Vorlesungsteil

Alles ist objektorientiert! Kurzer Rückblick

Lernen ist wie Rudern gegen den Strom.
Hört man damit auf, treibt man zurück.
Laozi

Übersicht zum Vorlesungsinhalt

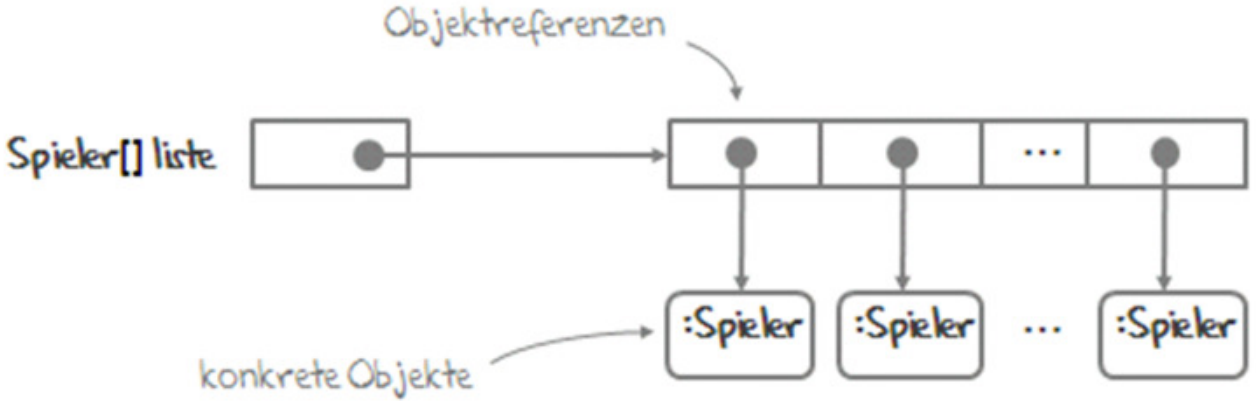
zeitliche Abfolge und Inhalte können variieren

Alles ist objektorientiert! Kurzer Rückblick

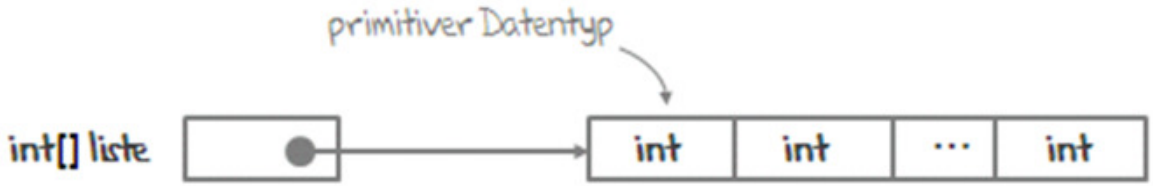
- Reservierung von Speicher
- Das clone-Schaf Dolly
- Alles erbt von Object
- Überschreiben von Methoden
- Doubleliste
- Äqzuivalenzrelationen
- Eigene Objektausgaben
- Wrapperklassen
- Referenzvariablen, this
- Zugriffsmodifikatoren
- Eigene Exception-Klasse erstellen
- getMessage/printStackTrace
- Interface versus Abstrakte Klassen
- Klassenexemplare casten
- Prinzip des Überladens
- Überladen von Konstruktoren
- Default-Konstruktor
- statische Attribute und Methoden
- String
- Klassen casten
- Konstruktoren privat
- Singleton
- Garbage Collector

Reservierung von Speicher

Wir haben mit Spieler[] ein Array von Objekten erzeugt. Genauer gesagt handelt es sich dabei um eine Objektreferenz, die auf eine Liste von Objektreferenzen:



Etwas anders verhält es sich bei Arrays von primitiven Datentypen:



Das clone-Schaf Dolly

Angenommen, uns steht eine Klasse Schaf mit den Eigenschaften name und farbe zur Verfügung:

```
public class Schaf {
   private String name;
   private String farbe;

   public Schaf(String name, String farbe) {
      setName(name);
      setFarbe(farbe);
   }

   public void setName(String name) {
      this.name = name;
   }

   public void setFarbe(String farbe) {
      this.farbe = farbe;
   }

   public String gibAus() {
      return "Schaf ("+name+", "+farbe+")";
   }
}
```

Das clone-Schaf Dolly II

Wenn wir jetzt eine kleine Schafherde erzeugen, diese clonen und anschließend eines der Schafe umbenennen und umfärben, hat das Effekte auf beide Herden:

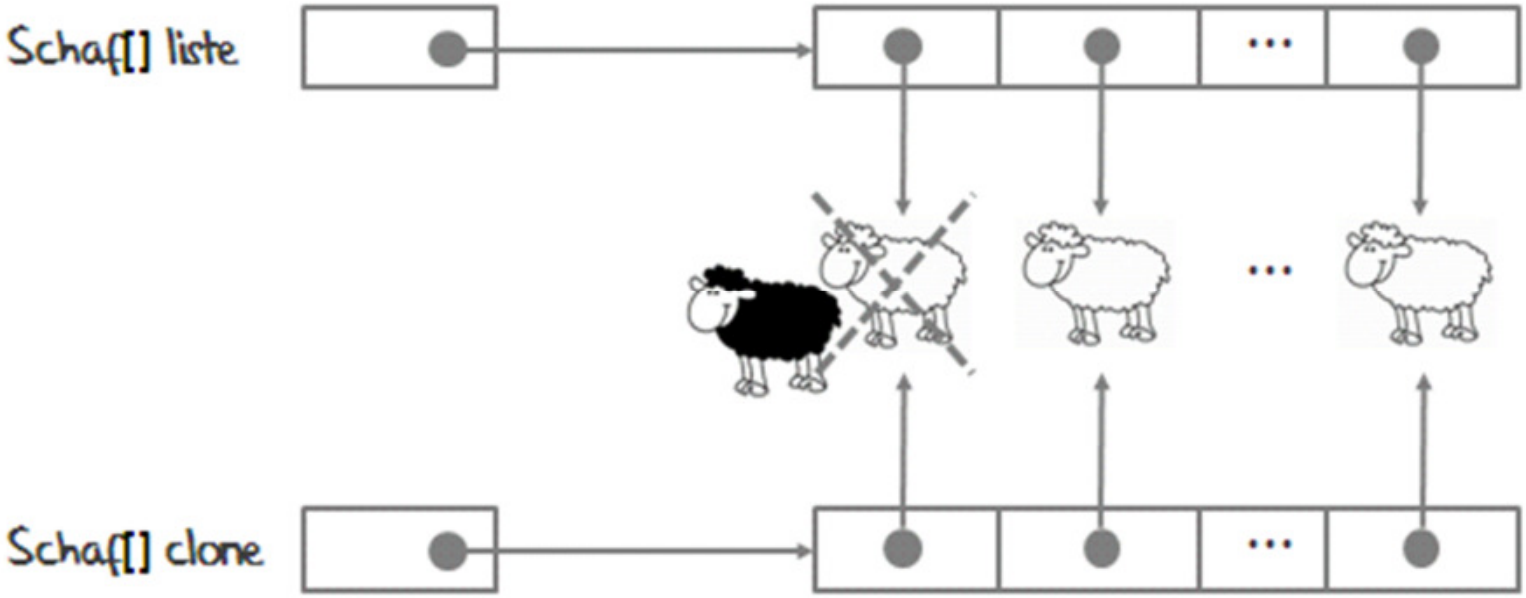
```
Schaf[] liste = new Schaf[4];
liste[0] = new Schaf("Bernd", "weiß");
liste[1] = new Schaf("Susi",  "weiß");
liste[2] = new Schaf("Hanni", "weiß");
liste[3] = new Schaf("Klaus", "weiß");

Schaf[] clone = liste.clone();
clone[0].setName("Falco");
clone[0].setFarbe("schwarz");

System.out.println(clone[0].gibAus());
```

Als Ausgabe erhalten wir:

```
Schaf (Falco,schwarz)
```

Das clone-Schaf Dolly III

Jetzt ist die Arbeitsweise der vorgestellten clone-Methode besser verständlich. Es wird eine flache Kopie der Arrayeinträge vorgenommen. Bei einem einfachen Array primitiver Datentypen ist das Resultat eine komplette Kopie des Arrays, aber bei einem Referenztyp nicht.



Alles erbt von Object

In der Bezeichnung Objektreferenz ist die wichtigste Information bereits vorhanden, denn alle Klassen erben von der Klasse Object. Die Klasse Object liegt auf dem obersten Platz in der VererbungsstrukturVererbungsstruktur und ist die Basis für jedes Javaprogramm.

Zwei Methoden der Klasse Object sind equals und toString. Schauen wir uns kurz mal ein Beispiel dazu an:

```
Object meinObjekt1 = new Object();
Object meinObjekt2 = new Object();

// Vergleich zweier Objekte
System.out.println(meinObjekt1.equals(meinObjekt2));

// Ausgabe der Objekte
System.out.println(meinObjekt1);
System.out.println(meinObjekt2);
```

Zwei Exemplare der Klasse Object wurden erzeugt, verglichen und anschließend ausgegeben. Bei der Ausgabe wird die Methode toString verwendet. Als Ausgabe erhalten wir:

```
false
java.lang.Object@3e25a5
java.lang.Object@19821f
```

Alles erbt von Object II

So ähnlich sah die Ausgabe in einem vorhergehenden Beispiel aus. Obwohl wir in der Klasse Trainer nicht explizit angegeben haben, dass von der Klasse Object geerbt wird, sehen wir in diesem Beispiel, dass die beiden Methoden equals und toString vorhanden sind:

```
Trainer neid   = new Trainer("Silvia Neid", 47, 8);
Trainer jaeger = new Trainer("Ferdinand Jaeger", 50, 3);

System.out.println(neid.equals(jaeger));
System.out.println(neid);
System.out.println(jaeger);
```

Die entsprechenden Ausgaben liefern:

```
false
kapitel7.Trainer@150bd4d
kapitel7.Trainer@1bc4459
```

Überschreiben von Methoden

Bei der Vererbung können wir Methoden aus den übergeordneten Klassen überschreiben. Dazu verwenden wir die gleiche Methodensignatur wie in der Basisklasse an und implementieren ein neues Verhalten.

```
public class Textausgabe {
   public void gibAus() {
      System.out.println("Basisklasse");
   }
}
```

Wann immer wir ein Exemplar der Klasse Textausgabe erzeugen und die Methode gibAus verwenden, wird die Zeichenkette „Basisklasse” ausgegeben. Wenn wir eine weitere Klasse von dieser ableiten, ohne eine Methode gibAus anzugeben, wissen wir bereits, dass diese auch dort zur Verfügung steht:

```
public class TextausgabeNeu extends Textausgabe {
   public static void main(String[] args) {
      new TextausgabeNeu().gibAus();
   }
}
```

Überschreiben von Methoden II

Jetzt wollen wir die Methode gibAus überschreiben:

```
public class TextausgabeNeu extends Textausgabe {
   public void gibAus() {
      System.out.println("Subklasse");
   }

   public static void main(String[] args) {
      new TextausgabeNeu().gibAus();
   }
}
```

Bei dem erneuten Programmstart erhalten wir die Zeichenkette „Subklasse”. Beim Ableiten einer Klasse konnten wir also das Verhalten durch Überschreiben einer Methode verändern.

Doubleliste erstellen I

```
public class DoubleListe {
   private double liste[];

   public DoubleListe(int num) {
      setListe(new double[num]);
   }

   public DoubleListe(double[] liste) {
      this.setListe(liste);
   }

   public double[] getListe() {
      return liste;
   }

   public void setListe(double liste[]) {
      this.liste = liste;
   }

   public static void main(String[] args) {
      DoubleListe d1, d2;

      d1 = new DoubleListe(new double[]{0.0, 0.1, 0.2, 0.3});
      d2 = new DoubleListe(new double[]{0.0, 0.1, 0.2, 0.3});

      if (!d1.equals(d2))
         System.out.println("Unterschiedliche Listen!");
      else
         System.out.println("Gleiche Listen!");

      System.out.println(d1);
      System.out.println(d2);
   }
}
```

Doubleliste erstellen II

Wir erhalten bei der Ausführung dieses Programms die folgende Ausgabe:

```
Unterschiedliche Listen!
kapitel7.DoubleListe@3e25a5
kapitel7.DoubleListe@19821f
```

Die Listen sind unterschiedlich, denn auch hier werden die Objektreferenzen verglichen. Bei der Ausgabe erhalten wir, wie schon bei dem Beispiel Person gesehen, kryptisch aussehende Zeichenketten.

Der Text mit dem @-Symbol ist die standardmäßige Textdarstellung eines Objekts, der aus drei Teilen besteht: die zu dem Objekt gehörige Klasse (in diesem Fall DoubleListe), dem @-Symbol und die in Java intern verwendete Hexadezimaldarstellung des Objekts.

Objekte vergleichen

Die Methode equals verwendet diese interne Darstellung zum Vergleich der Objekte, deshalb sind d1 und d2 unterschiedlich, obwohl sie den gleichen Inhalt haben.

Wenn wir diese Methoden in unserer Klasse mit folgenden beiden Definitionen überschreiben, können wir die Ausgabe und Funktionalität nach unseren Wünschen anpassen:

```
public boolean equals(Object obj) {
   // Handelt es sich um eine DoubleListe?
   if (obj instanceof DoubleListe) {
      DoubleListe dl = (DoubleListe)obj;

      // Listen unterschiedlich lang?
      if (dl.liste.length != liste.length)
         return false;

      // Inhalte vergleichen
      for (int i=0; i<liste.length; i++)
         if (dl.liste[i] != liste[i])
            return false;

      return true;
   }
   return false;
}
```

Eigenschaften der Äquivalenzrelation

Formal gesehen muss die überschriebene Methode equals alle Eigenschaften einer Äquivalenzrelation erfüllen. Sie muss reflexiv sein, also für jede gültige Eingabe x folgenden Aufruf zum Resultat true führen:

```
x.equals(x)
```

Sie muss symmetrisch sein, also für jede gültige Kombination x und y den folgenden Aufruf zum Resultat true führen:

```
x.equals(y) == y.equals(x)
```

Auch die Transitivität muss erfüllt sein: Wenn also für jede gültige Kombination x, y und z die folgenden Aufrufe zu true führen:

```
x.equals(y)     // ist true
y.equals(z)     // ist true
```

dann folgt daraus auch, dass der folgende Aufruf true ist:

```
x.equals(z)
```

Zusätzlich zu diesen drei Eigenschaften muss equals immer konsistent sein, also für gleiche Eingaben immer das gleiche Resultat liefern. Bei der Überprüfung des besonderen Falls mit der Eingabe null

```
x.equals(null)
```

sollte die Methode immer false liefern.

Eigene Objektausgabe definieren

Die Methode toString verwendet die Klasse Klasse!StringBufferStringBuffer zur Erzeugung der Ausgabe. Zuerst wird DoubleListe[in den BufferBuffer eingefügt, anschließend die Inhalte der Listenelemente und zu guter Letzt die schließende Klammer], wie es der folgende Codeabschnitt zeigt:

```
public String toString() {
   StringBuffer sb = new StringBuffer();
   sb.append("DoubleListe [");

   for (int i=0; i<liste.length-1; i++)
      sb.append(liste[i] + ", ");

   if (liste.length>0)
      sb.append(liste[liste.length-1]);
   sb.append("]");
   return sb.toString();
}
```

Durch das Überschreiben der beiden Methoden equals und toString erhalten wir ein neues Verhalten:

```
Gleiche Listen!
DoubleListe [0.0, 0.1, 0.2, 0.3]
DoubleListe [0.0, 0.1, 0.2, 0.3]
```

Trainer vergleichen

Wir könnten auf diese Weise auch die Klasse Trainer erweitern und bei Prüfung auf Gleichheit die Attribute vergleichen:

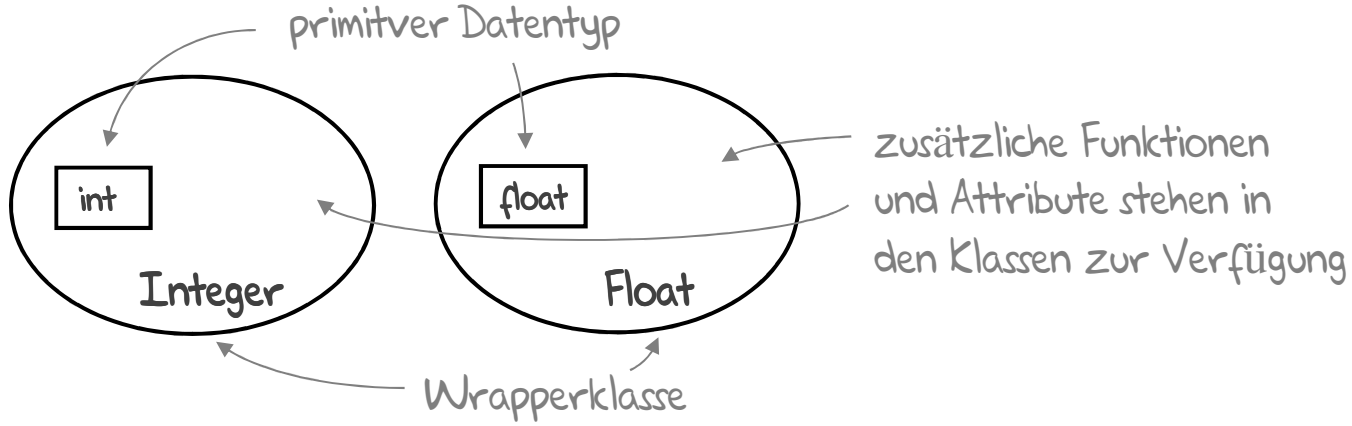
```
public boolean equals(Object obj) {
   if (obj instanceof Trainer) {
      Trainer t = (Trainer)obj;
      return (t.getAlter() == getAlter() &&
              t.getErfahrung() == getErfahrung() &&
              t.getName() == getName());
   }
   return false;
}

public String toString() {
   StringBuffer sb = new StringBuffer();
   sb.append("Trainer: ");
   sb.append(getName() + " (");
   sb.append(getAlter() + " Jahre, Erfahrung ");
   sb.append(getErfahrung() + ")");
   return sb.toString();
}
```

Sollten den Vergleich und die Ausgabe zweier Trainer, erhalten wir jetzt:

```
Trainer: Silvia Neid (47 Jahre, Erfahrung 8)
Trainer: Ferdinand Jaeger (50 Jahre, Erfahrung 3)
```

Primitive Datentypen und ihre Wrapperklassen I

Zu jedem primitiven Datentyp gibt es eine entsprechende Wrapperklasse (=Hülle). In dieser Klasse werden unter anderem eine Reihe von Konvertierungsmethoden angeboten:



Ein paar Beispiele:

```
int a     = 4;
Integer i = new Integer(a);
int b     = i.intValue();
```

Primitive Datentypen und ihre Wrapperklassen II

Da die Umwandlung von primitiven Datentypen zu ihren Hüllklassen und umgekehrt sehr oft verwendet wird. Aus diesem Grund gibt es seit Java 1.5 eine vereinfachte Möglichkeit der direkten Typumwandlung :

```
int a = 4;
Integer i = a;    // boxing
int b = i;        // unboxing
```

Die Umwandlung von einem primitiven Datentypen zu seiner Wrapperklasse wird als boxing und die Umkehrung als unboxing bezeichnet.

Damit vereinfachen sich die Ausdrücke zwar erheblich, allerdings werden die automatischen Typumwandlungen über die entsprechenden get- und set-Methoden realisiert, worauf in geschwindigkeitskritischen Programmabschnitten auf jeden Fall verzichtet werden sollte.

Analog zu den weiteren primitiven Datentypen heißen die entsprechenden Wrapperklassen Boolean, Byte, Short, Long, Float und Double:

```
Boolean   b  = new Boolean(true);
Byte      by = new Byte((byte)5);
Short     s  = new Short((short)2);
Character c  = new Character('!');
Long      l  = new Long(13983816);
Float     f  = new Float(0.13872f);
Double    d  = new Double(1.314159);
```

Primitive Datentypen und ihre Wrapperklassen III

Neben den zahlreichen Konvertierungsmethoden liefern die Wrapperklassen auch Konstanten für die Extremwerte des entsprechenden Wertebereichs. Den minimalen und maximalen Wert eines int, haben wir bereits kennengelernt.

Auf diese Weise lassen sich auch die Extremwerte der anderen Wrapperklassen auslesen. So z. B. der kleinste Wert eines byte und der größte eines float:

```
byte  bMin = Byte.MIN_VALUE;
float fMax = Float.MAX_VALUE;
```

Es sei noch kurz erwähnt, dass in Java durch die Klassen BigInteger und BigDecimal beliebig große, bzw. beliebig genaue Zahlendarstellungen zu verschiedenen Basen möglich sind.

Referenzvariablen

Zur Erinnerung: Wir haben beim Fußballmanager die Klasse Person implementiert.
Hier noch einmal der Programmcode dieser Klasse:

```
public class Person {
   // Eigenschaften einer Person
   private String name;
   private byte alter;

   // Konstruktoren
   public Person(String n, int a) {
      name  = n;
      alter = (byte)a;
   }

   // Funktionen (get und set):
   ...
}
```

Referenzen kopieren?

Um zu verstehen, dass Referenzen Adressverweise auf einen reservierten Speicherplatz sind, schauen wir uns folgendes Beispiel an:

```
Person p1, p2;
p1 = new Person("Hugo", 12);
p2 = p1;

if (p1 == p2)
   System.out.println("Die Referenzen sind gleich");
```

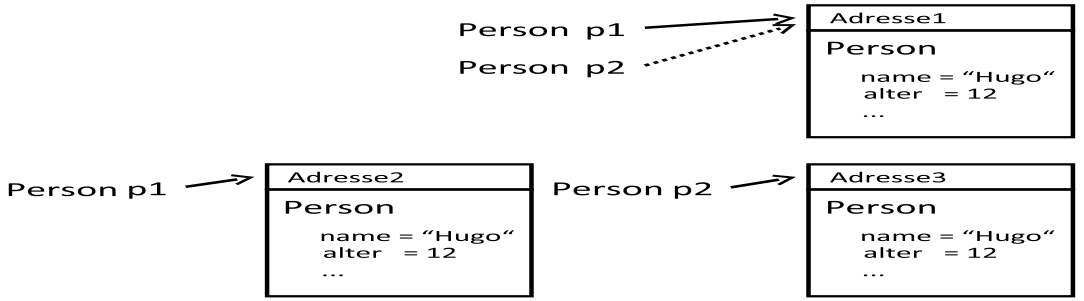
In der ersten Zeile werden p1 und p2 deklariert. p1 ist eine Referenzvariable und beinhaltet eine Adresse.

Die zweite Zeile stellt Speicherplatz bereit und erzeugt ein Objekt der Klasse Person und vergibt den Attributen gleich Werte. p1 zeigt nun auf diesen Speicherbereich.

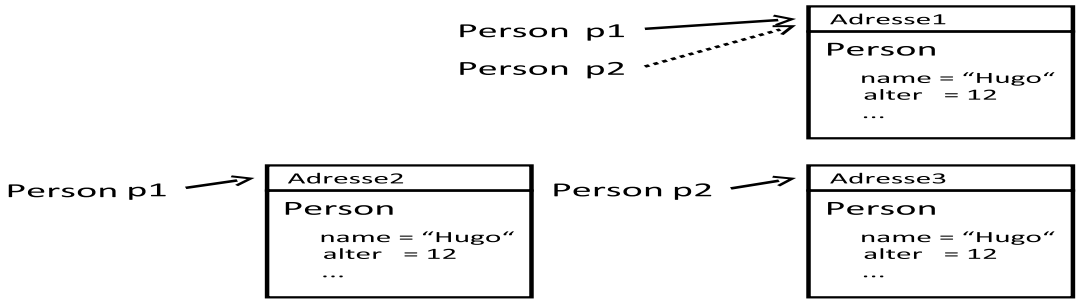
In Zeile drei weisen wir die Adresse von p1 der Variablen p2 zu. Beide Variablen zeigen nun auf den gleichen Speicherplatz, auf dasselbe Objekt. Sollten wir Veränderungen in p1 vornehmen, so treten diese Veränderungen ebenfalls bei p2 auf, denn es handelt sich um dasselbe Objekt!

Zweimal Speicherplatz reservieren

```
Person p1;
p1 = new Person("Hugo", 12);

Person p2;
p2 = p1;
```



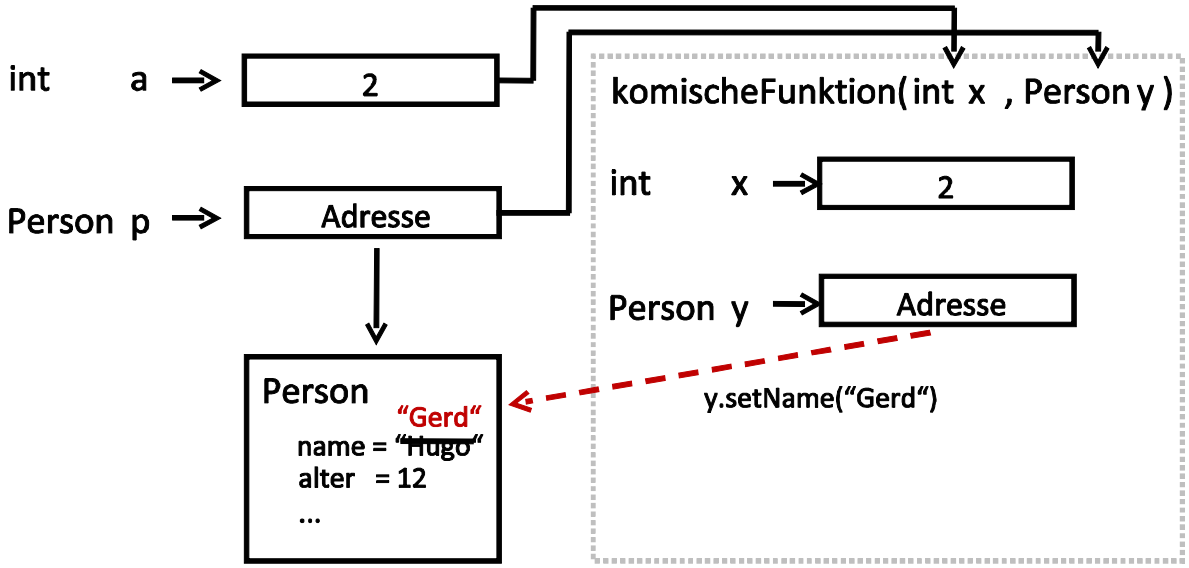

```
Person p1, p2;
p1 = new Person("Hugo", 12);
p2 = new Person("Hugo", 12);
```



Zweimal Speicherplatz reservieren

```
int a    = 2;
Person p = new Person("Hugo", 12);

// An dieser Stelle hat p.getName() den Rückgabewert "Hugo"

komischeFunktion(a, p);
```

```
public void komischeFunktion(int x, Person y){
   // Wir ändern die Werte der Eingabeparameter:
   x = 7;
   y.setName("Gerd");
}
```



Live-Coding-Session

Referenzvariable this

Wir haben schon in einem vorhergehenden Abschnitt gesehen, wie sich die Klassenvariablen von "außen" setzen lassen. Jedes Objekt kann aber auch mittels einer Referenzvariablen auf sich selbst referenzieren und seine Variablen und Funktionen ansprechen.

Dazu wurde die Klasse Person im folgenden Beispiel vereinfacht:

```
public class Person{
   private String name;

   public void setName(String name){
      this.name  = name;
   }
}
```

Zugriffsmodifikatoren

Neben private und public gibt es noch protected:

"package private"

	private	public	protected	(ohne)
Klasse selbst	ja	ja	ja	ja
Paketklassen/ innere Klassen	nein	ja	ja	ja
Unterklassen	nein	ja	ja	nein
sonst. Klassen	nein	ja	nein	nein

Eigene Exceptionklasse erstellen

Wir können jetzt eine eigene Fehlerklasse erzeugen:

```
public class MyException extends Exception {
    public MyException() {
    }

    public MyException(String s) {
       super(s);
    }
}
```

und werfen:

```
public class MyExceptionTest {
   public static void main(String[] args) {
      try {
         throw new MyException("Bekannter Fehler aufgetreten ...");
      } catch(MyException e) {
         System.out.print(e.getMessage());
      }
   }
}
```

Message und StackTrace

Mit getMessage liefert wir den zum Fehler zugehörigen String. Mit printStackTrace können wir den Fehlerort lokalisieren:

```
public class MyExceptionTest {

   private static void funktion() {
       try {
          throw new MyException("Bekannter Fehler aufgetreten ...");
        } catch(MyException e) {
          System.out.println("Message:    " + e.getMessage());
          System.out.println("StackTrace: ");
          e.printStackTrace();
        }
   }

   public static void main(String[] args) {
        funktion();
   }
}
```

Liefert:

```
Message:    Bekannter Fehler aufgetreten ...
StackTrace:
MyException: Bekannter Fehler aufgetreten ...
     at MyExceptionTest.funktion(MyExceptionTest.java:6)
     at MyExceptionTest.main(MyExceptionTest.java:15)
```

Interface versus abstrakte Klasse

Im Vergleich zu einem Interface, bei dem es nur Funktionsköpfe gibt, besteht bei einer abstrakten Klasse die Möglichkeit, neben abstrakte Funktionabstrakten Funktionen (Funktionsköpfe) bereits bei der Definition der Klasse, Funktionen vollständig zu implementieren.

Dabei muss eine abstrakte Klasse weder zwingend eine abstrakte noch eine implementierte Methode besitzen. Deshalb kann das Interface als Spezialfall der abstrakten Klasse mit ausschliesslich abstrakten Funktionen betrachtet werden.

Abstrakte Klasse

Schauen wir uns ein ganz kurzes Beispiel dazu an. Die Klasse A verwaltet die Variable wert und bietet bereits die Methode getWert.

Die Methode setWert soll aber implementiert werden und deshalb wird sie mit dem Schlüsselwort abstract versehen:

```
public abstract class A {
   protected int wert;

   public int getWert() {
      return wert;
   }

   public abstract void setWert(int w);
}
```

Die Klasse B erbt von A, also wert und die Methode getWert, muss aber die Methode setWert implementieren. Durch das Schlüsselwort protected ist es der Klasse B nach der Vererbung erlaubt, auf die Variable wert zuzugreifen:

```
public class B extends A{
   public void setWert(int w){
      this.wert = w;
   }
}
```

Kleiner Test mit A

Jetzt nehmen wir noch eine Testklasse Tester dazu und testen mit ihr die gültige Funktionalität. Als erstes wollen wir versuchen ein Exemplar der Klasse A zu erzeugen:

```
A a = new A();
```

Das schlägt mit der folgenden Ausgabe fehl:

```
C:\JavaCode>javac Tester
Tester:3: A is abstract; cannot be instantiated
                A a = new A();
                      ^
1 error
```

Es ist nicht erlaubt von einer abstrakten Klasse ein Exemplar zu erzeugen!

Noch ein Test aber mit B

Von der Klasse B können wir problemlos ein Exemplar erzeugen:

```
B b = new B();
b.setWert(4);
System.out.println(""+b.getWert());
```

Wir erzeugen ein Exemplar der Klasse B, die nun keine abstrakten Methoden enthält und verwenden beide Methoden:

```
C:\JavaCode>javac Tester
C:\JavaCode>java Tester
4
```

Es hat dieses Mal funktioniert.

Live-Coding-Session

Person abstrakt machen

Die Klasse Person eignet sich, um sie als abstrakte Klasse zu definieren. Wir wollen von Person kein Exemplar erzeugen:

```
public abstract class Person {
   // Eigenschaften einer Person
   private String name;
   private byte alter;

   // Konstruktoren
   public Person(String n, int a) {
      name  = n;
      alter = (byte)a;
   }

   // Funktionen (get und set):
   ...
}
```

Wir erzwingen hier also einen Vererbungsschritt.

Klassenexemplare casten

Wir haben den Mechanismus der impliziten und expliziten Typumwandlungen kennengelernt. Wenn wir primitive Datentypen in Abhängigkeit ihres Typs und dem Speicherbedarf umwandeln können, warum sollte das nicht auch für Objekte funktionieren?

Mit der Vererbungshierarchie der Klassen Person, Spieler und Trainer sind wir bestens vertraut, daher schauen wir uns folgendes Programmbeispiel an:

```
Person  p = new Person("Person", 10);
Spieler s = new Spieler("Spieler", 20, 1, 1, 1);
Trainer t = new Trainer("Trainer", 30, 1);
```

Über die vorhandenen Konstruktoren können wir wie gewohnt, Exemplare erzeugen.

Klassenexemplare casten II

Da abgeleitete Klassen Spezialisierungen darstellen, bei denen zusätzliche Methoden angeboten werden können, ist ebenfalls die folgende Erstellung von Exemplaren erlaubt:

```
Person pSpieler = new Spieler("Spieler", 20, 1, 1, 1);
Person pTrainer = new Trainer("Trainer", 30, 1);
```

Bei der Erzeugung der beiden Objekte haben wir Exemplare der Klassen Spieler und Trainer erstellt. Die Sichtbarkeit der Methoden ist allerdings abhängig vom angegebenen Deklarationstyp, in diesem Fall Person.

Es handelt sich dabei um eine implizite Objektumwandlung (narrowing), da zunächst ein Objekt vom Typ Spieler erzeugt wird (real type) und dieses anschließend nach aussen, über die Objektreferenz vom Typ Person, sichtbar ist (apparent type). Erlaubt ist die implizite Objektumwandlung nur zu Klassen, die in der Vererbungshierarchie über dem Objekt liegen. So dürfen wir beispielsweise alle Objekttypen implizit zum Typ Object umwandeln.

Nicht erlaubt sind Typumwandlungen von Objekten, die nicht auf einem Vererbungspfad liegen, so z. B. Trainer und Spieler. Beide könnten wir als Geschwisterklassen bezeichnen, da sie einen gemeinsamen Vorfahren haben.

Klassenexemplare casten III

Wir dürfen also auch nur Methoden aufrufen, die durch die Klasse Person sichtbar sind:

```
System.out.println(pSpieler.getName());
System.out.println(pTrainer.getAlter());
```

Nicht erlaubt ist der Aufruf der Methoden aus den spezialisierten Klassen:

```
System.out.println(pSpieler.getTore());
System.out.println(pTrainer.getErfahrung());
```

Wir dürfen Objekte auch nachträglich implizit zu Vertretern der Superklassen umwandeln:

```
pSpieler = s;
pTrainer = t;
```

Da pSpieler und pTrainer durch die spezialisierten Konstruktoren entworfen wurden, können wir die Sichtbarkeit auch entsprechend explizit wieder rückgängig machen (widening):

```
Spieler sPerson = (Spieler)pSpieler;
Trainer tPerson = (Trainer)pTrainer;
```

Klassenexemplare casten IV

Der Compiler erlaubt zwar zunächst die folgende Typumwandlung, bei der Ausführung wird allerdings ein Laufzeitfehler erzeugt und das Programm abgebrochen:

```
sPerson = (Spieler)p;
tPerson = (Trainer)p;
```

Der Grund ist auch klar, denn bei der Erzeugung des Exemplars p zu Beginn dieses Abschnitts wurde der Konstruktor der Klasse Person verwendet und der Klasse Person müssen die Eigenschaften von Spieler und Trainer nicht bekannt sein.