Konstruktoren

```
import java.lang.Math;

public class Konstruktoren
{
    public String name;
    public int min=0;
    public int max=10;

    private static int objCount=0;

public Konstruktoren()
{
    name="Konstruktor-"+objCount++;
}

public Konstruktoren(String name)
{
    objCount++;
    // this wegen gleichem Namen nötig
    this.name=name;
}

public Konstruktoren(String name, int min)
{
    objCount++;
    this.name=name;
    // Oder: this(name);
    // Dann aber OHNE objCount++!!!!!
    this.min=min;
}

public Konstruktoren(String name, int min, int
max)
{
    this(name,min);
    this.max=max;
}
```

```
/* Alternative: Andersherum aufzäumen
   public Konstruktoren()
   {
       this("Konstruktor-"+objCount,0,10);
   }
*/

public long round(double val)
{
    if(val<min)
        return min;
    if(val>max)
        return max;
    // ACHTUNG: Bei return round(val)
    // entsteht eine Endlosschleife!!!!!
    return Math.round(val);
}

static public void main(String args[])
{
    System.out.println("Programmstart\n");
    Konstruktoren k1=new Konstruktoren();
    Konstruktoren k2=new
Konstruktoren("Test2",6);
    System.out.println("Objekt 1:
"+k1.round(5.3));
    System.out.println("Objekt 2:
"+k2.round(5.3));
    System.out.println("\nProgrammende");
}

}
```

Ausgabe:

```
Programmstart
Objekt 1: 5
Objekt 2: 6
Programmende
```

Konstruktoren überladen

Beispiel 1: "Privater" machen

```
class A extends Object
{
    public A(){}
}

class B extends A
{
    private B(){}
}
```

Beispiel 2: "Öffentlicher" machen

```
class A extends Object
{
    private A(){}
}

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

Dies ist **nicht** möglich! Beim kompilieren von Klasse B wird im Kostruktor automatisch `super();` eingefügt, dieser Konstruktor ist aber private und damit **nicht** zugänglich

⇒ Es ist **keinerlei** weitere Ableitung mehr möglich!!!!

Ausnahme: Es gibt einen anderen öffentlichen Konstruktor:

```
class A extends Object
{
    private A(){}
    protected A(int param){}
}

class B extends A
{
    public B()
    {
        super(int);
    }
}
```

Dies ermöglicht noch Subklassen.

Überschreiben von private-Methoden

```
class A extends Object
{
    public A()
    {
        doIt();
    }

    private doIt()
    {
        println("A");
    }
}

class B extends A
{
    public B(){}

    public doIt()
    {
        println("B");
    }
}
```

`new A()` bewirkt: `A`
`new B()` bewirkt: `A` -> doIt wurde **nicht** überschrieben

Änderung von `A.doIt()` auf `protected:`
`new A()` bewirkt: `A`
`new B()` bewirkt: `B` -> doIt wurde überschrieben

Statische Initialisierungen / Klassenmethoden

```
class Factory
{
    static String[] names=null;

    static
    {
        names=new String[2];
        names[0]="java.lang.String";
        names[1]="java.lang.StringBuffer";
    }

    protected Factory(){}

    static Object getInstance(int index)
    {
        String className=names[index];
        Object obj=null;
        try
        {
obj=Class.forName(className).newInstance();
        }
        catch(Exception e){}
        return obj;
    }

    public static void main(String[] args)
    {
        Object obj=Factory.getInstance(0);
        if(obj instanceof String)
System.out.println("Obj 1 is a String!");
        else
System.out.println("Obj 1 is NO String!");
```

```
        obj=Factory.getInstance(1);
        if(obj instanceof String)
            System.out.println("Object 2 is a
String!");
        else
            System.out.println("Object 2 is NO
String!");
        if(obj instanceof
java.lang.StringBuffer)
            System.out.println("Object 2 is a
StringBuffer!");
        else
            System.out.println("Object 2 is NO
StringBuffer!");
 }
}
```

Testausgabe:

```
Object 1 is a String!
Object 2 is NO String!
Object 2 is a StringBuffer!
```

Typischerweise:

- Es würden Subklassen von Factory zurückgegeben
- Die Liste der verfügbaren Implementationen wird zur Laufzeit aus einer Datei eingelesen oder sonstwie dynamisch erstellt

Variableninitialisierung

```
public class Initializers
{
    /** Ist öffentlich bekannt, kann nicht
        verändert werden, gilt für ALLE Buffer.
        Typische Konstanten */
    public static final int N_MIN=-10;
    public static final int N_MAX=10;
    /* Ist im Paket bekannt, kann nicht
       verändert werden, gilt für ALLE Buffer.
       Typische Konstante */
    static final int BUFLEN=5;

    /* Ist nur lokal bekannt, kann verändert
      werden, gilt nur für JEWEILIGES Objekt */
    private int curBufferPos;

    /* Ganz normale Variablen */
    int[] buffer1=null;
    int[] buffer2=new int[BUFLEN];
    boolean full;

    /* Existiert nur einmal; ist für alle
       Objekte gleich! */
    private static int bufferCount;

    /* Objekt kann nicht mehr verändert werden,
       nur mehr die internen Werte */
    final GregorianCalendar cal=
                        new GregorianCalendar();

    /* Objekt kann nicht mehr verändert werden,
       nur mehr die internen Werte; Gilt für
       alle Objekte dieser Klasse! */
    /* KEIN new, da eine neuen Instanz über
       die statische Methode erzeugt wird */
    static final DateFormat df=
              DateFormat.getDateTimeInstance();
```

```
public Initializers (int i)
{
    // Initialisierung im Konstruktor;
    // nur für Instanzvariablen
    curBufferPos=i;
    // Geht, da Objekt dazu existiert (=this)!
    System.out.println("Derzeitige
              Bufferposition: "+curBufferPos);
}

public void doIt()
{
    /* Geht, da cal das selbe Objekt bleibt;
       nur innere Werte anders! */
    cal.setTime(new Date());
    System.out.println("Zeit: "+
                       df.format(cal.getTime()));
//  cal=new GregorianCalendar();  Geht nicht,
da cal final!

    /* df kann nicht verändert werden, da es
       final ist. Es existieren keine Methoden
       zur Änderung der Formatierung, daher
       neues Objekt nötig */
    DateFormat fulldf=
              DateFormat.getDateTimeInstance(
              DateFormat.FULL,DateFormat.FULL);
    cal.add(Calendar.HOUR,1);
    System.out.println("Zeit: "+
                  fulldf.format(cal.getTime()));
    /* Statische Variablen können immer und
       überall verändert werden */
    bufferCount++;
}
```

```
static public void main(String args[])
{
    System.out.println("Programmstart\n");

//  N_MIN=-8; Geht nicht, da ein statischer
Wert nicht verändert werden kann
//  System.out.println("Derzeitige
Bufferposition: "+curBufferPos); Geht nicht, da
kein Objekt dazu existiert!

    // Statischer Wert existiert ohne Objekt
    System.out.println("Bufferanzahl: "+
                        bufferCount);

    Initializers ini1=new Initializers(1);
    Initializers ini2=new Initializers(0);
// Instanzvariable -> jedes Objekt ein eigenes
    if(ini1.cal==ini2.cal)
        System.out.println("Calendar gleich");
    else
       System.out.println("Calendar ungleich");
    // static -> Gleich für alle Objekte
    if(ini1.df==ini2.df)
        System.out.println("DFormat gleich");
    else
        System.out.println("DFormat ungleich");
    // Objekt bearbeiten
    ini1.doIt();
    /* Statischen Werte überprüfen: Änderungen
       in einer Instanzmethode bleiben auch
       außerhalb erhalten */
    System.out.println("Bufferanzahl:
"+bufferCount);
    System.out.println("\nProgrammende");
}

}
```

Ausgabe:

```
Programmstart

Bufferanzahl: 0
Derzeitige Bufferposition: 1
Derzeitige Bufferposition: 0
Calendar ungleich
DFormat gleich
Zeit: 11-Feb-99 1:17:56 AM
Zeit: Thursday, February 11, 1999 2:17:56
o'clock AM PST
Bufferanzahl: 1

Programmende
```

Zugriffsrechte

Verwendbar in	Private	<package>	protected	public
Dieser Klasse	**4**	**4**	**4**	**4**
Allen Klassen dieses Paketes	**8**	**4**	**4**	**4**
Subklassen, auch wenn in anderem Paket	**8**	**8**	**4**	**4**
Sämtlichem Code	**8**	**8**	**8**	**4**

Wann verwendet man bei Methoden:

- Private: Interne Hilfs-Prozeduren, die von Subklassen nicht verwendet werden können/sollen
- Protected: Interne Hilfs-Prozeduren, die auch für Subklassen von Interesse sind
- Public: Öffentliche Schnittstelle
- Package access: Wenn zugehörige Klassen (=selbes Paket) auch darauf zugreifen können sollen (z. B. Einfachheit, Effizienz, …)

Wann verwendet man bei Variablen:

- Private: Wenn Daten nur über Zugriffsmethoden veränderbar sein sollen (Datenkapselung)
- Protected: Variablen, die z. B. aus Effizienzgründen auch von Subklassen verändert werden können sollen (Vorsicht, genaue Dokumentation des Inhalts und der Bedeutung notwendig!)
- Public: Eigentlich gar nicht (Höchstens: Schnittstelle zu native-code-Programmen)!
- Package access: In allen anderen Fällen

Wann verwendet man bei Konstanten:

- Private: Klasseninterne Konstanten. Ohne Bedeutung für Subklassen
- Protected: Konstanten, die auch in Subklassen von Bedeutung sind
- Public: Öffentliche Konstanten
- Package access: Eher selten. Nur bei Konstanten, die für mehrere Klassen von Bedeutung sind, aber nicht darüber hinaus.

Beispiel zu Zugriffsrechten

Die Klasse Message:

```
package A;

public class Message
{
    protected String content;
}
```

Die Klasse SpecialMessage:

```
package A.B;

import A.Message;

public class SpecialMessage extends Message
{
    public void doIt(Message otherMsg)
    {
        content="String1";
        otherMsg.content="String2";
    }
}
```

Was ist hier falsch?

Die Zuweisung otherMsg.content ist nicht erlaubt!

Grund: otherMsg ist ein Objekt der Klasse Message und befindet sich in einem anderen Package. Daher sind die protected-Felder/Methoden dieses Objektes NICHT zugreifbar. Im eigenen Objekt SIND sie zugreifbar, da hier auf die Elemente einer Superklasse zugegriffen wird; otherMsg ist hingegen von einer "völlig" fremden Klasse.

Die Klasse Person

```
package Abenteuer.Daten.Objekte;

import Abenteuer.Daten.Orte.Raum;
import java.util.Vector;
import java.util.Enumeration;
import java.text.NumberFormat;

/**
 * Diese Klasse repräsentiert eine Person. Sie hat einen Namen
 * eine kurze und eine ausführliche Beschreibung und besitzt
 * eine bestimmte Tragkraft.
 * @author Dipl.-Ing. Michael Sonntag
 * @version 1.0
*/
public class Person
{
     /** Statische Liste zur Speicherung aller Personen.
         Package access, sodaß auch unser Enumerator darauf
         zugreifen kann */
     static Vector personenListe=new Vector();

     private String beschreibung;
     private String kurzBeschreibung;
     private String name;
     private double tragkraft=1.0;

     private Raum derzeitigerOrt;
     /* Kein externer Zugriff, damit Gewicht funktioniert! */
     private Vector inventar=new Vector();
     private double derzeitigesGewicht=0.0;


     public Person(String name,String kurzBeschreibung,
                   String beschreibung,Raum derzeitigerOrt)
     {
          this.name=name;
          this.derzeitigerOrt=derzeitigerOrt;
          this.kurzBeschreibung=kurzBeschreibung;
          this.beschreibung=beschreibung;
          personenListe.addElement(this);
     }

    public Raum getDerzeitigerOrt()
    {
        return derzeitigerOrt;
    }
```

```
/**
    Bewegt die Person von dem derzeitigen Raum in eine
    bestimmte Richtung. Gibt es keinen Ausgang in diese
    Richtung, so wird eine Meldung ausgegeben.<br> Nur wenn
    die Person diesen Raum in diese Richtung verlassen darf
    und den neuen Raum vom derzeitige aus betreten darf,
    erfolgt tatsächlich ein Ortswechsel.<br>
    Anschließend wird die Beschreibung des neuen (oder alten)
    Raums ausgegeben.
    */
    public void bewege(int richtung)
    {
        if(derzeitigerOrt==null) return;
        Raum neu=derzeitigerOrt.Ausgang(richtung);
        if(neu!=null)
        {
               if(derzeitigerOrt.verlassen(this,richtung)
                  && neu.betreten(this,derzeitigerOrt))
                   derzeitigerOrt=neu;
               derzeitigerOrt.beschreibeRaum(this);
        }
        else System.out.println("Kein Weg in diese Richtung");
    }

     /**
     Fügt dem Inventar einen Gegenstand hinzu. Dies erfolgt
     nur, wenn zusammen mit dem
     neuen Gegenstand die maximale Tragkraft nicht
     überschritten wird.
     */
     public void addGegenstand(Gegenstand g)
     {
          if(!inventar.contains(g))
          {
               if(derzeitigesGewicht+g.getGewicht()>tragkraft)
               {
                    System.out.println("Zu schwer!");
                    return;
               }
               derzeitigesGewicht+=g.getGewicht();
               inventar.addElement(g);
          }
     }

     public void removeGegenstand(Gegenstand g)
     {
          if(inventar.contains(g))
          {
               derzeitigesGewicht-=g.getGewicht();
               inventar.removeElement(g);
          }
     }
```

```
    /**
    Liefert eine Enumeration aller Gegenstände die diese
    Person bei sich trägt zurück
    @return Enumeration aller Gegenstände
    @see java.util.Enumeration
    */
    public Enumeration gegenstaende()
    {
         return inventar.elements();
    }

    /**
    Stellt fest, ob die Person einen Gegenstand mit einem
    bestimmten Namen bei sich trägt.
    @return Irgendeiner der Gegenständ mit diesem Namen im
            Inventar oder null falls keiner gefunden wurde
    */
    public Gegenstand besitzt(String name)
    {
         Enumeration e=inventar.elements();
         Gegenstand res=null;
         while(e.hasMoreElements() && res==null)
         {
              Gegenstand g=(Gegenstand)e.nextElement();
              if(g.getName().equalsIgnoreCase(name))
                   res=g;
         }
         return res;
    }

    /**
    Stellt fest, ob die Person den Gegenstand g besitzt
    */
    public boolean besitztGegenstand(Gegenstand g)
    {
         return inventar.contains(g);
    }


    /**
    Liefert eine Aufzählung aller Personen in einem Raum
    @param raum Der Raum, für den alle darin befindlichen
                Personen geliefert werden sollen
    @return Aufzählung der Personen im Raum
    @see java.util.Enumeration
    */
    public static Enumeration personenInRaum(Raum raum)
    {
        return new PersonenEnumeration(raum);
    }
}
```

```
/**
Hilfsklasse die alle Personen in einem bestimmten Raum liefert
*/
class PersonenEnumeration implements Enumeration
{
    private Raum raum;
    private Enumeration enum=null;
    private Person naechste=null;

    public PersonenEnumeration(Raum raum)
    {
        this.raum=raum;
        enum=Person.personenListe.elements();
        geheZuNaechsterPerson();
    }

    public boolean hasMoreElements()
    {  return naechste!=null; }

    public Object nextElement()
    {
        Person res=naechste;
        geheZuNaechsterPerson();
        return res;
    }

    private void geheZuNaechsterPerson()
    {
        do
        {
            if(enum.hasMoreElements())
                naechste=(Person)enum.nextElement();
            else
                naechste=null;
        }
  while(naechste!=null && naechste.getDerzeitigerOrt()!=raum);
    }
}
```