Fakultät

Elektrotechnik und Informatik

SEMINARARBEIT

Die Java Reflection API und ihre Anwendungen

Verfasser: Michael Koch

Fach: Virtuelle Plattformen, Programmiersprachen und Compilerbau

Betreuender Dozent: Prof. Volkhard Pfeiffer

Abgabetermin: 19.05.2009

Inhaltsverzeichnis Seite

Abbildungsverzeichnis Seite

Anlagenverzeichnis Seite

0. Problemstellung

Diese wissenschaftliche Arbeit bezieht sich auf die Reflection API von Java und deren praktische Anwendungen.
Um das Thema dem Leser verständlich zu machen, wird zuerst erklärt, was man unter dem Begriff Reflection allgemein versteht und welche Architektur der API zugrunde liegt. Aufgrund dieser Grundlagen werden anschließend die wichtigsten Funktionen der Java Reflection API erläutert. Als Grundlage für die Codebeispiele dient dabei ein stark vereinfachtes Klassenmodell eines imaginären Fahrzeugherstellers. Innerhalb der Dokumentation wird aus Gründen der Übersichtlichkeit auf die Behandlung von Exceptions verzichtet, obwohl diese an manchen Stellen nötig wäre.
Nach der Beschreibung der API folgt eine Reihe von praktischen Anwendungen dieser API, wobei sich auch diese Beispiele auf das Modell eines Fahrzeugherstellers beziehen. Zum Abschluss dieser Seminararbeit werden die wesentlichen Vor- und Nachteile von Reflection gegenübergestellt und das komplette Thema kurz und prägnant zusammengefasst.

Diese Seminararbeit setzt ein fundamentales Verständnis von objektorientierter Programmierung, sowie grundlegende Kenntnisse der englischen Sprache voraus.
Alle Beispielprogramme basieren auf dem Java Runtime Environment (JRE) Version 6 von der Firma Sun Microsystems.

1. Allgemeines über Reflection

1.1. Definition laut Lahres und Raýman

„Reflexion (engl. Reflection) ist ein Vorgang, bei dem ein Programm auf Informationen zugreift, die nicht zu den Daten des Programms, sondern zur Struktur des Programms selbst gehören. Diese Informationen können dabei über eine definierte Schnittstelle ausgelesen werden.“[1]

[1] /Lahres06/ S.538

1.2. Definition laut Dr. Forman

„Reflection is the ability of a running program to examine itself and its software environment, and to change what it does depending on what it finds."[2]

1.3. Meta- und Base-Level-Objekte

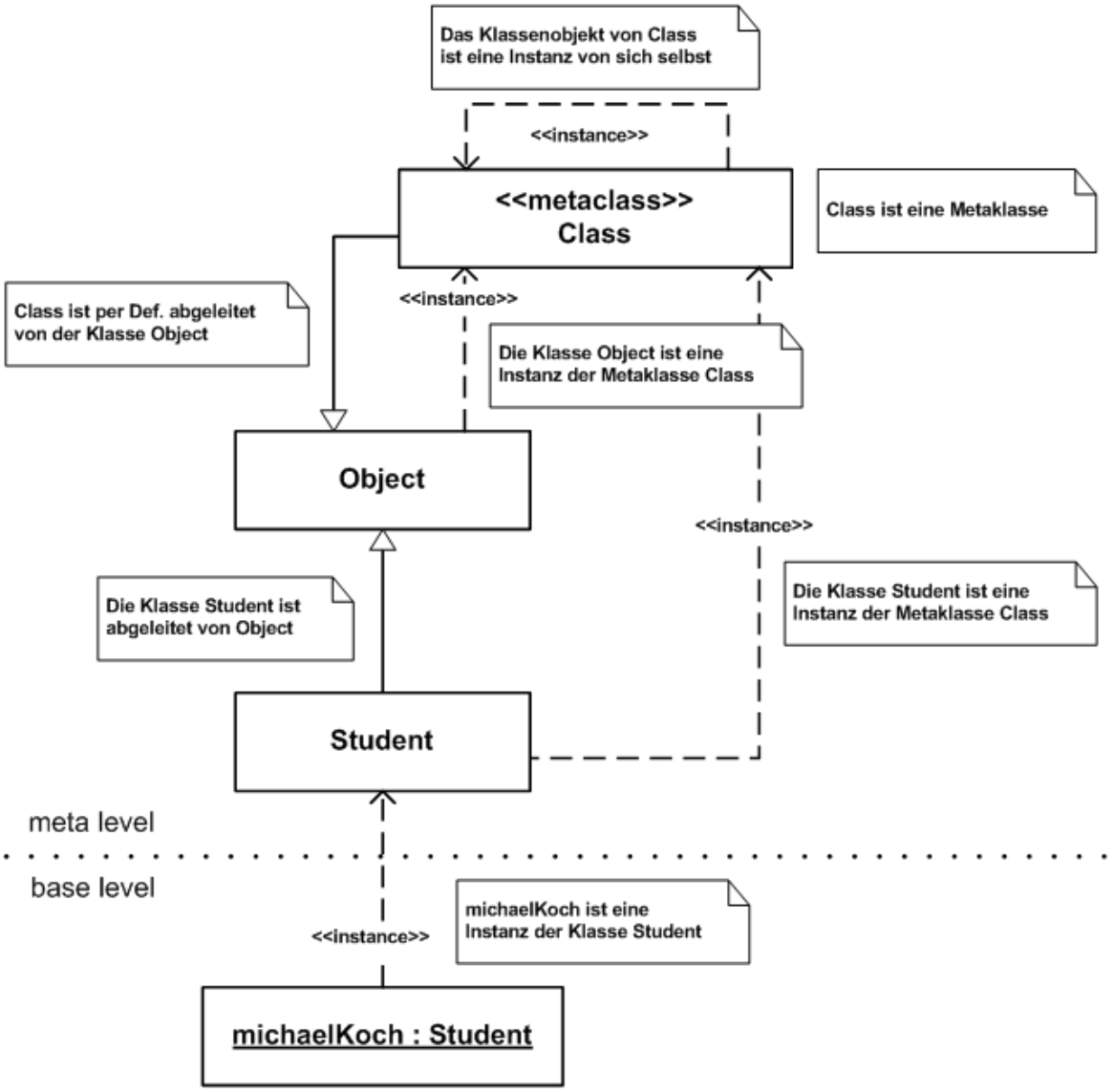


Abbildung 1: Meta- und Base-Level[3]

[2] /Forman05/ S. 3
[3] vgl. /Forman05/ S. 248 Figure A.4

Damit ein Programm seinen eigenen Aufbau bzw. den Aufbau von fremden Klassen bestimmen kann, ist es nötig, dass jede Klasse von sich aus Informationen über ihren eigenen Aufbau besitzt.[4] Diese Informationen werden allgemein als Metadaten bezeichnet.
Java und andere objektorientierte Programmiersprachen speichern diese Metadaten in Objekten, welche dann wiederum als Metaobjekte bezeichnet werden.
Eine Metaklasse erzeugt als Instanzen Klassen, wobei `Class` die einzige Metaklasse in Java ist.[5] Metaklassen und deren Instanzen werden als Meta-Level zusammengefasst. Die Instanzen einer Klasse werden als Base-Level zusammengefasst.

Interessant sind die Tatsachen, dass auch `Class` von `Object` abgeleitet ist und dass das Klassenobjekt von `Class` eine Instanz von sich selbst ist[6] (Kreisschluss).
Das lässt sich durch den folgenden Aufruf beweisen (Erklärung unter 2.1.4):

```
System.out.println(Class.class.isInstance(Class.class));  //true
```

1.4. Das Meta-Object-Protocol (MOP)[7]

Die Schnittstelle zu den Metaobjekten wird als Meta-Object-Protocol bezeichnet. Die Operationen, die sich auf das MOP beziehen lassen sich in zwei Bereiche kategorisieren:

- **Introspection**
  - Instanzvariablen auflisten
  - Methoden auflisten
  - Werte der Instanzvariablen ermitteln
- **Intercession**
  - Neue Instanzvariablen zu einer Klasse hinzufügen
  - Neue Methoden zu einer Klasse hinzufügen
  - Methoden neu definieren
  - Eingriffe in die Klassenhierarchie
  - Kontrollierung von Methodenaufrufen

Das MOP von Java fast ausschließlich auf introspektive Operationen beschränkt.

[4] /Forman05/ S.9
[5] vgl. /Forman05/ S. 249f
[6] vgl. /Forman05/ S. 25
[7] Informationen beziehen sich auf Kapitel A.4 aus /Forman05/ S.248f

2. Die Java Reflection API

In diesem Kapitel soll anhand des nachfolgenden Beispiels erklärt werden, welche Möglichkeiten Java bietet, um Metadaten zu ermitteln und wie diese genutzt werden können.

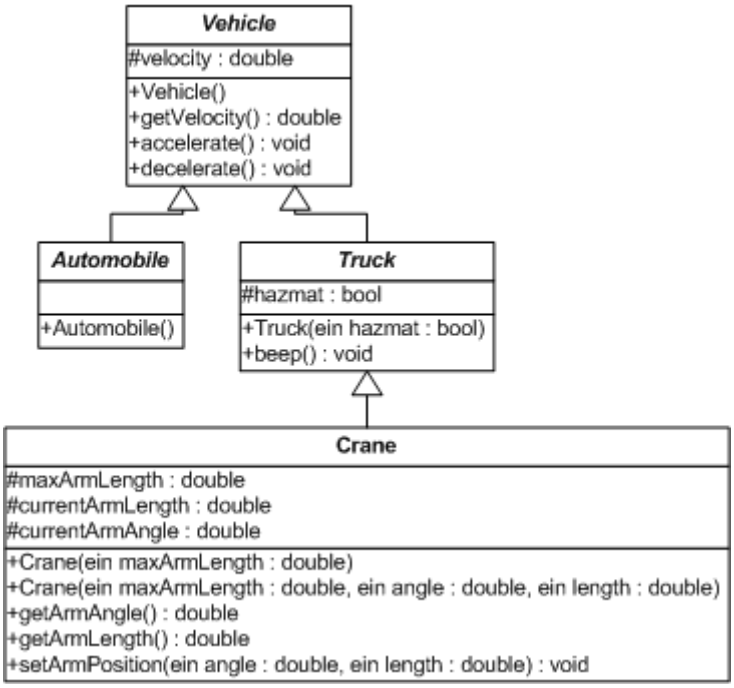


Abbildung 2: Klassendiagramm für Fahrzeuge

Um die in diesem Kapitel vorgestellten Typen nutzen zu können, muss das Package `java.lang.reflect.*` komplett importiert werden.

2.1. Klassen

2.1.1. Klassenobjekte beziehen

In Java wird eine Klasse als Instanz der Metaklasse `Class` betrachtet (siehe Abbildung 1). Diese Instanz wird auch als Klassenobjekt bezeichnet.
„Die Klasse `Class` stellt Methoden zur Abfrage von Eigenschaften der Klasse zur Verfügung und erlaubt es, Klassen dynamisch zu laden und Instanzen dynamisch zu erzeugen. Darüber hinaus ist sie der Schlüssel zur Funktionalität des Reflection-API.“[8]

[8] /Krüger08/ S. 1024

Klassenobjekte können allgemein auf drei Arten bezogen werden[9]:

- Direkt über die Klasse:

```
Class<?> currentClass = Crane.class;
```

- Über eine Instanz der Klasse:

```
Class<?> currentClass = myCrane.getClass();
```

- Über den Namen der Klasse:

```
Class<?> currentClass = Class.forName("Crane");
```

„Ist der Klassenname zur Compilezeit bekannt, kann anstelle des Aufrufs von `forName()` auch die abkürzende Schreibweise `.class` verwendet werden. Das hat den Vorteil, dass bereits der Compiler überprüfen kann, ob die genannte Klasse vorhanden ist.“[10]

Ein interessanter Aspekt von Java ist, dass auch primitive Datentypen (sogar `void`), die eigentlich nicht von `Object` abstammen trotzdem durch ein Klassenobjekt repräsentiert werden. Hierbei muss man allerdings beachten, dass das Klassenobjekt einer Wrappper-Klasse ungleich dem Klassenobjekt des entsprechenden primitiven Datentyps ist. Jedoch besitzt jede Wrapper-Klasse das Attribut `TYPE`, welches das Klassenobjekt des entsprechenden primitiven Datentyps repräsentiert.

Es gilt also: `double.class` = `Double.TYPE` ≠ `Double.class`

2.1.2. Klassenhierarchie ermitteln

In Java kann eine Klasse maximal von einer Oberklasse abgeleitet werden. Das Klassenobjekt der Oberklasse lässt sich über die Methode `getSuperclass()` beziehen.

```
Class<?> currentClass = Crane.class.getSuperclass();
```

Die Variable `currentClass` würde jetzt auf das Klassenobjekt der Klasse `Truck` verweisen. In manchen Fällen ist es nötig, dass man die komplette Klassenhierarchie bis zur Klasse `Object` durchlaufen muss. Ein Beispiel dafür wäre die Serialisierung von Objekten mit Hilfe von Reflection (siehe 3.2.2).

[9] siehe auch /SUN002/
[10] /Krüger08/ S. 1034

Das nachfolgende Beispiel zeigt, wie man rekursiv die Klassenhierarchie bis zur Klasse `Object` durchlaufen kann und gibt den jeweiligen Klassennamen auf der Konsole aus.

```
public static void printClassHierarchy(Class<?> c)
{
  if ((c == null) || (c == Object.class))
  {
    return;
  }
  else
  {
    printClassHierarchy(c.getSuperclass());
    System.out.println(c.getSimpleName());
  }
}
```

Welche Interfaces von einer Klasse implementiert werden lässt sich mit der Methode `getInterfaces()` ermitteln. Diese Methode liefert jedoch nur die Klassenobjekte der Intefaces zurück die in der Klasse selbst implementiert werden und ignoriert die Interfaces die bereits innerhalb der Oberklasse implementiert wurden.
Die nachfolgende Methode listet alle implementierten Interfaces auf.

```
public static void printInterfaces(Class<?> c)
{
  while(c != null)
  {
    for (Class<?> i : c.getInterfaces())
    {
      System.out.println(i.getName());
    }
    c = c.getSuperclass();
  }
}
```

2.1.3. Instanzen zur Laufzeit erzeugen

Gerade im Bezug auf PlugIns (siehe 3.1) ist es wichtig, dass es eine Möglichkeit geben muss während der Laufzeit des Hauptprogramms eine neue Instanz der PlugIn-Klasse erzeugen zu können.
Möchte man eine Instanz anhand des Default-Konstruktors erstellen, so kann man dies mit Hilfe der Methode `newInstance()` machen:

```
Radio r = (Radio) Class.forName("SuperHiFiRadio").newInstance();
// Radio und SuperHiFiRadio beziehen sich auf das Beispiel aus 3.1
```

Wenn man eine Instanz über einen parametrisierten Konstruktor erstellen möchte bzw. muss, so kann man dies folgendermaßen machen:

```
Truck tr = (Truck) Class.forName("Crane").getConstructor(
    double.class, double.class, double.class).newInstance(
    70          , 0           , 12.5);
```

Der Methode `getConstructor()` müssen die Datentypen entsprechend der Signatur des Konstruktors übergeben werden. Diese Methode liefert dann ein Objekt vom Typ `Constructor` zurück, von dem man dann über die Methode `newInstance()` eine Instanz der Klasse erzeugen kann. Zuletzt muss man der Methode `newInstance()` noch die entsprechenden Werte in der richtigen Reihenfolge übergeben.

Über die Methode `getConstructors()` erhält man ein Array, welches alle Konstruktoren der Klasse beinhaltet. Das nachfolgende Code-Fragment listet diese auf:

```
public static void printConstructors(Class<?> c)
{
  for (Constructor<?> constructor : c.getConstructors())
    System.out.println(constructor);
}
```

2.1.4. Type-Comparison und Type-Casting

Zwei Objekte sind immer dann vom gleichen Typ, wenn sie das gleiche Klassenobjekt besitzen.

```
System.out.println(myCrane.getClass() == Crane.class);    //true
System.out.println(myCrane.getClass() == Vehicle.class);  //false
```

Um festzustellen, ob eine Instanz einer Variablen von einem bestimmten Typ zugewiesen werden kann, gibt es zwei Möglichkeiten:

- Über den **`instanceof`** Operator (nur möglich, wenn der Typ, mit dem verglichen wird, zur Compilezeit bekannt ist)

  ```
  System.out.println(myCrane instanceof Vehicle);          //true
  ```

- Über die Methode `isInstance()`. Das ist immer möglich, aber nicht performant!

  ```
  System.out.println(Vehicle.class.isInstance(myCrane));    //true
  ```

Wenn man keine konkrete Instanz besitzt, so kann man über `isAssignable()` prüfen, ob man theoretisch z.B. eine Instanz der Klasse `Crane` einer Variablen vom Typ `Vehicle` zuweisen könnte.

```
System.out.println(Vehicle.class.isAssignableFrom(Crane.class));  //true
```

Ein Downcast lässt sich auf zwei Arten durchführen:

- Wenn die Typen zur Compilezeit bekannt sind.

  ```
  Vehicle v = (Vehicle) myCrane;
  ```

- Über die Methode `cast()`. Das ist immer möglich, aber nicht performant!

  ```
  Vehicle v = Vehicle.class.cast(myCrane);
  ```

2.2. Methoden

2.2.1. Welche Methoden besitzt eine Klasse?

In Java werden Methoden als Objekte vom Typ `Method` gekapselt. Welche öffentlichen Methoden eine Klasse besitzt kann mit Hilfe der Methode `getMethods()` ermittelt werden, welche ein Array mit den entsprechenden Method-Objekten zurückliefert. Dabei werden alle öffentlichen Methoden berücksichtigt, die entlang der Vererbungshierarchie irgendwann hinzugekommen sind.
Analog dazu gibt es noch die Methode `getDeclaredMethods()`, welche unabhängig von ihren Modifiern alle Method-Objekte zurückliefert, die unmittelbar in der untersuchten Klasse deklariert wurden.
Die nachfolgende Methode gibt alle Methoden mit ihrer vollen Signatur auf der Konsole aus. Dieses Vorgehen könnte man z.B. zur Dokumentation des Quellcodes verwenden.

```
public static void printMethods(Class<?> c)
{
  while(c != null)
  {
    for (Method m : c.getDeclaredMethods())
      System.out.println(m);
    c = c.getSuperclass();
  }
}
```

2.2.2. Die Signatur einer Methoden ermitteln

Die Signatur einer Methode setzt sich aus ihrem Namen, den Datentypen ihrer Parameter (abhängig von deren Reihenfolge) und ihrem Rückgabedatentyp zusammen.[11]

Angenommen, das Objekt `m` repräsentiert eine beliebige Methode, dann lassen sich die Signaturinformationen folgendermaßen ermitteln:

```
//Vollständiger Name der Methode (incl. Package)
String methodenName = m.getName();

//Namen der Methode ohne Packageangabe
String methodenKurzname = m.getSimpleName();

//Alle Parameterdatentypen in der richtigen Reihenfolge
Class<?>[] parameter = m.getParameterTypes();

//Der Rückgabedatentyp
Class<?> ergebnis = m.getReturnType();
```

2.2.3. Ein Method-Objekt anhand der Signatur ermitteln

Das Method-Objekt einer bestimmten öffentlichen Methode lässt sich wie im folgenden Beispiel direkt über ihren Namen ermitteln:

```
Method m = Crane.class.getMethod("getArmAngle");
```

Besitzt eine Methode Parameter, so müssen die Klassenobjekte der jeweiligen Parameter in der richtigen Reihenfolge mit angegeben werden. Bei primitiven Datentypen muss man jedoch beachten, dass deren Klassenobjekte ungleich den Klassenobjekten der entsprechenden Wrapper-Klasse ist (siehe 2.1.1).

```
Method m = Crane.class.getMethod("setArmPosition",
                                 double.class, double.class);
```

Um auch auf nicht öffentliche Method-Objekte zugreifen zu können, müsste man dieses aus dem Array von `getDeclaredMethods()` herausfiltern. Unter 2.3.2 ist ein Beispiel abgebildet, wie man dies für Attribute (bzw. Felder) machen kann. Bei Methoden müsste man die komplette Signatur auf Identität prüfen.

[11] vgl. /Krüger08/ S.170

2.2.4. Methoden zur Laufzeit dynamisch aufrufen

Die Methoden, die durch ein Method-Objekt repräsentiert werden lassen sich auch zur Laufzeit dynamisch aufrufen. Um dies durchzuführen besitzt jedes Method-Objekt eine Methode namens `invoke()`, der man als ersten Parameter die Instanz übergeben muss auf die sich die Operation beziehen soll. Handelt es sich um eine Klassenmethode, so muss man hier `null` übergeben. Besitzt die Methode Parameter, so müssen anschließend noch die Parameterwerte übergeben werden. `Invoke()` liefert den Rückgabewert der Methode pauschal als `Object` zurück. Bei primitiven Rückgabedatentypen findet ein Autoboxing statt.

Das Resultat der beiden nachfolgenden Befehle ist absolut gleichwertig:

```
myCrane.setArmPosition(30.5, 12.5);
```

```
Crane.class.getMethod("setArmPosition", double.class, double.class)
             .invoke(myCrane         , 30.5        , 12.5);
```

Von der Laufzeit her betrachtet, ist jedoch der Aufruf über `invoke()` viel langsamer. „Im experimentellen Vergleich ergaben sich (...) Unterschiede um den Faktor 10 bis 100.“[12]

Über Reflection lassen sich auch nicht öffentliche Methoden ausführen, sofern zuvor die Methode `setAccessible(true)` des Method-Objekts aufgerufen wurde. Da dies jedoch das Prinzip der Abstraktion außer Kraft setzt, kann es zu fatalen, unvorhergesehenen Fehlern kommen (siehe 4).

2.3. Attribute (Fields)

2.3.1. Welche Attribute besitzt eine Klasse?

Ähnlich, wie Methoden, werden auch Attribute als Objekt gekapselt. Um diese zu ermitteln, existieren die Methoden `getFields()`, welche alle öffentlichen Attribute auflistet und die Methode `getDeclaredFields()`, welche auch nicht öffentliche Attribute auflistet. Ähnlich, wie bei Method-Objekten beschränkt sich `getDeclaredFields()` nur auf die in der Klasse selbst deklarierten Attribute. Ein konkretes öffentliches Attribut lässt sich anhand seines Namens über `getField("Attributsname")` ermitteln.

[12] /Krüger08/ S.1055

2.3.2. Attributwerte ermitteln und dynamisch ändern

Die meisten Attribute sind aus Gründen der Datenkapselung als nicht öffentlich deklariert. Da man deshalb meistens nicht über `getField("Attributsname")` auf diese zugreifen kann, muss man sich eines „Tricks“ behelfen:

```
public static Field getFieldByName(Class<?> c, String name)
{
  Field field = null;
  for (Field f : c.getDeclaredFields())
  {
    if(f.getName().equals(name))
    {
      field = f;
      field.setAccessible(true); //Auch auf private Felder zugreifen
    }
  }
  return field;
}
```

Um den Wert des Feldes auszulesen, existieren die Methoden `getInt()`, `getDouble()`, …, `getObject()`, denen man bei Instanzvariablen die entsprechende Instanz übergeben muss. Bei Klassenvariablen muss man **`null`** übergeben.
Um den Wert des Feldes zu ändern, existieren analog dazu die Methoden `setInt()`, `setDouble()`, …, `setObject()`, denen man ebenfalls die Instanz bzw. **`null`** übergeben muss. Als zweiten Parameter muss man dann den konkreten Wert übergeben.

```
Crane myCrane = new Crane(70.0);    //max. Armlänge 70.0m
myCrane.setArmPosition(30.5, 40.5); //aktuelle Armlänge 40.5m

Field maxLength = getFieldByName(Crane.class, "maxArmLength");

System.out.println(maxLength.getDouble(myCrane)); //max. 70.0m
maxLength.setDouble(myCrane, 10.0);
System.out.println(maxLength.getDouble(myCrane)); //max. 10.0m
```

In diesem Beispiel tritt z.B. der unvorhergesehene Zustand auf, dass der Arm des Krans weiter ausgefahren ist (40.5m), als er nach dem Eingriff sein dürfte (10.0m).
Bei sicherheitskritischen Anwendungen könnte ein derartiger Eingriff natürlich fatale Folgen nach sich ziehen!

Ein direkter Zugriff auf die Attribute sollte daher nur in Ausnahmefällen (z.B. Serialisierung mit Hilfe von Reflection 3.2.2) getätigt werden.

2.4. Modifier und Annotations

Modifier und Annotationen können für Class-, Method-, Field- und Constructor-Objekte festgelegt werden.

2.4.1. Modifier

Modifier werden als ein Integer (`int`) Wert repräsentiert, welcher über `getModifiers()` ermittelt werden kann. Diesen Wert kann man an eine der nachfolgenden statischen Methoden der Klasse `Modifier` übergeben, welche dann `true` oder `false` zurückliefern.

Aufgrund der Trivialität der Methodennamen erfolgt hier keine nähere Erläuterung der einzelnen Methoden. Die wichtigsten Methoden zum Prüfen von Modifiern sind:

- isAbstract
- isFinal
- isInterface
- isNative
- isPrivate
- isProtected
- isPublic
- isStatic

```
int mod = Vehicle.class.getModifiers();

System.out.println(Modifier.isAbstract(mod)); //true
System.out.println(Modifier.isStatic(mod));   //false
System.out.println(Modifier.isNative(mod));   //false
```

2.4.2. Annotations

Annotationen stellen dem Programm zusätzliche Metainformationen zur Verfügung, ohne die Arbeitsweise des Programms zu beeinflussen.[13] Sie sind ähnlich, wie ein Interface definiert und werden durch ein Klassenobjekt repräsentiert.

Annotationen eignen sich besonders zur Dokumentation des Quellcodes (siehe 3.3).
Eine häufig verwendete Annotation ist `@Deprecated`, welche anzeigt, dass eine Methode nicht weiter verwendet werden soll. Die nachfolgende Methode prüft, ob diese Annotation bei einer Methode gesetzt wurde oder nicht:

```
public static boolean isMethodDeprecated(Method m)
{
  return m.getAnnotation(Deprecated.class) != null;
}
```

[13] vgl. /Krüger08/ S.1047

3. Praktische Anwendungen von Reflection

3.1. PlugIns

Ein wesentliches Problem von „statisch“ programmierten Programmen ist, dass alle Klassen bereits zur Compilezeit vorhanden sein müssen. Ist das Programm kompiliert, so lassen sich Änderungen und Erweiterungen nur realisieren, indem man den Quellcode des bestehenden Programms abändert und es komplett neu kompiliert. Dies kann unter Umständen sehr aufwändig werden, wenn das Programm weit verbreitet ist.
Ebenso kann es sein, dass man fremden Softwareentwicklern eine Möglichkeit anbieten möchte, das eigene Programm an bestimmten, klar definierten Stellen erweitern zu können, ohne das diesen Entwicklern der komplette Quellcode der eigenen Applikation zur Verfügung gestellt werden muss.
Das nachfolgende Beispiel beschreibt, wie ein Autohersteller diversen Radioherstellern eine PlugIn-Schnittstelle zur Verfügung stellen kann:

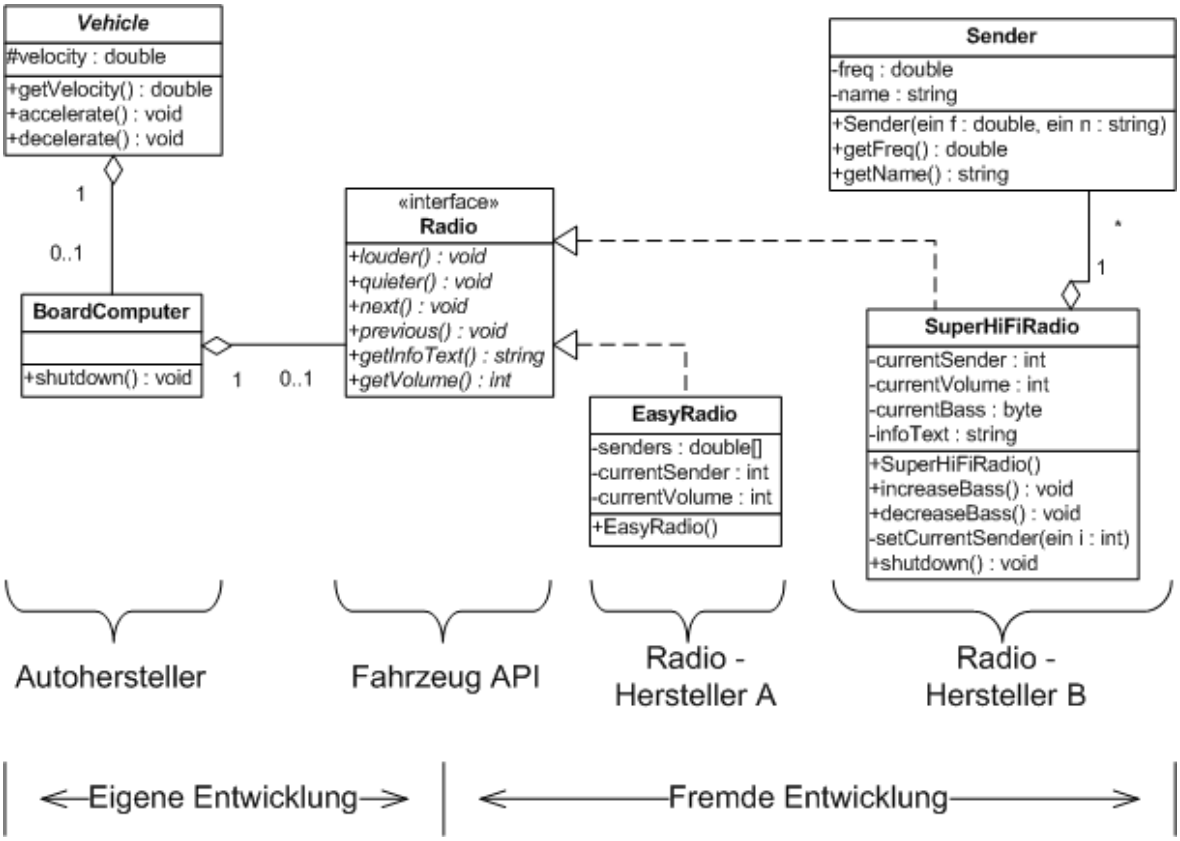


Abbildung 3: Klassendiagramm für eine Autoradio Schnittstelle

Jedes Fahrzeug (Vehicle) besitzt entweder keinen oder genau einen Boardcomputer. Dieser besitzt wiederum kein oder genau ein Radio.
Autoradios werden von verschiedenen Herstellern angeboten, denen der Quellcode des Fahrzeugherstellers verborgen bleibt. Sie kennen nur das Interface `Radio`.
Ein Fahrzeughersteller kennt hingegen zur Compilezeit keine konkreten Radioklassen, da diese evtl. auch erst viel später entwickelt werden, als die Software des Fahrzeugherstellers.
Deshalb muss der Boardcomputer mit Hilfe von Reflection während der Laufzeit eine Instanz des konkreten Radios anhand seines Klassennamens erstellen (siehe 2.1.3). In diesem Beispiel erhält er den Klassennamen aus der Datei bc.ini, die ein Techniker nach dem Einbau des Radios einmalig konfigurieren müsste.
Ist diese Instanz erstellt, so muss ein Cast auf den Typ `Radio` durchgeführt werden, da der Fahrzeughersteller nur von dessen Methoden die Semantik kennt.
Das Fahrzeug kann nun über alle Methoden, die durch die Schnittstelle Radio definiert sind, mit der dynamisch erzeugten Radio-Instanz kommunizieren.
So könnten über den Boardcomputer die Methoden `louder()`, `quieter()`, `next()` und `previous()` an eine Taste am Lenkrad gebunden werden. Die Werte aus `getInfoText()` und `getVolume()` könnten auf dem Bildschirm des Boardcomputers visualisiert werden.
Der Boardcomputer stellt außerdem eine Methode namens `shutdown()` zur Verfügung. Diese versucht bei jedem Gerät, welches an den Boardcomputer angeschlossen ist, ein Method-Objekt, anhand eines in der Datei bc.ini angegebenen Methodennamens zu ermitteln (siehe 2.2.3).
Wenn, wie z.B. bei der Klasse `SuperHiFiRadio` ein derartiges Method-Objekt gefunden wurde, so wird die Methode nun über `invoke()` aufgerufen (siehe 2.2.4).

Das hier beschriebene Beispiel befindet sich auf der CD im Package „beispielPlugIn“.

3.2. Serialisierung

„Unter Serialisierung verstehen wir die Fähigkeit, ein Objekt, das im Hauptspeicher der Anwendung existiert, in ein Format zu konvertieren, das es erlaubt, das Objekt in eine Datei zu schreiben oder über eine Netzwerkverbindung zu transportieren.“[14]

[14] /Krüger08/ S. 963

3.2.1. Realisierung mit Java

Standardmäßig ist in Java das Interface `java.io.Serializable` vorgesehen. Implementiert eine Klasse dieses Interface, so können Instanzen von ihr serialisiert werden.
Leider wird dieses Interface nicht von allen Klassen implementiert, weil diese Funktionalität von den Entwicklern der Klasse entweder bewusst nicht vorgesehen war, oder weil nicht alle Aspekte zur Nutzung der Klasse bei der Planung berücksichtigt wurden.

3.2.2. Realisierung mit Reflection

Die Java Reflection API bietet die Möglichkeit, auch Instanzen von Klassen zu serialisieren, die nicht die Schnittstelle `Serializable` implementieren.

Dazu muss man die Werte aller Attribute (auch **`private`** und **`protected`**) auslesen können, die innerhalb der Klassenhierarchie deklariert wurden (siehe 2.1.2 und 2.3.2). Bei primitiven Datentypen und auch bei Strings stellt das in der Regel kein Problem dar. Allerdings müssen auch rekursiv alle Objekte serialisiert werden, auf die direkt oder indirekt über ein anderes Objekt verwiesen wird.

Dabei kann es aber zu einem Kreisschluss kommen, der in einer nicht terminierenden Rekursion endet, falls zwei Objekte direkt oder indirekt über ein anderes Objekt aufeinander verweisen.

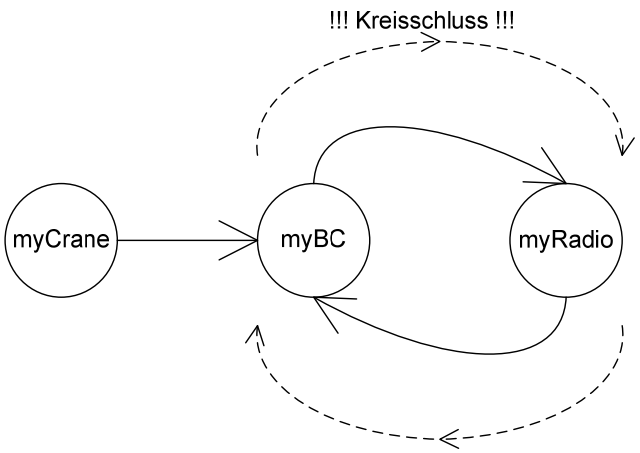


Abbildung 4: Kreisschluss bei der Serialisierung von Objekten mit Hilfe von Reflection

Angenommen, die Fahrzeug API aus dem Kapitel 3.1 (siehe Abbildung 3) bietet Radios die Möglichkeit, auf den Boardcomputer zuzugreifen, an dem sie angeschlossen sind, dann könnte ein Radio z.B. eine Diebstahlsicherung implementieren, indem es nur funktioniert, wenn es mit einem bestimmten Boardcomputer betrieben wird.
Neben dem standardmäßigen Verweis des Boardcomputers auf das Radio, käme also noch ein Verweis vom Radio auf den Boardcomputer hinzu.
Bei der Serialisierung des Boardcomputers würde auch das Radio serialisiert werden. Da im Radio aber ein Verweis auf den Boardcomputer vorkommt, würde auch dieser wieder serialisiert werden.
Um das Problem zu lösen, müsste man sich also nebenbei merken, welche Objekte bereits serialisiert wurden.

3.3. Dokumentation des Quellcodes

Durch Annotationen (siehe 2.4.2) ist es möglich, dem Programm während der Entwicklung weitere Metainformationen zur Verfügung zu stellen, die später zur Laufzeit mit Hilfe von Reflection ausgelesen werden können.
Als Beispiel könnte man Methoden während der Entwicklung mit weiteren Informationen über deren letzte Änderung mitgeben, indem man eine Annotation erstellt, welche die Personalnummer, das letzte Änderungsdatum und eine Beschreibung der Änderungen zur Verfügung stellt.
Jedes Mal, wenn ein Entwickler etwas im Quellcode ändert, könnte er seine Veränderungen über die Annotation vermerken.
Mit einem Dokumentationstool könnte man später alle Methoden und die dazu gehörenden Informationen über ihre Veränderungen ermitteln und z.B. einen HTML-Bericht erstellen lassen.

Das hier beschriebene Beispiel befindet sich auf der CD im Package „beispielAnnotation“.

3.4. Weitere Anwendungen

Reflection wird v.a. auch auf folgenden Gebieten angewandt, welche aber hier nicht weiter erklärt werden:

- Java Beans
- Debug-Tools
- Class-To-Class-Transformation

4. Wesentliche Vor- und Nachteile von Reflection[15]

Reflection ermöglicht dynamische Programmerweiterungen

Bezogen auf PlugIns (siehe 3.1) stehen zusätzliche Klassen erst nach der Veröffentlichung des Hauptprogramms zur Verfügung, welche zur Laufzeit dynamisch geladen werden müssen. Das wäre ohne Reflection nicht möglich.

Probleme durch Verletzung der Datenkapselung

Wenn man mit Hilfe der Reflection API auf nicht öffentliche Methoden oder Attribute zugreift, kann es dazu kommen, dass das Programm in einen undefinierten Zustand gerät und nicht mehr richtig arbeitet.

Reflection hebt Abstraktion auf

Evtl. kann es beim direkten Zugriff auf interne Komponenten dazu kommen, dass das Programm bei einem Update der untersuchten Klasse nicht mehr funktioniert, weil z.B. ein internes Attribut herausgenommen wurde, oder eine interne Methode durch mehrere andere Methoden ersetzt wurde.

Probleme können sehr allgemeingültig gelöst werden

Da man dynamisch alle Attributwerte einer Instanz ermitteln kann, könnte man z.B. einmalig eine universell einsetzbare hashCode Methode schreiben und diese in jeder Klasse verwenden.

Hohe Laufzeit → Geringere Performance

Da bei Reflection auf bestimmte Datentypen erst zur Laufzeit dynamisch zugegriffen werden kann, können diverse Optimierungen der Java VM nicht vollzogen werden. Deshalb ist hier Ausführungszeit von Operationen signifikant größer, als bei einer Implementierung, die auf Reflection verzichtet.

Dokumentation und Testmöglichkeiten

Mit Reflection können Klassenbrowser und Testtools entwickelt werden.

Einschränkungen durch Sicherheitsrichtlinien

Die Methoden der Reflection API benötigen zur Laufzeit bestimmte Rechte, welche z.B. bei Applets nicht vorhanden sind.

[15] siehe auch /SUN001/

5. Zusammenfassung

Reflection ist eine Technik, mit der ein Programm zur Laufzeit Informationen über seinen eigenen Aufbau abrufen kann. Um die Programmstruktur zu beschreiben, werden Metadaten benötigt, welche in Form von Metaobjekten bereitgestellt werden.
In Java sind alle Klassen Instanzen von der einzigen Metaklasse `Class`. Instanzen von Klassen werden als Base-Level-Objekte bezeichnet, während die zuvor genannten Elemente unter dem Begriff Meta-Level-Objekte zusammengefasst werden.
Über Reflection lassen sich Klassenobjekte anhand ihres Namens beziehen und es ist möglich, zur Laufzeit Instanzen von ihnen zu erstellen.
Von einem Klassenobjekt lassen sich zur Laufzeit Methoden, Konstruktoren, Felder und Annotationen, sowohl gezielt über ihren Namen bzw. ihre Signatur, als auch als komplette Liste ermitteln. Auch diese Elemente werden als Objekte repräsentiert und stellen Informationen über ihre Deklaration (z.B. Modifier) zur Verfügung.
Method-Objekte können zur Laufzeit dynamisch ausgeführt werden, jedoch dauert dieser Vorgang viel länger als ein statischer Aufruf und sollte daher möglichst vermieden werden.
Mit Reflection ist es möglich, auch auf interne Attribute (schreibend) zuzugreifen, was jedoch das Prinzip der Datenkapselung und Abstraktion verletzt und was zu undefinierten Zuständen und den Verlust der Portabilität des Programms führen kann.
Jedoch ist die Reflection API gerade in Bezug auf dynamische Erweiterungen des Programms (PlugIns) unverzichtbar. Auch für Dokumentations- und Debug-Tools sind ihre Fähigkeiten von essentieller Bedeutung.
Letztendlich sollte man aber mit Reflection so sparsam wie möglich umgehen, da reflexiv programmierte Lösungen deutlich langsamer und unsicherer sind als die äquivalenten statischen Implementierungen, da der Compiler hier keine Möglichkeit hat, den Code hinsichtlich seiner Geschwindigkeit und Typsicherheit zu optimieren.

Glossar

Begriff	Definition
API	API ist die Abkürzung für **A**pplication **P**rogramming **I**nterface. Darunter versteht man eine Sammlung von Interfaces und Klassendefinitionen, die ein Softwareanbieter anderen Entwicklern zur Verfügung stellt, damit deren Programme mit dem Programm des Softwareanbieters kommunizieren können.
Applet	„Applets sind ebenfalls lauffähige Java-Programme. Anders als bei Applikationen, werden sie aus aus einer HTML-Seite heraus aufgerufen.“[16] Für sie gelten einige Einschränkungen hinsichtlich der Sicherheit.
Beans	„Als Beans werden in Java eigenständige, wiederverwendbare Softwarekomponenten zum Aufbau von Applets und Applikationen bezeichnet.“[17]
Primitiver Datentyp	Primitive Datentypen sind keine Objekte. Java kennt insgesamt 8 primitive Datentypen: boolean, byte, short, int, long, float, double, char.
Wrapper-Klasse	Für jeden primitiven Datentyp existiert eine Klasse, welchen den primitiven Datentyp als Objekt kapselt.

[16] /Krüger08/ S. 302
[17] /Krüger08/ S.1057

Literaturverzeichnis

/Forman05/ Forman, I.; Forman, N.: JAVA Reflection In Action, Manning Publications Co., Greenwich (USA), 2005

/Konrad03/ Konrad, R.; Kothe, A.: TU Darmstadt, http://www.aop.informatik.tu-darmstadt.de/pages/lectures/aood/ws03-04/download/reflection.pdf, Stand: 12.05.2009

/Krüger08/ Krüger, G.; Stark, T.: Handbuch der Java Programmierung, 5. Auflage, Addison-Wesley Verlag, München, 2008

/Lahres06/ Lahres, B.; Raýman, G.: Praxisbuch Objektorientierung, 1. Auflage, Galileo Press, Bonn, 2006

/Lochbihler08/ Lochbihler, A.: Universität Karlsruhe, http://pp.info.uni-karlsruhe.de/lehre/SS2008/foo/reflection.pdf, Stand: 26.03.2009

/ Middendorf02/ Middendorf, S.; Singer, R.; Heid J.: Java Programmierhandbuch und Referenz, 3. Auflage, Dpunkt Verlag, Heidelberg, 2002

/SUN001/ Ohne Verfasser; Sun Microsystems Inc., http://java.sun.com/docs/books/tutorial/reflect/index.html, Stand: 12.05.2009

/SUN002/ Ohne Verfasser; Sun Microsystems Inc., http://java.sun.com/docs/books/tutorial/reflect/class/classNew.html, Stand: 12.05.2009

/SUN003/ Ohne Verfasser; Sun Microsystems Inc., http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html, Stand: 13.05.2009

/SUN004/ McCluskey, G.; Sun Microsystems Inc., http://java.sun.com/developer/technicalArticles/ALT/Reflection/, Stand: 16.05.2009

/Ullenboom09/ Ullenboom, C.: Java ist auch eine Insel, 8. Auflage, Galileo Press, Bonn, 2009

Anlagen

Anlage 1: Liste der Beispielprogramme

Um die Anwendungen der Reflection API zu verdeutlichen, existieren auf der CD die nachfolgenden Beispielprogramme. Jedes dieser Programme befindet sich in einem eigenen Package, welches den Namen des Beispiels trägt.
Die jeweilige Main-Methode befindet sich jeweils in der Datei Program.java.

beispielMetaInfos

Dieses Programm besitzt eine Klasse namens `ClassInfo`, welche alle statischen Methoden aus Kapitel 2 beinhaltet. Diese werden in der Main-Methode mit konkreten Objekten aufgerufen, sodass man deren Funktion auf der Konsole nachvollziehen kann.

beispielPlugIn

Dieses Programm verdeutlicht das unter 3.1 beschriebene Beispiel. Der Boardcomputer eines Fahrzeugs lädt aus einer Konfigurationsdatei (bc.ini) den Klassennamen des Radios und erzeugt zur Laufzeit eine Instanz davon.
Um das Konzept dieser PlugIn-Schnittstelle zu verstehen, muss man den Wert „Radio“ in der Datei bc.ini abwechselnd auf einen der beiden Werte ändern und das Programm danach neu ausführen:

- radioherstellerA.EasyRadio
- radioherstellerB.SuperHiFiRadio

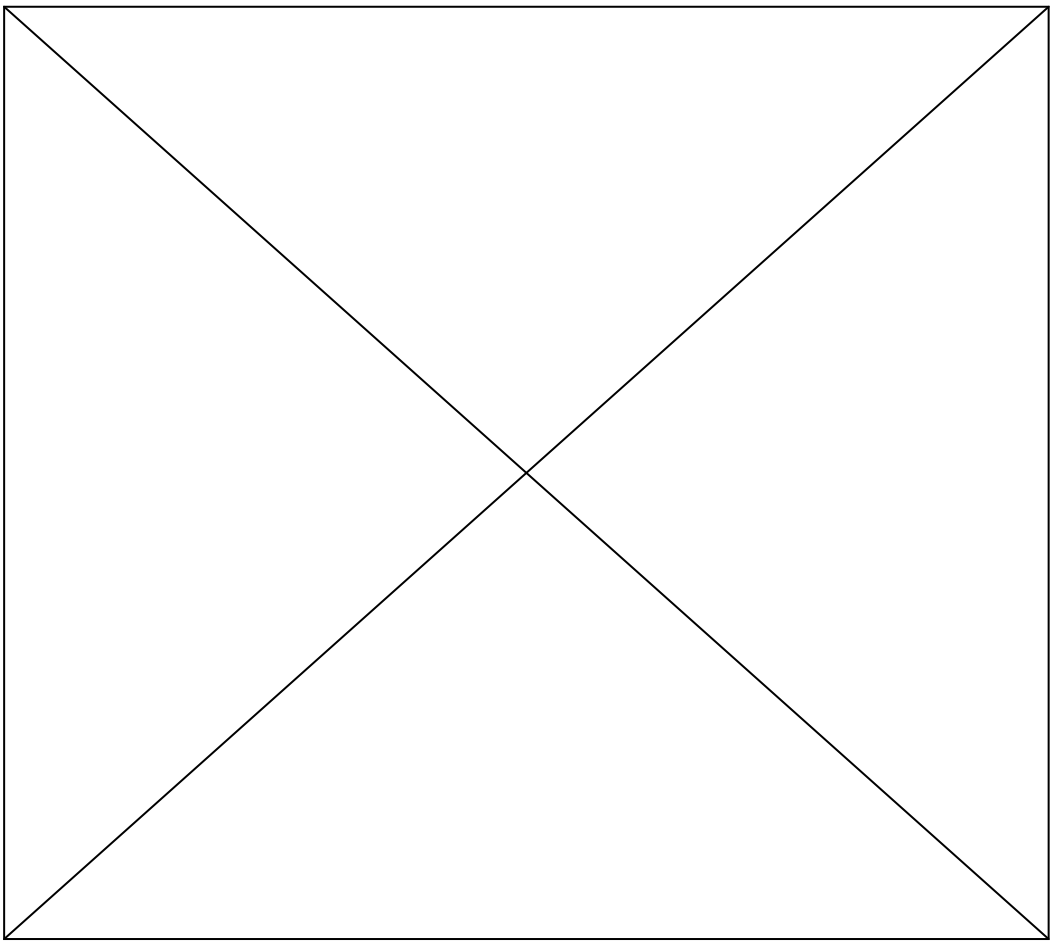
beispielAnnotation

Dieses Programm zeigt, wie man eine eigene Annotation definieren kann (das ist kein Thema dieser Arbeit) und wie man deren Informationen mit Reflection zur Laufzeit auslesen kann. In diesem Beispiel werden Methoden einer Klasse mit Informationen über deren letzte Änderung versehen, welche anschließend ausgelesen werden können.

beispielHashCode

Dieses Programm implementiert die im Kapitel 4 erwähnte hashCode Methode. Es handelt sich praktisch um ein Negativbeispiel für die Verwendung von Reflection.

Anlage 2: CD



Ehrenwörtliche Erklärung

Ich versichere hiermit, dass ich meine Seminararbeit mit dem Thema

Die Java Reflection API und ihre Anwendungen

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ort

Datum

Unterschrift