Die Java Reflections-API

von Tiziano DeGaetano und Michael Keller

Datum: 27.5.2004

Kurs: TIT02AGR

4. Semester

Vorlesung: Seminar Informaionstechnik

Inhaltsverzeichnis

0. Einleitung

Wenn man eine Entwicklungsumgebung in einer Programmiersprache schreibt, die für dieselbe Programmiersprache genutzt werden soll, so ist es manchmal hilfreich, „hinter die Kulissen“ zu schauen. Der Vorgang des Hineinschauens nennt sich Inspektion (engl. introspection). D.h. Informationen über die Komponenten selbst und nicht über den Verwendungszweck zu erhalten, ist das Ziel der Inspektion. Ab der Version 1.1 der Java Kern-Klassen wird eine neue API angeboten, die Reflections-API, welche das Fundament der Inspektion von Java-Klassen bildet.

Im Gegensatz zur statischen Inspektion hat die Reflections-API den Vorteil, daß Klassen und Komponenten während der Laufzeit untersucht werden können.

Eine Stärke in der Entwicklung von Java ist die Annahme, daß es in einer Umgebung laufen würde, die sich dynmisch ändert. Klassen werden dynamisch geladen, die Bindung wird dynamisch gemacht und Objekte werden zur Laufzeit dynamisch erstellt, wenn sie gebraucht werden. In der Vergangenheit wurde jedoch das Manipulieren von anonymen Klassen nicht sehr dynamisch gemacht. Eine anonyme Klasse ist in diesem Zusammenhang ist eine Klasse, die in einem Java Programm geladen oder dargestellt wird, dessen Typ dem Java Programm allerdings nicht bekannt ist.

Sollen anonyme Klassen implementiert werden, so heißt das, daß Java Objekte in den weiteren Verlauf eines laufenden Programmes eingebunden werden. Beispiele hierfür sind z.B. der Kommando-Zeilen-Interpreter von Java oder Java Applets, die in einen Web-Browser geladen werden. Die Applets werden von der Java Virtual Machine im Kontext des Web-Browsers geladen und ausgeführt. Das Java Programm hat allerdings im Voraus nicht die notwendigen Informationen, um die Klasse auszuführen.

Das Problem konnte bisher so gelöst werden, daß im Voraus ein „Vertrag“ eingegangen wurde. Jedes Objekt im Laufzeitsystem konnte so verwendet werden, wenn es die Richtlinien des „Vertrages“ einhielt, d.h. die Basisklasse oder Schnitstelle benutzte. Eine Basisklasse wie java.applet.Applet oder Schnittstellen wie AppletContext stellen solch einen „Vertrag“ dar.

Der Nachteil dieses Lösungsansatzes, vor allem bei häufigen Aufrufen, ist daß die Objekte des Laufzeitsystems nicht wiederverwendet werden können, solange sie nicht den ganzen „Vertrag“ annehmen. Wenn z.B. das Hosting-System die Schnittstelle AppletContext implementiert, können damit nur Applets geladen werden. Möchte man allerdings Hashtables anstelle von Applets laden, funktioniert das System nicht mehr.

Im Beispiel des Kommando-Zeilen-Interpreter stellt sich die Frage, wo die Abarbeitung einer dynamisch geladenen Klasse beginnen soll. Bisher wurde vereinbart, daß eine statische (also unikate) Funktion namens main aufgerufen wird, die nur Arrays von Strings als Übergabeparameter annimmt.

Die dynmische Inspektion bietet dagegen z.B. das Laden von graphischen Elementen zur Laufzeit oder das Laden von Geräte-Treibern in einem Java-basiertem Betriebssystem! Den Ausschlag zur Entwicklung der Reflection-API gab jedoch die Modellierung des Java „object component models“, welches jetzt als JavaBeans bekannt ist.

Die Verwendung einer graphischen Oberfläche schildert sehr gut, wieso die dynamische Inspektion so wichtig ist: einerseits bilden die Objekte eine Benutzeroberfläche in einer Anwendung, andererseits muß es eine Möglichkeit zum Manipulieren der Objekte geben, ohne die Objekte selbst zu kennen bzw. ohne Zugriff auf deren Quellcode zu haben.

Aus den Notwendigkeiten des JavaBeans Benutzer-Interfaces ist die Reflection-API entstanden.

1. Was ist Reflection?

Die Reflection-API besteht aus zwei Teilen: den Objekten, welche die verschiedenen Teile ein Klassen-Datei bilden, und dem Mittel, mit dem die Informationen aus den Objekten sicher ausgelesen werden können. Dies ist sehr wichtig, denn Java bietet viele Sicherheits-Vorkehrungen, die durch das Auslesen von fremden Objekten nicht ausgehebelt werden dürfen.

Java birgt eine universelle Klasse namens Class. Sie hat alle Informationen zum Beschreiben von Objekten in Java. Werden Klassen geladen, so werden Objekte vom Typ Class zurückgegeben.

Die vier fundamentalen Teile einer Klasse sind die Konstruktoren, Felder (Instanz-Variablen, engl. fields), Methoden und Attribute (Klassen-Variablen). Auf die Attribute gewährt die Reflection-API keinen Zugriff, die anderen drei Teile können jedoch mittels Reflection ermittelt werden.

Die Reflection-API erweitert die Klasse Class um Methoden, mit denen das Innenleben einer Klasse abgefragt werden kann, sowie werden Klassen angeboten, mit denen die Antwort auf die Frage des Innenlebens abgebildet werden kann.

Die drei wichtigsten Methoden der erweiterten Klasse Class sind:

forName(): lädt eine Klasse mit dem angegebenen Namen
getName(): liefert den Namen der Klasse als String (nützlich zum Identifizieren einer Klasse anhand ihres Namens)

newInstance(): erzeugt und liefert ein Objekt dieser Klasse mit dem leeren Konstruktor (wenn er existiert)

Hinweis: die Methode **forName()** liefert nicht den Dateinamen der Klasse, sondern den Java Klassennamen. Untersucht man beispielsweise die Klasse „java.math.BigInteger“, so bekommt man als Ergebnis „java ReflectClass java.math.BigInteger“ und nicht den Pfad, in dem die Klasse abgelegt ist.

Außerdem bietet die Reflection weitere nützliche Methoden der Klasse Class:

- **getConstructor(), getConstructors(), getDeclaredConstructor()**
- **getMethod(), getMethods(), getDeclaredMethods()**
- **getField(), getFields(), getDeclaredFields()**
- **getSuperclass()**
- **getInterfaces()**
- **getDeclaredClasses()**

Hinweis: der Unterschied zwischen getConstructors und getDeclaredConstructors (dito bei getMethods...):
getDeclaredConstructors liefert nur die aufrufbaren, öffentlichen (engl. public) Konstruktoren, getConstructors liefert alle Konstruktoren, auch private und geschützte (engl. protected).

Die Reflection-API ist symmetrisch, d.h. man kann von einer Klasse auf ihre Methoden schließen und von Methoden erfahren, in welcher Klasse sie deklariert wurden. Weiterhin kann man vorwärts und rückwärts von einer Klasse zu ihren Methoden, von den Methoden zu den Parametern und wieder zur Klasse navigieren.

2. Motivation

Die Reflection-API ist ein mächtiges Tool zum Identifizieren von Klassen. Außerdem können die Methoden und Konstruktoren tatsächlich aufgerufen werden. Die Klassen können aber erst modifiziert werden, wenn sie geladen werden und der Zugriff auf die Attribute ist nicht möglich. Ein großer Schritt für Java mit der Reflection-API ist die Lösung des Problems, eine Instanz einer dynamisch geladenen Klasse zu erzeugen, die keinen leeren Konstruktor besitzt. Früher konnte man hier keine Instanz erzeugen.

Mit Reflection werden die Konstruktoren abgefragt, die newInstance-Methode ruft den Konstruktor auf und übergibt objects für die benötigten Parameter. Auch ist es nicht mehr nötig, daß Klassen eine statische Methode main haben, die nur String-Arrays als Parameter akzeptiert. Man könnte z.B. eine Programm schreiben, daß eine beliebige Methode einer Klasse aufruft, z.B.

meinProgramm meineKlasse(parameter1, parameter2).eineMethode("Dies ist die Methode, mit der gestartet wird");

In diesem Beispiel wird das Kommando meinProgramm gestartet, welches als Übergabeparameter eine Methode einer Klasse startet. Die Klasse wird geladen, mittels Reflection werden die Datentypen der Übergabeparameter ermittelt. Daraufhin wird der, zu den Datentypen der Parametern passende Konstruktor benutzt. Schließlich wird die Methode aufgerufen und ihr wird der obige String übergeben.

3. Code-Beispiele

(aus: http://java.sun.com/docs/books/tutorial/reflect/class/index.html)

3.1 Klassen-Objekte erhalten

Man kann ein Klassenobjekt auf verschiedene Wege erhalten:

Wenn eine Instanz der Klasse verfügbar ist, kann man **Object.getClass()** aufrufen. Die **getClass()**-Methode ist sehr nützlich, wenn man ein Objekt hat und nicht weiß, aus welcher Klasse es stammt. Der folgende Code erhält das Klassenobjekt des Objekts mystery:

```
Class c = mystery.getClass();
```

Möchte man die Superklasse einer gegebenen Klasse bekommen, so ruft man die **getSuperclass()**-Methode auf.
Im folgenden Beispiel gibt die Methode **getSuperclass()** ein Klassenobjekt der Klasse **TextComponent** zurück, da **TextComponent** die Superklasse von **TextField** ist:

```
TextField t = new TextField();
Class c = t.getClass();
Class s = c.getSuperclass();
```

Wenn man zur Zeit der Kompilierung den Namen einer Klasse kennt, so kann man das Klassenobjekt direkt durch anhängen von **.class** an den Klassennamen erhalten. In diesem Beispiel wird der Klasse den Knopfes (Button) zugewiesen.

```
Class c = java.awt.Button.class;
```

Ist der Klassenname zur Zeit der Kompilierung nicht bekannt, kann man die **forName()**-Methode benutzen. Im folgenden Beispiel wird der String namens strg mit „java.awt.Button“ initialisiert. Daraufhin liefert die **forName()**-Methode das Klassenobjekt der Klasse Button:

```
Class c = Class.forName(strg);
```

3.2 Den Klassen-Namen herausfinden

Jede Klasse in Java hat einen Namen. Wird eine Klasse deklariert, dann folgt der Name der Klasse direkt hinter dem Schlüsselwort „class“. In der folgenden Klassendeklaration ist der Name der Klasse Point:

public class Point {int x, y;}

Zur Laufzeit bekommt man den Klassennamen eines Klassenobjektes durch Aufruf der Methode **getName()**. Die Methode liefert einen String mit den voll qualifizierten Klassennamen:

```
import java.lang.reflect.*;
import java.awt.*;

class SampleName {

  public static void main(String[] args) {
    Button b = new Button();
    printName(b);
  }

  static void printName(Object o) {
     Class c = o.getClass();
     String s = c.getName();
     System.out.println(s);
  }
}
```

Das Programm gibt die folgende Zeile aus:

java.awt.Button

3.3 Die Modifier einer Klasse herausfinden

Eine Klassendeklaration kann die folgenden Modifier enthalten: **public**, **abstract**, oder **final**. Die Modifier stehen vor dem Schlüsselwort „class“. Im Beispiel sind die Modifier **public** und **final**:

public final Coordinate {int x, int y, int z}

Um die Modifier zur Laufzeit zu erhalten, geht man folgendermaßen vor:

Aufrufen von **getModifiers()** einer Klasse gibt die Modifier zurück. Diese werden nun auf **isPublic()**, **isAbstract()**, und **isFinal()** überprüft.

Das nächste Programm liefert die Modifier einer String-Klasse.

```
import java.lang.reflect.*;
import java.awt.*;

class SampleModifier {

  public static void main(String[] args) {
    String s = new String();
    printModifiers(s);
  }

  public static void printModifiers(Object o) {
    Class c = o.getClass();
    int m = c.getModifiers();
    if (Modifier.isPublic(m))
      System.out.println("public");
    if (Modifier.isAbstract(m))
      System.out.println("abstract");
    if (Modifier.isFinal(m))
      System.out.println("final");
  }
}
```

Die Ausgabe des Programms zeigt, daß die Klasse String public und final ist:

```
public
final
```

3.4 Identifizieren von Schnittstellen (Interfaces)

Ein Objekt wird nicht nur durch seine Klasse und Superklasse bestimmt, sondern auch durch seine Schnittstellen. In einer Klassen-Deklaration folgt die Schnittstellen-Definition nach dem Schlüsselwort „implements“. Die RandomAccessFile benutzt beispielsweise die Schnittstellen DataOutput und DataInput:

public class RandomAccessFile implements DataOutput, DataInput

Man kann die Methode **getInterfaces()** aufrufen, um herauszufinden, welche Schnittstellen eine Klasse implementiert. In der Reflection-API werden auch Interfaces durch Klassenobjekte dargestellt. Jedes Klassenobjekt im Array, den die Methode **getInterfaces()** liefert, entspricht einem Interface.

Das Programm gibt die Schnittstellen der Klasse RandomAccessFile aus:

```
import java.lang.reflect.*;
import java.io.*;

class SampleInterface {

  public static void main(String[] args) {
    try {
       RandomAccessFile r = new RandomAccessFile("myfile", "r");
       printInterfaceNames(r);
    } catch (IOException e) {
       System.out.println(e);
    }
  }

  static void printInterfaceNames(Object o) {
    Class c = o.getClass();
    Class[] theInterfaces = c.getInterfaces();
    for (int i = 0; i < theInterfaces.length; i++) {
      String interfaceName = theInterfaces[i].getName();
      System.out.println(interfaceName);
    }
  }
}
```

Die Interface-Namen, die ausgegeben werden, sind voll qualifiziert:
java.io.DataOutput
java.io.DataInput

3.5 Schnittstellen erforschen

Da Klassen auch Interfaces sein können, kann man mit der Methode **isInterface()** herausfinden, ob es sich um eine Klasse oder ein Interface handelt.

Um herauszufinden, welche öffentlichen (public) Konstanten ein Interface besitzt, benutzt man die **getFields()**-Methode. Auch bei einem Interface können die Modifier herausgefunden werden – **getModifiers()**.

Dieses Programm enthüllt, das der Observer ein Interface ist und Observable eine Klasse:

```
import java.lang.reflect.*;
import java.util.*;

class SampleCheckInterface {

  public static void main(String[] args) {
    Class observer = Observer.class;
    Class observable = Observable.class;
    verifyInterface(observer);
    verifyInterface(observable);
  }

  static void verifyInterface(Class c) {
    String name = c.getName();
    if (c.isInterface()) {
      System.out.println(name + " is an interface.");
    } else {
      System.out.println(name + " is a class.");
    }
  }
}
```

Dies ist die Ausgabe:

```
java.util.Observer is an interface.
java.util.Observable is a class.
```

3.6 Klassen Felder bestimmen

Schreibt man eine Anwendung, wie z.B. den Klassen-Browser, möchte man vielleicht herausfinden, welche Felder zu einer Klassen gehören. Die getFields()-Methode liefert einen Array mit Objekten die öffentlich (public) zugreifbare Felder enthält.

Ein Feld ist öffentlich, wenn es entweder:
- zu dieser Klasse
- zu der Superklasse der Klasse
- zu einem Interface der Klasse
- zu einer vererbten Schnittstelle der Klasse

gehört.

Die Methoden, die auf die Felder zugreifen, liefern den Feld-Namen, -Typ und die Modifier. Man kann sogar die Werte eines Feldes setzen oder auslesen. Das folgende Programm liest die Felder der Klasse **GridBagConstraints** aus und wendet die Methoden **getName()** und **getType()** auf diese an:

```
import java.lang.reflect.*;
import java.awt.*;

class SampleField {

  public static void main(String[] args) {
    GridBagConstraints g = new GridBagConstraints();
    printFieldNames(g);
  }

  static void printFieldNames(Object o) {
    Class c = o.getClass();
    Field[] publicFields = c.getFields();
    for (int i = 0; i < publicFields.length; i++) {
      String fieldName = publicFields[i].getName();
      Class typeClass = publicFields[i].getType();
      String fieldType = typeClass.getName();
      System.out.println("Name: " + fieldName +
        ", Type: " + fieldType);
      }
    }
}
```

Die Ausgabe sieht etwa so aus:
Name: RELATIVE, Type: int Name: REMAINDER, Type: int Name: NONE, Type: int Name: BOTH, Type: int Name: HORIZONTAL, Type: int Name: VERTICAL, Type: int . . .

3.7 Konstruktoren von Klassen enthüllen

Um eine Klasse zu erzeugen, wird eine spezielle Methode, der Konstruktor, aufgerufen. So wie Methoden können auch Konstruktoren überladen werdenund unterscheiden sich untereinander durch ihre Übergabeparameter.

Man erhält Information über die Konstruktoren einer Klasse, indem man die getConstructors()-Methode aufruft. Sie gibt einen Array von Konstruktoren zurück. Die Konstruktor-Klassen liefern weiterhin Methoden, mit denen der Name, die Modifier, die Parameterlisten und Exception des Konstruktors identifiziert werden können.
Ruft man **newInstance()** des Konstruktors auf, wird die zugehörige Klasse erzeugt.

Nun werden die Parameter der Klasse Rectangle ausgegeben:

```
import java.lang.reflect.*;
import java.awt.*;

class SampleConstructor {

  public static void main(String[] args) {
    Rectangle r = new Rectangle();
    showConstructors(r);
  }

  static void showConstructors(Object o) {
    Class c = o.getClass();
    Constructor[] theConstructors = c.getConstructors();
    for (int i = 0; i < theConstructors.length; i++) {
      System.out.print("( ");
      Class[] parameterTypes =
        theConstructors[i].getParameterTypes();
      for (int k = 0; k < parameterTypes.length; k ++) {
        String parameterString = parameterTypes[k].getName();
        System.out.print(parameterString + " ");
        }
      System.out.println(")");
      }
  }
}
```

Der erste Konstruktur hat keine Übergabeparameter. Bei den folgenden Konstruktoren werden die notwendigen Datentypen angegeben:

```
( )
( int int )
( int int int int )
( java.awt.Dimension )
( java.awt.Point )
( java.awt.Point java.awt.Dimension )
( java.awt.Rectangle )
```

3.8 Informationen über Methoden erhalten

Um herauszufinden, welche öffentlichen Methoden zu einer Klasse gehören, wird die Methode **getMethods()** aufgerufen. Sie liefert Methoden-Objekte zurück. Über diese kann der Methodenname, Rückgabetyp, die Modifier und Exceptions herausgefunden werden.

All diese Informationen wären brauchbar, wenn man einen Class-Browser oder einen Debugger schreibt. Mit Methode.**invoke()**, kann man sogar die Methode selbst aufrufen.

Das Beispielprogramm gibt den Namen, Rückgabetyp und die Parameter jeder öffentlichen Method der Polygon Klasse:

```
import java.lang.reflect.*;
import java.awt.*;

class SampleMethod {

  public static void main(String[] args) {
    Polygon p = new Polygon();
    showMethods(p);
  }

  static void showMethods(Object o) {
    Class c = o.getClass();
    Method[] theMethods = c.getMethods();
    for (int i = 0; i < theMethods.length; i++) {
      String methodString = theMethods[i].getName();
      System.out.println("Name: " + methodString);
      String returnString =
        theMethods[i].getReturnType().getName();
      System.out.println("   Return Type: " + returnString);
      Class[] parameterTypes = theMethods[i].getParameterTypes();
      System.out.print("   Parameter Types:");
      for (int k = 0; k < parameterTypes.length; k ++) {
        String parameterString = parameterTypes[k].getName();
        System.out.print(" " + parameterString);
      }
      System.out.println();
    }
  }
}
```

Eine gekürzte Version der Ausgabe des Programms sieht so aus:

Name: equals
** Return Type: boolean**
** Parameter Types: java.lang.Object**
Name: getClass
** Return Type: java.lang.Class**
** Parameter Types:**
Name: hashCode
** Return Type: int**
** Parameter Types:**
.
.
.
Name: intersects
** Return Type: boolean**
** Parameter Types: double double double double**
Name: intersects
** Return Type: boolean**
** Parameter Types: java.awt.geom.Rectangle2D**
Name: translate
** Return Type: void**
** Parameter Types: int int**

4. Fazit: Die Reflection API

von Dale Green (Mitbegründer von Java)

Die Reflection API repräsentiert, bzw. reflektiert die Klassen, Schnittstellen und Objekte in der Java Virtual Machine. Man kann die Reflection-API benutzen, wenn man Entwicklungs-Werkzeuge wie Debugger, Klassen-Browser und Oberflächen-Gestaltung schreibt. Mit der Reflection-API kann man:

- die Klasse eines Objektes herausfinden
- Information über die Modifier, Felder, Methoden, Konstruktoren und Superklassen einer Klasse erhalten
- herausfinden welche Konstanten und Methoden-Deklarationen zu einem Interface gehören
- Instanzen von Klassen erzeugen, deren Name zur Laufzeit nicht bekannt sind
- Lesen und setzen der Werte von Feldern, selbst wenn der Feldname zur Laufzeit nicht bekannt ist
- eine Methode eines Objekts aufrufen, selbst wenn die Methode zur Laufzeit nicht bekannt ist
- Einen Array erstellen, dessen Größe und Datentyp nicht vor Laufzeit bekannt sind, um dann den Inhalt des Arrays zu verändern

Allerdings sollte man die Reflection-API nicht benutzen, wenn andere Wege natürlicher oder ausreichend für die Java-Programmierung sind. Die Gewohnheiten anderer Programmiersprachen, wie z.B. Funktions-Pointer zu benutzen, verleiten, die Reflection-API auch gleiche Weise zu benutzen. Dies sollte nicht der Fall sein!

Ihr Programm wird einfacher zu debuggen und zu warten, wenn man nicht die Methoden-Objekte benutzt. Stattdessen sollte man eine Schnittstelle definieren und in ihr die Klassen implementieren, welche diese Tätigkeiten vollziehen.