Ausgabe 04/2019

Java aktuell

iJUG Verbund
www.ijug.eu

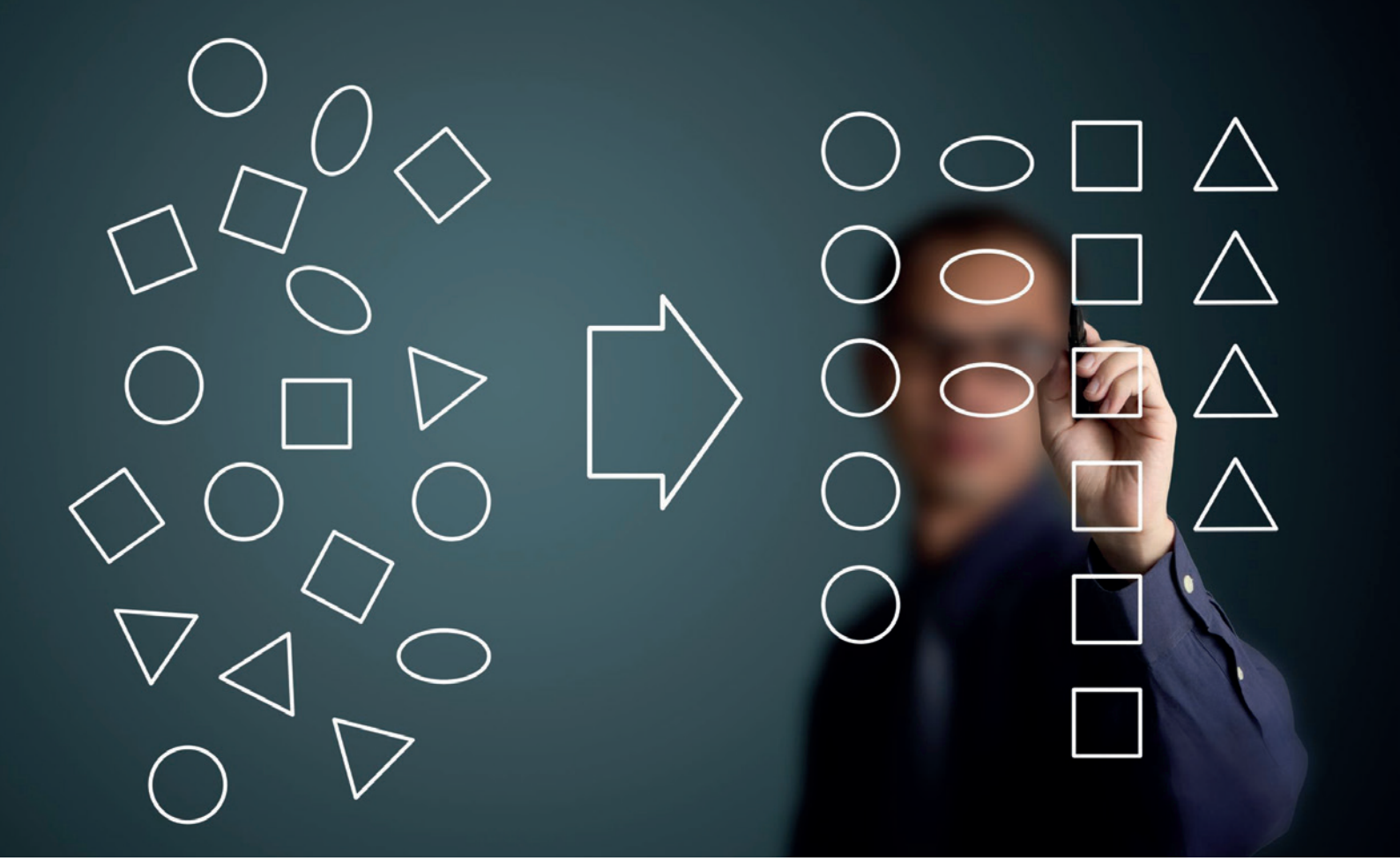
Java 12
Die neuen Features im Überblick

Jakarta EE
Neuigkeiten und Ergebnisse der Verhandlungen

Web-APIs
Diese Programmierschnittstellen erleichtern Ihnen die Arbeit

Java im Wandel

Java aktuell
Praxis. Wissen. Networking. Das Magazin für Entwickler
D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR
4 191978 304903 04



Eine moderne und konsistente Implementierung natürlicher Ordnung bei Java-Objekten – Teil 2

Christian Heitzmann, SimplexaCode

Eigene Java-Klassen sortierbar, also komparabel zu machen, ist eine elementare und wichtige Aufgabe. Gemessen an seiner Bedeutung wird dieses Thema in der Grundlagenliteratur jedoch nur sehr spärlich behandelt. Dieser zweiteilige Artikel zeigte in der letzten Ausgabe die Evolution der Methoden über die verschiedenen Java-Versionen hinweg auf und bot eine Code-Vorlage für eine saubere Implementierung ab Java 8 an. Nachdem der Leser darin mit der Theorie einer modernen und konsistenten Implementierung betraut wurde, wirft der hier vorliegende zweite Teil des Artikels einen Blick auf etwas realistischere Beispiele.

Die Beispielklasse, um die sich in diesem Artikel alles dreht, nennt sich `Customer` und repräsentiert einen Kunden aus einem Onlineshop. Natürlich müssen diese Kunden vergleichbar, also komparabel sein, denn sie werden ziemlich sicher irgendwo und irgendwann einmal sortiert werden müssen *(siehe Listing 1)*.

Natürlich sollen in einer solchen Klasse zu Demonstrationszwecken so viele verschiedene Attribute (englisch „fields“) wie möglich enthalten sein. Das Attribut `dateOfBirth` darf ausdrücklich `null` sein für den Fall, dass der Kunde sein Geburtsdatum nicht bekannt geben will (und der Onlineshop nicht davon abhängig ist). Das Attribut `orderCount` ist dazu gedacht, die Anzahl der Bestellungen des Kunden zu zählen, `averageOrderValue` sollte den durchschnittlichen Betrag aller Bestellungen angeben und der Boolean `premiumCustomer` ist genau dann `true`, wenn dieser Kunde aus irgendeinem Grund vom System als „Premium-Kunde“ angesehen wird, wobei die Geschäftslogik dahinter hier nicht zu interessieren

```
import java.util.*;
import java.time.*;

public final class Customer implements Comparable<Customer> {
    private String firstName;
    private String lastName;
    private LocalDate dateOfBirth;  /* May remain null. */
    private int orderCount;
    private double averageOrderValue;
    private boolean premiumCustomer;

    public Customer(String firstName, String lastName, LocalDate dateOfBirth,
                    int orderCount, double averageOrderValue, boolean premiumCustomer) {
        this.firstName = Objects.requireNonNull(firstName, "Parameter \"firstName\" is null.");
        this.lastName = Objects.requireNonNull(lastName, "Parameter \"lastName\" is null.");
        this.dateOfBirth = dateOfBirth;
        this.orderCount = orderCount;
        this.averageOrderValue = averageOrderValue;
        this.premiumCustomer = premiumCustomer;
    }
```

Listing 1

```
    public static void main(String[] args) {
        /* Create and initialize customer array. */
        Customer[] customerArray = {
                new Customer(>"Richard", >"Leblanc", LocalDate.of(1975, Month.OCTOBER, 6),
                             2, 498.30, false),
                new Customer(>"William", >"Robinson", null,
                             2, 737.35, true),
                new Customer(>"Adella", >"Wheeler", null,
                             7, 824.33, true),
                new Customer(>"Ronald", >"Hogan", null,
                             2, 297.99, false),
                new Customer(>"Joseph", >"Davis", LocalDate.of(1981, Month.NOVEMBER, 3),
                             9, 783.16, true),
                new Customer(>"Charles", >"Quintana", LocalDate.of(1989, Month.JANUARY, 2),
                             4, 748.35, true),
                new Customer(>"Graham", >"Reamer", LocalDate.of(1967, Month.MARCH, 16),
                             3, 255.96, false),
                new Customer(>"Suzanne", >"Murray", LocalDate.of(1975, Month.OCTOBER, 6),
                             2, 199.49, false),
                new Customer(>"Ronald", >"Hogan", LocalDate.of(1997, Month.NOVEMBER, 26),
                             2, 368.10, false),
                new Customer(>"Dolores", >"Falco", null,
                             3, 556.41, true)
        };
        Arrays.sort(customerArray);
        /* Print customer array. */
        for (Customer customerAct : customerArray) {
            System.out.println(customerAct);
        }
    }
    @Override
    public String toString() {
        return String.format(>"Name: %-7s %-8s | DoB: %10s | Orders: %d | AOV: %3.2f | Premium? %5b",
                             firstName, lastName, dateOfBirth,
                             Integer.valueOf(orderCount),
                             Double.valueOf(averageOrderValue),
                             Boolean.valueOf(premiumCustomer));
    }
```

Listing 2

hat. Im echten Leben würde man diese drei Eigenschaften ziemlich sicher nicht als direkte Klassenattribute darstellen, sondern vielmehr durch eine Geschäftslogik jedes Mal, wenn sie benötigt werden, bestimmen lassen. Das ist jedoch eine andere Geschichte und soll nicht vom Grundgedanken der Beispiele in diesem Artikel ablenken.

Der Konstruktor ist selbsterklärend. Es ist nur wichtig zu betonen, dass `firstName` und `lastName` auf `null` überprüft werden, weil sie niemals `null` sein dürfen, wohingegen `dateOfBirth` `null` sein darf.

Der nächste Code-Ausschnitt *(siehe Listing 2)* beinhaltet die main-Funktion, die ein Array mit zehn fingierten Kunden initialisiert, diese dann sortiert und abschließend formatiert auf der Konsole ausgibt. Zu Demonstrationszwecken haben zwei Personen den gleichen Namen (Ronald Hogan) und zwei andere Personen tei-

len sich das gleiche Geburtsdatum (Richard Leblanc und Suzanne Murray am 6. Oktober 1975).

Die `Customer`-Klasse ist `Comparable` und implementiert darum die `compareTo`-Methode. Seit Java 8 basiert die moderne Implementierung jedoch mit Vorteil auf `Comparator`, weswegen man seine `compareTo`-Methode einfach auf einen solchen `Comparator` verweisen lässt. Vor diesem Hintergrund wurden in diesem Artikel sieben verschiedene `Comparators` für die Beispiele implementiert, die allesamt folgend erläutert werden. Um nicht unnötige Compiler-Warnungen, die ungenutzten Code betreffen, zu generieren, gibt es noch einen „Comparator Switch", mit dem, wie in der fett gedruckten Zeile gezeigt *(siehe Listing 3)*, die jeweilige Sortierfunktionalität ausgewählt werden kann. Die `compareTo`-Methode schaut dann einfach auf den Switch und leitet zum ausgewählten `Comparator` weiter.

Sortieren nach Nachnamen

Der wahrscheinlich häufigste Fall besteht darin, die Kunden alphabetisch anhand ihres Nachnamens („last name") zu sortieren. Wenn sich zwei oder mehr Kunden den gleichen „last name" teilen, dann werden ihre „first names" als zweitwichtigstes Attribut behandelt. Wenn sowohl „last" als auch „first name" gleich sind, dann wird das nächstsignifikante Attribut „date of birth" herangezogen. Um es kurz zu halten, werden anschließend keine weiteren Attribute mehr untersucht *(siehe Listing 4)*. Die Ausgabe des Programms ist in *Ausgabe 1* zu sehen.

```
private static enum ComparatorSwitch {
        LAST_NAME, FIRST_NAME, DATE_OF_BIRTH,
        ORDER_COUNT, ORDER_COUNT_REVERSED, PREMIUM_CUSTOMER,
        LAST_NAME_LENGTH
    }
    private static final ComparatorSwitch COMPARATOR_SWITCH
            = ComparatorSwitch.LAST_NAME;
    @Override
    public int compareTo(Customer otherCustomer) {
        switch (COMPARATOR_SWITCH) {
        case LAST_NAME:
            return LAST_NAME_COMPARATOR.compare(this, otherCustomer);
        case FIRST_NAME:
            return FIRST_NAME_COMPARATOR.compare(this, otherCustomer);
        case DATE_OF_BIRTH:
            return DATE_OF_BIRTH_COMPARATOR.compare(this, otherCustomer);
        case ORDER_COUNT:
            return ORDER_COUNT_COMPARATOR.compare(this, otherCustomer);
        case ORDER_COUNT_REVERSED:
            return ORDER_COUNT_REVERSED_COMPARATOR.compare(this, otherCustomer);
        case PREMIUM_CUSTOMER:
            return PREMIUM_CUSTOMER_COMPARATOR.compare(this, otherCustomer);
        case LAST_NAME_LENGTH:
            return LAST_NAME_LENGTH_COMPARATOR.compare(this, otherCustomer);
        default:
            assert false;
            return 0;
        }
    }
}
```

Listing 3

```
private static final Comparator<Customer> LAST_NAME_COMPARATOR
            = Comparator.comparing((Customer c) -> c.lastName)
            .thenComparing(c -> c.firstName)
            .thenComparing(c -> c.dateOfBirth, Comparator.nullsLast(LocalDate::compareTo));
```

Listing 4

```
Name: Joseph  Davis    | DoB: 1981-11-03 | Orders: 9 | AOV: 783.16 | Premium? true
Name: Dolores Falco    | DoB:       null | Orders: 3 | AOV: 556.41 | Premium? true
Name: Ronald  Hogan    | DoB: 1997-11-26 | Orders: 2 | AOV: 368.10 | Premium? false
Name: Ronald  Hogan    | DoB:       null | Orders: 2 | AOV: 297.99 | Premium? false
Name: Richard Leblanc  | DoB: 1975-10-06 | Orders: 2 | AOV: 498.30 | Premium? false
Name: Suzanne Murray   | DoB: 1975-10-06 | Orders: 2 | AOV: 199.49 | Premium? false
Name: Charles Quintana | DoB: 1989-01-02 | Orders: 4 | AOV: 748.35 | Premium? true
Name: Graham  Reamer   | DoB: 1967-03-16 | Orders: 3 | AOV: 255.96 | Premium? false
Name: William Robinson | DoB:       null | Orders: 2 | AOV: 737.35 | Premium? true
Name: Adella  Wheeler  | DoB:       null | Orders: 7 | AOV: 824.33 | Premium? true
```

Ausgabe 1

Die beiden „Ronald Hogans" besitzen den gleichen Namen. In diesem Fall ist „date of birth" das Zünglein an der Waage. Allerdings ist eines dieser „date of birth" gleich `null` – was aber überhaupt kein Problem darstellt. `Comparator.nullsLast(LocalDate::compareTo)` ermöglicht einen einfachen Umgang mit `null`-Werten. Es behandelt `null`-Werte so, dass sie in der Reihenfolge zuletzt kommen (im Gegensatz zu `nullsFirst`, in dem sie in der Reihenfolge zuerst kommen).

Sortieren nach Vornamen

Durch einfaches Vertauschen von „first name" und „last name" wird ein `Comparator` erzeugt, der nach Vornamen („first name") sortiert, dann nach „last name" und letztlich nach „date of birth" *(siehe Listing 5)*. Die Beispielausgabe ist in *Ausgabe 2* zu sehen.

Sortieren nach Geburtsdatum

Kunden nach ihrem Geburtsdatum („date of birth") zu sortieren, ist ebenfalls keine Zauberei *(siehe Listing 6 und Ausgabe 3)*.

Es ist schön zu erkennen, wie die `null`-Geburtstage in der Reihenfolge zuletzt kommen. Wenn zwei Personen den gleichen Geburtstag teilen, so wie Richard Leblanc und Suzanne Murray, dann werden sie nach „last name" (und anschließend nach „first name") sortiert.

Sortieren nach Anzahl der Bestellungen

Es wird empfohlen, die Primitiv-Typ-Versionen `[then]comparingXXX` zu verwenden, wenn Zahlen verglichen werden *(siehe Listing 7)*. In diesem Beispiel werden die Kunden anhand ihrer Anzahl von Bestellungen („order count") – also einem Integer – sortiert *(siehe Ausgabe 4)*.

Das mag wahrscheinlich nicht das beste Ergebnis sein, für das sich der Onlineshop interessiert. Es würde mehr Sinn ergeben, die Kunden mit der größten „number of orders" zuerst anzuzeigen. Diese Kunden sollten anschließend nach ihrer „average order value" sortiert werden, erneut in absteigender Reihenfolge.

Umgekehrtes Sortieren nach Anzahl der Bestellungen

Das `Comparator`-Interface stellt eine Default-Methode `reversed()` zur Verfügung, die in die `[then]comparing[XXX]`-Aufrufkette eingebunden werden kann. Allerdings ist es dabei wichtig, nicht in die Falle zu tappen und anzunehmen, dass sich diese Methode nur auf den letzten `[then]comparing[XXX]`-Aufruf bezieht. Nein, `reversed()` dreht den gesamten `Comparator` um, inklusive aller `[then]comparing[XXX]`-Methodenaufrufe davor.

Wenn man im hier vorliegenden Fall `.reversed()` ganz am Ende der Methodenkette schreiben würde, wäre die Ausgabe zufällig richtig. Die letzte `reversed`-Methode dreht sowohl den „order count" als auch den „average order value" um, was eigentlich genau dem gewünschten Verhalten entspräche. Das Gleiche gilt auch, wenn man `.reversed()` nur nach der „order count"-`comparingInt`-Methode (das heißt, nach der mittleren Zeile) schreiben würde. Auch in diesem Fall ist das Ergebnis – wenn auch nur zufällig – richtig.

Selbst wenn die Ausgabe richtig ist, ist das Schreiben derartigen Codes trügerisch. Es verlässt sich auf einen Denkfehler, dessen Implementierung etwas – zufällig – richtig macht. Salopp ausgedrückt,

```
private static final Comparator<Customer> FIRST_NAME_COMPARATOR
            = Comparator.comparing((Customer c) -> c.firstName)
            .thenComparing(c -> c.lastName)
            .thenComparing(c -> c.dateOfBirth, Comparator.nullsLast(LocalDate::compareTo));
```

Listing 5

```
Name: Adella  Wheeler  | DoB:       null | Orders: 7 | AOV: 824.33 | Premium? true
Name: Charles Quintana | DoB: 1989-01-02 | Orders: 4 | AOV: 748.35 | Premium? true
Name: Dolores Falco    | DoB:       null | Orders: 3 | AOV: 556.41 | Premium? true
Name: Graham  Reamer   | DoB: 1967-03-16 | Orders: 3 | AOV: 255.96 | Premium? false
Name: Joseph  Davis    | DoB: 1981-11-03 | Orders: 9 | AOV: 783.16 | Premium? true
Name: Richard Leblanc  | DoB: 1975-10-06 | Orders: 2 | AOV: 498.30 | Premium? false
Name: Ronald  Hogan    | DoB: 1997-11-26 | Orders: 2 | AOV: 368.10 | Premium? false
Name: Ronald  Hogan    | DoB:       null | Orders: 2 | AOV: 297.99 | Premium? false
Name: Suzanne Murray   | DoB: 1975-10-06 | Orders: 2 | AOV: 199.49 | Premium? false
Name: William Robinson | DoB:       null | Orders: 2 | AOV: 737.35 | Premium? true
```

Ausgabe 2

```
private static final Comparator<Customer> DATE_OF_BIRTH_COMPARATOR
            = Comparator.comparing((Customer c) -> c.dateOfBirth,
                                   Comparator.nullsLast(LocalDate::compareTo))
            .thenComparing(c -> c.lastName)
            .thenComparing(c -> c.firstName);
```

Listing 6

```
Name: Graham  Reamer   | DoB: 1967-03-16 | Orders: 3 | AOV: 255.96 | Premium? false
Name: Richard Leblanc  | DoB: 1975-10-06 | Orders: 2 | AOV: 498.30 | Premium? false
Name: Suzanne Murray   | DoB: 1975-10-06 | Orders: 2 | AOV: 199.49 | Premium? false
Name: Joseph  Davis    | DoB: 1981-11-03 | Orders: 9 | AOV: 783.16 | Premium? true
Name: Charles Quintana | DoB: 1989-01-02 | Orders: 4 | AOV: 748.35 | Premium? true
Name: Ronald  Hogan    | DoB: 1997-11-26 | Orders: 2 | AOV: 368.10 | Premium? false
Name: Dolores Falco    | DoB:       null | Orders: 3 | AOV: 556.41 | Premium? true
Name: Ronald  Hogan    | DoB:       null | Orders: 2 | AOV: 297.99 | Premium? false
Name: William Robinson | DoB:       null | Orders: 2 | AOV: 737.35 | Premium? true
Name: Adella  Wheeler  | DoB:       null | Orders: 7 | AOV: 824.33 | Premium? true
```

Ausgabe 3

ist es nichts anderes, als darauf zu hoffen, dass „falsch und falsch" im Resultat wieder „richtig" ergibt. Ein korrekter Weg, einen einzelnen spezifischen „Zwischen-Comparator" umzudrehen, wird in *Listing 8* aufgezeigt.

Er sieht nicht ganz so elegant wie die vorherigen Code-Ausschnitte aus. Die obige Implementierung verwendet eine überladene Version `thenComparing(Function, Comparator)`, die nebst der eigentlichen Key-Extractor-`Function` noch einen zusätzlichen `Comparator` entgegennimmt. Dieser zusätzlich zur Verfügung gestellte `Comparator` ist `Comparator.reverseOrder()` (nicht zu verwechseln mit `Comparator.reversed()`), der ganz einfach die natürliche Ordnung des betreffenden Datentyps umdreht.

Leider existieren diese überladenen Methoden nicht für die Primitiv-Typ-Versionen `thenComparingInt`, `thenComparingLong` und `thenComparingDouble`, sodass man hier auf die allgemeinen Methoden zurückgreifen muss, die jeweils auch ein (Auto-)Boxing und Unboxing für primitive Typen durchführen. Es ist eine Frage der persönlichen Vorliebe, ob man seine Entwicklungsumgebung (IDE) so einrichtet, dass man bei jedem automatischen Boxing und Unboxing eine Warnung erhält. Wird man von der IDE gezwungen, alle Konvertierungen manuell vorzunehmen, so kann dies sehr hilfreich sein, um Stolperfallen oder Performance-Lecks zu erkennen, die andernfalls unerkannt blieben.

Ein weiterer Nachteil ist, dass man dem Compiler vermehrt „nachhelfen" muss, indem man ihm den spezifischen Typ (hier: `Customer`) innerhalb des Lambda-Ausdrucks angibt. Die Beispielausgabe ist in *Ausgabe 5* zu sehen.

Sortieren nach Premium-Kunden

Listing 9 zeigt die Sortierung der Kunden nach Premium-Kunden („premium customer", `true` soll vor `false` kommen), dann nach „order count" (in absteigender Reihenfolge) und letztlich nach „average order value" (erneut in absteigender Reihenfolge). Alle drei Sortierschritte sind explizit umgedreht. Sie könnten zwar durch ein einzelnes `.reversed()` ganz am Ende der `[then]comparing`-Kette ersetzt werden, doch sollte man die Warnung aus dem letzten Abschnitt im Hinterkopf behalten. Wer sich dennoch dafür entschei-

```
private static final Comparator<Customer> ORDER_COUNT_COMPARATOR
            = Comparator.comparingInt((Customer c) -> c.orderCount)
            .thenComparingDouble(c -> c.averageOrderValue);
```

Listing 7

```
Name: Suzanne Murray   | DoB: 1975-10-06 | Orders: 2 | AOV: 199.49 | Premium? false
Name: Ronald  Hogan    | DoB:       null | Orders: 2 | AOV: 297.99 | Premium? false
Name: Ronald  Hogan    | DoB: 1997-11-26 | Orders: 2 | AOV: 368.10 | Premium? false
Name: Richard Leblanc  | DoB: 1975-10-06 | Orders: 2 | AOV: 498.30 | Premium? false
Name: William Robinson | DoB:       null | Orders: 2 | AOV: 737.35 | Premium? true
Name: Graham  Reamer   | DoB: 1967-03-16 | Orders: 3 | AOV: 255.96 | Premium? false
Name: Dolores Falco    | DoB:       null | Orders: 3 | AOV: 556.41 | Premium? true
Name: Charles Quintana | DoB: 1989-01-02 | Orders: 4 | AOV: 748.35 | Premium? true
Name: Adella  Wheeler  | DoB:       null | Orders: 7 | AOV: 824.33 | Premium? true
Name: Joseph  Davis    | DoB: 1981-11-03 | Orders: 9 | AOV: 783.16 | Premium? true
```

Ausgabe 4

```
private static final Comparator<Customer> ORDER_COUNT_REVERSED_COMPARATOR
            = Comparator.comparing((Customer c) -> Integer.valueOf(c.orderCount),
                                   Comparator.reverseOrder())
            .thenComparing((Customer c) -> Double.valueOf(c.averageOrderValue),
                           Comparator.reverseOrder());
```

Listing 8

```
Name: Joseph  Davis    | DoB: 1981-11-03 | Orders: 9 | AOV: 783.16 | Premium? true
Name: Adella  Wheeler  | DoB:       null | Orders: 7 | AOV: 824.33 | Premium? true
Name: Charles Quintana | DoB: 1989-01-02 | Orders: 4 | AOV: 748.35 | Premium? true
Name: Dolores Falco    | DoB:       null | Orders: 3 | AOV: 556.41 | Premium? true
Name: Graham  Reamer   | DoB: 1967-03-16 | Orders: 3 | AOV: 255.96 | Premium? false
Name: William Robinson | DoB:       null | Orders: 2 | AOV: 737.35 | Premium? true
Name: Richard Leblanc  | DoB: 1975-10-06 | Orders: 2 | AOV: 498.30 | Premium? false
Name: Ronald  Hogan    | DoB: 1997-11-26 | Orders: 2 | AOV: 368.10 | Premium? false
Name: Ronald  Hogan    | DoB:       null | Orders: 2 | AOV: 297.99 | Premium? false
Name: Suzanne Murray   | DoB: 1975-10-06 | Orders: 2 | AOV: 199.49 | Premium? false
```

Ausgabe 5

```
private static final Comparator<Customer> PREMIUM_CUSTOMER_COMPARATOR
            = Comparator.comparing((Customer c) -> Boolean.valueOf(c.premiumCustomer),
                                   Comparator.reverseOrder())
            .thenComparing((Customer c) -> Integer.valueOf(c.orderCount),
                           Comparator.reverseOrder())
            .thenComparing((Customer c) -> Double.valueOf(c.averageOrderValue),
                           Comparator.reverseOrder());
```

Listing 9

```
Name: Joseph  Davis    | DoB: 1981-11-03 | Orders: 9 | AOV: 783.16 | Premium? true
Name: Adella  Wheeler  | DoB:       null | Orders: 7 | AOV: 824.33 | Premium? true
Name: Charles Quintana | DoB: 1989-01-02 | Orders: 4 | AOV: 748.35 | Premium? true
Name: Dolores Falco    | DoB:       null | Orders: 3 | AOV: 556.41 | Premium? true
Name: William Robinson | DoB:       null | Orders: 2 | AOV: 737.35 | Premium? true
Name: Graham  Reamer   | DoB: 1967-03-16 | Orders: 3 | AOV: 255.96 | Premium? false
Name: Richard Leblanc  | DoB: 1975-10-06 | Orders: 2 | AOV: 498.30 | Premium? false
Name: Ronald  Hogan    | DoB: 1997-11-26 | Orders: 2 | AOV: 368.10 | Premium? false
Name: Ronald  Hogan    | DoB:       null | Orders: 2 | AOV: 297.99 | Premium? false
Name: Suzanne Murray   | DoB: 1975-10-06 | Orders: 2 | AOV: 199.49 | Premium? false
```

Ausgabe 6

```
private static final Comparator<Customer> LAST_NAME_LENGTH_COMPARATOR
            = Comparator.comparingInt((Customer c) -> c.lastName.length())
            .thenComparing(c -> c.lastName)
            .thenComparing(c -> c.firstName);
```

Listing 10

```
Name: Joseph  Davis    | DoB: 1981-11-03 | Orders: 9 | AOV: 783.16 | Premium?  true
Name: Dolores Falco    | DoB:       null | Orders: 3 | AOV: 556.41 | Premium?  true
Name: Ronald  Hogan    | DoB:       null | Orders: 2 | AOV: 297.99 | Premium? false
Name: Ronald  Hogan    | DoB: 1997-11-26 | Orders: 2 | AOV: 368.10 | Premium? false
Name: Suzanne Murray   | DoB: 1975-10-06 | Orders: 2 | AOV: 199.49 | Premium? false
Name: Graham  Reamer   | DoB: 1967-03-16 | Orders: 3 | AOV: 255.96 | Premium? false
Name: Richard Leblanc  | DoB: 1975-10-06 | Orders: 2 | AOV: 498.30 | Premium? false
Name: Adella  Wheeler  | DoB:       null | Orders: 7 | AOV: 824.33 | Premium?  true
Name: Charles Quintana | DoB: 1989-01-02 | Orders: 4 | AOV: 748.35 | Premium?  true
Name: William Robinson | DoB:       null | Orders: 2 | AOV: 737.35 | Premium?  true
```

Ausgabe 7

det, sollte zumindest einen erklärenden Kommentar hinzufügen. Die Programmausgabe sehen Sie in *Ausgabe 6*.

Es ist wichtig zu beachten, dass die standardmäßige (natürliche) Ordnung für Booleans zuerst `false` und dann `true` ist. Da hier die Reihenfolge umgedreht wurde, erscheint korrekt zuerst `true` vor `false`.

Sortieren nach der Länge des Nachnamens

„Nun, das ist ja alles schön und gut, was mir da bis jetzt gezeigt wurde, aber was ist, wenn ich nicht nur einfach Attribute vergleichen möchte, sondern kompliziertere Dinge?" – Kein Problem! Im abschließenden Beispiel werden Kunden anhand der Länge ihres Nachnamens sortiert *(siehe Listing 10)*. Das Ergebnis ist in *Ausgabe 7* zu sehen.

Man muss einfach eine Key-Extractor-`Function` zur Verfügung stellen, die das macht, was man wünscht. Man sieht, dass sich an der Art, wie `Comparators` geschrieben sind, ansonsten wenig ändert.

Zusammenfassung und Ausblick

Im zweiten Teil dieses Artikels wurden viele Beispiele gezeigt, wie die Vergleichsfunktionalität in Java implementiert werden kann. Auch wenn das Umdrehen einzelner Sortierschritte (immer noch) etwas umständlich ist, sind die Konsistenz und der Stil solcher Vergleichsmethoden ansonsten beeindruckend.

In naher Zukunft gilt es, nach einer besseren Lösung Ausschau zu halten, was die umgedrehten Sortierschritte anbelangt. Denkbar wäre eine Hilfsklasse, die das Umdrehen einzelner Sortierschritte vereinfacht und darüber hinaus solche Methoden auch für die primitiven Datentypen anbietet. Die Frage, ob man nennenswerte Performance-Einbußen bei der Verwendung solcher Methodenketten in Kauf nehmen muss, wurde zwar nicht direkt, aber in einem vergleichbaren Fall im Rahmen von `equals`- und `hashCode`-Methoden in einem zweiteiligen Blogeintrag des Autors beantwortet *(https://link.simplexacode.ch/zxxx und https://link.simplexacode.ch/5gih)*. Kurz: Die Performance-Einbußen bewegen sich innerhalb eines unbedeutenden Rahmens und rechtfertigen es nicht, von der modernen und konsistenten Implementierung natürlicher Ordnung bei Java-Objekten abzukommen – sofern man nicht aufzeigen kann, dass die Zeit, die mit dem Vergleichen von Java-Objekten verbracht wird (namentlich beim Sortieren) einen unstreitig signifikanten Großteil der gesamten Laufzeit einer Applikation ausmacht. Das wiederum dürfte für die meisten realen Applikationen ganz klar nicht der Fall sein.

Hinweis: Die Quelltexte zu diesem Artikel findet der Leser unter *https://link.simplexacode.ch/txx9*. Die englische Version des Artikels steht unter *https://link.simplexacode.ch/gurm* zur Verfügung.

Christian Heitzmann
christian.heitzmann@simplexacode.ch

Christian Heitzmann ist Gründer und Geschäftsführer der SimplexaCode AG in Luzern, die sich auf Software-Entwicklung, -Schulung und -Beratung vor allem für MINT-Anwendungen und technische Implementierungsthemen in Java spezialisiert hat. Er ist seit fünfzehn Jahren mit Java vertraut und hat während vieler Jahre Algorithmen und Mathematik unterrichtet.