Funktionale Programmierung

Lambda-Ausdrücke

Funktionale versus objektorientierte Programmierung

Java ist als objektorientierte Programmiersprache in den neunziger Jahren des 20. Jahrhunderts entsprechend den damals aktuellen Programmierparadigmen entwickelt worden. Möglichkeiten und Prinzipien der damals schon durch Sprachen wie Lisp oder Scheme bekannten funktionalen Programmierung wurden bei der Sprachdefinition nicht berücksichtigt.

In den vergangenen Jahren wurde die funktionale Programmierung neu bewertet. Dabei wurde unter anderem erkannt, dass diese Vorgehensweise eine Reihe von Vorteilen bietet. So lassen sich ereignisbasierte (durch Ereignis ausgelöste Aktionen, beispielsweise in grafischen Oberflächen) und nebenläufige (die parallele Ausführung von Aufgaben bzw. das gemeinsame Lösen einer Aufgabe durch mehrere Prozesse) Anwendungen sehr gut mittels funktionaler Programmierung umsetzen. Letztgenannter Aspekt wurde dabei besonders im Zusammenhang mit der Einführung von Mehrkernprozessoren interessant.

Im Ergebnis entstanden Sprachen wie beispielsweise Scala oder Clojure, deren Programme jeweils in der Java Virtuell Machine ausgeführt werden können. Mit dem zunehmenden Erfolg dieser Sprachen wuchs die Anforderung an Java, funktionale Techniken im Sprachkern zu unterstützen. Die Hauptschwierigkeit bei der Umsetzung bestand dabei darin, das Konzept in die bestehende Sprache zu integrieren und gleichzeitig die Kompatibilität zu bestehenden Programmen zu gewährleisten.

Java bietet seit Version 8 Möglichkeiten der funktionalen Programmierung. Die folgenden Abschnitte stellen die zugrundeliegenden Ideen und Techniken vor.

Der Begriff Lambda-Ausdruck

Der Begriff des **Lambda-Ausdrucks** geht auf den Lambda-Kalkül zurück, Dieses wurde ca. 1930 von Alonzo Church und Stephen Cole Kleene in die mathematische Logik eingeführt. Beim Lambda-Kalkül handelt es sich um eine formale Sprache, welche zur Untersuchung von Funktionen genutzt wird. Sie dient der Beschreibung von Funktionen und gebundenen Parametern. Verwendung findet sie heute unter anderem in der theoretischen Informatik und in der Linguistik.

Das Symbol des Lambda-Kalküls ist der elfte Buchstabe des griechischen Alphabets, das Lambda, in der Version der Kleinschreibung. Dies geht auf das Zeichen ^ zurück, welches in der Principia Mathematica (einem mathematischen Grundlagenwerk vom Anfang des 20. Jahrhunderts) benutzt wurde, um freie Variablen zu kennzeichnen.

Was sind Lambda-Ausdrücke?

Ein Lambda-Ausdruck – im Rahmen der funktionalen Programmierung als **anonyme Funktionen** bezeichnet – ist ein Block Programmcode mit Parametern, der im Programm 'herumgereicht' werden kann und dessen Code zu einem späteren Zeitpunkt (ein- oder mehrfach) ausgeführt werden soll.

Die Übergabe des Codes erfolgt an funktionale Interfaces, also an Interfaces, welche genau eine abstrakte Methode definieren. In den Versionen vor Java 8 wurden in derartigen Szenarien anonyme Klassen genutzt, welche genau diese eine Methode des Interfaces implementieren. Lambda-Ausdrücke sind eine kompaktere Schreibweise zu deren Umsetzung. Sie besitzen damit den Vorteil, dass der Code übersichtlicher wird, da mehrere Programmzeilen einer inneren Klasse zu einem kompakten Aufruf verdichtet werden.

Beispiel: *com\herdt\java9\kap19\Item.java* und *com\herdt\java9\kap19\ComperatorTest.java*

Das Beispiel stellt die vor der Version 8 genutzte Lösung einer inneren Klasse und die Verwendung eines Lambda-Ausdrucks prinzipiell gegenüber (auf die konkrete Syntax des Lambda-Ausdrucks wird in einem folgenden Abschnitt eingegangen). Es wird jeweils eine Liste von Artikelnamen, die Bestandteil einer Collection vom Typ `List` sind, durch die Implementierung der Vergleichsfunktion `compare()` des Interfaces `Comperator` sortiert.

Im Vergleich wird ersichtlich, dass das sogenannte 'vertikale Problem' anonymer innerer Klassen, die Anzahl mehrerer Zeilen, durch die 'horizontale Lösung' einer einzeiligen Anweisung gelöst wird.

```
   package com.herdt.java9.kap19;
①  import java.util.ArrayList;
   import java.util.List;
   class Item
   {
②     private String article;
      private int quantity;

③     Item(String article, int quantity)
      {
        this.article = article;
        this.quantity = quantity;
      }
④  …
⑤     public void printName()
      {
        System.out.println("Artikel: " + article);
      }
```

```
⑥    public static List<Item> createList(){
       List<Item> itemList = new ArrayList<>();
       itemList.add( new Item("Bleistift", 75) );
       …
       itemList.add( new Item("Block A5", 175) );
       return itemList;
     }
}
```

① Der Import der notwendigen Klassen.

② Die Klasse `Item` besitzt zwei Attribute zur Speicherung des Artikelnamens und des -bestands.

③ Der Konstruktor der Klasse.

④ Die Klasse enthält weitere Methoden, die für dieses Beispiel nicht benötigt werden.

⑤ Die Methode `printName` gibt auf der Konsole den Namen des Artikels aus.

⑥ Die Methode `createList()` erstellt eine Liste mit Beispielartikeln. Als Rückgabewert liefert sie eine Liste mit Objekten der Klasse `Item`.

```
…
① import java.util.Comparator;
public class ComparatorTest
{
  public static void main(String[] args) {
②   List<Item> itemList = Item.createList();
③   printList(itemList, "Originalliste");

④   Collections.sort(itemList, new Comparator<Item>()
      {
        public int compare(Item it1, Item it2){
          return
            it1.getItemName().compareTo(it2.getItemName());
      }
    });
⑤   printList(itemList,
      "Artikel aufsteigend sortiert (innere Klasse)");

⑥   Collections.sort(itemList,
      (Item it1, Item it2) ->
        it2.getItemName().compareTo(it1.getItemName()));
⑦   printList(itemList,
      "Artikel absteigend sortiert (Lambda-Ausdruck)");
  }

⑧  static void printList(List<Item> list, String text)
  { … }
}
```

① Das Interface `Comparator` wird importiert. Dieses definiert die Methode `compare`.
② Mit dem Aufruf der Methode `createList` der Klasse `Item` wird eine Beispielliste erzeugt.
③ Die Namen der in der erzeugten Liste enthaltenen Artikel werden mit der Hilfsmethode `printList()` ausgegeben.
④ Die Methode `sort` der Klasse `Collections` benötigt zwei Parameter: Die zu sortierende Liste sowie eine Implementierung des Interfaces `Comparator`, welche die Methode `compare(e1, e2)` implementiert. Zu diesem Zweck wird eine anonyme innere Klasse mit der benötigten Methode erstellt und als zweiter Parameter übergeben.
⑤ Ausgabe der mit der inneren Klasse aufsteigend sortierten Liste.
⑥ Bei der Sortierung mit einem Lambda-Ausdruck wird dem Interface die für die Methode `compare` zu verwendende Implementierung als Funktion zuzüglich der in dieser zu verwendenden Parameter übergeben.
⑦ Ausgabe der mit dem Lambda-Ausdruck absteigend sortierten Liste.
⑧ Die Methode `printList` gibt die übergebene Liste nach einem Überschriftstext aus.

Lambda-Ausdrücke

Voraussetzungen

Voraussetzung für die Integration von Lambda-Ausdrücken in die bestehende Sprache Java waren einige Spracherweiterungen:

- Default-Methoden in Interfaces
- Funktionale Interfaces
- Referenzen auf Methoden

Einsatz

Verwendung finden die Lambda-Ausdrücke

- überall dort, wo funktionale Interfaces eingesetzt werden, beispielsweise für die nebenläufige Programmierung,
- in der seit Java 8 neuen Stream-API, einer Erweiterung des Collection-Frameworks, welche unter anderem Parallelzugriffe auf Collections ermöglicht,
- bei der Erweiterung von diversen APIs, beispielsweise für den Umgang mit Dateien.

Syntax

Lambda-Ausdrücke bestehen aus drei Teilen:

Aufbau	Parameterliste	-> Operator	Anweisungscode
Beispiel	`(int x, int y)`	`->`	`x + y`

- Die **Parameterliste** enthält die mit Komma getrennten **Formalparameter** der abstrakten Methode des funktionalen Interfaces.
  - Kann der Compiler die Parametertypen aus dem Kontext ableiten, ist deren Angabe optional.
  - Enthält die Parameterliste nur einen Wert, können die Klammern entfallen.
  - Bei einer leeren Parameterliste müssen die Klammern angegeben werden.
- Der **Operator ->** steht zwischen der Parameterliste und dem Anweisungscode.
- Beim **Anweisungscode** kann es sich um eine einzelne Anweisung oder um einen Anweisungsblock mit mehreren Anweisungen handeln.
  - Mehrzeilige Anweisungsblöcke sind in ein Klammerpaar `{}` einzuschließen.
  - Wird das Klammerpaar `{}` verwendet, muss mit `return` ein Rückgabewert definiert werden.
  - Direkt im Anweisungscode dürfen die Schlüsselwörter `break` und `continue` nicht verwendet werden. Innerhalb von Schleifen im Anweisungscode ist dies jedoch erlaubt.
  - Aus dem Anweisungscode heraus ist der Zugriff auf als `final` gekennzeichnete Variablen des umschließenden Bereichs möglich.

Beispiele für mögliche Lambda-Ausdrücke

```
(int x, int y) -> x + y
(x,y) -> { return x + y; }
(int x, int y) -> { System.out.println ( x + y ); return x + y; }
() -> 42
```

Funktionale Interfaces

Aufbau

Lambda-Ausdrücke können in Java nur an **funktionale Interfaces** übergeben werden. Ein Interface ist funktional, wenn es neben einer beliebigen Anzahl von Default- und statischen Methoden genau eine abstrakte Methode definiert.

Über die in Java 8 neu eingeführte Annotation `@FunctionalInterface` kann ein Interface als funktional gekennzeichnet werden. Der Compiler meldet in diesem Fall einen Fehler, wenn das Interface nicht den Anforderungen an ein funktionales Interface entspricht.

Beispiel: com\herdt\java9\kap19\Converter.java und com\herdt\java9\kap19\UseConverter.java

Im Beispiel wird ein funktionales Interface mit der abstrakten Methode `convert` zur Konvertierung von Werten definiert und im Testprogramm verwendet. Die jeweils auszuführende Methodenimplementierung wird als Lambda-Ausdruck definiert.

```
package com.herdt.java9.kap19;
① @FunctionalInterface
② interface Converter
{
    double convert(double input);
}
```

① Durch die Annotation `@FunctionalInterface` wird das Interface als funktional gekennzeichnet und diesbezüglich vom Compiler geprüft.

② Die Methode `convert` wird deklariert. Sie nimmt einen Eingabewert vom Typ `double` entgegen und gibt einen Wert vom Typ `double` zurück.

```
package com.herdt.java9.kap19;
public class UseConverter
{
①  static double calculation
     (Converter converter, double input)
   {
     return converter.convert(input);
   }

   public static void main(String[] args)
   {

②    System.out.println
       (calculation (input -> (input-32)*5.0/9.0, 100 ));
③    System.out.println
       (calculation (input -> (input/1.609344), 10));
   }
}
```

① Die statische Methode `calculation` besitzt zwei Eingabeparameter: eine Instanz des funktionalen Interfaces `Converter` und den umzurechnenden Wert. Als Rückgabewert liefert sie das Resultat der auf den umzurechnenden Wert ausgeführten Methode `convert` der Interface-Instanz.

② Beim Aufruf der Methode `calculation` wird ein Lambda-Ausdruck zur Konvertierung eines Fahrenheit- in einen Celcius-Wert angegeben. Dieser Code wird als Methodenimplementierung der `Converter`-Instanz genutzt. Als zweiter Parameter wird der umzurechnende Wert (`100`) übergeben.

③ Die zweite Verwendung von `calculation` nutzt als Lambda-Ausdruck die Umrechnung von Kilometern in Meilen. Dieser Code wird wiederum als Methodenimplementierung von `convert` genutzt. Der umzurechnende Wert ist hier `10` Kilometer.

Vordefinierte funktionale Interfaces

Neben der Möglichkeit eigene Interfaces zu erstellen, bietet Java im Package `java.util.function` bereits eine Reihe vordefinierter funktionaler Interfaces an. Einige listet die Tabelle auf:

Bei den verwendeten Parametertypen handelt es sich um generische Typen. Generics sind nicht Gegenstand dieses Buches. Im nachfolgenden Beispiel wird als Typ eine Klasse genutzt.

Name	Typ Rückgabewert	Name der abstrakten Methode	Beschreibung
`Consumer<T>`	`void`	`accept()`	konsumiert einen Wert vom Typ `T`
`Function<T, R>`	`R`	`apply()`	eine Funktion mit einem Argument vom Typ `T`
`Predicate<T>`	`boolean`	`test()`	eine Funktion, welche einen Wert vom Typ boolean liefert
`Supplier<T>`	`T`	`get()`	liefert einen Wert vom Typ `T`

Beispiel: *com\herdt\java9\kap19\Item.java* und *com\herdt\java9\kap19\StockControl.java*

Das Beispiel nutzt die beiden vordefinierten funktionalen Interfaces `Predicate` und `Consumer`. Mittels `Predicate` wird überprüft, ob der Bestand eines Artikels kleiner 100 ist. Ist dies der Fall, wird `Consumer` genutzt, um eine Bestandserhöhung durchzuführen.

```
…
class Item
{
…
①   void addUnits(int number)
    {
      quantity += number;
    }

②   int getItemQuantity()
    {
      return quantity;
}

③   @Override
④   public String toString()
    {
      return "Artikel: "  + article + ", Bestand: " + quantity;
    }
…
}
```

① Die Methode `addUnits` erhöht den Bestand um die übergebene Anzahl.

② Die Methode `getItemQuantity` liefert den aktuellen Bestand eines Artikels.

③ Die Annotation `@Override` kennzeichnet die Methode `toString` als überschriebene Methode.

④ Die Methode `toString` der Superklasse `Object` wird überschrieben. Sie gibt einen Text mit dem Namen und Bestand des Artikels aus.

```
   package com.herdt.java9.kap19;
①  import java.util.function.Consumer;
   import java.util.function.Predicate;

   public class StockControl
   {
     public static void main(String[] args)
     {
②      Item[] items =
       {
         new Item("Bleistift", 75),
         new Item("Ordner", 10)
       };

③      for (Item item: items)
       {
④        refill(item,
          item_ -> item_.getItemQuantity() < 100,
          item_ -> item_.addUnits(100));
          System.out.println(item);
       }
     }

⑤    public static void refill(Item item,
       Predicate<Item> predicate,
       Consumer<Item> consumer)
     {
⑥    if (predicate.test(item))
⑦      consumer.accept(item);
     }
   }
```

① Die beiden funktionalen Interfaces `Consumer` und `Predicate` werden importiert.

② Ein Feld `items` mit zwei Artikeln (Objekten der Klasse `Item`) wird erstellt.

③ Das Feld `items` wird in einer foreach-Schleife durchlaufen.

④ Für jedes Element des Feldes wird die Methode `refill` aufgerufen. Als Parameter werden dieser, das aktuelle Objekt (der jeweilige Artikel) sowie zwei Lambda-Ausdrücke übergeben. Der erste prüft mit der Methode `getItemQuantity`, ob der Bestand des Artikels kleiner `100` ist. Der zweite beinhaltet die Auffüllaktion, indem mit der Methode `addUnits` der Bestand um `100` erhöht wird.

⑤ Die Definition der Methode `refill`. Diese besitzt drei Eingabeparameter. Neben einem Objekt der Klasse `Item` wird jeweils eine Instanz der Interfaces `Predicate` und `Consumer` erwartet. Die Interfaces sind dabei jeweils auf die Klasse `Item` typisiert. Werden den beiden Parametern Lambda-Ausdrücke übergeben, werden diese als auszuführende Methodenimplementierung der jeweils vorhandenen abstrakten Methode genutzt.

⑥ Mit der Methode `test` wird durch den übergebenen Lambda-Ausdruck die Bestandsgröße geprüft. Als Rückgabewert liefert die Methode `true` oder `false`.

⑦ Hat die Überprüfung `true` ergeben, wird die Methode zum Auffüllen des Bestandes mit dem Code des im Lambda-Ausdruck übergebenen Codes ausgeführt ('konsumiert').

```
Artikel: Bleistift,Bestand: 175
Artikel: Ordner,Bestand: 110
```

Ausgabe des Programms „StockControl.java“

Referenzen auf Methoden

Methodenreferenzen

Lambda-Ausdrücke stellen die Implementierung einer Methode dar. Sie stellen anonyme Methoden dar, welche als Instanzen eines funktionalen Interfaces genutzt werden. Ist die Methode jedoch schon vorhanden, muss man nicht erst eine analoge definieren, sondern kann die vorhandene direkt verwenden. Für diese Fälle werden Methodenreferenzen genutzt.

Eine Methodenreferenz bezieht sich auf eine Methode ohne diese aufzurufen und kann sowohl eine 'normale' Methode als auch einen Konstruktor referenzieren. Letztgenannte werden auch als Konstruktorreferenzen bezeichnet.

Die allgemeine Syntax einer Methodenreferenz lautet

`<Klasse>|<Objekt>::<Methode>`

Es gibt vier Arten von Methodenreferenzen:

Referenz auf	Syntax	Beispiel
eine statische Methode	`<Klassenname>::<NameStatischeMethode>`	`String::valueOf`
eine Instanzmethode eines Objekts	`<Objektname>::<NameInstanzMethode>`	`s::toString`
eine Instanzmethode einer Klasse	`<Klassenname>::<NameInstanzMethode>`	`Object::toString`
auf einen Konstruktor	`<Klassenname>::<new>`	`String:new`

Beispiel: *com\herdt\java9\kap19\MethodRef.Java*

Das Beispiel nutzt das funktionale Interface `Supplier`. Dieses besitzt die Methode `get`, welche einen Wert liefert. Die Implementierung wird einmal als Lambda-Ausdruck und einmal als Methodenreferenz jeweils unter Verwendung der vorhandenen Methode `toString` geliefert.

```
package com.herdt.java9.kap19;
import java.util.function.Supplier;

public class MethodRef
{
①  public static void print(Supplier<String> supplier)
   {
     System.out.println(supplier.get());
   }
   public static void main(String[] args)
   {
     String s =
       "Ausgabe mit Methodenreferenz bzw. Lambda-Ausdruck";
②    print(() -> s.toString());
③    print(s::toString);
   }
}
```

① Die Methode `print` nimmt den Lambda-Ausdruck als Implementierung des Interface `Supplier` entgegen. Mit dem Aufruf von `get` wird der Wert geliefert und ausgegeben.

② Die vorhandene Methode `toString` wird in einem Lambda-Ausdruck verwendet und der Methode `print` übergeben.

③ Hier erfolgt die Übergabe an die Methode `print` in Form einer Methodenreferenz, welche sich auf die Methode `toString` bezieht. Der Aufruf erstellt den gleichen Lambda-Ausdruck wie unter ②.