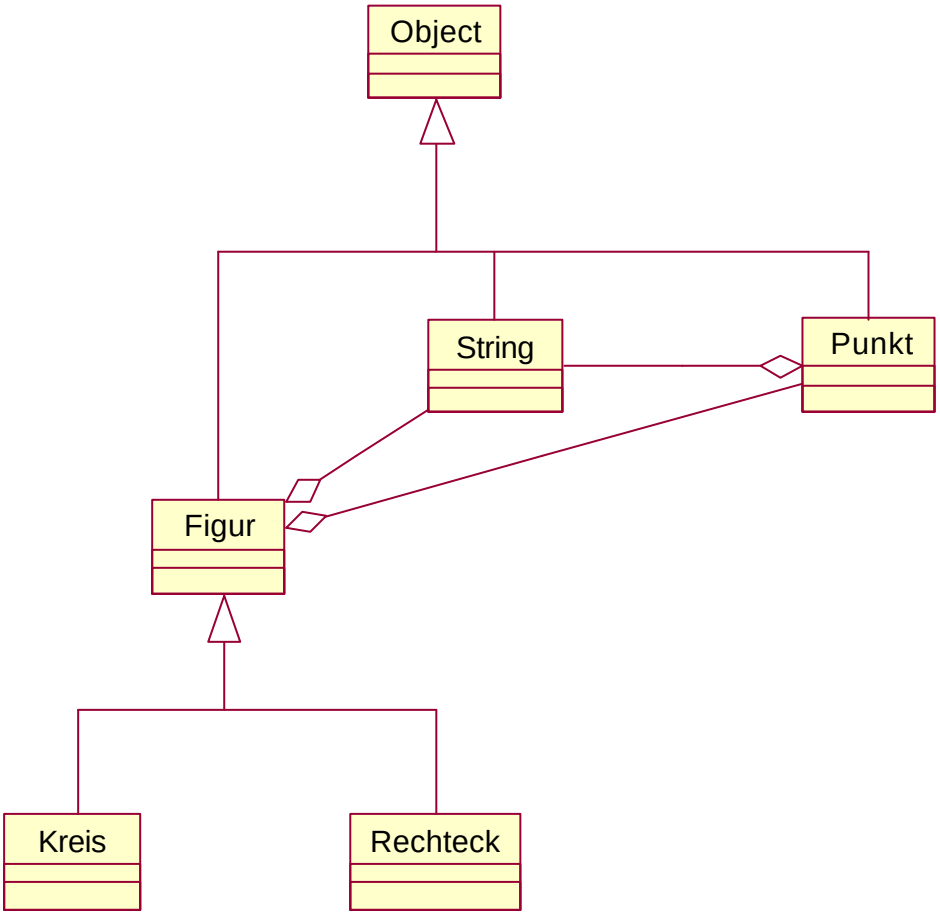
Erweitern von Klassen ("ist ein")

Jede Klasse hat höchstens *eine* Superklasse.

Jede Klasse, die nicht explizit von einer Superklasse erbt, erbt von der in Java definierten Klasse **`Object`**.

`Object` ist die einzige Klasse in Java, die keine Superklasse besitzt.

Beispiel: Die Figuren-Klassen



Vererbung

Erweitern von Klassen ("ist ein")

Beispiel: Die Klasse `Kreis`

```
class Kreis extends Figur
{
   double radius;

   Kreis(String n, int x, int y, double r)
   {
      name = n;
      bezugspunkt = new Punkt();
      bezugspunkt.x = x ;
      bezugspunkt.y = y;
      radius = r;
   }

   double getRadius()
   {  return radius; }

   void setRadius(double r)
   {  radius = r; }

   double flaeche()
   {  return Math.PI * radius * radius; }

   double umfang()
   {  return 2 * Math.PI * radius; }
```

Vererbung

Erweitern von Klassen ("ist ein")

Beispiel: Die Klasse `Kreis` (Fortsetzung)

```
    void ausgeben()
    {
       System.out.println("Radius = " + radius);
    }

    public String toString()
    {
       return "Radius = " + radius;
    }

}
```

Die Methoden `ausgeben()` und `toString()` der Klasse `Figur` werden in der Klasse `Kreis` neu deklariert, sie werden *überschrieben*.

Überschreiben einer Methode bedeutet, die Implementierung einer Methode der Superklasse in der Subklasse zu ersetzen. Die Signaturen *und* die Rückgabetypen müssen identisch sein.

Überladen einer Methode bedeutet, dass mehrere Methoden mit demselben Namen bereitgestellt werden, die zu ihrer Unterscheidung unterschiedliche Signaturen besitzen.

Vererbung

Erweitern von Klassen ("ist ein")

Beispiel: Überschreiben von Methoden

```
class FigurenTest
{
   public static void main(String[] argv)
   {
      Figur f = new Figur("Figur", 1, 2);
      f.ausgeben();
      System.out.println();

      Kreis k = new Kreis("Kreis", 1, 2, 3);
      k.ausgeben();
      System.out.println();

      Rechteck r = new Rechteck("Rechteck", 1,2,3,4);
      r.ausgeben();
      System.out.println();
   }
}
```

Ausgabe des Programms:

```
Figur: Figur, Bezugspunkt: Pkt(1,2)

Radius = 3.0

Laenge = 3.0, Breite = 4.0
```

Vererbung

Das Schlüsselwort `super`

Aufruf von Methoden und Konstruktoren der Superklasse

Beispiel: Die Klasse `Kreis`

```
class Kreis extends Figur
{
   double radius;

   Kreis(String n, int x, int y, double r)
   {
      super(n, x, y);
      radius = r;
   }

   ...

   double flaeche()
   {  return Math.PI * radius * radius; }

   double umfang()
   {  return 2 * Math.PI * radius; }

   void ausgeben()
   {
      super.ausgeben();
      System.out.println("Radius = " + radius);
   }

   public String toString()
   { return super.toString() + "Radius = " + radius; }
}
```

Vererbung

Das Schlüsselwort `super`

Wenn ein *Attribut* einer Subklasse den gleichen Namen hat wie ein Attribut der Superklasse, dann **verdeckt** das Attribut der Subklasse das Attribut der Superklasse.

Zugriff auf verdeckte Attribute der Superklasse mit `super` oder mit expliziter Typanpassung (cast).

Beispiel: Verdecken von Attributen

```
class A { int x = 5; }

class B extends A { char x = 'a'; }

class C extends B {
  double x = 47.11;

  void test()
  {
    System.out.println(this.x);     // x in C: 47.11
    System.out.println(x);          // x in C: 47.11

    System.out.println(super.x);        // x in B: a
    System.out.println(((B)this).x);    // x in B: a

//  System.out.println(super.super.x);  // verboten
    System.out.println(((A)this).x);    // x in A: 5
  }
}
```

Vererbung

Das Schlüsselwort super

Beispiel: Methoden überschreiben, Attribute verdecken

```
class A
{
  int i = 5;
  int f() { return i; }
}

class B extends A
{
  int i = 77;            // verdeckt i in Klasse A
  int f() { return -i; } // überschreibt f in Klasse A
}

class VerdeckenTest2
{
  public static void main(String[] args)
  {
    B b = new B();
    System.out.println(b.i);     // B.i: 77
    System.out.println(b.f());   // B.f(): -77

    // b in ein Objekt der Klasse A umwandeln
    A a = (A) b;
    // a verweist auf das gleiche Objekt wie b !!!
    System.out.println(a.i);
    // verweist nun auf A.i; gibt 5 aus
    System.out.println(a.f());
    // verweist immer noch auf B.f(); gibt -77 aus
  }
}
```

Vererbung

Das Schlüsselwort `super`

Konstruktoren in Subklassen

Aufruf des Superklassenkonstruktors im Rumpf des Subklassenkonstruktors entweder

- explizit *als erste Anweisung* :

```
super(arg1, arg2);
```

- oder implizit: Falls die erste Anweisung kein `super`- oder `this`-Aufruf ist, wird der `super`-Aufruf ohne Argumente automatisch eingefügt. Falls die Superklasse keinen Konstruktor ohne Argumente hat, führt dies zu einem Übersetzungsfehler.

Vom Compiler erzeugter Standard-Konstruktor:

```
public Beispielklasse()
{
  super();
}
```

Vererbung

Das Schlüsselwort `super`

Konstruktoren in Subklassen

Beim Erzeugen eines Objekts werden seine Attribute initialisiert. Die Reihenfolge der Initialisierung ist:

1. Alle Attribute werden auf voreingestellte Anfangswerte für ihren jeweiligen Typ gesetzt. (Null für alle numerischen Typen, `'\u0000'` für `char`, `false` für `boolean` und `null` für Objektreferenzen)

2. Aufruf des Konstruktors der Basisklasse

3. Initialisierung der Attribute durch die angegebenen Initialisierungsausdrücke

4. Ausführung des Konstruktorrumpfs

Konstruktoren werden automatisch verkettet, Finalisierer jedoch nicht. Der Finalisierer der Subklasse muss den Finalisierer der Superklasse explizit aufrufen, sonst wird der Finalisierer der Superklasse nicht ausgeführt:

```
super.finalize();
```

Vererbung

Zuweisungskompatibilität

Einer Variablen vom Typ einer Klasse K kann man jede Variable vom Typ einer Subklasse von K zuweisen.

Umgekehrt geht das nicht.

Beispiel: Zuweisung von Referenzvariablen

```
class FigurenTest1
{
   public static void main(String[] argv)
   {
      Punkt p = new Punkt(0, 0, "Punkt");
      Figur f = new Figur("Figur", 1, 2);
      Kreis k = new Kreis("Kreis", 1, 2, 3);
      Rechteck r = new Rechteck("Rechteck",1,2,3,4);

//      p = f;        // Zuweisung nicht erlaubt

//      k = f;        // Zuweisung nicht erlaubt
//      r = f;        // Zuweisung nicht erlaubt

      f = k;
      f.ausgeben();

      f = r;
      f.ausgeben();
   }
}
```

Vererbung

Zuweisungskompatibilität

Beispiel: Zuweisung von Referenzvariablen

```
class FigurenTest2
{
   public static void main(String[] argv)
   {
      Figur f1, f2;
      f1 = new Kreis("Kreis 1", 10, 20, 8);
      f1.ausgeben();
      f1.verschieben(3, 3);
      f1.ausgeben();
      System.out.println();

      f2 = new Rechteck("Rechteck", 1, 2, 3, 4);
      f2.ausgeben();
      System.out.println();

      f1 = new Kreis("Kreis 2", 40, 40, 10);
      f1.ausgeben();

      f2 = f1;
      f2.ausgeben();
   }
}
```

Die Auswahl der aufzurufenden Methode ist bestimmt durch den tatsächlichen Typ des Objekts.

Vererbung

Zuweisungskompatibilität

Der Operator `instanceof`

Der Ausdruck (`o instanceof K`) liefert den Wert `true`, wenn das Objekt o ein Objekt der Klasse K oder einer von K abgeleiteten Klasse ist. Andernfalls wird `false` zurückgegeben.

Ist die linke Seite `null`, wird `false` zurückgeliefert.

Beispiel: `instanceof`

```
class InstanceOf
{
   public static void zeigeTypVon(Figur f)
   {
      if (f instanceof Figur)
         System.out.println("Typ: Figur");
      else if (f instanceof Kreis)
         System.out.println("Typ: Kreis");
      else if (f instanceof Rechteck)
         System.out.println("Typ: Rechteck");
      else
         System.out.println("Typ: unbekannt");
   }
```

Vererbung

Zuweisungskompatibilität

Beispiel: `instanceof` (Fortsetzung)

```
   public static void main(String[] argv) {
      Figur f1, f2 = null;
      zeigeTypVon(f2);

      f2 = new Figur();
      zeigeTypVon(f2);

      f1 = new Kreis("Kreis", 10, 20, 8);
      f2 = new Rechteck("Rechteck", 1, 2, 3, 4);
      zeigeTypVon(f2);

      f2 = f1;
      zeigeTypVon(f2);
   }
}
```

Ausgabe des Programms:

```
Typ: unbekannt
Typ: Figur
Typ: Figur
Typ: Figur
```

Vererbung

Zuweisungskompatibilität

Die Reihenfolge der Typabfrage mit `instanceof` muss in der Klassenhierarchie von den untersten Subklassen in Richtung Superklasse erfolgen.

Beispiel: `instanceof` (Fortsetzung)

```
public static void zeigeTypVon(Figur f)
{
   if (f instanceof Kreis)
      System.out.println("Typ: Kreis");
   else if (f instanceof Rechteck)
      System.out.println("Typ: Rechteck");
   else if (f instanceof Figur)
      System.out.println("Typ: Figur");
   else
      System.out.println("Typ: unbekannt");
}
```

Die Ausgabe des Programms lautet jetzt:

```
Typ: unbekannt
Typ: Figur
Typ: Rechteck
Typ: Kreis
```

Vererbung

Abstrakte Klassen

Beispiel:

```
class FigurenTest3
{
   public static void main(String[] argv)
   {
      Figur f1;
      f1 = new Kreis("Kreis 1", 10, 20, 8);
      f1.ausgeben();
      System.out.println("Flaeche: " +
         f1.flaeche());
   }
}
```

Fehlermeldung des Compilers:

```
Method flaeche() not found in class Figur.
```

Die Methode `flaeche()` ist in der Klasse `Figur` nicht deklariert. Sie wird erst in den Klassen `Kreis` und `Rechteck` eingeführt.

Die Klasse `Figur` ist noch nicht so konkret, dass wir die Methode `flaeche()` implementieren können.
Die Klasse `Figur` ist *abstrakt*.

Vererbung

Abstrakte Klassen

```
abstract class Figur
{
   Punkt bezugspunkt;
   String name;

   Figur(String n, int x, int y)
   {
      bezugspunkt = new Punkt(x, y);
      name = n;
   }

   Figur()
   {  this("nichts", 0, 0);  }

   ...

   /**
   * abstrakte Methoden,
   * die erst in den Unterklassen
   * definiert werden koennen
   */
   abstract double flaeche();
   abstract double umfang();

}
```

Vererbung

Abstrakte Klassen

Beispiel:

```
class FigurenTest
{
   public static void main(String[] argv)
   {
      Figur f1, f2;
      f1 = new Kreis("Kreis", 10, 20, 8);
      f1.ausgeben();
      System.out.println("Flaeche: " + f1.flaeche());
      System.out.println();

      f2 = new Rechteck("Rechteck", 1, 2, 3, 4);
      f2.ausgeben();
      System.out.println("Flaeche: " + f2.flaeche());
      System.out.println();
   }
}
```

Ausgabe des Programms:

```
Figur: Kreis, Bezugspunkt: Pkt(10,20)
Radius = 8.0
Flaeche: 201.06192982974676

Figur: Rechteck, Bezugspunkt: Pkt(1,2)
Laenge = 3.0, Breite = 4.0
Flaeche: 12.0
```

Vererbung

Zugriffsrechte

Datenkapselung

Verbergen von Daten innerhalb einer Klasse
Zugriff auf die Daten nur über Methoden

Die internen Details der Implementierung einer Klasse werden versteckt.

In Java gibt es vier Stufen von Zugriffsrechten, welche den Zugriff auf Methoden und Attributen einer Klasse regeln. Sie sind gekennzeichnet durch die Schlüsselwörter

- `public`
- `private`
- `protected`

bzw. durch *kein* Schlüsselwort.

Vererbung

Zugriffsrechte

Der Modifizierer `public`

Wird eine Klasse mit `public` gekennzeichnet, dann heißt das, dass die Klasse überall benutzt werden kann.

Eine als `public` vereinbarte Methode kann von jeder anderen Klasse verwendet werden.

Auf ein `public`-Attribut hat man von überall her lesenden und schreibenden Zugriff.

Kein Modifizierer

Klassen, Attribute und Methoden ohne einen der Modifizierer `public`, `protected` oder `private` sind in jeder Klasse sichtbar, die im gleichen Modul definiert ist.

Der Modifizierer `protected`

Auf eine mit `protected` gekennzeichnete Komponente (Attribut, Methode oder Konstruktor) können alle Klassen desselben Moduls und alle Subklassen zugreifen. Klassen können nicht als `protected` deklariert werden.

Vererbung

Zugriffsrechte

Der Modifizierer `private`

Attribute und Methoden, die mit `private` gekennzeichnet sind, können nur in Methoden dieser Klasse benutzt werden. Klassen können nicht als `private` deklariert werden.

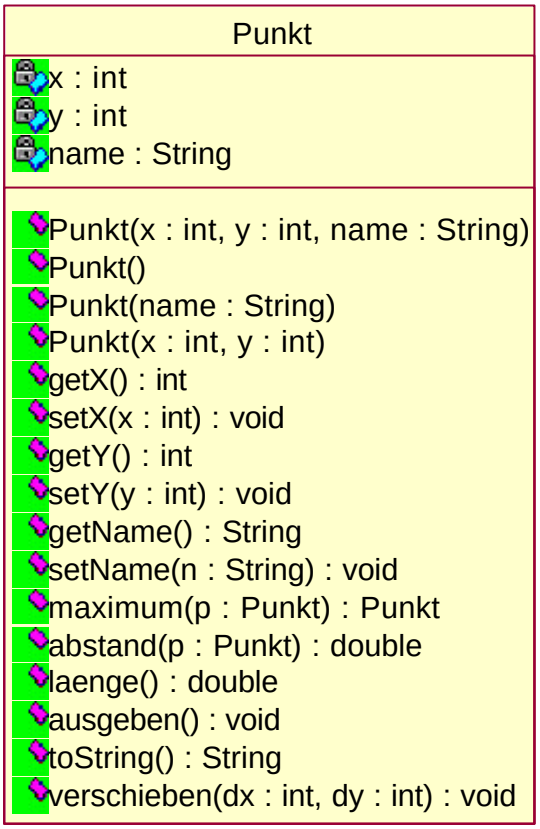
Beispiel: Zugriff auf private Komponenten eines Objekts

```
public class Punkt
{
   private int x, y;
   private String name;

   public Punkt(int x, int y, String name)
   {
      this.x = x;
      this.y = y;
      this.name = name;
   }

   ...

   public double abstand(Punkt p)
   {
      return Math.sqrt((x-p.x)*(x-p.x) +
                       (y-p.y)*(y-p.y));
   }
}
```

Vererbung

Zugriffsrechte

Darstellung im UML-Klassendiagramm



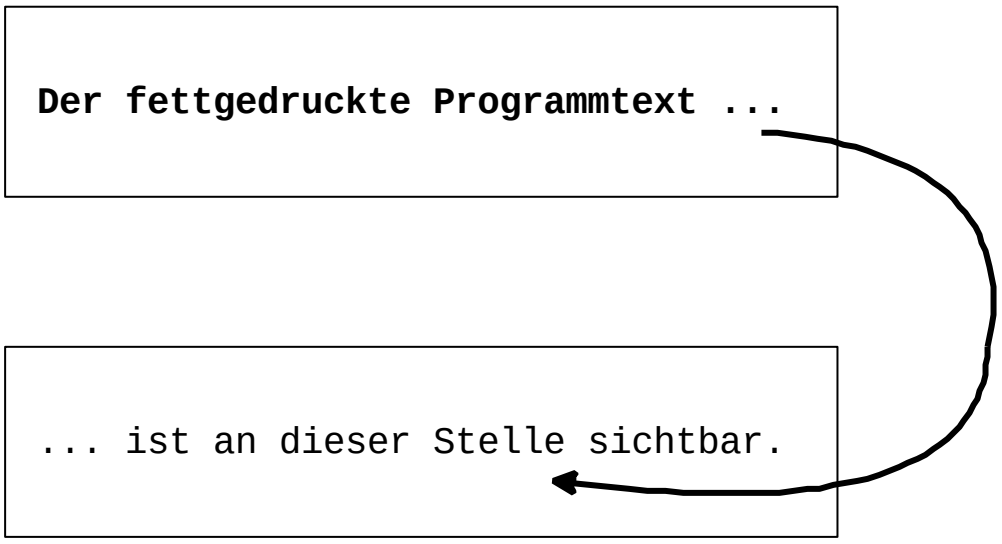
Vererbung

Zugriffsrechte

Zusammenfassung

Verfügbar für:	Sichtbarkeit			
	public	protected	package	private
Gleiche Klasse	ja	ja	ja	ja
Klasse im gleichen Paket	ja	ja	ja	nein
Subklasse in anderem Paket	ja	ja	nein	nein
Nicht-Subklasse, anderes Paket	ja	nein	nein	nein

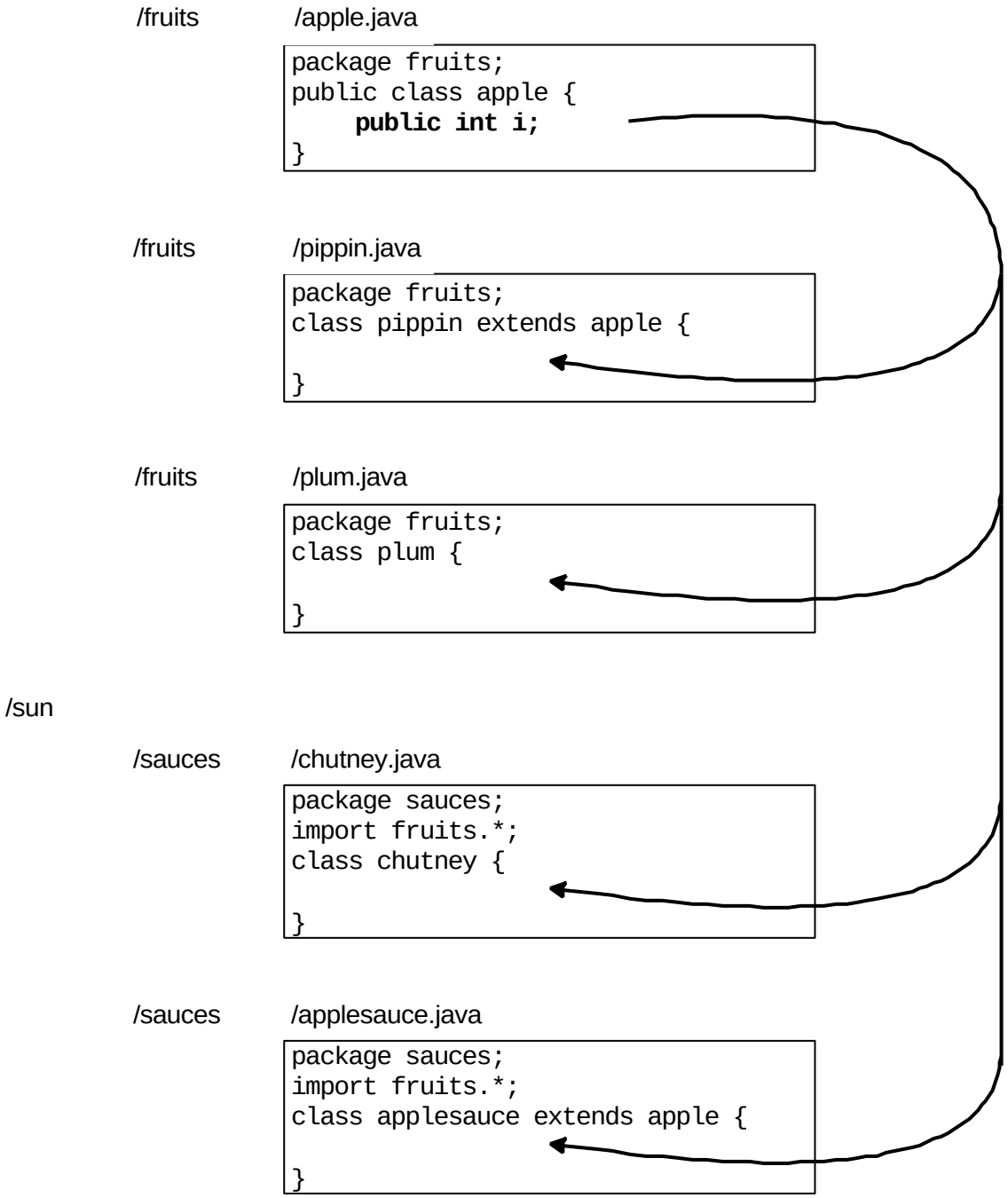
Legende für die folgenden vier Diagramme:



Vererbung

Zugriffsrechte

1. `public` - überall sichtbar



Vererbung

Zugriffsrechte

2. `protected` - zugänglich im selben Package und in beliebigen Unterklassen

/fruits /apple.java

```
package fruits;
public class apple {
     protected int i;
}
```

/fruits /pippin.java

```
package fruits;
class pippin extends apple {

}
```

/fruits /plum.java

```
package fruits;
class plum {

}
```

/sun

/sauces /chutney.java

```
package sauces;
import fruits.*;
class chutney {

}
```

/sauces /applesauce.java

```
package sauces;
import fruits.*;
class applesauce extends apple {

}
```

Vererbung

Zugriffsrechte

3. Voreinstellung (package) - nur sichtbar in diesem Package

/fruits /apple.java

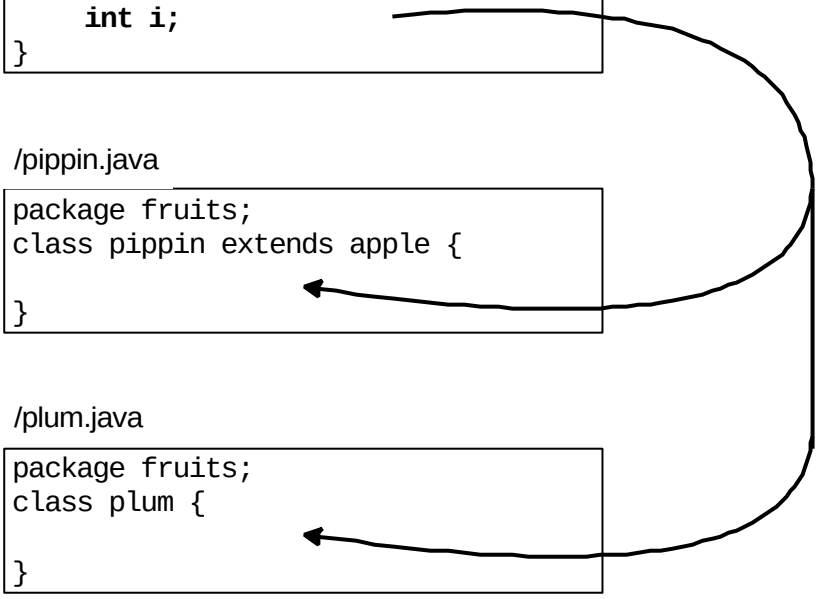
```
package fruits;
public class apple {
    int i;
}
```

/fruits /pippin.java

```
package fruits;
class pippin extends apple {

}
```

/fruits /plum.java

```
package fruits;
class plum {

}
```



/sun

/sauces /chutney.java

```
package sauces;
import fruits.*;
class chutney {

}
```

/sauces /applesauce.java

```
package sauces;
import fruits.*;
class applesauce extends apple {

}
```

Vererbung

Zugriffsrechte

4. `private` - außerhalb dieser Klasse nicht zugänglich

/fruits /apple.java

```
package fruits;
public class apple {
    private int i;
}
```

/fruits /pippin.java

```
package fruits;
class pippin extends apple {

}
```

/fruits /plum.java

```
package fruits;
class plum {

}
```

/sun

/sauces /chutney.java

```
package sauces;
import fruits.*;
class chutney {

}
```

/sauces /applesauce.java

```
package sauces;
import fruits.*;
class applesauce extends apple {

}
```

Vererbung

Der Modifizierer `final`

Der Wert eines mit dem Modifizierer `final` gekennzeichneten Attributs oder Parameters kann nicht mehr verändert werden.

Eine als `final` deklarierte Methode kann in Subklassen nicht mehr überschrieben werden. Sie wird unveränderbar an die Subklasse weitervererbt, falls sie nicht `private` ist.

Von einer als `final` gekennzeichneten Klasse können keine Subklassen abgeleitet werden.

Beispiel: `final`

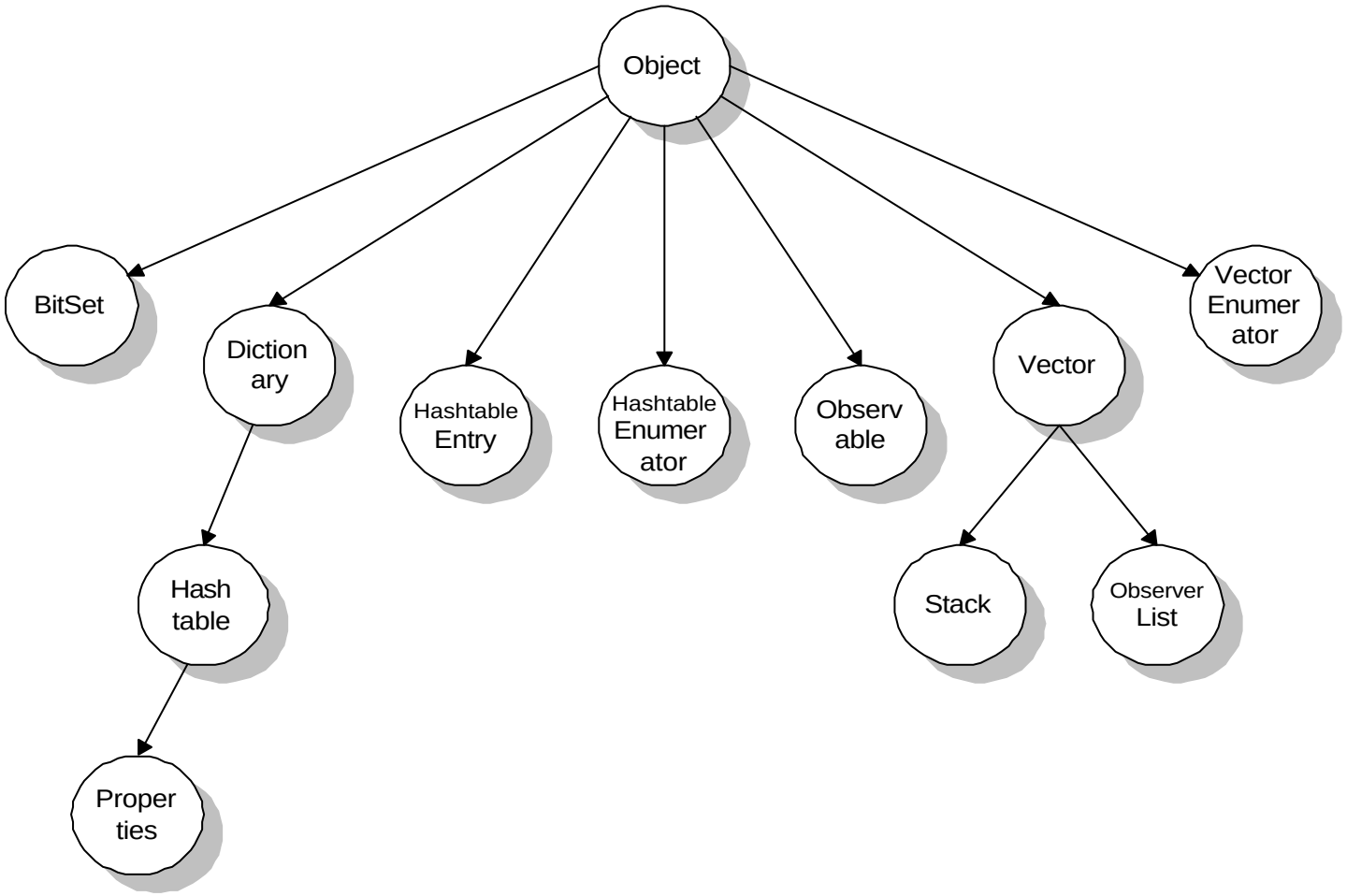
```
final class FinalTest
{
  final static int KONSTANTE = 17;

  final int methode(final int i)
  {
    // KONSTANTE *= i;   // nicht erlaubt
    // i = 38;           // nicht erlaubt
    return i + KONSTANTE + 4;
  }
}
```

Vererbung

Schnittstellen

Java kennt keine Mehrfachvererbung.
Eine abgeleitete Klasse besitzt genau eine Basisklasse.

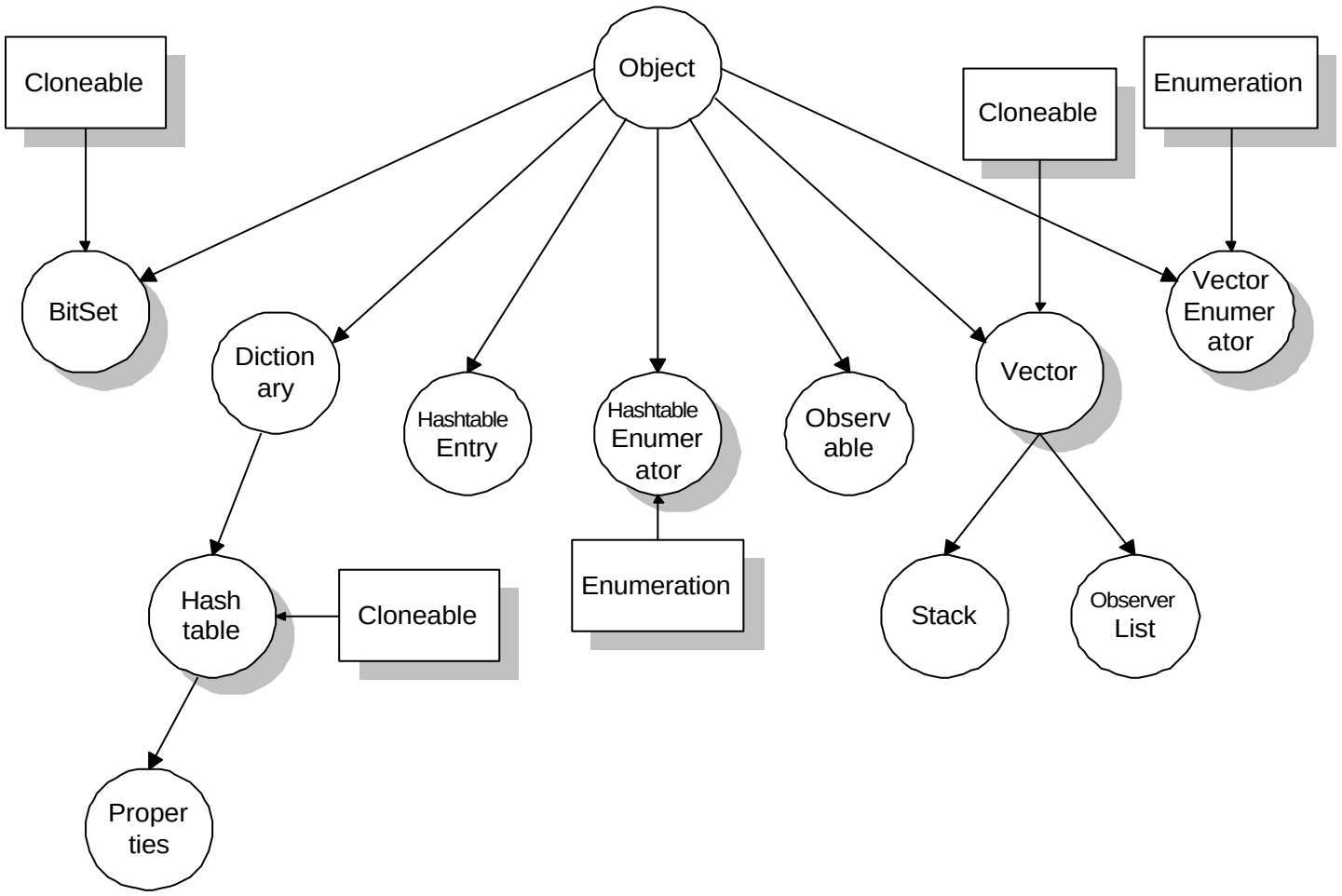
Beispiel einer Klassenhierarchie:



Vererbung

Schnittstellen

Eine Schnittstelle (engl. *interface*) dient der Spezifikation von Methoden, die eine Klasse zu implementieren verspricht.



Vererbung

Schnittstellen

Durch eine Schnittstellen-Deklaration legt man fest, **was** man mit dieser Schnittstelle machen kann; dabei ist noch völlig offen, **wie** dies realisiert wird.

Eine Schnittstelle in Java ähnelt stark einer Klasse, die ausschließlich abstrakte Methoden enthält.

Außer abstrakten Methoden darf eine Schnittstelle noch Konstanten enthalten.

Alle in einer Schnittstelle deklarierten Methoden sind implizit `abstract` `public`; alle Attribute sind implizit `final` `public` `static`.

Die Schnittstelle selbst ist implizit abstrakt; ihre Sichtbarkeit ist entweder `public` oder Standard (*package*).

Vererbung

Schnittstellen

Schnittstellen `vergleichbar`, `englisch`, `deutsch`

```
interface vergleichbar {
   /**
   *  Vergleich auf Gleichheit.
   *  @params o zu vergleichendes Objekt
   *  @return true, falls gleich; false sonst
   */
   boolean gleich(Object o);

   /**
   *  Vergleich
   *  @params o zu vergleichendes Objekt
   *  @return 0, falls gleich;
   *          <0, falls kleiner;
   *          >0, falls groesser
   */
   int vergleiche(Object o);
}

interface englisch {
   String GLEICH =  "equals ",
          KLEINER = "less than ",
          NICHT =   "not ";
}

interface deutsch {
   String GLEICH =  "gleich ",
          KLEINER = "kleiner als ",
          NICHT =   "nicht ";
}
```

Vererbung

Schnittstellen

Die abstrakten Methoden einer Schnittstelle werden von einer Klasse *implementiert*, indem man im Klassenkopf das Schlüsselwort `implements` gefolgt vom Schnittstellennamen angibt.

Jede Klasse, die eine Schnittstelle implementiert, *muss alle* Methoden der Schnittstelle implementieren. Eine Klasse kann mehrere Schnittstellen implementieren.

Schnittstellen sind auch Typen (wie Klassen).

Es kann keine Objekte dieser Schnittstellen-Typen geben. Aber Variablen oder Argumente können als Typ einen Schnittstellen-Typ haben; das referenzierte Objekt muss von einer Klasse sein, die diese Schnittstelle implementiert.

Vererbung

Schnittstellen

Beispiel: Punktvergleich

```
class Punkte extends Punkt
   implements vergleichbar, englisch
{
   public Punkte()
   {  }

   public Punkte(int x, int y)
   {  super(x, y, "");  }

   public boolean gleich(Object o)
   {
      Punkte p = (Punkte)o;
      return ((x == p.x) && (y == p.y));
   }

   public int vergleiche(Object o)
   {
      Punkte p = (Punkte)o;
      double lg, lgp;
      // Länge dieses Punktes
      lg = laenge();
      lgp = p.laenge();
      return lg==lgp ? 0 : lg<lgp ? -1 : 1;
   }
```

Vererbung

Schnittstellen

Beispiel: Punktvergleich (Fortsetzung)

```
    void IstGleich(Punkte p)
    {
       ausgeben();
       if (!gleich(p))
          System.out.print("un");
       System.out.print(GLEICH);
       p.ausgeben();
    }

    void test()
    {
       Punkte p1 = new Punkte(2, 4),
              p2 = new Punkte(3, 2),
              p3 = new Punkte(3, 2);
       p1.IstGleich(p2);
       System.out.println();
       p2.IstGleich(p3);
       System.out.println();
       p1.ausgeben();
       if (p1.vergleiche(p2) >= 0)
          System.out.print(NICHT);
       System.out.print(KLEINER);
       p2.ausgeben();
       System.out.println();
    }
}
```

Vererbung

Schnittstellen

Beispiel: Punktvergleich (Fortsetzung)

```
public class PunkteTest
{
   public static void main(String[] args)
   {
      Punkte p = new Punkte();
      p.test();
   }
}
```

Ausgabe des Programms:

```
(2,4)
unequals (3,2)

(3,2)
equals (3,2)

(2,4)
not less than (3,2)
```

Vererbung

Schnittstellen

Wie bei Klassen können wir eine Schnittstelle erweitern und so eine Sub-Schnittstelle definieren.

Im Gegensatz zu Klassen ist bei Schnittstellen Mehrfachvererbung möglich. Damit kann man in einer Schnittstelle mehrere andere Schnittstellen kombinieren.

Beispiel: Vererbung bei Schnittstellen

```
interface englischVergleichbar
   extends vergleichbar, englisch
{
   String version = "englischVergleichbar (V 1.0)";
}
```

Vererbung

Die Klasse `Object`

Basisklasse für alle Klassen eines Java-Programms ist die Klasse `Object`. Alle Klassen werden direkt oder indirekt von ihr abgeleitet, weshalb alle `public`- und `protected`-Elemente der Klasse `Object` in jeder Klasse verfügbar sind.

Einige der so verfügbaren Methoden werden in vielen Klassen überschrieben, so dass die Standarddefinition verdeckt wird.

`protected Object clone()`

- erzeugt eine byteweise Kopie des aktuellen Objekts.

`public boolean equals(Object obj)`

- prüft, ob zwei Objekte Byte für Byte übereinstimmen.

`public final Class getClass()`

- liefert Informationen über Klasse, Basisklasse und die Namen der Methoden.

Die Klasse `Object`

`public int hashCode()`

- gibt einen Hashcode für das aktuelle Objekt zurück. Dieser Wert wird bei der Speicherung von Objekten in Hashtabellen verwendet.

`public String toString()`

- liefert eine Darstellung des aktuellen Objekts als Zeichenkette.

`protected void finalize()`

- wird aufgerufen, bevor ein Objekt vom Garbage Collector entfernt wird.

Wrapper-Klassen

Die Klassen `Boolean`, `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float` und `Double` sind sogenannte Hüllenklassen (engl. *wrapper classes*) für die einfachen Datentypen `boolean`, `char`, `byte`, `short`, `int`, `long`, `float` und `double`.

Sie stellen jeweils die Konstanten `MIN_VALUE` und `MAX_VALUE` sowie die folgenden Konstruktoren und Methoden bereit:

- Einen Konstruktor, der zu einem einfachen Typ ein Objekt der Typklasse erzeugt.
  (z. B. `Integer(int i)`).

- Einen Konstruktor mit `String`-Argument.

- Eine Methode *Typ*`Value()`, die den Wert als einfachen *Typ* zurückliefert.
  (z. B. `Integer.intValue()`).

- Die Methoden `toString()`, `equals()` und `hashcode()`.

Vererbung

Wrapper-Klassen

Darüber hinaus stellen einige der Klassen noch weitere Methoden bereit.
Die Klasse `Character` enthält z. B. die Klassenmethoden:

- `isLowerCase()`
- `isUpperCase()`
- `isDigit()`
- `isLetter()`
- `isSpaceChar()`
- `toLowerCase()`
- `toUpperCase()`

Die Wrapper-Klassen sind aus der Klasse `Object` abgeleitet. Damit können Objekte dieser Klassen verwendet werden, wenn ein Objekt des Typs `Object` verlangt wird

Beispiel: Umwandlung eines `int`-Werts in ein `Integer`-Objekt und umgekehrt

```
Integer mango;
int i = 49;

mango = new Integer(i);
i = mango.intValue();
```

Vererbung

Wrapper-Klassen Das Modul `java.lang`