Michael Inden

Java

Die Neuerungen in Version 9 bis 12

Modularisierung, Syntax- und API-Erweiterungen

dpunkt.verlag

2 Syntaxerweiterungen in JDK 9

Bereits in JDK 7 wurden unter dem Projektnamen Coin verschiedene kleinere Syntaxerweiterungen in Java integriert. Für JDK 9 gab es ein Nachfolgeprojekt, dessen Neuerungen wir uns jetzt anschauen.

2.1 Anonyme innere Klassen und der Diamond Operator

Bei der Definition anonymer innerer Klassen konnte man den Diamond Operator bis JDK 8 leider nicht nutzen, sondern der Typ aus der Deklaration war auch bei der Definition explizit anzugeben. Praktischerweise ist es mit JDK 9 (endlich) möglich, auf diese redundante Typangabe zu verzichten. Als Beispiel dient die Definition eines Komparators mit dem Interface `java.util.Comparator<T>`.

Beispiel mit JDK 8

Bis JDK 8 musste man bei der Definition einer anonymen inneren Klasse den Typ noch wie folgt angeben:

```
final Comparator<String> byLengthJdk8 = new Comparator<String>()
{
    ...
};
```

Beispiel mit JDK 9

Die Änderung zu JDK 8 ist kaum sichtbar: Mit JDK 9 ist es nun erlaubt, die Typangabe wegzulassen und somit den Diamond Operator zu verwenden, wie wir dies von anderen Variablendefinitionen bereits gewohnt sind:

```
final Comparator<String> byLength = new Comparator<>()
{
    ...
};
```

Tipp: Alternative Definitionsvarianten von Komparatoren seit JDK 8

Für Komparatoren bietet es sich an, folgende Neuerungen aus JDK 8 zu nutzen:

1. Einen Lambda-Ausdruck

```
Comparator<String> byLength = (str1, str2) ->
                  Integer.compare(str1.length(), str2.length());
```

2. Die Methode `comparing()` aus dem Interface `Comparator<T>`

```
Comparator<String> byLength = Comparator.comparing(String::length);
```

Im Anhang A gehe ich auf einige Neuerungen aus JDK 8 ein. Dabei behandle ich unter anderem auch die Erweiterungen bei Komparatoren.

2.2 Erweiterung der `@Deprecated`-Annotation

Die `@Deprecated`-Annotation dient bekanntlich zum Markieren von obsoletem Sourcecode und besaß bislang keine Parameter. Das ändert sich mit JDK 9: Die `@Deprecated`-Annotation wurde um die zwei Parameter `since` und `forRemoval` erweitert. Die Annotation ist nun im JDK wie folgt definiert:

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(value={CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, PARAMETER,
      TYPE})
public @interface Deprecated {
    /**
     * Returns the version in which the annotated element became deprecated.
     * The version string is in the same format and namespace as the value of
     * the {@code @since} javadoc tag. The default value is the empty
     * string.
     *
     * @return the version string
     * @since 9
     */
    String since() default "";

    /**
     * Indicates whether the annotated element is subject to removal in a
     * future version. The default value is {@code false}.
     *
     * @return whether the element is subject to removal
     * @since 9
     */
    boolean forRemoval() default false;
}
```

Diese Erweiterung wurde nötig, weil in Zukunft geplant ist, veraltete Funktionalität aus dem JDK zu entfernen, statt sie – wie bislang für Java üblich – aus Rückwärtskompati-

bilitätsgründen ewig beizubehalten. Das folgende Beispiel zeigt eine Anwendung, wie sie aus dem JDK stammen könnte:

```
@Deprecated(since = "1.5", forRemoval = true)
```

Mithilfe der neuen Parameter kann man für veralteten Sourcecode angeben, in welcher Version (`since`) dieser mit der Markierung als `@Deprecated` versehen wurde und ob der Wunsch besteht, die markierten Sourcecode-Teile in zukünftigen Versionen zu entfernen (`forRemoval`). Weil beide Parameter Defaultwerte besitzen (`since = ""` und `forRemoval = false`), können die Angaben jeweils für sich alleine stehen oder ganz entfallen.

Diese Erweiterung der `@Deprecated`-Annotation kann man selbstverständlich auch für eigenen Sourcecode nutzen und so anzeigen, dass gewisse Funktionalitäten für die Zukunft nicht mehr angeboten werden sollen. Darüber hinaus empfiehlt es sich, in einem Javadoc-Kommentar das `@deprecated`-Tag zu verwenden und dort den Grund der Deprecation und eine empfohlene Alternative aufzuführen. Nachfolgend ist dies exemplarisch für eine veraltete Methode `someOldMethod()` gezeigt:

```
/**
 * @deprecated this method is replaced by someNewMethod()
 * ({@link #someNewMethod()}) which is more stable
 */
@Deprecated(since = "7.2", forRemoval = true)
private static void someOldMethod()
{
    // ...
}
```

2.3 Private Methoden in Interfaces

Allgemein bekannt ist, dass Interfaces der Definition von Schnittstellen dienen. Leider verlieren in Java die Interfaces immer mehr von ihrer eigentlichen Bedeutung. Unter anderem wurden mit JDK 8 statische Methoden und Defaultmethoden in Interfaces erlaubt. Mit beiden kann man Implementierungen in Interfaces vorgeben.[1] Das führt allerdings dazu, dass sich Interfaces kaum mehr von einer abstrakten Klasse unterscheiden: Abstrakte Klassen können ergänzend einen Zustand in Form von Attributen besitzen, was in Interfaces (noch) nicht geht.

Mit JDK 9 wurde der Unterschied zwischen Interfaces und abstrakten Klassen nochmals verringert, weil nun auch die Definition privater Methoden in Interfaces erlaubt ist. Das Argument dafür war, dass sich damit die Duplikation von Sourcecode in Defaultmethoden reduzieren ließe. Das mag richtig sein. Allerdings ist es für die meisten Anwendungsprogrammierer eher fraglich, ob diese jemals Defaultmethoden selbst

[1] Dieser Schritt war designtechnisch nicht schön, aber nötig, um Rückwärtskompatibilität und doch Erweiterbarkeit zu erreichen und um vor allem die Neuerungen im Bereich der Streams nahtlos ins JDK 8 integrieren zu können.

implementieren sollten. Trotz dieser Kritik möchte ich Ihnen das Feature anhand eines Beispiels vorstellen, da es eventuell für Framework-Entwickler von Nutzen sein kann.

Beispiel

Schauen wir uns zur Demonstration privater Methoden in Interfaces das nachfolgende Listing und vor allem die private Methode `myPrivateCalcSum(int, int)` sowie deren Aufruf aus den beiden öffentlichen Defaultmethoden an:

```
public interface PrivateMethodsExample
{
    // Tatsächliche Schnittstellendefinition - public abstract ist optional
    public abstract int method1();
    public abstract String method2();

    public default int sum(final String num1, final String num2)
    {
        final int value1 = Integer.parseInt(num1);
        final int value2 = Integer.parseInt(num2);

        return myPrivateCalcSum(value1, value2);
    }

    public default int sum(final int value1, final int value2)
    {
        return myPrivateCalcSum(value1, value2);
    }

    // Neu und unschön in JDK 9
    private int myPrivateCalcSum(final int value1, final int value2)
    {
        return value1 + value2;
    }
}
```

Kommentar

Vielleicht fragen Sie sich, warum ich den privaten Methoden in Interfaces so ablehnend gegenüberstehe. Tatsächlich wurde die Büchse der Pandora bereits mit JDK 8 und den Defaultmethoden geöffnet. Die privaten Methoden mögen für Framework-Entwickler mitunter praktisch sein, jedoch besteht die Gefahr, dass sie für »normale« Entwickler noch attraktiver werden und von diesen somit ohne großes Hinterfragen zur Applikationsentwicklung eingesetzt werden. Das wäre aber im Hinblick auf das Design und die Klarheit von Business-Applikationen ein Schritt in die falsche Richtung.[2] Dadurch wird unter Umständen dem Schnittstellenentwurf weniger Aufmerksamkeit gewidmet, basierend auf der Annahme, dass benötigte Funktionalität immer noch nachträglich hinzugefügt werden kann.

[2] Dieser Nachteil verliert durch Nutzung einer modernen Microservice-Architektur etwas an Gewicht, da die Designsünde dann relativ isoliert existiert.

2.4 Verbotener Bezeichner '_'

Bei den Bezeichnern gibt es eine kleine Änderung: Der Compiler erlaubt mit JDK 9 das Zeichen _ (Unterstrich) nicht mehr als Bezeichner.

Während folgende Zeile mit JDK 8 noch kompilierte

```
final String _ = "Underline";
```

produziert der Java-Compiler mit JDK 9 folgende Fehlermeldung:

```
as of release 9, '_' is a keyword, and may not be used as an identifier
```

Ich persönlich halte ein einzelnes Zeichen als Variablenbezeichner fast immer für einen Bad Smell und insbesondere gilt dies für den Unterstrich. Vermutlich sehen Sie dies ähnlich. Insofern stellt diese Änderung wohl eher selten ein Problem dar.