Inhalt

9 Klassen ... 9-2
9.1 Instanzvariablen und Instanzmethoden ... 9-2
9.1.1 Vereinbarung ... 9-2
9.1.2 this ... 9-5
9.1.3 toString() ... 9-5
9.2 Klassenvariablen und Klassenmethoden ... 9-7
9.2.1 Vereinbarung ... 9-7
9.2.2 super ... 9-10
9.3 Klassenkonstanten ... 9-11
9.4 Instanziierung und Initialisierung ... 9-13
9.4.1 Konstruktoren ... 9-13
9.4.2 Beispiel „Boot“ ... 9-15
9.4.3 Der Mechanismus der Objekterzeugung ... 9-22
9.5 Pakete ... 9-23
9.5.1 Erstellen von Paketen ... 9-23
9.5.2 Installation komprimierter Pakete ... 9-24
9.6 Zugriffsrechte ... 9-25

9 Klassen

9.1 Instanzvariablen und Instanzmethoden

9.1.1 Vereinbarung

Instanzvariablen und Instanzmethoden sind die *objektspezifischen* Bestandteile einer Klasse.

Deklaration und Definition

Instanzvariablen wurden schon im Kapitel über Referenzdatentypen - Klassen eingeführt. Sie fassen die *Daten eines Objekts* zusammen und sind in der Regel *nicht öffentlich*.

private *Typ Variablenname [= Ausdruck]* **;**

Instanzmethoden werden zur *Datenbereitstellung* und zur *Datenmanipulation des Objektes* aufgenommen. Sie sollten *nur dann* öffentlich sein, wenn sie als Dienstleistung von anderen Objekten angefordert werden.

public / private *Ergebnistyp Methodenname* **(** *Parameterliste* **)** // Methodenkopf
{
Anweisungen // Methodenrumpf
}

Zugriff

Da die Instanzvariablen und Instanzmethoden zum Objekt gehören, werden sie auch über dieses mittels **Punktoperator** aufgerufen, vorausgesetzt das Zugriffsrecht ist gegeben.

Objektname **.** *Variablenname*
Objektname **.** *Methodenname* (*Argumentenliste*)

Ein typisches Beispiel für die Verwendung von Instanzmethoden wurde bereits mit der Klasse `java.lang.String` besprochen. Der Vergleich zweier Zeichenketten `s1` und `s2` als Objekte dieser Klasse wird durch `s1.equals( s2)` aufgerufen. `equals` ist eine Instanzmethode der Klasse `String`. Der Aufruf ist im Sinne der Objektorientierung so zu verstehen: Das Objekt `s1` vergleicht sich mit dem Argument der Methode, dem Objekt `s2`.

In der abstrakten Klasse `Boot` wurden eine Instanzvariable `name`, drei Instanzmethoden `setName`, `getName` und `getTyp` und eine abstrakte Instanzmethode `bremsen` definiert.

Boot.java

```
// Boot.java                                      MM 2005

public abstract class Boot
{
  private String name;                  // Instanzvariable

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

```
    public String getName()
    {
      return this.name;
    }

    public String getTyp()
    {
      return this.getClass().getName();
    }

    public abstract void bremsen( int delta);
  }
```

Von einer abstrakten Klasse können keine Objekte instanziiert werden. Deshalb betrachten wir ein Objekt einer Klasse `PaddelBoot`. Diese Klasse sei von der Klasse `Boot` abgeleitet, erbt sowohl die Instanzvariable als auch die Instanzmethoden und überschreibt die abstrakte Methode `bremsen`. Sie hat zusätzlich noch eine Instanzvariable `paddel`, welche die Anzahl der Paddel festhält und zwei Methoden `setPaddel` und `getPaddel` als Zugriff auf die Instanzvariable.

PaddelBoot.java

```
  // PaddelBoot.java                                MM 2005

  public class PaddelBoot extends Boot
  {
    private int paddel;                // Instanzvariable

    public void setPaddel( int paddel) // Instanzmethoden
    {
      this.paddel = paddel;
    }

    public int getPaddel()
    {
      return this.paddel;
    }

    public void bremsen( int delta)
    {
      System.out.println
      ( "Boot " + this.getName() + ": Paddel einziehen!");
    }
  }
```

In einem Test bilden wir ein Objekt und legen dessen Namen und Paddelanzahl fest, erfragen den Namen und bremsen das Paddelboot.

TestBoot.java

```
  // TestBoot.java                                  MM 2005

  public class TestBoot
  {
```

```
    public static void main( String[] args)
    {
  // Objektbildung
      PaddelBoot swen = new PaddelBoot();

  // set-Methoden
      swen.setName( "Swen");
      swen.setPaddel( 1);

  // Bremsen
      swen.bremsen( 5);
    }
  }
```

Als Konsolenausgaben erhalten wir:

TestBoot.out

```
  Boot Swen: Paddel einziehen!
```

Der Zugriff auf alle Instanzvariablen in unserem Beispiel erfolgt immer über entsprechende Methoden. Ein *direkter* Zugriff wäre *nicht* möglich, da diese *nicht öffentlich* sind.

Die folgende Anweisung liefert einem *Zugriffsfehler*:

```
  swen.name = "Swen";
  name has private access in Boot
```

Validierung von Eingaben

Instanzmethoden werden zur Validierung von Eingaben genutzt. Ein Vorteil der Eingabe von Werten über Methoden besteht darin, dass man von vornherein *Fehleingaben* abfangen kann. Auf diese Weise werden den Instanzvariablen nur *zugelassene Werte* zugewiesen.
In unserem Fall könnte man darauf achten, dass alle Bootsnamen in Großbuchstaben abgespeichert werden. Deshalb wird in der Methode `setName` dafür gesorgt. Die Instanzmethode `toUpperCase` der Klasse `java.lang.String` wandelt in einer Zeichenkette alle Kleinbuchstaben in Großbuchstaben um.

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

Es gibt nur Paddelboote mit 1, 2 oder 4 Paddel. Dafür sorgt die modifizierte Eingabemethode der Klasse `Paddelboot`.

```
  public void setPaddel( int paddel)
  {
    if( paddel == 2 || paddel == 4)
      this.paddel = paddel;
    else
      this.paddel = 1;
  }
```

9.1.2 this

Das Schlüsselwort `this` liefert innerhalb eines Objekte eine *Referenz auf das Objekt selbst* zurück. Ruft ein Objekt seine eigenen Instanzvariablen bzw. Instanzmethoden auf, so erfolgt dies entsprechend der oben angegebenen Syntax mit `this` und dem Punktoperator (`this.name`, `this.getName()`).

Die Methode `setName` setzt die Instanzvariable `name`. Die Methode `getName` liest den Inhalt der Instanzvariablen `name` und liefert diesen als Resultat zurück. In beiden Methoden wird das Attribut im Methodenrumpf mit `this.name` angesprochen:

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

public String getName()
{
  return this.name;
}
```

Damit wird in der Methode `setName` ein Konflikt mit dem Parameter `name` in der Parameterliste vermieden. Ist ein solcher Konflikt ausgeschlossen, so kann das Schlüsselwort `this` weggelassen werden. In der Methode `getName` wäre also auch die folgende Implementierung möglich:

```
public String getName()
{
  return name;
}
```

9.1.3 toString()

Stringausgaben auf der Konsole: `System.out.println( String str);`

Ergebnisausgaben auf der Konsole wurden bisher mittels der Methode `System.out.println` mit einem *String* als Argument erzeugt.

Objektbeschreibungen auf der Konsole: `System.out.println( Object obj);`

Für die Beschreibung eines *Objekt* auf der Konsole besteht die Möglichkeit, die Methode `System.out.println` mit einem *Objekt* als Argument aufzurufen.

```
System.out.println( swen);
```

Für dieses Beispiel wird die folgende *nicht sehr sinnvolle* Konsolenausschrift erzeugt:

```
PaddelBoot@1cde100
```

Jedes Objekt erbt eine Methode `toString`. Die Methode `System.out.println` verwendet diese Methode. Durch *Überschreiben* von `toString` kann man eine Objektbeschreibung gezielt steuern.

Legen wir in der Klasse `Boot` die Methode `toString` wie folgt fest:

```
public String toString()
{
  String str = "Bootstyp:      " + getTyp();
  str += "\nBootsname:    \"" + name + "\"";
  return str;
}
```

Der Methodeaufruf `System.out.println( swen);` liefert dann als Ausgabe:

```
Bootstyp:      PaddelBoot
Bootsname:     "SWEN"
```

9.2 Klassenvariablen und Klassenmethoden

9.2.1 Vereinbarung

Klassenvariablen und Klassenmethoden gehören einer Klasse, sind also *klassenspezifische* Bestandteile. Sie existieren sogar, wenn es *keine* Objekte dieser Klasse gibt. Im ersten Teil der Vorlesung wurde dieser Umstand ausgenutzt, um am Anfang *nicht* auf die objektorientierte Sicht einzugehen. *Alle Methoden waren Klassenmethoden*. Obwohl keine Instanz dieser Klassen erzeugt wurde, konnte mit den Methoden gearbeitet werden.

Deklaration und Definition
Klassenvariablen und *Klassenmethoden* werden von den *Instanzvariablen* und den *Instanzmethoden* durch das Schlüsselwort `static` unterschieden. Auch hier gilt Klassenvariablen speichern *Klassendaten* und sollten in der Regel *nicht öffentlich* sein. Klassenmethoden dienen der Datenbereitstellung und der Datenmanipulierung der *Klassendaten*. Sie sollten *nur dann* öffentlich sein, wenn sie als Dienstleistung von anderen Klassen angefordert werden.

private static *Typ Variablenname [= Ausdruck]***;**

public / private static *Ergebnistyp Methodenname* **(** *Parameterliste* **)** // Methodenkopf
{
Anweisungen // Methodenrumpf
}

Zugriff
Da beide Komponenten zur Klasse gehören, erfolgt der Zugriff über den Klassennamen.

Klassenname **.** *Methodenname* **(** *Argumentenliste* **)**
Klassenname **.** *Variablenname*

Ein typisches Beispiel für die Verwendung von *Klassenmethoden* wurde bereits mit der Klasse `java.lang.Math` besprochen, zum Beispiel `Math.sin()`.
In der Klasse `System` ist eine *Klassenvariable* `out` als Objekt einer Klasse `PrintStream` definiert. In der Anweisung `System.out.println();` wird eine Instanzmethode der Klasse `PrintStream` aufgerufen.

Eine Rederei möchte einen Überblick über die Anzahl der angeschafften Boote haben. In die abstrakte Klasse `Boot` wird deshalb ein Zähler `anzahl` eingeführt. Eine entsprechende Methode `incAnzahl` sorgt für das Weiterzählen des Zählers.
Dieser Zähler kann nicht als Instanzvariable aufgenommen werden, da sein Wert unabhängig von den Objekten ist. Es handelt sich um eine Eigenschaft der Klasse und nicht um eine Aussage über das einzelne Boot. Der Zähler und damit auch das Weiterzählen sind somit eine Klassenvariable bzw. eine Klassenmethode, also statische Komponenten der Klasse `Boot`:

```
private static int anzahl = 0;        // Klassenvariable

public static void incAnzahl()         // Klassenmethode
{
  Boot.anzahl++;
}
```

Innerhalb einer Klasse kann der Klassenname weggelassen werden, falls Verwechslungen ausgeschlossen sind (analog `this`).

```
public static void incAnzahl()        // Klassenmethode
{
  anzahl++;
}
```

Im UML-Diagramm werden Klassenvariablen und Klassenmethoden durch Unterstreichen gekennzeichnet.

Boot	
- name:	String
- anzahl:	int
+ setName(String):	void
+ getName():	String
+ getTyp():	String
+ bremsen(int):	*void*
+ toString():	String
+ incAnzahl():	void

Boot.java

```
// Boot.java                                        MM 2003

public abstract class Boot
{
  private static int anzahl = 0;     // Klassenvariable

  prublic static void incAnzahl()     // Klassenmethode
  {
    anzahl++;
  }

/*------------------------------------------------------*/
  private String name;               // Instanzvariablen

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

  public String getName()
  {
    return name;
  }

  public String getTyp()
  {
    return getClass().getName();
  }

  public abstract void bremsen( int delta);  //abstract
```

```
    public String toString()
    {
      String str = "Bootstyp:      " + getTyp();
      str += "\nBootsname:    \"" + name + "\"";
      str += "\nBootsanzahl:  " + anzahl;
      return str;
    }
  }
```

Die Methode `toString` wurde erweitert, so dass zusätzlich der Zähler `anzahl` ausgegeben wird. Das Weiterzählen der Bootsnummer erfolgt explizit vor der **Instanziierung** eines neuen Bootes.

TestBoot.java

```
// TestBoot.java                                    MM 2005

public class TestBoot
{
  public static void main( String[] args)
  {
// Objektbildung
    Boot.incAnzahl();
    PaddelBoot swen = new PaddelBoot();

// set-Methoden
    swen.setName( "Swen");
    swen.setPaddel( 1);

// Bremsen
    swen.bremsen( 5);
    System.out.println( swen);
  }
}
```

TestBoot.out

```
Boot SWEN: Paddel einziehen!

Bootstyp:     PaddelBoot
Bootsname:    "SWEN"
Bootsanzahl:  1
```

Klassenvariablen und Klassenmethoden werden *nicht* vererbt, d.h. mit der Klassenvariablen `anzahl` der Klasse `Boot` lässt sich *nicht* die Anzahl der Objekte der Klasse `Paddelboot` zählen.

9.2.2 super

Mit dem Schlüsselwort `super` kann auf Attribute und Methoden der Superklasse zugegriffen werden. Das ist besonders dann interessant, wenn man in den voneinander abgeleiteten Klassen gleiche Attribut- bzw. Methodennamen hat. Mit `super.super` greift man auf Superklasse der Superklasse zu usw. usf.

Die abgeleitete Klasse `PaddelBoot` erweitert die Klasse `Boot` um eine Instanzvariable `paddel`, welche die Anzahl der Paddel speichert. Es gibt Paddelboote mit einem, zwei oder vier Paddel. Die Methode `toString` aus der Klasse `Boot` soll für Paddelboote erweitert werden, indem zusätzlich die Paddelanzahl bereitgestellt wird.

Die Methode `toString` wird in der Klasse `PaddelBoot` überschrieben:

```
public String toString()
{
  String str = super.toString();
  str += "\nPaddelanzahl: " + paddel;
  return str;
}
```

Das Testprogramm ***TestBoot.java*** liefert für Paddelboote eine erweiterte Beschreibung:

TestBoot.out

```
Boot SWEN: Paddel einziehen!                              // bremsen

Bootstyp:     PaddelBoot         // toString-Methode aus Boot
Bootsname:    "SWEN"
Bootsanzahl:  1
Paddelanzahl: 1                                       // Erweiterung
```

9.3 Klassenkonstanten

So wie man Klassenvariablen definieren kann, ist es auch möglich, **Klassenkonstanten** festzulegen. Klassenkonstanten sind Konstanten, die für die gesamte Klasse gelten.

Deklaration der Klassenkonstanten

public / private static final *Typ KONSTANTENNAME* **[** = *Ausdruck* **];**

```
public static final double PI;                          // Math.PI
```

`final` verbietet den Schreibzugriff und `static` weist die Gültigkeit für die Klasse aus. Für *KONSTANTENNAME* wählt man Großbuchstaben. Der *Ausdruck* ist ein **Initialisierer**, er initialisiert die Konstante mit einem Wert.

```
public static final int MONTAG = 1;
public static final int DIENSTAG = 2;
public static final int MITTWOCH = 3;
public static final int DONNERSTAG = 4;
public static final int FREITAG = 5;
public static final int SONNABEND = 6;
public static final int SONNTAG = 7;
```

Konstanten sind oft notwendig, um Schlüssel zu verwalten, ohne sich diese zu merken.

```
private int wochenTag; ... datum.setWochenTag( MITTWOCH);
```

Ist eine Klassenkonstante eine *Referenz*, so verweist sie immer auf *ein und dasselbe* Objekt. Die Objektdaten selbst sind *nicht* konstant.

Deklaration der Klassenkonstanten mittels `static`-Block

Die Deklaration und Definition der Klassenkonstanten können getrennt voneinander vereinbart werden.

public / private static final *Typ KONSTANTENNAME* **;**

Nach der Deklaration kann an beliebiger Stelle die Definition durch einen **statischen Initialisierer** (**static-Block**) erfolgen. Mit dem ersten Aufruf der Klasse wird dieser abgearbeitet.

```
static                                                  // static-Block
{
  Anweisungen
}
```

Hat man mehrere **statische Initialisierer** eingerichtet, so werden diese in der Reihe ihres Auftretens abgearbeitet.

Statische Initialisierer haben nur Zugriff auf *statische* Komponenten einer Klasse, die davor definiert wurden. Deshalb ist es empfehlenswert, diese *an das Ende* einer Klassendefinition zu setzen.

In unserem Beispiel fügen wir ein ausgezeichnetes Boot `SCHIFFS_WRACK` in die Klasse `PaddelBoot` als Konstante ein:

```
                        // Klassenkonstante als Referenz
public static final PaddelBoot SCHIFFS_WRACK;

static                      // statischer Initialisierer
{
  Boot.incAnzahl();
  SCHIFFS_WRACK = new PaddelBoot();
  SCHIFFS_WRACK.setName( "Schiffswrack Eva");
  SCHIFFS_WRACK.setPaddel( 4);
}
```

Wird im Testprogramm ***TestBoot.java*** zusätzlich die Beschreibung der Konstante `SCHIFFS_WRACK` abgefragt,

```
System.out.println( PaddelBoot.SCHIFFS_WRACK);
```

so erhält man als Ergebnis die folgende Ausschrift:

TestBoot.out

```
...
Bootstyp:      PaddelBoot
Bootsname:     "SCHIFFSWRACK EVA"
Bootsanzahl:   2
Paddelanzahl:  4
```

9.4 Instanziierung und Initialisierung

9.4.1 Konstruktoren

In objektorientierten Sprachen sorgen **Konstruktoren** für die *Objektbildung* (**Instanziierung**). Durch diese kann man Objekte mit *Startwerten* versehen, *Methoden* aufrufen und vieles andere mehr.

Das Weiterzählen der Bootsnummer haben wir jeweils *vor* der *Instanziierung* eines neuen Bootes explizit aufgerufen.

```
Boot.incAnzahl();
PaddelBoot swen = new PaddelBoot();

Boot.incAnzahl();
SCHIFFS_WRACK = new PaddelBoot();
```

Um ein korrektes Weiterzählen zu garantieren, müssen beide Anweisungen stets gemeinsam auftreten. Am günstigsten wäre es, wenn bei jeder Objekterzeugung mit dem **new**-Operator das Weiterzählen automatisch geschehen würde.

Bei der Instanziierung ruft der **new**-Operator einen sogenannten **Konstruktor** auf. Dieser erzeugt ein Objekt einer Klasse. Bisher wurde ein **Standardkonstruktor** aufgerufen.
Man kann die Objekterzeugung durch klasseneigene *Konstruktoeren* beeinflussen. Die *Definition eines Konstruktors* erfolgt *ähnlich* der einer Methode, mit drei wesentlichen Unterschieden:

1. Der Name des Konstruktors entspricht dem der Klasse.
2. Er besitzt keinen Rückgabewert, nicht einmal `void`.
3. Er wird grundsätzlich *nur* im Zusammenhang mit dem **new** -Operator aufgerufen.

public *Klassenname* **(** *Parameterliste* **)**
{
Anweisungen
}

Für die Klasse `Boot` soll ein Konstruktor automatisch die Bootsanzahl weiterzählen. Außerdem wird der Name des Bootes gleich bei der Objekterzeugung übergeben.

```
public Boot( String name)
{
  incAnzahl();
  setName( name);
}
```

Die Möglichkeit eines *öffentlichen Zugriffs* auf die Methode `incAnzahl` wird mit dem expliziten Konstruktor überflüssig und sollte deshalb unterbunden werden. Deshalb ist es sinnvoll, dieser den öffentlichen Zugriff zu entziehen.

```
private static void incAnzahl()      // Klassenmethode
{
```

```
    anzahl++;
  }
```

Wir haben uns bisher einem **Defaultkonstruktor** (*Standardkonstruktor*) bedient. Dieser wird vom System *dann und nur dann* aufgerufen, wenn die Klasse explizit *keinen eigenen Konstruktor* aufweist. Er hat *keinerlei* Parameter, dient der Objektbildung und macht sonst *nichts*.

```
  public PaddelBoot(){}
```

Zu einer Klasse kann man *ein* oder *mehrere* Konstruktoren angeben. Mehrere Konstruktoren werden entsprechend dem *Überladen von Methoden* nach *Anzahl*, *Typ* und *Position* der Argumente behandelt.

Klassenname Objektname **;**
Objektname **=** **new** ***Klassenname*** **(** ***Parameterliste*** **) ;**

bzw.

Klassenname Objektname **=** **new** ***Klassenname*** **(** ***Parameterliste*** **) ;**

this()
Möchte man innerhalb eines Konstruktors einen anderen Konstruktor *derselben Klasse* aufrufen, so wird dieser als *erste Anweisung* mit dem Schlüsselwort `this` aktiviert.

Ein zweite Konstruktor sollte einen späteren Nachtrag des Namen gestatten.

```
  public Boot()
  {
    this( "");          // Konstruktor ruft Konstruktor auf
  }
```

super()
Soll ein Konstruktor einer abgeleiteten Klasse auf einen *Konstruktor seiner Superklasse* zugreifen, wird als *erste Anweisung* dieser mit `super()` aktiviert.

Nun benötigen wir noch Konstruktoren für die Klasse `PaddelBoot`.

```
  public PaddelBoot( String name, int paddel)
  {
    super( name);    // Konstruktor ruft Super-Konstruktor auf
    setPaddel( paddel);
  }
```

Weitere Konstuktoren ermöglichen unterschiedliche Formen der Objektbildung:

```
  public PaddelBoot( int paddel)
  {
    this( "", paddel);
```

```
  }

  public PaddelBoot( String name)
  {
    this( name, 1);
  }

  public PaddelBoot()
  {
    this( "", 1);
  }
```

9.4.2 Beispiel „Boot“

Zusammenfassend sollen die modellierten Bootsklassen noch einmal angegeben werden. Zusätzlich wurde in der Klasse `Boot` noch eine Instanzvariable `nummer` aufgenommen. Sie ordnet jedem Boot eine Bootsnummer zu.
Außerdem wurden durch entsprechende Kommentare alle Klassen, Attribute und Methoden in diesen, zum Erzeugen einer Online-Dokumentation mittels `javadoc` vorbereitet.

```
Boot
- name:                 String
- nummer:               int
- anzahl:               int
+ Boot( String)
+ Boot()
+ setName( String):     void
+ getName():            String
+ getTyp():             String
+ bremsen( int):        void
+ toString():           String
- incAnzahl():          void
```

```
PaddelBoot
- paddel:               int
+ SCHIFFS_WRACK:        PaddelBoot
+ PaddelBoot( String, int)
+ PaddelBoot( int)
+ PaddelBoot( String)
+ PaddelBoot()
+ setPaddel( int):      void
+ getPaddel():          int
+ bremsen( int):        void
+ toString():           String
```

Boot.java

```
// Boot.java                                              MM 2005

/**
 * Boot, Beispiel fuer abstrakte Klassen.
 */
public abstract class Boot
{
/*---------------------------------------------------*/
                                          // Konstruktoren
/**
 * Konstuktor, erzeugt neues Boot mit Namen.
 * @param name Name des Bootes
 */
  public Boot( String name)
  {
    incAnzahl();
    nummer = anzahl;
```

```
    setName( name);
  }

/**
 * Konstuktor, erzeugt neues Boot.
 */
  public Boot()
  {
    this( "");                  // ruft Konstruktor auf
  }

/*-----------------------------------------------------*/
                                     // Klassenvariable
/**
 * Gesamtzahl der vorhandenen Boote.
 */
  private static int anzahl = 0;

/*-----------------------------------------------------*/
                                      // Klassenmethode
/**
 * Klassenmethode, Erhoehen der Gesamtzahl der Boote.
 */
  private static void incAnzahl()
  {
    anzahl++;
  }

/*-----------------------------------------------------*/
                                    // Instanzvariablen
/**
 * Bootsname.
 */
  private String name;

/**
 * Bootsnummer.
 */
  private int nummer;

/*-----------------------------------------------------*/
                                     // Instanzmethoden
/**
 * Veraendern des Bootsnamens.
 * @param name Bootsname
 */
  public void setName( String name)
  {
    this.name = name.toUpperCase();
  }

/**
```

```
 * Lesen des Bootsnamen.
 */
  public String getName()
  {
    return name;
  }

/**
 * Ermitteln des Bootstyp.
 */
  public String getTyp()
  {
    return getClass().getName();
  }

/**
 * Abbremsen des Bootes, abstrakt.
 * @param delta Zielgeschwindigkeit
 */
  public abstract void bremsen( int delta);

/**
 * Beschreibung eines Bootes.
 */
  public String toString()
  {
    String str = "Bootstyp:      " + getTyp();
    str += "\nBootsname:    \"" + name + "\"";
    str += "\nNummer:        " + nummer;
    str += "\nBootsanzahl:  " + anzahl;
    return str;
  }
}
```

PaddelBoot.java

```
  // PaddelBoot.java                                      MM 2005

  /**
   * Paddelboot, Beispiel fuer abgeleitete Klassen.
   */
  public class PaddelBoot extends Boot
  {
  /*-----------------------------------------------------*/
                                          // Konstruktoren
  /**
   * Konstuktor,
   * erzeugt Paddelboot mit Namen und Paddelanzahl.
   * @param name Name des Bootes
   * @param paddel Paddelanzahl
   */
    public PaddelBoot( String name, int paddel)
```

```
    {
      super( name);
      setPaddel( paddel);
    }

  /**
   * Konstuktor,
   * erzeugt Paddelboot mit Paddelanzahl.
   * @param paddel Paddelanzahl
   */
    public PaddelBoot( int paddel)
    {
      this( "", paddel);
    }

  /**
   * Konstuktor,
   * erzeugt Paddelboot mit Namen und einem Paddel.
   * @param name Name des Bootes
   */
    public PaddelBoot( String name)
    {
      this( name, 1);
    }

  /**
   * Konstuktor,
   * erzeugt Paddelboot mit einem Paddel.
   */
    public PaddelBoot()
    {
      this( "", 1);
    }

  /*---------------------------------------------------*/
                                      // Instanzvariable
  /**
   * Paddelanzahl.
   */
    private int paddel;

  /*---------------------------------------------------*/
                                      // Instanzmethoden
  /**
   * Festlegen der Paddelanzahl.
   * @param paddel Paddelanzahl (1, 2 oder 4)
   */
    public void setPaddel( int paddel)
    {
      if( paddel == 2 || paddel == 4)
        this.paddel = paddel;
      else
```

```
        this.paddel = 1;
    }

  /**
   * Ermitteln der Paddelanzahl.
   */
    public int getPaddel()
    {
      return paddel;
    }

  /**
   * Abbremsen eines Paddelbootes,
   * ueberschreibt abstrakte Methode.
   * @param delta Zielgeschwindigkeit
   */
    public void bremsen( int delta)
    {
      System.out.println
      ( "Boot " + getName() + ": Paddel einziehen!");
    }

  /**
   * Beschreibung eines Paddelbootes,
   * verwendet Stringbeschreibung eines Bootes.
   */
    public String toString()
    {
      String str = super.toString();
      str += "\nPaddelanzahl: " + paddel;
      return str;
    }

  /*---------------------------------------------------*/
                          // Klassenkonstante als Referenz
  /**
   * SCHIFFS_WRACK, untergegangenes Paddelboot.
   */
    public static final PaddelBoot SCHIFFS_WRACK;

    static                    // statischer Initialisierer
    {
      SCHIFFS_WRACK = new PaddelBoot( "Schiffswrack Eva", 4);
    }
  }
```

TestBoot.java

```
  // TestBoot.java                                    MM 2005

  /**
   * Testet die Klassen Boot und PaddelBoot.
```

```
 */
public class TestBoot
{
  public static void main( String[] args)
  {
// Objektbildung 4. Konstruktor
    PaddelBoot swen = new PaddelBoot();

// set-Methoden
    swen.setName( "Swen");
    swen.setPaddel( 1);

// Bremsen
    swen.bremsen( 5); System.out.println();

// Objektbildung 1. Konstruktor
    PaddelBoot anita = new PaddelBoot( "Anita", 2);
    PaddelBoot marie = new PaddelBoot( "Marie", 3);

// Beschreibung der Objekte mit Methode toString
    System.out.println( swen); System.out.println();
    System.out.println( anita); System.out.println();
    System.out.println( marie); System.out.println();

// Klassenkonstante
    System.out.println( PaddelBoot.SCHIFFS_WRACK);
  }
}
```

TestBoot.out

```
Boot SWEN: Paddel einziehen!

Bootstyp:      PaddelBoot
Bootsname:     "SWEN"
Nummer:        2
Bootsanzahl:   4
Paddelanzahl:  1

Bootstyp:      PaddelBoot
Bootsname:     "ANITA"
Nummer:        3
Bootsanzahl:   4
Paddelanzahl:  2

Bootstyp:      PaddelBoot
Bootsname:     "MARIE"
Nummer:        4
Bootsanzahl:   4
Paddelanzahl:  1

Bootstyp:      PaddelBoot
Bootsname:     "SCHIFFSWRACK EVA"
Nummer:        1
```

```
Bootsanzahl:  4
Paddelanzahl: 4
```

Wie aus dem Beispiel ersichtlich, wird vor allen anderen Instanziierungen zuerst die Konstante `PaddelBoot.SCHIFFS_WRACK` eingerichtet.

Schließlich fertigen wir noch eine Online-Dokumentation unseres Bootsprojektes an:

```
$javadoc -private ..\*.java
```

Boot - SeaMonkey

Datei Bearbeiten Ansicht Gehe Lesezeichen Extras Fenster Hilfe

Zurück Vor Neu laden Stopp file:///C:/Users/meiler/Desktop/Arbeitsmappe/MuP1_WS10/Programme/E Suchen Drucken

Startseite Lesezeichen Dr. Monika Meiler Lernplattform der Univ... GL Inf/Numerik MuP I Propäd(C) Propäd (Ank)

All Classes
Boot
PaddelBoot
TestBoot

Class Boot

```
java.lang.Object
  └Boot
```

Direct Known Subclasses:
PaddelBoot

```
public abstract class Boot
extends java.lang.Object
```

Boot, Beispiel fuer abstrakte Klassen.

Field Summary

`private static int`	`anzahl` Gesamtzahl der vorhandenen Boote.
`private java.lang.String`	`name` Bootsname.
`private int`	`nummer` Bootsnummer.

Constructor Summary

`Boot()`   Konstuktor, erzeugt neues Boot.
`Boot(java.lang.String name)`   Konstuktor, erzeugt neues Boot mit Namen.

Method Summary

`abstract void`	`bremsen(int delta)` Abbremsen des Bootes, abstrakt.
`java.lang.String`	`getName()` Lesen des Bootsnamen.
`java.lang.String`	`getTyp()` Ermitteln des Bootstyp.
`private static void`	`incAnzahl()` Klassenmethode, Erhoehen der Gesamtzahl der Boote.
`void`	`setName(java.lang.String name)` Veraendern des Bootsnamens.
`java.lang.String`	`toString()` Beschreibung eines Bootes.

9.4.3 Der Mechanismus der Objekterzeugung

Mit dem **`new`**-Operator wird ein neues Objekt einer Klasse erzeugt. Hier soll nun dieser *Erzeugungsprozess* etwas genauer betrachtet werden.

Klassenname Objektname **;**
Objektname = **new** *Klassenname* **(** *Parameterliste*) **;**

bzw.

Klassenname Objektname = **new** *Klassenname* **(** *Parameterliste*) **;**

Folgende Schritte werden beim Anlegen eines Objektes der Reihe nach durchlaufen:

1. ***Speicherplatz***
   Das System organisiert Speicherplatz, um den Inhalt aller *Instanzvariablen* abspeichern zu können und weist dem Objekt die Referenz darauf zu. Sollte nicht genügend Speicher vorhanden sein, so bricht das Programm mit einem so genannten `OutOfMemory`-Fehler ab.

2. ***Standardwerte***
   Die Variablen werden mit Standardwerten (*Defaultwerte*) versehen:

Datentyp	Standardwert
`byte`	`(byte) 0`
`short`	`(short) 0`
`int`	`0`
`long`	`0L`
`float`	`0.0f`
`double`	`0.0d`
`char`	`(char) 0`
`boolean`	`false`
Referenz	`null`

3. ***Konstruktor***
   Der Konstruktor wird mit den übergebenen Werten aufgerufen. Das weitere Vorgehen unterscheidet sich nach der *ersten Anweisung* des Konstruktors.
   - `super(...)`: Der Konstruktor der direkten Superklasse wird aufgerufen.
   - `this(...)`: Der entsprechende Konstruktor derselben Klasse wird aufgerufen.
   - Weder `super(...)` noch `this(...)`: Der Standardkonstrukter `super()` der direkten Superklasse wird aufgerufen.

4. ***Initialisierer***
   Danach werden alle in der Klasse mit Initialisierern versehenen Variablen und Konstanten mit den angegebenen Werten versehen.

5. ***Konstruktor***
   Jetzt erst werden die restlichen Anweisungen der Konstruktorrumpfes ausgeführt.

9.5 Pakete

9.5.1 Erstellen von Paketen

Ein Paket stellt eine *Sammlung von Klassen* dar und ist eine weitere Möglichkeit, Programme zu gliedern. Ein Beispiel hierfür ist das Paket `java.lang` mit den Standardklassen `String` und `Math`. Dieses Paket wird automatisch vom System eingebunden. Anders verhält es sich bei dem im Kurs verwendeten Paket `Tools`, welches die Klassen `IOTools` und `Euklid` und andere enthält. Dieses Paket wurde neu installiert und bei jeder Verwendung importiert.

Schritte bei der Erzeugung eigener Pakete, z.B. `myPackage`:

1. Im Arbeitsverzeichnis, z.B. `myJava`, wir ein Unterverzeichnis mit dem Namen des Pakets, z.B. `myPackage`, erzeugt. In diesem Verzeichnis wird der Quellcode aller Klassen des Pakets abgespeichert.

2. Die *erste* Nichtkommentarzeile im Quellcode jeder Klasse, z.B. `myClass.java`, des Paketes lautet:

   ```
   package myPackage;
   ```

3. Die Übersetzung der Klassen erfolgt wie üblich:

   ```
   javac myClass.java
   ```

   Die erzeugt Datei `myClass.class` befindet sich danach im Verzeichnis `myPackage`.

4. Soll die Klasse in einem anderen Programm, z.B. `myTest.java`, benutzt werden, so muss

   a. durch die Anweisung `import myPackage.myClass;` die Klasse `myClass.java` eingebunden und

   b. beim Übersetzen und Ausführen die Umgebungsvariable CLASSPATH auf das Arbeitsverzeichnis `myJava` gesetzt werden:

      ```
      testpackage> javac -classpath .;...\myJava myTest.java
      testpackage> java -classpath .;...\myJava myTest
      ```

      oder die Umgebungsvariable CLASSPATH wird generell auf das Arbeitsverzeichnis `myJava` gesetzt:

      Windows: `CLASSPATH=.;C:\...\myJava`
      Unix: `export CLASSPATH=.:/.../myJava`

9.5.2 Installation komprimierter Pakete

Um dem Anfänger Eingaberoutinen zur Verfügung zu stellen, haben wir das Paket `Tools` in komprimierter Form eingebunden.

Das geht natürlich auch mit jedem anderen Paket. Dazu sind alle zum Paket gehörigen Klassen, bereits übersetzt, zusammenzufassen. Zur Installation eines so vorhandenen komprimierten Paketes, z.B. `myPackage.zip`, sind drei Schritte erforderlich:

1. Das komprimierte Paket `myPackage.zip` wird in ein Verzeichnis, z.B. `myJava`, gespeichert, *ohne* es auszupacken.

2. Jedes Java-Programm, welches Klassen des Pakets `myPackage` verwendet, muss diese einbinden. Das erfolgt am Anfang des Programms mit der Anweisung

   ```
   import myPackage.myClass;
   ```

3. Zum Übersetzen und Ausführen muss die Umgebungsvariable `CLASSPATH` auf `myPackage.zip` gesetzt werden:

   Windows: `CLASSPATH=.;C:\...\myJava\myPackage.zip`
   Unix: `export CLASSPATH=.:/.../myJava/myPackage.zip`

Nach diesen Vorbereitungen können die Klassen des Paketes `myPackage` verwendet werden.

9.6 Zugriffsrechte

Entsprechend der Datenkapselung sollte man nur Methoden der Öffentlichkeit zur Verfügung stellen, welche als Dienstleistung abverlangt werden (**Prinzip der maximalen Verschattung: „so privat wie möglich – so öffentlich wie nötig“**). Attribute sollen, gemäß der Idee des *data hiding*, grundsätzlich nur durch eigene Methoden manipuliert werden. Damit erreicht man einen kontrollierten Zugriff auf diese.

Betrachten wir die Klasse `Boot`. In dieser Klasse wird der aktuelle Name des Bootes `name` als Attribut *nicht öffentlich* angelegt. *Öffentliche* Methoden, wie `getName` und `setName`, aktualisieren das Attribut. Diese Methoden gehören zur *Schnittstelle* der Klasse.

Das Schlüsselwort `final` definiert Konstanten und verbietet generell den *Schreibzugriff*.

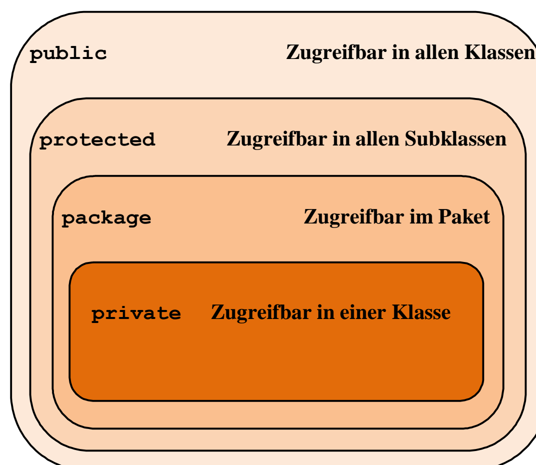
Die Schlüsselworte `public`, `private` und `protected` steuern die Zugriffsrechte. Sind keine Zugriffsrechte angegeben, so hat die Komponente das sogenannte `package` - Zugriffsrecht.

`public` (UML +) gestattet bei Attributen den Zugriff, bei Methoden den uneingeschränkten Aufruf von jeder Klasse aus.

`package` (kein Modifikator) erlaubt den Zugriff aller Klassen desselben Pakets.

`protected` (UML #) gestattet den Zugriff für die Klasse selbst und aller ihrer Subklassen.

`private` (UML -) verhindert den öffentlichen Zugriff. Ein Zugriff ist nur innerhalb der Klasse möglich, auch zwischen verschiedenen Objekten derselben Klasse. Super- bzw. Subklassen haben keine Zugriffsrechte.



Spezifikation	Klasse	Paket	Subklasse	Welt
`private`	X			
ohne	X	X		
`protected`	X	X	X	
`public`	X	X	X	X