PM2 Java: Packages und Sichtbarkeit

PACKAGES

Packages

- Packages sind Gruppen zusammengehöriger Klassen.
- Die Java Bibliothek ist in packages organisiert. So gehören z.B. ***Point*** und ***Polygon*** in das package ***java.awt***.
- Packages definieren Namensräume, die Klassennamen eindeutig machen. Der eindeutige Name einer Klasse ist der Package-Name plus Klassennamen.
- Man spricht auch vom ***voll qualifizierten Namen*** einer Klasse. Für ***Point*** ist der voll qualifizierte Name: ***java.awt.Point***

Packages

- Jede Klasse muss einem Package angehören. Die Package-Deklaration steht immer an erster Stelle vor der Klassendefinition im Quelltext.

- Der Package-Name wird im Dateisystem auf eine Ordnerhierarchie abgebildet:

  java.awt wird zu Ordner ***java*** mit dem Unterordner ***awt***.

- Klassendefinitionen ohne Package-Deklaration, werden dem Default-Package zugeordnet.

- **Vorsicht:** Klassen aus dem Default-Package können aus anderen Packages **nicht** importiert werden.

Voll qualifizierte Namen von Klassen und import

- Der Compiler benötigt den voll qualifizierten Namen einer Klasse für die Übersetzung des Quelltextes.

- Es gibt zwei Möglichkeiten den Namen anzugeben:
  1. Indem der vollständige Name der Klasse an den Stellen der Verwendung ausgeschrieben wird.
  2. Durch ***import*** des Klassennamen aus dem Package. Der Klassename wird vom Compiler um den Package-Namen ergänzt.

- Klassen aus dem Package ***java.lang***, z.B. die Klasse ***String***, müssen nicht importiert werden.

Voll qualifizierte Namen von Klassen und import

1.)

```
package qualifiziertversusimport;
public class PackageDemoQualifiziert {
    public static void main(String[] args) {
        java.awt.Point p=new java.awt.Point();
        java.awt.Polygon poly=new java.awt.Polygon();
        poly.addPoint( 10, 10 );
        poly.addPoint( 10, 20 );
    }
}
```

2.)

```
package qualifiziertversusimport;
import java.awt.Point;
import java.awt.Polygon;
public class PackageDemoImport {
    public static void main(String[] args) {
        Point p=new Point();
        Polygon poly=new Polygon();
        poly.addPoint( 10, 10 );
        poly.addPoint( 10, 20 );
    }
}
```

import * - es gibt keine Package Hierarchie

- Mit ***import ****können alle Klassen eines Packages importiert werden.
- So werden durch ***import java.util.**** alle Klassen, die direkt im Package ***java.util*** liegen, importiert.
- **Aber nicht** die Klassen aus den Packages ***java.util.color*** und anderer Packages, deren Präfix ***java.util*** ist.
- Auch wenn die Namen dies suggerieren, bilden Packages in Java keine Hierarchie.

Namenskonvention

- Alle in einer Datei definierten Typen, die Package Deklaration und die ***import*** Statements bilden zusammen eine **Unit of Compilation.**

- Ein Package-Name kann beliebig sein, doch beginnt er in Regel mit umgedrehten Domänennamen.

- Aus der Domäne ***http://haw-hamburg.de*** wird ***de.hawhamburg***. Diese Namenskonvention macht Klassen weltweit eindeutig.

- Package-Namen werden klein geschrieben.

- Werden Klassen gleichen Namens aus unterschiedlichen Packages in einer Unit of Compilation verwendet, dann müssen die voll qualifizierten Namen verwendet werden.

Statischer Import

- Mit statischem ***import*** besteht die Möglichkeit, Klassenvariablen und -methoden direkt zu verwenden, ohne den Klassennamen der importierten Klasse voran stellen zu müssen.

- **Beispiele**: statischer Import der Methoden der Klasse ***util.Printer*** / statischer Import der Methoden und Konstanten der Klasse ***java.lang.Math***.

```
package statischerimport;
import static util.Printer.p;
import static java.lang.Math.*;
public class StatischerImportDemo {
    public static void main(String[] args) {
        Double d = 5.78;
        p(d);
        p(sqrt(d));
        p(sin(d));
        p(sin(PI));
        p(cos(PI));
        p(log(E));
    }
}
```

SICHTBARKEITEN

Sichtbarkeiten

- Sichtbarkeiten regeln den Zugriff auf Attribute und Methoden von Klassen und auf Klassen und Interfaces.

- Innerhalb einer Klasse sind alle Methoden und Attribute für die Methoden eines Objektes sichtbar (auch die privaten).

- Externe Klassen für eine Klasse sind
  1. Klassen im gleichen Package.
  2. Klassen eines anderen Package, die von der eigenen Klasse ableiten.
  3. Klassen eines anderen Package, von der die eigene Klasse ableitet.
  4. Klassen in anderen Packages, die die eigene Klasse nutzen aber nicht von ihr ableiten.
  5. Klassen in anderen Packages, die die eigene Klasse nutzt, von der die eigene Klasse aber nicht ableitet

- Java unterstützt insgesamt 4 Sichtbarkeiten für Methoden und Attribute:
  - ***public*** (sichtbar für 1. - 5.)
  - ***protected*** (sichtbar für 1., 2. und 3.)
  - **package private** (sichtbar für 1.)
  - ***private*** (sichtbar für alle Objekte derselben Klasse)

Private Sichtbarkeit schützt vor „gefährlichen“ Übergriffen

Beispiel:

- Die Klasse ***Adresse*** verwaltet Postleitzahlen in einer Zeichenketten ***plz***.
- Sie muss sicher stellen, dass die Postleitzahl genau 5 Zeichen lang ist und nur aus Ziffern besteht.
- Wäre die Variable ***plz*** nicht ***private***, dann könnten externe Klassen den Wert der Variable direkt setzen. Es kann nicht garantiert werden, dass der geschriebene Wert gültig ist.
- Ist die Variable ***plz private***, dann können die Änderungen an der Variable von außen nur über eine Methode erfolgen (***setPlz***).
- Diese Methode prüft, ob der übergebene Wert gültig ist und generiert ggf. eine Ausnahme (***throw PlzFormat-Exception***).

Private Sichtbarkeit schützt vor „gefährlichen“ Übergriffen

```
public class Adresse {
    private String str, hnr, plz, ort;
    public Adresse(String str, String hnr, String plz, String ort)
            throws PlzFormatException {
        this.str = str;
        this.hnr = hnr;
        setPlz(plz);
        this.ort = ort;
    }
    public void setPlz(String plz) throws PlzFormatException {
        if (!plz.matches("\\d{5}"))
            throw new PlzFormatException("keine gültige Postleitzahl " + plz);
        else this.plz=plz;
    }
    public String toString() {
        return String.format("%s %s\n%s %s",str,hnr,plz,ort);
    }

    public static void main(String[] args) throws PlzFormatException {
        p(new Adresse("Berliner Tor","7","21099","Hamburg"));
        p(new Adresse("Berliner Tor","7","99","Hamburg"));
    }
}
```

```
Berliner Tor 7
21099 Hamburg
Exception in thread "main" sichtbarkeiten.PlzFormatException:
keine gültige Postleitzahl 99
at sichtbarkeiten.Adresse.setPlz(Adresse.java:17)
at sichtbarkeiten.Adresse.<init>(Adresse.java:11)
at sichtbarkeiten.Adresse.main(Adresse.java:27)
```

Private Sichtbarkeit ist nicht für alle Objekte privat

- Objekte der gleichen Klasse haben Zugriff auf die privaten Variablen und Methoden der Klasse. Beim Vergleich zweier Adressobjekte darf ***this*** auf die privaten Attribute von ***o*** zugreifen.

```
public class Adresse {
   private String str, hnr, plz, ort;
   @Override
   public boolean equals(Object o) {
      if (this == o)
         return true;
      if (o == null)
         return false;
      if (getClass() != o.getClass())
         return false;
      Adresse oa = (Adresse) o;
      return Arrays.deepEquals(new String[] { str, hnr, plz, ort },
            new String[] {oa.str, oa.hnr, oa.plz, oa.ort });
   }
...
}
```

→ Package sichtbarkeiten

Paketsichtbarkeit ist der Default

- Steht kein ausdrücklicher Sichtbarkeits-Modifikator vor Methoden, Variablen oder Klassen, dann ist deren Sichtbarkeit **package private**.

- Alle Objekte von Klassen desselben Package haben Zugriff auf die **package private** Methoden, Variablen und Klassen.

- Package private wird z.B. eingesetzt, wenn eine häufig verwendete Funktionalität von mehreren Klassen des Packages intern benötigt wird, aber nicht nach außen bekannt gegeben werden soll.

Protected ist package private und mehr

- ***protected*** Methoden und Variablen sind für alle ableitenden Klassen und für alle Klassen eines Packages sichtbar. Klassen aus anderen Packages haben keinen Zugriff.
- ***protected*** Methoden erlauben den ableitenden Klassen, die Methoden zu überschreiben.
- ***protected*** Methoden werden verwendet, wenn Subklassen diese Methoden überschreiben können oder sollen.
  - **Beispiel**: Für die Ausgabe der Medieninfo in DOME aus PM1 lieferte die Methode ***medien_spezifisch_to_s*** nur eine Teilfunktionalität.
  - **Beispiel**: Die Methode clone vn Object ist protected. Wird diese überschrieben, dann können die ableitenden Klassen die Sichtbarkeit auf public erweitern.
- ***protected*** Methoden werden auch verwendet, wenn die Subklassen darüber entscheiden sollen, ob die Funktionalität der Superklasse für die Außenwelt erreichbar ist. **Beispiel**: Methode ***clone*** von ***Object*** ist ***protected***. Subklassen machen die Methode ***public***, damit Kopien erzeugt werden können.

Bsp 1: Subklassen sollen Methoden überschreiben

```
public abstract class GeometricObject {

    public void translate(Point off) {
        setReferencePoint(getReferencePoint().add(off));
    }
    protected abstract Point getReferencePoint();
    protected abstract void setReferencePoint(Point d);

    public void scale(double scalar) {
        List<Double> newValues = new ArrayList<Double>();
        for (Double d : getValues()) {
            newValues.add(d * scalar);
        }
        setValues(newValues);
    }
    protected abstract List<Double> getValues();
    protected abstract void setValues(List<Double> values);
}
```

nur die Subklassen kennen ihren Referenzpunkt und können ihn modifizieren

nur die Subklassen kennen die Form-gebenden Größen und können diese modifi-zieren

Bsp 1: Subklassen sollen Methoden überschreiben

```
public class Rechteck extends GeometricObject {
    private Point upLeftCorner;
    private Double width;
    private Double height;
    public Rechteck(Point upLeftCorner, Double width, Double height) {
        this.upLeftCorner = upLeftCorner;
        this.width = width;
        this.height = height;
    }
    @Override
    protected Point getReferencePoint() {
        return upLeftCorner;
    }
    @Override
    protected void setReferencePoint(Point d) {
        this.upLeftCorner = d;
    }
    @Override
    protected List<Double> getValues() {
        return Arrays.asList(new Double[]{width,height});
    }
    @Override
    protected void setValues(List<Double> values) {
        width = values.get(0);
        height = values.get(1);
    }
}
```

Referenzpunkt ist die obere linke Ecke

Form-gebende Größen sind Breite und Höhe

Bsp 1: Subklassen sollen Methoden überschreiben

```java
public class Kreis extends GeometricObject {
    private Point center;
    private double radius;

    public Kreis(Point center, double radius) {
        this.center = center;
        this.radius = radius;
    }
    @Override
    protected Point getReferencePoint() {
        return center;
    }
    @Override
    protected void setReferencePoint(Point d) {
        this.center = d;
    }
    @Override
    protected List<Double> getValues() {
        return Arrays.asList(new Double[]{radius});
    }
    @Override
    protected void setValues(List<Double> values) {
        radius = values.get(0);
    }
}
```

Referenzpunkt ist der Mittelpunkt

Form-gebende Größe ist der Radius

Bsp 2: Subklassen können Methoden überschreiben und die Sichtbarkeit erweitern

```
@Override
public Rational clone() throws
    CloneNotSupportedException {
    return (Rational) super.clone();
}
```

clone in Object ist protected

Rational erweitert die Sichtbarkeit von clone auf public

Nutzt die für Rational sichtbare Methode clone von Object.