Reflection

Carsten Gips (FH Bielefeld)

Ausgaben und Einblicke zur Laufzeit

```
public class FactoryBeispielTest {
    @Test
    public void testGetTicket() {
        fail("not implemented");
    }
}
```

```
@Target(value = ElementType.METHOD)
@Retention(value = RetentionPolicy.RUNTIME)
public @interface Wuppie {}
```

Wer bin ich? ... Informationen über ein Programm (zur Laufzeit)

`java.lang.Class`: Metadaten über Klassen

```
// usual way of life
Studi heiner = new Studi();
heiner.hello();

// let's use reflection
try {
    Object eve = Studi.class.getDeclaredConstructor().newInstance();
    Method m = Studi.class.getDeclaredMethod("hello");
    m.invoke(eve);
} catch (ReflectiveOperationException ignored) {}
```

Vorgehen

1. Gewünschte Klasse über ein `Class`-Objekt laden
2. Informationen abrufen (welche Methoden, welche Annotationen, ...)
3. Eine Instanz dieser Klasse erzeugen, und
4. Methoden aufrufen

Schritt 1: *Class*-Objekt erzeugen und Klasse laden

```java
// Variante 1 (package.MyClass dynamisch zur Laufzeit laden)
Class<?> c = Class.forName("package.MyClass");


// Variante 2 (Objekt)
MyClass obj = new MyClass();
Class<?> c = obj.getClass();

// Variante 3 (Klasse)
Class<?> c = MyClass.class;
```

Schritt 2: In die Klasse reinschauen

```java
// Studi-Klasse dynamisch (nach-) laden
Class<?> c = Class.forName("reflection.Studi");


// Parametersatz für Methode zusammenbasteln
Class<?>[] paramT = new Class<?>[] { String.class };

// public Methode aus dem **Class**-Objekt holen
Method pubMethod = c.getMethod("setName", paramT);
// beliebige Methode aus dem **Class**-Objekt holen
Method privMethod = c.getDeclaredMethod("setName", paramT);


Method[] publicMethods = c.getMethods();  // all public methods (incl. inherited)
Method[] allMethods = c.getDeclaredMethods();  // all methods (excl. inherited)
```

Schritt 3: Instanz der geladenen Klasse erzeugen

```java
// Class-Objekt erzeugen
Class<?> c = Class.forName("reflection.Studi");


// Variante 1
Studi s = (Studi) c.newInstance();

// Variante 2
Constructor<?> ctor = c.getConstructor();
Studi s = (Studi) ctor.newInstance();

// Variante 3
Class<?>[] paramT = new Class<?>[] {String.class, int.class};
Constructor<?> ctor = c.getDeclaredConstructor(paramT);
Studi s = (Studi) ctor.newInstance("Beate", 42);
```

Schritt 4: Methoden aufrufen ...

```java
// Studi-Klasse dynamisch (nach-) laden
Class<?> c = Class.forName("reflection.Studi");
// Studi-Objekt anlegen (Defaultkonstruktor)
Studi s = (Studi) c.newInstance();
// Parametersatz für Methode zusammenbasteln
Class<?>[] paramT = new Class<?>[] { String.class };
// Methode aus dem **Class**-Objekt holen
Method method = c.getMethod("setName", paramT);


// Methode auf dem **Studi**-Objekt aufrufen
method.invoke(s, "Holgi");
```

Demo: reflection.ReflectionDemo

Hinweis: Klassen außerhalb des Classpath laden

```
File folder = new File("irgendwo");
URL[] ua = new URL[]{folder.toURI().toURL()};

URLClassLoader ucl = URLClassLoader.newInstance(ua);
Class<?> c1 = Class.forName("org.wuppie.Fluppie", true, ucl);
Class<?> c2 = ucl.loadClass("org.wuppie.Fluppie");
```

Bemerkung zu Ordnerstruktur und Classpath; Demo: reflection.ClassLoaderDemo

Licht und Schatten

Nützlich:

- Erweiterbarkeit: Laden von “externen” Klassen in eine Anwendung
- Klassen-Browser, Debugger und Test-Tools

Nachteile:

- Verlust von Kapselung, Compiler-Unterstützung und Refactoring
- Performance: Dynamisches Laden von Klassen etc.
- Sicherheitsprobleme/-restriktionen

Gibt es eine Lösung ohne Reflection, wähle diese!

Wrap-Up

- Inspektion von Programmen zur Laufzeit: **Reflection**
  - `java.lang.Class`: Metadaten über Klassen
  - Je Klasse ein `Class`-Objekt
  - Informationen über Konstruktoren, Methoden, Felder
  - Anwendung: Laden und Ausführen von zur Compile-Zeit unbekanntem Code
  - Vorsicht: Verlust von Refactoring und Compiler-Zusicherungen!

LICENSE