Modularisierung in Java: Pakete

Software Entwicklung 1

Annette Bieniusa, Mathias Weber, Peter Zeller

Um zusammengehörende Klassen, Interfaces, etc. gemeinsam zu verwalten, Sichtbarkeiten einzugrenzen und Namenskonflikte zu vermeiden, kann man in Java verwandte Typen in Pakete organisieren. Wie wir bereits in zahlreichen Beispielen gesehen haben, kann man diese Typen dann durch einen Import in einem Programm verfügbar machen. In diesem Kapitel werden wir sehen, wie Pakte in Java erstellt und verwaltet werden können.

Lernziele dieses Kapitels:

- Die Rolle von Modularisierung bei der Programmierung zu erläutern.
- Paketen in Java zu deklarieren und zu organisieren.
- Sichtbarkeitsregeln für Pakete geeignet anzuwenden.

1 Organisation von Klassen- und Interface-Deklaration

Java-Programme bestehen aus ***Typdeklarationen***, d.h. Klassen- und Interface-Deklarationen. Es gibt drei Sprachkonstrukte zur Strukturierung der Typdeklarationen:

- Schachtelung von Klassen (vgl. Kapitel 17)
- Gruppierung von Klassen zu ***Übersetzungseinheiten***; diese entsprechen einer Datei und können mehrere Typdeklarationen zusammenfassen, wobei maximal eine öffentlich sein darf.
- Gruppierung von Übersetzungseinheiten zu Paketen

Java-Programme sind Ansammlungen von übersetzten Typdeklarationen. Die Gruppierung in Übersetzungseinheiten und die Schachtelung von Klassendeklarationen existiert auf der Ebene nicht mehr.
Große Programme bestehen aus hunderten und tausenden von Typendeklarationen, die von unterschiedlichen EntwicklerInnen und Institutionen realisiert und verwaltet werden. Es ist von großer Bedeutung, diese Klassen geeignet zu gruppieren und einzuteilen. Wichtige Aspekte dabei sind:

- Auffinden der Deklarationen und Verstehen des Zusammenhangs

- Übersetzungs-, Installationsorganisation
- Zugriffsrechte, Namensräume
- Pflege und Wartung

Für diese Aufgabenbereiche gibt es Modulkonzepte. ***Module*** sind abgeschlossene funktionale Einheiten, die getrennt voneinander übersetzt und gewartet werden können. Einzelne Module können zu einem größeren Ganzen zusammengesetzt werden und interagieren dabei über Schnittstellen.

1.1 Pakete in Java

Javas ***Pakete*** bieten ein relativ einfaches Modulkonzept.

- Ein Paket hat einen Namen `p`. Der Name besteht ggf. aus mehreren Teilen, die durch einen Punkt voneinander getrennt sind. Paketnamen werden typischerweise klein geschrieben.
- Ein Paket `p` ist üblicherweise in einem Dateiverzeichnis mit Namen `p` abgelegt.

  Beispiel:

  Das Paket `java.lang` ist abgelegt in einem Verzeichnis `.../java/lang`
- Ein Paket ist eine endliche Menge von Übersetzungseinheiten (Dateien); der Paketname erscheint am Anfang der Übersetzungseinheiten.
  Syntax:

  ```
  package  Paketname;
  ```
- Pakete bilden Namensräume: Ist `p` ein Paket, dann können alle Klassen `K` in `p` mittels `p.K` angesprochen werden (***vollständiger Name***); analog für Interfaces. Innerhalb eines Pakets darf ein Klassenname (bzw. Interface-Name) nur einmal vorkommen.
- Es gibt **keine** Paket-Hierarchie/Unterpakete (insbesondere ist z.B. `java.awt.event` kein Unterpaket von `java.awt`).
- Die Schnittstelle eines Java-Pakets besteht aus den öffentlich sichtbaren Elemente, die in dem Paket definiert sind (Modifikator `public`).

Beispiel In der Regel wird die Paketstruktur in die Verzeichnisstruktur des Quellcodes umgesetzt. Angenommen, wir wollen Pakete `se1.list` und `se1.tree` erstellen. Die Listen umfassen dabei die Klassen `LinkedList`, `Node`, `ListIterator`, die Bäume umfassen die Klassen `SearchTree`, `Node`. Wir legen folgenden Dateien an:

```
/home/karlheinz/uni/se1/list/LinkedList.java
/home/karlheinz/uni/se1/list/ListIterator.java
/home/karlheinz/uni/se1/list/Node.java
/home/karlheinz/uni/se1/tree/SearchTree.java
/home/karlheinz/uni/se1/tree/Node.java
```

Die Java-Typen der Schnittstelle sind folgendermaßen definiert:

```
package se1.list;
public class LinkedList {...}
```

bzw.

```
package se1.tree;
public class SearchTree {...}
```

Die Klasse `Node` ist nur innerhalb des Pakets sichtbar, daher ist sie nicht als `public` deklariert (kein Modifikator):

```
package se1.list;
class Node {...}
```

Ein Programm, das den hier definierten Typ `LinkedList` verwenden will, muss diesen Typen importieren, wenn es nicht Teil des Pakets `se1.list` ist:

```
import se1.list.LinkedList;
class LinkedListTest {
  public static void main(String[] args) {
    LinkedList ll = new LinkedList();
    ...
  }
}
```

Die Anweisung `import se1.list.*` importiert alle Typen aus dem Paket `se1.list`. Alternativ kann der Typ auch durch den vollständigen Namen adressiert werden:

```
// keine Import-Anweisung
class LinkedListTest {
  public static void main(String[] args) {
    se1.list.LinkedList ll = new se1.list.LinkedList();
    ...
  }
}
```

Dies ist notwendig, wenn ein weiterer Typ mit gleichem Namen im Programm verwendet werden soll:

```
class LinkedListTest {
  public static void main(String[] args) {
    se1.list.LinkedList ll = new se1.list.LinkedList();
    java.util.LinkedList<String> ls =
      new java.util.LinkedList<String>();
    ...
  }
}
```

Deklaration ohne Zugriffmodifikator Pakete stellen auch einen Zugriffsbereich dar: Alle Programmelemente ohne Zugriffsmodifikator sind paketweit zugreifbar.

Frage 1: Folgende Tabelle listet die verschiedenen Zugriffsmodifkatoren und die Zugreifbarkeit auf. Vervollständigen Sie die Tabelle!

Hat Zugriff auf:	`public`	`private`	`protected`	ohne
Klasse selbst				
(nicht Sub-)Klasse, gleiches Paket				
Subklasse, gleiches Paket				
Subklasse, anderes Paket				
(nicht Sub-)Klasse, anderes Paket				

Frage 2: Gegeben ist folgende (korrekte) Implementierung einer `equals`-Methode für Klasse `C`.

```
1  public class C {
2    private int a;
3    ...
4    public boolean equals(Object o) {
5      if (o == null) {
6        return false;
7      }
8      if (o instanceof C) {
9        C other = (C)o;
10       return other.a == this.a;
11     } else {
12       return false;
13     }
14   }
15 }
```

Erklären Sie, warum der Zugriff auf `other.a` in Zeile 10 erlaubt ist!