2.11 Pakete, Geltungsbereiche & Zugreifbarkeit

Eine Übersetzungseinheit ist die umfassendste Struktur, die man in Java spezifizieren kann. Sie enthält in der Regel den Quellcode für eine Klassen- oder Interface-Deklaration. Mehrere Klassendeklarationen sind möglich.

Beim Aufruf des Compilers **`javac`** werden die Dateinamen derjenigen Übersetzungseinheiten als Kommandozeilen-Argumente übergeben, die in Bytecodes übersetzt werden sollen.

In einem **Paket** fasst man eine Gruppe zusammen arbeitender Klassen (genauer: ihre Übersetzungseinheiten) zusammen. Es entsteht dadurch ein neuer Geltungsbereich für die in dem Paket deklarierten Typen. Jede Java-Übersetzungseinheit gehört genau einem Paket an, das man durch eine **Package-Deklaration** spezifiziert.

Pakete

Die Package-Deklaration legt den Namen des Pakets fest, dem die deklarierten Klassen oder Interfaces angehören sollen. Sie kann nur am Beginn einer Übersetzungseinheit stehen:

```
package util.misc;
class PackProg {
  public static void main(String[] args) {
    System.out.println(„Klasse im Paket util.misc“);
  }
}
```

Diese Deklarationen bewirken beispielsweise, dass die Klasse **`PackProg`** dem Paket **`util.misc`** angehört. **`util.misc`** selbst ist wieder Element des Pakets **`util`**, d. h. Pakete können hierarchisch organisiert werden.

Internetweite Pakete

Für Klassen, die internetweit zur Verfügung gestellt werden sollen, sollten die Paketnamen dem eindeutigen Namensschema für Klassen folgen: sie sollten mit dem in **umgekehrter Reihenfolge** geschriebenen Domain-namen beginnen, also zum Beispiel

`de.uni-karlsruhe.etu.util.misc.`

Das Paket **`java`** ist für Klassen des Java-Kern-Sprachstandards, das Paket **`javax`** für Standarderweiterungen reserviert.

Pakete und Dateien

Gewöhnlich – wie auch vom JDK – wird das Dateisystem benutzt. Für das Abspeichern von Paketen in Dateien wird dann pro Paket bzw. Unterpaket ein Verzeichnis bzw. Unterverzeichnis angelegt, das mit dem entsprechenden Bezeichner aus dem Paketnamen benannt wird.

Im obigen Beispiel liegt **`PackProg.java`** im Unterverzeichnis **`util/misc/`**.

Geltungsbereiche und Zugriffsrechte

Die Organisation von Klassen in Pakete bleibt nicht ohne Auswirkung auf die Geltungsbereiche der Klassen- und Elementnamen; ebenso werden die Zugriffsrechte tangiert.

Alle zusammen gehörenden Programme sollten sich im selben Paket befinden, da sonst auch Zugriffsrechte verletzt werden.

Geltungsbereiche von Klassennamen

Der Geltungsbereich eines Klassennamens besteht aus allen Klassen- und Interfacedeklarationen in allen Übersetzungseinheiten des Pakets, in dem der Name deklariert ist.

Ausgenommen sind die Namen eingebetteter Klassen, die den Klassenrumpf ihrer umgebenden Klasse als Geltungsbereich erhalten.

Interfacenamen gelten wie Klassennamen in allen Klassen- und Interfacedeklarationen ihres Pakets.

Geltungsbereiche der übrigen Namen

- Der Geltungsbereich eines **Klassenelements** ist der gesamte Rumpf der Klassendeklaration. Dabei spielt es keine Rolle, ob es sich um in der Klasse deklarierte oder um geerbte Elemente handelt.

  Analog erhalten die Namen von Interface-Elementen als Geltungsbereich den Rumpf der Interfacedeklaration.

- Der Geltungsbereich einer **lokalen Variablen** ist der Rest des Blocks, in dem die Deklaration enthalten ist.

- Der Geltungsbereich einer **lokalen Variable**, die im **`For-Init`**-Teil einer **`for`**-Anweisung deklariert ist, ist der Rest der **`for`**-Anweisung (**Ausdruck**, **For-Update** und **Anweisung**).

- Der Geltungsbereich eines Methoden- oder Konstruktorparameters ist der gesamte Methoden- bzw. Konstruktorrumpf.

Ein Teil der aufgeführten Konstrukte kann man auch von außerhalb ihres Geltungsbereichs verwenden, wenn man einen qualifizierten Namen verwendet.

Zugriffsrechte

Zugriffsrechte werden durch die Modifzierer **`public`**, **`protected`** und **`private`** bzw. die Standardeinstellungen gesteuert.

- Klassen können **`public`** oder ohne Modifzierer spezifiziert werden.
- Klassenelemente und Konstruktoren können **`public`**, **`protected`**, **`private`** oder ohne Modifzierer spezifiziert werden.
- Interfaces können **`public`** oder ohne Modifzierer spezifiziert werden; ihre Elemente sind implizit **`public`**.

Zugriffsrechte auf Klassen und Interfaces

- Wenn eine Klasse **`public`** deklariert ist, ist sie für jeden Java-Code zugreifbar, für den das Paket, dem sie angehört, zugreifbar ist.
- Wenn eine Klasse nicht **`public`** deklariert ist, erhält sie die Standard-zugriffsrechte und ist nur noch für Java-Code aus demselben Paket, in dem sie enthalten ist, zugreifbar.
- Analog gilt für **`public`** Interfaces, dass sie für sämtlichen Code, der auf ihr Paket zugreifen kann, zugreifbar sind. Ansonsten beschränkt sich ihre Zugreifbarkeit auf Code innerhalb ihres Pakets.

Zugriffsrechte auf Elemente von Klassen

- **`public`**-Elemente oder Konstruktoren sind für jeglichen Java-Code zugreifbar, für den ihre Klasse zugreifbar ist.

  Analoges gilt für die Zugreifbarkeit von Interface-Elementen, die implizit **`public`** sind: Sie sind zugreifbar, wenn das Interface zugreifbar ist.

- **`protected`**-Elemente oder Konstruktoren sind in Subklassen ihrer Klasse zugreifbar, sofern die Klasse selbst zugreifbar ist, sowie allgemein innerhalb des Pakets, das die Deklaration ihrer Klasse enthält. Die Subklasse kann einem anderen Paket angehören.

- **`private`**-Elemente oder Konstruktoren sind nur innerhalb ihrer Klassendeklaration zugreifbar.

- Ohne Verwendung von **`public`**, **`protected`** oder **`private`** sind Elemente und Konstruktoren nur innerhalb ihres Pakets zugreifbar. In diesem Fall spricht man von den **Standardzugriffsrechten**.

Zugriffsrechte, Beispiel 1

```
class A {
  private int i;
  private void m() {System.out.println("A.m");}
}
class B {
  void l() {
    A a = new A();
    a.i = 10;  // Fehler
    a.m();     // Fehler
  }
}
```

Die Klassen **A** und **B** gehören hier demselben unbenannten Standardpaket an. Die Zugriffe scheitern, weil **i** und **m** nur im Rumpf von Klasse **A** zugreifbar sind.

Zugriffsrechte, Beispiel 2 (1)

```
package packi;
class X {
  protected int i;
  protected void m() {System.out.println("X.m"); }
}
```

Zugriffsrechte, Beispiel 2 (2)

```
package packi;
class Y {
  public static void main(String[] args) {
    new Y().l();
  }
  void l() {
    X a = new X();
    a.i = 10;
    a.m();
  }
}
```

Beide Klassen gehören dem Paket **`packi`** an. Die Zugriffe sind möglich, weil **`i`** und **`m`** für beliebigen Code in **`packi`** zugreifbar sind.

Zugriffsrechte, Beispiel 3

```
package packii;
class Z extends packi.X {    // Fehler
  void l() {
    i = 10;
    m();
  }
}
```

Hier scheitert bereits das Deklarieren der Subklasse **`Z`**, weil **`X`** einem anderen Paket angehört und der Klassenname somit nicht zugreifbar ist. Wenn man **`X`** jedoch zur **`public`**-Klasse macht, ist die Deklaration von **`Z`** einschließlich der Zugriffe auf **`i`** und **`m`** korrekt.

Zugriffsrechte, Beispiel 4 (1)

```
package neu;
public class U {
  public int i;
  public void m() {System.out.println("U.m");}
}
```

Zugriffsrechte, Beispiel 4 (2)

```
package alt;
class V {
  public static void main(String[] args) {
    V v = new V();
    v.l();
  }
  void l() {
    neu.U a = new neu.U();
    a.i = 10;
    a.m();
  }
}
```

Beide Klassendeklarationen gehören hier verschiedenen Paketen an. Die Zugriffe in **`V.l`** sind nur möglich, weil sowohl die Klasse **`U`** als auch ihre Elemente **`public`** sind.

Zugriffsrechte und Vererbung

Beim Überschreiben von Methoden ist zu beachten, dass die überschreibende Methode die Zugriffsrechte der überschriebenen Methode nicht reduzieren darf.

Das bedeutet:

- eine **`public`** Methode kann nur **`public`** überschrieben werden,
- eine **`protected`** Methode kann nur **`public`** oder **`protected`** überschrieben werden und
- eine Methode ohne Modifizierer kann nicht **`private`** überschrieben werden.

Zugriffsrechte und Vererbung – Beispiel

```
class X {
  int l() {…}
  public void m() {…}
  public int a = 5;
}

class Y extends X {
  protected int l() {…}
  void m() {…}        // Fehler
  int a = -5;
}
```

Import-Deklarationen

Voll qualifizierte Namen von importierten Klassen und Methoden

```
class QualifizierteNamen {
  public static void main(String[] args)
  throws java.io.IOException {
    java.io.BufferedReader in =
        new java.io.BufferedReader(
        new java.io.InputStreamReader(System.in));
    System.out.print("a? ");
    int a = Integer.parseInt(in.readLine());
    System.out.println("a = " + a);
  }
}
```

Die import-Anweisung

Zur Vereinfachung der Schreibweise bietet Java die Möglichkeit, Namen mit einer Import-Deklaration zu **importieren** und dann einfach die Bezeichner zu verwenden.

```
import java.io.IOException;
import java.io.BufferedReader;
import java.io.InputStreamReader;

class ImportierteNamen {
  public static void main(String[] args) throws IOException {
    BufferedReader in = new BufferedReader(
        new InputStreamReader(System.in));
    System.out.print("a? ");
    int a = Integer.parseInt(in.readLine());
    System.out.println("a = " + a);
  }
}
```

Die `import`-Anweisung mit „*“

Es ist auch möglich, alle als **`public`** deklarierten Typen eines Pakets mit einer einzigen Import-Deklaration, die den Paketnamen mit „*****“ abschließt, zu importieren. Das Paket muss wieder zugreifbar sein. Das vorherige Beispiel vereinfacht sich damit zu:

```
import java.io.*;
class KomplettImport {
  public static void main(String[] args) throws IOException {
    … wie oben
  }
}
```

Java-Archive

Nach Abschluss der Entwicklungsarbeiten packt man üblicherweise alle zu einem Projekt gehörenden und an Anwender auszuliefernden **`class`**-Dateien in ein **Java-Archiv**. Da die Dateien hierbei komprimiert werden, dient dies nicht nur der Übersichtlichkeit, sondern verkürzt auch die Übertragungszeiten im Internet.

In ein Java-Archiv kann man neben Klassen auch Audio- und Image-Dateien und – falls er mit versandt werden soll – Quellcode aufnehmen.

Zum Archivieren verwendet man das **`jar`**-Tool, dessen Syntax analog zum Unix **`tar`**-Kommando gestaltet wurde.

2.12 Interfaces

In Kapitel 2.10 hatten wir gesehen, dass es sinnvoll sein kann, eine Klasse wie **`Kto`** oder **`VersVertrag`** abstrakt zu deklarieren, wenn sie als Superklasse für eine Vererbungsstruktur vorgesehen ist und eine oder mehrere ihrer Methoden nur in Subklassen sinnvoll implementierbar sind.

In der Praxis treten auch Fälle auf, in denen in der Superklasse keinerlei Variablen benötigt werden und keine Methodenimplementierungen (außer einem leeren Rumpf) von Interesse sind. Die Klasse wird dann zur **Protokollklasse** oder **rein abstrakten** Klasse. Zum Beispiel:

```
public abstract class Eingabe {
  public abstract boolean liesBooelan() throws IOException;
  public abstract char liesChar() throws IOException;
  …
}
```

In einer derartigen Situation ist es sinnvoller, ein **Interface** anstelle der Klasse zu deklarieren.

Unterschiede zwischen Interface und Klasse

Im Unterschied zu einer Klassenvererbung kann eine Klasse **mehrere direkte** Superinterfaces haben; ein weiterer Unterschied ist das Fehlen eines gemeinsamen Superinterfaces analog zu **`Object`**.

Interfaces bilden somit eine unabhängig zur Klassenvererbungshierarchie existierende Vererbungsstruktur.

Interface-Deklarationen

Eine Interface-Deklaration besteht aus bis zu fünf Teilen:

- den **optionalen Modifizierern `public`** und **`abstract`**,
- dem Schlüsselwort **`interface`**,
- einem **Bezeichner**, mit dem das Interface benannt wird,
- einem **optionalen `extends`** mit nachfolgender Ausgabe einer Liste von Superinterfaces und
- dem **Interfacerumpf**, der – in { und } eingeschlossen – die Deklarationen einer beliebigen Anzahl von Variablen und Methoden enthält.

Der Geltungsbereich eines Interfacenamens ist, wie bei Klassennamen, das gesamte Paket, in dem das Interface deklariert ist. Dies bedeutet, dass Interfaces und Klassen im selben Paket verschiedene Namen erhalten müssen.

Interface-Methoden

Alle in einem Interface deklarierten Methoden sind implizit **`abstract`**, d. h., bei ihrer Deklaration werden lediglich Ergebnistyp und Signatur der Methode festgelegt, der Methodenrumpf entfällt. Darüber hinaus sind Interfacemethoden implizit **`public`**, die Angabe dieses Modifizierers erübrigt sich somit.

- Eine Interfacemethode darf nicht **`static`** spezifiziert werden (**`static`**-Methoden können nicht abstrakt sein).
- Sie darf auch nicht **`final`** spezifiziert werden – dies würde das Überschreiben unterbinden, was ja gerade beabsichtig ist.

Interface-Methoden - Beispiel

```
// Eingabe.java
import java.io.IOException
public interface Eingabe {
  boolean liesBoolean() throws IOException;
  char liesChar() throws IOException;
  long liesLong() throws IOException;
  double liesDouble() throws IOException;
  String liesString() throws IOException;
}
```

Interface-Variablen

Die Variablen eines Interfaces sind implizit **`public`**, **`static`** und **`final`**, es handelt sich also um klassenspezifische symbolische Konstanten. Die Verwendung der Modifizierer ist zulässig, aber überflüssig. Für alle Variablen muss man bei ihrer Deklaration einen Initialisierer angeben.

Interface-Variablen – Beispiel

```
public interface Ausgabe {
  char BS = '\b', HT = '\t', LF = '\n', FF = '\f', CR = '\r';
  int STD_BREITE = 10;
  void schreibe(String s);
  void schreibe(String s, int breite);
  void schreibe(boolean b);
  void schreibe(boolean b, int breite);
  void schreibe(char c);
  void schreibe(char c, int breite);
  void schreibe(double d);
  void schreibe(double d, int breite);
  void schreibe(long l);
  void schreibe(long l, int breite);
}
```

Die Implementierung von Interfaces

Bei der Deklaration einer Klasse kann festgelegt werden, dass die Klasse ein oder mehrere Interfaces **implementiert**. Hierzu gibt man die Namen dieser Interfaces durch Kommata getrennt vor dem Klassenrumpf an und leitet diese Liste mit dem Schlüsselwort **`implements`** ein.

Im Unterschied zu Superklassen sind jetzt mehrere direkte Superinterfaces zulässig. Die deklarierte Klasse erbt alle Methoden und Konstanten aus den direkten Superinterfaces, die sie implementiert.

Die Implementierung von Interfaces (Beispiel (1))

```
import java.io.*;
public class StandardEingabe implements Eingabe {
  private static BufferedReader in = new BufferedReader(
        new InputStreamReader(System.in));
  public String liesString() throws IOException {
  return in.readLine();
  }
  public boolean liesBoolean() throws IOException {
  return Boolean.valueOf(in.readLine()).booleanValue();
  }
  public char liesChar() throws IOException {
    return in.readLine().charAt(0);
  }
  public long liesLong() throws IOException {
    return Long.parseLong(in.readLine());
  }
  public double liesDouble() throws IOException {
    return Double.parseDouble(in.readLine());
  }
}
```

Die Implementierung von Interfaces (Beispiel (2))

```
import java.io.*;
class StandardEingabeTest {
  public static void main(String[] args) throws
  IOException {
    Eingabe ein = new StandardEingabe();
    System.out.print(“l (long): ");
    long l = ein.liesLong();
    …
  }
}
```

Die Implementierung von Interfaces (Beispiel (3))

Eine andere Implementierung des **`Eingabe`**-Interfaces, zum Beispiel für die Eingabe mit Barcode-Scannern, könnte folgendermaßen aussehen:

```
import java.io.*;

public class ScannerEingabe implements Eingabe {
  public String liesString() throws IOException {
    … Scanner ablesen
  }
  public boolean liesBoolean() throws IOException {
  … Scanner ablesen und Boolean extrahieren
  }
  …
}
```

Vererbung bei Interfaces

Interfaces können in Sub-/Supertyp-Beziehungen stehen, wobei auch hier wieder – wie bei Klassen – das Schlüsselwort **`extends`** verwendet wird. Im Gegensatz zur Vererbung bei Klassen ist es jetzt jedoch zulässig, mehrere Superinterfaces zu deklarieren.

Die nach **`extends`** aufgeführten Interfaces sind die **direkten Superinterfaces** des deklarierten Interfaces. Umgekehrt heißt das deklarierte Interface **direktes Subinterface**. Ein Interface erbt alle Konstanten seiner direkten Superinterfaces, ebenso alle Methoden mit Ausnahme derjenigen, die es überschreibt.

Die Vererbungsstruktur (**`extends`**) zwischen Interfaces ist **unabhängig** von der zwischen Klassen. Die beiden Konzepte werden durch die **`implements`**-Beziehungen zwischen Klassen und Interfaces miteinander verknüpft .

Mehrdeutigkeiten

Dadurch, dass eine Klasse mehrere Interfaces implementieren kann bzw. ein Interface mehrere Superinterfaces haben kann, ist es möglich, dass verschiedene Elemente gleichen Namens vererbt werden.

Wenn es sich dabei um Variablen handelt, muss der Zugriff in der Subklasse bzw. im Subinterface qualifiziert erfolgen, ansonsten ist die Deklaration fehlerhaft. Zum Beispiel:

```
interface A {int a = 5;}
interface B {float a = -1.25f;}
class C {byte a = -12;}
interface T1 extends A, B {
  double t = a;      // Fehler
  double x = A.a;
  double y = B.a;
}
```

(Fortsetzung)

Beispiel für eine mehrdeutige Variable

```
class T2 implements A, B {
  double t = a;      // Fehler
  double x = A.a;
  double y = B.a;
}
class T3 extends C implements A, B {
  double t = a;      // Fehler
  double x = A.a;
  double y = B.a;
  double z = super.a;
}
```

Mehrdeutige Methoden (1)

Bei gleicher Signatur überschreibt die Methode in der Subklasse oder im Subinterface die Methode in der Superklasse bzw. im Superinterface. Zum Beispiel kann **a** hier für **S2**- und **S3**-Objekte aufgerufen werden:

```
interface D {void a(int i);}
interface E {void a(int i);}
class F {
  public void a(int i) {System.out.println("F.a");}
}
interface S1 extends D, E {void a(int i);}

class S2 implements D, E {
  public void a(int i) {System.out.println("S2.a");}
}

class S3 extends F implements D, E { }
```

Mehrdeutige Methoden (2)

Im Fall verschiedener Signaturen wird überladen und gegebenfalls mit anderen Deklarationen zusätzlich überschrieben. Zum Beispiel kann **a** hier für **R2**- und **R3**-Objekte aufgerufen werden:

```
interface U {void a(int i);}
interface V {void a(double d);}
class W {
  public void a(double d) {…}
}
interface R1 extends U, V {void a(int i);}
class R2 implements U, V {
  public void a(int i) {…};
  public void a(double d) {…}
}
```

(Fortsetzung)

Mehrdeutige Methoden (3)

```
class R3 extends W implements U, V {
  public void a(int i) {…}
}
```

Wie immer ist der Versuch, mit gleicher Signatur, aber unterschiedlichem Ergebnistyp zu überschreiben, ein Fehler.

Ein umfangreiches Beispiel (1)

Zum Abschluss dieses Kapitels betrachten wir ein umfangreiches Beispiel. An diesem wird das sinnvolle Überschreiben von Interfacemethoden sowie die Möglichkeit, Objektreferenzen implizit in Referenzen auf Interfaces umzuwandeln und diese dann für polymorphe Aufrufe zu nutzen, demonstriert.

Das Beispiel zeigt, wie ein Getriebe von einem Schaltpult aus gesteuert wird:

- Die Steuerung wird über die Interfacemethode **`fuehreAus`** vorgenommen.
- Die Klassen **`Hoch`**, **`Herunter`**, **`Leerlauf`** und **`Aus`** implementieren das Interface und überschreiben die Interfacemethode **`fuehreAus`**.
- Bei der Initialisierung des **`aktion`**-Felds werden diese Klassentypen jeweils in den Interfacetyp **`Methode`** konvertiert.

Ein umfangreiches Beispiel (2)

```
class Getriebe {
  private final short ANZ_GAENGE = 5;
  private short gang;
  Getriebe() { gang = 0; }
  void hoch() { if (gang < ANZ_GAENGE) ++gang; }
  void herunter() { if (gang > 0) --gang; }
  void leerlauf() { gang = 0; }
  int gang() { return gang; }
}
```

Ein umfangreiches Beispiel (3)

```
import java.io.*;

interface Methode {
  void fuehreAus(Getriebe g);
}

class Pult {
  private Getriebe g;
  private String[] anzeige =  {
    "0.  Ausschalten", "1.  Hochschalten",
    "2.  Herunterschalten", "3.  Leerlauf"};
  private Methode[] aktion = {
    new Aus(), new Hoch(), new Herunter(), new Leerlauf()};
  private BufferedReader in = new BufferedReader(
                new InputStreamReader(System.in));
```

(Fortsetzung)

Ein umfangreiches Beispiel (4)

```
Pult(Getriebe g) throws IOException {
 this.g = g;
 schalte();
}
void schalte() throws IOException {
 int auswahl = 0;
 while ((auswahl = waehle()) != 0)
            aktion[auswahl].fuehreAus(g);
}
int waehle() throws IOException {
     System.out.println("Gang: " + g.gang());
 for (int i = 0; i < anzeige.length; i++)
            System.out.println(anzeige[i]);
            System.out.print("\nAktion wählen: ");
 return Integer.parseInt(in.readLine());
}
```

(Fortsetzung)

Ein umfangreiches Beispiel (5)

```
  public static void main(String[] args) throws
  IOException {
    new Pult(new Getriebe());
  }
}
class Hoch implements Methode {
  public void fuehreAus(Getriebe g) {g.hoch();}
}
class Herunter implements Methode {
  public void fuehreAus(Getriebe g) {g.herunter();}
}
class Leerlauf implements Methode {
  public void fuehreAus(Getriebe g) {g.leerlauf();}
}
class Aus implements Methode {
  public void fuehreAus(Getriebe g) { }
}
```