HANSER

Leseprobe

Rainer Oechsle

Java-Komponenten

Grundlagen, prototypische Realisierung und Beispiele für Komponentensysteme

ISBN (Buch): 978-3-446-43176-8

ISBN (E-Book): 978-3-446-43591-9

Weitere Informationen oder Bestellungen unter

http://www.hanser-fachbuch.de/978-3-446-43176-8

sowie im Buchhandel.

2 Reflection

Mit Hilfe von Reflection kann man zur Laufzeit unterschiedliche Informationen über eine Klasse erfragen: Welche Attribute hat eine Klasse? Wie heißen ihre Attribute, welchen Typ und welche Sichtbarkeit haben sie? Welche Konstruktoren (Parametertypen, Ausnahmen, Sichtbarkeit) hat eine Klasse? Welche Methoden (Namen, Parametertypen, Rückgabetyp, Ausnahmen, Sichtbarkeit) hat eine Klasse? Welche Schnittstellen implementiert eine Klasse und von welcher Klasse ist sie abgeleitet? Dies ist für jede beliebige Klasse möglich, ohne dass man die betreffende Klasse beim Schreiben seines Codes schon kennen muss. Auch muss der Quellcode der zu untersuchenden Klasse zur Laufzeit nicht vorhanden sein; was man zur Laufzeit braucht, ist die entsprechende Class-Datei, die sich auch in einem Jar-Archiv befinden kann. Der Name der Klasse muss im Programm nicht „fest eingebrannt" sein („fest eingebrannt" ist beispielsweise die Klasse X, wenn man in seinem Programm `new X()` benutzt; der Klassenname kann nach dem Übersetzen des Programms zur Laufzeit nicht mehr dynamisch geändert werden). Es ist im Gegenteil möglich, dass man den Namen der zu untersuchenden Klasse dynamisch als String, z. B. aus einer Konfigurationsdatei oder von der Tastatur, einliest und damit die oben beschriebenen Informationen erfragen und anzeigen kann.

Mit Hilfe von Reflection kann man aber nicht nur Informationen auslesen, sondern man kann damit auch die Attribute eines Objekts einer Klasse ändern, ohne dass man beim Programmieren schon weiß, welche Attribute die Klasse hat. Ferner kann man in seinem Programm Objekte einer Klasse erzeugen, ohne dass man beim Programmieren weiß, wie die Klasse heißt und welche Parametertypen der Konstruktor benötigt. Entsprechendes gilt für das Aufrufen von Methoden.

Im Zusammenhang mit Java-Komponenten kann ein Komponenten-Framework mit Hilfe von Reflection zum Beispiel ein Objekt einer neuen Komponente erzeugen, deren Klasse beim Schreiben des Komponenten-Framework-Codes noch nicht bekannt war. Dem Komponenten-Framework muss zur Laufzeit lediglich der Name der Komponentenklasse mitgeteilt werden, dann kann das Framework Objekte davon erzeugen. Wie Sie im Laufe dieses Kapitels sehen werden, spielen Generics im Zusammenhang mit Reflection eine gewisse Rolle.

2.1 Grundlagen von Reflection

In diesem ersten Abschnitt über Reflection berücksichtigen wir die Tatsache, dass die durch Reflection beschriebenen Klassen, Schnittstellen und Methoden Generics verwenden können, zunächst noch nicht. Wie Reflection für Generics erweitert wurde, erfahren Sie dann in Abschnitt 2.2.

2.1.1 Die Klasse Class

Die zentrale Klasse von Reflection ist Class. Ein Objekt der Klasse Class beschreibt eine Klasse. Es gibt eine ganze Reihe von Möglichkeiten, wie man an ein Class-Objekt gelangt. Wenn man schon ein Objekt der Klasse hat, dann kann man sich mit getClass einfach das dazugehörige Class-Objekt geben lassen (X sei irgendeine Klasse):

```
X x = new X();
Class<?> c = x.getClass();
```

Wie oben zu sehen ist, ist die Klasse Class eine generische Klasse. Der Typparameter ist die Klasse, die vom Class-Objekt beschrieben wird (in obigem Beispiel also Class<X>). Da getClass eine Methode der Klasse Object ist, muss der Rückgabetyp für alle Objekte aller möglichen Klassen passen; er ist deshalb Class<?>. Es wäre zwar oben möglich, auf Class<X> zu casten. Dies hat aber eine (unterdrückbare) Warnung zur Folge, da eine vollständige Typüberprüfung zur Laufzeit nicht möglich ist, wie im vorherigen Kapitel schon erläutert wurde. Weil man aber weiß, dass die Variable x vom Typ X ist, muss das Objekt, das durch x referenziert wird, vom Typ X oder einer aus X abgeleiteten Klasse sein. Der Typparameter kann deshalb zumindest auf X eingeschränkt werden:

```
Class<? extends X> c = x.getClass();
```

Eine weitere Möglichkeit, an ein Class-Objekt zu gelangen, ist in folgendem Beispiel zu sehen:

```
Class<X> c = X.class;
```

In diesem Fall kann das Class-Objekt einer Variablen des Typs Class<X> ohne Warnungen zugewiesen werden, da der Compiler überprüfen kann, dass es sich in diesem Fall tatsächlich um ein Class<X>-Objekt handeln muss. (Bitte beachten Sie, dass dies im ersten Fall bei der Verwendung von getClass im Allgemeinen nicht geht, da bei der Zuweisung an die Variable x nicht immer die Anwendung des Operators new im Spiel sein muss, sondern es könnte z. B. auch sein, dass der Rückgabewert irgendeiner Methode an x zugewiesen wird. Diese Methode könnte auch ein Objekt einer von X abgeleiteten Klasse zurückgeben.) Bei der Verwendung von .class muss sich die Programmiererin allerdings bereits beim Schreiben ihres Programms auf die Klasse X festlegen; dies kann zur Laufzeit nicht mehr geändert werden (X ist statisch in den Programmcode „eingebrannt“). Wenn man sich das Class-Objekt von der statischen Methode forName der Klasse Class geben lässt, muss die Festlegung auf eine bestimmte Klasse nicht zur Programmierzeit erfolgen. Als Parameter benötigt die Methode forName den vollständigen Klassennamen (d. h. einschließlich des Package-

Namens) in Form eines Strings. Diesen String kann sich das Programm dynamisch beschaffen (z. B. aus einer Eingabe des Benutzers oder einer Konfigurationsdatei); zur Zeit des Programmierens muss die Programmiererin die Klasse nicht kennen:

```
try
{
    Class<?> c = Class.forName("javax.swing.JButton");
}
catch(ClassNotFoundException e)
{
    ...
}
```

Wie durch den Try-Catch-Block angedeutet, kann die Methode forName eine ClassNotFoundException werfen, falls sich die als String angegebene Klasse bzw. Schnittstelle nicht im Classpath des ausführenden Prozesses befindet.

Will man auf das Class-Objekt für ein Feld zugreifen, kann man sowohl die Variante mit .class als auch Class.forName einsetzen. Die folgenden beiden Anweisungen, in denen ein Zugriff auf ein Class-Objekt für ein zweidimensionales String-Feld beschafft wird, sind äquivalent, wobei in dem String-Argument der Methode Class.forName die Notation verwendet wird, die Java intern zur Beschreibung des Typs von Feldern nutzt:

```
Class<?> c1 = String[][].class;
Class<?> c2 = Class.forName("[[Ljava.lang.String;");
```

Übrigens erhält man auf beide Arten nicht nur das gleiche, sondern sogar dasselbe Class-Objekt (d. h. nach Ausführung der beiden Zeilen gilt c1 == c2 und nicht nur c1.equals(c2)).

Auch für Schnittstellen existieren Class-Objekte. Diese lassen sich über .class oder Class.forName besorgen. Die Variante über ein Objekt und getClass funktioniert nicht, weil hier immer die Klasse des Objekts zurückgegeben wird (selbst wenn die Variable, auf die getClass angewendet wird, den Typ der Schnittstelle hat).

Und auch für die primitiven Datentypen wie boolean, byte und int gibt es entsprechende Class-Objekte. Der Zugriff darauf lässt sich aber auch in diesem Fall nicht über getClass bewerkstelligen:

```
boolean b = true;
Class<?> c = b.getClass(); //Syntaxfehler!!!!
```

Es funktioniert aber die Variante mit .class:

```
Class<?> c = boolean.class;
```

Für die Beschaffung der Class-Objekte von Feldern primitiver Datentypen gibt es mehrere Varianten. Sowohl getClass als auch .class sowie Class.forName funktionieren. Die folgenden Zeilen zeigen ein Beispiel für die Beschaffung des Class-Objekts eines dreidimensionalen Felds des Typs boolean:

```
boolean[][][] boolArray = new boolean[10][20][30];
Class<?> c1 = boolArray.getClass();
Class<?> c2 = boolean[][][].class;
Class<?> c3 = Class.forName("[[[Z");
```

Auch hier gilt: c1 == c2 und c2 == c3 (und damit auch c1 == c3). Beachten Sie bitte, dass B schon für den primitiven Datentyp byte vergeben ist, so dass für boolean Z verwendet wird.

Hat man ein Class-Objekt, kann man z. B. unterschiedliche Eigenschaften der damit beschriebenen Klasse abfragen, z. B. ob die Klasse public ist, ob die Klasse final ist und ob es sich überhaupt um eine Klasse oder um eine Schnittstelle handelt. Auch kann man sich von einem Class-Objekt die Basisklasse sowie die implementierten Schnittstellen geben lassen. Sowohl die Basisklasse als auch die implementierten Schnittstellen werden wiederum als Class-Objekte zurückgeliefert. Weiterhin können die Attribute, Konstruktoren und Methoden einer Klasse erfragt werden.

2.1.2 Die Klasse Field

Mit Hilfe der Methoden getField, getDeclaredField, getFields und getDeclaredFields der Klasse Class kann man Informationen über die Attribute einer Klasse abfragen. Ein Attribut ist durch ein Objekt der Klasse Field repräsentiert, die ebenfalls eine der zu Reflection gehörenden Klassen ist. Wie die Methodennamen vermuten lassen, haben die Methoden getField und getDeclaredField den Rückgabetyp Field, weil sie genau ein Field-Objekt zurückgeben, während getFields und getDeclaredFields mehrere Field-Objekte in Form eines Field-Feldes (Field[]) zurückgeben. Bei den Methoden getField und getDeclaredField muss man den Namen des Attributs bereits kennen; dieser wird als String-Argument angegeben. Ein Field-Objekt wird zurückgeliefert, falls der Name des Attributs gültig war (andernfalls wird eine Ausnahme geworfen). Die Methoden getFields und getDeclaredFields sind dagegen parameterlos. Die Methoden, die „Declared“ in ihrem Namen haben, beziehen sich nur auf solche Attribute, die genau in der betrachteten Klasse definiert wurden (geerbte Attribute werden also nicht berücksichtigt), unabhängig von deren Sichtbarkeit (also auch einschließlich nicht-öffentlicher Attribute). Im Gegensatz dazu beziehen sich die Methoden, in deren Namen „Declared“ nicht vorkommt, auf alle, also auch auf die geerbten Attribute, allerdings nur auf die öffentlichen.

So wie ein Class-Objekt eine Klasse repräsentiert, beschreibt ein Field-Objekt ein Attribut einer Klasse. Dies sind einige wichtige Methoden der Klasse Field:

```
public Class<?> getType();
public Object get(Object obj) throws IllegalArgumentException,
                                     IllegalAccessException;
public void set(Object obj, Object value)
                               throws IllegalArgumentException,
                                     IllegalAccessException;
public void setAccessible(boolean flag) throws SecurityException;
public int getModifiers();
```

Über die Methode getType der Klasse Field kann man den Typ des Attributs erfragen (zur Erinnerung: es gibt auch für primitive Datentypen Class-Objekte). Hat man ein Objekt der Klasse, die das Attribut besitzt, das durch ein Field-Objekt repräsentiert wird, kann man mit Hilfe der Methoden get und set den Attributwert lesen und ändern. Dazu muss natürlich das Objekt, dessen Attributwert gelesen bzw. geändert werden soll, als Parameter angegeben werden; die Methoden get und set werden ja auf das Field-Objekt angewendet und nicht auf das Objekt, dessen Attribut gelesen oder geändert werden soll. Beim Zugriff auf ein

Attribut wird die Sichtbarkeit des Attributs standardmäßig respektiert. Das heißt, wenn auf ein privates Attribut von außerhalb der betreffenden Klasse zugegriffen wird, werfen die Methoden get und set die Ausnahme IllegalAccessException. Entsprechendes gilt für Attribute mit Standard- bzw. Protected-Sichtbarkeit. Wenn man will, kann man diesen Schutzmechanismus aber einfach abschalten: Wenn man setAccessible mit dem Parameter true auf das Field-Objekt vor dem Zugriff anwendet, sind alle Zugriffe auf das Attribut möglich, ganz gleich, welche Sichtbarkeit das Attribut besitzt.

Über die Methode getModifiers kann neben der Sichtbarkeit abgefragt werden, welche Java-Schlüsselwörter noch bei der Deklaration des Attributs verwendet wurden (z. B. transient).

2.1.3 Die Klasse Method

Ganz analog zu den Methoden für Attribute besitzt die Klasse Class die Methoden getMethod, getDeclaredMethod, getMethods und getDeclaredMethods für Methoden. Ähnlich wie bei den Attributen muss bei den Singularvarianten der Name einer Methode angeben werden und der Rückgabetyp ist Method, während die Pluralvarianten parameterlos sind und ein Method-Feld (Method[]) zurückliefern. Wegen der Konzepts des Überladens ist ein Name für eine Methode nicht eindeutig. Deshalb müssen bei den Singularvarianten neben dem Methodennamen auch die Typen der Parameter (in Form von Class<?>-Objekten) angegeben werden. Auch der Unterschied zwischen den Methoden mit und ohne „Declared“ im Namen ist derselbe wie bei den Methoden für die Attribute. So wie ein Objekt der Klasse Field für ein Attribut einer Klasse steht, repräsentiert ein Method-Objekt eine Methode. Die Klasse Method besitzt u. a. die folgenden Methoden:

```
public Class<?>[] getParameterTypes();
public Class<?> getReturnType();
public Class<?>[] getExceptionTypes();
public Object invoke(Object obj, Object... args)
          throws IllegalAccessException, IllegalArgumentException,
                 InvocationTargetException;
public void setAccessible(boolean flag) throws SecurityException;
public int getModifiers();
```

Wie aufgrund der Methodennamen erahnt werden kann, kann man mit getParameterTypes die Typen der Parameter, mit getReturnType den Rückgabetyp und mit getExceptionTypes die möglicherweise geworfenen Ausnahmen der Methode abfragen (die Parametertypen, der Rückgabetyp und die Ausnahmen werden alle durch Class-Objekte repräsentiert, wobei der Rückgabetyp void vom Class-Objekt void.class repräsentiert wird). Mit invoke kann die Methode, für die das Method-Objekt steht, aufgerufen werden. Da invoke auf das Method-Objekt angewendet wird und nicht auf das Zielobjekt, auf das die eigentliche Methode angewendet werden soll, muss das Zielobjekt als Parameter angegeben werden (auch dies ist ganz analog zu den Get- und Set-Methoden der Klasse Field). Weitere Parameter von invoke sind die aktuellen Parameter des Methodenaufrufs, der durch Method beschrieben wird. Diese Parameter müssen kompatibel sein zu den durch getParameterTypes beschriebenen Typen. Wegen der Allgemeinheit können es beliebig viele Parameter des Typs Object sein. Die variable Anzahl wird durch das Sprachkonzept Varargs von Java (notiert durch drei Punkte) realisiert. Falls ein Parametertyp ein primitiver Datentyp ist, muss das Argument

als ein Objekt der entsprechenden Wrapper-Klasse (für int z. B. ein Integer-Objekt) übergeben werden. Der Rückgabetyp ist im Allgemeinen ebenfalls Object. Falls das Method-Objekt zu einer Void-Methode gehört, ist der Rückgabewert immer null.

Bezüglich der Sichtbarkeitsüberprüfung und setAccessible gilt dasselbe wie bei der Klasse Field. Auch getModifiers entspricht der Methode desselben Namens der Klasse Field.

2.1.4 Die Klasse Constructor

Ganz analog zu den Attributen und Methoden kann man von einem Class-Objekt die Konstruktoren über die Methoden getConstructor, getDeclaredConstructor, getConstructors und getDeclaredConstructors erfragen. Rückgabetyp ist Constructor bzw. Constructor[]. Einige Besonderheiten der Konstruktoren spiegeln sich in der Reflection-Schnittstelle wider: Da alle Konstruktoren denselben Namen haben, muss bei den Singularvarianten getConstructor und getDeclaredConstructor kein Name, sondern es müssen nur die Typen der Parameter angegeben werden. Aufgrund der Tatsache, dass Konstruktoren nicht vererbt werden, ist der Unterschied zwischen den Methoden mit und ohne „Declared“ im Namen geringer als für Attribute und Methoden; es geht immer nur um Konstruktoren dieser Klasse: Die Methoden getDeclaredConstructor und getDeclaredConstructors beziehen sich auf alle Konstruktoren der Klasse, die Methoden getConstructor und getConstructors nur auf die öffentlichen Konstruktoren der Klasse.

Die Klasse Konstruktor ist eine generische Klasse. Der Typparameter entspricht der Klasse, um deren Konstruktor es geht. Wichtige Methoden der Klasse Constructor<T> sind:

```
public Class<?>[] getParameterTypes();
public Class<?>[] getExceptionTypes();
public T newInstance(Object... initArgs)
         throws InstantiationException, IllegalAccessException,
                IllegalArgumentException, InvocationTargetException;
public void setAccessible(boolean flag) throws SecurityException;
public int getModifiers();
```

Die Methode newInstance entspricht der Methode invoke der Klasse Method. Ein Unterschied besteht darin, dass es kein Objekt gibt, auf das der Konstruktor angewendet wird. Folglich sind die Parameter von newInstance nur die entsprechenden Konstruktorparameter. Ferner ist der Rückgabetyp nicht der allgemeine Typ Object, sondern T, der Typparameter der Klasse Constructor. Alle anderen Methoden sollten aufgrund der Erläuterungen zur Klasse Method selbsterklärend sein.

Falls eine Klasse einen parameterlosen Konstruktor besitzt und man ein Objekt mit Hilfe dieses Konstruktors erzeugen möchte, muss man übrigens die Klasse Constructor nicht notwendigerweise verwenden. In diesem Fall kann man auch die parameterlose Methode newInstance der Klasse Class benutzen. Auch diese Methode hat als Rückgabetyp den Typparameter T.