Java Annotationen und Reflection

Aufgabe 7 & 8

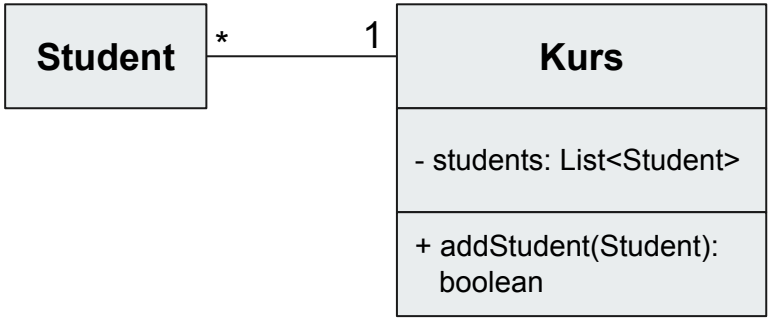
Gliederung

1. Aufgabe 7: Sichtbarkeit, Lebensdauer und Gültigkeit (Scope)
2. Aufgabe 8: Klassendiagramm
3. Annotationen
4. Reflection

A.7: Sichtbarkeit, Lebensdauer u. Gültigkeit

```
class Scope {
	int i ; // paketsichtbare Variable
	public static void main(String[] args) {
		int j = 12; // private Variable, Gültigkeitsbereich in der ganzen main
		{
			int x = 1234; // private Variable, gültig innerhalb der geschweiften Klammern
		}
		x = 0; // nicht deklarierte Variable -> Compilerfehler
	}
	// Java ist laut eigener Definition “pass by value”
	// d.h. ein Pointer wird als Parameter übergeben
	public void foo( int i) { // Parameter ist innerhalb der Methode sichtbar
		this.i = i; // globaler Variable i wird übergebene Variable i zugewiesen
	}
}
```

A.8: Klassendiagramm




```
class Student { /* … */ }

class Kurs {
     private final int maxStudents = 30;
     private List<Student> students = new
     ArrayList<Student>();
     /* … */
     public boolean addStudent(Student
student) {
          if(students.size() > maxStudents)
               return false;
          students.add(student);
          return true;
     }
     /* … */
}
```

Annotation

- besondere Semantik für Compiler (@Override, @SuppressWarnings)
- Definition von Web-Services (@WebService, @WebMethod,...)
- Definition von Komponenten mit XML-Abbildung (@XmlRootElement, @XmlElement, ...)

Compiler übersetzt Annotationstypen in Schnittstellen!

```
public @interface Annotationname {...}
```

Beispiel:

```
public @interface Programmkopf{

public String version(); … }
```

```
@Programmkopf(

version=”2.1”, … )
```

Vordefinierte Annotations

- @Override :
  - Methode überschreibt Methode der Basisklasse
- @Deprecated:
  - Klasse/Methode soll nicht mehr verwendet werden
- @SuppressWarnings
  - unterdrückt Compiler-Warnungen, z.B. @SuppressWarnings(“deprecated”)
- @Target
  - beschreibt eine andere Annotation
- @Retention
  - steuern das Verhalten des Compilers für eine Annotation

Annotation - Nachteile

- Annotationen sind stark mit Quellcode verbunden -> nur dort änderbar!
- Annotation sind invasiv und binden Implementierung an einen bestimmten Typ, wie es Schnittstellen tun -> Annotationstypen nicht im Klassenpfad -> Compilerfehler
- keine Vererbung von Annotationen möglich
- Werte lassen sich zur Laufzeit erfragen, aber nicht modifizieren
- unbekannte Schreibweise: @interface

Alternative: moderne Frameworks wie JPA oder JSF2 aus Java EE Standard sehen Einsatz von XML vor

Reflection Inspizieren des Programms zur Laufzeit

Bsp.: Aufruf der Methode *addStudent* (siehe Folie 4) mittels Reflection

```
try {
   Kurs kurs = new Kurs();
   Method method = kurs.getClass().getMethod("addStudent", Student.class);
   method.invoke(kurs, new Student()); // kurs.addStudent(new Student());
} catch (NoSuchMethodException e) {
   e.printStackTrace();
} catch (IllegalAccessException e) {
   e.printStackTrace();
} catch (InvocationTargetException e) {
   e.printStackTrace();
}
```

Reflection Inspizieren des Programms zur Laufzeit

Bsp.: Konstante *maxStudents* ändern (ohne Behandlung von Fehlern)

Funktioniert aufgrund von Optimierungen des Compilers teilweise nicht.

```
Kurs kurs = new Kurs();
Field maxStudents = kurs.getClass().getDeclaredField("maxStudents");
maxStudents.setAccessible(true); // IllegalAccessException verhindern
Field modifiers = Field.class.getDeclaredField("modifiers");
modifiers.setAccessible(true);
modifiers.setInt(maxStudents, maxStudents.getModifiers() & ~Modifier.FINAL);
maxStudents.set(kurs, 50);

System.out.println(maxStudents.get(kurs)); // 50
for (int i = 0; i <= 50; i++) if (kurs.addStudent(new Student())) System.out.println(i); // 30
```