Packages

Softwareentwicklung II (IB)

Prof. Dr. Oliver Braun

Letzte Änderung: 28.01.2020 17:34

Arbeitsweise

Motivation

- Pro übersetzter Bytecode-Datei (Extension `.class`): Bytecode einer Klasse
- Großes Programm: Viele Bytecode-Dateien
- Organisatorische Probleme:
  - Orientierung im Code
  - Bezüge zwischen Klassen
  - Namenskonflikte zwischen gleich benannten Klassen
  - Abgrenzung von Programmteilen
- Vergleichbar mit Dateien auf einer Festplatte

Idee

- **Package** = Sammlung zusammengehörender Klassen
- Packages benannt mit Identifiern
- Konvention: Englische Namen in kleinen Buchstaben und Ziffern, zum Beispiel `project51`
- Einschränkungen:
  - keine Großbuchstaben
  - keine Interpunktionszeichen
  - keine anderen Sonderzeichen
- Klassennamen in unterschiedlichen Packages unabhängig, keine Kollision

Package-Hierarchie

- Packages bilden Hierarchie (Baumstruktur):
  - In einem Package untergeordnete Packages: **Sub-Packages**
  - Jedes Packages in höchstens einem übergeordneten Package
- Vergleichbar mit Directories in einem Filesystem

Packagepfad (1/2)

- **Packagepfad**: Liste geschachtelter Packagenamen, getrennt mit Punkten
- Beispiele für Packagepfade:

```
project51
project51.frontend
project51.frontend.web
project51.frontend.text
project51.datastore
```

Packagepfad (2/2)

- Innerhalb eines Packages eindeutige Namen für alle „Bewohner“:
  - Subpackages
  - Klassen (konkret, abstrakt)
  - Enums (als spezielle Klassen)
  - Interfaces
- Beispiel: Im Package `project51` darf es keine Klasse namens `frontend` geben, weil schon ein Subpackage dieses Namens existiert
- Packagehierarchie nur äußerlich: Logisch alle Packages gleichrangig flach nebeneinander

Dynamisches Laden von Bytecode

- JVM lädt Klassen erst bei Bedarf (on the fly), nicht beim Programmstart
- Vorteile:
  - Schneller Start
  - Nicht benutzte Klassen werden nicht geladen
- Bedingung: Bytecode effizient lokalisierbar
- Maßnahme: Packagepfad einer Klasse gibt Lage des Bytecodes im Filesystem vor

Abbildung ins Filesystem

- Direkte Zuordnung Packagepfad $\Rightarrow$ Directorypfad im Filesystem (nur eine Möglichkeit unter mehreren)
- Beispiel: Klasse `Rational` im Package `project51.datastore` wird gesucht im

	Directory `project51`,
dort im	Subdirectory `datastore`,
dort die	Datei `Rational.class`

Einschränkugen für Packagenamen

- Problem:
  - Packagenamen sind Java-Identifier
  - Directorynamen sind Namen des Betriebssystems
- Directorynamen unterliegen Einschränkungen des spefizischen File- und Betriebssystems
  - Groß-/Kleinschreibung (Unix signifikant, CD-ROMs keine Unterscheidung)
  - Unicode-Zeichen (Unix ok, Windows-VFAT verboten)
  - Spezielle Namen (Windows verbietet „`CON`“)
  - …
- Besser defensive Benennung

CLASSPATH (1/3)

- Große Filesysteme: Startpunkt der Suche?
- Umgebungsvariable **CLASSPATH** legt Startdirectory fest, unter dem Bytecodedateien gesucht werden
- Kompletter Pfadname für eine Bytecodedatei:

```
    CLASSPATH
+   Packagepfad
+   Klassenname
+   .class
```

CLASSPATH (2/3)

- Beispiel unter den Annahme `CLASSPATH=/home/developer`:
  Pfadname der Klasse `Rational` im Package `project51.datastore`:

`CLASSPATH`	`/home/developer`
Packagepfad	`project51.datastore`
Klassenname	`Rational`

CLASSPATH (3/3)

Insgesamt (Unix):

```
/home/developer/project51/datastore/Rational.class
CLASSPATH      +Packagepfad        +Klassenname.class
```

Organisationsschema

- Ziel: reibungsloser Austausch von Bytecode, unabhängig von der Quelle
- Konvention: Packagepfad aus Internet-Domainnamen ableiten
- **Toplevel-Domain** liefert oberstes Package, Subdomains untergeordnete Packages
- Beispiel: Domainname dieser Fakultät

  `cs.hm.edu`

  Klassen aus dieser Domain unter dem Packagepfad

  `edu.hm.cs`

- Weitere Packageorganisation Sache der einzelnen Institution

Organisation des Laufzeitbibliothek

- Laufzeitbibliothek von Sun folgt weitgehend einem anderen Organisationsschema

  `java` „Standardklassen“ die jede Java-Implementierung liefern muß `java.lang`
  Häufig gebrauchte Standardklassen, wie zum Beispiel `String`, `Math`, `Integer` `javax`
  Kandidaten für künftige Standardklassen (Java extensions) `org.w3c`, `org.xml`, ...
  Weitere Packages der Laufzeitbibliothek folgen dem Organisationsschema nach Domainnamen

- `java.lang` hat Sonderstatus: automatisch verfügbar, alle anderen Packages erst nach explizitem Import

- Eigene Packages besser nicht in `java` oder `javax` einordnen

Umgang mit Packages

Qualifizierte Namen

- **Qualifizierter Name** = Packagepfad + Package-Element
- Beispiel: Qualifizierter Name der Klasse `Random` im Package

  `java.util` `java.util.Random`

- Qualifizierte Namen im gesamten Quelltext als Identifier zulässig

```
java.util.Random random;                // Typangabe
random = new java.util.Random();        // Konstruktoraufruf
System.out.println(random.nextInt());
```

Qualifizierte Namen vs. Elementzugriff (1/2)

- Syntax qualifizierter Namen ähnelt Elementzugriff
- Beispiel:

  `a.b.c`

  könnte bedeuten:

  - Klasse `c` im Package `a.b`
  - Objektvariable `c` in Objektvariable `b` von Objekt `a`
  - Klassenvariable `c` der Klasse `b` im Package `a`

- Compiler erkennt zutreffende Bedeutung

Qualifizierte Namen vs. Elementzugriff (2/2)

- Beispiel: Qualifizierter Name

  `java.lang.Math.PI`

  wird zerlegt in

  - Package `java.lang`
  - Klasse `Math`
  - Klassenvariable `PI`

import-Klausel

- Klassen sind mit qualifizierten Namen über Packagegrenzen hinweg ansprechbar
- Aber: Schreibaufwändig, mühsam, fehlerträchtig
- Einfacher mit **import**-Klausel `import packagepath.name;`
- name im Rest des Quelltextes ohne Packagepfad nutzbar

```
import java.util.Random;
 ...
Random random;
random = new Random();
System.out.println(random.nextInt());
```

Jokerimport (1/2)

- Import aller Klassen eines Packages mit **Jokerzeichen** `*`

  `import packagepath.*;`

- Beispiel:

```
import java.util.*;
 ...
Random random;
random = new Random();
System.out.println(random.nextInt());
```

- Joker kann nur Klassen eines Packages ansprechen

- Nur einmal und nur als letztes Element in einer `import`-Klausel zulässig

Jokerimport (2/2)

- Unzulässig:

```
import *.util;
import java.*.*;
```

- Joker schließt keine Subpackages ein
- Jokerimport führt aber dazu, dass unklar ist, was tatsächlich benötigt ist
    - daher verbietet Checkstyle bei uns den Jokerimport

Klausel vs. Anweisung

- Mehrere `import`-Klauseln zulässig, Reihenfolge ohne Bedeutung

```
import project51.datastore.Rational;
import project51.*;
import project51.frontend.web.*;
```

- `import`-Klauseln stehen am Programmanfang, vor der Klassendefinition
- Bezeichnung **Klausel** (nicht „Anweisung“): werden nur vom Compiler ausgewertet, generieren keinen ausführbaren Code

package-Klausel (1/2)

- `import`-Klauseln regeln Zugriff auf andere Packages
- Gegenstück: **package-Klausel** definiert Packagezugehörigkeit einer Klassendefinition
- Syntax

  `package` packagepath`;`
- Beispiel:

```
package project51.datastore;

class Rational { ... }
```

`package`-Klausel (2/2)

- `package`-Klausel als Erstes im Quelltext (abgesehen von Kommentar), vor `import`-Klauseln und Definitionen
- `package`-Klausel und Pfad im Filesystem müssen übereinstimmen
- Aufruf von Compiler und JVM mit Packagepfad

```
$ javac project51/datastore/Rational.java
$ java project51/datastore/Main
```

Default-Package

- Quelltext ohne `package`-Klausel: **Defaultpackage** (auch: „anonymes Package“)
- Pfad im Filesystem = **CLASSPATH** ohne Zusatz
- Kein Import des Defaultpackages in andere Packages möglich (kein Name!)
- bisher war alles was wir im Praktikum selbst gemacht haben, im Defaultpackage

Namenskollisionen

- Gleiche Namen in verschiedenen Packages können kollidieren
- Explizit qualifizierte Namen eindeutig, problemlos

Eigenes und fremdes Package

- Import mit Joker: Eigenes Package hat Vorrang

```
package a;
import b.*;           // Import mit Joker
class Main {
  new X();          // ok - a.X
}
```

- Expliziter Einzelimport hat Vorrang vor eigenem Package

```
package a;
import b.X;           // Expliziter Import
class Main {
  new X();          // ok - b.X
}
```

Verschiedene fremde Packages

- Definitionen aus verschiedenen anderen Packages ohne Package-Prefix mehrdeutig

  `c/Main.java`:

```java
package c;
import a.*;          // ok
import b.*;          // ok
class Main {
  new X();         // Fehler - mehrdeutige Verwendung:
                   // a.X oder b.X?
}
```

- Imports ok, Verwendung unzulässig

Mehrdeutiger Einzelimport

- Einzelimport aus verschiedenen anderen Packages unzulässig

```
package c;

import a.X;
import b.X; // Fehler - mehrdeutiger Import: a.X und b.X

class Main {
  …
}
```

- Imports unzulässig

Statischer Import (1/2)

- **Statische `import`-Klausel** macht statische Elemente (nur diese) einer Klasse zugänglich
- Schema: `import static` packagepath`.`name`.`identifier`;`
- identifier: Klassenvariable oder statische Methode der Klasse name im Package packagepath

Statischer Import (2/2)

- Beispiel: sin(45^0) ausgeben (vgl. frühere Fassung):

```
import static java.lang.Math.sin;
import static java.lang.Math.PI;
import static java.lang.System.out;

class Sin {
  public static void main(String[] args) {
    System.out.println(Math.sin(Math.PI/4));
    out.println(sin(PI/4));
  }
}
```

Statischer Jokerimport

- Jokerzeichen `*` importiert alle statischen Elemente einer Klasse

  `import static` packagepath.name`.*;`

- Beispiel:

```
import static java.lang.Math.*;
import static java.lang.System.*;


class Sin {…}
```

- Einschränkung: Kein Import aus dem anonymen Default-Package, auch nicht statisch

Statischer **`import`** von Enums (1/2)

- Anwendung für statischen Jokerimport: Enums
- Beispiel (siehe oben):

```
package project51;
public enum Color {Red, Green, Blue, Yellow}
```

Statischer **import** von Enums (2/2)

- Elementzugriff nach statischem Import

```
import static project51.Color.*;

class ColorMain {
  public static void main(String... args) {
    Color color = valueOf(args[0]);
    if(color == Blue) {
      System.out.printf("%s is one out of %d colors%n",
          color,
          values().length);
    }
  }
}
```

Einzel- vs. Jokerimport

- Jokerimport bequem, aber heikel
- Beispiel: Welche Namen werden hier importiert?

```
import java.util.*;
```

- Besser Einzelimports, selbst wenn zahlreich
- Importlisten oft von Werkzeugen automatisch verwaltet

Zugriffsschutz

package und **`public`**

- Modifier `private`: Zugriff beschränkt auf die eigene Klasse (Datenkapselung)
- Ohne Modifier (package): Zugriff für alle Klassen im gleichen Package
- Modifier `public`: Zugriff aus beliebigen anderen Packages

Regeln

- Möglichst restriktiver Zugriffsschutz verhindert unerwünschte Abhängigkeiten und Zugriffe
- Schrittweises Einstellen des Zugriffsschutzes:
  1. Alle Definitionen `private`
  2. `private` wegnehmen bei Elementen, die von anderen Klassen im gleichen Package gebraucht werden
  3. `protected` anfügen bei Elementen, die in abgeleiteten Klassen genutzt werden dürfen
  4. `public` anfügen bei Definitionen, die für beliebige „Kunden“ bereitgestellt werden

Archivdateien

Jar-Files (1/2)

- Bytecode in einer gepackten Archivdatei = alternative Organisationsform für Packages
- Archivformat **Jar** (java archive), Dateien = „Jar-Files“, Extension `.jar`
- Technisch: Zip-Files mit Erweiterungen
- Vorteile gegenüber Directories:
  - Leicht zusammen zu halten
  - Kompression für effiziente Übertragung
  - Metainformation bzgl. Inhalt
- Nachteile:
  - Kein direkter Zugriff auf Bestandteile

Jar-Files (2/2)

    - Kompression und Expansion kostet Rechenzeit

- Packagehierarchie in Jar-Files unverändert
- JVM findet Bytecode in Jar-Files
- Jar-Files im `CLASSPATH`: Gleichrangige Eintrittspunkte für Packagepfade `CLASSPATH=/somewhere/project51.jar`
- Dateien aller Arten in Jar-Files erlaubt, JVM wertet nur Bytecode aus

`jar`-Tool

- Kommandozeilenwerkzeug `jar` zur Bearbeitung von Jarfiles
- Aufgaben:

  `jar -cf` jarfile file file … Packt die angegebenen Dateien in ein neues Jarfile (`c` = create). • Jokerzeichen sind erlaubt, Directories werden rekursiv eingepackt.

  `jar -tf` jarfile
  Liste des Inhalts (`t` = table of content) `jar -xf` jarfile

  `jar -xf` jarfile file file …
  Packt alle bzw. die angegebenen Dateien aus (`x` = extract). Directories werden rekonstruiert.

- Aufruf mit zusätzlichen Schalter `v` (= verbose): Protokollausgaben auf dem Bildschirm
- Aufruf ohne Parameter: kurze Bedienungsanleitung

Metainformation

- **Metainformation** (= Verwaltungsdaten, Information über das Jarfile) in der Datei `META-INF/MANIFEST.MF`
- Textdatei mit Zeilen der Form name`:`value
- Wird automatisch angelegt mit Standard-Einträgen der Art

```
Manifest-Version: 1.0
Created-By: 1.5.0_04 (Sun Microsystems Inc.)
```

- Viele Einträge mit fester Bedeutung
- Zusätzliche Einträge erlaubt

Beispiel `Main-Class`

- Eintrag `Main-Class` legt Klasse mit der `main`-Methode fest
- Beispiel: Programm `Hello`
- Vorbereitung: Temporäre Textdatei `meta.tmp` erstellen mit einer Zeile:

```
Main-Class: Hello
```

- Jar-File mit zusätzlicher Metainformation erzeugen:

```
$ jar -cfm hello.jar meta.tmp Hello.class
```

- Bestehendes Jar-File um Metainformation erweitern (`-u` = update):

```
$ jar -ufm hello.jar meta.tmp
```

- Programm im Jar-File starten (wird nicht ausgepackt):

```
$ java -jar hello.jar
Hello, World!
```

Eintrittsklasse

- Definition der `main`-Klasse in einem Jar-File oft gebraucht
- Kurzform mit Schalter „`-e`“ (entry class):

```
$ jar -cef Hello hello.jar
```

  legt Eintrittspunkt `Hello.main` fest

- Aufruf des Jar-Files, wie vorher:

```
$ java -jar hello.jar
Hello, World!
```

- Direkter Aufruf von Jarfiles mit geeignetem geeignetem Betriebssystem:

```
$ hello.jar
Hello, World!
```