Java Reflection

Andreas Lochbihler

Lehrstuhl Programmierparadigmen
Universität Karlsruhe

15. Mai 2008

Was ist Reflection?

Ein Paket (package) von Interfaces und Klassen, die dem Programm zur Laufzeit Zugriff auf geladene Klassen, deren Felder und Methoden (= reflektive Informationen) ermöglichen bis hin zur Manipulation der entsprechenden Objekte unter Wahrung oder Umgehung der Sichtbarkeitseinschränkungen.

Was bietet Reflection?

- Informationen über Klassen, Methoden und Felder
- Erzeugen von Objekten und Arrays mit dynamischen Typen
- Dynamische Methodenaufrufe
- Auslesen und Beschreiben von Felder
- Abschalten von Zugriffsbeschränkungen
- Auslesen der Annotations

Wozu kann man Reflection verwenden?

- Alle Arten von Programmier-Tools:
  - Debugger
  - Interpreter
  - Object Inspector
  - Class Browser
  - Testtools
- Object Serialisation
- Java Beans
- Dynamisches Laden von Code

Beispiel: Java Beans

Auszug aus einer graphischen Kompontente:

```
class SampleBean {
    ...
    public void setBackground(Color c) { ... }
    public Color getBackground() { ... }
    public void setForeground(Color c) { ... }
    public Color getBackground() { ... }
    ...
}
```

Reflection kann:

- Die beiden Properties `Background` und `Foreground` durch Abfrage aller Methoden ermitteln (Namenskonventionen).
- Einen Editor erstellen, der es möglich macht, die Kompontente *live* zu ändern.
- Eine Abhängigkeit von `SampleBean` ist nicht nötig

Reflection API: `java.lang.Class<T>`

Für jede Klasse existiert zur Laufzeit ein Objekt des Typs `Class`. `T` ist der Typ der Klasse.

Wichtigste Methoden dieser Klasse:

- `static Class<?> forName(String className)`
- `T newInstance()`
- `Field[] getDeclaredFields()`
- `Method[] getDeclaredMethods()`
- `Method[] getMethods()`
- `Method getMethod(String name, Class<?>... parameterTypes)`

Reflection API

Auch für Methoden und Felder existieren Klassen bzw. Objekte:

- `java.lang.reflect.Method`
  - `Class<?>[] getParameterTypes()`
  - `Class<?> getReturnType()`
  - `Object invoke(Object obj, Object... args)`

- `java.lang.reflect.Field`
  - `Object get(Object obj)`
  - `void set(Object obj, Object value)`
  - `Class<?> getType()`
  - `int getInt(Object obj)`

Analog für Konstruktoren:
`java.lang.reflect.Constructor<T>`

Beispiel: Zeitmessung

```
class Timer {
   static public void main(String[] args) throws ... {
      Class c = Class.forName(args[0]);
      Method m = c.getMethod("main", String[].class);

      Object[] args2 = new String[args.length - 1];
      System.arraycopy(args, 1, args2, 0, args2.length);

      long start = System.currentTimeMillis();
      m.invoke(null, args2);
      long end = System.currentTimeMillis();

      System.err.println("execution took "
                         + (end - start) + "ms");
   }
}
```

Beispiel: Zeitmessung (2)

```
> java Add 2 3
5
> java Time Add 2 3
5
Running 'Add' took 23ms
>
```

Reflection API: `java.lang.reflect.AccessibleObject`

Oberklasse von allen Klassen, die zugriffsbeschränkte Eigenschaften repräsentieren (`Method`, `Field`).

- ```
  public static void setAccessible
  (AccessibleObject[] array, boolean flag)
  throws SecurityException;
  ```
- ```
  public void setAccessible (boolean flag)
  throws SecurityException;
  ```

`SecurityException` wird bei installiertem `SecurityManager` und fehlenden Privilegien geworfen.

Ohne solche Hintertüren ist z.B. Serialisierung unmöglich.

Beispiel: Zugriff auf `private` Members

```
class Superhero {
   public final String name;
   private final String secretID;
   public Superhero(String name, String secretID) {
      this.name = name;
      this.secretID = secretID;
   }
}

class Reporter {
   public static void main(String[] args) throws ... {
      Superhero s = new Superhero("Batman",
                                  "Bruce Wayne");
      hackIdentity(s);
   }
```

Beispiel: Zugriff auf `private` Members (2)

```
    static void hackIdentity(Superhero s) throws ... {
        Field secret = Superhero.class.
                       getDeclaredField("secretIdentity");
        secret.setAccessible(true);
        System.out.println("Identity is " + secret.get(s));
        secret.set(s, "Clark Kent");
        System.out.println("Identity is now "
                           + secret.get(s));
    }
}
```

```
> java Reporter
Identity is Bruce Wayne
Identity is now Clark Kent
```

Warum ist `Class` generisch?

Mit Typ-Parametern können keine Objekte erzeugt werden:

```
class Factory<T> {
   ...
   void produce() {
      return new T(); // Compiler-Fehler
   }
}
```

Mit Reflection kann das Problem gelöst werden:

```
static <T> T factory(Class<T> c, String param) {
   T result = c.newInstance();
   ...
   return result;
}
factory(Person.class, "Andreas Lochbihler");
```

Praxis: Anwendung von Reflection

- *Eclipse*: Entwicklungsumgebung
  - Plugins werden über Reflection geladen und gestartet
- *KAWA*: Scheme-Interpreter
  - Aufruf von Java-Methoden
- Java-API
  - RMI / CORBA
  - Auswahl des Swing Look and Feels
  - Exception-Handling beim Event-Dispatch

Reflection und Annotations: generische Testklasse

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test { }

public class Foo {
    @Test public static void m() { ... }
    public static void m2() { ... }
    @Test public static void m3() {
        throw new RuntimeException("Error");
    }
}
```

Reflection und Annotations: generische Testklasse (2)

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

public class RunTestMethods {
   public static void main(String[] args) throws ... {
      for (Method m : Class.forName(args[0])
                        .getMethods())
         if (m.isAnnotationPresent(Test.class)) {
            try {
               m.invoke(null);
            } catch (... e) { ... }
         }
   }
}
```

Zusammenfassung

- Reflection bietet zahlreiche Informationen über die Struktur eines laufenden Programms
- Beschränkte Manipulation von Objekten möglich
- Benutzung der API ist bisweilen sperrig (Exceptions, primitive Typen)
- Für bestimmte Anwendungen oft die einzige Alternative zu nativem Code