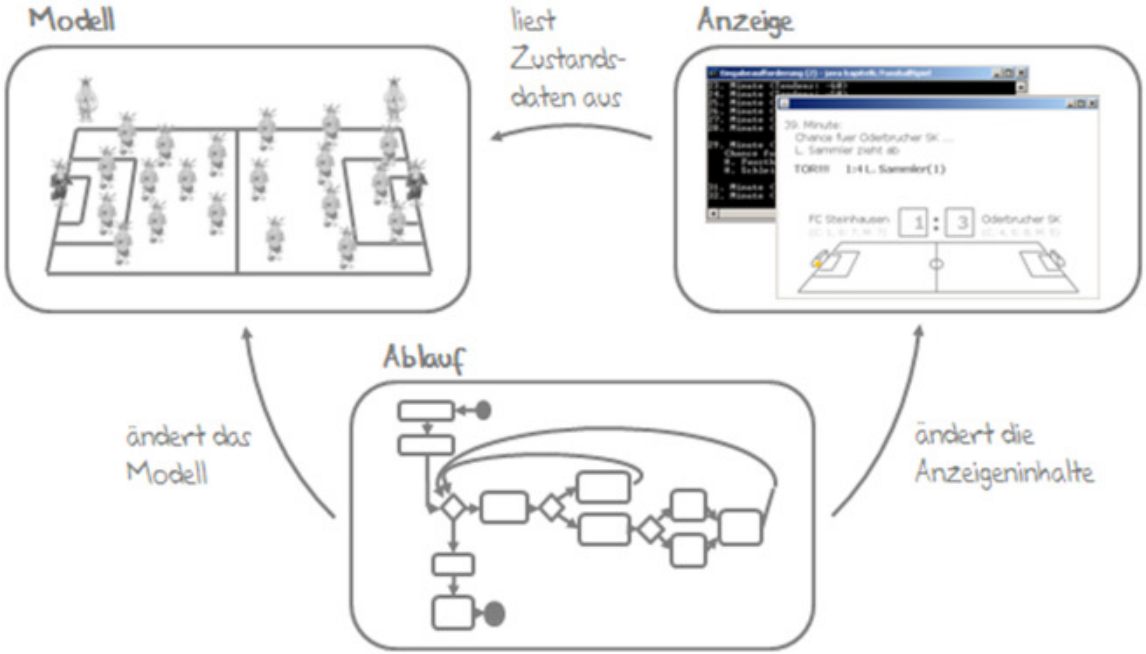
Modularisierung in Ablauf, Modell und Anzeige

Wir haben viele kleine Programmabschnitte im Aktivitätsdiagramm definiert und haben jetzt eine grobe Vorstellung von der Funktionsweise des Programms. Jetzt müssen wir diese in Klassen unterbringen.

Eine Möglichkeit wäre sicherlich, alle Methoden in einer Klasse Fussballspiel abzulegen, aber aus Entwurfssicht ist das keine gute Lösung.

Eine bessere Lösung ist sicherlich:



Das gewählte Entwurfkonzept ist auf Grund der fehlenden Interaktion mit dem Benutzer eine vereinfachte Variante des Model-View-Controller-Ansatzes (MVC).

Implementierung des Ablaufs I

Fangen wir in top-down-Manier damit an, zunächst den Ablauf zu implementieren und gehen davon aus, dass Anzeige und Modell entsprechende Funktionen bereitstellen:

```
import java.io.FileNotFoundException;

public class Fussballspiel {
   private FussballspielModell   modell  = null;
   private FussballspielAnzeige  anzeige = null;

   public FussballspielStandard() {
      modell  = new FussballspielModell();
      anzeige = new FussballspielAnzeigeKonsole(modell);
   }

   public static void warte(int ms) {
      try {
         Thread.sleep(ms);
      } catch(InterruptedException e) {
         e.printStackTrace();
      }
   }
   ...
```

Implementierung des Ablaufs II

weiter gehts:

```
...
public void starteSpiel(String m1, String m2) throws FileNotFoundException {
   modell.initSpiel(m1, m2);
   anzeige.zeigeIntro();
   warte(1000);

   // gameloop
   while (modell.getSpielLaeuft()) {
      if (modell.aktTendenz()) {
         anzeige.zeigeTorchance();
         warte(1500);
         if (modell.torschussErfolgreich())
            anzeige.zeigeTor();
         else
            anzeige.zeigeParade();
         warte(1500);
         modell.resetTendenz();
      } else
         anzeige.zeigeMinute();
      warte(800);
   }
   warte(1500);
   modell.resetTendenz();
   anzeige.zeigeErgebnis();
}
...
```

Implementierung des Ablaufs III

und schließlich:

```
   ...
   public static void main(String[] args) {
      Fussballspiel fc = new Fussballspiel();

      try {
         fc.starteSpiel("steinhausen.txt", "oderbruch.txt");
      } catch(FileNotFoundException e) {
         e.printStackTrace();
      }
   }
}
```

In der main-Funktion wird ein Fussballspiel erstellt. Die Funktion starteSpiel ist das Herzstück dieser Klasse und repräsentiert eine Umsetzung der Programmschleife.

Bedingung für einen Spielstart ist das erfolgreiche Einlesen zweier Mannschaften. Sollte es Probleme beim Einlesen geben, wird ein Fehler geworfen und ausgegeben.

Implementierung des Modells I

In dem Modell wollen wir die aktuellen Spieldaten zum Schreiben und Lesen bereithalten:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class FussballspielModell {
   private Mannschaft heim, gast;
   private byte toreHeim, torSchuesseHeim, toreGast, torSchuesseGast, spieldauer, aktZeit, tendenz;
   private final byte TENDENZ_MAX = 10;
   private boolean spielLaeuft;

   private Mannschaft aktMannschaft;
   private Spieler    aktSpieler;
   private Torwart    aktTorwart;

   public void initSpiel(String m1, String m2) throws FileNotFoundException {
      heim = liesMannschaft(m1);
      gast = liesMannschaft(m2);

      toreHeim        = 0;      toreGast        = 0;
      torSchuesseHeim = 0;      torSchuesseGast = 0;
      aktMannschaft   = null;
      aktSpieler      = null;
      aktTorwart      = null;
      spielLaeuft     = false;
      spieldauer      = 0;      tendenz         = 0;

      spielLaeuft     = true;
      // 90 Minuten + mögliche Nachspielzeit
      spieldauer      = (byte)(90 + (byte)(Math.random()*5));
   }
   ...
```

Implementierung des Modells II

Es folgen für alle Attribute entsprechende get-Methoden, die später vom Spielablauf und der Anzeige ausgelesen werden können.

Der Parameter tendenz wird mit 0 initialisiert, pro Minute aktualisiert und repräsentiert die aktuelle Torchancetendenz für beide Mannschaften. Ist der Wert positiv, dann steigt die Wahrscheinlichkeit für die Gastmannschaft. Bei einem negativen Wert, steigt die Wahrscheinlichkeit entsprechend für die Heimmannschaft. Ist der Wert größer TENDENZ_MAX oder kleiner -TENDENZ_MAX, erhält die jeweilige Mannschaft eine Torchance.

Für die visuelle Auswertung liefert die Funktion getTendenz in Anhängigkeit zur Konstanten TENDENZ_MAX die prozentuelle Tendenz:

```
public byte getTendenz() {
   byte val = (byte)(Math.min(TENDENZ_MAX,
                    Math.max(-TENDENZ_MAX, tendenz)));
   return (byte)((val*100.0)/TENDENZ_MAX);
}
```

Da die tendenz zwischenzeitlich ausserhalb der Grenzen liegen kann, werden diese in val überprüft. Nach jeder Torchance wird die tendenz mit resetTendenz wieder auf 0 gesetzt.

Implementierung des Modells III

Beginnen wir jetzt mit dem Einlesen der Mannschaftsdaten. Hier sehen wir beispielsweise den Inhalt der Datei steinhausen.txt:

```
C:\>type steinhausen.txt
FC Steinhausen
Paul Steinwerfer, 39, 7
H. Schleifer, 22, 8, 1, 9, 7
A. Baumfaeller, 23, 9, 5, 9
F. Feuerstein, 25, 8, 2, 7
P. Knochenbrecher, 22, 9, 2, 8
M. Holzkopf, 29, 7, 5, 8
B. Geroellheimer, 26, 9, 8, 9
D. Bogenbauer, 22, 7, 5, 8
B. Schnitzer, 22, 2, 3, 2
L. Schiesser, 21, 7, 8, 9
M. Klotz, 28, 10, 9, 7
O. Mammut, 33, 8, 8, 7
```

Die vorliegende Struktur der Daten ist allerdings etwas komplizierter als unser Einstiegsbeispiel und daher ist es notwendig, die einzelnen Zeilen in ihre Wortbestandteile zu zerlegen.

Einlesen und Zerlegen von Text

```
private Mannschaft liesMannschaft(String dateiName) throws FileNotFoundException {
   Scanner scanner = new Scanner(new File(dateiName));
   String zeile = null;
   int anzZeilen = 0;
   String name  = null;
   Trainer t    = null;
   Torwart tw   = null;
   Spieler[] sp = new Spieler[10];

   while (scanner.hasNextLine()) {
      zeile = scanner.nextLine();
      anzZeilen++;

      if (anzZeilen == 1) {      // Mannschaftsname auslesen
         name = zeile;
         continue;
      }

      // Trainer, Torwart und Spieler auslesen
      String[] worte = zeile.split(", ");
      if (anzZeilen == 2) {                  // Trainer einlesen
         t = new Trainer(worte[0], Integer.parseInt(worte[1]), Integer.parseInt(worte[2]));
      } else if (anzZeilen == 3) {           // Torwart einlesen
         tw = new Torwart(worte[0], Integer.parseInt(worte[1]), Integer.parseInt(worte[2]),
                          Integer.parseInt(worte[3]), Integer.parseInt(worte[4]),
                          Integer.parseInt(worte[5]));
      } else {                               // Spieler einlesen
         sp[anzZeilen-4] = new Spieler(worte[0], Integer.parseInt(worte[1]),
                          Integer.parseInt(worte[2]), Integer.parseInt(worte[3]),
                          Integer.parseInt(worte[4]));
      }
   }
   scanner.close();
   return new Mannschaft(name, t, tw, sp);
}
```

Torschussverhalten

Das Programmverhalten bei einem Torschuss wird durch die Funktion torschussErfolgreich ermittelt:

```
public boolean torschussErfolgreich() {
   if (!aktTorwart.pariertTorschuss(aktSpieler.schiesstAufsTor())) {
      aktSpieler.addTor();
      if (aktMannschaft == heim)
         toreHeim++;
      else
         toreGast++;
      aktZeit+=2;
      return true;
   }
   aktZeit++;
   return false;
}
```

Der aktuell ausgewählte Spieler gibt einen Torschuss ab und der gegnerische Torwart versucht den Torschuss zu parieren. Kann der Torwart den Ball nicht halten, schreiben wir dem Schützen und der Mannschaft ein Tor gut.

Wir berechnen noch pauschal zwei Minuten für die Torfeier samt Aufstellung zum Wiederanstoss. Wird das Tor nicht getroffen, zählen wir nur eine Minute dazu.

Torchancetendenz I

Kommen wir zu der wichtigsten Funktion, die entscheiden wird, zu Gunsten welcher Mannschaft sich die Torchancetendenz verändern wird:

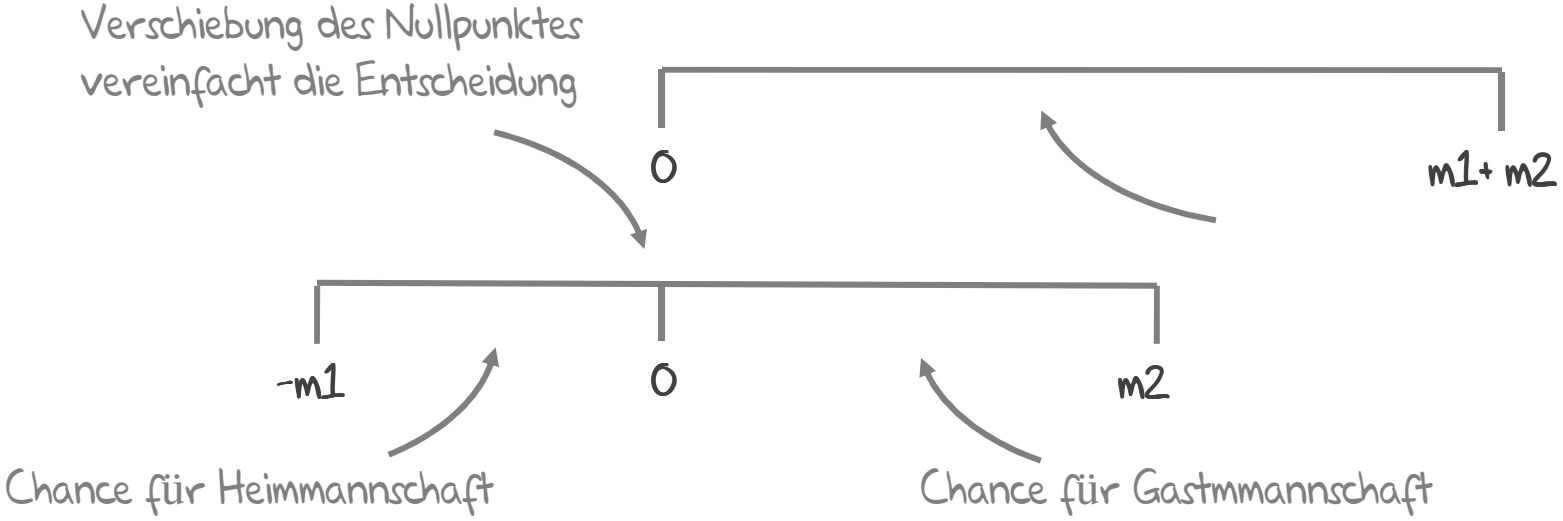
```
public boolean aktTendenz() {
     aktZeit++;
     if (aktZeit >= spieldauer) {
        spielLaeuft = false;
        return false;
     }

     double m1 = 0.8  * heim.getStaerke()    + 0.15 * heim.getMotivation() +
                 0.05 * heim.getTrainer().getErfahrung();
     double m2 = 0.8  * gast.getStaerke()    + 0.15 * gast.getMotivation() +
                 0.05 * gast.getTrainer().getErfahrung();
     tendenz += (byte)(Math.random() * (m1+m2) - m1);

     if (tendenz<=-TENDENZ_MAX) {
        // Heimmannschaft erhält die Torschussmöglichkeit
        aktMannschaft = heim;
        aktSpieler    = heim.getFeldspieler()[(int)(Math.random()*10)];
        aktTorwart    = gast.getTorwart();
        torSchuesseHeim++;
        return true;
     } else if (tendenz>=TENDENZ_MAX) {
        // Gastmannschaft erhält die Torschussmöglichkeit
        aktMannschaft = gast;
        aktSpieler    = gast.getFeldspieler()[(int)(Math.random()*10)];
        aktTorwart    = heim.getTorwart();
        torSchuesseGast++;
        return true;
     }
     return false;
  }
}
```

Torchancetendenz II

Solange wir noch innerhalb der Spielzeit sind, errechnen wir mit m1 und m2 für beiden Mannschaften jeweils einen Wert, der die aktuelle Stärke wiederspiegeln soll.

Dieser ergibt sich zu 80% aus der durchschnittlichen Spielerstärke, zu 15% aus der durchschnittlichen Spielermotivation und zu 5% aus der Erfahrung des Trainers.

Wir ermitteln jetzt eine Zufallszahl aus dem Intervall [0, m1+m2] und verschieben den Nullpunkt anschließend um m1 nach rechts:

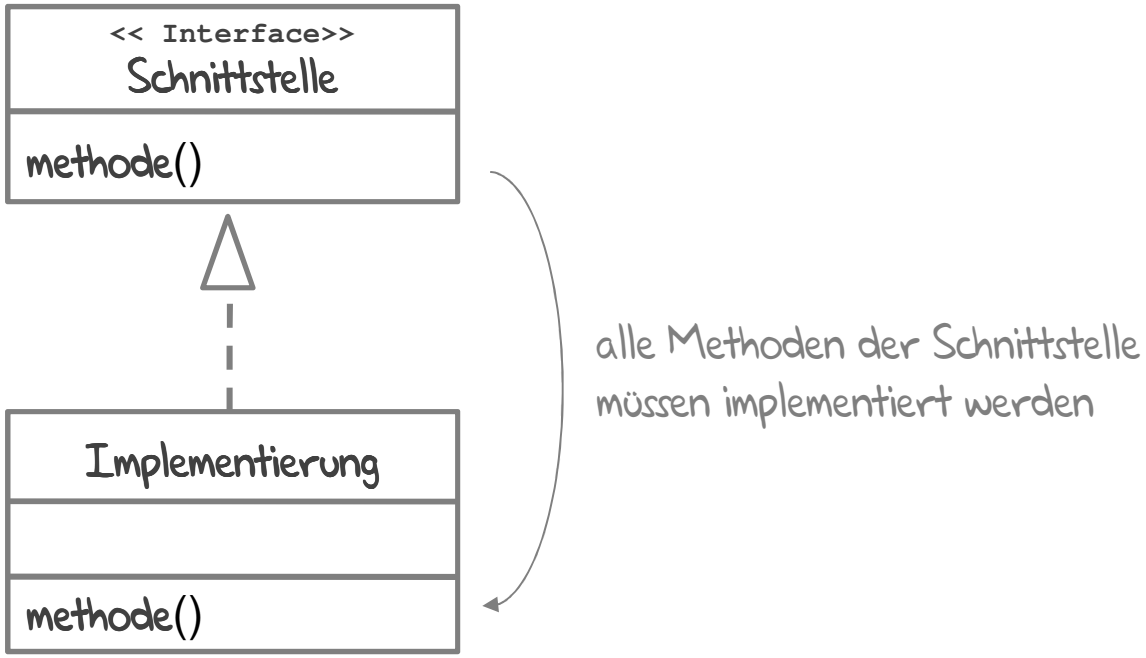


Jetzt genügt es zu überprüfen, ob die Zufallszahl kleiner oder größer 0 ist. Je dominierender eine Mannschaft ist, desto wahrscheinlicher ist das Auftreten der Zufallszahl in dem größeren Intervallbereich.

Konsolenvisualisierung über ein Interface

Vorher wurde erwähnt, dass es sinnvoll ist, die Visualisierung vom Rest des Projekts zu trennen, damit dieser Teil einfacher austauschbar ist. Eine Möglichkeit wäre, dass wir uns auf dem Papier notieren, welche Methoden mit entsprechenden Signaturen eine Anzeige enthalten muss, damit wir diese austauschen können.

Glücklicherweise gibt es in Java den Vererbungsmechanismus Interface, mit dem das sehr einfach umzusetzen ist:



Ein Interface definieren

Wir beschreiben mit dem Schlüsselwort interface eine Sammlung von Methoden, die jeder implementieren muss, der dazu gehören möchte:

```
public interface <Bezeichnung> {
    <Methodensignatur1>;
    <Methodensignatur2>;
    ...
}
```

In der Klasse Fussballspiel haben wir bereits ein paar Methodenaufrufe gesehen, die wir noch implementieren müssen.

Das dazugehörige Interface FussballspielAnzeige sieht entsprechend so aus:

```
public interface FussballspielAnzeige {
   public void zeigeIntro();
   public void zeigeMinute();
   public void zeigeTorchance();
   public void zeigeTor();
   public void zeigeParade();
   public void zeigeErgebnis();
}
```

Jeder Programmierer einer konkreten FussballspielAnzeige muss diese Methoden in seiner Klasse implementieren. Es können mehr dazukommen, es darf aber keine fehlen.

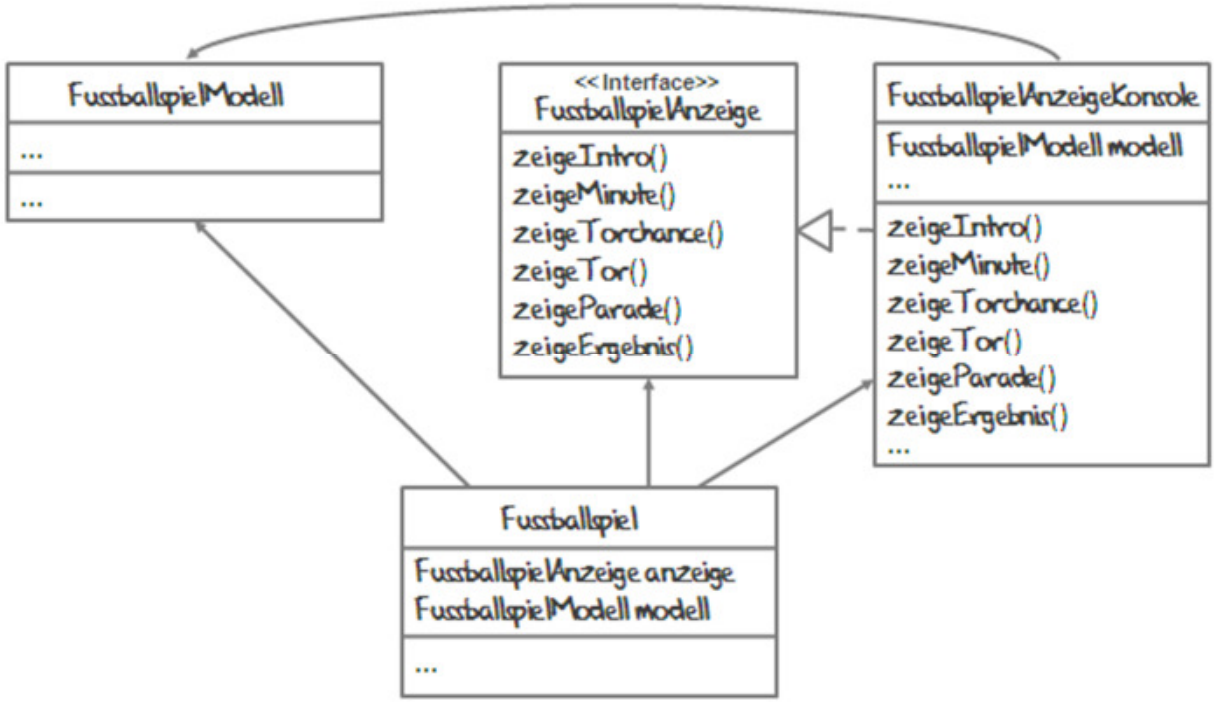
Ein Interface implementieren

Da es sich ebenfalls um einen Vererbungsmechanismus handelt, müssen wir in der neuen Klasse das zu implementierende Interface angeben. Die Syntax mit dem Schlüsselwort implements sieht dafür wie folgt aus:

```
public class <Klassename> implements <Interface> {
    <Methodensignatur1> {
        ...
    }

    <Methodensignatur2> {
        ...
    }
    ...
}
```

MVC mit Interface für Anzeige

Jetzt müssen alle Methoden einen Implementierungsteil haben:



Konsolenanzeige implementieren I

Als erstes Beispiel werden wir gleich das Interface FussballspielAnzeige mit einer einfachen Visualisierung auf der Konsole implementieren:

```
public class FussballspielAnzeigeKonsole implements FussballspielAnzeige {
   private FussballspielModell model;

   public FussballspielAnzeigeKonsole(FussballspielModell m) {
      model = m;
   }

   public void zeigeIntro() {
      System.out.println("-------------------------------------------------");
      System.out.println("Freundschaftspiel startet zwischen: ");
      System.out.println(model.getNameHeim()+" und "+model.getNameGast());
      System.out.println("-------------------------------------------------");
   }

   public void zeigeMinute() {
      System.out.println(model.getAktZeit()+". Minute (Tendenz: "+model.getTendenz()+")");
   }

   public void zeigeTorchance() {
      System.out.println();
      System.out.println(model.getAktZeit()+". Minute (Tendenz: "+model.getTendenz()+"):");
      System.out.println("   Chance fuer "+model.getAktMannschaftName() + " ...");
      System.out.println("   " + model.getAktSpielerName() + " zieht ab");
   }
   ...
```

Konsolenanzeige implementieren II

Im Konstruktor übergeben wir eine Referenz auf das Modell, damit die Anzeige auf die entsprechenden get-Methoden zugreifen kann.

Das Modell kennt die Anzeige allerdings nicht.

```
   public void zeigeTor() {
      System.out.println("   TOR!!!     " + model.getToreHeim() + ":" +
         model.getToreGast() + " " + model.getAktSpielerName() + "("  +
         model.getAktSpielerTore() + ")");
      System.out.println();
   }

   public void zeigeParade() {
      System.out.println("   " + model.getAktTorwartName() + " pariert glanzvoll.");
      System.out.println();
   }

   public void zeigeErgebnis() {
      System.out.println();
      System.out.println("-------------------------------------------------");
      System.out.println("Das Freundschaftsspiel endete:");
      System.out.println(model.getNameHeim()+
         " "+model.getToreHeim()+":"+model.getToreGast()+
         " "+model.getNameGast());
      System.out.println("-------------------------------------------------");
   }
}
```

Freundschaftsspiel FC Steinhausen-Oderbucher SK I

Nach der ganzen Theorie und den vielen Programmzeilen, können wir uns endlich ein wenig zurücklehnen und ein Derby anschauen, dass sich so oder so ähnlich in der Steinzeit zugetragen haben könnte (die Ausgabe wurde in den ausgeglichenen Phasen etwas gekürzt):

```
C:\>java Fussballspiel
--------------------------------------------------
Freundschaftspiel startet zwischen:
FC Steinhausen und Oderbrucher SK
--------------------------------------------------
1. Minute (Tendenz: -30)
2. Minute (Tendenz: -40)
...
18. Minute (Tendenz: 30)

19. Minute (Tendenz: 100):
   Chance fuer Oderbrucher SK ...
   R. Birkenpech zieht ab
   H. Schleifer pariert glanzvoll.

21. Minute (Tendenz: 0)
...
31. Minute (Tendenz: -80)

32. Minute (Tendenz: -100):
   Chance fuer FC Steinhausen ...
   B. Geroellheimer zieht ab
   T. Faenger pariert glanzvoll.

34. Minute (Tendenz: 60)
...
41. Minute (Tendenz: 30)

42. Minute (Tendenz: 100):
   Chance fuer Oderbrucher SK ...
   L. Sammler zieht ab
   TOR!!!     0:1 L. Sammler(1)
```

Freundschaftsspiel FC Steinhausen-Oderbucher SK II

Halbzeitpfiff - Ob dem FC Steinhausen der Ausgleich oder sogar der Spielsieg noch gelingt, sehen wir in der zweiten Halbzeit:

```
45. Minute (Tendenz: 30)
...
60. Minute (Tendenz: -90)

61. Minute (Tendenz: -100):
   Chance fuer FC Steinhausen ...
   D. Bogenbauer zieht ab
   T. Faenger pariert glanzvoll.

63. Minute (Tendenz: 70)
...
69. Minute (Tendenz: 90)

70. Minute (Tendenz: 100):
   Chance fuer Oderbrucher SK ...
   K. Zahnlos zieht ab
   H. Schleifer pariert glanzvoll.

72. Minute (Tendenz: -40)
...
86. Minute (Tendenz: 60)

87. Minute (Tendenz: 100):
   Chance fuer Oderbrucher SK ...
   K. Zahnlos zieht ab
   TOR!!!     0:2 K. Zahnlos(1)

90. Minute (Tendenz: 60)
91. Minute (Tendenz: 0)
92. Minute (Tendenz: 0)

------------------------------------------------
Das Freundschaftsspiel endete:
FC Steinhausen 0:2 Oderbrucher SK
------------------------------------------------
```

Obwohl es das Spiel der Torhüter hätte werden können, trafen die Derbyspezialisten L. Sammler und K. Zahnlos der doch etwas stärkeren Gastmannschaft und sicherten damit den Sieg.

Live-Coding-Session

Ausblick

Unser Konzept ermöglicht es uns, später die Ausgabe (wenn wir Kenntnisse zu visuellen Ausgabe erlangt haben) einfach auszutauschen:

Vorlesungsteil

Alles ist objektorientiert! Kurzer Rückblick

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

Übersicht zum Vorlesungsinhalt

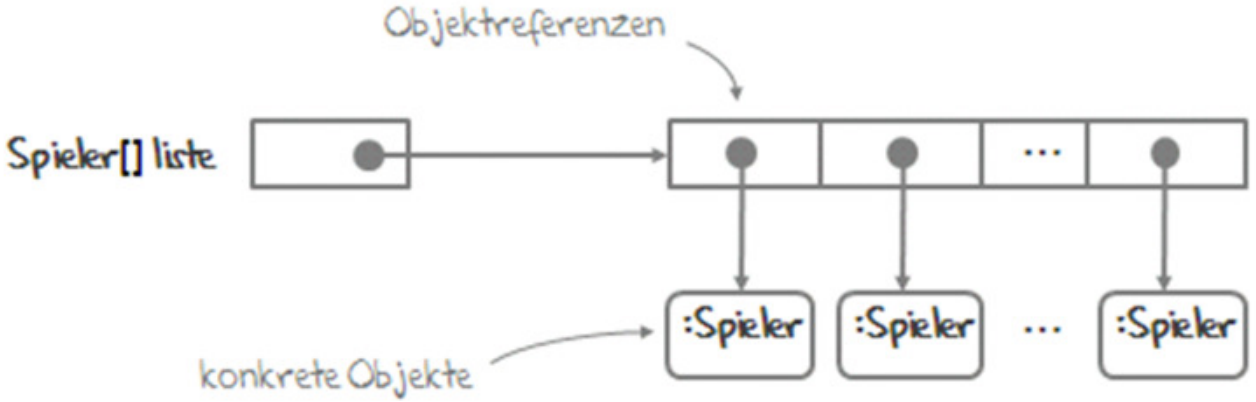
zeitliche Abfolge und Inhalte können variieren

Alles ist objektorientiert! Kurzer Rückblick

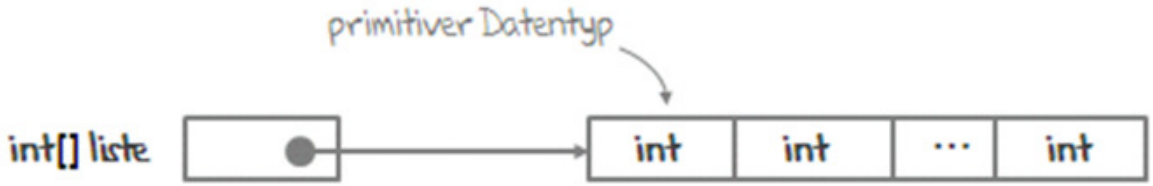
- Reservierung von Speicher
- Das clone-Schaf Dolly
- Alles erbt von Object
- Überschreiben von Methoden
- Doubleliste
- Äqzuivalenzrelationen
- Eigene Objektausgaben
- Wrapperklassen
- Referenzvariablen, this
- Zugriffsmodifikatoren
- Eigene Exception-Klasse erstellen
- getMessage/printStackTrace
- 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:

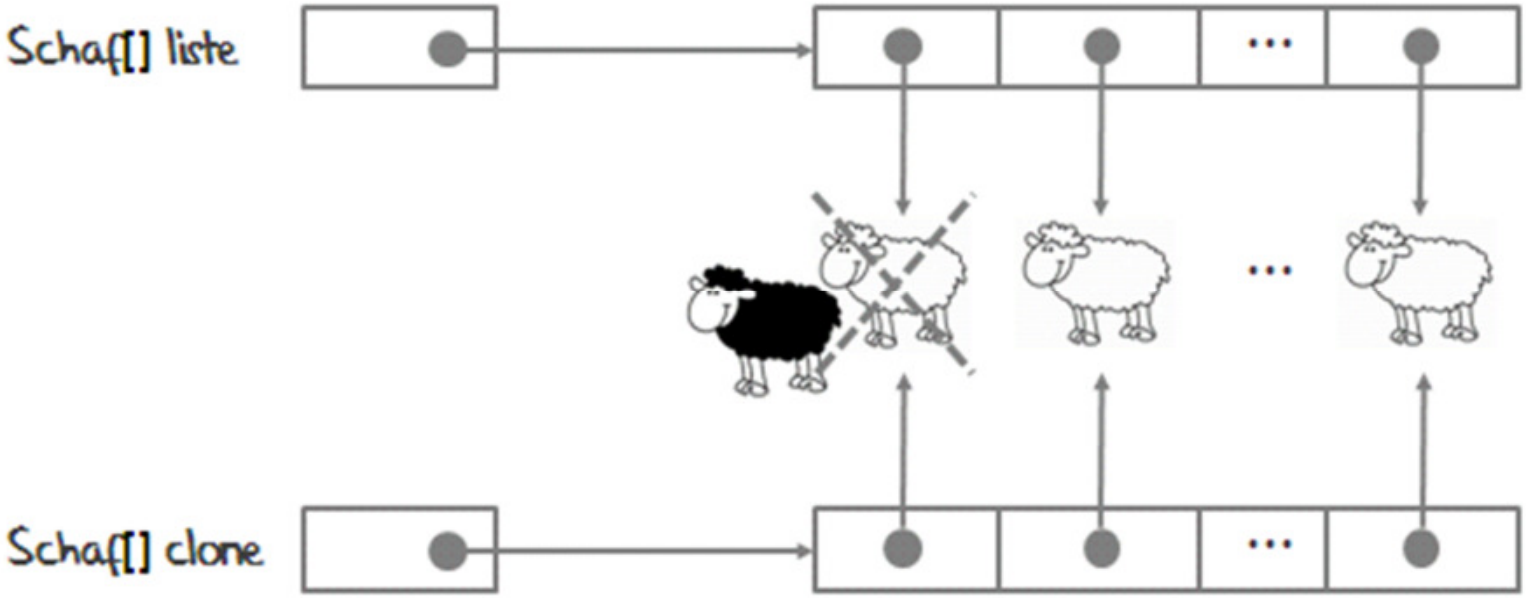
```
Schaf[] liste = new Schaf[4];
liste[0] = new Schaf("Bernd", "weiß");
liste[1] = new Schaf("Susi",  "weiß");
liste[2] = new Schaf("Hanni", "weiß");
liste[3] = new Schaf("Klaus", "weiß");

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.