Programmierzettel: Records

Dominikus Herzberg, THM
Version 0.1, 1.6.2023

Lerninhalte

- Motivation
- Records in Java
- Syntax und Deklaration von Records
  - Grundlegende Syntax
  - Record mit explizitem Konstruktor
  - Record mit statischen Methoden:
  - Record mit zusätzlichen Instanzmethoden:
  - Record mit Implementierung von Interfaces:
- Übungen
  - Datenklasse `Student`
  - Streaming von Datenklassen
  - Datenklassen von geometrischen Objekten
- Besonderheiten von Datenklassen in Java
- Record-Pattern-Matching
  - Beispiel: Record-Pattern-Matching mit `instanceof` und Sealed Classes
  - Beispiel: Record-Pattern-Matching in einer Switch-Anweisung
  - Noch einmal: Pattern-Matching und Polymorphie

Motivation

In der modernen objektorientierten Programmierung sind Datenklassen ein weit verbreitetes Konzept. Eine Datenklasse ist eine spezielle Form der Klasse, die hauptsächlich dazu dient, Daten in Form von Variablen zu aggregieren, zu speichern und zu verwalten, ohne umfangreiche Logik oder Verhalten bereitstellen zu müssen. Datenklassen werden oft verwendet, um einfache Datenstrukturen oder Objekte zu repräsentieren. Einige Beispiele für Datenklassen sind `Point` (mit x- und y-Koordinaten), `Person` (mit Namen, Alter und Adresse) oder `Article` (mit Titel, Autor und Inhalt). Aber man kann mit Ihnen auch wesentlich mehr machen.

Vor der Einführung von Records in Java, hat man Datenklassen mit viel zusätzlichem Code aufgesetzt. In einer Klasse deklariert man die benötigten Variablen, legt Konstruktoren an und erstellt sogenannte Getter- und Setter-Methoden zum Auslesen und Setzen von Datenwerten. Zudem gilt es die Methoden `equals`, `hashCode` und `toString` mit geeigneten Implementierungen zu überschreiben. Das ist viel Arbeit und stets mit dem Risiko verbunden, dass man ungewollt Fehler macht. Der ganze Codeaufbau wird unübersichtlich und verstellt den Blick auf die Tatsache, dass es hier eigentlich "nur" um eine Datenklasse geht — einen Container für Daten.

Mit dem Schlüsselwort `record` bietet Java mittlerweile eine sehr einfache und kompakte Möglichkeit an, Datenklassen zu deklarieren. Records reduzieren den Boilerplate-Code und verbessern gleichzeitig die Lesbarkeit und Wartbarkeit des Codes. Zudem setzen Records auf das Prinzip der Unveränderlichkeit, der Immutabilität der Daten, indem alle Variablen der Datenklasse standardmäßig `final` sind und es ausschließlich Getter-Methoden und keine Setter-Methoden gibt. Ein Standard-Konstruktor, die Getter-Methoden und die Methoden `equals`, `hashCode` und `toString` werden automatisch generiert.

Records und die in ihnen angelegte Immutabilität bereichern einerseits die Gestaltungsmittel in Java und helfen andererseits dabei, mit weniger Code besser wartbaren Code zu schreiben, der aufgrund der Immutabilität besser wartbar und testbar ist.

Records in Java

Java-Records sind ein neues Sprachfeature, das mit Java 14 eingeführt und in Java 16 als Standardkonstrukt aufgenommen wurde.

Records sind spezielle Klassen, die mit einer kompakten und kurzen Syntax deklariert werden. Sie bestehen aus einem Namen und einer Liste von Komponenten, die die Variablen der Datenklasse repräsentieren. Durch die Deklaration eines Records generiert der Java-Compiler automatisch die folgenden Methoden:

- Getter-Methoden für alle Komponenten (ohne Setter, da Records unveränderlich sind)
- Einen Konstruktor, der alle Komponenten als Parameter akzeptiert
- Überschriebene `equals` - und `hashCode` -Methoden, die auf den Komponenten basieren
- Eine überschriebene `toString` -Methode, die eine lesbare Darstellung des Records liefert

Ein Beispiel für einen Record könnte eine Klasse "Person" sein:

```java
record Person(String name, int age, String address) {}
```

Mit dieser einzigen Zeile wird eine komplette Datenklasse erstellt, die alle notwendigen Methoden wie Getter, Konstruktor, `equals()` , `hashCode()` und `toString()` beinhaltet.

```text
jshell> new Person("Me", 22, "THM")
$4 ==> Person[name=Me, age=22, address=THM]

jshell> $4.age
|  Fehler:
|  age hat private-Zugriff in Person
|  $4.age
|  ^----^

jshell> $4.age() // so called getter method
$5 ==> 22

jshell> $4.equals(new Person("Me", 22, "THM"))
$6 ==> true
```

Syntax und Deklaration von Records

Die Syntax und Deklaration von Records in Java sind recht einfach und dennoch flexibel. Hier sind die grundlegenden Elemente der Syntax und einige mögliche Varianten:

Grundlegende Syntax

```java
record RecordName(Typ1 komponente1, Typ2 komponente2, ...) {}
```

Die grundlegende Syntax beginnt mit dem Schlüsselwort `record`, gefolgt vom Namen des Records und einer Liste von Komponenten, die durch Kommas getrennt sind. Jede Komponente hat einen Typ und einen Namen. Schließlich werden die Record-Definitionen in geschweiften Klammern eingeschlossen.

Beispiel: Grundlegende Syntax

```java
record Point(int x, int y) {}
```

Es sind auch Varargs verwendbar:

Beispiel: Record mit Varargs

```java
record IntTuple(int... numbers) {}
```

Da Records eine spezielle Form der Klasse sind, können auch Konstrukturen explizit gemacht oder etwa die Getter-Methoden angepasst werden.

Record mit explizitem Konstruktor

Manchmal ist es notwendig, zusätzliche Logik im Konstruktor eines Records einzufügen, z.B. um Parameter zu validieren oder zu transformieren. In diesem Fall können Sie einen expliziten Konstruktor hinzufügen.

Beispiel: Kompakte Deklaration des Record-Konstruktors

```java
record Point(int x, int y) { // assertions must be enabled
    public Point {
        assert x >= 0 && y >= 0 : "x and y must be non-negative";
    }
}
```

Diese kompakte Deklarationsmöglichkeit ist eine Kurzform des regulären Konstruktors. Wie man sieht: Die Kurzform möchte den Code kompakt halten und auch dafür sorgen, dass man z.B. die Variablenzuweisung nicht vergisst.

Beispiel: Reguläre (kanonische) Deklaration des Record-Konstruktors

```java
record Point(int x, int y) {
    public Point(int x, int y) {
        assert x >= 0 && y >= 0 : "x and y must be non-negative";
        this.x = x;
        this.y = y;
    }
}
```

Record mit statischen Methoden:

Sie können statische Methoden in einem Record definieren, ähnlich wie bei einer normalen Klasse.

Beispiel: Record mit statischer Methode

JAVA

```java
record Point(int x, int y) {
    public static Point origin() {
        return new Point(0, 0);
    }
}
```

Record mit zusätzlichen Instanzmethoden:

Es ist möglich, zusätzliche Instanzmethoden in einem Record zu definieren, die das Verhalten des Records über eine reine Datenhaltung hinaus erweitern.

Beispiel: Record mit zusätzlicher Instanzmethode

JAVA

```java
record Point(int x, int y) {
    public double distanceFromOrigin() {
        return Math.sqrt(x * x + y * y);
    }
}
```

Record mit Implementierung von Interfaces:

Records können Interfaces implementieren, ähnlich wie normale Klassen.

Beispiel: Record implementiert Interface

JAVA

```java
interface Printable {
    String print();
}

record Point(int x, int y) implements Printable {
    @Override public String print() {
        return "(" + x + ", " + y + ")";
    }
}
```

Es ist wichtig zu beachten, dass Records zwei wichtige Einschränkungen haben: (1) die Unveränderlichkeit der Daten (alle Komponenten sind implizit `final`) und (2) wird keine Vererbung unterstützt. Diese Einschränkungen sind gewollt (sonst muss man auf "normale" Klassen zurückgreifen) und werden neben der prägnanten Syntax als Vorteile verstanden.

Übungen

Datenklasse `Student`

Erstellen Sie eine Datenklasse namens `Student` mit Matrikelnummer, Namen und Alter.

1. Erzeugen Sie eine beispielhafte Instanz dieser Datenklasse namens `student`.
2. Lesen Sie das Alter des Studierenden aus.
3. Was machen Sie, wenn die Studentin bzw. der Student um ein Jahr älter geworden ist? Wie aktualisieren Sie den Datensatz?
4. Verändern Sie die Datenklasse so, dass negative Altersangaben nicht toleriert werden.
5. Im Dialog mit ChatGPT meinte die KI: Um ein Objekt zu löschen, müsse man es auf `null` setzen. Mit Bezug auf das Beispiel: `student = null`. Was halten Sie davon?

Streaming von Datenklassen

Erstellen Sie eine Liste von `Student`-Objekten und verwenden Sie die Stream-API, um

1. das Durchschnittsalter der Studierende zu ermitteln
2. nur die Studierenden zu filtern, die über 20 Jahre alt sind.

Datenklassen von geometrischen Objekten

Erstellen Sie verschiedene geometrische Objekte, die als Datenklassen realisiert sind.

1. Definieren Sie ein Interface `GeometricObject`, das eine Methode `area` enthält, um die Fläche des geometrischen Objekts zu berechnen.
2. Implementieren Sie Records für verschiedene geometrische Objekte, z.B. `Circle`, `Rectangle` und `Triangle`. Lassen Sie jedes Record das `GeometricObject`-Interface implementieren und die entsprechende `area`-Methode bereitstellen.
3. Erstellen Sie eine Liste von `GeometricObject`-Instanzen und berechnen Sie die Gesamtfläche aller Objekte in der Liste.

Besonderheiten von Datenklassen in Java

Records in Java bieten viele Vorteile, wie die Reduzierung von Boilerplate-Code, die Verbesserung der Lesbarkeit und die Unveränderlichkeit der Daten. Allerdings gibt es auch einige Einschränkungen und Best Practices, die Sie beachten sollten, um sie effektiv zu nutzen.

Unveränderlichkeit: Records sind unveränderlich, das bedeutet, dass alle ihre Komponenten implizit `final` sind. Sobald ein Record-Objekt erzeugt ist, können seine Daten nicht mehr geändert werden. Dies entspricht einer modernen, objektorientierten Programmierweise, wo man zunehmend auf die Immutabilität von Daten und eine Mischung mit einem funktionalen und datenflussorientierten Programmierstil Wert legt.

Im Gegensatz zu mutablen (veränderlichen) Datenobjekten, geht man bei immutablen Datenobjekten anders vor: Wertveränderungen resultieren in neuen Datenobjekten mit den geänderten Werten. Statt *in* einem Datenobjekte einen Wert zu ändern, wie man das bei mutablen Datenobjekten macht, wir ein *neues* Datenobjekt mit den geänderten Werten erzeugt.

JAVA

```java
record Counter(int n) {
    static Counter init() {
        return new Counter(0);
    }
    Counter tick() {
        return new Counter(n() + 1);
    }
}
```

TEXT

```
jshell> Counter c = Counter.init()
c ==> Counter[n=0]

jshell> c = c.tick()
c ==> Counter[n=1]

jshell> c = c.tick()
c ==> Counter[n=2]
```

Dem scheinbaren Mehraufwand, den Java mit dem Neuanlegen von Objekten und schlußendlich dem Entsorgen "alter", d.h. nicht referenzierte Objekte hat, stehen deutliche Engineering-Vorteile entgegen: Immutable Datenobjekte sind geeignet für die nebenläufige Verarbeitung und sie sind besser testbar, da keine impliziten Zustandsänderungen erfasst werden müssen.

Keine Vererbung: Records können keine Klassen erweitern und selbst nicht erweitert werden. Sie können jedoch Interfaces implementieren. Diese Einschränkung ist gewollt und führt zu einem unterschiedlichen Gebrauch von Record-Objekten (Datenobjekten) und Klassen-Objekten (Zustandsobjekten). Während die Vererbung, d.h. die Erweiterbarkeit von Klassen mit `extends` , die Polymorphie von Objekten zum Mittel der Wahl als Unterscheidungsmechanismus macht, müssen Datenobjekte explizit über das Instanz-Verhältnis mit einer Datenklasse unterschieden werden. Da der ausschließliche Vergleich mit dem `instanceof` -Operator ein wenig umständlich ist, hat Java das sogenannte Pattern-Matching (Mustervergleich) als syntaktische Neuerung im Zusammenspiel mit Records eingeführt.

Kombination mit Interfaces: Wenn Sie Records in Kombination mit Interfaces verwenden, können Sie das Beste aus beiden Welten nutzen: die einfache Syntax und Unveränderlichkeit von Records sowie die gemeinsame Struktur und das gemeinsame Verhalten, das Interfaces bieten.

Record-Pattern-Matching

Pattern Matching für Records ist eine leistungsfähige Funktion, die in Kombination mit der `instanceof`-Anweisung und Sealed Classes verwendet werden kann, um den Code lesbarer und einfacher zu gestalten. Hier ist eine Einführung in das Record-Pattern-Matching und einige Beispiele, wie es in der Praxis eingesetzt werden kann.

Record-Pattern-Matching: Pattern Matching ermöglicht es Ihnen, die Struktur eines Objekts zu untersuchen und dabei bestimmte Eigenschaften oder Zustände zu extrahieren. In Java wird das Pattern Matching mit der `instanceof`-Anweisung kombiniert, um den Code noch einfacher und lesbarer zu gestalten.

Sealed Classes: Sealed Classes bzw. Sealed Interfaces sind eine Ausdrucksmöglichkeit, die in Java eingeführt wurde, um die Erweiterung auf definierte Subtypen einzuschränken. Sie sind besonders nützlich in Kombination mit Pattern Matching, da sie eine endliche Anzahl von Subtypen garantieren und so den Compiler in die Lage versetzen, mögliche Fälle bei der Verwendung von Pattern Matching umfassend zu behandeln.

Beispiel: Record-Pattern-Matching mit `instanceof` und Sealed Classes

In diesem Beispiel erstellen wir eine einfache Hierarchie von geometrischen Objekten mit Sealed Classes und verwenden das Record-Pattern-Matching, um die Fläche der Objekte zu berechnen.

JAVA

```java
// Definieren Sie eine Sealed Class, um die Vererbung auf bestimmte Klassen einzuschränken
sealed interface GeometricObject permits Circle, Rectangle {}

record Circle(double radius) implements GeometricObject {}
record Rectangle(double width, double height) implements GeometricObject {}

double calculateArea(GeometricObject object) {
    if (object instanceof Circle c) {
        return Math.PI * c.radius() * c.radius();
    } else if (object instanceof Rectangle r) {
        return r.width() * r.height();
    } else {
        throw new IllegalArgumentException("Unsupported geometric object");
    }
}
```

In diesem Beispiel verwenden wir das Pattern Matching in Kombination mit der `instanceof`-Anweisung, um den Typ des `GeometricObject`-Objekts zu überprüfen und die entsprechende Fläche zu berechnen. Es geht aber deutlich eleganter mit Pattern-Matching.

Beispiel: Record-Pattern-Matching in einer Switch-Anweisung

> Für Java 20 ist das Pattern-Matching in einer `switch`-Anweisung ein Sprachfeature, das standardmäßig noch nicht freigeschaltet ist. Mit der Compiler-Option `--enable-preview` können Sie es aktivieren.

JAVA

```java
String classifyGeometricObject(GeometricObject object) {
    return switch (object) {
        case Circle c -> "Circle with radius " + c.radius();
        case Rectangle r -> "Rectangle with width " + r.width() + " and height " + r.height();
        default -> "Unknown geometric object";
    };
}
```

In diesem Beispiel verwenden wir das Record-Pattern-Matching in einer `switch`-Anweisung, um eine Beschreibung für verschiedene `GeometricObject`-Instanzen zu generieren. Die `switch`-Anweisung prüft den Typ des Objekts und gibt entsprechend eine Beschreibung zurück.

Indem Sie Record-Pattern-Matching in Ihren Java-Anwendungen verwenden, können Sie leicht verständlichen und einfach zu wartenden Code erstellen, der die Struktur von Objekten überprüft und ihre Eigenschaften extrahiert.

Noch einmal: Pattern-Matching und Polymorphie

Pattern Matching und Polymorphie sind beides mächtige Techniken in der objektorientierten Programmierung, aber sie haben unterschiedliche Anwendungsfälle und bieten unterschiedliche Vorteile. In einigen Situationen kann Pattern Matching mit Records gegenüber Polymorphie bevorzugt werden, weil es den Code übersichtlicher und einfacher zu verstehen machen kann.

Pattern Matching ermöglicht es, die Struktur eines Objekts zu untersuchen und bestimmte Eigenschaften oder Zustände zu extrahieren. In Kombination mit Records kann es dazu beitragen, den Code lesbarer und einfacher zu gestalten.

Polymorphie ermöglicht es, dass verschiedene Klassen das gleiche Interface implementieren und unterschiedliche Implementierungen der gleichen Methode anbieten. Dies erlaubt es, den Code flexibel und erweiterbar zu gestalten.

Das obige Record-Beispiel sieht mit "gewöhnlichen" Klassen und mit Polymorphie wie folgt aus:

JAVA

```java
interface GeometricObject {
    double area();
}

record Circle(double radius) implements GeometricObject {
    public double area() {
        return Math.PI * radius * radius;
    }
}

record Rectangle(double width, double height) implements GeometricObject {
    public double area() {
        return width * height;
    }
}
```

In diesem Beispiel verwenden wir Polymorphie, um die Berechnung der Fläche für verschiedene geometrische Objekte zu abstrahieren. Jede Klasse implementiert die `area`-Methode entsprechend.

Das Pattern Matching sticht in zwei Aspekten hevor:

1. Es macht den Code übersichtlicher und einfacher zu verstehen, da die Logik zur Berechnung der Fläche an einem zentralen Ort ist, anstatt über verschiedene Klassen verteilt zu sein.
2. Es ermöglicht es, die Anzahl der möglichen Typen, die von der `calculateArea`-Methode verarbeitet werden können, besser einzuschränken, insbesondere wenn man Sealed Classes verwendet.

Das bedeutet jedoch nicht, dass Pattern Matching generell Polymorphie ersetzen sollte. Zusammen erhöhen sie die zur Verfügung stehenden sprachlichen Ausdrucksmittel; es ist der Anwendungsfall und eine Design-Entscheidung die bestimmen, welches Sprachmittel zum Einsatz kommt.

Version 0.1
Last updated 2023-06-01 09:35:45 +0200