Objektorientierte Programmierung

Kapitel 15: Quelldateien, Pakete, Zugriffsschutz

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2018/19

http://www.informatik.uni-halle.de/~brass/oop18/

Inhalt

Klassen und Quelldateien (1)

- Für kleine Programme (aus sehr wenigen Klassen) kann man alle Klassen in eine einzige `.java`-Quelldatei schreiben.

  Der Compiler erzeugt für jede Klasse eine eigene `.class`-Datei mit JVM-Bytecode. Bei mehreren Klassen wird es übersichtlicher, wenn man für das Programm ein eigenes Verzeichnis anlegt.

- Ansonsten kann man grundsätzlich nichts falsch machen, wenn man jede Klasse in eine eigene Quelldatei schreibt (Datei `C.java` für Klasse `C`).

  Der Dateiname (ohne die Extension `.java`) sollte exakt mit dem Klassennamen übereinstimmen, inklusive Groß-/Kleinschreibung.
  Wenn die Klasse mit dem Schlüsselwort `public` markiert ist (s.u., Folie 39), erzwingt der Compiler diese Regel. Daher kann bei mehreren Klassen in einer Quelldatei höchstens eine `public` sein.

Klassen und Quelldateien (2)

- Wenn Klassen `D1, ..., Dn` ausschließlich von der Klasse `C` verwendet werden, bietet es sich an, diese Klassen mit in die Quelldatei `C.java` zu schreiben.
  - Man reduziert so die Anzahl der Quelldateien, und macht klar, dass `D1, ..., Dn` nur im Kontext der Klasse `C` relevant sind.
  - Dies ist aber Geschmackssache: Manche Programmierer würden dennoch `D1.java, ..., Dn.java` vorziehen.
  - Möglicherweise wären geschachtelte Klassen die beste Lösung (in dieser Vorlesung nicht behandelt).
- Es wäre dagegen sehr unübersichtlich, wenn die Klasse `D` in einer Quelldatei `X.java` verwendet wird, und weder in `X.java` selbst noch im üblichen `D.java` definiert ist.

Mehrere Quelldateien: Compilierung (1)

- Man kann beim Aufruf des Compilers auch mehrere Quelldateien angeben:

  ```
  javac C.java D.java
  ```

  Die Reihenfolge der Quelldateien spielt keine Rolle. Der Compiler meldet Fehler wegen nicht definierten Klassen erst, nachdem er alle angeschaut hat. So sind auch wechselseitige Referenzen möglich (`C` benutzt `D`, `D` benutzt `C`).

- Bei Programmen mit wenigen Quelldateien ist die einfachste Lösung, immer alle neu zu compilieren:

  ```
  javac *.java
  ```

  Da das Starten des Compilers einen gewissen Overhead erfordert, ist dieser Aufruf schneller, als `javac` einzeln für jede Quelldatei einzeln aufzurufen.

- Wenn man nur eine von vielen Quelldateien verändert hat, wird man nur die veränderte Quelldatei neu compilieren ($\rightarrow$).

Mehrere Quelldateien: Compilierung (2)

- Beispiel: Klasse `D` (definiert in `D.java`) benutzt Klasse `C` (in `C.java`). Wenn man `C.java` ändert und nur diese Datei neu compiliert, könnten folgende Probleme auftreten:
  - Wenn `D` eine Methode von `C` benutzt, die nach der Änderung nicht mehr angeboten wird (oder deren Argument-/Resultattypen geändert wurden), bemerkt man den Fehler erst zur Laufzeit.

    Wenn die nicht mehr existierende Methode aufgerufen wird. Dies kann von Eingabewerten abhängen und wird deswegen nicht unbedingt beim Testen gefunden.
  - Eine in `C` definierte Konstante (mit `static final` und einfacher Initialisierung) wird in `D` eincompiliert: Es gibt keine Fehlermeldung, aber `D` verwendet den alten Wert.

Mehrere Quelldateien: Compilierung (3)

- Man sollte in gewissen Abständen (spätestens vor der Abgabe), alle Quelldateien neu compilieren.
- Entwicklungsumgebungen wie `Eclipse` und `Netbeans` sorgen automatisch dafür, dass die benötigten Dateien neu compiliert werden.

  Es gibt außerdem Werkzeuge, die nur auf diese Aufgabe spezialisiert sind, z.B. das klassische `make`, oder Apache Ant [http://ant.apache.org/], oder Maven [http://maven.apache.org/].
- Bei direkt abhängigen Datei erkennt der Java-Compiler selbst, wenn Sie neu compiliert werden müssen:
  - Wenn `D` die Klasse `C` verwendet, und `C.java` ist neuer als `C.class`, dann wird beim Compilieren von `D.java` die Datei `C.java` automatisch auch compiliert.

Verwendung der .class-Dateien

- Wenn die Klasse `D` die Klasse `C` verwendet, braucht man zum Compilieren von `D.java`
  - nur die Datei `C.class` (compilierter Bytecode),
  - aber nicht unbedingt die Datei `C.java` (Quellcode).
- Für Bibliotheken werden üblicherweise nur `class`-Dateien ausgeliefert (und eine Dokumentation), aber kein Quellcode.

  Meist in Form einer `.jar`-Datei (s.u.).
- Zur Laufzeit (bei Ausführung des Programms) werden die `.class`-Dateien erst geladen (und mit dem Rest des Programms verknüpft), wenn sie wirklich benutzt werden.

  Die benötigten Dateien werden von Compiler und JVM über den "Classpath" gefunden (Umgebungsvariable `CLASSPATH` oder Option `-cp`, Default "`.`").
  Außerdem Bootstrap Klassen: `System.getProperty("sun.boot.class.path")`.

Jar-Archive (1)

- Weil ein Programm meist aus vielen `class`-Dateien besteht, werden diese normalerweise als Archiv ausgeliefert, also in einer einzigen Datei verpackt.
- Zum "Java Development Kit" gehört das Programm `jar`, mit dem Archive erstellt werden können.

  [http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jar.html]
  Tutorial: [http://docs.oracle.com/javase/tutorial/deployment/jar/]
- Die übliche Endung ("extension") für solche Archiv-Dateien ist "`.jar`".

  Tatsächlich handelt es sich um ein ZIP-Archiv. Man kann die Archive daher auch mit Programmen wie `zip` und `unzip` erstellen/entpacken. Allerdings enthalten `jar`-Archive eine spezielle Datei `MANIFEST`, in der z.B. die Hauptklasse (mit der Methode `main`) festgelegt ist.

Jar-Archive (2)

- Man legt ein `jar`-Archiv an mit folgendem Befehl:

```
jar cfe prog.jar MainClass *.class
```

  Die Option "`c`" steht für "create" (neues Archiv erzeugen), der Buchstabe "`f`" für "file" (erlaubt die Angabe des Archiv-Namens "`prog.jar`" nach den Optionen), und der Buchstabe "`e`" für "entry point" (erlaubt die Angabe der Klasse "`MainClass`" mit der `main`-Methode). Anschließend werden alle Dateien und Verzeichnisse angegeben, die in das Archiv verpackt werden sollen. Das müssen nicht nur `class`-Dateien sein, sondern können z.B. auch Bilder oder andere Resourcen sein, die das Programm benötigt. Wenn man den Buchstaben "`v`" (für "verbose") zu den Optionen hinzufügt, werden die Dateien nochmal aufgelistet. Den Inhalt eines Archives kann man sich anschauen mit "`jar tvf prog.jar`" ("`t`: "table of contents").

- Man führt ein Programm in einem `jar`-Archiv aus mit

```
java -jar prog.jar
```

Inhalt

Pakete: Einführung, Motivation (1)

- Ein Paket (engl. “package”) ist eine Zusammenfassung von Dateien (und damit Klassen, aber ggf. auch weiterer Resourcen wie Bild-Dateien), die
  - inhaltlich zusammenhängen, also einem gemeinsamen größeren Zweck dienen,
  - ggf. als Ganzes in anderen Programmen wiederverwendet werden (wie eine Unterprogramm-Bibliothek).
- Ein Paket ist also eine Strukturierungsebene
  - oberhalb der einzelnen Klassen, aber
  - unterhalb eines größeren Anwendungsprogramms.
- Klassennamen müssen nur innerhalb eines Paketes eindeutig sein (Vermeidung von Namenskonflikten).

Pakete: Einführung, Motivation (2)

- Für Pakete ist eine hierarchische Struktur möglich (sie entsprechen Datei-Ordnern).

  Es kann also ein Paket in einem anderen Paket angelegt werden. Paketnamen sind entsprechend hierarchisch strukturiert, die Teile werden durch "`.`" getrennt.

- Ein Paket wird meist von einem einzelnen Programmierer erstellt, oder von einem Team, das eng zusammenarbeitet.

  Deswegen ist die Voreinstellung bei Java, das innerhalb eines Paketes beliebige Zugriffe möglich sind (s.u.). Aber auch ein einzelner Programmierer macht Fehler, kann sich nicht mehr erinnern, oder verliert die Übersicht. Deswegen sollte man alles, was nicht von anderen Klassen aus zugreifbar sein muss, als "`private`" deklarieren. Wenn sich später herausstellt, dass man darauf doch zugreifen muss, kann man diese Einstellung ja ändern, oder entsprechende Zugriffsmethoden einführen.

Zuordnung von Klassen zu Paketen (1)

- Eine Quelldatei wird einem Paket "P" zugeordnet durch die Deklaration

  ```
  package P;
  ```

  ganz zu Anfang der Datei.

  Also vor der oder den Klassen-Deklarationen. Kommentare wären natürlich auch vor der package-Deklaration möglich.

- Die Quelldatei muss dann auch in einem Verzeichnis "P" stehen.

  Allgemein hängt es vom "Class Loader" ab, wie er die Klassen findet.
  Es ist aber üblich, dass er Klassen, die Paketen zugeordnet sind, in einem Verzeichnis sucht, dass dem Paketnamen entspricht. Dabei werden hierarchisch strukturierte Paket-Namen entsprechend in Unterverzeichnissen gesucht, z.B. würde die Klasse "C" im Paket "P.Q" der Datei "P/Q/C.class" entsprechen.

Zuordnung von Klassen zu Paketen (2)

- Man ruft `java` und `javac` vom übergeordneten Verzeichnis aus auf.

  Z.B. "`javac P/C.java`" und "`java P/C`" oder "`java P.C`". Der Paket-Name steht in der `class`-Datei. Selbst wenn man im Verzeichnis `P` ist, funktioniert "`java C`" nicht. Das Compilieren ist dagegen auch jeweils im Unterverzeichnis möglich. Wenn man aber auf Klassen in anderen Paketen zugreift, muss man den `CLASSPATH` entsprechend setzen, z.B. "`javac -cp ".:.." C.java`" (UNIX) bzw. "`-cp .;..`" (Windows). Damit die Klasse `C` im Paket `P` gefunden wird, muss der `CLASSPATH` ein Verzeichnis enthalten, das ein Unterverzeichnis `P` hat, in dem `C.class` steht. Wenn man nichts angibt, besteht der `CLASSPATH` aus dem aktuellen Verzeichnis "`.`". Die Standard-Pakete werden seit dem JDK 1.2 unabhängig vom `CLASSPATH` gefunden (Systemeigenschaft "`sun.boot.class.path`"). Anstatt den Pfad bei jedem Kommando anzugeben, kann man auch die Environment Variable setzen, z.B. "`setenv CLASSPATH /home/brass/oop`" bzw. "`set CLASSPATH=c:\classes`". [http://docs.oracle.com/javase/1.4.2/docs/tooldocs/findingclasses.html].

Zuordnung von Klassen zu Paketen (3)

- Ohne `package`-Deklaration gehört die Klasse zum Default-Paket oder anonymen Paket.
- Das funktioniert gut, solange man relativ kleine Programme zum eigenen Gebrauch entwickelt.
- Will man Klassen im Internet austauschen, so sollten sie unbedingt in einem Paket "verpackt" werden.
- Um weltweit eindeutige Paketnamen zu bekommen, ist es üblich, den umgekehrten Domain-Namen der Firma als Präfix zu verwenden, z.B. "`com.sun.eng`".

  Das führt allerdings zu entsprechend tief geschachtelten Verzeichnis-Hierarchien. Solange man keinen weltweiten Austausch anstrebt, ist das nicht nötig.

Zugriff auf Klassen in Paketen (1)

- Eine Klasse "`C`" im Paket "`P`" kann man im Programm mit "`P.C`" ansprechen.
- Wenn man diese Klasse öfter braucht, kann man auch zu Anfang der Quelldatei folgende Deklaration schreiben:

```
import P.C;
```

  Dies muss nach der `package`-Deklaration (soweit vorhanden) stehen, und vor der Klassen-Deklaration.

- Anschließend kann man die Klasse einfach mit dem Namen "`C`" ansprechen.
- Es ist auch möglich, alle Klassen aus einem Paket unter ihrem einfachen Namen verfügbar zu machen:

```
import P.*;
```

  Der Compiler fügt automatisch "`import java.lang.*;`" zum Programm hinzu.

Zugriff auf Klassen in Paketen (2)

Beispiel:

- Z.B. haben wir die Klasse "`java.util.Scanner`" verwendet, also die Klasse "`Scanner`" im Paket "`java.util`".
- Man kann den vollen Typ-Namen verwenden:

```
java.util.Scanner s =
        new java.util.Scanner(System.in);
```

- Wenn man das als umständlich empfindet, kann man eine Import-Deklaration verwenden:

```
import java.util.Scanner;
```

- Statt dem vollen Namen der Klasse kann man jetzt auch ihren einfachen Namen verwenden:

```
Scanner s = new Scanner(System.in);
```

Zugriff auf Klassen in Paketen (3)

Beispiel, Forts.:

- Falls man viele `import`-Deklarationen für Klassen aus dem Paket `java.util` braucht, kann man festlegen, dass alle sonst nicht definierten Klassen dort gesucht werden sollen:

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

- Damit verliert man natürlich etwas Kontrolle:
  - Eventuell werden unabsichtlich Klassen aus dem Paket benutzt (bei Tippfehlern etc.).
  - Falls man das für mehrere Pakete macht, kann es (auch später noch bei Erweiterung der Pakete) zu Namenskonflikten zwischen Klassen dieser Pakete kommen, beim Compilieren des eigenen Programms gibt es dann plötzlich einen Fehler.

Zugriff auf Klassen in Paketen (4)

Feinheit:

- Wenn es mehrere Klassen mit dem gleichen Namen "`C`" gibt, wird die erste in folgender Reihenfolge genommen:
  - Die aktuelle Klasse und ihre Oberklassen.
  - In der aktuellen Klasse geschachtelte Klassen.
  - Explizit importierte Klassen ("single type import").

    Man kann natürlich nicht eine Klasse importieren, die genauso heisst wie eine Klasse in der aktuellen Quelldatei.
  - Andere Klassen aus dem aktuellen Paket.
  - Mit "`*`" importierte Klassen ("import on demand").

    Falls zwei mit "`*`" importierte Pakete eine Klasse mit gleichem Namen enthalten, kann man sie nicht mit einfachem Namen ansprechen.

Zugriff auf Klassen in Paketen (5)

Statische Imports:

- Es ist auch möglich, statische Komponenten von Klassen unter ihrem einfachen Namen verfügbar zu machen.
- Wenn man z.B. auf die Konstante `PI` in der Klasse `Math` im Paket `java.lang` sehr häufig zugreifen muss, kann man Folgendes schreiben:

```
import static java.lang.Math.PI;
```

- Es gibt auch davon eine "Wildcard"-Version:

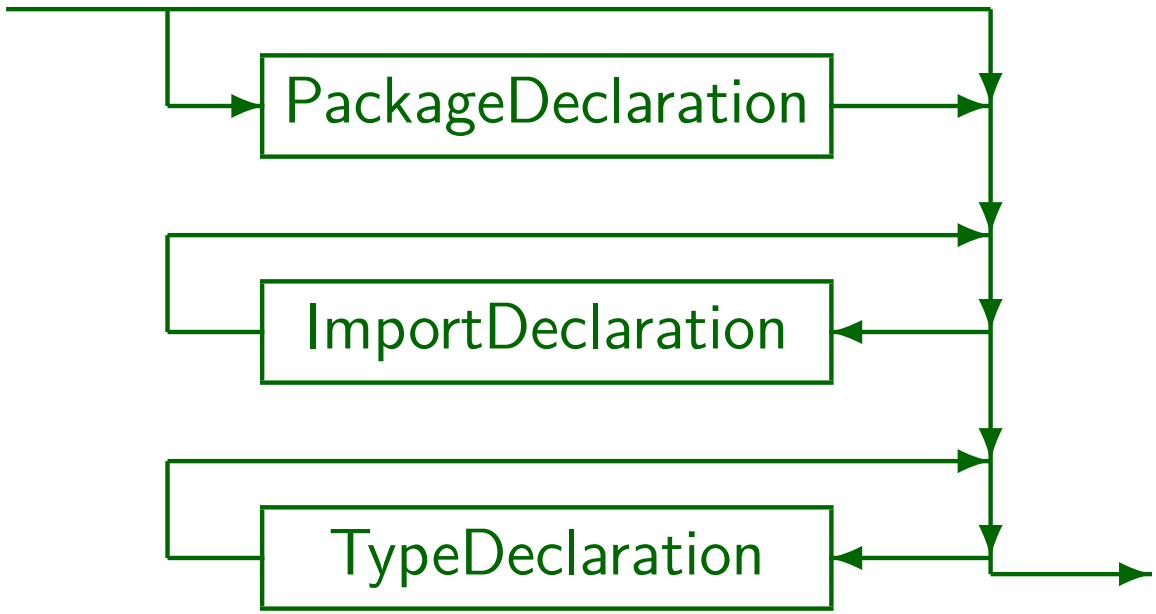
```
import static java.lang.Math.*;
```

- Dann kann man auch alle mathematischen Funktionen aus dieser Klasse unter ihrem einfachen Namen ansprechen.

Inhalt

Quelldateien: Syntax (1)

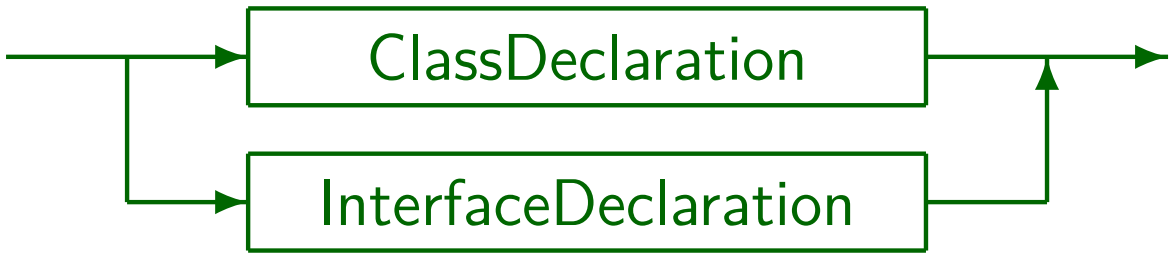
- CompilationUnit:



- Eine Quell-Datei (Einheit der Übersetzung) besteht also aus:
  - Am Anfang eine `package`-Deklaration (optional),
  - dann `import`-Deklarationen (null oder mehr),
  - dann Typ-Deklarationen (z.B. Klassen) (null oder mehr).

Quelldateien: Syntax (2)

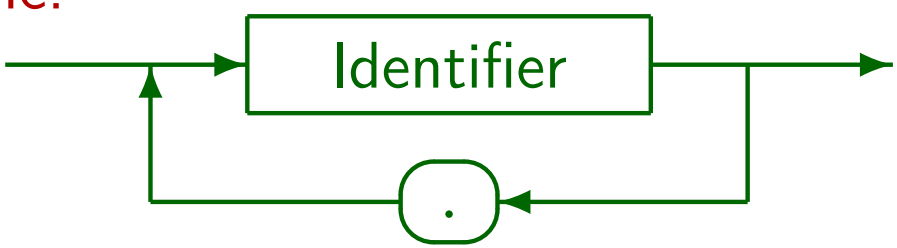
- TypeDeclaration:



- PackageDeclaration:

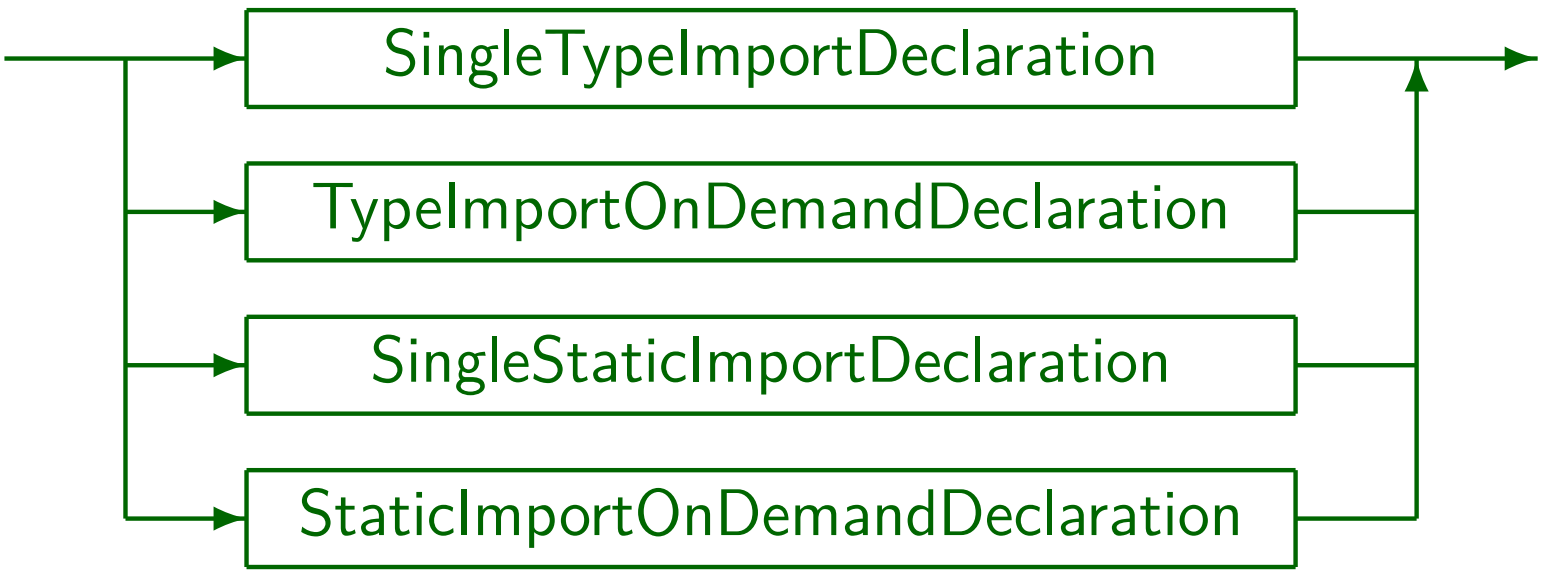
Annotationen werden in dieser Vorlesung nicht behandelt (außer `@Override`).

- PackageName:

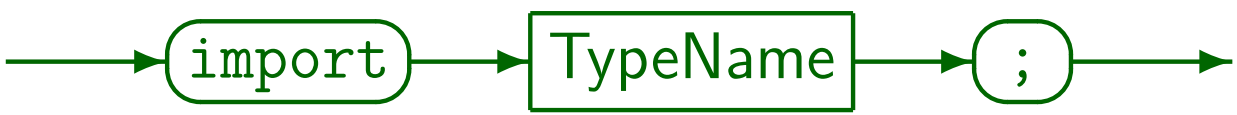


Quelldateien: Syntax (3)

- ImportDeclaration:



- SingleTypeImportDeclaration:



Quelldateien: Syntax (4)

- TypeImportOnDemandDeclaration:

Ein Typ-Name wäre hier nur für geschachtelte Klassen interessant, die werden aber in dieser Vorlesung nicht behandelt.

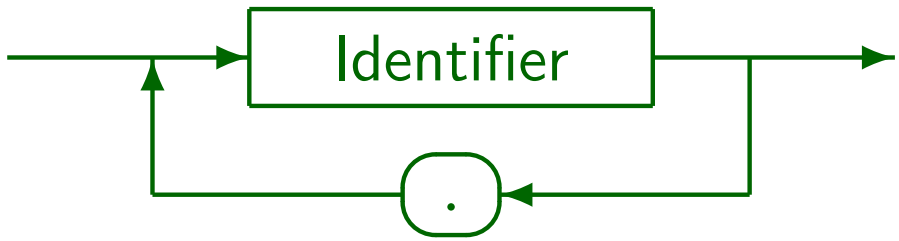
- SingleStaticImportDeclaration:

Mit dieser Variante können z.B. in einer anderen Klasse definierte Konstanten unter ihrem einfachen Namen bekannt gemacht werden.

- StaticImportOnDemandDeclaration:

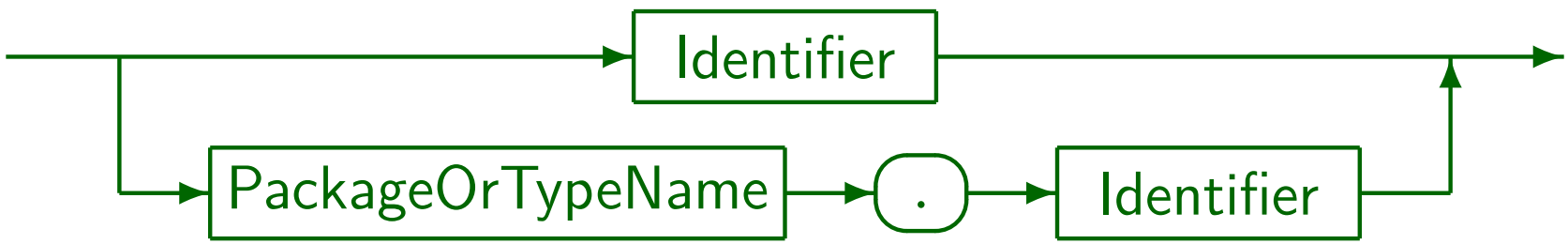
Quelldateien: Syntax (5)

- PackageOrTypeName:



Wenn "PackageOrTypeName" verwendet wird, ist syntaktisch nur klar, dass sowohl ein Paket-Name wie ein Typ-Name (z.B. Klasse) möglich wäre. Beides ist einfach eine Folge von Bezeichnern ("Identifier"). In der Java Spezifikation [http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf] gibt es dann Regeln, um eventuelle Mehrdeutigkeiten aufzulösen (Abschnitt 6.5.4).

- TypeName:



Inhalt

Zugriffsschutz/Sichtbarkeit (1)

- Für Komponenten einer Klasse sowie Konstruktoren kann man Zugriffe von außerhalb erlauben oder verbieten mit den folgenden Schlüsselworten ("access modifiers"):
  - `private`: Die Komponente/der Konstruktor ist nur innerhalb der Klasse zugreifbar/sichtbar.

    D.h. nur in Programmcode, der in der Klasse steht, also Rümpfe von Methoden der Klasse und Initialisierungs-Blöcke.
  - (keine Angabe): Zugriffe sind von allen Klassen des Paketes erlaubt.

    Dieser Zugriffsschutz wird "package", "package-scope" oder "default access" genannt. Es gibt kein Schlüsselwort dafür.
  - `protected`: Zugriffe sind von allen Klassen des Paketes erlaubt, und von Unterklassen (auch außerhalb des Paketes).
  - `public`: Keine Einschränkungen.

Zugriffsschutz/Sichtbarkeit (2)

Zur Syntax:

- Die "Access Modifier" `private`, `protected`, `public` müssen in jeder Komponenten-Deklaration einzeln angegeben werden.

  Wenn man nichts angibt, bekommt man den Default "package".

  Das ist anders als in C++: Dort schreibt man z.B. "`public:`", und das gilt dann für alle folgenden Komponenten, bis man den Zugriffsschutz explizit ändert.

- Die Angabe muss am Anfang der Deklaration erfolgen (vor dem Typ bzw. dem Konstruktornamen).

- Innerhalb der "Modifier", zu denen z.B. auch "`static`" und "`final`" zählen, ist die Reihenfolge egal.

  Es ist aber üblich, den Zugriffsschutz nach Annotationen (s.u.) und vor allen anderen Modifiern zu schreiben. Z.B. ist "`public static void main(...)`" die normale Schreibweise, aber "`static public void main(...)`" geht auch.

Zugriffsschutz/Sichtbarkeit (3)

- Inhalt der Datei "`Test.java`" im Verzeichnis "`P`":

```
(1)  package P;
(2)
(3)  public class Test {
(4)      private   int priv;
(5)                int pack;
(6)      protected int prot;
(7)      public    int publ;
(8)  }
```

- In dieser Klasse sind vier Attribute mit den vier möglichen Festlegungen für den Zugriffsschutz deklariert.

  Natürlich kann man die Attribute nicht so nennen wie den Zugriffsschutz: `private`, `protected`, `public` sind reservierte Worte. Deswegen die Abkürzung. Wenn man nichts angibt, erhält man die "package" oder "default visibility".

Zugriffsschutz/Sichtbarkeit (4)

- Inhalt der Datei "SamePackage.java" im Verzeichnis "P":

```
(1)  package P;
(2)
(3)  class SamePackage {
(4)      public static void main(String[] args) {
(5)          Test x = new Test();
(6)          System.out.println(x.priv); // Fehler
(7)          System.out.println(x.pack); // Ok
(8)          System.out.println(x.prot); // Ok
(9)          System.out.println(x.publ); // Ok
(10)     }
(11) }
```

- Bei Klassen im gleichen Paket kann man also auf alle Komponenten zugreifen, die nicht als `private` deklariert sind.

Zugriffsschutz/Sichtbarkeit (5)

- Inhalt der Datei "`Subclass.java`" im Verzeichnis "`Q`":

```
(1)  package Q;
(2)  import P.Test;
(3)
(4)  class Subclass extends Test {
(5)      public static void main(String[] args) {
(6)          Test x = new Test();
(7)          System.out.println(x.priv); // Fehler
(8)          System.out.println(x.pack); // Fehler
(9)          System.out.println(x.prot); // Fehler
(10)         System.out.println(x.publ); // Ok
(11)     }
(12) }
```

- Von außerhalb des Paketes kann man nur auf öffentliche (`public`) Komponenten von `Test`-Objekten zugreifen.

Zugriffsschutz/Sichtbarkeit (6)

- Inhalt der Datei "`Subclass2.java`" im Verzeichnis "`Q`":

```
(1)  package Q;
(2)  import P.Test;
(3)
(4)  class Subclass2 extends Test {
(5)      public static void main(String[] args) {
(6)          Subclass2 x = new Subclass2();
(7)          System.out.println(x.priv); // Fehler
(8)          System.out.println(x.pack); // Fehler
(9)          System.out.println(x.prot); // Ok
(10)         System.out.println(x.publ); // Ok
(11)     }
(12) }
```

- Für Objekte der Subklasse kann man dagegen auch auf in `Test` deklarierte `protected`-Komponenten zugreifen.

Zugriffsschutz/Sichtbarkeit (7)

- Inhalt der Datei "`OtherClass.java`" im Verzeichnis "`Q`":

```
(1)  package Q;
(2)  import P.Test;
(3)
(4)  class OtherClass {
(5)      public static void main(String[] args) {
(6)          Test x = new Test();
(7)          System.out.println(x.priv); // Fehler
(8)          System.out.println(x.pack); // Fehler
(9)          System.out.println(x.prot); // Fehler
(10)         System.out.println(x.publ); // Ok
(11)     }
(12) }
```

- Von außerhalb des Paketes kann man nur auf öffentliche (`public`) Komponenten zugreifen.

Zugriffsschutz/Sichtbarkeit (8)

Zu "`protected`":

- Der Zugriffsschutz "`protected`" ist schwächer als wenn man gar nichts angibt: Aus dem Paket sind alle Zugriffe möglich.
- Wichtig wird "`protected`" also nur, wenn es Subklassen zur aktuellen Klasse in anderen Paketen geben kann.

  Weil man z.B. Basisfunktionalität oder eine Dummy-Implementierung schon im eigenen Paket hat, aber der Programmierer, der das Paket nutzen will, bestimmte Methoden überschreiben (selbst implementieren) muss (s. Kap. 12).

- Es sind nicht beliebige Zugriffe im Programmcode der Subklasse möglich, sondern nur für Objekte der Subklasse.

  Siehe obiges Beispiel. Es spielt dabei nur der statische Typ eine Rolle, da der Zugriffsschutz zur Compilezeit geprüft wird.

Zugriffsschutz/Sichtbarkeit (9)

Zusammenfassung:

Zugriffsschutz	`private`	(default)	`protected`	`public`
eigene Klasse	ok (1)	ok	ok	ok
andere Klasse gleiches Paket	nein	ok	ok	ok
Subklasse anderes Paket	nein	nein	siehe (2)	ok
andere Klasse anderes Paket	nein	nein	nein	ok

(1) Nur für Objekte, deren statischer Typ die Klasse selbst ist.

(2) Nur für Objekte, deren statischer Typ die Subklasse ist.

Hinweis zu Unterklassen

- Der Zugriffsschutz in einer Unterklasse kann gegenüber der Oberklasse nicht eingeschränkt werden.

  Sonst wäre das Substitutionsprinzip verletzt: Man kann ein Objekt der Unterklasse ja überall verwenden, wo ein Objekt der Oberklasse korrekt wäre.

- Wenn also eine Methode in der Oberklasse "`public`" ist, muss sie auch in der Unterklasse "`public`" sein.

- Das gilt entsprechend auch für Interfaces:

  - Dort sind alle Methoden implizit "`public`".
  - Sie müssen dann in jeder Klasse, die das Interface implementiert, "`public`" sein.

Zugriffsschutz für Klassen (1)

- Damit Klassen in einem fremden Paket verwendet werden können, müssen sie mit dem Schlüsselwort "`public`" markiert werden:

```
public class C { ... }
```

- Wenn man nichts angibt, sind sie nur innerhalb des Paketes sichtbar.

  Das ist genau wie bei Attributen und Methoden innerhalb einer Klasse.
  Die Schlüsselworte "`private`" und "`protected`" können für nicht-geschachtelte Klassen ("top-level classes") nicht verwendet werden. Geschachtelte Klassen können in dieser Vorlesung leider nicht besprochen werden.

- Da eine "`public class`" so heißen muss wie die Quelldatei (natürlich ohne die Endung `.java`), kann pro Java-Quelldatei nur eine öffentliche Klasse definiert werden.

Zugriffsschutz für Klassen (2)

- `public`-Komponenten in einer Klasse, die nicht `public` ist, werden z.B. bei Unterklassen verwendet:
  - Außerhalb des Paketes ist nur die "`public`"-Oberklasse bekannt (Verwendung z.B. für Variablen-Deklarationen).
  - Die Variablen mit dem Typ der Oberklasse können aber Objekte von Unterklassen enthalten, die nicht `public` sind.

    Dies zeigt, warum der Zugriffsschutz von Methoden in Unterklassen nicht eingeschränkt werden kann: Sie sind so zugreifbar.

- Auch für Klassen gilt, dass man `public` nur verwenden sollte, wenn der Zugriff von außen nötig ist.

  Ein Paket hat oft relativ viele Hilfsklassen, die für die Benutzung der eigentlichen Funktion des Paketes von außen nicht sichtbar sein müssen. Man hat mehr Optionen für eine spätere Änderung der Paket-Implementierung, wenn man sicher sein kann, dass diese Klassen außerhalb nicht verwendet wurden.