Java I – Vorlesung 9
Generics und Packages

21.6.2004

Generics
Packages
Qualifizierte Namen
Mehr zu Zugriffsklassen

Generics (Java 1.5)

- Die Klassen im Java Collections Framework sind mit dem Typ ihrer Elemente parametrisiert:

  `List<Integer>` `List<String>` usw.

- Solche parametrisierbaren Klassen und Interfaces heißen Generics.
- Wir können unsere eigenen Generics programmieren, die dann von anderen instanziiert werden können.

`http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf`

Generics: Syntax

- Man definiert eine generische Klasse (oder Interface), indem man in spitzen Klammern Typvariablen angibt:

  ```
  class GenerischeKlasse<E, T> {...}
  ```

- Innerhalb der Klassendeklaration kann man dann die Typvariablen (fast) wie normale Referenztypen verwenden.
- Bei Verwendung von GenerischeKlasse<A,B> mit konkreten Referenztypen A und B wird dann überall A für E und B für T eingesetzt.

Beispiel: Generics

```
class MyArrayList<E> {
  private E[] array;
  private int nextPos;

  public MyArrayList() {
    array = new E[10]; nextPos = 0;
  }

  public void add(E x) {
    array[nextPos++] = x;
  }

  public E get(int index) {
    return array[index];
  }
}
```

Beispiel: Generics

```
MyArrayList<String> x =
  new MyArrayList<String>();

x.put("1234");
 // x.get(0) ist Ausdruck von Typ String


MyArrayList<Integer> y =
  new MyArrayList<Integer>();

y.put(new Integer(3));
// y.get(0) ist Ausdruck von Typ Integer
```


Bounded Types

- Compiler weiß nichts über die Referenztypen, die für Typ-Parameter eingesetzt werden.
- Deshalb sind für Ausdrücke von parametrischem Typ nur die Methoden von `Object` verfügbar.
- Wir können erzwingen, dass Typ-Parameter nur mit Typen instanziiert wird, die spezieller als ein bestimmter Referenztyp sind:

```
class A<E extends XY> { ... }
```

- Damit können wir Methoden von XY verwenden.

Beispiel: Bounded Types

```
class A {
  public void meth() { ... }
}

class SomeClass<E> {
  public void doSomething(E obj) {
      obj.meth();     // verboten
  }
}

A x = new A();
SomeClass<A> sca = new SomeClass<A>();

sca.doSomething(x);
```


Beispiel: Bounded Types

```
class A {
  public void meth() { ... }
}

class SomeClass<E extends A> {
  public void doSomething(E obj) {
      obj.meth();     // ok
  }
}

A x = new A();
SomeClass<A> sca = new SomeClass<A>();

sca.doSomething(x);
```

Beispiel: Bounded Types

```
class A {
  public void meth() { ... }
}

class SomeClass<E extends A> {
  public void doSomething(E obj) {
      obj.meth();     // ok
  }
}

// verboten:
SomeClass<Object> sca =
       new SomeClass<Object>();
```


Generics und Subtyping

- Subtyping auf den generischen Klassen selbst:
  - Wenn C spezieller als D ist, ist natürlich C<A> spezieller als D<A>.
- Kein Subtyping über die Typ-Parameter:
  - Auch wenn A speziellerer Typ ist als B, sind C<A> und C<B> unvergleichbare Typen.
- Warum?

Generics und Subtyping

```
class C<E> {
  public void put(E x) { ... }
  public E get() { ... }
}

class B extends A { ... }

C<B> cb = new C<B>();
C<A> ca = cb;          // automatischer Upcast
A a = new A();

// wollen wir folgendes dürfen?
ca.put(a);
B b = cb.get();
```


Teil-Lösung: Wildcards

- Es gibt einen Wildcard-Typen C<?>, der allgemeiner ist als jedes C<A>.
- Man kann damit aber nicht viel machen:
  - Von "innen" kann man nur annehmen, dass Typ-Parameter Object ist.
  - Von "außen" kann man keine Argument an eine Methode mit Parameter von Parametertyp geben.
  - Returnwerte von unbekanntem Parametertyp sind immerhin Object oder spezieller.

Generics und Subtyping

```
List<?> list = new ArrayList<String>(); // ok

// illegal: passt Parameter-Typ zu String?
list.add("hallo");

int s = list.size(); // ok

Object o = list.get(0); // ok
```


Bounded Wildcards

- Man kann Typ-Bounds auch bei Wildcards einsetzen:
  `List<? extends A>`
- Dieser Typ ist allgemeiner als List<A> und List<B>.
- Nicht allgemeiner als List<C>, wenn C nicht spezieller als A ist.
- Kann dann annehmen, dass Returntypen spezieller als A (statt Objekt) sind.

Generische Methoden

- Auch Methoden können Typ-Parameter bekommen:

```
public <E, T> int meth(E x, T y) {...}
```

- Typ-Parameter kann mehrfach auftreten: Abhängigkeiten zwischen Typen.
- Beim Aufruf werden Typ-Parameter aus den Typen der Argumente inferiert.
- Konkrete Typ-Parameter können nicht explizit angegeben werden.

Beispiel: Generische Methoden

```
static <T> void toCollection(T[] a,
                             Collection<T> c) {
  for( T element : a ) {
    c.add(element);
  }
}


String[] array = { "a", "b", "c" };
List<String> list = new ArrayList<String>();

toCollection(array, list);
```

Bounds bei generischen Methoden

- Auch bei generischen Methoden können Wildcards und Bounded Wildcards verwendet werden:
  - Parametertyp C<?>: Alle Typen C<A>
  - Typ C<? extends A>: Alle Typen C<B>, bei denen B spezieller als A ist
  - Typ C<? super A>: Alle Typen C<B>, bei denen B allgemeiner als A ist

Beispiel: Bounds bei Methoden

```
public static <T> T writeAll(
    Collection<T> coll, Sink<T> sink)  {
  T last = null;
  for( T t : coll ) {
    sink.write(t);    last = t;
  }
  return last;
}

Sink<Object> sink = ...;
Collection<String> coll = ...;

String str = writeAll(coll, sink);
// nein: Inkompatible Typen für T!
```

Beispiel: Bounds bei Methoden

```
public static <T> T writeAll(
    Collection<? extends T> coll, Sink<T> sink)  {
  T last = null;
  for( T t : coll ) {
    sink.write(t);    last = t;
  }
  return last;
}

Sink<Object> sink = ...;
Collection<String> coll = ...;

String str = writeAll(coll, sink);
// besser (String extends Object) --
// aber falscher Returntyp!
```


Beispiel: Bounds bei Methoden

```
public static <T> T writeAll(
    Collection<T> coll, Sink<? super T> sink)  {
  T last = null;
  for( T t : coll ) {
    sink.write(t);    last = t;
  }
  return last;
}

Sink<Object> sink = ...;
Collection<String> coll = ...;

String str = writeAll(coll, sink);
// jetzt ok!
```

Generics: Einschränkungen

- Typ-Parameter müssen Referenztypen sein, keine primitiven Typen.
  - Man kann aber mit List<Integer> u.ä. Listen von primitiven Typen simulieren.
  - Autoboxing und -unboxing: Compiler führt oft Konvertierung zwischen int und Integer (Boxing) und umgekehrt automatisch aus.
- Innerhalb der Klassendefinition dürfen Typ-Parameter nicht beliebig verwendet werden:
  - keine Instanziierung, z.B. new E()

Packages

- Man kann verschiedene Klassen zu einer Package zusammenfassen.
- Packages bilden größere logische Einheit.
- Aufteilung von Namensräumen.
- Klassen haben privilegierten Zugriff auf Felder und Methoden der Klassen in gleicher Package.
- Klassen der gleichen Package liegen typischerweise im gleichen Verzeichnis.

Deklaration von Packages

- Man ordnet eine Klasse (oder Interface) durch eine package-Anweisung einer Package zu:
  ```
  package name.mit.punkten;
  ```
- Der qualifizierte Name der Klasse ist dann
  ```
  name.mit.punkten.Klassenname
  ```
- Der Compiler legt die Bytecode-Datei ins Verzeichnis
  ```
  name/mit/punkten
  ```


Package ohne Namen

- Wenn die Datei keine package-Anweisung enthält, gehört sie zur "namenlosen Package".
- Voll qualifizierter Name der Klasse C in der namenlosen Package ist einfach C.
- Verhält sich ansonsten genau wie richtige Packages.

Zugriff auf Klassen in Packages

- Man kann immer den voll qualifizierten Namen einer Klasse verwenden.
- Auf Klassen aus der gleichen Package kann man mit Namen ohne Packagebezeichnung zugreifen.
- Mit einer import-Anweisung kann man Klassen aus anderen Packages importieren:

```
import andere.pkg.Klassenname;
import andere.pkg.*;
```


Packages und Verzeichnisse

- JVM sucht Klassen der Package a.b.c im Verzeichnis a/b/c. Dorthin werden sie auch kompiliert.
- Guter Stil: Auch Quelltexte entlang der Package-Struktur organisieren.
- Guter Stil: Verwendet Packages! D.h. außer in winzigen Programmen keine Klassen in namenloser Package.
- Guter Stil: Eindeutiges Packagenamen-Präfix (com.sun...; edu.mit...; de.saar...; usw.)

Die Zugriffsklasse "package"

- Wir kennen bisher zwei Zugriffsklassen:
  - public: Feld/Methode von überall zugreifbar
  - private: nur aus Klasse selbst zugreifbar
- Zugriffsklasse "package": von allen Klassen in der Package aus zugreifbar.
- "package" ist Standard-Zugriffsklasse: Wird verwendet, wenn keine besondere Zugriffsklasse angegeben ist.

Zugriffsklassen von Klassen/Interfaces

- Auch Klassen und Interfaces haben Zugriffsklassen, die bestimmen, von wo man sie verwenden kann.
- Standard-Zugriffsklasse ist "package", d.h. Klasse kann nur aus eigener Package verwendet werden.
- Muss ggf. Klassen/Interfaces als "public" deklarieren.
- Klasse mit main-Methode muss public sein.

Zusammenfassung

- Generics: Parametrisierte Referenztypen und Methoden
- Bounds, Wildcards
- Packages strukturieren größere Projekte