Objektorientierte Programmierung mit Java

Inhalt:

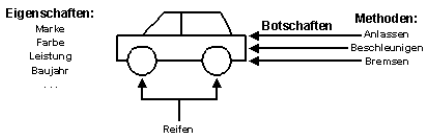
- Generalisierung, Spezialisierung
- Module, Klassen, Objekte
- Klassenhierarchien, Vererbung, abstrakte Klassen, Schnittstellen
- Entwicklung eines Fußballmanagers (EM 2008)

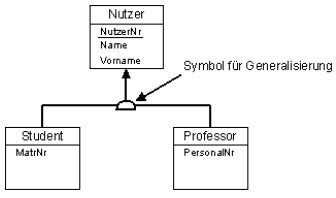
Block M.: "*Java-Intensivkurs - In 14 Tagen lernen Projekte erfolgreich zu realisieren*" , Springer-Verlag 2007
Gries D., Gries P.: "*Multimedia Introduction to Programming Using Java*", Springer-Verlag 2005
Abts D.: „*Grundkurs JAVA: Von den Grundlagen bis zu Datenbank- und Netzanwendungen*", Vieweg-Verlag 2007

Objektorientierte Programmierung mit Java

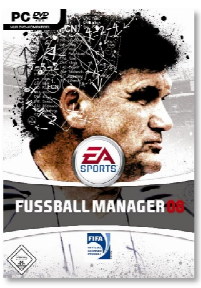
Einführung in die Objektorientierung (üblicherweise)

- Autos und Autoteile
- Professoren und Studenten
- ...





Wir machen das nicht! Die **Europomeisterschaft 2008** ist gerade vorbei und wir entwickeln zusammen einen einfachen Fußballmanager ...



Vielleicht nicht so professionell, wie der von EA Sports, aber dafür können wir hinterher sagen, dass wir es selber gemacht haben ☺.

Marco Block Sommersemester 2008

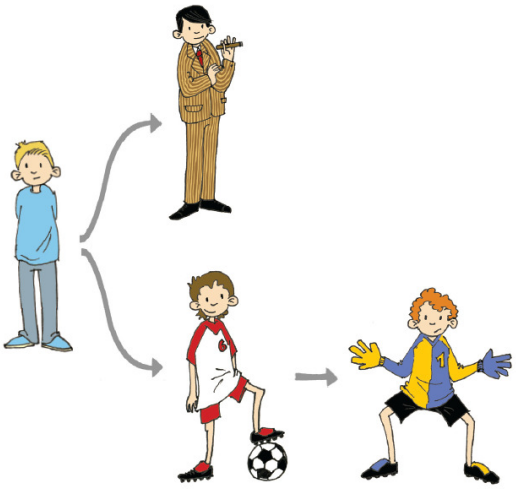
Freie Universität Berlin

Generalisierung und Spezialisierung

Unter den beiden Begriffen Generalisierung und Spezialisierung verstehen wir zwei verschiedene Vorgehensweisen, Kategorien und Stammbäume von Dingen zu beschreiben. Wenn wir bei Dingen Gemeinsamkeiten beschreiben und danach kategorisieren, dann nennen wir das eine **Generalisierung**.

Mit der **Spezialisierung** beschreiben wir den umgekehrten Weg, aus einem „Ur-Ding“ können wir durch zahlreiche Veränderungen der Eigenschaften neue Dinge kreieren, die Eigenschaften übernehmen oder neue entwickeln.

Spezialisierung von links nach rechts.



Generalisierung von rechts nach links.

Freie Universität Berlin

Gemeinsamkeiten von Spieler und Trainer

Wir haben **11 Spieler**, z.B. mit den Eigenschaften *Name*, *Alter*, *Stärke*, *Torschuss*, *Motivation* und *Tore*. Neben den Spielern haben wir einen Trainer mit den Eigenschaften *Name*, *Alter* und *Erfahrung*.

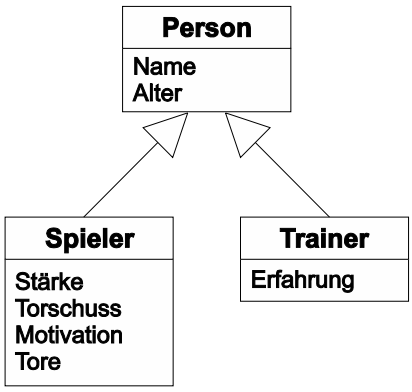
Spieler
Name Alter Stärke Torschuss Motivation Tore

Trainer
Name Alter Erfahrung

Wir sehen schon, dass es Gemeinsamkeiten gibt.

Klassen und Vererbung

Die Gemeinsamkeiten können wir zu einer eigenen Kategorie zusammenfassen und diese Eigenschaften an beide, **Spieler** und **Trainer**, vererben. Diese neue Kategorie nennen wir z.B. **Person**.
Spieler, **Trainer** und **Person** sind Kategorien oder Klassen.



Die Pfeile zeigen in die Richtung des Vorfahren. Wir haben also eine Repräsentation gefunden, die die Daten nicht mehr unnötig doppelt darstellt, wie es bei *Name* und *Alter* gewesen wäre, sondern sie übersichtlich nur einmal in der Klasse **Person** abgelegt. Des Weiteren wurde die Spezialisierung der **Person** zu den Klassen **Spieler** und **Trainer** durch zusätzliche Eigenschaften beschrieben.

Objektorientierte Programmierung mit Java

Umsetzung in Java I

Um diese Darstellung in einem Javaprogramm zu implementieren, müssen wir drei Klassen **Person**, **Spieler** und **Trainer** anlegen.

Beginnen wir mit der Klasse **Person**:

Das ist als „Bauplan“ zu verstehen.

```
public class Person{
   // Eigenschaften einer Person:
   public String name;
   public int alter;
}
```

Jetzt wollen wir die Klasse **Spieler** von **Person** ableiten und alle Eigenschaften, die die Klasse anbietet, übernehmen. In Java erweitern wir die Definition einer Klasse mit dem Befehl extends und den Namen der Klasse, von der wir ableiten wollen.

```
public class A extends B{
}
```

Umsetzung in Java II

Wir erweitern den Bauplan von Person.

Jetzt können wir **Spieler** von **Person** ableiten:

```
public class Spieler extends Person{
   // Zusätzliche Eigenschaften eines Spielers:
   public int staerke;      // von 1 (schlecht) bis 10 (super)
   public int torschuss;    // von 1 (schlecht) bis 10 (super)
   public int motivation;   // von 1 (schlecht) bis 10 (super)
   public int tore;
}
```

Das war alles.

Jetzt haben wir zwei Klassen **Person** und **Spieler**. Alle Eigenschaften oder Attribute der Klasse **Person** sind nun auch in **Spieler** enthalten, darüber hinaus hat ein Spieler noch die Attribute *staerke*, *torschuss*, *motivation* und *tore*.

Mit diesen beiden Klassen haben wir eine einfache Vererbung realisiert.

Objektorientierte Programmierung mit Java

Modifizierer public und private I

Nehmen wir an, dass zwei verschiedene Programmierer diese Klassen geschrieben haben und dass der Programmierer der Klasse **Person** für die Variable *alter* nur positive Zahlen zwischen 1 und 100 akzeptieren möchte. Er hat aber keinen Einfluss auf die Verwendung, da die Variable *alter* mit dem zusätzlichen Attribut public versehen wurde.

Das bedeutet, dass jeder, der diese Klasse verwenden möchte auf diese Attribute uneingeschränkt zugreifen kann!

Es gibt die Möglichkeit diese Variablen vor Zugriff zu schützen, indem das Attribut private verwendet wird.

Modifizierer public und private II

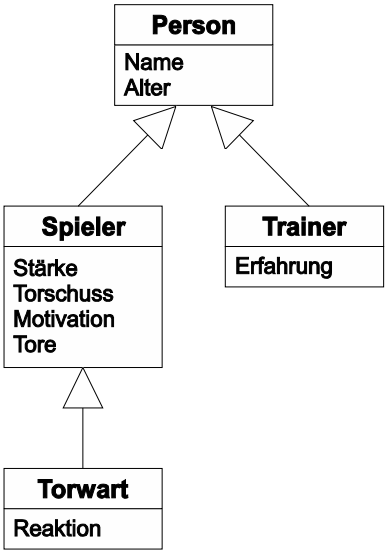
Jetzt kann diese Variable nicht mehr außerhalb der Klasse angesprochen werden. Um aber die Variablen verändern und lesen zu können, schreiben wir zwei Funktionen und vergeben ihnen das Attribut public:

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

   // Funktionen (get und set):
   public String getName(){
      return name;
   }

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

   public int getAlter(){
      return alter;
   }

   public void setAlter(int a){
      alter = a;
   }
}
```

get-set-Methoden zur Veränderung der Klassen-variablen

Torwart als Spezialisierung eines Spielers

Bei den Spielern gibt es noch einen Sonderling, den Torwart. Als zusätzliche Eigenschaft hat er *reaktion*, damit wird später entschieden, ob er die Torschüsse hält oder nicht. Sicherlich könnten wir an dieser Stelle die Spieler noch in Abwehr, Mittelfeld und Angriff unterscheiden, aber fürs erste soll die Spezialisierung der Klasse **Spieler** zum **Torwart** als Spezialfall eines Spielers genügen:

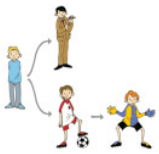


Freie Universität Berlin

Umsetzung in Java

Hier die aktuelle Klasse **Trainer** mit den entsprechenden get-set-Funktionen:

```
public class Trainer extends Person{
   // Zusätzliche Eigenschaften eines Trainers:
   private int erfahrung;

   // Funktionen (get und set):
   public int getErfahrung(){
      return erfahrung;
   }

   public void setErfahrung(int e){
      erfahrung = e;
   }
}
```



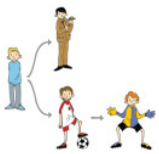
Objektorientierte Programmierung mit Java

Freie Universität Berlin

Objekte und Instanzen I

Bisher haben wir die Dinge klassifiziert (einen Bauplan erstellt) und die Gemeinsamkeiten in zusätzlichen Klassen beschrieben, aber noch keinen Spieler oder Trainer, der diese Eigenschaften und Funktionen besitzt erzeugt und untersucht.

Wenn von einer Klasse ein Exemplar erzeugt wird (die Klasse stellt sozusagen den Bauplan fest), dann nennt man das ein Objekt oder eine Instanz dieser Klasse.



Freie Universität Berlin

Objekte und Instanzen II

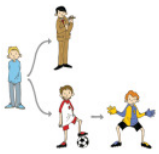
Wenn wir beispielsweise ein Programm schreiben wollen, dass mit Jürgen Klinsmann als Trainer arbeiten möchte, dann erzeugen wir eine neue Instanz der Klasse **Trainer** und geben ihm die Informationen name="Jürgen Klinsmann, alter=42 und erfahrung=7.

```
public class Test{
   public static void main(String[] args){
      Trainer trainer = new Trainer();
      trainer.setName("Jürgen Klinsmann");
      trainer.setAlter(42);
      trainer.setErfahrung(7);
   }
}
```

Wir erzeugen eine Instanz der Klasse Trainer. Ähnlich wie bei den primitiven Datentypen müssen wir Speicherplatz reservieren und machen das hier in der dritten Zeile.

Jetzt können wir mit dem Objekt trainer arbeiten. Sollten die Daten des Objekts geändert werden, so können wir das über die mit public versehenen Funktionen der Klassen **Trainer** und **Person** tun.
Wir sehen schon, dass die Funktion *setErfahrung* in der Klasse **Trainer** definiert wurde und verwendet werden kann.

Die **Person** Funktion *setName* wurde aber in definiert und kann trotzdem verwendet werden.



Objektorientierte Programmierung mit Java

Freie Universität Berlin

Konstruktoren I

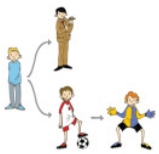
Im vorhergehenden Beispiel war es etwas umständlich, erst ein Objekt zu erzeugen und anschließend die Parameter über die set-Methoden zu setzen. Es geht bedeutend einfacher mit den Konstruktoren. Sie sind quasi die Funktionen, die bei der Reservierung des Speicherplatzes, bei der Erzeugung eines Objekts, ausgeführt werden.

Der Konstruktor entspricht der Syntax:

```
public <Klassenname>(Parameterliste){
}
```

Beim **Trainer-Beispiel** könnten wir zum Beispiel den folgenden Konstruktor angeben:

```
public class Trainer extends Person{
   // Zusätzliche Eigenschaften eines Trainers:
   private int erfahrung;

   // Konstruktoren
   public Trainer(String n, int a, int e){
      super(n, a);
      erfahrung       = e;
   }

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



Freie Universität Berlin

Konstruktoren II

So sah die Erzeugung ☺ von Trainer Jürgen Klinsmann vorhin aus:

```
public class Test{
   public static void main(String[] args){
      Trainer trainer = new Trainer();
      trainer.setName("Jürgen Klinsmann");
      trainer.setAlter(42);
      trainer.setErfahrung(7);
   }
}
```

Und so können wir es jetzt viel einfacher mit dem Konstruktur machen:

```
public class Test{
   public static void main(String[] args){
      Trainer trainer = new Trainer("Jürgen Klinsmann", 42, 7);
   }
}
```

Freie Universität Berlin

Konstruktoren III

Die Anweisung super mit den Parametern *name* und *alter* ruft den Konstruktor der Klasse auf, von der geerbt wird. In diesem Beispiel also den Konstruktor der Klasse **Person**.

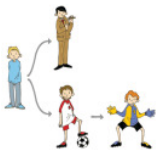
Da aber *name* und *alter* in der Klasse **Person** gespeichert sind und wir dort ebenfalls mit einem K Konstruktor eine einfachere Initialisierung eines Objekts haben möchten, ändern wir die Klasse **Person**, wie folgt:

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

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

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

Die Klasse Spieler I

Die Klasse **Spieler** verwendet den Konstruktor der Klasse **Person**:

```
import java.util.Random;

public class Spieler extends Person{
   // Zusätzliche Eigenschaften eines Spielers:
   private int staerke;      // von 1 (schlecht) bis 10 (super)
   private int torschuss;    // von 1 (schlecht) bis 10 (super)
   private int motivation;   // von 1 (schlecht) bis 10 (super)
   private int tore;

   // Konstruktoren
   public Spieler(String n, int a, int s, int t, int m){
      super(n, a);
      staerke         = s;
      torschuss       = t;
      motivation      = m;
      tore            = 0;
   }

   ...
```



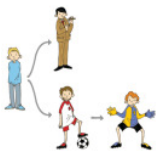
Die Klasse Spieler II

Der **Spieler** erhält zwei neue Methoden *addTor* und *schiesstAufTor*:

```
   ...

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

   // Spielerfunktionen:

   // Der Spieler hat ein Tor geschossen
   public void addTor(){
      tore++;
   }

   // eine Zahl von 1-10 liefert die Qualität des Torschusses mit
   // einem kleinen Zufallswert +1 oder -1
   public int schiesstAufTor(){
      Random r = new Random();
      // Entfernungspauschale :)
      torschuss = Math.max(1, Math.min(10, torschuss - r.nextInt(3)));
      // +-1 ist hier die Varianz
      int ret = Math.max(1, Math.min(10, torschuss + r.nextInt(3)-1));
      return ret;
   }
}
```

Zufallszahlen haben wir bereits bei Conways Game of Life gesehen.

Die Klasse **Spieler** hat eine Funktion *schiesstAufTor*, falls dieser Spieler in der Partie eine Torchance erhält, dann wird eine zufällige Zahl im Bereich von 1-10 gewählt, die abhängig von der Torschuss-qualität des Spielers ist.



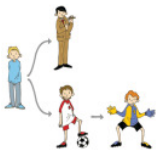
Freie Universität Berlin

Die Klasse Torwart

Der Torwart erhält eine Funktion, die entscheidet, ob der abgegebene Torschuss pariert oder durchgelassen wird (ebenfalls mit einem zufälligen Ausgang):

```
import java.util.Random;
public class Torwart extends Spieler{
   // Zusätzliche Eigenschaften eines Torwarts:
   private int reaktion;

   // Konstruktoren
   public Torwart(String n, int a, int s, int t, int m, int r){
      super(n, a, s, t, m);
      reaktion = r;
   }

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

   // Torwartfunktionen:

   // Als Parameter erhält der Torwart die Torschussstärke und nun muss
   // entschieden werden, ob der Torwart hält oder nicht
   public boolean haeltDenSchuss(int schuss){
      Random r = new Random();
      // +-1 ist hier die Varianz
      int ret = Math.max(1, Math.min(10, reaktion + r.nextInt(3)-1));
      if (ret>=schuss)
         return true;  // gehalten
      else
         return false; // TOR!!!
   }
}
```

Hier werden 3 Konstruktoren aufgerufen! Towart, Spieler und Person.



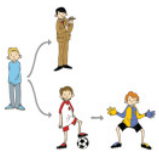
Objektorientierte Programmierung mit Java

Freie Universität Berlin

Die Mannschaft I

Wir haben die Spieler und den Trainer modelliert, jetzt ist es an der Zeit eine Mannschaft zu beschreiben. Eine **Mannschaft** ist eine Klasse mit den Eigenschaften *name*, *Trainer*, *Torwart* und *Spieler*. Es gibt wieder einen Konstruktor und die get-set-Funktionen.

```
public class Mannschaft{
   // Eigenschaften einer Mannschaft:
   private String name;
   private Trainer trainer;
   private Torwart torwart;
   private Spieler[] kader;

   // Konstruktoren
   public Mannschaft(String n, Trainer t, Torwart tw, Spieler[] s){
      name            = n;
      trainer         = t;
      torwart         = tw;
      kader           = s;
   }

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

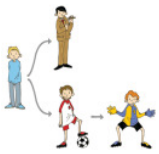


Freie Universität Berlin

Die Mannschaft II

Zusätzlich besitzt die Klasse **Mannschaft** die Funktionen *getStaerke* und *getMotivation* , die die durchschnittliche Stärke, bzw. Motivation der Mannschaft als Zahlenwert wiedergibt.

```
   ...
   // Mannschaftsfunktionen:

   // liefert die durchschnittliche Mannschaftsstaerke
   public int getStaerke(){
      int summ = 0;
      for (int i=0; i<10; i++)
         summ += kader[i].getStaerke();
      return summ/10;
   }

   // liefert die durchschnittliche Mannschaftsmotivation
   public int getMotivation(){
      int summ = 0;
      for (int i=0; i<10; i++)
         summ += kader[i].getMotivation();
      return summ/10;
   }
}
```



Freie Universität Berlin

Interfaces I

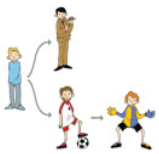
Die Klasse **Mannschaft** und alle dazugehörigen Klassen **Spieler**, **Torwart** und **Trainer** wurden definiert. Es ist an der Zeit, ein Freundschaftsspiel zweier Mannschaften zu realisieren. Auf die hier vorgestellte Weise könnten ganze Ligen und Turniere implementiert werden. Das wird Teil der Übungsaufgaben und der eigenen Motivation sein ☺.

Wir lernen dabei eine weitere wichtige Vererbungsvariante kennen. Zunächst beschreiben wir allgemein, wie wir uns ein Freundschaftsspiel vorstellen und implementieren es dann.

Jeder der ein Freundschaftsspiel, so wie wir es verstehen, implementieren möchte, kann sich an dieser Schnittstelle orientieren und bleibt mit unserem Kontext kompatibel. Es werden in diesem Interface (Schnittstelle) nur die Funktionen beschrieben, die jeder implementieren muss. Es gibt keine funktionsfähigen Programme, sondern nur die Funktionsköpfe:

Interface = Schnittstelle

```
public interface Freundschaftsspiel{
   String getHeimMannschaft();
   String getGastMannschaft();
   int getHeimPunkte();
   int getGastPunkte();

   String getErgebnisText();
}
```

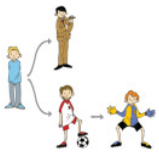


Die Klasse Fussballfreundschaftsspiel I

Anhand dieses Interfaces können wir speziell für den Fußball eine neue Klasse **FussballFreundschaftsspiel** entwerfen. Wir müssen alle vorgegebenen Funktionen eines Interfaces implementieren. Es können noch mehr dazu kommen, aber es dürfen keine fehlen.

Das Schlüsselwort implements signalisiert, dass wir ein Interface implementieren.

```
import java.util.Random;

public class Fussballfreundschaftsspiel implements Freundschaftsspiel{
  private String nameHeimMannschaft;
  private String nameGastMannschaft;
  private int punkteHeim;
  private int punkteGast;

  // Konstruktor
  public Fussballfreundschaftsspiel(){
     punkteHeim      = 0;
     punkteGast      = 0;
  }

  // Methoden des Interface, die implementiert werden müssen:
  public String getHeimMannschaft(){
     return nameHeimMannschaft;
  }

  public String getGastMannschaft(){
     return nameGastMannschaft;
  }

  ...
```



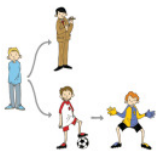
Die Klasse Fussballfreundschaftsspiel II

Die restlichen zwei Funktionen:

```
  ...
  public int getHeimPunkte(){
     return punkteHeim;
  }

  public int getGastPunkte(){
     return punkteGast;
  }
```

Nachdem die Funktionen *getHeimMannschaft*, *getGastMannschaft*, *getHeimPunkte* und *getGastPunkte* implementiert sind, folgt die Methode *starteSpiel*.

```
  // Ein Fussballfreundschaftsspiel zwischen beiden Mannschaften wird
  // gestartet.
  public void starteSpiel(Mannschaft m1, Mannschaft m2){
     nameHeimMannschaft      = m1.getName();
     nameGastMannschaft      = m2.getName();
     punkteHeim              = 0;
     punkteGast              = 0;

     // jetzt starten wir das Spiel und erzeugen für die 90 Minuten
     // Spiel plus Nachspielzeit die verschiedenen Aktionen
     // (wahrscheinlichkeitsbedingt) für das Freundschaftsspiel
     Random r = new Random();
     ...
```

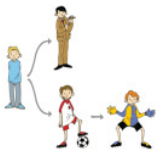


Freie Universität Berlin

Die Klasse Fussballfreundschaftsspiel III

Weiter geht's mit der Methode *starteSpiel*:

```
...
boolean spiellaeuft  = true;
int spieldauer       = 90 + r.nextInt(5);
int zeit             = 1;
int naechsteAktion;

// solange das Spiel laeuft, koennen Torchancen entstehen...
while (spiellaeuft){
   naechsteAktion = r.nextInt(15)+1;

   // Ist das Spiel schon zu Ende?
   if ((zeit + naechsteAktion>spieldauer)||(zeit>spieldauer)){
      spiellaeuft = false;
      break;
   }

   // *******************************************************
   // Eine neue Aktion findet statt...
   zeit = zeit + naechsteAktion;
...
```



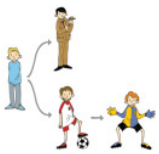
Freie Universität Berlin

Die Klasse Fussballfreundschaftsspiel IV

Jetzt müssen wir entscheiden, wer den Torschuss abgibt, wie stark er schießt und ob der Torwart den Ball hält. Das berechnen wir nach folgender Wahrscheinlichkeitsverteilung.:

1) Wähle eine Mannschaft für eine Torchance aus, als Kriterien dienen Stärke und Motivation der Mannschaften sowie die Erfahrung des Trainers. Nach der Berechnung aller Einflüsse, hat die bessere Mannschaft eine größere Torchance.

```
...
   // Einfluss der Motivation auf die Stärke:
   float staerke_1 = (m1.getStaerke()/2.0f) +
         ((m1.getStaerke()/2.0f) * (m1.getMotivation()/10.0f));
   float staerke_2 = (m2.getStaerke()/2.0f) +
         ((m2.getStaerke()/2.0f) * (m2.getMotivation()/10.0f));

   // Einfluss des Trainers auf die Stärke:
   int abweichung = r.nextInt(2);
   if (staerke_1 > m1.getTrainer().getErfahrung())
      abweichung = -abweichung;
   staerke_1 = Math.max(1, Math.min(10, staerke_1 + abweichung));

   abweichung = r.nextInt(2);
   if (staerke_2 > m2.getTrainer().getErfahrung())
      abweichung = -abweichung;
   staerke_2 = Math.max(1, Math.min(10, staerke_2 + abweichung));
...
```

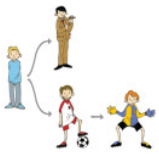


Freie Universität Berlin

Die Klasse Fussballfreundschaftsspiel V

2) Wähle zufällig einen Spieler aus dieser Mannschaft, berechne den Torschuss und gib dem Torwart der anderen Mannschaft die Möglichkeit, diesen Ball zu halten.

```
...
   int schuetze    = r.nextInt(10);

   if ((r.nextInt(Math.round(staerke_1+staerke_2))-staerke_1)<=0){
      Spieler s       = m1.getKader()[schuetze];
      Torwart t       = m2.getTorwart();
      int torschuss   = s.schiesstAufTor();
      // haelt er den Schuss?
      boolean tor     = !t.haeltDenSchuss(torschuss);

      System.out.println();
      System.out.println(zeit+".Minute: ");
      System.out.println("   Chance fuer "+m1.getName()+" ...");
      System.out.println("   "+s.getName()+" zieht ab");

      if (tor) {
         punkteHeim++;
         s.addTor();
         System.out.println("   TOR!!!    "+punkteHeim+":"+
                punkteGast+" "+s.getName()+"("+s.getTore()+")");
      } else {
         System.out.println("   "+m2.getTorwart().getName()
                           +" pariert glanzvoll.");
      }
    }
    else
 ...
```

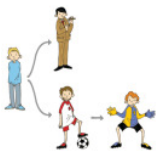


Freie Universität Berlin

Die Klasse Fussballfreundschaftsspiel VI

Teil 2:

```
...
    {
       Spieler s       = m2.getKader()[schuetze];
       Torwart t       = m1.getTorwart();
       int torschuss   = s.schiesstAufTor();
       boolean tor     = !t.haeltDenSchuss(torschuss);

       System.out.println();
       System.out.println(zeit+".Minute: ");
       System.out.println("   Chance fuer "+m2.getName()+" ...");
       System.out.println("   "+s.getName()+" zieht ab");

       if (tor) {
          punkteGast++;
          s.addTor();
          System.out.println("  TOR!!!    "+punkteHeim+":"+
                 punkteGast+" "+s.getName()+"("+s.getTore()+")");
       } else {
          System.out.println("   "+m1.getTorwart().getName()
                            +" pariert glanzvoll.");
       }
    }
    // ************************************************************
  }
}
```



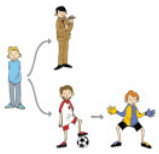
Freie Universität Berlin

Die Klasse Fussballfreundschaftsspiel VII

Es fehlt für die vollständige Implementierung aller Funktionen des Interfaces noch die Methode *getErgebnisText:*

```
   ...
   public String getErgebnisText(){
      return  "Das Freundschaftsspiel endete \n\n"+nameHeimMannschaft
              +" - "+nameGastMannschaft+" "+punkteHeim+":"
              +punkteGast+".";
   }
}
```

Jetzt haben wir alle notwendigen Funktionen implementiert und können schon im nächsten Schritt mit einem Spiel beginnen.

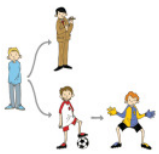


Freie Universität Berlin

Freundschaftsspiel Deutschland-Brasilien der WM-Mannschaften von 2006 I

Nach der ganzen Theorie und den vielen Programmzeilen, können wir uns zurücklehnen und ein, bei der WM nicht stattgefundenes Spiel Deutschland gegen Brasilien, anschauen. Dazu müssen wir zunächst beide Mannschaften definieren und anschließend beide mit der Klasse Fußballfreundschaftsspiel eine Partie spielen lassen.

```
public class FussballTestKlasse{
   public static void main(String[] args){
      // ************************************************************
      // Mannschaft 1
      Trainer t1       = new Trainer("Juergen Klinsmann", 34, 9);
      Torwart tw1      = new Torwart("J. Lehmann", 36, 8, 1, 9, 7);

      Spieler[] sp1    = new Spieler[10];
      sp1[0]           = new Spieler("P. Lahm", 23, 9, 5, 9);
      sp1[1]           = new Spieler("C. Metzelder", 25, 8, 2, 7);
      sp1[2]           = new Spieler("P. Mertesacker", 22, 9, 2, 8);
      sp1[3]           = new Spieler("M. Ballack", 29, 7, 5, 8);
      sp1[4]           = new Spieler("T. Borowski", 26, 9, 8, 9);
      sp1[5]           = new Spieler("D. Odonkor", 22, 7, 5, 8);
      sp1[6]           = new Spieler("B. Schweinsteiger", 22, 2, 3, 2);
      sp1[7]           = new Spieler("L. Podolski", 21, 7, 8, 9);
      sp1[8]           = new Spieler("M. Klose", 28, 10, 9, 7);
      sp1[9]           = new Spieler("O. Neuville", 33, 8, 8, 7);
      // ************************************************************

      ...
```

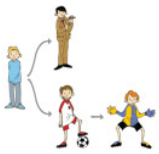


Freie Universität Berlin

Freundschaftsspiel Deutschland-Brasilien der WM-Mannschaften von 2006 II

weiter geht's:

```
        ...
        // ************************************************************
        // Mannschaft 2
        Trainer t2       = new Trainer("Carlos Alberto Parreira", 50, 3);
        Torwart tw2      = new Torwart("Dida", 25, 9, 1, 6, 8);

        Spieler[] sp2    = new Spieler[10];
        sp2[0]           = new Spieler("Cafu", 33, 8, 4, 6);
        sp2[1]           = new Spieler("R. Carlos", 32, 9, 9, 2);
        sp2[2]           = new Spieler("Lucio", 29, 10, 9, 9);
        sp2[3]           = new Spieler("Ronaldinho", 25, 10, 9, 5);
        sp2[4]           = new Spieler("Zé Roberto", 27, 7, 7, 4);
        sp2[5]           = new Spieler("Kaká", 22, 10, 8, 10);
        sp2[6]           = new Spieler("Juninho", 26, 7, 10, 3);
        sp2[7]           = new Spieler("Adriano", 23, 8, 8, 4);
        sp2[8]           = new Spieler("Robinho", 19, 9, 8, 9);
        sp2[9]           = new Spieler("Ronaldo", 28, 4, 10, 2);
        // ************************************************************

        Mannschaft m1 = new Mannschaft("Deutschland WM 2006",t1,tw1,sp1);
        Mannschaft m2 = new Mannschaft("Brasilien WM 2006",t2,tw2,sp2);
        Fussballfreundschaftsspiel f1 = new Fussballfreundschaftsspiel();
        ...
```



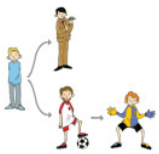
Freie Universität Berlin

Freundschaftsspiel Deutschland-Brasilien der WM-Mannschaften von 2006 III

Wir wollen das Spiel starten und das Ergebnis ausgeben:

```
        ...
        System.out.println("-----------------------------------------");
        System.out.println("Start des Freundschaftspiels zwischen");
        System.out.println();
        System.out.println(m1.getName());
        System.out.println("   Trainer: "+m1.getTrainer().getName());
        System.out.println();
        System.out.println("   und");
        System.out.println();
        System.out.println(m2.getName());
        System.out.println("   Trainer: "+m2.getTrainer().getName());
        System.out.println("-----------------------------------------");

        f1.starteSpiel(m1, m2);

        System.out.println();
        System.out.println("-----------------------------------------");
        System.out.println(f1.getErgebnisText());
        System.out.println("-----------------------------------------");
    }
}
```

Das Spiel ist vorbereitet und die Akteure warten ungeduldig auf den Anpfiff ...

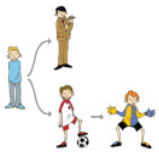


Freie Universität Berlin

Abstrakte Klassen I

Im Vergleich zu einem Interface, bei dem es nur Funktionsköpfe gibt, die implementiert werden müssen, gibt es bei der abstrakten Klasse die Möglichkeit, Funktionen bereits bei der Definition der Klasse zu implementieren.

Eine abstrakte Klasse muss mindestens eine abstrakte Methode (also ohne Implementierung) besitzen. Demzufolge ist das Interface ein Spezialfall der abstrakten Klasse, denn ein Interface besteht nur aus abstrakten Methoden und Konstanten.



Objektorientierte Programmierung mit Java

Freie Universität Berlin

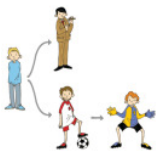
Abstrakte Klassen II

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 die Informationen der Klasse **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;
   }
}
```



Abstrakte Klassen III

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

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

Das schlägt mit der folgenden Ausgabe fehl:

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

Es ist nicht erlaubt von einer abstrakten Klasse eine Instanz zu erzeugen!

Dann erzeugen wir eine Instanz der Klasse **B**:

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

Und erhalten:

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

Objektorientierte Sicht auf die ersten Konzepte

Inhalt:

- Referenzvariablen, Punktnotation, this
- Überladung
- Garbage Collector
- Statische Datentypen, Wrapperklassen
- Die Klasse String

Block M.: "*Java-Intensivkurs - In 14 Tagen lernen Projekte erfolgreich zu realisieren*" , Springer-Verlag 2007
Gries D., Gries P.: "*Multimedia Introduction to Programming Using Java*", Springer-Verlag 2005
Abts D.: „*Grundkurs JAVA: Von den Grundlagen bis zu Datenbank- und Netzanwendungen*", Vieweg-Verlag 2007

Freie Universität Berlin

Referenzvariablen I

Zur Erinnerung, wir haben im vorherigen Abschnitt die Klasse **Person** implementiert. Hier noch einmal der Programmcode dieser Klasse:

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

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

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


Objektorientierte Sicht auf die ersten Konzepte

Freie Universität Berlin

Referenzvariablen II

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

```
Person p1;
p1 = new Person("Hugo", 12);
Person p2;
p2 = p1;
if (p1 == p2)
   System.out.println("Die Referenzen sind gleich");
```

In der ersten Zeile wird p1 deklariert. p1 ist eine Referenzvariable und beinhaltet eine Adresse. Diese Adresse ist momentan nicht gegeben, sollten wir versuchen auf p1 zuzugreifen, würde Java einen Fehler beim Kompilieren mit der Begründung verursachen: „Variable p1 ist nicht initialisiert“. Die zweite Zeile stellt nun aber einen Speicherplatz bereit und erzeugt ein Objekt der Klasse **Person** und vergibt den Attributen gleich Werte. p1 zeigt nun auf diesen Speicherbereich.

Die dritte Zeile verhält sich äquivalent zur ersten. In Zeile vier weisen wir aber nun 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!

Freie Universität Berlin

Zugriff auf Attribute und Methoden durch Punktnotation I

Wir haben diese Syntax bereits schon mehrfach verwendet, aber nun werden wir es konkretisieren. Existiert eine Referenz auf ein Objekt, so können wir mit einem Punkt nach der Referenz und dem Namen des entsprechenden Attributs (das nennen wir dann Instanzvariable) bzw. der entsprechenden Methode (analog Instanzmethode) auf diese zugreifen.

Referenz.Attribut
Referenz.Methode

Dazu schauen wir uns ein kleines Beispiel an und werden den Unterschied zwischen Variablen, die primitive Datentypen repräsentieren und Variablen, die Referenzen repräsentieren, erläutern.

```
int a    = 2;
Person p = new Person("Hans", 92);
// An dieser Stelle hat p.getName() den Rückgabewert "Hans"
komischeFunktion(a, p);
```

Wir werden nun a und p an eine Funktion übergeben. Für den primitiven Datentypen a gilt die Übergabe , "call by value". Das bedeutet, dass der Inhalt, also die in die Funktion übergeben wird. Anders sieht es bei dem Referenzdatentyp aus, hier wird die Referenz, also die Adresse übergeben, wir nennen das „call by reference".

Objektorientierte Sicht auf die ersten Konzepte

Freie Universität Berlin

Zugriff auf Attribute und Methoden durch Punktnotation II

Hier nochmal der Aufruf:

```
int a    = 2;
Person p = new Person("Hans", 92);
// An dieser Stelle hat p.getName() den Rückgabewert "Hans"
komischeFunktion(a, p);
```

und die „komische Funktion":

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

Sollten wir nach Ausführung der Zeile *komischeFunktion(a, p);* wieder den Rückgabewert von p erfragen, so erhalten wir „Gerd". Die Variable a bleibt indes unangetastet.

Freie Universität Berlin

Referenzvariable this

Wir haben schon in einem vorhergehendenAbschnitt gesehen, wie sich die Instanzvariablen von „außen“ setzen lassen. Jedes Objekt kann aber auch mittels einer Referenzvariablen auf sich selbst reflektieren 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;
   }
}
```

Wir können mit der Referenzvariablen this auf „unsere“ Instanzvariablen und -methoden zugreifen.

Objektorientierte Sicht auf die ersten Konzepte

Freie Universität Berlin

Prinzip des Überladens I

Oft ist es sinnvoll eine Funktion für verschiedene Eingabetypen zur Verfügung zu stellen. Damit wir nicht jedes Mal einen neuen Funktionsnamen wählen müssen, gibt es die Möglichkeit der Überladung in Java. Wir können einen Funktionsnamen mehrfach verwenden, falls sich die Signatur der Funktionen eindeutig unterscheiden lässt. Als Signatur definieren wir die Kombination aus *Rueckgabewert*, *Funktionsname* und *Eingabeparameter*, wobei nur der Typ der Eingabe und nicht die Bezeichnung entscheidend ist.

Schauen wir uns ein Beispiel an:

```
// Max-Funktion für Datentyp int
int max(int a, int b){
   if (a<b) return b;
   return a;
}

// Max-Funktion für Datentyp double
double max(double a, double b){
   ... // analog zu der ersten Funktion
}
```

Beim Aufruf der Funktion *max* wird die passende Signatur ermittelt und diese Funktion entsprechend verwendet. Wir können uns merken, trotz Namensgleichheit sind mehrere Funktionen mit verschiedenen Signaturen erlaubt.

Freie Universität Berlin

Prinzip des Überladens II

Dieses Prinzip lässt sich auch auf Konstruktoren übertragen:

```
public class Person{
   private String name;
   private int alter;
   // Konstruktor 1
   public Person(){
      name       = "";   // z.B. als Defaultwert
      alter      = 1;    // z.B. als Defaultwert
   }
   // Konstruktor 2
   public Person(String name){
      this.name  = name;
      alter      = 1;    // z.B. als Defaultwert
   }
   // Konstruktor 3
   public Person(String name, int alter){
      this.name  = name;
      this.alter = alter;
   }
}
```

Nun aber ein Beispiel zu der neuen **Person-Klasse**:

```
Person p1 = new Person();
Person p2 = new Person("Herbert", 30);
```

Die erste Zeile verwendet den passenden Konstruktor (Konstruktor 1) mit der parameterlosen Signatur und die zweite verwendet den dritten Konstruktor.

Freie Universität Berlin

Prinzip des Überladens III

Kleiner Hinweis: Genau dann, wenn kein Konstruktor angegeben sein sollte, denkt sich Java den Defaultkonstruktor hinzu, der keine Parameter erhält und keine Attribute setzt. Es kann trotzdem ein Objekt dieser Klasse erzeugt werden.

Freie Universität Berlin

Der Copy-Konstruktor

Bei der Übergabe von Objekten, werden diese nicht kopiert, sondern lediglich die Referenz übergeben, das wissen wir bereits. Wir müssen also immer daran denken, eine lokale Kopie des Objekts vorzunehmen.

Elegant lässt sich das mit dem Copy-Konstruktor lösen:

```
public class CopyKlasse{
   public int a, b, c, d;

   public CopyKlasse(){
      a=0, b=0, c=0, d=0;
   }

   public CopyKlasse(CopyKlasse ck){
      this.a = ck.a;
      this.b = ck.b;
      this.c = ck.c;
      this.d = ck.d;
   }
}
```

Zuerst erzeugen wir eine Instanz der Klasse **CopyKlasse** und anschließend erstellen wir eine Kopie:

```
CopyKlasse a = new CopyKlasse();
CopyKlasse copyVonA = new CopyKlasse(a);
```


Freie Universität Berlin

Garbage Collector

Bisher haben wir andauernd Speicher für unsere Variablen reserviert, aber wann wird dieser Speicher wieder frei gegeben? Java besitzt mit dem Garbage Collector eine, „Müllabfuhr“, die sich automatisch um die Freigabe des Speichers kümmert.

Das ist gegenüber anderen Programmiersprachen ein Komfort. Dieser Komfort ist aber nicht umsonst, Java entscheidet eigenständig den Garbage Collector zu starten. Es könnte also auch dann geschehen, wenn gerade eine performancekritische Funktion gestartet wird.